

Introduction

CS410 - Langages et Compilation

Julien Henry, Catherine Oriat

Grenoble-INP Esisar

2012-2013

Références

- *Compilation*, Catherine Oriat,
Année Spéciale Informatique, Grenoble-INP Ensimag

Summary

- 1 **Compilateur, Interprète**
- 2 Structure d'un compilateur

Qu'est ce que la compilation ?

Le développeur écrit un programme dans un langage lisible par l'Homme : C, C++, Java, etc.

La machine fonctionne avec des programmes *exécutables* : dans un langage lisible par l'ordinateur (binaire).

La phase de *compilation* transforme un programme écrit dans un langage "humain" en un programme exécutable par la machine.

Compilateur, Interprète

Un *compilateur* est un programme :

- Entrée : un programme P écrit dans le langage L
- Sortie : un programme P' écrit dans le langage L'

P et P' ont la même *sémantique* : pour toute entrée I , la sortie de P et la sortie de P' sont les même.

Exemple :

- un compilateur qui transforme un code C ou C++ en langage d'assemblage (gcc, clang, ...)
- un compilateur qui transforme un code Java en bytecode Java (javac)

Compilateur, Interprète

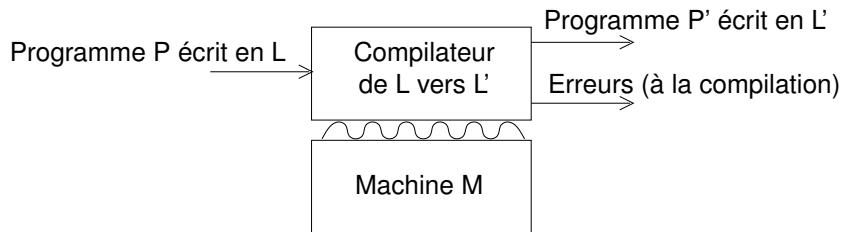


FIGURE: Compilation de P

Compilateur, Interprète

Un *interprète* est un programme :

- Entrée : un programme P écrit dans le langage L des données d'entrée I
- Sortie : le résultat de l'exécution de P avec les entrées I

Exemple : interprète de machine virtuelle Java, qui prend en entrée un programme en bytecode Java et ses entrées, et l'exécute.

Compilateur, Interprète

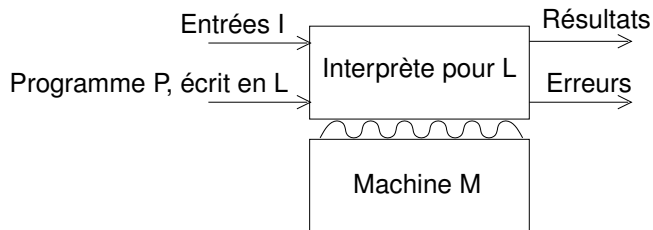


FIGURE: Interprétation de P avec les entrées I

Langages “Mixtes”

Certains langages sont à mi-chemin entre interprétation et compilation (ex : Java).

- Ils sont compilés en une représentation compacte non exécutable (ex : bytecode Java)
- Cette représentation est ensuite interprétée par une machine virtuelle

Rôle du compilateur

Pas seulement transformer un programme d'un langage vers un autre.
Mais aussi :

- Vérifier des propriétés sur le programme
- Lever des erreurs
- Optimiser le code

Propriétés statiques / dynamiques

- *statique* : propriété qui peut être déterminée lors de la phase de compilation, vraie pour toutes les exécutions possibles du programme.
- *dynamique* : propriété qui concerne uniquement une exécution donnée.

Propriétés statiques / dynamiques

- *statique* : propriété qui peut être déterminée lors de la phase de compilation, vraie pour toutes les exécutions possibles du programme.
- *dynamique* : propriété qui concerne uniquement une exécution donnée.

Exemple : l'expression $b * b - 4.0 * a * c$

Propriétés statiques / dynamiques

- *statique* : propriété qui peut être déterminée lors de la phase de compilation, vraie pour toutes les exécutions possibles du programme.
- *dynamique* : propriété qui concerne uniquement une exécution donnée.

