

# Programmation Avancée IMA 3 – S6

## Cours 1 : Éléments de compilation

Laure Gonnord

<http://laure.gonnord.org/pro/teaching/>  
 Laure.Gonnord@polytech-lille.fr

Université Lille 1 - Polytech Lille

Mars 2011



La chaîne de compilation

### 1 La chaîne de compilation

- Le front-end
- In the middle
- Le back-end

### 2 Optimiser ?

- Le compilateur à l'action
- Et moi, si je veux optimiser ?

Laure Gonnord (Lille1/Polytech)

Programmation Avancée S6

Mars 2010

← 2 / 42 →

La chaîne de compilation

## Que fait un compilateur ? 1/2

### 1 La chaîne de compilation

- Le front-end
- In the middle
- Le back-end

### 2 Optimiser ?

Le **compilateur** transforme un langage source en un autre langage. Souvent (et ici), le deuxième langage est un langage assembleur (spécifique machine).

```
#include <stdio.h>
int main (int argc, char **argv) {
    printf("bonjour\n") ;
    return 0 ;
}
```

```
gcc -S hello.c
```

fournit hello.s (langage machine).

► La dernière phase fournit une suite d'octets.

## Que fait un compilateur ? 2/2

Voici une partie du code généré (cf cours Microprocesseurs).

```
[cutcut]
.LC0:
    .string      "bonjour"
    .text
    .globl main
    .type        main, @function
main:
    pushl       %ebp
    movl        %esp, %ebp
    andl        $-16, %esp
    subl        $16, %esp
    movl        $.LC0, (%esp)
    call        puts
    movl        $0, %eax
    leave
    ret
```

► **ebp** = pointeur de base (sert à adresser les vars locales et les paramètres), **esp** sommet de pile, **eax** valeur retournée.

## Étapes du début : front-end

La partie du compilateur souvent nommée **front-end** transforme le code source  $C$  en un **code intermédiaire**. Les différentes étapes sont les suivantes :

- Transformation du code source en un code sans macros (préprocesseur)
- Transformation du source sans macros en un arbre abstrait (deux étapes)
- Transformation de l'arbre en code intermédiaire

## Préprocesseur

```
gcc -E hello.c      ou      cpp hello.c
```

Réalise (entre autres) :

- L'expansion des **macros**
- Le remplacement des **en-têtes** par les **signatures** des fonctions qu'elles contiennent et des informations pour trouver leur code.

```
extern int printf (__const char *__restrict __format, ...);
```

## Source vers arbre abstrait 1/2

(Lexing) L'**Analyse Lexicale** : dans cette étape les **mots-clés** du langage sont retrouvés. Cette étape fournit une liste d'unités lexicales, ou **tokens**

```
int y = 12 + 4*x;
```

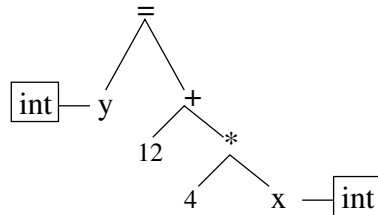
⇒ [TKINT, TKVAR("y"), TKEQ, TKINT(12), TKPLUS, TKINT(4), TKFOIS, TKVAR("x"), TKPVIRG]

## Source vers arbre abstrait 2/2

(Parsing) L'**Analyse Syntaxique** transforme cette liste de tokens en un **Arbre Syntaxique Abstrait** (AST)

[TKINT, TKVAR("y"), TKEQ, TKINT(12), TKPLUS, TKINT(4), TKFOIS, TKVAR("x"), TKPVIRG]

⇒



## Arbre abstrait vers code intermédiaire - 1

Une ou plusieurs phases :

- Sur l'arbre abstrait on peut faire plein de choses dont du typage et d'autres optimisations.
- On produit ensuite du code pour une **machine idéale**. (Avantage ?)
- Cette étape donne du sens (**sémantique**) au langage.
- ▶ Cette machine idéale a un **assembleur lisible et simple**, et un nombre infini de registres !

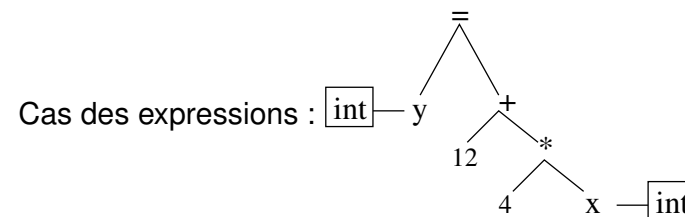
## Arbre abstrait vers code intermédiaire -2

Instructions de la machine idéale (abstraite)

- Affectation :  $x = y \text{ op } z$ ,  $x = \text{op } y$ , et  $x = y$
- Affectation avec pointeurs :  $x = [y + \text{constante}]$  et  $[x + \text{constante}] = y$
- Branchement inconditionnel : `jump label`
- Branchement conditionnel : `cjump(x oprel y, label)`
- Appel de fonction : `call(label, x1, ..., xn)`
- Prélude de fonction : `prelude(taille_local)`
- Postlude de fonction : `postlude(taille_local, taille_arg)`
- Label : `label:`

$x$ ,  $y$ ,  $z$  et  $i$  sont des registres, des constantes ou des emplacements mémoire. Les instructions peuvent être préfixées par un label.

