



Institut  
Mines-Télécom

# GNU Make et Makefiles

...ou comment automatiser un peu les choses

Alexis Polti



# Licence de droits d'usage



## Licence académique avec modifications

Par le téléchargement ou la consultation de ce document, l'utilisateur accepte la licence d'utilisation qui y est attachée, telle que détaillée dans les dispositions suivantes, et s'engage à la respecter intégralement.

La licence confère à l'utilisateur un droit d'usage sur le document consulté ou téléchargé, totalement ou en partie, dans les conditions définies ci-après et à l'exclusion expresse de toute utilisation commerciale.

Le droit d'usage défini par la licence autorise un usage dans un cadre académique, par un utilisateur donnant des cours dans un établissement d'enseignement secondaire ou supérieur et à l'exclusion expresse des formations commerciales et notamment de formation continue. Ce droit comprend :

- le droit de reproduire tout ou partie du document sur support informatique ou papier,
- le droit de diffuser tout ou partie du document à destination des élèves ou étudiants
- le droit de modifier la forme ou la présentation du document
- le droit d'intégrer tout ou partie du document dans un document composite et de le diffuser dans ce nouveau document dérivé à condition que :
  - L'auteur soit informé
  - le nouveau document soit diffusé dans un cadre académique sous la même licence que le document source
  - Les mentions relatives à la source du document et/ou à son auteur doivent être conservées dans leur intégralité **et de façon clairement lisible.**

Tout autre usage que ceux prévus par la licence est soumis à autorisation préalable et expresse de l'auteur :

[alexis.polti@telecom-paris.fr](mailto:alexis.polti@telecom-paris.fr)

# tl;dr

## ● Comment ça marche ?

- règles explicites et syntaxe de base
- règles implicites
- variables / variables automatiques
- tout (ou presque) est déjà écrit

## ● Conseils :

- un bon Makefile est un Makefile court
- lancez-vous !

## • Sources de documentation

- ce cours...
- documentation officielle :
  - <http://www.gnu.org/software/make/manual/>
  - man make
- livres :
  - "Managing projects with GNU Make" - O'Reilly
  - "Make — A Program for Maintaining Computer Programs" - S.I. Feldman / BSD and System V manuals.
- profs
- avant toute question, lisez la documentation officielle...

## • make

- permet d'automatiser l'enchaînement de tâches interdépendantes, en n'effectuant que le minimum d'opérations nécessaires
- Exemples :
  - compilation
  - synthèse de code Verilog
  - transformations batch d'images
  - ...

## GNU make

- particulièrement adapté quand chaque tâche :
  - utilise des fichiers
  - pour produire un ou plusieurs fichiers
  - → comparaison de dates de modification des fichiers
- particulièrement adapté à la compilation de programmes (C, C++, Ada, ...)

## ● Fichier de commande :

1. Tout fichier spécifié par - f
2. GNUmakefile
3. makefile
4. Makefile

## ● contient :

- règles
- définitions de variables / macros
- directives
- commentaires

- **Une règle indique deux choses :**
  - quand est-ce qu'une cible doit être reconstruite
  - comment la reconstruire
  
- À partir l'ensemble des règles, make :
  - construit un arbre de dépendances
  - examine les dates des dépendances / cibles
  - détermine le minimum d'actions à effectuer
  - les effectue



## • Règles

- tâche à effectuer
- une ou plusieurs
- généralement : cible = fichier de sortie

- 0 ou plusieurs
- cibles préalables à l'exécution de la tâche

`cible ... : dépendances ...`

`commandes`

`...`

`...`

- Commencent par une tabulation
- Liste des commandes à effectuer

## ● Quand ?

- la cible est reconstruite :
  - si elle n'a aucune dépendance et qu'elle n'existe pas déjà
  - ou si une de ses dépendances est plus récente ou inexistante

## ● Comment ?

- chaque commande est
  - affichée, sauf si elle est précédée de @
  - puis exécutée dans son propre shell
- une erreur déclenche l'arrêt de make,
  - sauf si la commande est précédée de -
  - ou si make est invoqué avec l'option -k

