

# Cours de Compilation 2008/2009

M1 informatique – Université Paris Diderot



# Chapitre 1

## Introduction

### 1.1 Détails pratiques

#### Emploi du temps

- cours : mercredi de 12h30 à 14h30 en Amphi 4C (normalement)
- TD/TP : début la semaine prochaine (en salle de TP)
- Chargés de TD/TP : Jaap Boender, Mehdi Dogguy, Grégoire Henry

#### Contrôle des connaissances

- Gros projet obligatoire par 3.
  - **Note individuelle.**
  - 1re partie à rendre en novembre
  - 2e partie et soutenance : début janvier 2010
- Examen final : janvier 2010
- Note Janvier :  $\frac{1}{2}$  note projet +  $\frac{1}{2}$  exam Janvier
- Session de rattrapage :  $\frac{1}{2}$  note projet +  $\frac{1}{2}$  exam Juin

#### Checklist

- inscrivez-vous tout de suite sur la mailing list :  
<https://sympa.mancoosi.univ-paris-diderot.fr/wws/info/m1-compilation>
- marquez la page web du cours dans vos signets :  
<http://www.pps.jussieu.fr/~balat/compilation>
- commencez à réviser OCaml
- Formez vite vos groupes pour le projet (maximum 3 personnes)

## 1.2 Généralités

### 1.2.1 Introduction : dictionnaire

**Généralités : le terme « compilateur »**

**compilateur** (Le Petit Robert)

« Personne qui réunit des documents dispersés »

**compilation** (Le Petit Robert)

« Rassemblement de documents »

Mais le programme qui « réunit des bouts de code dispersés » s'appelle aujourd'hui un *Éditeur de Liens* (comm nde ld)

**Le terme « compilateur »**

On appelle « compilateur » une autre chose :

**com·pil·er** (Webster)

1 : one that compiles

2 : *a computer program that translates an entire set of instructions written in a higher-level symbolic language (as COBOL) into machine language before the instructions can be executed*

### 1.2.2 Le cours

Ce cours

« De l'AST à l'exécutable »

Suite du cours *Analyse et compilation* de L3

Support de cours :

d'après Roberto Di Cosmo (cours 2005/2006)

**Plan (provisoire) du cours**

1. Notions préliminaires : structure d'un compilateur (front-end, back-end, cœur)
2. Description de la machine cible (MIPS R2000), Assembleur
3. Analyse lexicale et syntaxique : bref rappel sur Lex, OCamlLex, Yacc, OCamlYacc  
Arbre de syntaxe abstraite : structure et représentation
4. Compilation des langages à blocs : bloc d'activation, lien statique
5. Interface entre front-end et cœur : le code intermédiaire (génération, optimisation, linéarisation)
6. Génération du code assembleur
7. Allocation des registres par coloriage de graphe
8. Extensions :
  - typage des types rékursifs ;
  - compilation des langages à objets ;

- compilation des langages fonctionnels ;
- machines virtuelles ;
- algèbre des T pour la génération et le bootstrap des compilateurs.

## Bibliographie

- **Compilers : Principles, Techniques and Tools** (dragon book). *Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman*, Addison-Wesley. Version française à la bibliothèque.
- **Modern Compiler Implementation in ML**. *Andrew Appel*, Cambridge University Press  
<http://www.cs.princeton.edu/~appel/modern/ml/>
- **Développement d'applications avec Objective Caml** *Emmanuel Chailloux, Pascal Manoury, Bruno Pagano, O'Reilly*.  
Bibliothèque et en ligne (<http://www.pps.jussieu.fr/Livres/ora/DA-OCAML/index.html>).
- Manuel OC ml en ligne : <http://caml.inria.fr/pub/docs/manual-ocaml/index.html>
- Mars, un simulateur RISC 2000 <http://courses.missouristate.edu/KenVollmar/MARS/>

## Le projet

Compilateur CTigre vers MIPS  
Implémentation en OCaml  
Exécution avec l'émulateur MARS

### 1.2.3 Définitions

#### Notions (abstraites) de base

**machine hardware** : le processeur (...)

**machine virtuelle** : une machine qui reconnaît un certain nombre d'instructions qui ne sont pas (toutes) « natives » pour la machine hardware. Exemples :

- Les machines virtuelles Java, OCaml, .net Compilation vers du *bytecode* (code-octet) indépendant de la machine hardware
- Le compilateur C produit du code assembleur qui fait appel à un ensemble de fonctions système (ex. : les E/S). Donc il produit du code pour la machine virtuelle définie par : *instructions hardware + appels système*.