## Arbre abstrait vers code intermédiaire - 3



⇒

```
LD R1 (@x)
LD R2 12
LD R3 4
MUL R4 R1 R3
ADD R5 R4 R2
ST R4 (@y)
```

▶ ( $@x$ ) et ( $@y$ ) sont des positions **relatives** sur la pile d'exécution dépendantes de :

- la taille mémoire prise (type/machine)
- la portée (locale, globale, paramètre) de ces variables.

## Arbre abstrait vers code intermédiaire - 4

## Génération de code abstrait pour le test

If expr then s1 else s2

(eval de expr -> stockage dans CR)

cjump(non R,lablfalse)

--code pour s1

jump lblend

lblfalse:

--code pour s2

lblend:

► Exo : faire pour le **while** !

## Arbre abstrait vers code intermédiaire - 5a

## Génération de code abstrait pour les fonctions

```
function dot(a, b: array[1..10] of byte): integer;
var result,i : byte;
begin
  result := 0;
  i := 1;
  repeat
    result := result + a[i]*b[i];
    i := i + 1;
  until i=11;
  produit_scalaire := result;
end;
```

## Arbre abstrait vers code intermédiaire - 5b

## Génération de code abstrait pour les fonctions

```
dot:
  prelude(2)
  [0(result)] = 0
  [0(i)] = 1
  label_repeat:
    r1 = 0(a)
    r2 = r1+[0(i)]
    r3 = [r2]           //a[i]
    r4 = 0(b)
    r5 = r4+[0(i)]
    r7 = [r5]           //b[i]
    r8 = r4*r7          //a[i]*b[i]
    [0(result)] = [0(result)] + r8 //result := result + a[i]*b[i]
    [0(i)] = [0(i)] + 1   //i := i + 1
    cjump([0(i)],,11,label_end_repeat)
    jump(label_repeat)
  label_end_repeat:
  result_register = [0(result)]
  postlude(2,10+10)
```

## Arbre abstrait vers code intermédiaire (conclusion)

La traduction vers le code intermédiaire

- Nécessite peu de connaissances de la machine
- Pour les fonctions, appel de call
- Plus de détails : cours IF, IMA 4 SC.

## Étapes intermédiaires

**En fait**, le code intermédiaire est une **représentation intermédiaire** (structure de donnée interne au compilateur au lieu du texte).

Sur le code intermédiaire on peut réaliser nombre d'analyses et d'optimisations.

- En pratique, exemples plus loin dans le cours.
  - Théorie et algorithmique, cours IF, IMA 4 SC.
- ▶ Le choix d'une **bonne représentation intermédiaire** (IR) est crucial
- ▶ Recherche très **active**.

## Étapes de la fin : back-end

La partie du compilateur souvent nommée **back-end** transforme le code source intermédiaire en un **code assembleur**. Les différentes étapes sont les suivantes :

- Sélection des instructions assembleur pour chaque instruction du code intermédiaire.
  - Adressage des données/constantes/variables.
  - Allocation des registres
  - (**éventuellement**) D'autres optimisations machine spécifiques
  - Génération de code machine.
- ▶ Ces étapes nécessitent une bonne connaissance de la machine, et notamment de la pile d'exécution.

## Sélection des instructions

But : transformer les instructions de la machine idéale en instructions de la machine cible (assembleur de telle machine).

- ▶ Il peut y avoir des **instructions particulières**.
- ▶ Nécessite de bien connaître la machine cible et ses **particularités**.

## Pile et Adressage - 1

Notre machine parfaite savait où sont les valeurs des paramètres, variables, registres, à chaque instant :

- Une machine réelle comprend une pile, un tas.
  - Le compilateur doit se servir de celle-ci pour réaliser l'adressage.
  - Le code généré doit utiliser des mécanismes de sauvegarde sur la pile d'exécution.
- ▶ Les adresses '@' de la représentation intermédiaire deviennent des **adresses relatives** par rapport à des pointeurs de pile (dont l'adresse de base est `ARP`)  $r2 = r1 + [@(i)]$  devient  $r2 = r1 + [ARP+1]$

## Pile et Adressage - 2

Définitions/Remarques :