## ● Lancement de make

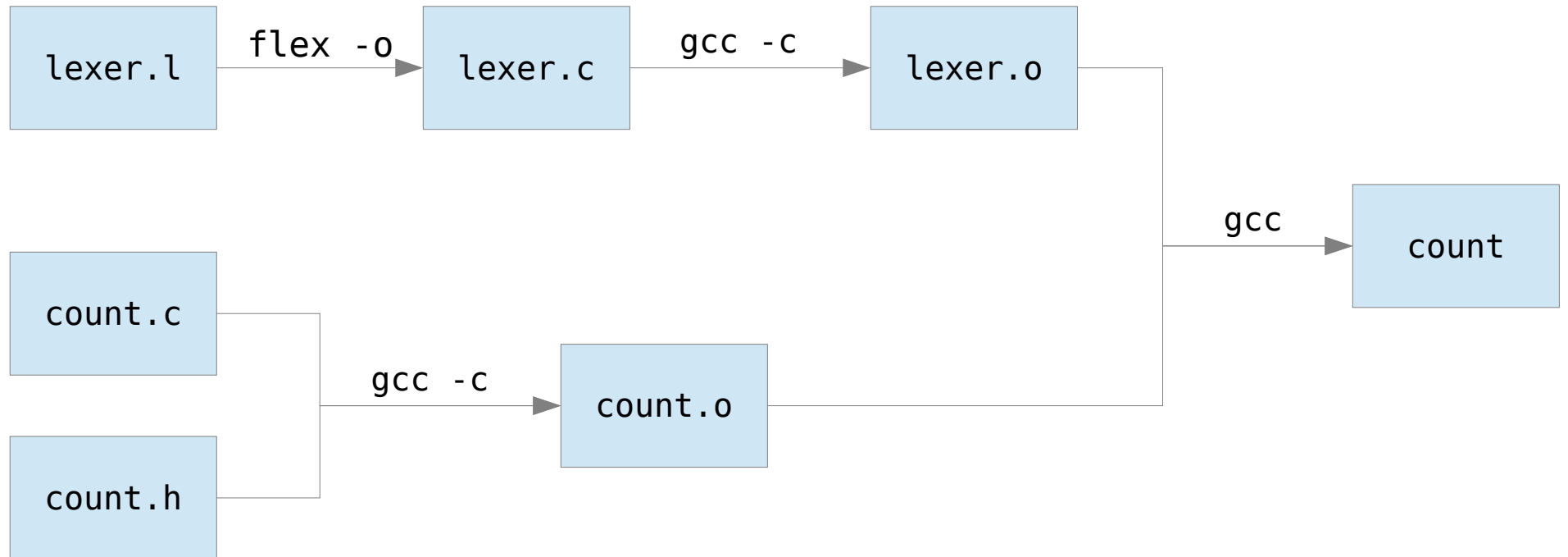
- `make` : reconstruit la première cible du `Makefile`
- `make cible` : reconstruit la cible spécifiée

## ● Options

- `-n` : simule une exécution de `make`
- `-r` : supprime les règles pré-écrites
- `-R` : supprime les variables pré-écrites
- `-k` : continue le plus loin possible même en cas d'erreur
- `-W fichier` : simule que `fichier` vient d'être modifié
- `-d` : affiche tout ce que `make` tente

# Exemple

## • Exemple



## Exemple

### • Exemple

```
count: lexer.o count.o
      gcc -g lexer.o count.o -lfl -o count
```

```
lexer.o: lexer.c
      gcc -O2 -g -c lexer.c -o lexer.o
```

```
count.o: count.c count.h
      gcc -O2 -g -c count.c -o count.o
```

```
lexer.c : lexer.l
      flex -o lexer.c lexer.l
```

```
clean:
      rm -f count
      rm -f *.o
      rm -f lexer.c
```

## Exemple

### • Exemple

```
count: lexer.o count.o
      gcc -g lexer.o count.o -lfl -o count
```

```
lexer.o: lexer.c
      gcc -O2 -g -c lexer.c -o lexer.o
```

```
count: lexer.o count.o
      gcc -g -o count count.o
```

```
lexer.c : lexer.l
      flex -o lexer.c lexer.l
```

```
clean:
      rm -f count
      rm -f *.o
      rm -f lexer.c
```

