Environnement de développement sous Linux
Module 2l012-2017fev

Dominique.Bereziat@lip6.fr,
Valérie Ménissier-Morain

Février 2017

Sixième partie VI

L'utilitaire `make`

Plan

Plan

La commande `make`

- Tout comme `bash` : permet d'automatiser une suite de tâches
- À la différence de `bash` :
  - `make` est orienté production d'un fichier à partir d'une ou plusieurs sources
  - les étapes de production ne sont exécutées que si nécessaire
- Origine de `make` :
  - 1977, par Stuart Feldman, pour automatiser la compilation C
  - le langage C introduit la notion de compilation séparée, très importante et incontournable, mais nécessite une suite complexe d'appels au compilateur.
- `make` aujourd'hui :
  - la déclinaison BSD
  - la déclinaison GNU (comprend le dialecte BSD)
  - dans la norme POSIX (la déclinaison BSD pas les extensions GNU)
  - Sur Linux, la commande `make` est celle de GNU, nous parlerons que du GNU make dans ce cours

Les grands principes

- La commande `make` cherche dans le répertoire courant un fichier `Makefile` (ou `makefile`, ou `GNUMakefile`) pour y lire ses instructions
- La commande `make` est donc rarement invoquée avec un fichier en paramètre
- Le fichier `Makefile` décrit :
  1. les dépendances : quels fichiers (à construire, cible) dépendent de quels autres fichiers (sources)
  2. associés aux dépendances, la recette : quelles commandes seront appliquées pour construire la cible à partir de ses sources
  3. des variables permettant de paramétrer plus facilement les étapes de la production

Les alternatives (crédibles)

- `autotools` : le système de compilation multiplateforme de GNU. Un script `sh` (il s'appelle `configure`) analyse l'environnement (autoconf) et fabrique les `Makefile` spécifiques à l'hôte à partir de fichiers `Makefile.am` (automake).
  - pour l'utilisateur : facile à utiliser, et ne nécessite que `sh`
  - pour le développeur : les `Makefile.am` sont faciles à écrire, la partie `autoconf` beaucoup moins !!
- `cmake` : autre système de compilation multiplateforme. La commande `cmake` doit être installée et lancée pour créer les `Makefile` *ad hoc*.
  - pour l'utilisateur : nécessite que `cmake` soit installé, moins de liberté que `configure`, mais reste facile à utiliser
  - pour le développeur : les équivalents du `Makefile.am` pour `cmake` sont faciles à écrire. Le système est néanmoins moins puissant que `autotools`
- Les environnements de développement (IDL) produisent généralement des `Makefile`, `CMakeLists.txt` et fichiers autotools et leur propres projets
- Même si on cache les fichiers `Makefile`, l'informaticien chevronné ne peut les ignorer

Usages courants de `make` (vue dans ce cours et en examen !)

- Compilation (séparée ou non) :
  - fabrication d'objets (compilation)
  - fabrication de binaires exécutables (édition des liens)
  - fabrication de bibliothèques
- Production d'une documentation
- Installation d'un logiciel
- Test d'un logiciel

Plan

Premier `Makefile`

- Soit le code `mycos.c` suivant :

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main(int argc, char **argv) {
    if( argc>1)
      printf("%g\n", cos(atof(argv[1])));
    return argc==1;
}
```

- Il se compile et s'utilise ainsi :

```
$ gcc -o mycos mycos.c -lm -O2
$ ./mycos 1
0.540302
```

Premier `Makefile` (suite)

- Écrivons le `Makefile` correspondant :

```
1 mycos: mycos.c
2 ␣␣␣␣␣␣␣␣gcc -o mycos mycos.c -lm -O2
3 test: mycos
4 ␣␣␣␣␣␣␣␣test `./mycos 1` = 0.540302
```

- Compilation :

```
% make mycos
gcc -o mycos mycos -lm -O2
% make mycos
make: `mycos' is up to date.
```

- Test :

```
% make test
test `./mycos 1` = 0.540302
```

Premier `Makefile` (suite)

- Donc :
  - `make` n'exécute les commandes que si nécessaire
  - `make` imprime les commandes qu'elle va exécuter
- Dans `Makefile`, ajoutons la cible `fauxtest`

```
1 fauxtest: mycos
2 ␣␣␣␣␣␣␣␣test `./mycos 1` = 0.540303
3 ␣␣␣␣␣␣␣␣echo OK
```

- Essayons :

```
$ make fauxtest
test `./mycos 1` = 0.540303
make: *** [fauxtest] Error 1
```

- Donc : la commande `make` s'arrête si une commande retourne une valeur non nulle

Structure d'un fichier `Makefile`

- Une ou plusieurs règles de production comprenant une cible et une ou plusieurs sources (dépendances) suivit d'une recette (une ou plusieurs commandes)

```
1 CIBLE1 : SOURCE1 SOURCE2 ...
2 ␣␣␣␣␣␣␣␣commande1
3 ␣␣␣␣␣␣␣␣commande2 \
4 suite de la commande2
5 CIBLE2 : SOURCE3 SOURCE4 ...
6 ␣␣␣␣␣␣␣␣command3
7 ␣␣␣␣␣␣␣␣...
```

- Le caractère TABULATION doit précéder la commande sinon erreur de syntaxe (`emacs` détecte les tabulations suspectes + mode `whitespace`).
- Une suite d'espaces N'EST PAS une tabulation
- Chaque ligne des recettes est exécutée par un appel système, fonction `system()`, donc un sous-shell !

Structure d'un fichier `Makefile` (suite)

- Les cibles et sources sont des fichiers (sauf exception, voir plus loin)
- Une règle peut être réduite à ses dépendances
- Une cible peut ne pas avoir de dépendances
- La première règle est la cible par défaut
- L'ordre des règles n'est pas important
- L'appel de la commande `make` dans le `shell` :

```
# Cible par defaut
% make
# Une cible particuliere
% make CIBLE1
# ou plusieurs cibles (traites dans l'ordre)
% make CIBLE1 CIBLE2
```

Déclenchement des recettes (1)

- La recette d'une règle n'est exécutée que si la cible n'est pas à jour :
  1. le fichier correspondant à la cible n'existe pas ou est plus ancien que les sources
  2. l'une des sources n'est pas à jour
- Illustration :

```
$ cat >Makefile
double: simple
        cat simple simple >double
$ echo 'Repete un peu pour voir?' > simple
$ make
cat simple simple >double
$ cat double
Repete un peu pour voir?
Repete un peu pour voir?
$ make double
make: `double' is up to date.
$ touch simple
$ make double
cat simple simple >double
```

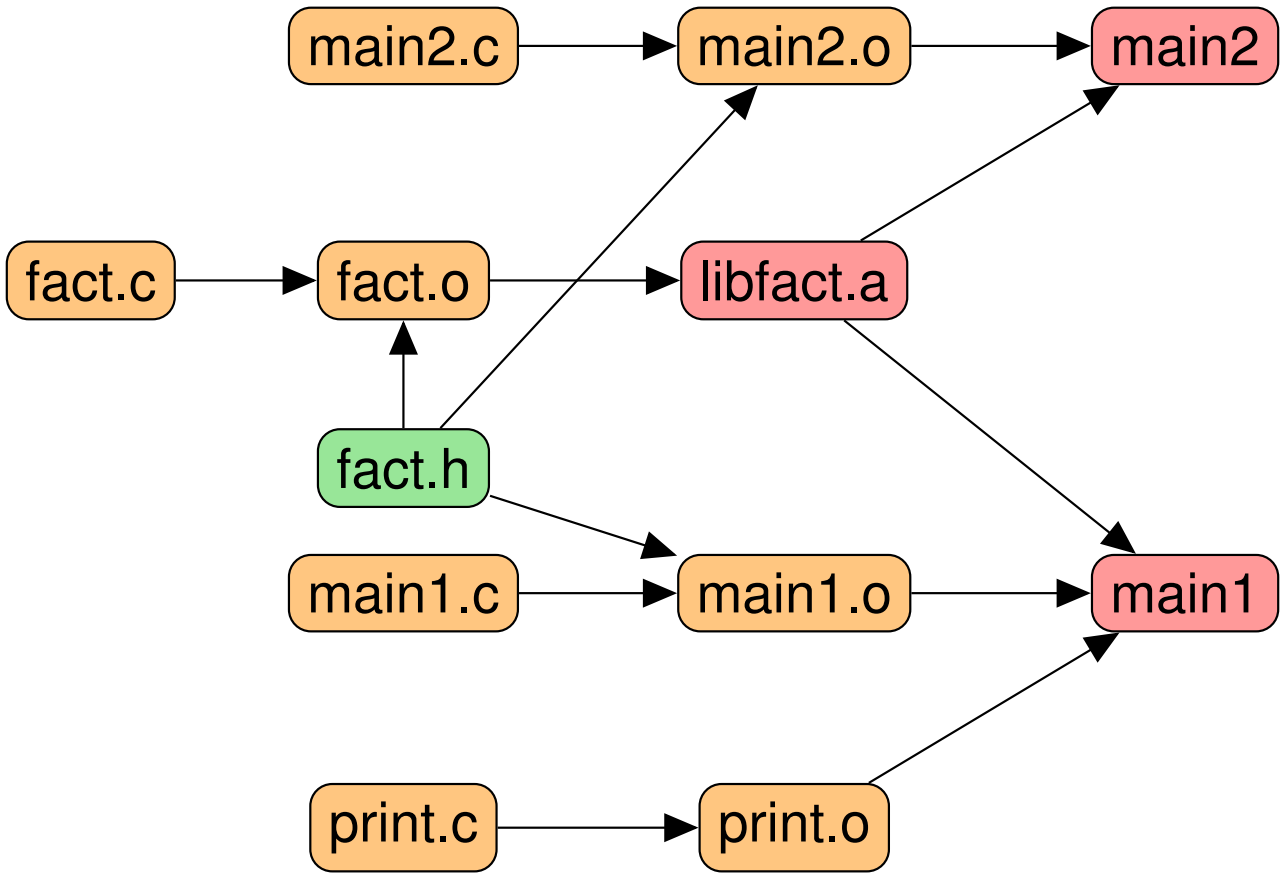
Déclenchement des recettes (2)

- Le `Makefile` est lu en intégralité pour construire un graphe des dépendances
- La présence de cycles dans le graphe est détectée et la dernière règle provoquant le cycle est ignorée :

