

## Cours de Compilation: 2004/2005

Maîtrise d'Informatique  
Paris VII

Roberto Di Cosmo  
e-mail: roberto@dicosmo.org  
WWW: <http://www.dicosmo.org>

### Modalités du cours

- ▶ Nb cours :  
12 (vendredi de 14h30 à 16h30 en Amphi 34A)
- ▶ Nb TD : 12 (début la semaine du 11 Octobre)
- ▶ Chargés de TD :  
Vincent Balat, Juliuz Chroboczek, Alexandre Miquel  
Lundi 16h30-18h30 (J6), Mardi 12h30-14h30 (J4 et J7), Jeudi 16h30-18h30 (J8)
- ▶ Partiel projet : mi-décembre
- ▶ Soutenance projet : fin décembre
- ▶ Examen final : entre le lundi 17 janvier 2005 et le samedi 5 février 2005
- ▶ Note Janvier :  
 $\frac{1}{2}$  note projet +  $\frac{1}{2}$  exam Janvier
- ▶ Note Septembre :  
 $\frac{1}{2}$  note projet +  $\frac{1}{2}$  exam Septembre

### Checklist

- ▶ inscrivez-vous tout de suite sur la mailing list:  
<http://ufr.pps.jussieu.fr/www/info/ml-0405-compilation>
- ▶ marquez la page web du cours dans vos signets:  
<http://www.dicosmo.org/CourseNotes/Compilation/>
- ▶ commencez à réviser OCaml
- ▶ réfléchissez à la composition des groupes pour le projet (maximum 3 personnes)

### Plan du cours

1. Notions préliminaires : structure d'un compilateur (front-end, back-end, coeur), description de la machine cible assembleur (RISC 2000)
2. Mise à niveau Ocaml faite exclusivement en TP (langage disponible librement par ftp depuis l'Inria <ftp.inria.fr>)
3. Analyse lexicale et syntaxique :  
bref rappel sur Lex, Ocamllex, Yacc, Ocaml yacc
4. Arbre de syntaxe abstraite :  
structure et représentation
5. Analyse statique : typage (rappels)
6. Structure de la machine d'exécution pour un langage a blocs
7. Pile des blocs d'activation pour fonctions/variables locales
8. Interface entre front-end et coeur: le code intermédiaire (génération, optimisation, linéarisation)
9. Génération du code assembleur
10. Allocation des registres
11. Extensions:
  - ▶ typage des types recursifs
  - ▶ allocation de registres par coloriage de graphes
  - ▶ 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.**  
*Alfred V. AHO, Ravi SETHI, Jeffrey D. ULLMAN*, Addison-Wesley. Version française en bibliothèque.
- ▶ **Modern Compiler Implementation in ML.**  
*Andrew APPEL*, CAMBRIDGE UNIVERSITY PRESS  
<http://www.cs.princeton.edu/~appel/modern/ml/>  
Une vingtaine de copies à la bibliothèque.

- ▶ **Développement d'applications avec Objective Caml**  
Emmanuel CHAILLOUX, Pascal MANOURY, Bruno PAGANO, O' Reilly. Bibliothèque et en ligne<sup>1</sup>.
- ▶ Manuel Ocaml en ligne: <http://caml.inria.fr/ocaml/htmlman/index.html>
- ▶ SPIM, un simulateur RISC 2000 <http://www.cs.wisc.edu/~larus/spim.html>

### Généralités: le terme "compilateur"

---

**compilateur** : "Personne qui reunit des documents dispersés" (Le Petit Robert)

**compilation** : "Rassemblement de documents" (Le Petit Robert)

Mais le programme qui "reunit des bouts de code dispersés" s'appelle aujourd'hui un *Éditeur de Liens*<sup>2</sup>

### 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<sup>3</sup>) into machine language before the instructions can be executed

Qu'est-ce que le "langage machine"?

### Notions (abstraites) de base

---

**machine hardware** le processeur

**machines virtuelle** une machine qui reconnaît un certain nombre d'instructions qui ne sont pas toutes "natives" pour la machine hardware.

Ex: 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 les instructions hardware, plus les appels système.

### On rencontre souvent les machines virtuelles suivantes

---

<sup>1</sup><http://www.pps.jussieu.fr/Livres/ora/DA-OCAML/index.html>

<sup>2</sup>ld

<sup>3</sup>;-)

- ▶ Programmes d'application
- ▶ Langage de programmation évolué
- ▶ Langage d'assemblage
- ▶ Noyau du système d'exploitation
- ▶ Langage machine

### Qu'est-ce qu'un interpréteur?

---

Un programme qui prends en entrée un autre programme, écrit pour une quelque machine virtuelle, et l'exécute *sans le traduire*.

Ex:

- ▶ l'interpréteur VisualBasic,
- ▶ le toplevel Ocaml,
- ▶ l'interpréteur PERL,
- ▶ l'interpréteur OCaml,
- ▶ etc.