**NON !**

## Exemple

### • Exemple

```
count: lexer.o count.o
      gcc -g lexer.o count.o -lfl -o count
```

```
lexer.o: lexer.c
      gcc -O2 -g -c lexer.c -o lexer.o
```

```
count.o: count.c count.h
      gcc -O2 -g -c count.c -o count.o
```

```
lexer.c : lexer.l
      flex -o lexer.c lexer.l
```

```
clean:
      rm -f count
      rm -f *.o
      rm -f lexer.c
```

**Duplication de  
code  
NOOOOOON !**



## Où en est-on ?



### ● On a vu

- la syntaxe de base
- ce qu'il ne faut jamais faire !

### ● On va voir maintenant

- les variables
- les règles
- que tout est déjà écrit : → customisation !

### ● On verra plus tard

- les subtilités



## • Variables

- types :
  - automatiques
  - explicites
    - simple / immédiates
    - récursives / différées

### ● Variables automatiques

- $\$@$  : la cible de la règle.
  - si plusieurs cibles : celle qui est actuellement produite
- $\$^$  : les dépendances sans doublon
- $\$<$  : la première des dépendances
- $\$?$  : les dépendances plus récentes que la cible
- $\$+$  : les dépendances avec doublons
- $\$*$  : la racine (*stem*)

## • Deux types

- immédiates / simples
- déferées / récursives
  
- utilisables dans tout contexte
- n'importe quel caractère sauf
  - #
  - :
  - =
  - et espace / tab
- sensibles à la casse

## Variables explicites

- utilisation : `$(COMPILE) ${COMPILE}`
- affectation :
  - selon le type
  - les espaces avant le contenu sont ignorés
  - les espaces après le contenu sont **gardés**

- exemple :

```
dir := /foo/bar      # répertoire utile  
dir contient maintenant "/foo/bar      "
```

# Variables : affectation

## ● Variables simples

- $CC := \$(PREFIX)gcc$
- lors de la définition, le membre de droite est évalué /expansé et affecté au membre de gauche

## ● Variables récursives

- $CC = \$(PREFIX)gcc$
- le membre de droite est stocké verbatim (sans être évalué / expansé) dans le membre de gauche

## ● Lors de l'affectation de jokers à une variable, l'expansion n'a lieu qu'au moment de l'utilisation :

- $OBJJS = *.o$
- OBJJS contient "\*.o"

## ● Opérateurs

- += : pour ajouter quelque chose à la fin
- ?= : n'affecte que si la variable n'existe pas encore
- substitution :

```
foo := a.o b.o c.o
```

```
bar := $(foo:%.o=%.c)
```

### ● Variables : règles d'expansion

- dans les affectations :
  - le membre de gauche est toujours expansé lors de la lecture de l'affectation
  - le membre de droite est expansé immédiatement pour :=
  - pour +=, cela dépend du type de la variable
- dans les règles :
  - les cibles et dépendances sont expansées immédiatement à la lecture du Makefile
  - les commandes sont expansées à l'utilisation

## Variables : exemple

### • Corrigez ce Makefile

- (sensé afficher l'espace libre de /tmp)

```
$(OUTPUT_DIR) :  
    cd $@  
    @$ (PRINTF) "Free disk space in $@\n"  
    @$ (DF) $(DF_FLAGS) .
```

```
OUTPUT_DIR := /tmp
```

```
PRINTF = $(USRBIN)/printf  
DF      = $(BIN)/df  
BIN     := /bin  
USRBIN  := /usr/bin  
DF_FLAGS = -h  
DF_FLAGS = $(DF_FLAGS) -v
```



## • Variables : d'où viennent-elles ?

- de la ligne de commande
  - exemple : `make CFLAGS=-g`
  - prime sur tout, sauf si directive `override`
- du Makefile
  - c'est ce qu'on vient de voir !
- de l'environnement
  - exemple : `CFLAGS=-g make`
  - plus basse des priorités

## ● Variables spécifiques

- spécifiques à une cible particulière
- portent sur la cible ainsi que toutes ses dépendances
- assignation défermée jusqu'au début de la construction de la cible

- format :

- `target: variable = value`
- `target: variable := value`
- `target: variable += value`
- `target: variable ?= value`

- exemple :

```
parser.o truc.o : CPPFLAGS += -DUSE_MALLOC=1
```

## • Quatre type de règles

- explicites
  - génériques
  - statiques
  - double ":" (double-colon)
- 
- La plupart sont déjà écrites !