```
$ echo 'toto:tata
tata:toto' > Makefile
$ make
make: Circular tata <- toto dependency dropped.
make: Nothing to be done for `toto'.
```

Plan

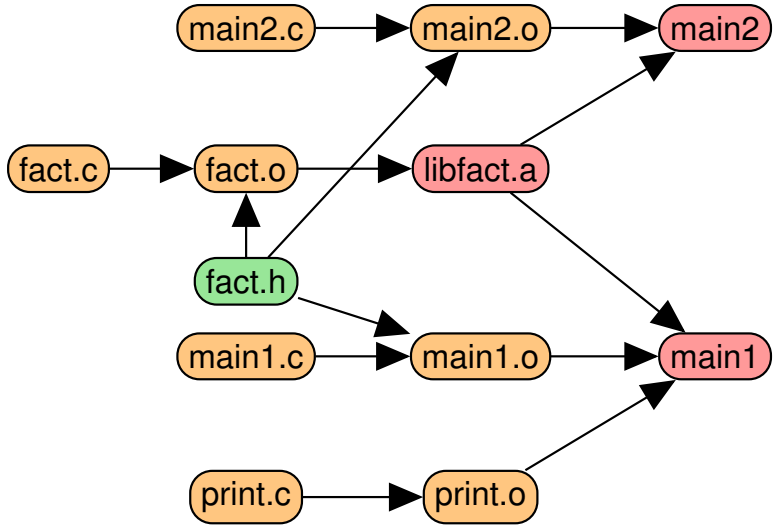
Un exemple plus complexe : compilation séparée



Un exemple plus complexe (2)

Description des dépendances :

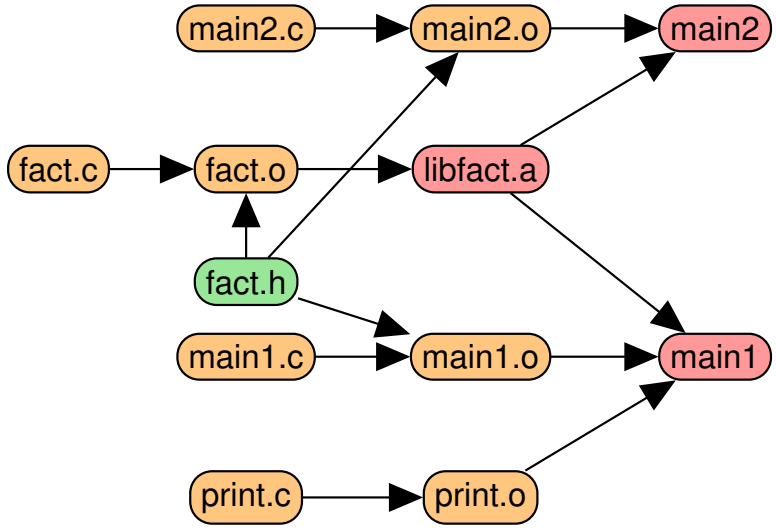
```
fact.o: fact.c fact.h
print.o: print.c
main1.o: main1.c fact.h
main2.o: main2.c fact.h
libfact.a: fact.o
main1: main1.o print.o libfact.a
main2: main2.o libfact.a
```



Un exemple plus complexe (3)

On peut factoriser aussi les sources :

```
fact.o: fact.c
print.o: print.c
main1.o: main1.c
main2.o: main2.c
libfact.a: fact.o
main1: main1.o print.o libfact.a
main2: main2.o libfact.a
fact.o main1.o main2.o: fact.h
```



Un exemple plus complexe (4)

`Makefile` avec les règles complètes :

```
1  main1: main1.o print.o libfact.a
2  ␣␣␣␣␣␣␣␣gcc -o main1 main1.o print.o libfact.a
3  main2: main2.o libfact.a
4  ␣␣␣␣␣␣␣␣gcc -o main2 main2.o libfact.a
5  libfact.a: fact.o
6  ␣␣␣␣␣␣␣␣ar cr libfact.o fact.o
7  fact.o: fact.c fact.h
8  ␣␣␣␣␣␣␣␣gcc -Wall -std=c99 -pedantic -c fact.c
9  print.o: print.c fact.h
10 ␣␣␣␣␣␣␣␣gcc -Wall -std=c99 -pedantic -c print.c
11 main1.o: main1.c fact.h
12 ␣␣␣␣␣␣␣␣gcc -Wall -std=c99 -pedantic -c main1.c
13 main2.o: main2.c
14 ␣␣␣␣␣␣␣␣gcc -Wall -std=c99 -pedantic -c main1.c
```

Dépendance et inclusion de fichiers

- On a écrit :

```
1 main1.o: main1.c fact.h
2         gcc -Wall -std=c99 -pedantic -c main1.c
```

probablement parce que `main1.c` contient :

```
1 #include "fact.h"
```

- On aurait pu écrire aussi écrire :

```
1 main1.o: main1.c
2         gcc -Wall -std=c99 -pedantic -c main1.c
3 main1.c: fact.h
```

mais c'est conceptuellement incorrect parceque `main1.c` n'est pas construit à partir de `fact.h`.

Les options de `make` à connaître

- `-f FILE` : utilise `FILE` comme fichier Makefile
- `-C DIR` : se place dans le répertoire `DIR`
- `-n` : affiche les commandes qui doivent être exécutées sans les exécuter
- `-W FILE` : considère que le fichier `FILE` est modifié. Une alternative classique est de modifier effectivement (commande `touch`) le fichier `-W` s'utilise fréquemment avec `-n`
- `-r` : supprime les règles implicites (voir plus loin)
- `-q` : retourne 0 si la cible est à jour, 1 sinon. N'exécute aucune commande
- `-s` : mode silencieux (n'imprime pas les commandes)
- `-p` : affiche toutes les règles et variables par défaut
- autres options : `man make`

Plan

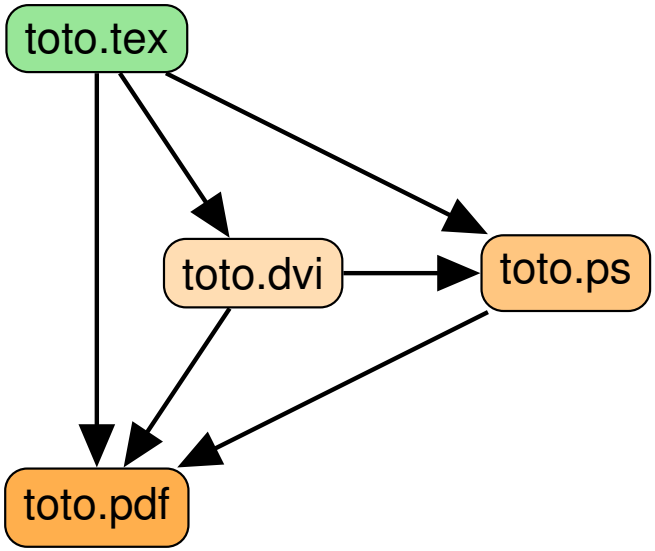
Comportement : cas de règles multiples

- si il existe plusieurs règles pour construire une cible : anomalie ! seule la dernière règle sera appliquée :

