Compilation

I. Introduction

Jacques Farré

Jacques.Farre@unice.fr

http://deptinfo.unice.fr/~jf/Compil-L3/

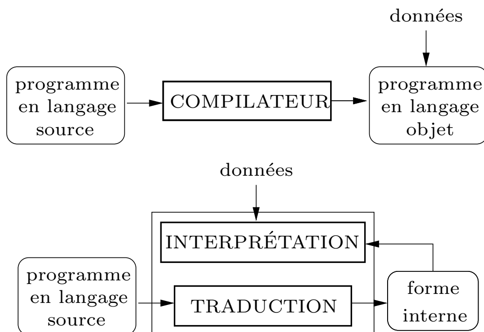
Introduction

- Compilation : traduction d'un langage dans un autre
  - pas seulement des langages de programmation : langages de script, d'interrogation, de description (LaTeX, XML . . .)
  - pas seulement pour obtenir un programme exécutable : enjoliveurs, éditeurs syntaxiques . . .
  - concepts et outils utilisés dans de nombreux autres domaines : web, bases de données, traitement des langues naturelles . . .
- Concepts et outils fondamentaux
  - théorie des langages (grammaires, automates)
  - sémantiques formelles
  - algorithmes et structures de données variés
  - principes de génie logiciel
- Vous rencontrerez inévitablement un jour ou l'autre un problème de compilation dans votre vie professionnelle

Terminologie

- *langage source* : celui qu'il faut analyser
- *langage objet* : celui vers lequel il faut traduire
- *langage machine* ou *d'assemblage* : langage machine sous forme symbolique
- traducteur : passage d'un langage à un autre, par exemple
  - *préprocesseur* : d'un sur-langage de L vers L (C par exemple)
  - *assembleur* : du langage d'assemblage vers le code machine binaire
  - *éditeur de liens* : d'un ensemble de sous-programmes en binaire *relogeable* (les `.o` de linux par exemple) vers un programme exécutable
- compilateur : traduction d'un langage source (généralement) de haut niveau vers un langage de plus bas niveau
- décompilateur : programme qui essaye de reconstruire le programme source à partir du code objet

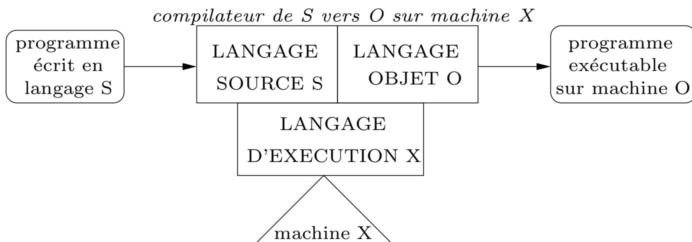
Compilateur et interprète



- avantages et inconvénients de chaque modèle
  - compilateur : produit du code machine → programme efficace mais pas toujours portable
  - interprète : programme portable mais moins efficace, mais qui peut manipuler le code source (méta-programmation)

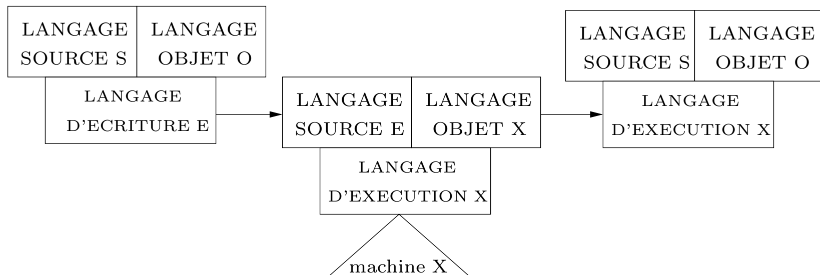
Compilation croisée

- Les traducteurs (compilateurs et interprètes) sont des programmes comme les autres
  - ils sont écrits dans un certain langage de programmation
  - la différence est qu'ils prennent des programmes comme données et produisent (compilateurs) du code exécutable
- compilateur de S vers un code O implanté sur une machine X : par exemple, compilateur Java sur un PC pour processeur embarqué (S = Java, O = code JVM, X = code intel core)