- Chaque exécution du corps d'une fonction est une **activation** de cette fonction.
- Deux activations d'une même fonction peuvent être vivantes en même temps.
- La pile d'exécution garde trace des activations en cours.
- ▶ enregistrement d'activation ou **activation record** "AR".

dessins et texte, C. Alias (Ens Lyon) et M. Moy (Ensimag)

## Pile et Adressage - 3 - Activation record

Un enregistrement d'activation va garder en mémoire

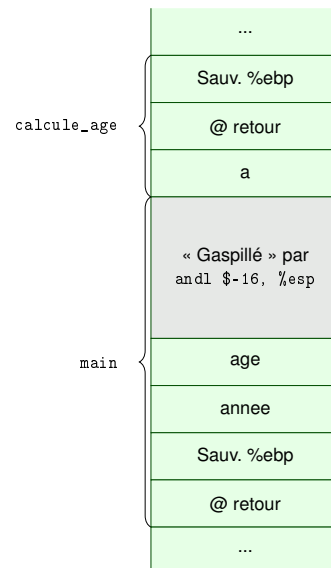
- les arguments de la fonction
- l'adresse de retour
- un lien vers l'enregistrement d'activation de l'appelante
- les variables locales
- (éventuellement un emplacement pour le résultat)
- ▶ Un registre spécialisé, ARP (**activation record pointer**), pointe vers l'enregistrement d'activation en cours
- ▶ La mise à jour des AR se fait avant l'appel, durant le prélude, en fin de fonction (postlude).
- ▶ ARP en Pentium est **%ebp**.

## Passage de paramètres : exemple – 1

```

unsigned calcule_age(unsigned a) {
    return 2010 - a;
}

int main(void) {
    unsigned annee, age;
    printf("Annee de naissance?");
    scanf("%u", &annee);
    age = calcule_age(annee);
    printf("Age: %u ans.\n", age);
    return 0;
}
    
```



%esp haut de pile, %ebp activation record, %eax val de retour

## Passage de paramètres : exemple – 2

```

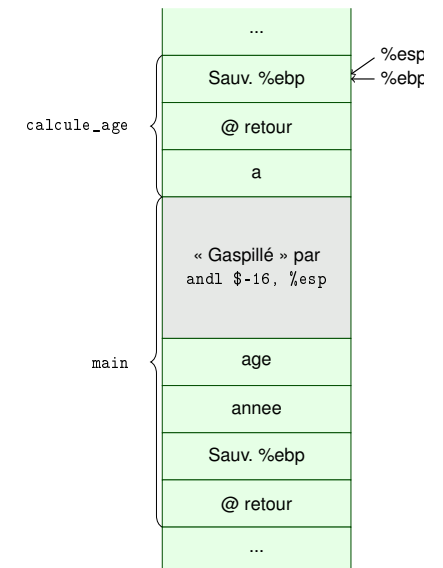
unsigned calcule_age(unsigned a) {
    return 2010 - a;
}

↓

calcule_age:
    pushl %ebp
    movl %esp, %ebp
    // Pas de variable locale

    movl $2010, %eax
    subl 8(%ebp), %eax

    // Valeur de retour dans %eax
    // (par convention)
    leave
    ret
    
```

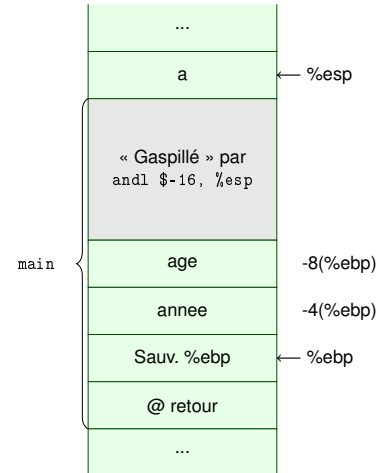


%esp haut de pile, %ebp activation record %eax val de retour

## Passage de paramètres : exemple –3

```
int main(void) {
    unsigned annee, age;
    // ...
    age = calcule_age(annee);
    // ...
}
```

```
main: pushl %ebp
      movl %esp, %ebp
      // 2 variables locales => 8
      // 2 parametres max => 8
      subl $(8+8), %esp
      // Rendre %esp multiple de 16
      andl $-16, %esp
      // ...
      // age = calcule_age(annee)
      movl -4(%ebp), %eax
      movl %eax, 0(%esp)
      call calcule_age
      // Valeur de retour
      // dans %eax
      movl %eax, -8(%ebp)
      // ...
      leave
      ret
```



%esp haut de pile, %ebp activation record, %eax val de retour

## Allocation de registres

Notre machine parfaite avait un nombre infini de registres :

- Ce n'est pas vrai sur une vraie machine.
  - Il faut **réutiliser** au maximum. Si on n'y arrive pas, on met sur la pile/le tas.
- Algorithmes plus ou moins **sioux** sur des graphes de dépendance. Problème **NP-complet**.