```
$ cat Makefile
mycos: mycos.c
	gcc -o mycos mycos.c
mycos: mycos.c
	gcc -o mycos mycos.c -lm
$ make mycos
Makefile:4: warning: overriding commands for target `mycos'
Makefile:2: warning: ignoring old commands for target `mycos'
gcc -o mycos mycos.c -lm
```

- les dépendances sont malgré tout cumulées mais certaines commandes seront écrasées (remplacées par celles de la dernière règle).
- Il est recommandé de corriger son `Makefile` pour éviter tout effet de bord.

Cas des règles multiples, un exemple




```
1  toto.pdf: toto.tex
2  ␣␣␣␣␣␣␣␣pdflatex toto.tex
3  toto.pdf: toto.ps
4  ␣␣␣␣␣␣␣␣ps2pdf toto.ps
5  toto.pdf: toto.dvi
6  ␣␣␣␣␣␣␣␣dvipdfm toto.dvi
7  toto.ps: toto.tex
8  ␣␣␣␣␣␣␣␣pslatex toto.tex
9  toto.ps: toto.dvi
10 ␣␣␣␣␣␣␣␣dvips toto.dvi -o toto.ps
11 toto.dvi: toto.tex
12 ␣␣␣␣␣␣␣␣latex toto.tex
```

Cas des règles multiples, un exemple

```
% make toto.pdf -n
Makefile:4: warning: overriding commands for target `toto.pdf'
Makefile:2: warning: ignoring old commands for target `toto.pdf'
Makefile:6: warning: overriding commands for target `toto.pdf'
Makefile:4: warning: ignoring old commands for target `toto.pdf'
Makefile:10: warning: overriding commands for target `toto.ps'
Makefile:8: warning: ignoring old commands for target `toto.ps'
latex toto.tex
dvips toto.dvi -o toto.ps
dvipdfm toto.dvi
```

- `make` utilise la dernière règle qu'il trouve pour construire `toto.pdf` : celle des lignes 5-6.
- il prend donc également la règle des lignes 11-12 pour construire `toto.dvi`
- accumulation des dépendances : à cause de la ligne 3 on a indiqué que `toto.pdf` dépend de `toto.ps`. Ce dernier est donc produit avec la règle des lignes 9-10 car c'est la dernière !
- écrasement des commandes : la commande de la ligne 4 a été écrasé car `make` a utilisé celle de la ligne 6

Cas des règles multiples, un exemple

```
% make toto.pdf -n
Makefile:4: warning: overriding commands for target `toto.pdf'
Makefile:2: warning: ignoring old commands for target `toto.pdf'
Makefile:6: warning: overriding commands for target `toto.pdf'
Makefile:4: warning: ignoring old commands for target `toto.pdf'
Makefile:10: warning: overriding commands for target `toto.ps'
Makefile:8: warning: ignoring old commands for target `toto.ps'
```

- Les messages de `make` indiquent que le `Makefile` se réduit à :

```
1 toto.pdf: toto.tex
2 toto.pdf: toto.ps
3 toto.pdf: toto.dvi
4 ␣␣␣␣␣␣␣␣dvipdfm toto.dvi
5 toto.ps: toto.tex
6 toto.ps: toto.dvi
7 ␣␣␣␣␣␣␣␣dvips toto.dvi -o toto.ps
8 toto.dvi: toto.ps
9 ␣␣␣␣␣␣␣␣latex toto.tex
```

Plan

Plan

Cibles virtuelles (1)

- Cible virtuelle : lorsque les cibles ne sont pas des fichiers !
- et il peut donc y avoir des dépendances virtuelles
- Cibles virtuelles standards : `all`, `clean`, `install`, `uninstall`, `depend` ...
  - par exemple `clean` est une cible de nettoyage
  - que se passe-t-il si un fichier `clean` existe sur le disque ?

```
% cat >Makefile
true: true.c
        cc true.c -o true
clean:
        rm true
% cat >true.c
int main() { return 0;}
% make true clean
cc true.c -o true
rm true
% touch clean
% make true clean
cc true.c -o true
make: Nothing to be done for `clean'.
```

Cibles virtuelles (2)

- Solution : indiquer leur virtualité à `make` à l'aide de la cible spéciale `.PHONY`

```
$ echo ".PHONY: clean" >> Makefile
$ cat Makefile
true: true.c
        cc true.c -o true
clean:
        rm true
.PHONY: clean
$ make clean
rm true
```

Plan

Les quantifieurs de commandes

- Rappels :
  - chaque commande est exécutée par un sous-shell (`man system`)
  - `make` s'arrête dès une commande retourne une valeur non nulle
  - `make` imprime les commandes qu'il exécute
- On peut modifier localement ces comportements en préfixant la commande par :
  - `@` : `make` n'imprimera pas la commande (globalement, option `-s`)
  - `-` : `make` continuera son exécution même si la commande retourne une erreur
  - on peut bien-sûr les cumuler : `@-`

Les quantifieurs : exemples

```
 $ cat >Makefile
all:
        @echo Je cree toto ...
        touch toto
clean:
        @echo "J'efface␣toto␣..."
        rm toto
        @echo "J'efface␣tata␣..."
        -rm tata
        rm tata
$ make all clean
Je cree toto ...
touch toto
J'efface toto ...
rm toto
J'efface tata ...
rm tata
rm: cannot remove 'tata': No such file or directory
make: [clean] Error 1 (ignored)
rm tata
rm: cannot remove 'tata': No such file or directory
make: *** [clean] Error 1
```

Plan

Variables

- On peut définir des variables dans un `Makefile`.
  - Affectation : `NOM = valeur`
    les espaces avant ou après l'opérateur d'affectation sont facultatifs !
  - Evaluation : `$(NOM)` ou `${NOM}`
  - Piège : `$NOM` veut en faite dire `${N}OM`
  - On peut définir la valeur sur plusieurs lignes avec le caractère `\`
- Exemple avec le Makefile suivant :

```
1 NOM = le monde
2 all:
3     echo Bonjour $(NOM)
```

et l'exécution dans le shell :

```
$ make
echo Bonjour le monde
Bonjour le monde
```

- Les variables peuvent être modifiées au lancement de `make` :

```
$ make NOM=Dom
echo Bonjour Dom
Bonjour Dom
```

Evaluation des variables (1)

- Comme `make` lit entièrement le `Makefile`, en cas d'initialisation multiple, c'est la dernière qui prévaut
- L'évaluation des variables est récursive, elle se fait non pas pendant l'affectation, mais à la fin, dans les règles (dépendances, cible et commandes) :

```
$ cat > Makefile
COPIE_DE_COPIE = $(COPIE)
COPIE = $(VAR)
VAR = 3
VAR = 2
all:
        echo $(COPIE_DE_COPIE)
$ make
echo 2
2
```

- Pour accroître la lisibilité du `Makefile`, on groupe au début les initialisations des variables, dans l'ordre.

Evaluation des variables (2)

- Inconvénient : on peut entrer en récursion infinie !