#### Qu'est-ce qu'un *interpréteur* (ou *interprète*) ?

**Définition 1** (Interpréteur). Un programme qui prends en entrée un autre programme, écrit pour une machine virtuelle, et l'exécute *sans le traduire*.

Ex : les interpréteurs des langages VisualBasic, PERL, Python, Bash, PHP, etc.

Le toplevel OCaml n'est pas un interpréteur (il compile vers le bytecode puis la machine virtuelle interprète le bytecode).

## Qu'est-ce qu'un *compilateur* ?

**Definition 2** (Compilateur). Un *programme*, qui *traduit* un programme écrit dans un langage L dans un programme écrit dans un langage L' différent de L (en général L est un langage évolué et L' est un langage moins *expressif*).

Quelques exemples :

- De C vers l'assembleur x86 plus les appels système Unix/Linux
  - De C vers l'assembleur x86 plus les appels système Win32
  - De  $\text{\LaTeX}$  vers Postscript ou HTML (latex2html/Hevea)
  - Compilation des expressions régulières (utilisées dans le shell Unix, les outils sed, awk, l'éditeur Emacs et les bibliothèques des langages C, Perl et OCaml)
  - Minimisation des automates (compilation d'un automate vers un automate plus efficace)
- On peut « compiler » vers une machine... *interprétée* (ex : bytecode Java, bytecode OCaml)

## La décompilation

On va dans le sens inverse, d'un langage moins structuré vers un langage évolué. Ex : retrouver le source C à partir d'un code compilé (d'un programme C).

Applications :

- récupération de vieux logiciels ;
- découverte d'API cachées ;
- ...

Elle est autorisée en Europe à des fins d'interopérabilité

## Notions (abstraites) de base, II

Il ne faut pas tricher... un compilateur ne doit pas faire de l'interprétation...

Pourtant, dans certains cas, il est inévitable de retrouver du code interprété dans le code compilé

- `printf("I v ut %d\n",i);` serait à la limite compilable ;
- `s="I v ut %d\n"; printf(s,i);` est plus dur ;
- alors que

```
void error(ch_r *s)
{ printf(s,i); }
```

devient infaisable.

## 1.2.4 Questions

### Un problème actuel

- on crée des nouveaux langages (essor des langages dédiés (DSL *Domain Specific Languages*)), ils ont besoin de compilateurs ;
- les anciens langages évoluent (C, C++, Fortran) ;
- les machines changent (architectures superscalaires, etc.) ;

- des concepts difficiles deviennent mieux compris, et implémentables (polymorphisme, modularité, polytypisme, etc.) ;
- les préoccupations changent (certification (travail de Andrew Appel), sécurité).

### (Presque) toute l'informatique dans un compilateur

algorithmique	parcours et coloration de graphes (allocation des registres)
	programmation dynamique (sélection optimale)
	stratégie gloutonne (MaximalMunch)
théorie	automates finis (lexeur), à pile (parseur)
	treillis, point fixe (liveness et dataflow analysis)
	réécriture (transformation de code)
intelligence artificielle	algorithmes adaptatifs
	recuit simulé
architecture	pipelining, scheduling
système	

### Devenir un bon programmeur

L'étude du fonctionnement des compilateurs permet :

- de comprendre les concepts des langages de programmation (et ne plus juger la qualité d'un langage sur le nombre de parenthèses des programmes mais avec des critères objectifs) ;
- d'éviter des erreurs de programmation et optimiser vos programmes (exemple : fonctions « *tail recursive* », etc.).

