La vie des programmes

Denis Bouhineau Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

4 avril 2020

Plan

Plan

Aujourd'hui

Nous allons étudier en détail les différentes étapes de compilation permettant de produire un exécutable à partir d'un ou plusieurs fichiers sources.

Aujourd'hui

Nous allons étudier en détail les différentes étapes de compilation permettant de produire un exécutable à partir d'un ou plusieurs fichiers sources.

Remarque : lorsque l'on compile plusieurs fichiers sources en un seul exécutable, on parle de compilation séparée.

Analyse et synthèse

La compilation comporte deux étapes principales :

Analyse et synthèse

La compilation comporte deux étapes principales :

- Étape d'analyse

Analyse et synthèse

La compilation comporte deux étapes principales :

- Étape d'analyse
  - Pré-traitement

Analyse et synthèse

La compilation comporte deux étapes principales :

- Étape d'analyse
  - Pré-traitement
  - Analyse lexicale

Analyse et synthèse

La compilation comporte deux étapes principales :

- Étape d'analyse
  - Pré-traitement
  - Analyse lexicale
  - Analyse syntaxique

Analyse et synthèse

La compilation comporte deux étapes principales :

- Étape d'analyse
  - Pré-traitement
  - Analyse lexicale
  - Analyse syntaxique
  - Analyse sémantique

Analyse et synthèse

La compilation comporte deux étapes principales :

- Étape d'analyse
  - Pré-traitement
  - Analyse lexicale
  - Analyse syntaxique
  - Analyse sémantique
- Étape de synthèse de code

Analyse et synthèse

La compilation comporte deux étapes principales :

- Étape d'analyse
  - Pré-traitement
  - Analyse lexicale
  - Analyse syntaxique
  - Analyse sémantique
- Étape de synthèse de code
  - Génération de code intermédiaire

Analyse et synthèse

La compilation comporte deux étapes principales :

- Étape d'analyse
  - Pré-traitement
  - Analyse lexicale
  - Analyse syntaxique
  - Analyse sémantique
- Étape de synthèse de code
  - Génération de code intermédiaire
  - Optimisation de code intermédiaire

Analyse et synthèse

La compilation comporte deux étapes principales :

- Étape d'analyse
  - Pré-traitement
  - Analyse lexicale
  - Analyse syntaxique
  - Analyse sémantique
- Étape de synthèse de code
  - Génération de code intermédiaire
  - Optimisation de code intermédiaire
  - Génération de code cible

Analyse et synthèse

La compilation comporte deux étapes principales :

- Étape d'analyse
  - Pré-traitement
  - Analyse lexicale
  - Analyse syntaxique
  - Analyse sémantique
- Étape de synthèse de code
  - Génération de code intermédiaire
  - Optimisation de code intermédiaire
  - Génération de code cible

Dans ce cours, nous nous préoccuperons **surtout** de la seconde étape.

Compilation et interprétation

Un programme peut être :

Compilation et interprétation

Un programme peut être :

- compiler, puis exécuter

Compilation et interprétation

Un programme peut être :

- compiler, puis exécuter
- ou exécuter directement *via* un interpréteur

Compilation et interprétation

Un programme peut être :

- compiler, puis exécuter
- ou exécuter directement *via* un interpréteur

Compilateurs et interpréteurs partagent la première étape de travail (étape d'analyse).

Compilation et interprétation

Un programme peut être :

- compiler, puis exécuter
- ou exécuter directement *via* un interpréteur

Compilateurs et interpréteurs partagent la première étape de travail (étape d'analyse).

Compilateurs et interpréteurs se distinguent au moment de l'exécution :

Compilation et interprétation

Un programme peut être :

- compiler, puis exécuter
- ou exécuter directement *via* un interpréteur

Compilateurs et interpréteurs partagent la première étape de travail (étape d'analyse).

Compilateurs et interpréteurs se distinguent au moment de l'exécution :

- le code cible produit par un compilateur est exécuté directement par la machine cible

Compilation et interprétation

Un programme peut être :

- compiler, puis exécuter
- ou exécuter directement *via* un interpréteur

Compilateurs et interpréteurs partagent la première étape de travail (étape d'analyse).

Compilateurs et interpréteurs se distinguent au moment de l'exécution :

- le code cible produit par un compilateur est exécuté directement par la machine cible
- la structure intermédiaire obtenue par l'interpréteur est exécutée par l'interpréteur lui-même (comme sur une machine virtuelle)

Analyse (rapidement)

L'étape d'analyse se décompose en plusieurs tâches :

Analyse (rapidement)

L'étape d'analyse se décompose en plusieurs tâches :

- **Précompilation :** Suppression des commentaires, inclusions, macros

  `arm-eabi-gcc -E monprog.c > monprog.i`

  Source `monprog.c` $\longrightarrow$ source $\ll$ enrichi $\gg$ `monprog.i`

Analyse (rapidement)

L'étape d'analyse se décompose en plusieurs tâches :

- **Précompilation :** Suppression des commentaires, inclusions, macros

  `arm-eabi-gcc -E monprog.c > monprog.i`

  Source `monprog.c` $\longrightarrow$ source $\ll$ enrichi $\gg$ `monprog.i`

- **Analyse lexicale :** lecture de `monprog.i` à l'aide d'automates décrivant les éléments lexicaux autorisés (lexèmes)

  source $\ll$ enrichi $\gg$ `monprog.i` $\longrightarrow$ `flux de lexèmes`

Analyse (rapidement)

L'étape d'analyse se décompose en plusieurs tâches :

- **Précompilation :** Suppression des commentaires, inclusions, macros

  `arm-eabi-gcc -E monprog.c > monprog.i`

  Source `monprog.c` $\longrightarrow$ source $\ll$ enrichi $\gg$ `monprog.i`

- **Analyse lexicale :** lecture de `monprog.i` à l'aide d'automates décrivant les éléments lexicaux autorisés (lexèmes)

  source $\ll$ enrichi $\gg$ `monprog.i` $\longrightarrow$ `flux de lexèmes`

- **Analyse syntaxique :** analyse du `flux de lexèmes` (souvent à l'aide d'automates à pile) selon la grammaire décrivant le langage de programmation

  `flux de lexèmes` $\longrightarrow$ `arbre syntaxique`

Analyse (rapidement)

L'étape d'analyse se décompose en plusieurs tâches :

- **Précompilation :** Suppression des commentaires, inclusions, macros

  `arm-eabi-gcc -E monprog.c > monprog.i`

  Source `monprog.c` → source ≪ enrichi ≫ `monprog.i`

- **Analyse lexicale :** lecture de `monprog.i` à l'aide d'automates décrivant les éléments lexicaux autorisés (lexèmes)

  source ≪ enrichi ≫ `monprog.i` → `flux de lexèmes`

- **Analyse syntaxique :** analyse du `flux de lexèmes` (souvent à l'aide d'automates à pile) selon la grammaire décrivant le langage de programmation

  `flux de lexèmes` → `arbre syntaxique`

- **Analyse sémantique :** vérification globale des définitions, types déclarés, dépendances, *etc.* des éléments de l'arbre syntaxique

  `arbre syntaxique` → `arbre enrichi (décoré)` et dictionnaires

Analyse (rapidement)

Exemple :

monprog.c :

```
/* monprog.c */
#include "malib.h"

#define max(a,b) ((a)>(b)?(a):(b))

main()
{
  int x,y,z;
  x = 10;
  y = 15;
  z = max(x,y);
  plus (&z);
}
```

malib.c :

```
void plus (int *ai)
{
  (*ai) = (*ai) + 1;
}
```

malib.h :

```
#ifndef MALIB

#define MALIB

/* incrementation d'un mot memoire */
extern void plus (int *);
#endif
```

Analyse (rapidement, la précompilation) : arm-eabi-gcc -E monprog.c > monprog.i (2/2)

monprog.i :

```
# 1 "monprog.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "monprog.c"
# 1 "malib.h" 1

extern void plus (int *);
# 3 "monprog.c" 2

main()
{
  int x,y,z;
  x = 10;
  y = 15;
  z = ((x)>(y)?(x):(y));
  plus (&z);
}
```

Analyse (rapidement, l'analyse lexicale)

