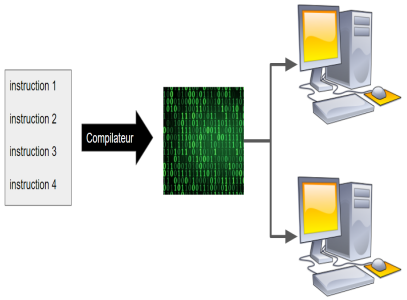


Introduction à la Compilation

Université Assane Seck
UFR Sciences et technologie
Département Informatique

Objectifs

Objectifs

Le but de ce cours est de présenter les principes de base inhérents à la réalisation de compilateurs. Les idées et techniques développées dans ce domaine sont si générales et fondamentales qu'un informaticien (et même un scientifique non informaticien) les utilisera

L'objectif général de ce cours est de permettre à l'étudiant de

- mieux comprendre les erreurs de compilation.
- mieux comprendre le fonctionnement d'un compilateur.

Les algorithmes et les outils de compilations sont repris dans d'autres domaines informatiques

- Traitement automatique des langages naturels
- Parser des documents (.XML).

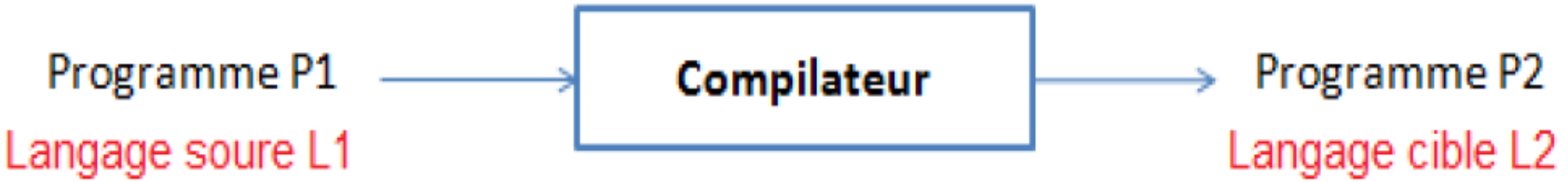
Analyse lexicale : Scanner

Analyse syntaxique : Parser

Introduction

Compilation

Le compilateur est un programme (fonction) qui prend en entrée un programme écrit dans un langage L1 et produit en sortie un programme équivalent écrit dans un langage L2.



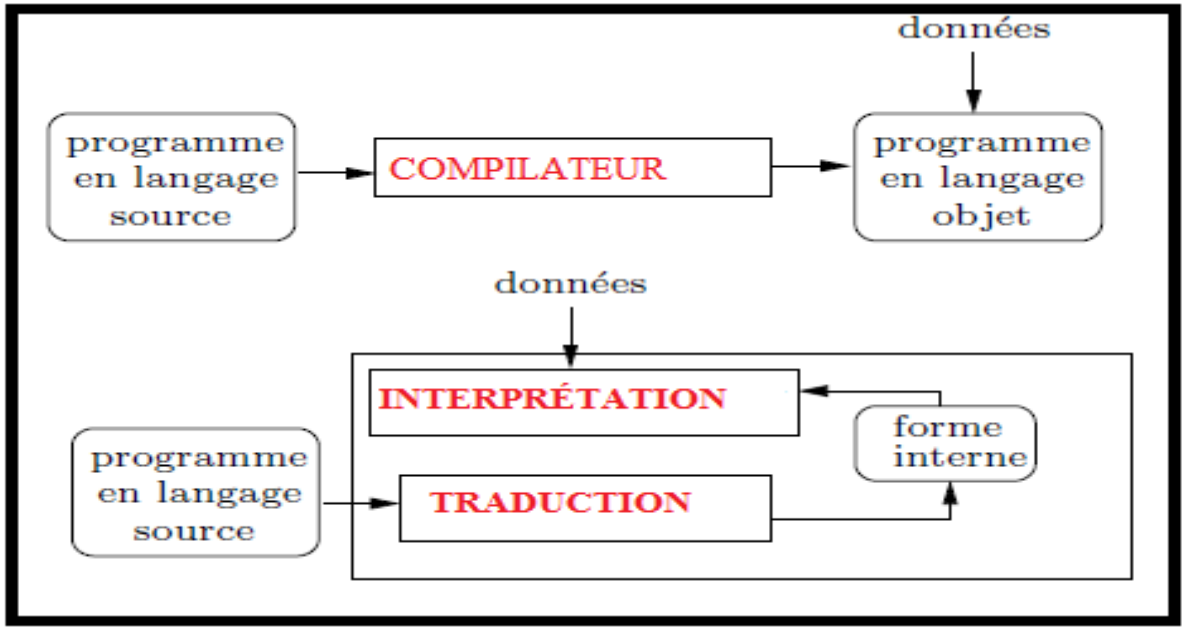
Introduction

- **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 : éditeurs syntaxiques . . .
  - concepts et outils utilisées dans de nombreux autres domaines : web, bases de données, traitement des langues naturelles . . .
- **Concepts et outils fondamentaux**
  - théorie des langages (grammaires, automates)
  - algorithmes et structures de données variés
  - principes de génie logiciel