### Qu'est-ce qu'un 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:

- 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)
- De LaTeX vers Postscript ou HTML (latex2html/Hevea)
- De C vers l'assembleur x86 plus les appels de système Unix/Linux
- De C vers l'assembleur x86 plus les appels de système Win32

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

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

  
devient infaisable

## Pause questions ...

Pourquoi étudier des compilateurs?

Pourquoi construire des compilateurs?

Pourquoi venir au cours?

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

algorithmique	parcours et coloration de graphes <sup>4</sup> programmation dynamique <sup>5</sup> stratégie gloutonne <sup>6</sup>
théorie	automates finis <sup>7</sup> , à pile <sup>8</sup> treillis, point fixe <sup>9</sup> réécriture <sup>10</sup>
intelligence artificielle	algorithmes adaptatifs recuit simulé
architecture	pipelining, scheduling

## Un problème actuel

- ▶ on crée des nouveaux langages (essor des DSL<sup>11</sup>), 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<sup>12</sup>, sécurité)

## Comparaison multicritères

Qu'est-ce qui est important<sup>13</sup> 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

<sup>11</sup>domain specific languages

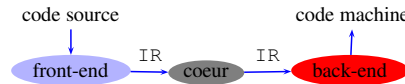
<sup>12</sup>Travail de Andrew Appel

<sup>13</sup>pour vous

## Structure logique d'un compilateur

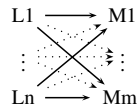
Un compilateur est un logiciel très complexe:

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

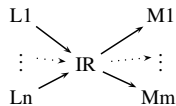


- un "front-end" lié au langage source
- un "back-end" lié à la machine cible
- un "code intermédiaire" commun IR sur lequel travaille le coeur du système

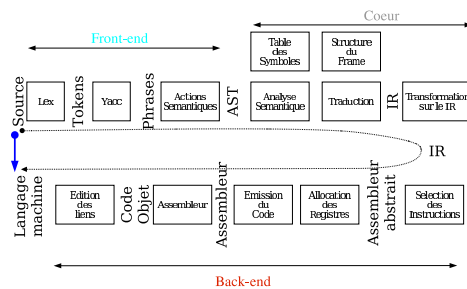
Cela permet d'écrire les  $nm$  compilateurs de  $n$  langages source à  $m$  machines cible



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



## Structure d'un compilateur moderne



## Structure détaillée d'un compilateur

Front-end:

- ▶ **analyse lexicale** (flot de lexemes)  
théorie: langages réguliers  
outils: automates finis  
logiciels: Lex (et similaires)
- ▶ **analyse syntaxique** (flot de reductions)  
théorie: langages algébriques  
outils: automates à pile  
logiciels: Yacc (et similaires)
- ▶ **actions sémantiques**  
(construction de l'arbre de syntaxe abstrait, AST)  
outils: grammaires attribuées  
logiciels: encore Yacc
- ▶ **analyse sémantique**  
(vérification des types, portée des variables, tables des symboles, gestion des environnements etc.) rends l'AST décoré et les tables des symboles  
outils: grammaires attribuées, ou à la main
- ▶ traduction en **code intermédiaire**  
(souvent un arbre, indépendant de l'architecture cible)

Coeur:

- ▶ **linéarisation** du code intermédiaire  
(transformation en *liste* d'instructions du code intermédiaire)
- ▶ 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.

Back-end:

- ▶ **sélection d'instructions** (passage du code intermédiaire à l'assembleur de la machine cible, éventuellement abstrait par rapport aux noms des registres)
- ▶ **émission du code** (production d'un fichier écrit dans l'assembleur de la machine cible)
- ▶ **assemblage**  
(production *des* fichiers contenant le code machine)

- [édition des liens](#) (production *du* fichier executable)

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

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

```
ranger> cat simplaffine.c
#include <stdio.h>
main ()
{int i=0;
  int j=0;
  for (i=0;i<10;i++)
    {j=6*i;};
  printf("Resultat: %d", j);
  exit(0);}

ranger> gcc -o simplaffine simplaffine.c
ranger> ls -l simplaffine*
-rwxr-xr-x 1 dicosmo 50784 Oct 2 15:43 simplaffine*
-rw-r--r-- 1 dicosmo 139 Oct 2 15:42 simplaffine.c
```

