Processus de Compilation

compilation C

source `.c` → objet `.o` (code natif)

```
gcc -Wall -c -o foo.o foo.c
```

objet `.o` → exécutable `.exe` (dépend du processeur)

```
gcc -Wall -o foo.exe foo.o
```

code de chargement, fichier objet, appel à `main`

Compilation ML

source `.ml` → objet `.cmo` (bytecode)

```
ocamlc -c foo.ml
```

objet `.cmo` → exécutable `.exe` (dépend de caml)

```
ocamlc -o foo.exe foo.cmo
```

machine virtuelle caml, exécution des phrases.

Les interfaces

Informations de types

Compiler un appel de fonction : le nom et le type (la déclaration) suffisent !

Déclaration dans un fichier `.h` implémentation dans une librairie (statique : `.a`, dynamique : `.so`)

```
typedef ... t;
t foo(t);
```

Les modules Caml

les déclarations sont dans un fichier d'interface `.mli`

```
type t
val foo: t -> t
```

le reste du code de `foo.ml` est caché. Les descriptions sont dans des fichiers `.cmi`.

ocamlc -c foo.mli

Visibilité

Fichiers `.cmi`

La valeur `foo` spécifiée dans le fichier `bar.mli` appartient au *module* `Bar` et est référencée par `Bar.foo`.

Directive `open` : de ne pas préfixer la valeur par le nom.
Compilation : `bar.cmo` dans la ligne de commande.

Les modules top level, `struct`, `sig`

```
module Foo =
struct
  let foo(x) = ...
end;;
module type Bar =
sig
  val foo: ...
end;;
```

Structurer les sources

Faciliter le développement

Découper en fichiers ou paquetages.

Réduire les dépendances (couplage).

Automatiser la production

Script de compilation → `Makefile`

Graphe sans cycle de description de dépendances.

Une cible (target) est un fichier, une dépendance est une liste de fichiers.

Une règle est une commande permettant de créer la cible à partir des dépendances.

Faciliter la maintenance

Gestion collaborative : `cvs`, `svn`.

Écrire un `Makefile`

Règles de production

```
# les commandes a la ligne avec une tabulation
foo.cmo: foo.ml
  ocamlc -c foo.ml
```

Déclaration des suffixes gérés

```
.SUFFIXES: .ml .mli .cmi .cmo .tex .pdf
```

Définition des variables

la variable `$@` est la cible, la variable `$<` les dépendances.

```
CC = gcc -ggdb -Wall -c
foo.o: foo.c
  $(CC) -o $@ $<
```

Règles par défaut

```
# n'ont pas de dépendances
.mli.cmi:
  $(CAMLC) $<
```

Dépendances explicites

```
main: $(ML_OBJ)
  ocamlc -o $@ $(ML_OBJ)
```

Automatisation, `ocamldep`

Fichiers sources (`.ml`, `.mli`) → règles pour `make`.

```
.depend:
  ocamldep $(ML_SRC) > $@
include .depend
```

Autres outils

Production de `Makefile`

`configure`, `automake` fabriquent des fichiers `Makefile` à partir de fichiers `Makefile.in`.

langage de macros M4 pour générer avec un script de configuration `configure`.

Distributions binaires

Formats de fichiers (`rpm`, ...) permettant de gérer des dépendances entre paquetages formés de listes de fichiers.

Détection des cycles, dépendances, automatisation (`yum`).

Cohérence déléguée au fabricant du paquet, pas de vérification statique.

Ingéniérie logicielle

Activité artisanale, développement collaboratif

Forte valeur ajoutée : 200 Milliards de dollars par an.

Gérer les ressources matérielles, logicielles et humaines.

Fabriquer des programmes, 30% coût en développement

Distribuer les tâches entre les programmeurs.

Adapter la structure du logiciel.

Langages adaptés (ML, C, HTML, shell, XML, Java, C++)

Gestion du cycle de vie, 70% coût en maintenance

Détection rapides des défaillances (tests).

Évolutions du logiciel (architecture, système, langages).

Durée de vie des sources, évolution des développeurs.