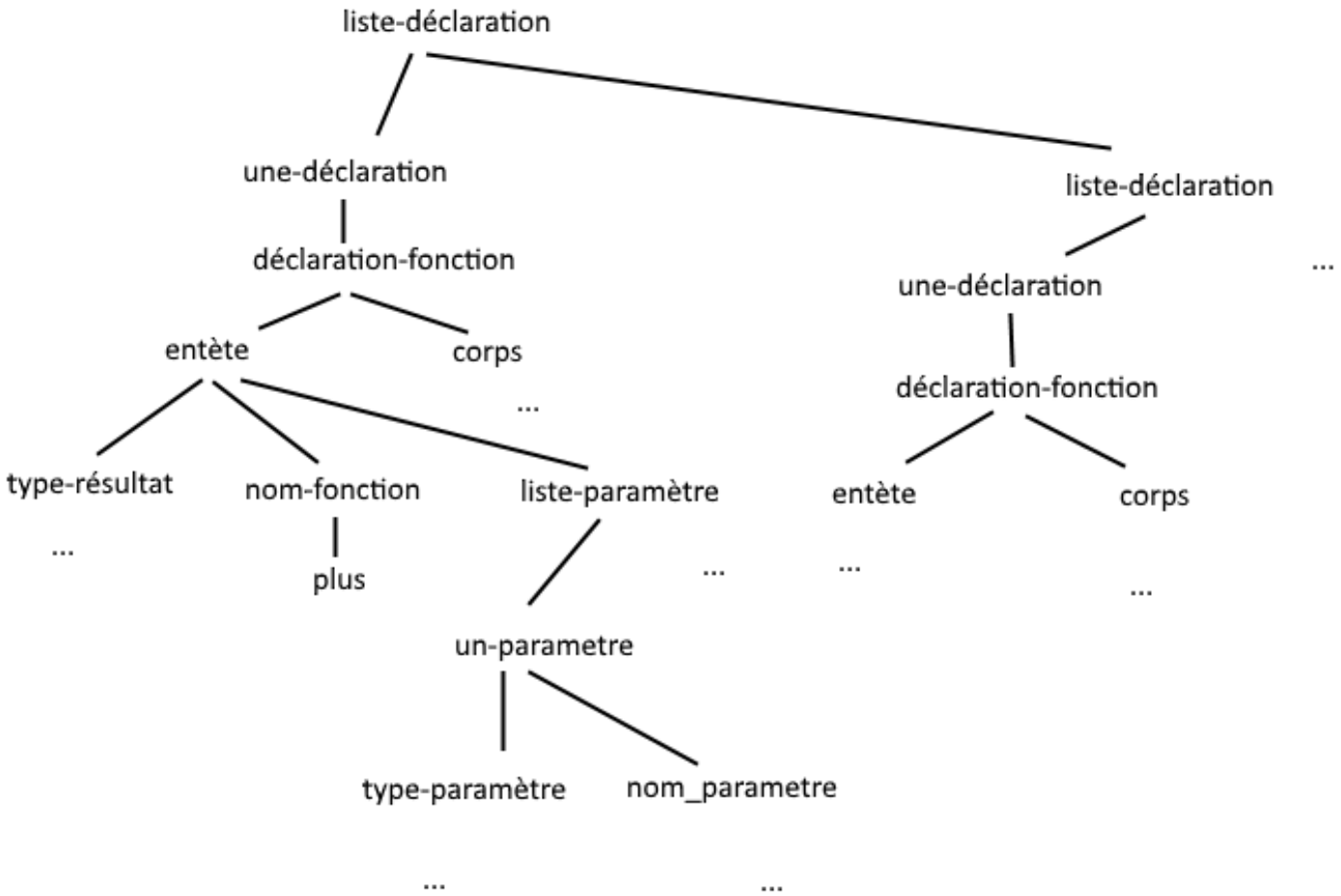
le flux de lexèmes (identifiants : (a..zA..Z)(a..zA..Z0..9)*, nombres : (0..9)+(,(0..9)+) ?, mots clés : "int" | "void" | "if" | "for" | ... , *etc.*) :

```
(identifiant,z)
(operateur,"=")
(operateur,"(")
(operateur,"(")
(identifiant,x)
(operateur,")")
(operateur,">")
(operateur,"(")
(identifiant,y)
(operateur,")")
(operateur,"?")
(operateur,"(")
(identifiant,x)
(operateur,")")
(operateur,":")
(operateur,"(")
(identifiant,y)
(operateur,")")
```

```
(operateur,")")
(operateur,";")
(identifiant,plus)
(operateur,"(")
(operateur,"&")
(identifiant,z)
(operateur,")")
(operateur,";")
(operateur,"}")
(mot-cle,extern)
(type,void)
(identifiant,plus)
(operateur,"(")
(type,int)
(operateur,"*")
(operateur,")")
(operateur,";")
(mot-cle,main)
```

```
(operateur,"(")
(operateur,")")
(operateur,"{")
(type,int)
(identifiant,x)
(operateur,",")
(identifiant,y)
(operateur,",")
(identifiant,z)
(operateur,";")
(identifiant,x)
(operateur,"=")
(nombre,10)
(operateur,";")
(identifiant,y)
(operateur,"=")
(nombre,15)
(operateur,";")
```

Analyse (rapidement, l'analyse syntaxique)



Analyse (rapidement, l'analyse syntaxique)

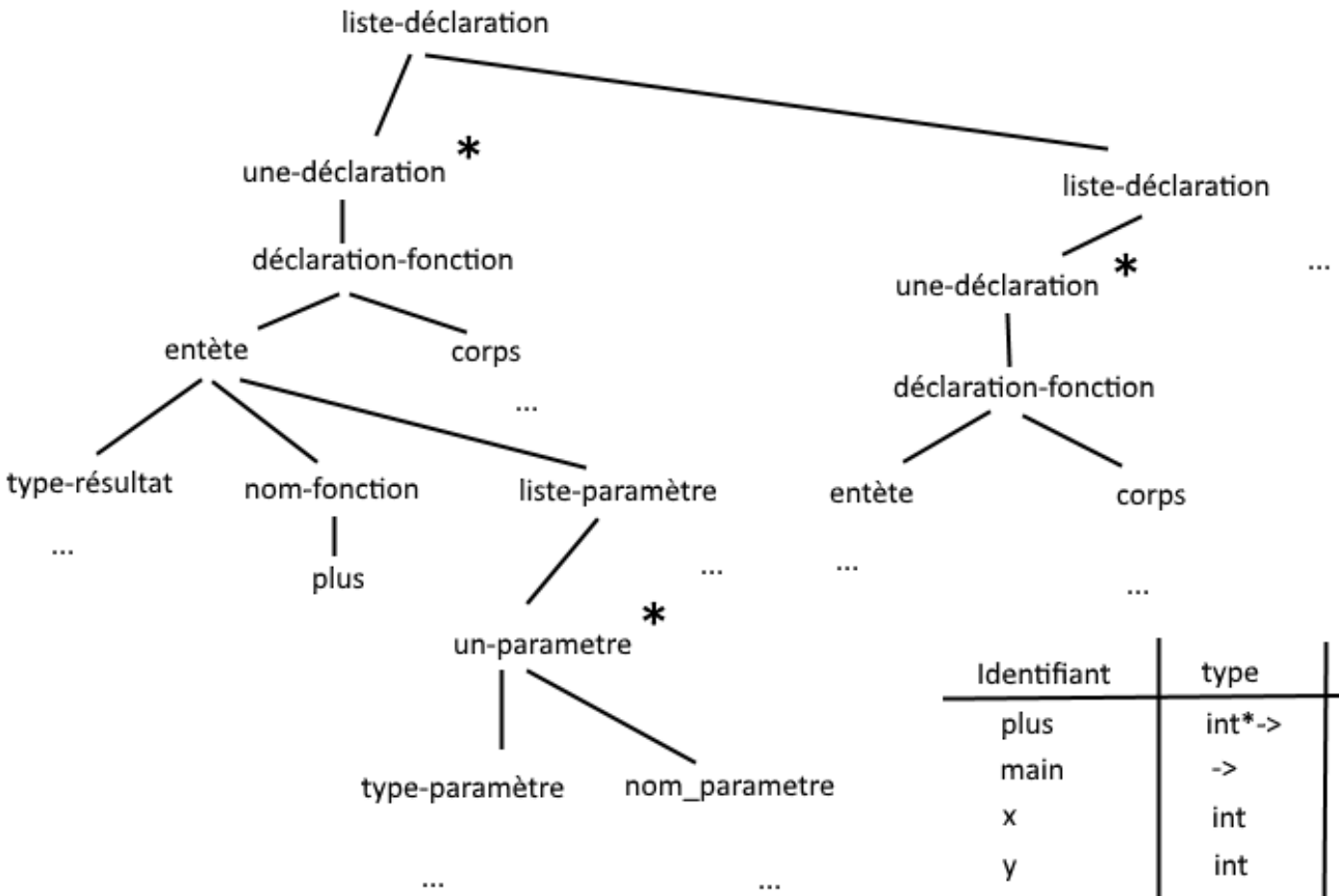
La syntaxe basée sur une grammaire

Exemple de règle de grammaire :

Affectation : := PartieGauche "=" Expression

Analyse (rapidement, l'analyse syntaxique)

L'arbre décoré + le dictionnaire des identifiants :



Identifiant	type	portée
plus	int*->	glob
main	->	glob
x	int	main
y	int	main
z	int	main

Plan

Un exemple en langage C

```
/* fichier fonctions.c */
 int somme (int *t, int n) {
 int i, s;
 s = 0;
 for (i=0;i<n;i++) s = s + t[i];
 return (s); }

 int max (int *t, int n) {
 int i, m;
 m = t[0];
 for (i=1;i<n;i++) if (m < t[i]) m = t[i];
 return (m); }

==================================

/* fichier main.c */
extern int somme (int *t, int n);
extern int max (int *t, int n);

#define TAILLE 10
static int TAB [TAILLE];

main () {
 int i,u,v;
 for (i=0;i<TAILLE;i++) scanf ("%d", &TAB[i]);
 u = somme (TAB, TAILLE);
 v = max (TAB, TAILLE); }
```

Un exemple en langage C

```
/* fichier fonctions.c */
 int somme (int *t, int n) {
 int i, s;
 s = 0;
 for (i=0;i<n;i++) s = s + t[i];
 return (s); }

 int max (int *t, int n) {
 int i, m;
 m = t[0];
 for (i=1;i<n;i++) if (m < t[i]) m = t[i];
 return (m); }

==================================

/* fichier main.c */
extern int somme (int *t, int n);
extern int max (int *t, int n);

#define TAILLE 10
static int TAB [TAILLE];

main () {
 int i,u,v;
 for (i=0;i<TAILLE;i++) scanf ("%d", &TAB[i]);
 u = somme (TAB, TAILLE);
 v = max (TAB, TAILLE); }
```