Exemple : l'expression $b * b - 4.0 * a * c$

- Propriété statique : son typage
- Propriété dynamique : sa valeur

Erreurs d'un programme

Plusieurs types d'erreurs :

- erreurs statiques :
 - Erreurs lexicales : utilisation de caractères incorrect, etc.
 - Erreurs de syntaxes : oubli d'un ' ;', mauvaise disposition de '(' ou '}', etc.
 - Erreurs de contexte : utilisation d'une variable non déclarée, erreur de typage, etc.
- erreurs dynamiques : débordement d'une opération arithmétique, déréférencement de pointeur nul, ...

Le compilateur détecte les erreurs statiques.

Ce que l'on attend d'un compilateur

- ① Efficacité :
 - le compilateur doit si possible être rapide.
 - il doit produire un code qui s'exécutera rapidement.
- ② Correction : le programme original et le programme compilé doivent représenter le même calcul.

Summary

- 1 Compilateur, Interprète
- 2 Structure d'un compilateur**

Structure du compilateur

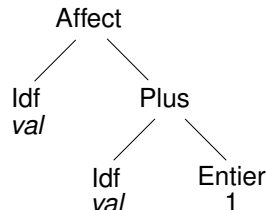
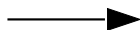
2 phases :

- ① Analyse :
 - construction d'une représentation structurée du programme
 - vérification des propriétés statiques
- ② Synthèse :
 - traduction de la représentation interne vers le langage cible

Représentation Intermédiaire : Arbre Abstrait

```
val = val + 1;
```

programme, sous forme d'une
suite de caractères



Arbre abstrait du programme

```
val = val + 1
```



Erreur de syntaxe

FIGURE: Construction de l'arbre abstrait

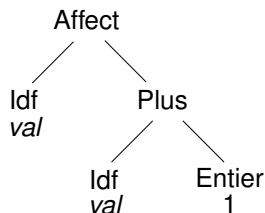
Analyse

Le programme à compiler est une suite de caractères.

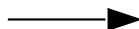
- Analyse lexicale : parcourt la suite de caractères et en extrait les unités lexicales (mots).
Soulève les *erreurs lexicales*.
- Analyse syntaxique : vérifie que la suite de mots vérifie bien la grammaire du langage, et construit une représentation interne et structurée du programme (*Arbre Abstrait*).
Soulève les *erreurs syntaxiques*.
- Analyse contextuelle : vérifie que le programme vérifie certaines propriétés statiques. Décore l'arbre abstrait avec des informations utiles pour l'optimisation / la génération du code cible.
Soulève les *erreurs contextuelles*.

Synthèse

Génération de Code : Génère un programme dans le langage cible, à partir de l'arbre abstrait décoré et optimisé.



Arbre abstrait du programme



```
LOAD #1, R0
ADD @val, R0
STORE R0, @val
```

@val : adresse de l'objet val
LOAD : chargement dans un registre
STORE : chargement à une adresse

FIGURE: Synthèse

Optimisations

Des optimisations du code peuvent avoir lieu à différentes étapes :

- Optimisation sur l'arbre abstrait
- Optimisation du code généré

Structure Modulaire

Les compilateurs modernes sont conçus de manière à pouvoir compiler plusieurs langages dans plusieurs langages cible.

- Front-End : un Front-end par langage source
 - analyse lexicale et syntaxique
 - construction d'une représentation intermédiaire
- “Coeur” : unique
 - analyse sémantique et optimisation sur la représentation intermédiaire (AST)
- Back-End : un Back-end par langage cible
 - sélection d'instructions
 - génération du code cible

Objectifs et Intérêts du cours

- ❶ Maîtriser les langages de programmation
 - vérifications
 - transformations du compilateurs
- ❷ Savoir concevoir un langage
- ❸ Avoir les outils théoriques pour écrire un compilateur

Application : Projet !

Outils théoriques

Théorie des langages, principalement :

- ❶ langages réguliers : pour décrire la lexicographie des langages de programmation.
- ❷ grammaires hors-contexte : pour décrire la syntaxe des langages de programmation.
- ❸ grammaires attribuées : pour faire des vérifications contextuelles sur le programme.