```
$ cat > Makefile
CFLAGS = -g
CFLAGS = $(CFLAGS) -O2
all:
        echo $(CFLAGS)
$ make
Makefile:2: *** Recursive variable `CFLAGS' references
itself (eventually).  Stop.
```

- Traitement : la "simple expansion" avec l'opérateur `:::=`

```
$ cat >Makefile
CFLAGS = -g
CFLAGS ::= $(CFLAGS) -O2
all:
     echo $(CFLAGS)
$ make
echo -g -O2
-g -O2
```

l'évaluation a lieu une seule fois pendant l'affection et sans récursion

Variables et environnement

- Toute variable d'environnement devient une variable `make` (comme avec `bash`)
- Si une variable locale au `Makefile` a le même nom qu'une variable d'environnement, alors la première prend le pas

```
$ cat >Makefile
USER = Tata
all:
	echo $(USER):$(HOME)
$ USER=Toto make -n
echo Tata:/Users/bereziat
```

- Mais par contre, rappelez-vous que :

```
$ make -n USER=Toto
echo Toto:/Users/bereziat
```

Variables : similitudes et différences avec `bash`

	`bash`	`make`
affectation	NOM=valeur	NOM = valeur
	NOM="valeur1 valeur2"	NOM = valeur1 valeur2
évaluation	$NOM ou ${NOM}	$(NOM) ou ${NOM}
		$NOM signifie ${N}OM

- `make` évalue récursivement les variables dans les règles et jamais dans les affections (sauf à utiliser `::=`) !
- `bash` évalue toujours les variables immédiatement

Plan

`make` et le shell (1)

- Rappel : chaque commande est exécutée par un sous-shell (`man system`) et donc pour nous Linuxiens, par `sh`
- Attention : `sh` n'est pas `bash` ! c'est un sous-ensemble réduit ! Par exemple, dans `sh` :
  - pas de tableaux associatifs
  - pas d'instruction de test `[[` (la commande de test `[` est OK)
  - utiliser `` `...` `` à la place de `$(...)`
  - ...
- La variable `SHELL` indique le shell utilisée par `make`

```
$ cat >Makefile
all:
        echo SHELL=$(SHELL)
        [[ -f Makefile ]] && echo "j'existe"
$ make -s
SHELL=/bin/sh
/bin/sh: 1: [[: not found
t:2: recipe for target 'all' failed
```

`make` et le shell (2)

- Conseil : dans cette UE, positionner la variable `SHELL` au chemin de `bash`

```
$ cat >Makefile
SHELL = /bin/bash
all:
        echo SHELL=$(SHELL)
        [[ -f Makefile ]] && echo "j'existe""
$ make -s
/bin/bash
j'existe
```

- Portabilité : n'écrire que du `sh` (hors de ce cours)

```
$ cat >Makefile
all:
        echo SHELL=$(SHELL)
        [ -f Makefile ] && echo "j'existe""
$ make -s
/bin/sh
j'existe
```

`make` et le shell (3)

- Dernière solution : déporter dans un script qui sera appelé par `make`.

```
$ cat >monscript
#! /bin/bash
[[ -f Makefile ]] && echo "j'existe"
$ chmod +x monscript
$ cat >Makefile
all:
        echo SHELL=$(SHELL)
        ./monscript
$ make -s
/bin/sh
j'existe
```

- Remarquons que `SHELL` n'est pas la variable d'environnement `SHELL` (sinon elle vaudrait `/bin/bash`)

Variables : passage de valeur aux commandes (1)

- Le symbole `$` est utilisé à la fois par `make` pour ses propres variables et le shell les siennes : comment les différentier ?

```
$ touch toto{1,2}.tex
$ cat >Makefile
FILES = toto1.tex toto2.tex toto3.tex
install: $(FILES)
        for a in $(FILES); do \
           cp $a /tmp ; \
        done
$ make
for a in toto1.tex toto2.tex toto3.tex; do \
   cp  /tmp ; \
done
cp: operande de fichier cible manquant apres "/tmp"
Saisissez "cp␣--help" pour plus d'informations.
cp: operande de fichier cible manquant apres "/tmp"
Saisissez ""cp --help"␣pour␣plus␣d'informations.
Makefile:3:␣recipe␣for␣target␣'install'␣failed
make:␣***␣[install]␣Error␣1
```

Variables : passage de valeur aux commandes (2)

- Il faut préfixer le `$` du shell par le `$` de `make` soit `$$` : le premier échappe le second
- Exemple précédent :

```
$ touch toto{1,2}.tex
$ cat >Makefile
FILES = toto1.tex toto2.tex toto3.tex
install: $(FILES)
        for a in $(FILES); do \
           cp $$a /tmp ; \
        done
$ make
for a in toto1.tex toto2.tex toto3.tex; do \
   cp $a /tmp ; \
done
```

- Remarquer la continuation de la ligne de la commande, ce qui revient à écrire :

```
1 FILES = toto1.tex toto2.tex toto3.tex
2 install: $(FILES)
3         for a in $(FILES); do cp $$a /tmp; done
```

Plan

Règles implicites (1)

- Les règles que nous avons vu jusqu'à présent sont explicites : elles se réfèrent à une cible et des sources correspondants à des fichiers dont on connaît les noms
- Or, on écrit toujours les mêmes choses dans un `Makefile`, par exemple compiler un fichier C, c'est toujours :
  `cc -c truc.c -o truc.o`.
  Seul `truc` et les options de compilation changent
- Les règles implicites décrivent ces règles de fabrication sans connaître à l'avance la cible à traiter. Les variables sont utilisées pour paramétrer les commandes
- Attention : GNU `make` utilise sa propre syntaxe, appelé *Pattern rules* pour les différentier des règles implicites POSIX, appelée *Old-fashioned rules* (GNU `make` comprend les deux systèmes)
- Dans ce cours : nous décrivons uniquement les *Pattern rules*

Règles implicites (2)

- Syntaxe général :

```
1 MOTIF : MOTIF
2         commande
3         commande
4          ...
```

  - Un motif est une séquence de caractères et le caractère spécial `%`, l'équivalent de `*` des motifs du shell.
  - `make` cherche les fichiers dont le nom correspond au motif, et leur appliquer la règle pour fabriquer dont le nom est déduit du motif.
- Exemples :
  - les fichiers .o dérivent des .c :
    `%.o: %.c`
  - les fichiers .o dans le répertoire `src` dépendent des .c dans le même répertoire :
    `src/%.o: src/%.c`
    ou des .c du répertoire courant :
    `src/%.o: %.c`
  - les fichiers `.tex` commençant par la lettre `a` pour fabriquer des `.pdf` :
    `a%.pdf: a%.tex`

Règles implicites et variables automatiques

- Erreur courante : écrire `*.o: *.c` au lieu de `%.o: %.c`
- Pour écrire les commandes des règles implicites, nous avons besoin de variables automatiques : elles sont positionnées aux noms des cibles et des dépendances lorsque la règle est appliquée
- Liste (non exhaustive) de variables automatiques :

nom	valeur
`$@`	cible de la règle
`$<`	la première dépendance
`$^`	toutes les dépendances
`$*`	la cible sans l'extension (la tige, *stem*) n'est définie que pour une règle implicite
`$+`	toutes les dépendances en gardant leur multiplicités
`$?`	toutes les dépendances plus récentes que la cible

Variables automatiques : exemples (1)

```
1 toto.o: toto.c main.c main.c
2 ␣␣␣␣␣␣␣@echo '                    cible =' $@
3 ␣␣␣␣␣␣␣@echo '                  1er dep =' $<
4 ␣␣␣␣␣␣␣@echo '                toutes dep =' $^
5 ␣␣␣␣␣␣␣@echo 'toutes dep avec multi =' $+
```

```
% make -W main.c -W toto.c
                  cible = toto.o
                1er dep = toto.c
              toutes dep = toto.c main.c
toutes dep avec multi = toto.c main.c main.c
```

Variables automatiques : exemples (2)

- Comportement de `$*` : il n'a de sens que pour une règle implicite :