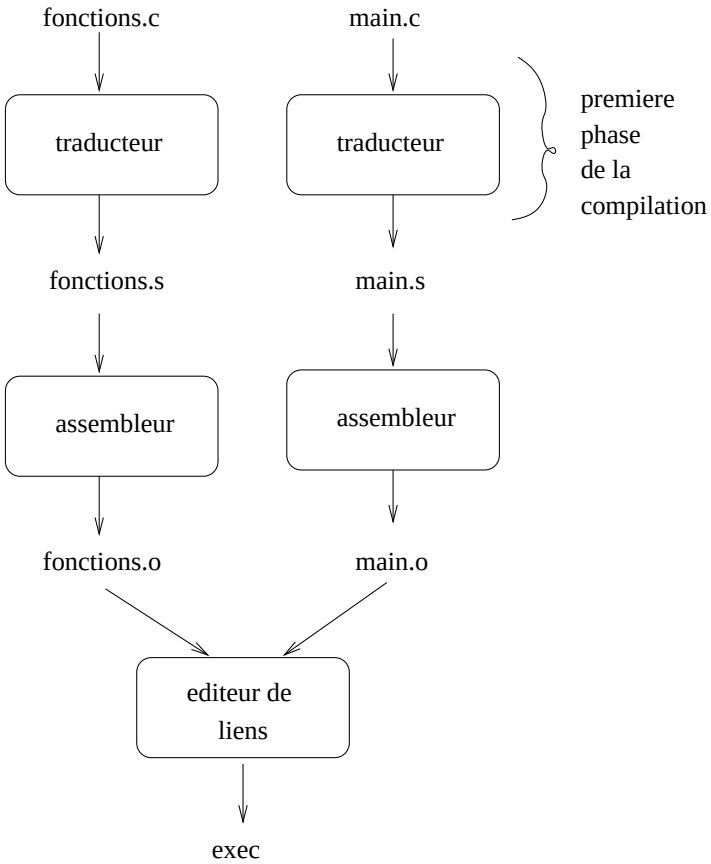
- Dans le fichier `main.c` les fonctions `somme` et `max` sont dites importées : elles sont définies dans un autre fichier.

Un exemple en langage C

```
/* fichier fonctions.c */
 int somme (int *t, int n) {
 int i, s;
 s = 0;
 for (i=0;i<n;i++) s = s + t[i];
 return (s); }

 int max (int *t, int n) {
 int i, m;
 m = t[0];
 for (i=1;i<n;i++) if (m < t[i]) m = t[i];
 return (m); }

================================

/* fichier main.c */
extern int somme (int *t, int n);
extern int max (int *t, int n);

#define TAILLE 10
static int TAB [TAILLE];

main () {
 int i,u,v;
 for (i=0;i<TAILLE;i++) scanf ("%d", &TAB[i]);
 u = somme (TAB, TAILLE);
 v = max (TAB, TAILLE); }
```

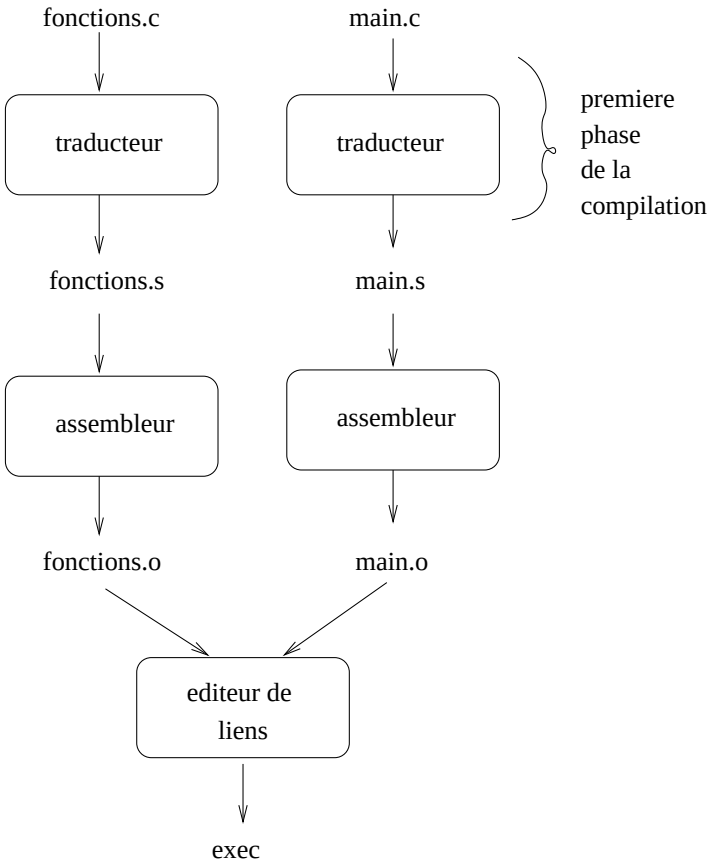
- Dans le fichier `main.c` les fonctions `somme` et `max` sont dites importées : elles sont définies dans un autre fichier.
- Dans le fichier `fonctions.c`, `somme` et `max` sont dites exportées : elles sont utilisables dans un autre fichier.

Compilation haut niveau



Remarque : la phase de traduction comporte une phase de pré-traitement dite phase de ≪ pré-compilation ≫ où le code source est transformé en code source ≪ enrichi ≫(les directives de pré-compilations — # — sont traitées)

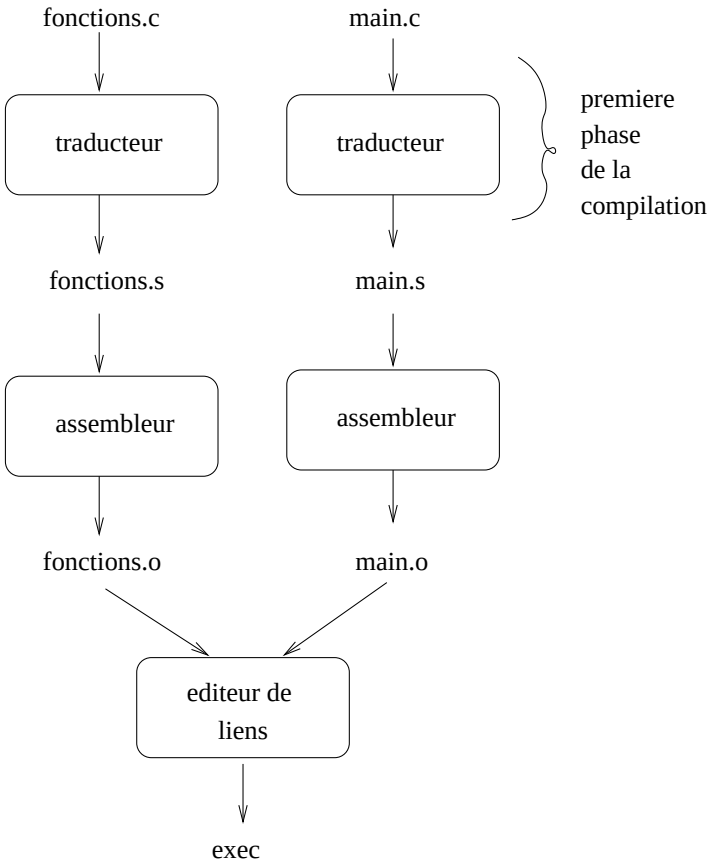
Compilation haut niveau



Remarque : la phase de traduction comporte une phase de pré-traitement dite phase de ≪ pré-compilation ≫ où le code source est transformé en code source ≪ enrichi ≫(les directives de pré-compilations — # — sont traitées)

- Pour ≪ compiler ≫, produire un exécutable, on enchaine les commandes :
  - `gcc -c fonctions.c`
  - `gcc -c main.c`
  - `gcc -o exec main.o fonctions.o`

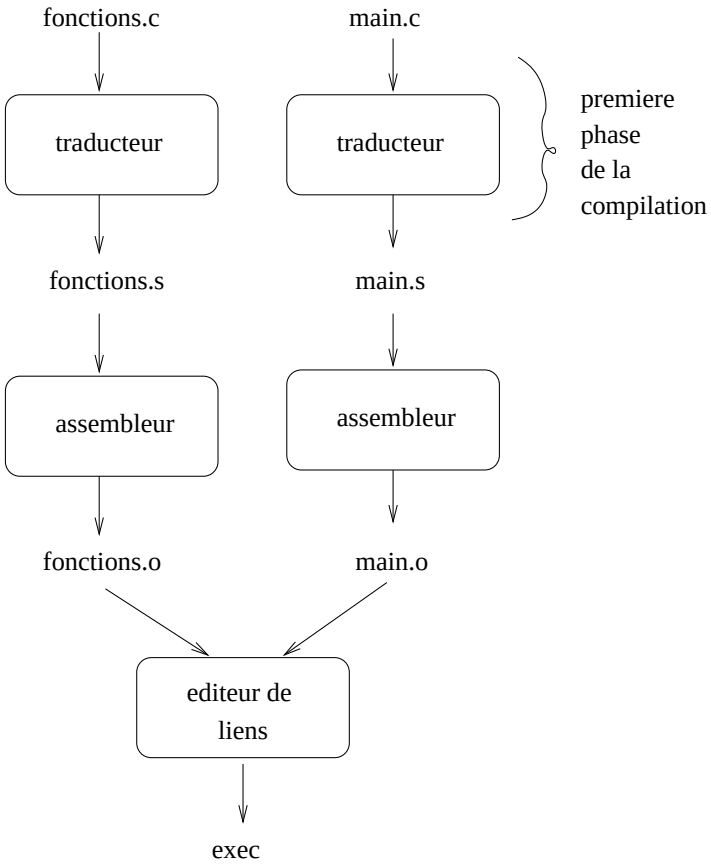
Compilation haut niveau



Remarque : la phase de traduction comporte une phase de pré-traitement dite phase de ≪ pré-compilation ≫ où le code source est transformé en code source ≪ enrichi ≫(les directives de pré-compilations — # — sont traitées)

- Pour ≪ compiler ≫, produire un exécutable, on enchaine les commandes :
  - `gcc -c fonctions.c`
  - `gcc -c main.c`
  - `gcc -o exec main.o fonctions.o`
- La commande `gcc -c main.c` produit un fichier appelé `main.o`.
- La commande `gcc -c fonctions.c` produit un fichier `fonctions.o`.
- Les fichiers `fonctions.o` et `main.o` contiennent du binaire translatable, c'est-à-dire, du code binaire qui ne peut pas directement être exécuté en mémoire.

