

Cours de Compilation: 2003/2004

Maîtrise d'Informatique Paris VII

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

Modalités du cours

- Nb cours :
13 (vendredi de 12h30 à 14h30 en Amphi 34A)
- Nb TD : 13 (début cette semaine)
- Chargés de TD :
Antonio Bucciarelli, Alexandre Miquel
Lundi 8h30-10h30 et 10h30-12h30 en salle 110,
Lundi 16h30-18h30 en salle 106
- Partiel projet : mi-décembre
- Examen final : entre le 19/01/2004 et le 07/02/2004
- Soutenance projet : entre le 01/02/2004 et le 07/02/2004
- Note Projet :
 $\frac{1}{2}$ note partiel projet + $\frac{1}{2}$ note soutenance
- Note Janvier :
 $\frac{1}{3}$ note projet + $\frac{2}{3}$ exam Janvier
- Note Septembre :
Max(exam Septembre, $\frac{1}{3}$ note projet + $\frac{2}{3}$ exam Septembre)

Plan du cours

1. Notions préliminaires :
structure d'un compilateur, bootstrap, machine cible assembleur (RISC 2000)
2. Mise à niveau Ocaml (langage disponible librement par ftp depuis l'Inria <ftp.inria.fr>)
3. Analyse lexicale :
rappels théoriques, Lex, Ocamllex
4. Analyse syntaxique :
rappels théoriques, analyse descendente, analyse ascendente, Yacc, Ocaml yacc
5. Arbre de syntaxe abstraite :
structure et représentation
6. Analyse statique : typage
7. Blocs d'activation pour fonctions/variables locales
8. Structure de la machine d'exécution
9. Génération de code intermédiaire, optimisations
10. Génération de code assembleur
11. Allocation des registres

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 ligne¹.
- 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>

¹<http://www.pps.jussieu.fr/Livres/ora/DA-OCAML/index.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 *Editeur de Liens*²

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 COBOL³) 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

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

²ld

³:-)

Qu'est-ce qu'un *interpreteur*?

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

Ex:

- l'interpreteur VisualBasic,
- le toplevel Ocaml,
- l'interpreteur PERL,
- etc.

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

- Un *traducteur*, en général d'un langage évolué vers 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

Boostrapping, tuiles (I)

Procédure de bootstrap pur un compilateur de \mathcal{L} à \mathcal{M} :

- écriture en \mathcal{M} d'un mini-compilateur c_0 pour un langage $\mathcal{L}' \subset \mathcal{L}$
- écriture en \mathcal{L}' du compilateur optimisant $c_1.L$ pour \mathcal{L}
- compilation avec c_0 de $c_1.L$, on obtient c_1
- compilation de $c_1.L$ avec c_1 , on obtient $c_1.opt$

Boostrapping, tuiles (I)

Comment "porter" des compilateurs d'une machine à une autre?

Notes non rédigées, voir tableau

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 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 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, independant de l'architecture cible)

Coeur:

- **linéarisation** du code intermédiaire
(transformation en *liste* d'instructions du code intermediaire)
- 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 exécutable)

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

La machine assembleur

Il y a plusieurs microprocesseurs, classés en deux grandes classes

CISC : Intel

RISC : MIPS, SPARC, etc.

Pour ce cours, on utilisera comme machine cible le processeur MIPS 2000, un RISC pour lequel on dispose d'un excellent émulateur, le SPIM.

Les phases cachées d'un compilateur: l'apparence

```
ranger> cat callfact.c
#include <stdio.h>
int fact(int i)
{if (i==0) return(1);
 else {return(i*fact(i-1));};}
main()
{ printf ("%d",fact(3)); return(0);}

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

Les phases cachées d'un compilateur: la réalité

```
ranger>gcc -v -o callfact --save-temps callfact.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_MIN__
GNU CPP version 2.95.4 20011002 (Debian prerelease) (i386 Linux/ELF)
 /usr/lib/gcc-lib/i386-linux/2.95.4/cc1 callfact.i -quiet -dumpbase callfact.o
GNU C version 2.95.4 20011002 (Debian prerelease) (i386-linux) compiled by GNU
as -V -Qy -o callfact.o callfact.s
GNU assembler version 2.12.90.0.1 (i386-linux) using BFD version 2.12.90.0.1 2
 /usr/lib/gcc-lib/i386-linux/2.95.4/collect2 -m elf_i386 -dynamic-linker /lib/

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

4 étapes:

- *preprocesseur*: `cpp0` traduit `callfact.c` en `callfact.i`
- *compilateur*: `cc1` traduit `callfact.i` en `callfact.s`
- *assembleur*: `as` traduit `callfact.s` en `callfact-callfact.o`
- *éditeur de liens*: `collect2`⁴ transforme `callfact.o` en exécutable `callfact`, 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 callfact -v -save-temps simplaffine.c
```

⁴un enrobage de `ld`

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 ;
  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:
    lw      $31,40($sp)    # 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 à 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) à 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 à $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 découvre que j est une fonction affine de i .

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