```
% cat >Makefile
test.z: test.y
        echo 'Tige de $@: $*'
test.w: test.v
        echo 'Tige de $@: $*'
%.w: %.v
% make test.z -W test.y
echo 'Tige de test.z:'
Tige de test.z:
% make test.w -W test.v
echo 'Tige de test.w: test'
Tige de test.w: test
```

- La tige `*` identifie le motif `%`

Variables automatiques : exemples (3)

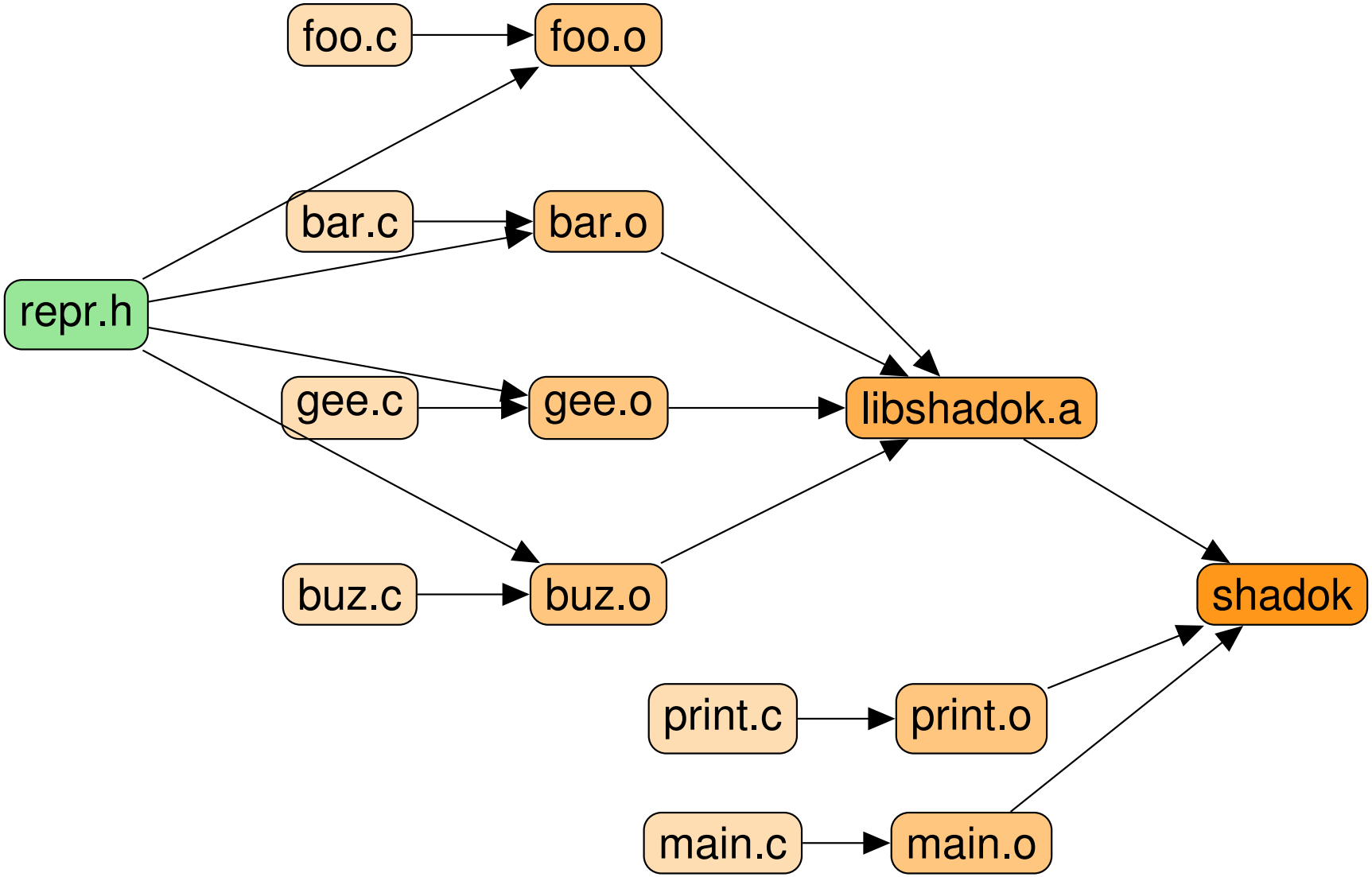
- La tige a un comportement particulier lorsque la cible contient un répertoire :

```
% cat >Makefile
%.b:
        @echo $* $@
a.%.b:
        @echo $* $@
% make dir/foo.b
dir/foo dir/foo.b
dir/foo dir/a.foo.b
```

- La tige filtre le suffixe et le préfixe tout en conservant le répertoire
- Ne pas utiliser `$*` à la légère et surtout pas dans les règles explicites

Règles implicites : exemple (1)

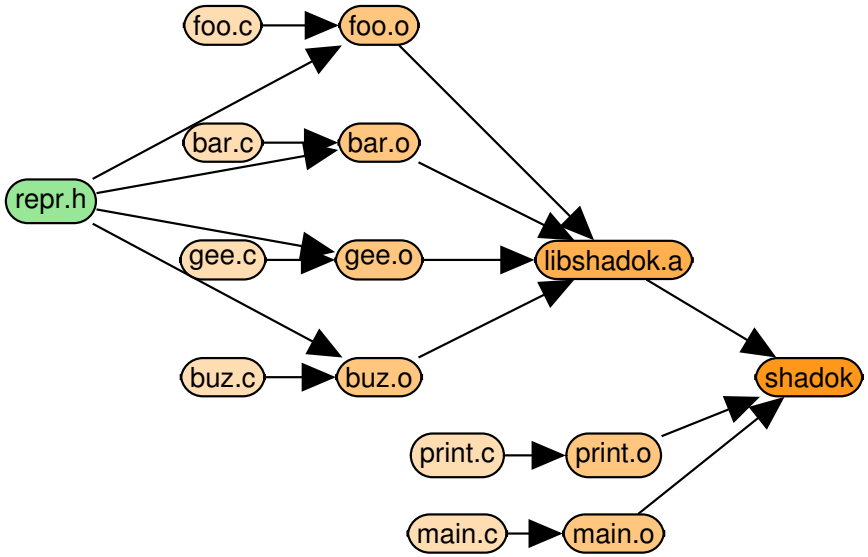
- Soit le graphe de dépendances suivant :



Règles implicites : exemple (2)

- Le `Makefile` associé avec des règles explicites :

```
shadok: libshadok.a print.o main.o
	gcc libshadok.a print.o main.o -o shadok

libshadok.a: foo.o bar.o gee.o buz.o
	ar cr libshadok.a foo.o bar.o gee.o buz.o

foo.o: foo.c repr.h
	gcc -Wall -ansi -pedantic -c foo.c

bar.o: bar.c repr.h
	gcc -Wall -ansi -pedantic -c bar.c

gee.o: gee.c repr.h
	gcc -Wall -ansi -pedantic -c gee.c

buz.o: buz.c repr.h
	gcc -Wall -ansi -pedantic -c buz.c

print.o: print.c
	gcc -Wall -ansi -pedantic -c print.c

main.o: main.c
	gcc -Wall -ansi -pedantic -c main.c

clean:
	rm -f foo.o bar.o gee.o buz.o print.o main.o
	rm -f libshadok.a shadok

.PHONY: clean
```



Règles implicites : exemple (3)

- Factorisation des règles fabriquant les objets (`.o`) à partir des sources :

```
1  shadok: libshadok.a print.o main.o
2  ________gcc libshadok.a print.o main.o -o shadok
3
4  libshadok.a: foo.o bar.o gee.o buz.o
5  ________ar cr libshadok.a foo.o bar.o gee.o buz.o
6
7  #foo.o: foo.c repr.h
8  # gcc -Wall -ansi -pedantic -c foo.c -o foo.o
9
10 #bar.o: bar.c repr.h
11 # gcc -Wall -ansi -pedantic -c bar.c -o bar.o
12
13 #gee.o: gee.c repr.h
14 # gcc -Wall -ansi -pedantic -c gee.c -o gee.o
15
16 #buz.o: buz.c repr.h
17 # gcc -Wall -ansi -pedantic -c buz.c -o buz.o
18
19 #print.o: print.c
20 # gcc -Wall -ansi -pedantic -c print.c -o princ.o
21
22 #main.o: main.c
23 # gcc -Wall -ansi -pedantic -c main.c -o main.o
24
25 %.o: %.c
26 ________gcc -Wall -ansi -pedantic -c $< -o $@
27
28 clean:
29 ________rm -f foo.o bar.o gee.o buz.o print.o main.o
30 ________rm -f libshadok.a shadok
31
32 .PHONY: clean
```

Règles implicites : exemple (4)

- Il n'est pas nécessaire d'indiquer les dépendances du type gee.o : gee.c puisque c'est la règle implicite qui s'en charge
- En revanche, on a perdu les dépendances au fichier `repr.h` ! Solution : ajouter des dépendances explicites

```
1  shadok: libshadok.a print.o main.o
2  ________gcc libshadok.a print.o main.o -o shadok
3  libshadok.a: foo.o bar.o gee.o buz.o
4  ________ar cr libshadok.a foo.o bar.o gee.o buz.o
5  %.o: %.c
6  ________gcc $(CFLAGS)  -c $<
7  foo.o bar.o gee.o buz.o: repr.h
8  clean:
9  ________rm -f foo.o bar.o gee.o buz.o print.o main.o libshadok.a shadok
10 .PHONY: clean
```

- Autre solution pour cet exemple (attention pas toujours possible) : indiquer la dépendance à `repr.h` dans la règle implicite

```
1  shadok: libshadok.a print.o main.o
2  ________gcc libshadok.a print.o main.o -o shadok
3  libshadok.a: foo.o bar.o gee.o buz.o
4  ________ar cr libshadok.a foo.o bar.o gee.o buz.o
5  %.o: %.c repr.h
6  ________gcc -Wall -ansi -pedantic -c $<
7  clean:
8  ________rm -f foo.o bar.o gee.o buz.o print.o main.a libshadok.a shadok
9  .PHONY: clean
```

Règles implicites : exemple (5)

- La touche finale : ajouter des variables pour paramètrer plus facilement la compilation :