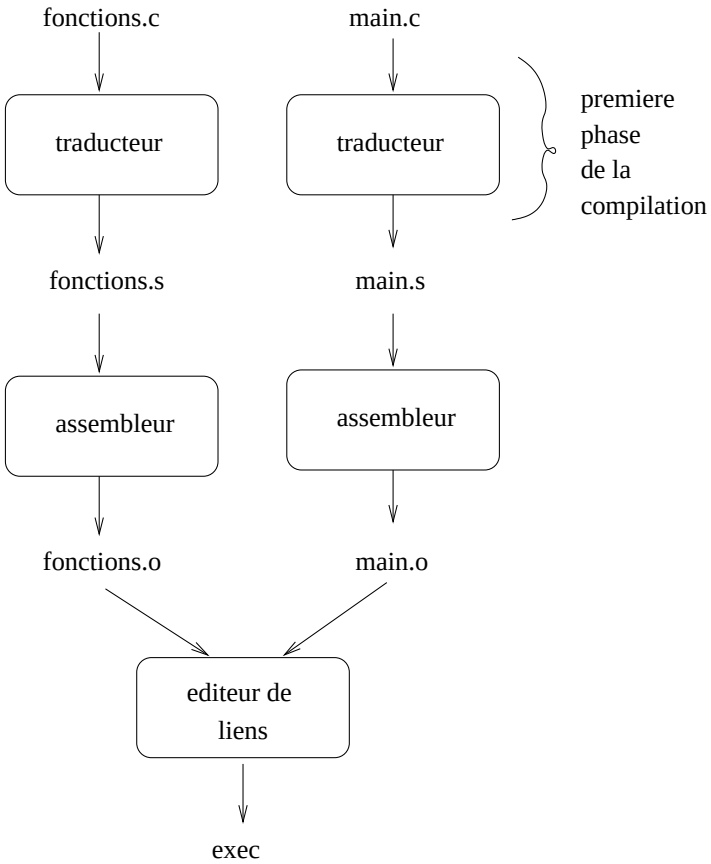
Compilation haut niveau



Remarque : la phase de traduction comporte une phase de pré-traitement dite phase de ≪ pré-compilation ≫ où le code source est transformé en code source ≪ enrichi ≫(les directives de pré-compilations — # — sont traitées)

- Pour ≪ compiler ≫, produire un exécutable, on enchaine les commandes :
  - `gcc -c fonctions.c`
  - `gcc -c main.c`
  - `gcc -o exec main.o fonctions.o`
- La commande `gcc -c main.c` produit un fichier appelé `main.o`.
- La commande `gcc -c fonctions.c` produit un fichier `fonctions.o`.
- Les fichiers `fonctions.o` et `main.o` contiennent du binaire translatable, c'est-à-dire, du code binaire qui ne peut pas directement être exécuté en mémoire.
- La commande `gcc -o exec main.o fonctions.o` produit le fichier `exec` qui contient du binaire exécutable. Ce fichier résulte de la liaison des deux fichiers objets (`.o`). On parle d'édition de liens.

Compilation haut niveau



Remarque : la phase de traduction comporte une phase de pré-traitement dite phase de ≪ pré-compilation ≫ où le code source est transformé en code source ≪ enrichi ≫(les directives de pré-compilations — # — sont traitées)

- Pour ≪ compiler ≫, produire un exécutable, on enchaine les commandes :
  - `gcc -c fonctions.c`
  - `gcc -c main.c`
  - `gcc -o exec main.o fonctions.o`
- La commande `gcc -c main.c` produit un fichier appelé `main.o`.
- La commande `gcc -c fonctions.c` produit un fichier `fonctions.o`.
- Les fichiers `fonctions.o` et `main.o` contiennent du binaire translatable, c'est-à-dire, du code binaire qui ne peut pas directement être exécuté en mémoire.
- La commande `gcc -o exec main.o fonctions.o` produit le fichier `exec` qui contient du binaire exécutable. Ce fichier résulte de la liaison des deux fichiers objets (`.o`). On parle d'édition de liens.
- **Remarque :** `gcc` cache l'appel à différents outils (logiciels).

Exemple avec ARM : `essai.s` et `lib.s`

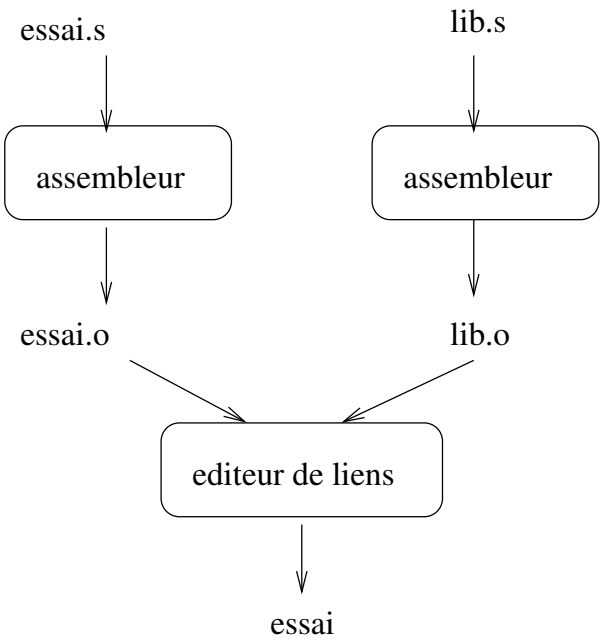
essai.s

```
    .text
    .global main
main:
      mov r0, #0
bcle: cmp r0, #10
      beq fin
       ldr r3, adr_xx
       ldr r2, [r3]
       bl add1
       str r2, [r3]
       add r0, r0, #1
       b bcle
fin:   b exit
adr_xx: .word xx
    .data
    .word 99
xx: .word 3
```

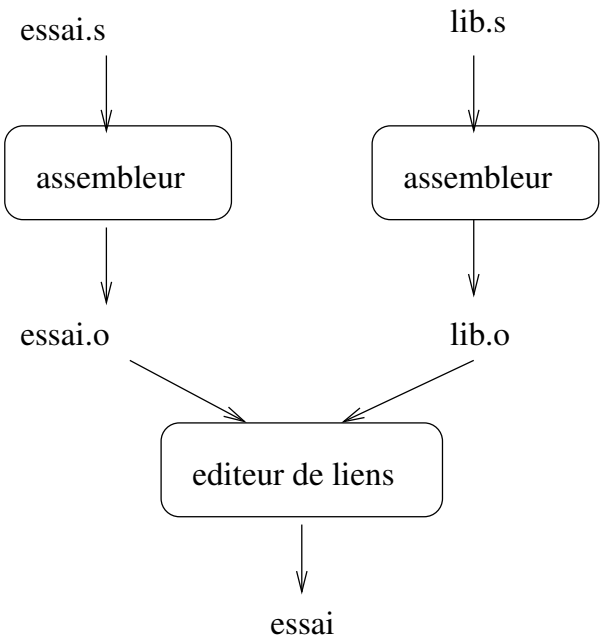
lib.s

```
    .text

    .global add1

add1 : add r2, r2, #1
       mov pc, lr
```

Compilation en assembleur

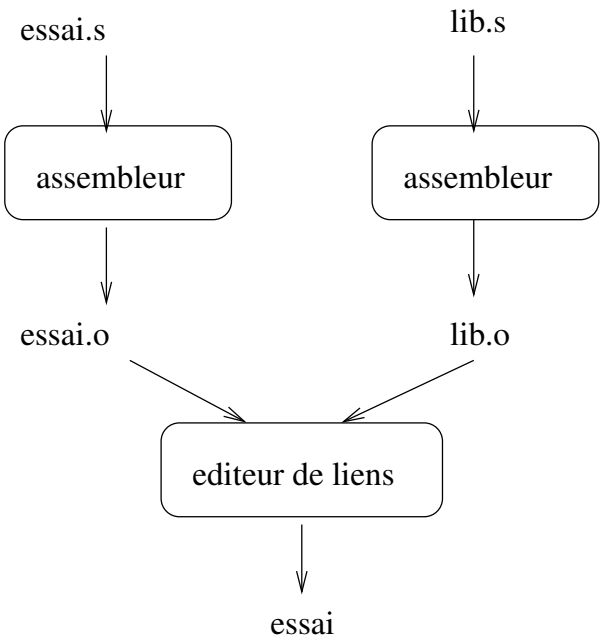


Compilation en assembleur



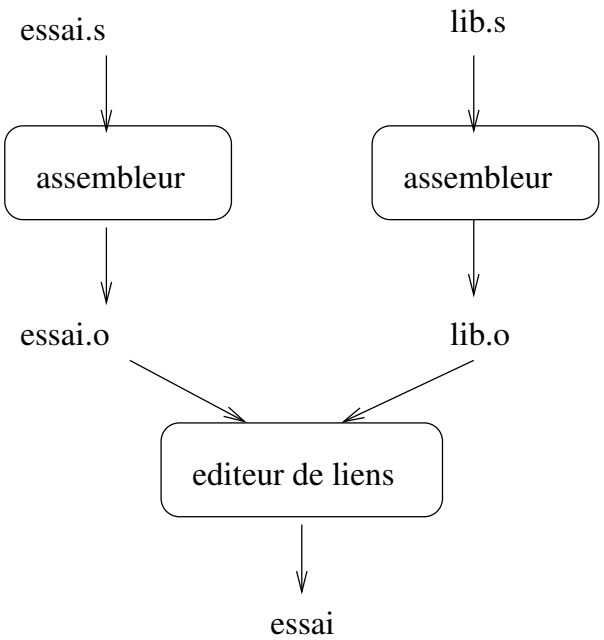
- Pour ≪ compiler ≫, on enchaine les commandes :
  - `arm-eabi-gcc -c essai.s`
  - `arm-eabi-gcc -c lib.s`
  - `arm-eabi-gcc -o essai essai.o lib.o`