## ● Règles explicites

- chaque cible / dépendance est mentionnée explicitement
- les dépendances sont cumulatives
- exemples :

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

```
kbd.o : kbd.c kbd.h  
gcc -c kbd.c -o kbd.o
```

```
command.o : command.c  
gcc -c command.c -o command.o
```

```
files.o : files.c  
gcc -c files.c -o files.o
```

### ● Règles explicites

- une même règle explicite peut concerner plusieurs cibles
  - la dépendance s'applique à chacune des cible
  - l'action est la même pour chaque cible

```
error warning : log.txt  
grep @$ $^ > @$
```

### ● Règles génériques

- les noms de fichiers sont spécifiés par des expressions régulières simples.
- caractère générique : "%" (*racine / stem*)
- ce caractère n'est pas utilisable dans les commandes. On y utilise à la place \$\*
- Exemples :

```
% .o : % .c % .h  
gcc -c $< -o $@
```

```
all : error.log warning.log  
%.log : output.txt  
grep $* $^ > $@
```

## ● Règles génériques

- si l'expression régulière ne contient pas de "/" :
  - les noms de répertoires sont enlevés lors de la recherche de l'expression régulière
  - mais sont rajoutés lors de l'utilisation de la racine
- Exemple
  - e%t :
    - dans la cible : e%t matche src/eat
    - dans les dépendances : c%r donne src/car
    - dans les commandes : \$\* est remplacé par src/a

### ● Règles génériques

- une règle générique possédant plusieurs cibles indique à make que la commande met à jour toutes ses cibles en même temps
- Exemple

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



### ● Règles génériques

- elles sont utilisées en dernier ressort, après avoir essayé toutes les règles explicites, **si les prérequis existent ou peuvent être construits.**
- elles peuvent être chaînées :
  - les fichiers intermédiaires sont alors automatiquement détruits
  - sauf si on les ajoute en dépendances de `.SECONDARY`



## Exemple

- **On simplifie notre exemple usuel !**

### ● Règles statiques

- permettent de personnaliser l'application d'une règle générique à un ensemble de fichier
- Exemple

```
file1.o file2.o: %.o: %.c  
$(CC) -Os -DDEBUG -c $< -o $@
```

### ● Règles double-deux-points

- anecdotiques....
- permettent de modifier la façon dont est construite une cible selon la dépendance qui déclenche la reconstruction
- voir la documentation pour plus de détails

### ● Cibles spéciales

- .PHONY : indique que la cible n'est pas un fichier, et qu'elle doit toujours être reconstruite
  - à quoi cela peut-il servir ?
- .INTERMEDIATE
- .SECONDARY
- .PRECIOUS
- .DELETE\_ON\_ERROR

## Où en est-on ?



### ● On a vu

- la syntaxe de base
- les variables
- les règles

### ● On va voir

- que tout est déjà écrit : → customisation !
- les subtilités

## Bonne nouvelle !

### • Tout (ou presque) est déjà écrit !

- des règles génériques et des variables ciblant les principaux langages ont déjà été pré-écrites
  - vous avez donc un Makefile par défaut (et propre)
  - on peut afficher son contenu par `make -p -f/dev/null`
  - vos variables d'environnement sont ajoutés au début du Makefile par défaut en variables immédiates
  - votre Makefile est ajouté à la fin du Makefile par défaut
- vous pouvez donc :
  - personnaliser les règles / variables pré-écrites
  - au besoin, rajouter vos propres règles / variables