```
1  CC      = gcc
2  CFLAGS  = -Wall -ansi -pedantic
3  AR      = ar
4  ARFLAGS = cr
5
6  shadok: libshadok.a print.o main.o
7  ________$(CC) libshadok.a print.o main.o -o shadok
8
9  libshadok.a: foo.o bar.o gee.o buz.o
10 ________$(AR) $(ARFLAGS) libshadok.a foo.o bar.o gee.o buz.o
11
12 foo.o bar.o gee.o buz.o: repr.h
13
14 %.o: %.c
15 ________$(CC) $(CFLAGS) -c $<
16
17 clean:
18 ________rm -f foo.o bar.o gee.o buz.o print.o main.a libshadok.a shadok
19
20 .PHONY: clean
```

Plan

Substitution dans les variables (1)

- On peut pousser plus loin la factorisation en utilisant des variables comme des listes de fichiers sources ou cibles :

```
1  CC      = gcc
2  CFLAGS  = -Wall -ansi -pedantic
3  AR      = ar
4  ARFLAGS = cr
5
6  OBJ_LIB  = foo.o bar.o gee.o buz.o
7  OBJ_PRG  = print.o main.o
8
9  shadok: libshadok.a $(OBJ_PRG)
10 ________$(CC) libshadok.a $(OBJ_PRG) -o shadok
11
12 libshadok.a: $(OBJ_LIB)
13 ________$(AR) $(ARFLAGS) libshadok.a $(OBJ_LIB)
14
15 $(OBJ_LIB): repr.h
16
17 %.o: %.c
18 ________$(CC) $(CFLAGS) -c $<
19
20 clean:
21 ________rm -f $(OBJ_LIB) $(OBJ_PRG) libshadok.a shadok
22
23 .PHONY: clean
```

- Ainsi, s'il l'on souhaite ajouter un module à la bibliothèque `shadok`, il n'y qu'une variable à modifier (ici `OBJ_LIB`)

Substitution dans les variables (2)

- Le `Makefile` reste obscure car on ne voit pas les fichiers sources. On préfère souvent écrire :

```
1  CC      = gcc
2  CFLAGS  = -Wall -ansi -pedantic
3  AR      = ar
4  ARFLAGS = cr
5
6  SRC_LIB  = foo.c bar.c gee.c buz.c
7  SRC_PRG  = print.c main.c
8
9  OBJ_LIB  = $(SRC_LIB:.c=.o)
10 OBJ_PRG  = $(SRC_PRG:.c=.o)
11
12 shadok: libshadok.a $(OBJ_PRG)
13 ________$(CC) libshadok.a $(OBJ_PRG) -o shadok
14
15 libshadok.a: $(OBJ_LIB)
16 ________$(AR) $(ARFLAGS) libshadok.a $(OBJ_LIB)
17
18 $(OBJ_LIB): repr.h
19
20 %.o: %.c
21 ________$(CC) $(CFLAGS) -c $<
22
23 clean:
24 ________rm -f $(OBJ_LIB) $(OBJ_PRG) libshadok.a shadok
25
26 .PHONY: clean
```

en introduisant deux commandes de substitution

Substitution dans les variables (3)

- Syntaxe :
  `$(VAR:suf1=suf2)`
  substitue à la fin de chaque mot de `VAR` le suffixe `suf1` par `suf2`
- Par exemple :

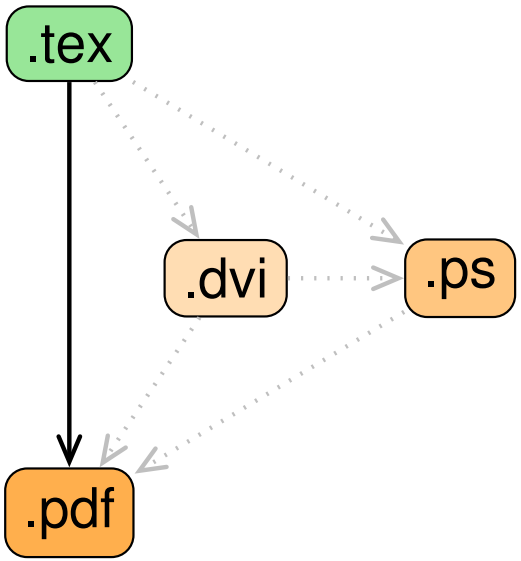
```
$ cat >Makefile
SRC = foo.c bar.c gee.c buz.c
all:
        @echo $(SRC:.c=.o)
$ make
foo.o bar.o gee.o buz.o
```

- La substitution est réalisée lorsque la variable est récursivement évaluée (dans une commande) à moins d'utiliser l'opérateur `::=`

Plan

Cas des règles implicites multiples (1)

- Le comportement diffère du cas des règles explicites :
  - `make` choisit le plus court chemin dans le graphe de dépendance
  - entre deux chemins de même longueur : c'est la première règle qui s'applique

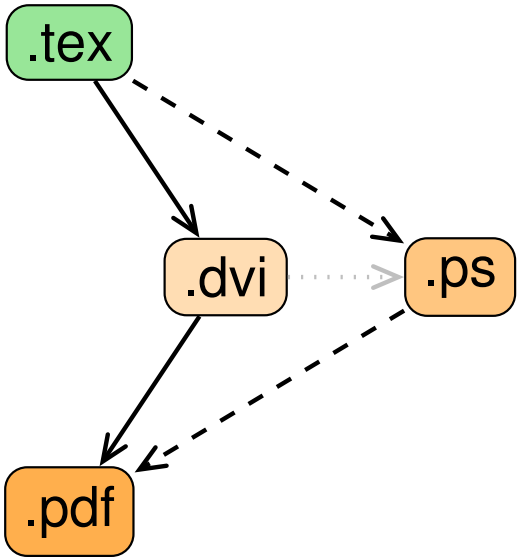


- Un seul plus court chemin : la position de la règle n'importe pas

```
1 %.pdf: %.ps
2 %.pdf: %.dvi
3 %.ps: %.tex
4 %.ps: %.dvi
5 %.dvi: %.tex
6 %.pdf : %.tex
```

Cas des règles implicites multiples (2)

- Le comportement diffère du cas des règles explicites :
  - `make` choisit le plus court chemin dans le graphe de dépendance
  - entre deux chemins de même longueur : c'est la première règle qui s'applique

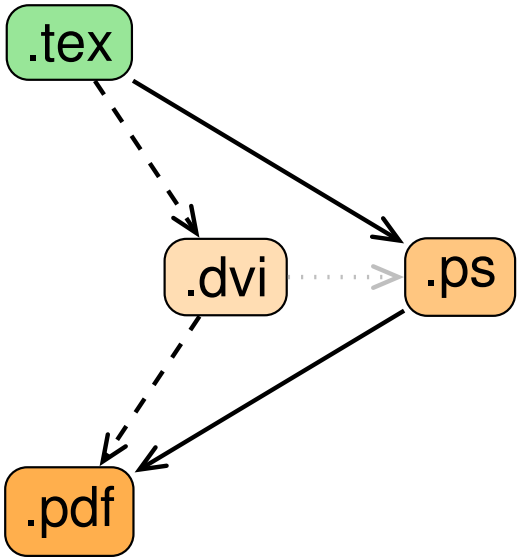


- Deux plus courts chemins : on prend la première

```
1 %.pdf : %.dvi
2 %.pdf : %.ps
3 %.ps : %.tex
4 %.ps : %.dvi
5 %.dvi : %.tex
```

Cas des règles implicites multiples (3)

- Le comportement diffère du cas des règles explicites :
  - `make` choisit le plus court chemin dans le graphe de dépendance
  - entre deux chemins de même longueur : c'est la première règle qui s'applique

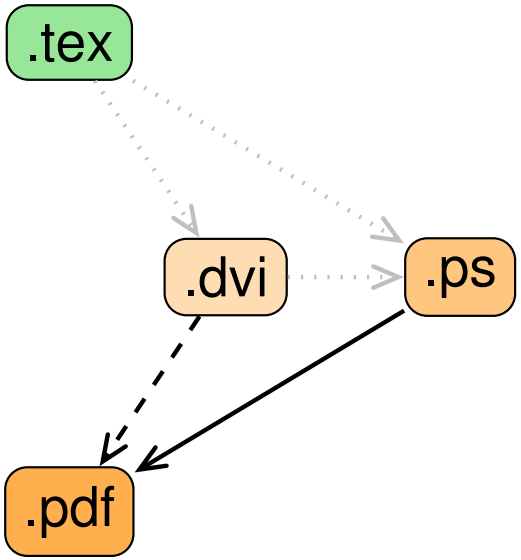


- Deux plus courts chemins : on prend la première