Compilation en assembleur



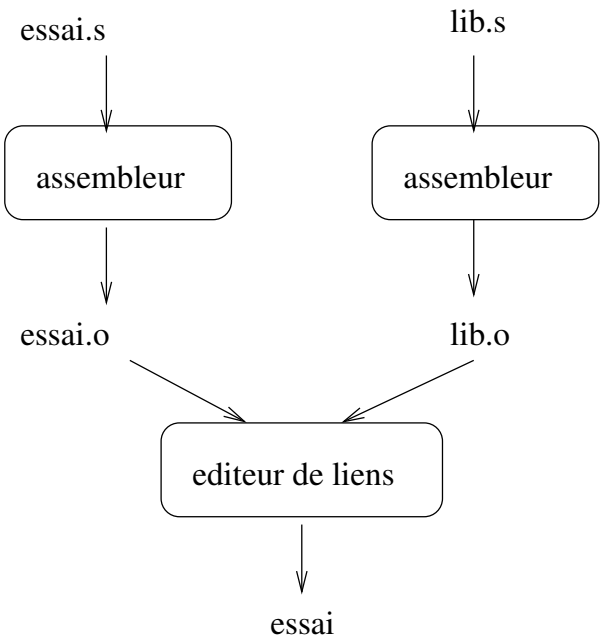
- Pour ≪ compiler ≫, on enchaine les commandes :
  - `arm-eabi-gcc -c essai.s`
  - `arm-eabi-gcc -c lib.s`
  - `arm-eabi-gcc -o essai essai.o lib.o`
- Les commandes `arm-eabi-gcc -c essai.s` et `arm-eabi-gcc -c lib.s` produisent les fichiers `essai.o` et `lib.o`.
- Les fichiers `essai.o` et `lib.o` contiennent du binaire translatable.

Compilation en assembleur



- Pour ≪ compiler ≫, on enchaine les commandes :
  - `arm-eabi-gcc -c essai.s`
  - `arm-eabi-gcc -c lib.s`
  - `arm-eabi-gcc -o essai essai.o lib.o`
- Les commandes `arm-eabi-gcc -c essai.s` et `arm-eabi-gcc -c lib.s` produisent les fichiers `essai.o` et `lib.o`.
- Les fichiers `essai.o` et `lib.o` contiennent du binaire translatable.
- La commande `arm-eabi-gcc -o essai essai.o lib.o` produit le fichier `essai` qui contient du binaire exécutable. Ce fichier résulte de la liaison des deux fichiers objets (`.o`). On parle d'édition de liens.

Compilation en assembleur



- Pour ≪ compiler ≫, on enchaine les commandes :
  - `arm-eabi-gcc -c essai.s`
  - `arm-eabi-gcc -c lib.s`
  - `arm-eabi-gcc -o essai essai.o lib.o`
- Les commandes `arm-eabi-gcc -c essai.s` et `arm-eabi-gcc -c lib.s` produisent les fichiers `essai.o` et `lib.o`.
- Les fichiers `essai.o` et `lib.o` contiennent du binaire translatable.
- La commande `arm-eabi-gcc -o essai essai.o lib.o` produit le fichier `essai` qui contient du binaire exécutable. Ce fichier résulte de la liaison des deux fichiers objets (`.o`). On parle d'édition de liens.
- **Remarque :** `arm-eabi-gcc` cache différents outils.
  - La commande `arm-eabi-gcc` appliqué à un fichier `.s` avec l'option `-c` correspond à la commande `arm-eabi-as`.
  - La commande `arm-eabi-gcc` avec l'option utilisée avec `-o` correspond à la commande d'édition de liens `arm-eabi-ld`.

Plan

Du langage C vers l'assembleur (ou un code intermédiaire)

- L'objectif d'un langage de haut niveau type langage C est de permettre au programmeur de s'abstraire des détails inhérents au fonctionnement de la machine.

Du langage C vers l'assembleur (ou un code intermédiaire)

- L'objectif d'un langage de haut niveau type langage C est de permettre au programmeur de s'abstraire des détails inhérents au fonctionnement de la machine.
- Il permet de manipuler des concepts bien plus élaborés.

Du langage C vers l'assembleur (ou un code intermédiaire)

- L'objectif d'un langage de haut niveau type langage C est de permettre au programmeur de s'abstraire des détails inhérents au fonctionnement de la machine.
- Il permet de manipuler des concepts bien plus élaborés.
- Mais il empêche la gestion de certains de ces détails.

Du langage C vers l'assembleur (ou un code intermédiaire)

- L'objectif d'un langage de haut niveau type langage C est de permettre au programmeur de s'abstraire des détails inhérents au fonctionnement de la machine.
- Il permet de manipuler des concepts bien plus élaborés.
- Mais il empêche la gestion de certains de ces détails.
- La première phase de la compilation consiste en la traduction systématique d'une syntaxe complexe en un langage plus simple et plus proche de la machine (langage machine ou code intermédiaire).

Exemple : traduction d'une conditionnelle

Prenons une conditionnelle :

`si Condition alors { ListeInstructions }`

Exemple : traduction d'une conditionnelle

Prenons une conditionnelle :

`si Condition alors { ListeInstructions }`

elle sera traduite selon le schéma (récursif) :

```
etiq_debut:
            traduction(Condition) avec positionnement de ZNCV
            branch si (non vérifiée) a etiq_suite
            traduction(ListeInstruction)
etiq_suite:
```

Les schéma de traduction

Il y a ainsi des schémas (récursifs) de traduction prévus **pour toutes les règles de grammaire** décrivant les concepts du langage de programmation. Ces schémas sont définis pour un type de machine (large).

Les schéma de traduction

Il y a ainsi des schémas (récursifs) de traduction prévus **pour toutes les règles de grammaire** décrivant les concepts du langage de programmation. Ces schémas sont définis pour un type de machine (large).

Exemples de schémas :

- pour l'évaluation d'opérateurs arithmétiques
- pour l'évaluation d'opérateurs relationnels
- pour l'affectation
- pour les structures de contrôle
- pour la définition de fonctions
- *etc.*

Plan

Problématique

L'objectif de l'assembleur est de produire du code binaire à partir du langage d'assemblage.

Problématique

L'objectif de l'assembleur est de produire du code binaire à partir du langage d'assemblage.

Il n'est pas toujours possible de produire du binaire qui puisse être directement copié en mémoire pour deux raisons principalement :

Problématique

L'objectif de l'assembleur est de produire du code binaire à partir du langage d'assemblage.

Il n'est pas toujours possible de produire du binaire qui puisse être directement copié en mémoire pour deux raisons principalement :

Cas 1 On ne connaît pas en général l'adresse à laquelle les zones **text** et **data** doivent être rangées en mémoire.

Problématique

L'objectif de l'assembleur est de produire du code binaire à partir du langage d'assemblage.

Il n'est pas toujours possible de produire du binaire qui puisse être directement copié en mémoire pour deux raisons principalement :

Cas 1 On ne connaît pas en général l'adresse à laquelle les zones **text** et **data** doivent être rangées en mémoire.

Cas 2 Le programme peut faire référence à des noms qui ne sont pas définis dans le fichier en cours de traduction.

Problématique

L'objectif de l'assembleur est de produire du code binaire à partir du langage d'assemblage.

Il n'est pas toujours possible de produire du binaire qui puisse être directement copié en mémoire pour deux raisons principalement :

Cas 1 On ne connaît pas en général l'adresse à laquelle les zones **text** et **data** doivent être rangées en mémoire.

Cas 2 Le programme peut faire référence à des noms qui ne sont pas définis dans le fichier en cours de traduction.

- Dans le premier cas, on peut produire une **image** du binaire à partir de l'adresse 0, à charge du matériel de translater l'image à l'adresse de chargement pour l'exécution (il faut garder les informations permettant de savoir quelles sont les adresses à translater)

Problématique

L'objectif de l'assembleur est de produire du code binaire à partir du langage d'assemblage.

Il n'est pas toujours possible de produire du binaire qui puisse être directement copié en mémoire pour deux raisons principalement :

Cas 1 On ne connaît pas en général l'adresse à laquelle les zones **text** et **data** doivent être rangées en mémoire.

Cas 2 Le programme peut faire référence à des noms qui ne sont pas définis dans le fichier en cours de traduction.

- Dans le premier cas, on peut produire une **image** du binaire à partir de l'adresse 0, à charge du matériel de translater l'image à l'adresse de chargement pour l'exécution (il faut garder les informations permettant de savoir quelles sont les adresses à translater)
- Dans le deuxième cas on ne peut rien faire.

