

# Programmation Structurée S6/S7

## Cours 1 : Éléments de compilation

Laure Gonnord

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

Université Lille 1 - Polytech Lille

septembre 2010



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 ?

### 1 La chaîne de compilation

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

### 2 Optimiser ?

## Que fait un compilateur ? 1/2

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.

```
[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;
```

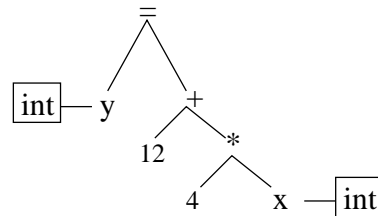
⇒ [TKINT, TKVAR("y"), TKEQ, TKINT(12), TKPLUS, TKINT(4), TKFOIS, TKVAR("x"), TKPVIRG]

## 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]

⇒

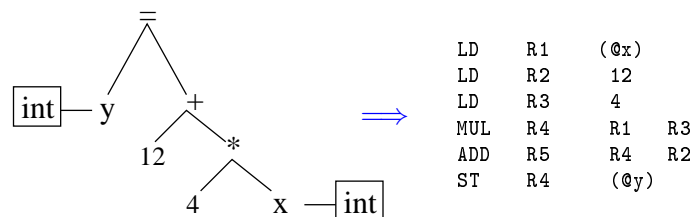


## Arbre abstrait vers code intermédiaire (1/2)

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 (2/3)



▶ (@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 (3/3)

Remarques :

- **En fait**, le code intermédiaire est une **représentation intermédiaire** (structure de donnée interne au compilateur au lieu du texte).
- Cette phase est **délicate**, notamment la traduction des procédures et des fonctions (valeur des paramètres, variables locales, ...)

## Étapes intermédiaires

Sur le code intermédiaire on peut réaliser nombre d'analyses et d'optimisations (vues plus tard).

- ▶ 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
- Allocation des registres
- (**éventuellement**) D'autres optims machine spécifiques
- Génération de code machine.

## 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**.

## 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.
- ▶ Algorithmes plus ou moins **sioux** sur des graphes de dépendance. Problème **NP-complet**.

## Après la compilation

Il reste deux étapes :

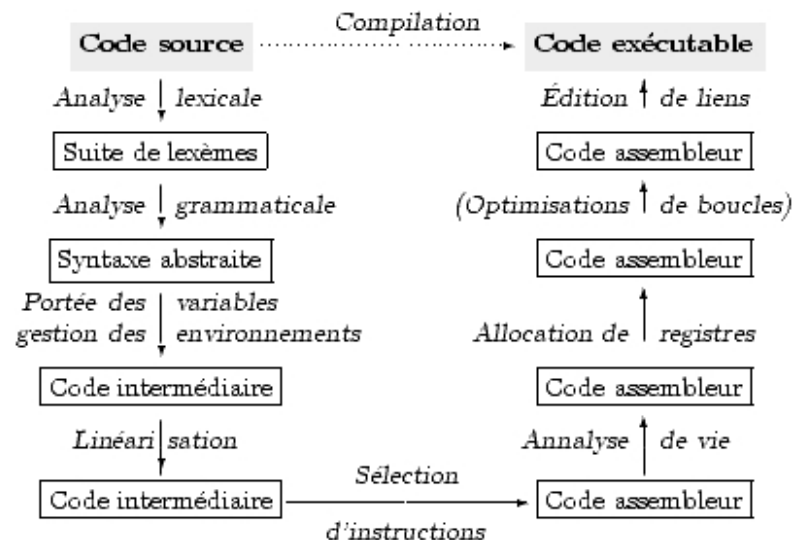
- **Assemblage** : transforme le code assembleur en instructions machine (outil `as`).
- **Édition de liens**.

## Édition de liens

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`

## En résumé



(dessin simplifié, par Luc Maranget, X)

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.

## 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://lcto.net/docs/C-optimization.php>
- Attention** Quelques fois, optimiser à la main va à l'encontre du compilateur !