### Comparaison multicritères

Qu'est-ce qui est important dans un compilateur ?

- Le code produit est rapide.
- Le compilateur est rapide.
- Les messages d'erreurs sont précis.
- Le code produit est correct.
- Le code produit est *certifié* correct.
- Il supporte un débogueur.
- Il supporte la compilation séparée.
- Il sait faire des optimisations poussées.

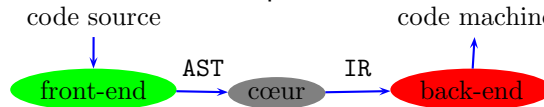
## 1.3 Survol : du source à l'exécutable

### 1.3.1 Les différentes phases de la compilation

#### Structure logique d'un compilateur

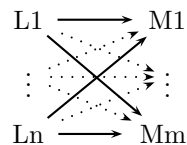
Un compilateur est un logiciel **très complexe** :

- on essaye de réutiliser au maximum ses composantes, donc on identifie :

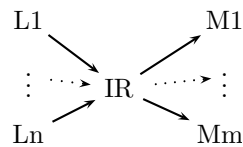


- un « front-end » lié au langage source ;
- un « back-end » lié à la machine cible ;
- un « code intermédiaire » commun IR sur lequel travaille le cœur du système.

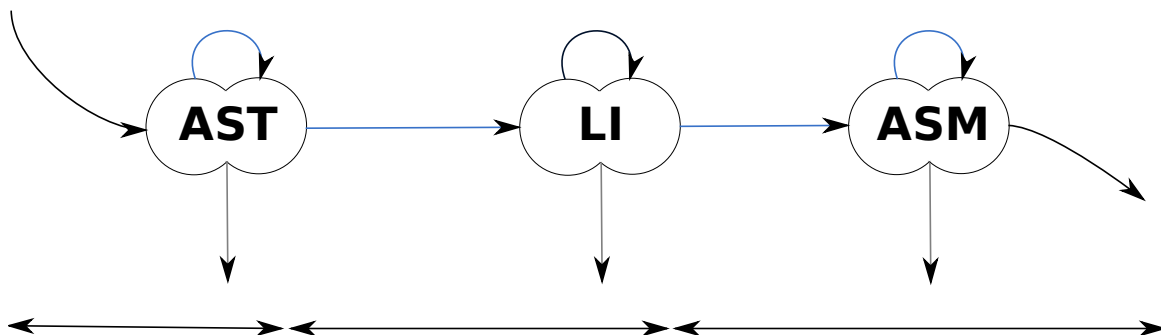
Cela permet d'écrire les  $nm$  compilateurs de  $n$  langages source à  $m$  machines cible



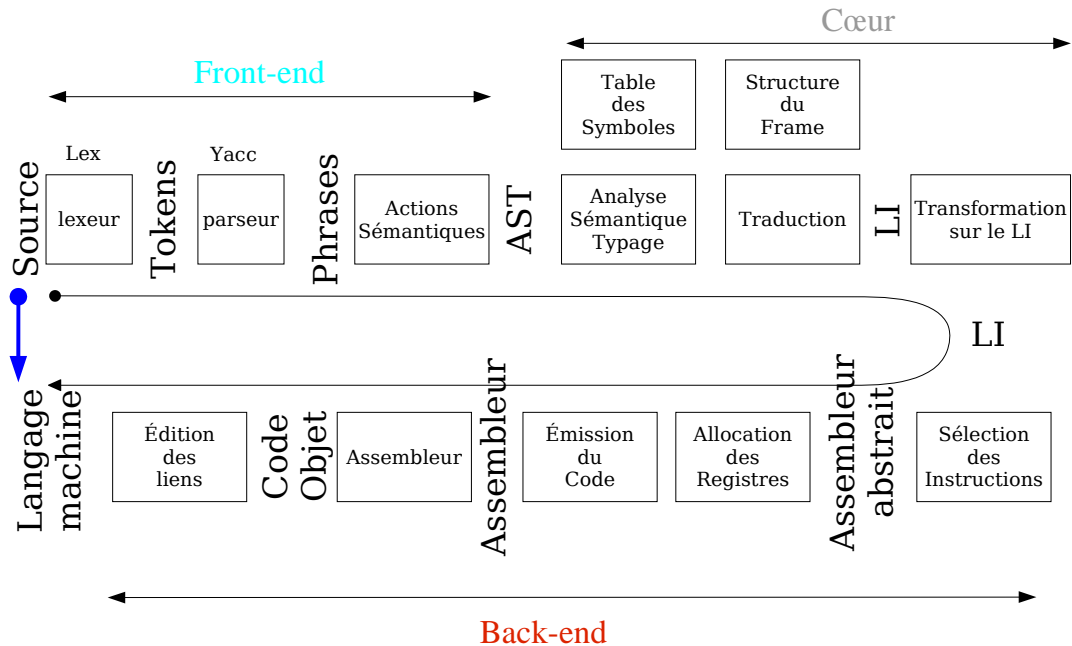
en écrivant seulement un cœur,  $n$  front-ends et  $m$  back-ends