Premier cas

```
    .text
    .global main
main:
      mov r0, #0
bcle: cmp r0, #10
      beq fin
       ldr r3, adr_xx
       ldr r2, [r3]
       bl add1
       str r2, [r3]
       add r0, r0, #1
       b bcle
fin:   b exit
adr_xx: .word xx
    .data
    .word 99
xx: .word 3
```

L'adresse associée au symbole `xx` est : adresse de début de la zone `data` + 4 mais encore faut-il connaître l'adresse de début de la zone `data` !

Premier cas

```
    .text
    .global main
main:
      mov r0, #0
bcle: cmp r0, #10
      beq fin
       ldr r3, adr_xx
       ldr r2, [r3]
       bl add1
       str r2, [r3]
       add r0, r0, #1
       b bcle
fin:   b exit
adr_xx: .word xx
    .data
    .word 99
xx: .word 3
```

L'adresse associée au symbole `xx` est : adresse de début de la zone `data` + 4 mais encore faut-il connaître l'adresse de début de la zone `data` !

Si on considère que la zone `data` est chargée à l'adresse **0**, l'adresse associée à `xx` est alors **4**. Si on doit translater le programme à l'adresse **2000**, il faut se rappeler que à l'adresse `adr_xx` on doit modifier la valeur **4** en **2000 + 4**.

Premier cas

```
    .text
    .global main
main:
      mov r0, #0
bcle: cmp r0, #10
      beq fin
       ldr r3, adr_xx
       ldr r2, [r3]
       bl add1
       str r2, [r3]
       add r0, r0, #1
       b bcle
fin:   b exit
adr_xx: .word xx
    .data
    .word 99
xx: .word 3
```

L'adresse associée au symbole `xx` est : adresse de début de la zone `data` + 4 mais encore faut-il connaître l'adresse de début de la zone `data` !

Si on considère que la zone `data` est chargée à l'adresse **0**, l'adresse associée à `xx` est alors **4**. Si on doit translater le programme à l'adresse **2000**, il faut se rappeler que à l'adresse `adr_xx` on doit modifier la valeur **4** en **2000 + 4**.

Cette information à mémoriser est appelée une donnée de translation (**relocation** en anglais).

Deuxième cas

```
    .text
    .global main
main:
      mov r0, #0
bcle: cmp r0, #10
      beq fin
       ldr r3, adr_xx
       ldr r2, [r3]
       bl add1
       str r2, [r3]
       add r0, r0, #1
       b bcle
fin:   b exit
adr_xx: .word xx
    .data
    .word 99
xx: .word 3
```

Deuxième cas

```
    .text
    .global main
main:
      mov r0, #0
bcle: cmp r0, #10
      beq fin
       ldr r3, adr_xx
       ldr r2, [r3]
       bl add1
       str r2, [r3]
       add r0, r0, #1
       b bcle
fin:   b exit
adr_xx: .word xx
    .data
    .word 99
xx: .word 3
```

- Dans le fichier `essai.o` il n'est pas possible de calculer le déplacement de l'instruction `bl add1` puisque l'on ne sait pas où est l'étiquette `add1` quand l'assembleur traite le fichier `essai.s`. En effet l'étiquette est dans un autre fichier : `lib.s`

Deuxième cas

```
    .text
    .global main
main:
      mov r0, #0
bcle: cmp r0, #10
      beq fin
       ldr r3, adr_xx
       ldr r2, [r3]
       bl add1
       str r2, [r3]
       add r0, r0, #1
       b bcle
fin:   b exit
adr_xx: .word xx
    .data
    .word 99
xx: .word 3
```

- Dans le fichier `essai.o` il n'est pas possible de calculer le déplacement de l'instruction `bl add1` puisque l'on ne sait pas où est l'étiquette `add1` quand l'assembleur traite le fichier `essai.s`. En effet l'étiquette est dans un autre fichier : `lib.s`
- **Reprenons l'exemple en langage C.** Suite à la traduction en langage d'assemblage, dans le fichier `main.s` les références aux fonctions `somme` et `max` ne peuvent être complétées car les fonctions en question ne sont pas définies dans le fichier `main.c` mais dans `fonctions.c`.

Que contient un fichier .s ?

```
    .text
    .global main
main:
      mov r0, #0
bcle: cmp r0, #10
      beq fin
       ldr r3, adr_xx
       ldr r2, [r3]
       bl add1
       str r2, [r3]
       add r0, r0, #1
       b bcle
fin:   b exit
adr_xx: .word xx
    .data
    .word 99
xx: .word 3
```

- des directives : `.data`, `.bss`, `.text`, `.word`, `.hword`, `.byte`, `.skip`, `.asciz`, `.align`
- des étiquettes appelées aussi symboles
- des instructions du processeur
- des commentaires : `@ blabla`

Note : Parfois une directive (`.org`) permet de fixer l'adresse où sera logé le programme en mémoire. Cette facilité permet alors de calculer certaines adresses dès la phase d'assemblage.

Que contient un fichier .o ?

Que contient un fichier .o ?

- un entête contenant la taille du fichier, les adresses des différentes tables, la taille de la zone de données non initialisées (`bss`), etc.

Que contient un fichier .o ?

- un entête contenant la taille du fichier, les adresses des différentes tables, la taille de la zone de données non initialisées (`bss`), etc.
- la zone de données (`data`) parfois incomplète

Que contient un fichier .o ?

- un entête contenant la taille du fichier, les adresses des différentes tables, la taille de la zone de données non initialisées (`bss`), etc.
- la zone de données (`data`) parfois incomplète
- la zone des instructions (`text`) parfois incomplète

Que contient un fichier .o ?

- un entête contenant la taille du fichier, les adresses des différentes tables, la taille de la zone de données non initialisées (`bss`), etc.
- la zone de données (`data`) parfois incomplète
- la zone des instructions (`text`) parfois incomplète
- les informations associées à chaque symbole rangées dans une section appelée : table des symboles.

Que contient un fichier .o ?

- un entête contenant la taille du fichier, les adresses des différentes tables, la taille de la zone de données non initialisées (`bss`), etc.
- la zone de données (`data`) parfois incomplète
- la zone des instructions (`text`) parfois incomplète
- les informations associées à chaque symbole rangées dans une section appelée : table des symboles.
- les informations permettant de compléter ce qui n'a pu être calculé... On les appelle informations de translation et l'ensemble de ces informations est rangé dans une section particulière appelée table de translation.

Que contient un fichier .o ?

- un entête contenant la taille du fichier, les adresses des différentes tables, la taille de la zone de données non initialisées (`bss`), etc.
- la zone de données (`data`) parfois incomplète
- la zone des instructions (`text`) parfois incomplète
- les informations associées à chaque symbole rangées dans une section appelée : table des symboles.
- les informations permettant de compléter ce qui n'a pu être calculé... On les appelle informations de translation et l'ensemble de ces informations est rangé dans une section particulière appelée table de translation.
- une table des chaines à laquelle la table des symboles fait référence.

Exemple : essai.o, entête

On obtient l'entête avec la commande `arm-eabi-readelf -a essai.o`.

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 61 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            ARM
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          184 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           40 (bytes)
  Number of section headers:         9
  Section header string table index: 6
```

Exemple : essai.o, organisation des tables

On obtient l'entête avec la commande `arm-eabi-readelf -a essai.o`.

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 00002c 00  AX  0   0  1
  [ 2] .rel.text         REL             00000000 00033c 000018 08      7   1  4
  [ 3] .data             PROGBITS        00000000 000060 000008 00  WA  0   0  1
  [ 4] .bss              NOBITS          00000000 000068 000000 00  WA  0   0  1
  [ 5] .ARM.attributes   ARM_ATTRIBUTES  00000000 000068 000010 00      0   0  1
  [ 6] .shstrtab         STRTAB          00000000 000078 000040 00      0   0  1
  [ 7] .symtab           SYMTAB          00000000 000220 0000f0 10      8  12  4
  [ 8] .strtab           STRTAB          00000000 000310 000029 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Exemple : essai.o, zone data

On obtient la zone `.data` avec la commande `arm-eabi-objdump -s -j .data essai.o`.

```
essai.o:     format de fichier elf32-littlearm

Contenu de la section .data:
 0000 63000000 03000000
```

Exemple : essai.o, zone text

On obtient la zone `.text` avec la commande `arm-eabi-objdump -j .text -s essai.o`.

```
essai.o:     format de fichier elf32-littlearm

Contenu de la section .text:
 0000 0000a0e3 0a0050e3 0500000a 14309fe5
 0010 002093e5 feffffeb 002083e5 010080e2
 0020 f7ffffea feffffea 04000000
```

Exemple : essai.o, zone text

La zone `.text` avec désassemblage avec la commande `arm-eabi-objdump -j .text -d essai.o`.

```
Disassembly of section .text:

00000000 <main>:
   0: e3a00000  mov r0, #0

00000004 <bcle>:
   4: e350000a  cmp r0, #10
   8: 0a000005  beq 24 <fin>
   c: e59f3014  ldr r3, [pc, #20] ; 28 <adr_xx>
  10: e5932000  ldr r2, [r3]
  14: ebffffe  bl 0 <add1>
  18: e5832000  str r2, [r3]
  1c: e2800001  add r0, r0, #1
  20: eafffff7  b 4 <bcle>

00000024 <fin>:
  24: eafffffe  b 0 <exit>

00000028 <adr_xx>:
  28: 00000004  .word 0x00000004
```

Exemple : essai.o, table des symboles

On obtient l'entête avec la commande `arm-eabi-readelf -s essai.o`.