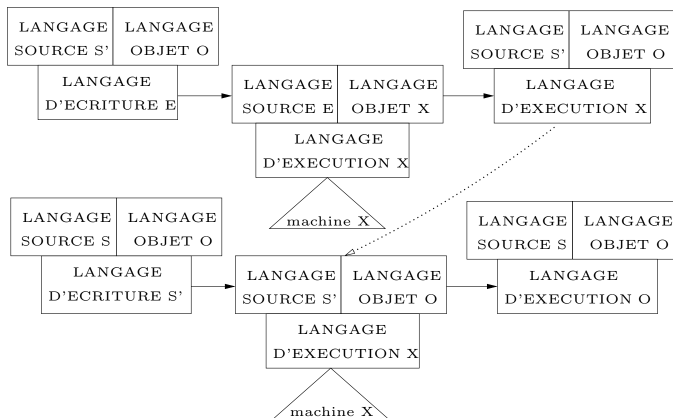
Comment obtenir un compilateur ?

- on l'écrit dans un langage E (de haut niveau si possible)
- on suppose qu'on dispose d'un compilateur de ce langage sur une machine X
- par exemple, un compilateur de Java écrit en C sur un PC :
  L = Java, O = JVM, E = langage C, X = code intel core



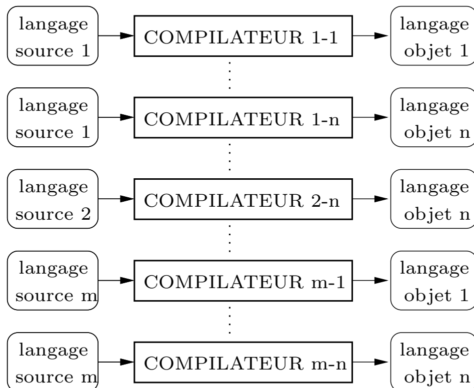
Comment obtenir un auto-compilateur ?

- si on veut écrire le compilateur dans son propre langage S
  - écrire d'abord un compilateur d'un sous-ensemble S' de S en E, et le compiler
  - écrire ensuite en S' le compilateur de S
  - c'est ce qu'on appelle un *bootstrap* du compilateur

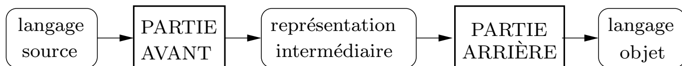


Compilateurs monolithiques

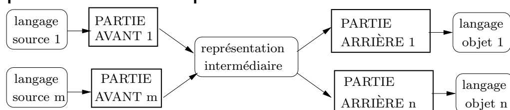
- les premiers compilateurs étaient monolitiques : ils pouvaient travailler en 1 seule passe (les langages étaient encore simples)
- autant de compilateurs que de couples (langage, machine cible), soit $m \times n$



Compilateurs modulaires

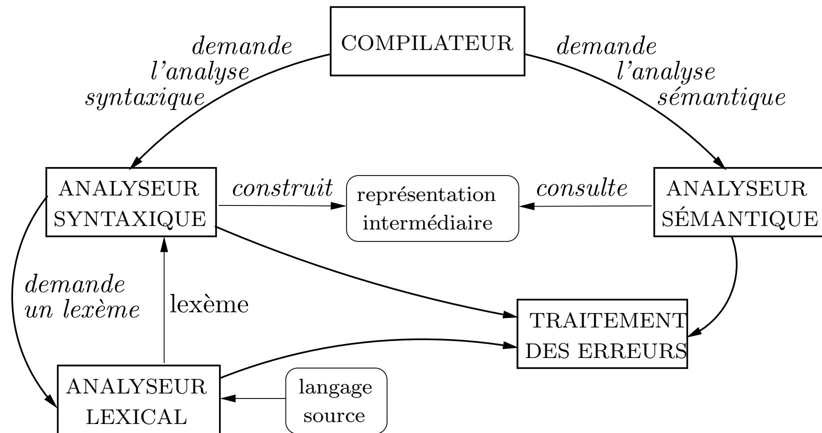


- partie avant (*analyse*) : analyses lexicale, syntaxique, sémantique (cours de L3)
- partie arrière (*synthèse*) : optimisation, production de code (cours de M1)
- avantages de cette décomposition : m parties avant $+$ n parties arrières permettent d'avoir $m \times n$ compilateurs



- le langage objet peut être celui d'une machine virtuelle (JVM ...) : le programme résultant sera portable
- on peut interpréter la représentation intermédiaire