# Makefile par défaut

- Exemple : produire un exécutable à partir d'un fichier C



Déjà écrit

```
CC = cc
LINK.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)
%: %.c
    $(LINK.c) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

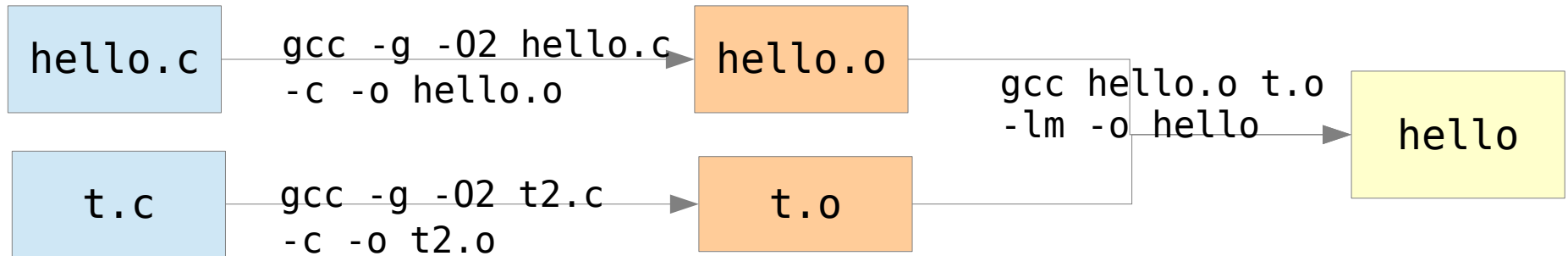
Votre Makefile

```
CFLAGS = -g -O2
LDLIBS = -lm
all : truc
```



# Makefile par défaut

- Exemple : produire un exécutable à partir de deux fichiers C



```
CC = cc
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
%.o: %.c
    $(COMPILE.c) $(OUTPUT_OPTION) $<
LINK.o = $(CC) $(LDFLAGS) $(TARGET_ARCH)
%: %.o
    $(LINK.o) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

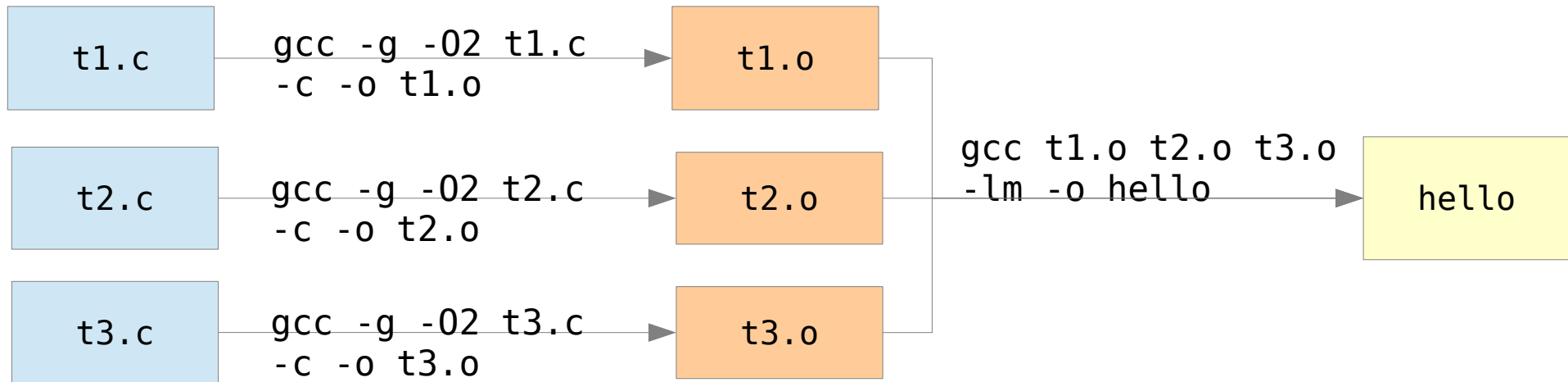
Déjà écrit

```
CFLAGS = -g -O2
LDLIBS = -lm
hello : hello.o t2.o
```