```
1 %.pdf : %.ps
2 %.pdf : %.dvi
3 %.ps : %.tex
4 %.ps : %.dvi
5 %.dvi : %.tex
```

Cas des règles implicites multiples (4)

- Le comportement diffère du cas des règles explicites :
  - `make` choisit le plus court chemin dans le graphe de dépendance
  - entre deux chemins de même longueur : c'est la première règle qui s'applique



- Deux plus courts chemins : on prend la première

```
1 %.pdf : %.ps
2 %.pdf : %.dvi
3 %.ps : %.dvi
4 %.ps : %.tex
5 %.dvi : %.tex
```

Règles implicites et fichiers intermédiaires

- Cas de cibles intermédiaires : les fichiers correspondants seront effacés par `make`
- Par exemple :

```
$ cat >Makefile
%: %.o
        cc -o $@ $^
%.o: %.c
        cc -c $<
$ cat >true.c
int main() {return 0;}
$ make -r true
cc -c true.c
cc -o true true.o
rm true.o
$ make true
make: `true' is up to date.
```

- option `-r` : voir plus loin
- La finalité est la construction de `true` et non pas `true.o`

Plan

Plan

Variables prédéfinies

- En plus des variables automatiques, `make` dispose de variables prédéfinies :

```
% man make | sed -ne '/^[[:blank:]]*-p/,/^$/'
-p, --print-data-base
      Print  the data base (rules and variable values) that results from
      reading the makefiles; then execute as usual or as otherwise spec-
      ified.   This  also prints the version information given by the -v
      switch (see below).  To print the  data  base  without  trying  to
      remake any files, use make -p -f/dev/null.
% LANG=C make -p -f/dev/null | grep -A1 '# default' \
  | sed -r 'd/(--|#|\.)' | wc -l
37
% make -p -f/dev/null | grep '^CC ='
CC = cc
```

- Certaines variables contiennent des . dans leur noms : elles sont réservées et ne doivent pas être modifiées par l'utilisateur

Variables fréquemment utilisées

Variable	Valeur	Remarque
CC	cc	compilateur C
CFLAGS		options du compilateur C
CPPFLAGS		options du préprocesseur C
CXX	c++	compilateur C++
CXXFLAGS		options du compilateur C++
LDLIBS		bibliothèques (édition des liens)
LDFLAGS		options pour l'édition des liens
AR	ar	création de bibliothèques
ARFLAGS		options de AR
MAKE	make	voir Makefile récursifs
LEX	lex	analyseur lexical
YFLAGS		options de YAACC
YACC	yacc	analyseur semantique
LFLAGS		options de LEX

Règles implicites par défaut

- `make` possède également des règles implicites déjà définies !
- Par exemple, `make` sait déjà compiler des sources C grâce cette série de règles :

```
 make -p -f/dev/null | grep -A2 '%.o: %.c$' | grep -v '^#'
%.o: %.c
        $(COMPILE.c) $(OUTPUT_OPTION) $<
% make -p -f/dev/null toto | grep '^COMPILE\.c = '
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
% make -p -f/dev/null toto | grep ' ^OUTPUT_OPTION = '
OUTPUT_OPTION = -o $@
```

- Voici la listes des fichiers faisant l'objet d'une règle par défaut :

```
$ make -p -f/dev/null | grep '.SUFFIXES:'
.SUFFIXES: .out .a .ln .o .c .cc .C .cpp .p .f .F .r .y .l
.s .S .mod  .sym .def .h .info .dvi .tex .texinfo .texi .txinfo
.w .ch .web .sh .elc
```

Plan

La commande `make` sans `Makefile` (1)

- Grâce aux règles par défaut, un `Makefile` peut se réduire aux dépendances

```
$ ls
hello.c hello.h main.c
$ echo "hello: hello.o main.o" > Makefile
$ make
cc     -c -o hello.o hello.c
cc     -c -o main.o main.c
cc   hello.o main.o   -o hello
```

- Grâce aux variables, on peut paramétrer la compilation :

```
$ rm *.o hello
$ make CFLAGS="-g -O2"
cc -g -O2    -c -o hello.o hello.c
cc -g -O2    -c -o main.o main.c
cc   hello.o main.o   -o hello
```

La commande `make` sans `Makefile` (2)

- D'autres règles pour la compilation C :

```
$ rm main.o
$ make main.o
cc     -c -o main.o main.c
$ echo 'hello: hello.c main.c' > Makefile
$ rm *.o hello
$ make
cc      hello.c main.c   -o hello
```

- Finalement, pour construire à partir d'une source unique, pas besoin de `Makefile` !

```
$ mkdir t; cd t
$ echo 'int main() {return 0;}' >true.c
$ make true
cc      true.c   -o true
```

- `make` est l'interface de compilation d'emacs, `M-x compile`, autant savoir s'en servir !

Plan

Plan

Dépendances statiques contre automatiques (1)

- Certains compilateurs (par exemple `gcc` ou `ocamel`) sont capables de déterminer les dépendances d'un fichier source.
- Exemple en compilation C : soit les trois fichiers suivants :

```
% more hello.c
#include <stdio.h>
 void hello(void) {
  printf("Hello world\n");
}
% more hello.h
void hello( void);
% more main.c
#include "hello.h"
int main(void) {
  hello();
  return 0;
}
```

- L'option `-MM` de `gcc` calcule les dépendances du ou des fichiers sources :

```
$ gcc -MM hello.c main.c
hello.o: hello.c
main.o: main.c hello.h
```

Dépendances statiques contre automatiques (2)

- On peut les copier dans le `Makefile`, mais il y a mieux : on peut les inclure avec la directive include :

```
1 -include .depend
2 depend:
3         gcc -MM hello.c main.c > .depend
4 .PHONY: depend
```

- Il faut préfixer include par - sinon on provoque une erreur si le fichier `.depend` n'existe pas !
- Le choix d'un nom commençant par . est à cause de la commande `ls`
- Dépendances automatique : très pratique pour les gros projets !

Dépendances statiques contre automatiques (3)

- Makefile complet :

```
1  CC = gcc
2
3  hello: hello.o main.o
4          $(CC) hello.o main.o  -o hello $(LDFLAGS)
5  %.o: %.c
6          $(CC) -c $< -o $@ $(CFLAGS)
7  -include .depend
8  depend:
9          gcc -MM hello.c main.c > .depend
10 clean:
11         rm -f hello.o main.o hello .depend
12
13 .PHONY: clean depend
```

- Expérimentons :

```
$ make
gcc -c hello.c -o hello.o
gcc -c main.c -o main.o
gcc hello.o main.o  -o hello
$ touch hello.h
$ make
make: `hello' is up to date.
$ make depend
gcc -MM hello.c main.c > .depend
$ make
gcc -c main.c -o main.o
gcc hello.o main.o  -o hello
```

Dépendances statiques contre automatiques (4)

- La cible `depend` doit être virtuelle, c'est-à-dire qu'il ne faut pas que ce soit `.depend`)
- si elle ne l'est pas, la production de `.depend` devient contradictoire avec la cible de nettoyage :

```
$ cat Makefile
-include .depend
.depend:
        gcc -MM hello.c main.c > .depend
clean:
        rm -f .depend
.PHONY: clean
$ rm -f .depend
$ make clean
gcc -MM hello.c main.c > .depend
rm -f .depend
```

- `make clean` : l'absence de `.depend` provoque implicitement sa fabrication, puis la cible `clean` l'efface . . .

Plan

Récursion dans les `Makefile` (1)

- Les sources d'un logiciel sont très souvent organisés en répertoires : bibliothèque, commande, documentation, tests, ...
- L'usage veut qu'il y ait un `Makefile` par répertoire, et un `Makefile` principal qui dirige l'ensemble
- Le principe : depuis le répertoire principal, on indique dans le Makefile des appels à `make` dans les sous-répertoires
- Nécessité de transmettre des variables aux nouvelles instances de `make`
- La variable `MAKE` est également utilisée pour être certain que c'est la même commande `make` qui sera utilisée :