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

```
ranger>gcc -v -o simplaffine --save-temps simplaffine.c
Reading specs from /usr/lib/gcc-lib/i386-linux/2.95.4/specs
gcc version 2.95.4 20011002 (Debian prerelease)
/usr/lib/gcc-lib/i386-linux/2.95.4/cpp0
  -lang-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) -Amachine(i386) -Di386 -D__i386 -D__i386__
  simplaffine.c simplaffine.i
GNU CPP version 2.95.4 20011002 (Debian prerelease) (i386 Linux/ELF)
/usr/lib/gcc-lib/i386-linux/2.95.4/ccl simplaffine.i
  -quiet -dumpbase simplaffine.c -version -o simplaffine.s
GNU C version 2.95.4 20011002 (Debian prerelease) (i386-linux) compiled by
as -V -Qy -o simplaffine.o simplaffine.s
GNU assembler version 2.12.90.0.1 (i386-linux) using BFD version 2.12.90.0.
/usr/lib/gcc-lib/i386-linux/2.95.4/collect2
  -m elf_i386 -dynamic-linker /lib/ld-linux.so.2
  -o simplaffine /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 simplaffine.o
  -lgcc -lc -lgcc
```

```
/usr/lib/gcc-lib/i386-linux/2.95.4/crtend.o /usr/lib/crti.o
```

```
ranger> ls -al simplaffine*
-rwxrwxr-x 1 dicosmo 4764 Sep 19 18:29 simplaffine
-rw-rw-r-- 1 dicosmo 136 Sep 19 18:28 simplaffine.c
-rw-rw-r-- 1 dicosmo 21353 Sep 19 18:29 simplaffine.i
-rw-rw-r-- 1 dicosmo 1028 Sep 19 18:29 simplaffine.o
-rw-rw-r-- 1 dicosmo 877 Sep 19 18:29 simplaffine.s
```

#### 4 étapes:

- *preprocesseur*: cpp0 traduit simplaffine.c en simplaffine.i
- *compilateur*: cc1 traduit simplaffine.i en simplaffine.s
- *assembleur*: as traduit simplaffine.s en simplaffine.o
- *éditeur de liens*: collect2<sup>14</sup> transforme simplaffine.o en executable simplaffine, en résolvant les liens externes

### Un exemple simple

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

#### Produit:

```
ranger> gcc -o simplaffine -v -save-temps simplaffine.c
```

### Un exemple simple: preprocesseur

```
#1 "simpleaffine.c"
#1 "/usr/include/stdio.h" 1
...
extern int printf ( const char * format , ... ) ;
...
#1 "simpleaffine.c" 2
main ( )
{ int i = 0 ;
  int j = 0 ;
```

<sup>14</sup>un enrobage de ld

```

for ( i = 0 ; i < 10 ; i ++ )
{ j = 6 * i ; } ;
printf ( "Resultat: %d" , j ) ;
exit ( 0 ) ; }

```

### Un exemple simple: compilation vers assembleur

```

.file 1 "example.c"
.rdata
.align 2
$LC0:
.ascii "Resultat: "
.text
.align 2
.globl main
.ent main

main:
# prologue de la fonction
subu $sp,$sp,48 # allocation variables sur la pile
sw $31,40($sp) # sauve adresse de retour
sw $fp,36($sp) # sauve vieux frame pointer
sw $28,32($sp) # sauve gp
move $fp,$sp # change frame pointer
sw $0,24($fp) # initialise variable i
sw $0,28($fp) # initialise variable j
sw $0,24($fp) # compilo bete

$L3:
lw $2,24($fp) # charge i dans $2
slt $3,$2,10 # |
bne $3,$0,$L6 # si i<10, va a L6
j $L4

$L6:
lw $2,24($fp) #charge i dans $2 (encore!)
move $4,$2 #i dans $4
sll $3,$4,1 # $3 = $4*2
addu $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!!!)
addu $3,$2,1 # $3 = i+1
sw $3,24($fp) #i= $3 = i+1
j $L3 #on reboucle a L3

```

```

$L4:
la $a0,$LC0 # on a fini!!!!!!
li $v0,4 # print_string
syscall
lw $a0,28($fp)
li $v0,1 # print_int
syscall

$L2:
move $sp,$fp # epilogue:
# on restaure sp, fp, gp, ra
lw $fp,36($sp)
addu $sp,$sp,48
j $31 # on revient la ou on nous a appele
.end main

```

OUFFF!

### Un exemple simple: optimisations I

Le compilateur arrive a faire tenir tout dans des registres, et trouve une façon plus efficace pour multiplier par 6.

```

main:
subu $sp,$sp,32 # alloue place sur la pile
sw $31,28($sp) # sauve adresse de retour
sw $28,24($sp) # sauve gp
move $3,$0 # initialise $3 (i) a 0

$L23:
sll $2,$3,1 # ]
addu $2,$2,$3 # met $3 * 6 dans $5 (j)
# ]
sll $5,$2,1 # ]
addu $3,$3,1 # incremente $3
#]
slt $2,$3,10 #]
bne $2,$0,$L23 #si $3 < 10, reviens a $L23:
la $a0,$LC0 # on a fini!!!!!!
li $v0,4 # print_string
syscall
li $v0,1 # print_int
addu $a0,$5,$0 # imprime j
syscall

```

## Un exemple simple: optimisations II

Le compilateur decouvre que  $j$  est une fonction affine de  $i$ .

```
main: li      $2,9          # boucle de 9 a 0
$L23:
      addu   $2,$2,-1      # decremente $2
      bgez  $2,$L23        # si pas 0, va a $L23
      la    $a0,$LC0       # on a fini!
      li    $v0,4          # print_string
      syscall
      li    $v0,1          # print_int
      li    $a0,54         # 0x36
      syscall
```