```
Symbol table '.symtab' contains 15 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 SECTION LOCAL  DEFAULT    1
     2: 00000000     0 SECTION LOCAL  DEFAULT    3
     3: 00000000     0 SECTION LOCAL  DEFAULT    4
     4: 00000000     0 NOTYPE  LOCAL  DEFAULT    1 $a
     5: 00000004     0 NOTYPE  LOCAL  DEFAULT    1 bcle
     6: 00000024     0 NOTYPE  LOCAL  DEFAULT    1 fin
     7: 00000028     0 NOTYPE  LOCAL  DEFAULT    1 adr_xx
     8: 00000028     0 NOTYPE  LOCAL  DEFAULT    1 $d
     9: 00000004     0 NOTYPE  LOCAL  DEFAULT    3 xx
    10: 00000000     0 NOTYPE  LOCAL  DEFAULT    3 $d
    11: 00000000     0 SECTION LOCAL  DEFAULT    5
    12: 00000000     0 NOTYPE  GLOBAL DEFAULT    1 main
    13: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND add1
    14: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND exit
```

Exemple : essai.o, table de translation

On obtient la table de translation avec la commande `arm-eabi-readelf -a essai.o`.

```
Relocation section '.rel.text' at offset 0x33c contains 3 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000014  00000d01 R_ARM_PC24        00000000   add1
00000024  00000e01 R_ARM_PC24        00000000   exit
00000028  00000202 R_ARM_ABS32       00000000   .data
```

Exemple : lib.o, organisation des tables

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 000008 00  AX  0   0  1
  [ 2] .data             PROGBITS        00000000 00003c 000000 00  WA  0   0  1
  [ 3] .bss              NOBITS          00000000 00003c 000000 00  WA  0   0  1
  [ 4] .ARM.attributes   ARM_ATTRIBUTES  00000000 00003c 000010 00      0   0  1
  [ 5] .shstrtab         STRTAB          00000000 00004c 00003c 00      0   0  1
  [ 6] .symtab           SYMTAB          00000000 0001c8 000070 10      7   6  4
  [ 7] .strtab           STRTAB          00000000 000238 000009 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Exemple : lib.o, tables des symboles

```
Symbol table '.symtab' contains 7 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 SECTION LOCAL  DEFAULT    1
     2: 00000000     0 SECTION LOCAL  DEFAULT    2
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 NOTYPE  LOCAL  DEFAULT    1 $a
     5: 00000000     0 SECTION LOCAL  DEFAULT    4
     6: 00000000     0 NOTYPE  GLOBAL DEFAULT    1 add1
```

Etapes d'un assembleur

1. Reconnaissance de la syntaxe (lexicographie et syntaxe) et repérage des symboles. Fabrication de la table des symboles utilisée par la suite dès qu'une référence à un symbole apparait.
2. Traduction = production du binaire.

Plan

Rôle de l'éditeur de liens

Le travail de l'éditeur de liens consiste à :

Rôle de l'éditeur de liens

Le travail de l'éditeur de liens consiste à :

- Identifier les symboles définis et exportés d'un côté et les symboles non définis de l'autre (importés).

Rôle de l'éditeur de liens

Le travail de l'éditeur de liens consiste à :

- Identifier les symboles définis et exportés d'un côté et les symboles non définis de l'autre (importés).
- Rassembler les zones de même type et effectuer les corrections nécessaires.

Rôle de l'éditeur de liens

Le travail de l'éditeur de liens consiste à :

- Identifier les symboles définis et exportés d'un côté et les symboles non définis de l'autre (importés).
- Rassembler les zones de même type et effectuer les corrections nécessaires.

Remarque : L'édition de liens rassemble des fichiers objets.

Rôle de l'éditeur de liens

Le travail de l'éditeur de liens consiste à :

- Identifier les symboles définis et exportés d'un côté et les symboles non définis de l'autre (importés).
- Rassembler les zones de même type et effectuer les corrections nécessaires.

Remarque : L'édition de liens rassemble des fichiers objets.
L'assembleur ne peut pas produire du binaire exécutable, il produit un binaire incomplet dans lequel il conserve des informations permettant de le compléter plus tard.
La phase d'édition de liens bien qu'elle permette de résoudre les problèmes de noms globaux produit elle aussi du binaire incomplet car les adresses d'implantation des zones `text` et `data` ne sont pas connues.

Phase de chargement : production de binaire exécutable

L'édition de liens peut être exécuter de façon statique ou de façon dynamique au moment ou on en a besoin.
Deux solutions dynamiques possibles :

Phase de chargement : production de binaire exécutable

L'édition de liens peut être exécuter de façon statique ou de façon dynamique au moment ou on en a besoin.
Deux solutions dynamiques possibles :

- édition de liens au moment du chargement en mémoire (au lieu de rassembler le contenu de plusieurs fichiers, on ne charge que le code des fonctions utilisées, par ex. pour les bibliothèques) ou

Phase de chargement : production de binaire exécutable

L'édition de liens peut être exécuter de façon statique ou de façon dynamique au moment ou on en a besoin.
Deux solutions dynamiques possibles :

- édition de liens au moment du chargement en mémoire (au lieu de rassembler le contenu de plusieurs fichiers, on ne charge que le code des fonctions utilisées, par ex. pour les bibliothèques) ou
- édition de liens au moment de l'exécution du code (appel de la fonction) ce qui permet de partager des fonctions et de ne pas charger en mémoire plusieurs fois le même code. (*e.g.*, les `DLLs` sous windows, *Dynamically Linked Libraries*)

Que contient un fichier exécutable ? entête

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 61 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            ARM
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x810c
  Start of program headers:          52 (bytes into file)
  Start of section headers:          144432 (bytes into file)
  Flags:                             0x2, has entry point, GNU EABI
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         24
  Section header string table index: 21
```

Que contient un fichier exécutable ? organisation des tables

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .init             PROGBITS        00008000 008000 000020 00  AX  0   0  4
  [ 2] .text             PROGBITS        00008020 008020 002500 00  AX  0   0  4
  [ 3] .fini             PROGBITS        0000a520 00a520 00001c 00  AX  0   0  4
  [ 4] .rodata           PROGBITS        0000a53c 00a53c 00000c 00   A  0   0  4
  [ 5] .eh_frame         PROGBITS        0000a548 00a548 00083c 00   A  0   0  4
  [ 6] .ctors            PROGBITS        00012d84 00ad84 000008 00  WA  0   0  4
  [ 7] .dtors            PROGBITS        00012d8c 00ad8c 000008 00  WA  0   0  4
  [ 8] .jcr              PROGBITS        00012d94 00ad94 000004 00  WA  0   0  4
  [ 9] .data             PROGBITS        00012d98 00ad98 00095c 00  WA  0   0  4
  [10] .bss              NOBITS          000136f4 00b6f4 000108 00  WA  0   0  4
  [11] .comment          PROGBITS        00000000 00b6f4 0001e6 00      0   0  1
  [12] .debug_aranges    PROGBITS        00000000 00b8e0 000350 00      0   0  8
  [13] .debug_pubnames   PROGBITS        00000000 00bc30 00069c 00      0   0  1
  [14] .debug_info       PROGBITS        00000000 00c2cc 00ea53 00      0   0  1
  [15] .debug_abbrev     PROGBITS        00000000 01ad1f 002956 00      0   0  1
  [16] .debug_line       PROGBITS        00000000 01d675 002444 00      0   0  1
  [17] .debug_str        PROGBITS        00000000 01fab9 001439 01  MS  0   0  1
  [18] .debug_loc        PROGBITS        00000000 020ef2 002163 00      0   0  1
  [19] .debug_ranges     PROGBITS        00000000 023058 0002e8 00      0   0  8
  [20] .ARM.attributes   ARM_ATTRIBUTES  00000000 023340 000010 00      0   0  1
  [21] .shstrtab         STRTAB          00000000 023350 0000df 00      0   0  1
  [22] .symtab           SYMTAB          00000000 0237f0 0013e0 10     23 213  4
  [23] .strtab           STRTAB          00000000 024bd0 0007d1 00      0   0  1
```

Que contient un fichier exécutable ? table des symboles

```
Symbol table '.symtab' contains 318 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00008000     0 SECTION LOCAL  DEFAULT    1
     2: 00008020     0 SECTION LOCAL  DEFAULT    2
     3: 0000a520     0 SECTION LOCAL  DEFAULT    3
 ...........................
     9: 00012ea8     0 SECTION LOCAL  DEFAULT    9
 ...........................
    74: 0000821c     0 NOTYPE  LOCAL  DEFAULT    2 bcle
    75: 0000823c     0 NOTYPE  LOCAL  DEFAULT    2 fin
    76: 00008240     0 NOTYPE  LOCAL  DEFAULT    2 adr_xx
    77: 00008240     0 NOTYPE  LOCAL  DEFAULT    2 $d
    78: 00012eb4     0 NOTYPE  LOCAL  DEFAULT    9 xx
 ...........................
   230: 00008244     0 NOTYPE  GLOBAL DEFAULT    2 add1
 ...........................
```

Que contient un fichier exécutable ? section data

```
Contents of section .data:
 ......................