#### Structure d'un compilateur moderne







### Structure détaillée d'un compilateur : Front-end

1. **Analyse lexicale** (flot de lexèmes)  
 théorie : langages rationnels  
 outils : automates finis  
 logiciels : Lex (et similaires)
2. **Analyse syntaxique** (flot de réductions)  
 théorie : langages algébriques  
 outils : automates à pile  
 logiciels : Yacc (et similaires)
3. **Actions sémantiques**  
 (construction de l'arbre de syntaxe abstrait, AST)  
 outils : grammaires attribuées  
 logiciels : encore Yacc

### Structure détaillée d'un compilateur : Cœur

1. **Analyse sémantique** (vérification des types, portée des variables, tables des symboles, gestion des environnements etc.) renvoie l'AST décoré et les tables des symboles  
 outils : grammaires attribuées, ou à la main
2. Traduction en **code intermédiaire** (souvent un arbre, indépendant de l'architecture cible)

3. [Linéarisation](#) du code intermédiaire (transformation en *liste* d'instructions du code intermédiaire)
4. Différentes [optimisations](#)
  - analyse de vie des variables et allocation des registres ;
  - transformation des boucles (déplacement des invariants, transformations affines) ;
  - fonction inlining, dépliement des boucles, etc.

### Structure détaillée d'un compilateur : Back-end

1. [Sélection d'instructions](#) (passage du code intermédiaire à l'assembleur de la machine cible, éventuellement abstrait par rapport aux noms des registres).
2. [Émission du code](#) (production d'un fichier écrit dans l'assembleur de la machine cible).
3. [Assemblage](#) (production *des* fichiers contenant le code machine).
4. [Édition des liens](#) (production *du* fichier exécutable).

Ces quatre dernières phases *seulement* dépendent de la machine assembleur cible.

### 1.3.2 Exemple : Compilation d'un programme C

#### Les phases cachées d'un compilateur : l'apparence

```
duodiscus> c t ex mple.c
```

```
#include <stdlib.h>
#include <stdio.h>
m in () {
    int i=0;
    int j=0;
    for (i=0;i<10;i++) {
        j=6*i; };
    printf("Result t : "); printf("%d", j);
    exit(0);
}
```

```
duodiscus> gcc -o ex mple ex mple.c
duodiscus> ls -l ex mple*
-rwxr-xr-x  1 toto    50784 Oct  2 15:43 ex mple*
-rw-r--r--  1 toto    139 Oct  2 15:42 ex mple.c
```