```
$ echo "all:;␣@echo␣MAKE=$(MAKE)" > Makefile
$ make
MAKE=make
$ gmake
MAKE=gmake
```

- Ici `gmake` est la version GNU et `make` est la version BSD

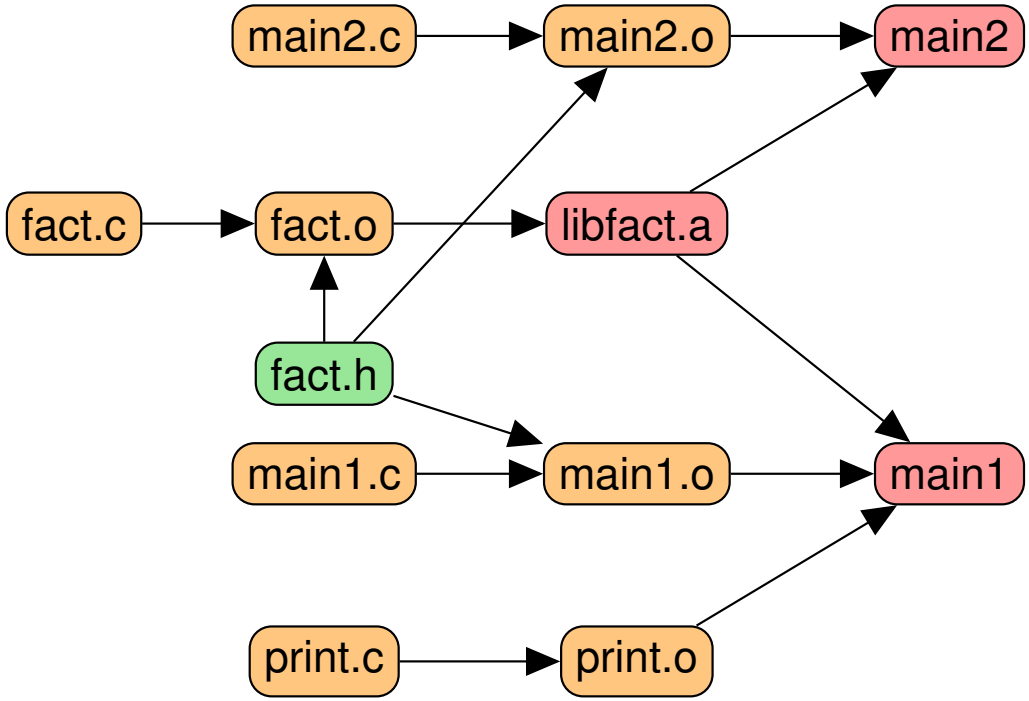
Récursion dans les `Makefile` (2)

- Un exemple complet (compilation, nettoyage et installation) avec la hiérarchie suivante :

```
$ tree projet
projet
|-- README
|-- COPYING
|-- lib
|   |-- fact.c
|   \-- fact.h
\-- src
    |-- main1.c
    |-- main2.c
    \-- print.c

2 directories, 7 files
```

Récursion dans les `Makefile` (3)

- Et son graphe des dépendances



- Pour illustrer la récursion, le graphe est découpé en deux parties : `lib` contiendra `libfact.a` et `src` les deux commandes `main1` et `main2`

Récursion dans les `Makefile` (4)

- La partie `lib`, le fichier `projet/lib/Makefile` :

```
1  CC = gcc
2  AR = ar
3  RANLIB = ranlib
4  DESTDIR = /tmp
5
6  libfact.a: fact.o
7          $(AR) cr $@ $^
8          $(RANLIB) $@
9  fact.o: fact.h
10 %.o: %.c
11         $(CC) -c $< $(CPPFLAGS) $(CFLAGS) -o $@
12 clean:
13         rm -f fact.o libfact.a
14 install:
15         mkdir -p $(DESTDIR)/include
16         cp fact.h $(DESTDIR)/include
17         mkdir -p $(DESTDIR)/lib
18         cp libfact.a $(DESTDIR)/lib
19 .PHONY: clean install
```

- Il faut indiquer au préprocesseur C où trouver le fichier `fact.h`

```
$ fgrep 'fact.h' projet/src/*.c
#include <fact.h>
#include <fact.h>
```

- Pour `projet/src/Makefile` :

```
1  CC       = gcc
2  CPPFLAGS = -I../lib
3  DESTDIR  = /tmp
4
5  SRC = print.c main1.c main2.c
6
7  all: main1 main2
8
9  main1: main1.o print.o ../lib/libfact.a
10         $(CC) $(LDFLAGS) -o $@ $^
11
12 main2: main2.o ../lib/libfact.a
13         $(CC) $(LDFLAGS) -o $@ $^
14
15 %.o: %.c
16         $(CC) -c $(CFLAGS) $(CPPFLAGS) \
17               -o $@ $<
18
19 ../lib/libfact.a:
20         cd ../lib && $(MAKE) libfact.a
21
22 install:
23         mkdir -p $(DESTDIR)/bin
24         cp main1 main2 $(DESTDIR)/bin
25 clean:
26         $(RM) main1 main2 $(SRC:.c=.o)
27
28 .PHONY: clean all install
```

Récursion dans les `Makefile` (5)

- Le `Makefile` principal est celui à la racine du projet : `projet/Makefile`
- Il se charge d'appeler correctement les `Makefile` des répertoires supérieurs, délègue les bonnes cibles et variables

```
1  DESTDIR = /usr/local
2  all :
3          cd lib && $(MAKE)
4          cd src && $(MAKE) all
5  clean:
6          cd src && $(MAKE) clean
7          cd lib && $(MAKE) clean
8  install:
9          cd lib && $(MAKE) install DESTDIR=$(DESTDIR)
10         cd src && $(MAKE) install DESTDIR=$(DESTDIR)
11         mkdir -p $(DESTDIR)/share/doc/projet
12         cp COPYING README $(DESTDIR)/share/doc/projet
13
14 .PHONY: all clean install
```

- La variable `DESTDIR` désigne le répertoire d'installation, ici `/usr/local` par défaut. Dans les autres `Makefile` elle est positionnée à `/tmp` ce qui permet de faire des tests d'installation

Récursion dans les `Makefile` (6)

- Exemple d'utilisation

```
$ cd projet
$ make all
cd lib && /usr/bin/make
gcc    -c -o fact.o fact.c
ar cr libfact.a fact.o
cd src && /usr/bin/make all
gcc -c  -I../lib -o main1.o main1.c
gcc -c  -I../lib -o print.o print.c
gcc  -o main1 main1.o print.o ../lib/libfact.a
gcc -c  -I../lib -o main2.o main2.c
gcc  -lm -o main2 main2.o ../lib/libfact.a
$ make install DESTDIR=/home/bereziat
cd lib && /usr/bin/make install DESTDIR=/home/bereziat
mkdir -p /home/bereziat/include
cp fact.h /home/bereziat/include
mkdir -p /home/bereziat/lib
cp libfact.a /home/bereziat/lib
cd src && /usr/bin/make install DESTDIR=/home/bereziat
mkdir -p /home/bereziat/bin
cp main1 main2 /home/bereziat/bin
mkdir -p /home/bereziat/share/doc/projet
cp COPYING README /home/bereziat/share/doc/projet
$ make clean
cd src && /usr/bin/make clean
rm -f main1 main2 print.o main1.o main2.o
cd lib && /usr/bin/make clean
rm -f fact.o libfact.a
```

Précautions avant diffusion

- Si l'on souhaite diffuser un logiciel, il convient de s'assurer que les `Makefile` contiennent toute l'information utile
- Ils ne doivent pas se reposer sur les variables et règles définies par défaut
- L'option `-r` de `make` désactive les règles et variables prédéfinies, il est recommandé de l'utiliser !

```
$ ls
main.c hello.c
$ cat >Makefile
main: main.o hello.o
        gcc $^ -o $@
$ make
gcc    -c -o hello.o hello.c
gcc    -c -o main.o main.c
gcc hello.o main.o   -o hello
$ rm *.o hello
$ make -r
make: *** No rule to make target `main.o'.  Stop.
```

Plan

Pour aller plus loin

- Le manuel texinfo de `GNUmake` : il est exhaustif et sa version en ligne est `http://www.gnu.org/software/make/manual/make.html`
  - les fonctions de `GNUmake` (non recommandé de les utiliser dans ce cours)
  - les variables locales à une règle
  - les cibles spéciales
  - ...
- La norme POSIX pour `make` `https://www.opengroup.org/onlinepubs/009695399/utilities/make.html`