  12ea8 00000000 00000000 63000000 03000000

 ......................
```

Que contient un fichier exécutable ? section text

```
00008218 <main>:
    8218: e3a00000  mov r0, #0

0000821c <bcle>:s
    821c: e350000a  cmp r0, #10
    8220: 0a000005  beq 823c <fin>
    8224: e59f3014  ldr r3, [pc, #20] ; 8240 <adr_xx>
    8228: e5932000  ldr r2, [r3]
    822c: eb000004  bl 8244 <add1>
    8230: e5832000  str r2, [r3]
    8234: e2800001  add r0, r0, #1
    8238: eafffff7  b 821c <bcle>

0000823c <fin>:
    823c: ea000007  b 8260 <exit>

00008240 <adr_xx>:
    8240: 00012eb4  .word 0x00012eb4

00008244 <add1>:
    8244: e2822001  add r2, r2, #1
    8248: e1a0f00e  mov pc, lr
```

Plan

Retour sur l'interprétation

Rappels : compilateurs et interpréteurs partagent la première étape de travail (étape d'analyse).

Retour sur l'interprétation

Rappels : compilateurs et interpréteurs partagent la première étape de travail (étape d'analyse).

Compilateurs et interpréteurs se distinguent principalement au moment de la seconde étape (étape de synthèse) et de l'exécution :

Retour sur l'interprétation

Rappels : compilateurs et interpréteurs partagent la première étape de travail (étape d'analyse).

Compilateurs et interpréteurs se distinguent principalement au moment de la seconde étape (étape de synthèse) et de l'exécution :

- le code cible produit par un compilateur est exécuté directement par la machine cible ;
  à chaque exécution, c'est le même code produit qui est utilisé

Retour sur l'interprétation

Rappels : compilateurs et interpréteurs partagent la première étape de travail (étape d'analyse).

Compilateurs et interpréteurs se distinguent principalement au moment de la seconde étape (étape de synthèse) et de l'exécution :

- le code cible produit par un compilateur est exécuté directement par la machine cible ;
  à chaque exécution, c'est le même code produit qui est utilisé
- la structure intermédiaire (interne ou traduite en langage de bas niveau) obtenue par l'interpréteur est exécutée "dynamiquement" par l'interpréteur (comme sur une machine virtuelle) ;
  à chaque exécution, la production du code intermédiaire, la mise en place d'une machine virtuelle et l'exécution sur cette machine virtuelle sont renouvelées

Entre compilateurs et interpréteurs ?

A mi-chemin entre compilateurs et interpréteurs, d'autres organisations sont possibles :

Entre compilateurs et interpréteurs ?

A mi-chemin entre compilateurs et interpréteurs, d'autres organisations sont possibles :

- l'interpréteur peut conserver la mémoire de la production du code intermédiaire ou de l'arbre syntaxique correspondant au programme et de la mise en place d'une machine virtuelle

Entre compilateurs et interpréteurs ?

A mi-chemin entre compilateurs et interpréteurs, d'autres organisations sont possibles :

- l'interpréteur peut conserver la mémoire de la production du code intermédiaire ou de l'arbre syntaxique correspondant au programme et de la mise en place d'une machine virtuelle
- certaines parties de code (bibliothèques, parties critiques, *etc.*) peuvent être compilées (statiquement ou dynamiquement), d'autres interprétés

Entre compilateurs et interpréteurs ?

A mi-chemin entre compilateurs et interpréteurs, d'autres organisations sont possibles :

- l'interpréteur peut conserver la mémoire de la production du code intermédiaire ou de l'arbre syntaxique correspondant au programme et de la mise en place d'une machine virtuelle
- certaines parties de code (bibliothèques, parties critiques, *etc.*) peuvent être compilées (statiquement ou dynamiquement), d'autres interprétés

Il y a une grande diversité d'organisations possibles intermédiaires.

Mode d'exécution des langages les plus utilisés

Parmi les langages les plus utilisés (*cf.* Tiobe Jan 18), lesquels peuvent-être compilés, lesquels peuvent-être interprétés :

Mode d'exécution des langages les plus utilisés

Parmi les langages les plus utilisés (*cf.* Tiobe Jan 18), lesquels peuvent-être compilés, lesquels peuvent-être interprétés :

- le langage `Java`
- le langage `C`
- le langage `C++`
- le langage `C#`
- le langage `Python`
- le langage `PHP`
- le langage `Javascript`

Mode d'exécution des langages les plus utilisés

Parmi les langages les plus utilisés (*cf.* Tiobe Jan 18), lesquels peuvent-être compilés, lesquels peuvent-être interprétés :

- le langage `Java`
- le langage `C`
- le langage `C++`
- le langage `C#`
- le langage `Python`
- le langage `PHP`
- le langage `Javascript`

D'autres langages « interprétés » importants : `SQL`, `PostScript`.

Avantages, inconvénients

Pourquoi, comment, choisir entre compilation et interprétation ?
Arguments classiques (sur les performances, la qualité du code, la productivité du programmeur) :

Avantages, inconvénients

Pourquoi, comment, choisir entre compilation et interprétation ?
Arguments classiques (sur les performances, la qualité du code, la productivité du programmeur) :

- l'exécution par interprétation est plus lente (surtout, si l'on ne compte pas le temps de compilation)

Avantages, inconvénients

Pourquoi, comment, choisir entre compilation et interprétation ?
Arguments classiques (sur les performances, la qualité du code, la productivité du programmeur) :

- l'exécution par interprétation est plus lente (surtout, si l'on ne compte pas le temps de compilation)
- l'exécution par interprétation facilite la mise au point (temps morts réduits, possibilité d'exécution de code incomplets, changement à chaud, débuggage dynamique, exécution interactive, *etc.*)

Avantages, inconvénients

Pourquoi, comment, choisir entre compilation et interprétation ?
Arguments classiques (sur les performances, la qualité du code, la productivité du programmeur) :

- l'exécution par interprétation est plus lente (surtout, si l'on ne compte pas le temps de compilation)
- l'exécution par interprétation facilite la mise au point (temps morts réduits, possibilité d'exécution de code incomplets, changement à chaud, débuggage dynamique, exécution interactive, *etc.*)
- l'exécution par interprétation facilite le portage (dans la mesure où il y a un interpréteur portable lui aussi ; pour la compilation, il faut non seulement un compilateur portable mais aussi une compilation [sans erreur] pour arriver au même résultat)

Avantages, inconvénients (suite)

Pourquoi, comment, choisir entre compilation et interprétation ?

Avantages, inconvénients (suite)

Pourquoi, comment, choisir entre compilation et interprétation ?

- l'exécution par compilation améliore la sécurité (les fonctions « `eval` » n'existent souvent que dans les formes interprétées d'exécution, et l'exécution de code dynamique est une source importante de trous de sécurité, *e.g.* injection de code)

Avantages, inconvénients (suite)

Pourquoi, comment, choisir entre compilation et interprétation ?

- l'exécution par compilation améliore la sécurité (les fonctions « `eval` » n'existent souvent que dans les formes interprétées d'exécution, et l'exécution de code dynamique est une source importante de trous de sécurité, *e.g.* injection de code)
- les langages orientés interprétation ont plus souvent un typage dynamique, une gestion dynamique de la mémoire, *etc.* souvent plus « haut niveau » (moins de sécurité, mais plus grande productivité du programmeur)

Avantages, inconvénients (suite)

Pourquoi, comment, choisir entre compilation et interprétation ?

- l'exécution par compilation améliore la sécurité (les fonctions « `eval` » n'existent souvent que dans les formes interprétées d'exécution, et l'exécution de code dynamique est une source importante de trous de sécurité, *e.g.* injection de code)
- les langages orientés interprétation ont plus souvent un typage dynamique, une gestion dynamique de la mémoire, *etc.* souvent plus « haut niveau » (moins de sécurité, mais plus grande productivité du programmeur)
- l'exécution par interprétation facilite les démarches « open source » (meilleur qualité de code, meilleur productivité, mais possible frein à la rentabilité)

Avantages, inconvénients (fin)

Pourquoi, comment, choisir entre compilation et interprétation ?

Avantages, inconvénients (fin)

Pourquoi, comment, choisir entre compilation et interprétation ?

- l'exécution par interprétation nécessite le code du programme et le code de l'interpréteur, chacun peut évoluer séparément.

Avantages, inconvénients (fin)

Pourquoi, comment, choisir entre compilation et interprétation ?

- l'exécution par interprétation nécessite le code du programme et le code de l'interpréteur, chacun peut évoluer séparément.
- au début de l'informatique, et chaque fois que l'informatique se déploie sur un nouveau dispositif aux ressources limitées, la compilation peut demander moins de ressource (sur le dispositif) et représenter la seule solution possible.

Le cas Java

Java est un langage semi-interprété :

Le cas Java

Java est un langage semi-interprété :

- la première phase de la vie d'un programme java `monProg.java` consiste en une ≪ compilation ≫(`javac`) produisant un code intermédiaire (le java bytecode) `monProg.class`

Le cas Java

Java est un langage semi-interprété :

- la première phase de la vie d'un programme java `monProg.java` consiste en une ≪ compilation ≫(`javac`) produisant un code intermédiaire (le java bytecode) `monProg.class`
- la seconde phase de la vie d'un programme java consiste en l'interprétation (`java`) du code intermédiaire `monProg.class` par la `JVM` (`Java Virtual Machine`)

Le cas Java

Java est un langage semi-interprété :

- la première phase de la vie d'un programme java `monProg.java` consiste en une ≪ compilation ≫(`javac`) produisant un code intermédiaire (le java bytecode) `monProg.class`
- la seconde phase de la vie d'un programme java consiste en l'interprétation (`java`) du code intermédiaire `monProg.class` par la `JVM` (`Java Virtual Machine`)

`Java` n'est pas tout seul (`Python`, `C#`, `ProLog`, `Lisp`, `Scheme`, `Ocaml`, `Perl`, `Ruby`, `Erlang`, `Lua`, *etc.*)

Remarque finale

A propos de bootstrap :

Remarque finale

A propos de bootstrap : un interpréteur demande toujours au départ un code exécutable (non interprété, même parfois produit « à la main », sans compilation)

Remarque finale

A propos de bootstrap : un interpréteur demande toujours au départ un code exécutable (non interprété, même parfois produit ≪ à la main ≫, sans compilation) jusqu'au moment où l'interpréteur est le code exécuté par lui-même, on parle alors d'**auto-interprétation**.