## Les phases cachées d'un compilateur : la réalité

```
duodiscus> gcc -v -o ex mple --save-temps ex mple.c
Re ding specs from /usr/lib/gcc-lib/i386-linux/2.95.4/specs
gcc version 2.95.4 20011002 (Debi n prerele se)
/usr/lib/gcc-lib/i386-linux/2.95.4/cpp0
-l ng-c -v -D__GNUC__=2 -D__GNUC_MINOR__=95 -D__ELF__ -Dunix
-D__i386__ -Dlinux -D__ELF__ -D__unix__ -D__i386__
-D__linux__ -D__unix -D__linux -Asystem(posix)
-Acpu(i386) -Am chine(i386) -Di386 -D__i386 -D__i386__
ex mple.c ex mple.i
GNU CPP version 2.95.4 20011002 (Debi n prerele se) (i386 Linux/ELF)

/usr/lib/gcc-lib/i386-linux/2.95.4/cc1 ex mple.i
-quiet -dumpb se ex mple.c -version -o ex mple.s
GNU C version 2.95.4 20011002 (Debi n prerele se) (i386-linux) compiled by GNU C version 2.95.4

s -V -Qy -o ex mple.o ex mple.s
GNU ssembler version 2.12.90.0.1 (i386-linux) using BFD version 2.12.90.0.1 20020307 Debi n/GNU

/usr/lib/gcc-lib/i386-linux/2.95.4/collect2
-m elf_i386 -dyn mic-linker /lib/ld-linux.so.2
-o ex mple /usr/lib/crt1.o /usr/lib/crti.o
/usr/lib/gcc-lib/i386-linux/2.95.4/crtbegin.o
-L/usr/lib/gcc-lib/i386-linux/2.95.4 ex mple.o
-lgcc -lc -lgcc
/usr/lib/gcc-lib/i386-linux/2.95.4/crtend.o /usr/lib/crtn.o

duodiscus> ls - l ex mple*
-rwxrwxr-x 1 toto 4764 Sep 19 18:29 ex mple
-rw-rw-r-- 1 toto 136 Sep 19 18:28 ex mple.c
-rw-rw-r-- 1 toto 21353 Sep 19 18:29 ex mple.i
-rw-rw-r-- 1 toto 1028 Sep 19 18:29 ex mple.o
-rw-rw-r-- 1 toto 877 Sep 19 18:29 ex mple.s
```

### 4 étapes :

- *préprocesseur* : cpp0 traduit example.c en example.i
- *compilateur* : cc1 traduit example.i en example.s
- *assembleur* : s traduit example.s en example.o
- *éditeur de liens* : collect2 (un enrobage de ld) transforme ex mple.o en exécutable ex mple, en résolvant les liens externes

## Un exemple simple

fichier ex mple.c

```
#include <stdlib.h>
#include <stdio.h>
main () {
    int i=0;
    int j=0;
    for (i=0;i<10;i++) {
        j=6*i; };
    printf("Result t : "); printf("%d", j);
    exit(0);
}
```

Compilation :

```
duodiscus> gcc -o ex mple -v -s ve-temps ex mple.c
```

## Un exemple simple : preprocesseur

fichier ex mple.i

```
#1 "ex mple.c"
#1 "/usr/include/stdio.h" 1
...
extern int printf ( const ch r * form t , ... ) ;
...
#1 "ex mple.c" 2
main ( ) {
    int i = 0 ;
    int j = 0 ;
    for ( i = 0 ; i < 10 ; i ++ ) {
        j = 6 * i ;
    } ;
    printf ( "Result t: %d" , j ) ;
    exit ( 0 ) ;
}
```