Votre Makefile

# Makefile par défaut

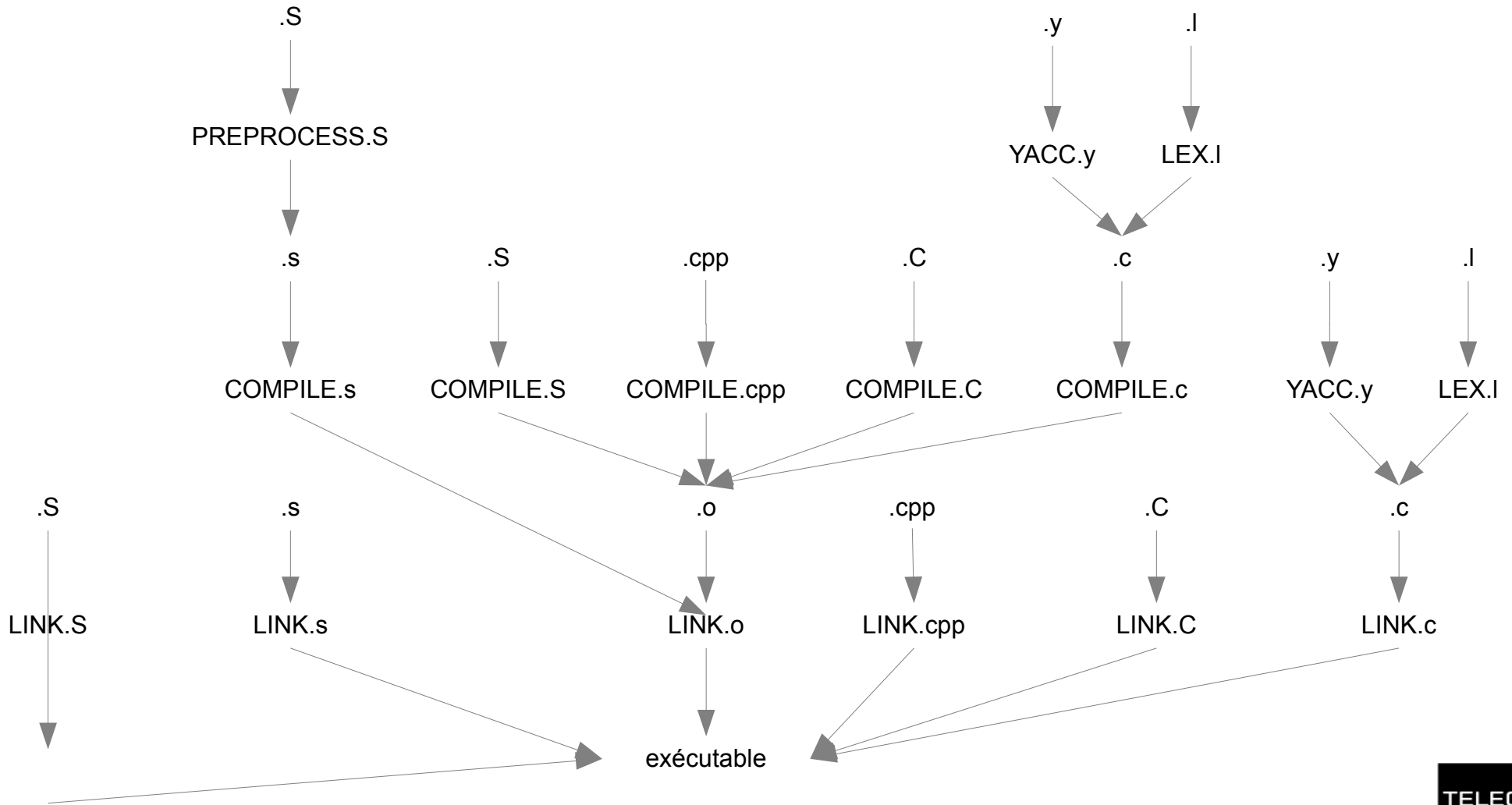
- Exemple : produire un exécutable à partir de n fichiers C