Schéma synthétique de la partie avant



Justification de l'architecture de la partie avant

- du point du vue lexico-syntaxique
  - le programme source est plein de "bruits" (espaces inutiles, commentaires ...)
  - pour la grammaire, de nombreux symboles sont équivalents (identificateurs, nombres ...)
  - ce qui justifie le pré-traitement (en général au vol) du texte source par l'analyseur lexical
- du point de vue sémantique
  - existence de *références en avant* (utilisation d'un identificateur avant sa déclaration par exemple)
  - unification du traitement de constructions équivalentes (`attr=0` et `this.attr=0` par exemple) ou proches (boucles notamment)
  - ce qui justifie la mémorisation (sous forme intermédiaire) du texte à compiler

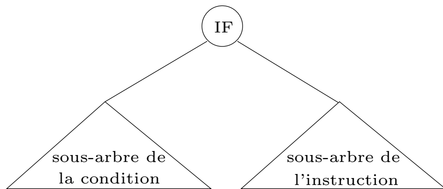
Concepts et structures de données utilisés

- analyse lexicale : langages réguliers, expressions régulières, automates finis pour l'essentiel, mais aussi tables d'adressage dispersé, arithmétique
- analyse syntaxique : grammaires hors-contexte, automates à pile (analyseurs descendants ou ascendants), attributs sémantiques
- analyse sémantique : diverses sémantiques formelles (mais l'analyse sémantique est souvent codée à la main), équations de type, table de symboles
- représentation intermédiaire : arbre ou graphe le plus généralement

Un exemple

Soit en Java : `if (i==0) --ifou ;`

- analyse lexicale : reconnaître les lexèmes : mot-clé `if` (mais pas dans l'identicateur `ifou`), reconnaître `i` et `ifou` comme lexèmes de la classe identificateur, `0` comme de la classe des nombres entiers, `--` comme un seul lexème, et pas la suite `- -`
- analyse syntaxique : analyser le texte selon la règle de grammaire : *instruction* → MOT-IF (*condition*) *instruction* et construire (par exemple) un sous-arbre comme



Un exemple (suite)

Soit en Java : `if (i==0) --ifou ;`

- analyse sémantique :
  - analyse de nom : vérifier que `i` et `ifou` sont bien déclarés, et déclarés comme des variables (ou des attributs) ; met en œuvre la table des symboles, qui permet de mémoriser les symboles déclarés (en Java, classes, attributs, méthodes, variables locales et paramètres)
  - analyse de type : vérifier que le type de `i` ou celui de `ifou` est compatible avec les opérations effectuées ; met en œuvre des équations de type du genre $int == int \rightarrow bool$
  - peut travailler en 1 ou plusieurs parcours de la représentation intermédiaire (selon la complexité du langage)
  - peut transformer/enrichir/normaliser la représentation intermédiaire
    par exemple, si `ifou` est un attribut, représenter de la même manière `--ifou` et `--this.ifou`

Propriétés d'un bon compilateur

- reconnaître tout le langage, et rien que le langage (ou alors avoir un mode permettant d'informer des constructions non standards)
  - sinon, on accepte des programmes non portables
- ne pas se restreindre à des constructions humainement lisibles : beaucoup de programmes sont créés par des programmes
- indiquer de façon claire les erreurs et éviter les messages d'erreur en cascade (bien que la source de l'erreur ne soit pas toujours évidente)
- traduire correctement vers le langage cible, et en donnant le meilleur code possible
- être raisonnablement rapide (utiliser des algorithmes quasi linéaires)

Bibliographie

- les bases
  - *Compilateurs*. Dick Grune , Henry E. Bal , Ceriel J.H. Jacobs , Koen G. Langendoen, Dunod 2002
  - *Compilateurs - Principes, techniques et outils - Avec plus de 200 exercices*. Alfred Aho, Monica Lam, Ravi Sethi et Jeffrey D. Ullman, Pearson Education, 2007
  - *Compiler Design*. Reinhard Wilhelm et Dieter Maurer, Adison-Wesley, 1995.
- pour les mordus de théorie
  - *The Theory of Parsing, Translation and Compiling* (2 volumes). Alfred V. Aho et Jeffrey D. Ullman, Prentice-Hall 1972
  - *Parsing Theory* (2 volumes). Seppo Sippu et Eljas Soisalon-Soininen, Springer-Verlag 1992