## Après la compilation

Il reste deux étapes :

- **Assemblage** : transforme le code assembleur en instructions machine (outil `as`).
- **Édition de liens**.

## Édition de liens – 1

But : trouver le code des fonctions de bibliothèques :

- Chargement **statique** (lien avec des bibliothèques, en C .a) : tout le code est embarqué dans le binaire.
  - Chargement **dynamique** (.so) : le code de la lib n'est pas embarqué
- Ceci est fait par `ld`.

## Édition de liens – 2

```
#include <stdio.h>
#include <math.h>
char* bonjour="hello";
int main(){

    printf("%s\n",bonjour);
    float x;
    scanf("%f",&x);
    printf("%f",cos(x));

    return 0;
}
```

→ compiler avec -c, puis nm du .o :

```
U __isoc99_scanf
00000000 D bonjour // data
U cos
00000000 T main //text
U printf //undefined
U puts
```

► L'édition de lien a pour but de trouver le code "undefined".

## Édition de liens – 3

```
gcc edliens.o
edliens.o: In function 'main':
edliens.c:(.text+0x33): undefined reference to 'cos'
collect2: ld returned 1 exit status
```

Lier avec la libmath :

```
gcc edliens.o -o edliens -lm
```

nm sur le binaire donne :

```
U cos@@GLIBC_2.0
```

(liaison dynamique)

## En résumé

Dessin !

Pour aller un peu (beaucoup) **plus loin** (en compil) :

- L'excellent poly de Tanguy Risset : <http://perso.citi.insa-lyon.fr/trisset/cours/compilStudent.pdf>
- Les livres « Compilers : Principles, Techniques and Tools » (Aho/Sethi/Ullmann), « Engineering a Compiler » (Cooper), ...

# Les optimisations du compilateur

## 1 La chaîne de compilation

## 2 Optimiser ?

- Le compilateur à l'action
- Et moi, si je veux optimiser ?

Ces optimisations sont de différentes natures :

- Optimisations de nature algorithmique pour : minimiser le nombre de registres, supprimer le code mort, éviter les recalculs inutiles. En général **indépendantes de la machine**
- Optimisations **machines dépendantes** pour augmenter la vitesse d'exécution : travail sur les instructions, ...

## Optimisations courantes

Faites par le compilateur :

- Variables vivantes
- Élimination de code mort
- Propagation de constantes
- Déroulement des boucles (élimination de tests mais code plus gros)

## Variables vivantes

Exemple :

```
x:=1;
y:=7;
if y < 20 then x:=78 else x:=42
```

► La première affectation est inutile : « x n'est pas vivante au test  $y < 20$  ».

## Propagation de constantes

Exemple :

```
x:=3;
[...pas d'affectation à x...]
if z < x then ...
```

- ▶ Le test peut être réécrit en  $z < 3$ .
- ▶ Ces optimisations sont classiques et sont appelées **analyses data-flow**.

## Les options -O2/-O3 de gcc

Ces options réalisent les optimisations suivantes (entre autres) :

- Inlining des petites fonctions
- Réutilisation de certains calculs effectués auparavant (accès mémoires entre autres)
- Transformations "intelligentes" de boucles

**Attention** Enlever toutes les options d'optimisation pour GDB.

## Démo options de compilation - 1

(sources : Bogdan Pasca, ENS Lyon)  
Variables, constantes, registres.

```
int main(int argc, char** argv)
{
    int i = 0, j = 0;
    j += 1;
    return j;
}
```

Démo et comparaison.

- Sans option, que peut-on améliorer ?
- Avec -O1, -O2, -O3.
- En changeant l'expression ?

## Démo options de compilation - 2

Optimisations de boucle

```
int main(int argc, char** argv)
{
    int i,j;
    j=0;
    for (i=0 ; i < 100; i++)
        j += 1;
    return j;
}
```

- ▶ Observons !
- ▶ Complexifier l'expression de la boucle ( $j = i*j + 42$  par ex).

## Démo options de compilation - 3

### Inlining

```
int fortytwo(void)
{
    return 42;
}
```

- ▶ Regarder le code généré pour un appel dans le main

## Optimisation de code C

**On DOIT d'abord réfléchir à la complexité** lors de la conception. Ensuite, on peut par exemple :

- parcourir ligne par ligne les matrices
  - privilégier les récursions terminales
  - précalculer certaines valeurs
  - ne pas optimiser (c'est pas toujours utile/souhaitable)
- ▶ Et encore : <http://leto.net/docs/C-optimization.php>
- Attention** Quelques fois, optimiser à la main va à l'encontre du compilateur !