Votre Makefile

```
CFLAGS = -g -O2
LDLIBS = -lm
OBJS = t1.o t2.o t3.o
EXE = hello
$(EXE) : $(OBJS)
    $(LINK.o) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

# Règles par défaut usuelles



## Où en est-on ?



### ● On a vu

- la syntaxe de base
- les règles, les variables
- que tout est déjà écrit

### ● On va voir

- quelques subtilités

## ● Fonctions

- `$(patsubst search_pattern,replace_pattern,text)`
  - exemple : `$(patsubst %.o,%.c,$(OBJECTS))`
  - `$(VAR:pattern=replacement)` est équivalent à `$(patsubst pattern,replacement,$(VAR))`
- `$(filter pattern...,text)`
  - exemple : `C_and_S = $(filter %.c %.s,$(SOURCES))`
- Il en existe beaucoup d'autres (`$shell`, `$wildcard`, ...)

## ● Dépendances

- complexes pour des gros projets : `.c`, `.h`, ...
- on peut les générer automatiquement avec `gcc`
  - on les stocke dans des fichiers `.d`
  - `gcc -MM -MF count.d -MP -MT count.o count.c`
- ces fichiers sont inclus dans le Makefile grâce à la directive `include`
  - `include` traite ses arguments comme
    - des cibles à reconstruire
    - des dépendances du Makefile
  - avant toute action, `make` cherche à reconstruire le Makefile

### ● Exemple

```
SOURCES = count.c lexer.c
```

```
count : count.o lexer.o -lfl
```

```
-include $(subst .c,.d,$(SOURCES))
```

```
%.d : %.c
```

```
$(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M -MF $@\
-MP $<
```

### • Autre exemple

```
SOURCES = count.c lexer.c  
CFLAGS += -MD -MP
```

```
count : count.o lexer.o -lfl
```

```
-include $(subst .c,.d,$(SOURCES))
```



## ● Séparation des sources et déclarations ?

- la variable VPATH indique les répertoires additionnels où chercher les dépendances
- la directive vpath permet de faire la même chose type de fichier par type de fichier

● exemple :

```
VPATH = ~/src ~/include  
vpath %.c ~/projet/src  
vpath %.h ~/projet/include
```

## Exercice

- **Écrivez le Makefile du projet que vous trouverez sur le site de l'UE, de façon à effectuer la compilation ainsi :**

```
arm-none-eabi-gcc -Wall -Werror -g -Og -mthumb -c -o main.o main.c
```

```
arm-none-eabi-gcc -Wall -Werror -g -Og -mthumb -c -o t1.o t1.c
```

```
arm-none-eabi-gcc -Wall -Werror -g -Og -mthumb -c -o t2.o t2.c
```

```
arm-none-eabi-gcc -g -O2 -mthumb -c -o stubs.o libs/stubs.c
```

```
arm-none-eabi-gcc -L/opt/mylibs -mthumb main.o t1.o stubs.o t2.o -lm -o hello
```

- On doit pouvoir générer hello en tapant seulement make
- N'oubliez pas :
  - la cible clean
  - les dépendances en .h
  - que le Makefile doit être propre et concis !

# Licence de droits d'usage



## Licence académique avec modifications

Par le téléchargement ou la consultation de ce document, l'utilisateur accepte la licence d'utilisation qui y est attachée, telle que détaillée dans les dispositions suivantes, et s'engage à la respecter intégralement.

La licence confère à l'utilisateur un droit d'usage sur le document consulté ou téléchargé, totalement ou en partie, dans les conditions définies ci-après et à l'exclusion expresse de toute utilisation commerciale.

Le droit d'usage défini par la licence autorise un usage dans un cadre académique, par un utilisateur donnant des cours dans un établissement d'enseignement secondaire ou supérieur et à l'exclusion expresse des formations commerciales et notamment de formation continue. Ce droit comprend :

- le droit de reproduire tout ou partie du document sur support informatique ou papier,
- le droit de diffuser tout ou partie du document à destination des élèves ou étudiants
- le droit de modifier la forme ou la présentation du document
- le droit d'intégrer tout ou partie du document dans un document composite et de le diffuser dans ce nouveau document dérivé à condition que :
  - L'auteur soit informé
  - le nouveau document soit diffusé dans un cadre académique sous la même licence que le document source
  - Les mentions relatives à la source du document et/ou à son auteur doivent être conservées dans leur intégralité **et de façon clairement lisible.**

Tout autre usage que ceux prévus par la licence est soumis à autorisation préalable et expresse de l'auteur :

[alexis.polti@telecom-paris.fr](mailto:alexis.polti@telecom-paris.fr)