## Un exemple simple : compilation vers assembleur

fichier ex mple.s

```
.file 1 "ex mple.c"
.rd t
.lign 2
$LCO:
. scii "Result t: "
.text
.lign 2
```

```

.globl  m in
.ent    m in

                                # prologue de l fonction
m in:
    subu $sp,$sp,48 # lloc tion v ri bles sur l pile
    sw   $31,40($sp) # s uve dresse de retour
    sw   $fp,36($sp) # s uve vieux fr me pointer
    sw   $28,32($sp) # s uve gp
    move $fp,$sp     # ch nge fr me pointer
    sw   $0,24($fp)  # initi lise v ri ble i
    sw   $0,28($fp)  # initi lise v ri ble j
    sw   $0,24($fp)  # compilo p s très futé

$L3:
    lw   $2,24($fp)  # ch rge i d ns $2
    slt  $3,$2,10    # |
    bne  $3,$0,$L6   # si i<10, v L6
    j    $L4
$L6:
    lw   $2,24($fp)  # ch rge i d ns $2 (encore !)
    move $4,$2       # i d ns $4
    sll  $3,$4,1     # $3 = $4*2
    ddu  $3,$3,$2    # $3 = $3+$2 = 3* $2
    sll  $2,$3,1     # $2 = $3 *2 = 6* $2
    sw   $2,28($fp)  # j = $2 = 6*i

$L5:
    lw   $2,24($fp)  # $2 = i (encore!!!)
    ddu  $3,$2,1     # $3 = i+1
    sw   $3,24($fp)  # i= $3 = i+1
    j    $L3         # on reboucle L3
$L4:
                                # on fini!!!!!!
    l    $ 0,$LC0
    li   $v0,4        # print_string
    sysc ll
    lw   $ 0,28($fp)
    li   $v0,1        # print_int
    sysc ll

$L2:
    move $sp,$fp      # épilogue:
    lw   $31,40($sp)  # on rest ure sp, fp, gp, r
    lw   $fp,36($sp)
    ddu  $sp,$sp,48
    j    $31          # on revient d'où l'on nous ppelé
    .end  m in

```



**.o** fichiers objets

**.a** bibliothèques (libraries) statiques (exemple : `libc.a`, `libm.a`, `libjpeg.a` )

**.so** bibliothèques (libraries) dynamiques (exemple : `libc.so`, `libm.so`, `libjpeg.so`)

Exemple de lignes de compilation :

```
gcc -c ex_mple.c
gcc ex_mple.o -L/home/b l t/lib -ljpeg -ltoto -o ex_mple
gcc ex_mple.o -static -L/home/b l t/lib -ljpeg -ltoto -o ex_mple
```

Exemple d'édition de liens :

```
ld -o ex_mple /usr/lib/crt1.o /usr/lib/crti.o /usr/lib/libc.o ex_mple.o
```

## Librairies statiques et dynamiques

Pour spécifier le répertoire des bibliothèques dynamiques :

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/b l t/lib
```

Sous Linux : fichier `/etc/ld.so.conf` puis lancer `ldconfig`

Création d'une bibliothèque statique :

```
ar -r libtoto.a titi.o t t .o
```

Création d'une bibliothèque dynamique :

```
gcc -o libtoto.so -shared titi.o t t .o
```

## Quelques commandes

**nm** Table des symboles du fichier objet

**file** Indique le type du fichier (magic number)

**size** Taille des différentes sections du fichier

**strip** Supprime la table des symboles

**ldd** Affiche les bibliothèques partagées utilisées par l'exécutable

## Fichiers objets (.o)

1. Identification
  - nom du module
  - Longueur des différentes parties
  - Date d'assemblage
2. Table des points d'entrée
  - Liste des symboles déclarés dans ce module, avec leur adresse relative
3. Table des références externes
  - Liste des symboles utilisés dans ce module et déclarés ailleurs (avec la liste des instructions qui les utilisent)
4. Code assemblé
5. Dictionnaire des translations pour les adresses absolues (ajout d'une constante à faire au chargement)
6. Indication de fin de module (avec l'adresse de début d'exécution)

## Édition des liens

### 1re passe

- Construction d'une table globale à tous les modules
- Construction de la table des symboles globale

### 2e passe

- Résolution des adresses et translation

## L'exécutable

L'exécutable contient :

- Le code (après édition des liens)
- Le dictionnaire des translations
- L'adresse de début d'exécution
- L'identification

La table des symboles n'est pas utile à l'exécution (mais utile par exemple pour le débogueur)

## Chargement en mémoire

Seul le code est chargé en mémoire, après translation.

Le code est composé de trois *segments* :

`.text` instructions

`.data` données initialisées

`.bss` (Block Storage Segment) données non initialisées

Le segment `.text` est protégé en écriture à l'exécution, et une seule copie pour plusieurs exécutions simultanées