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

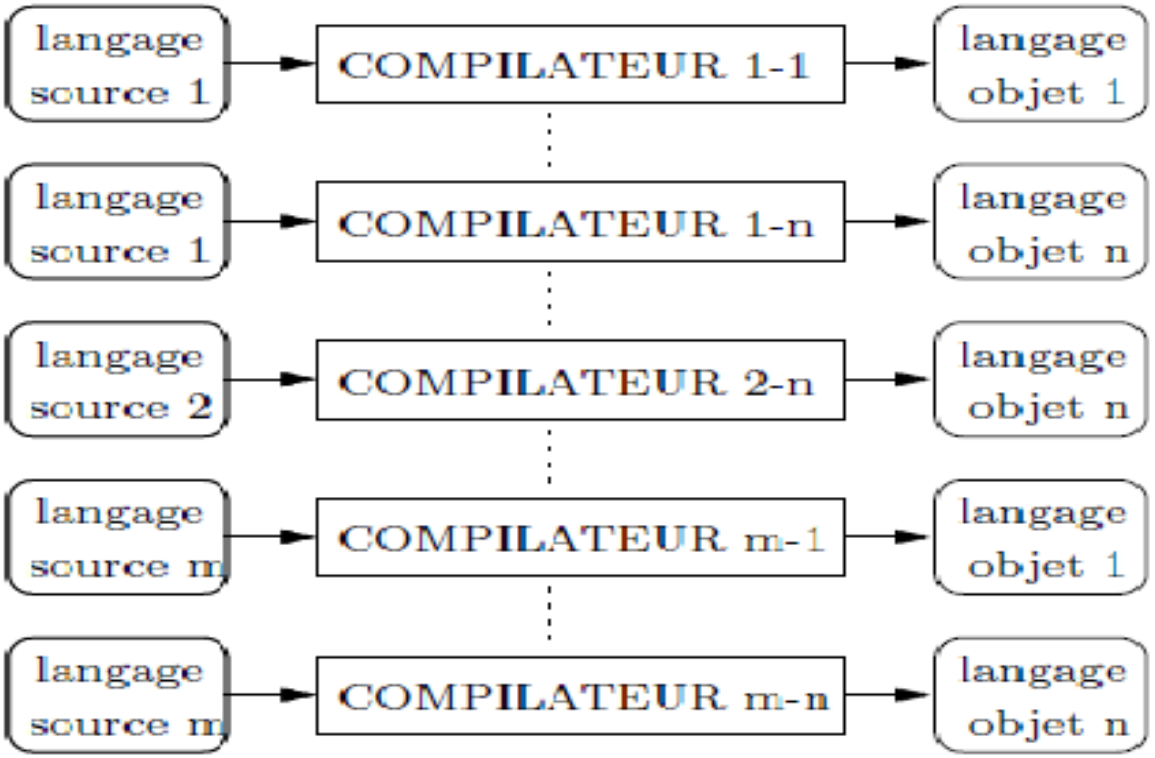
Compilateur vs Interprète



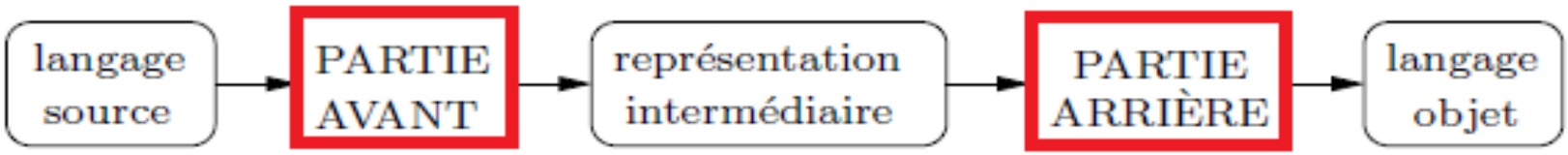
- Avantages et inconvénients de chaque modèle
  - compilateur : produit du code machine ! programme efficace mais pas toujours portable
  - interprète : programme portable mais moins efficace

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 *x* n



Compilateur modulaires



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

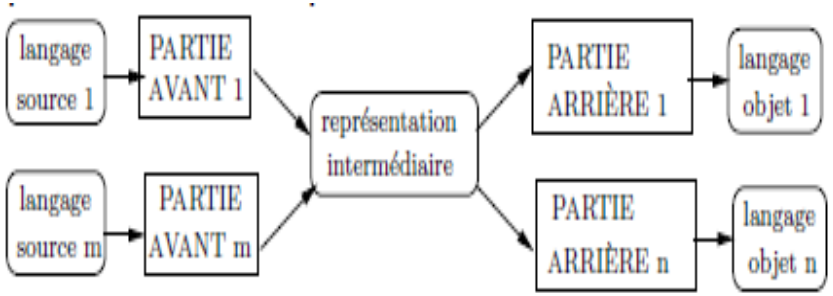
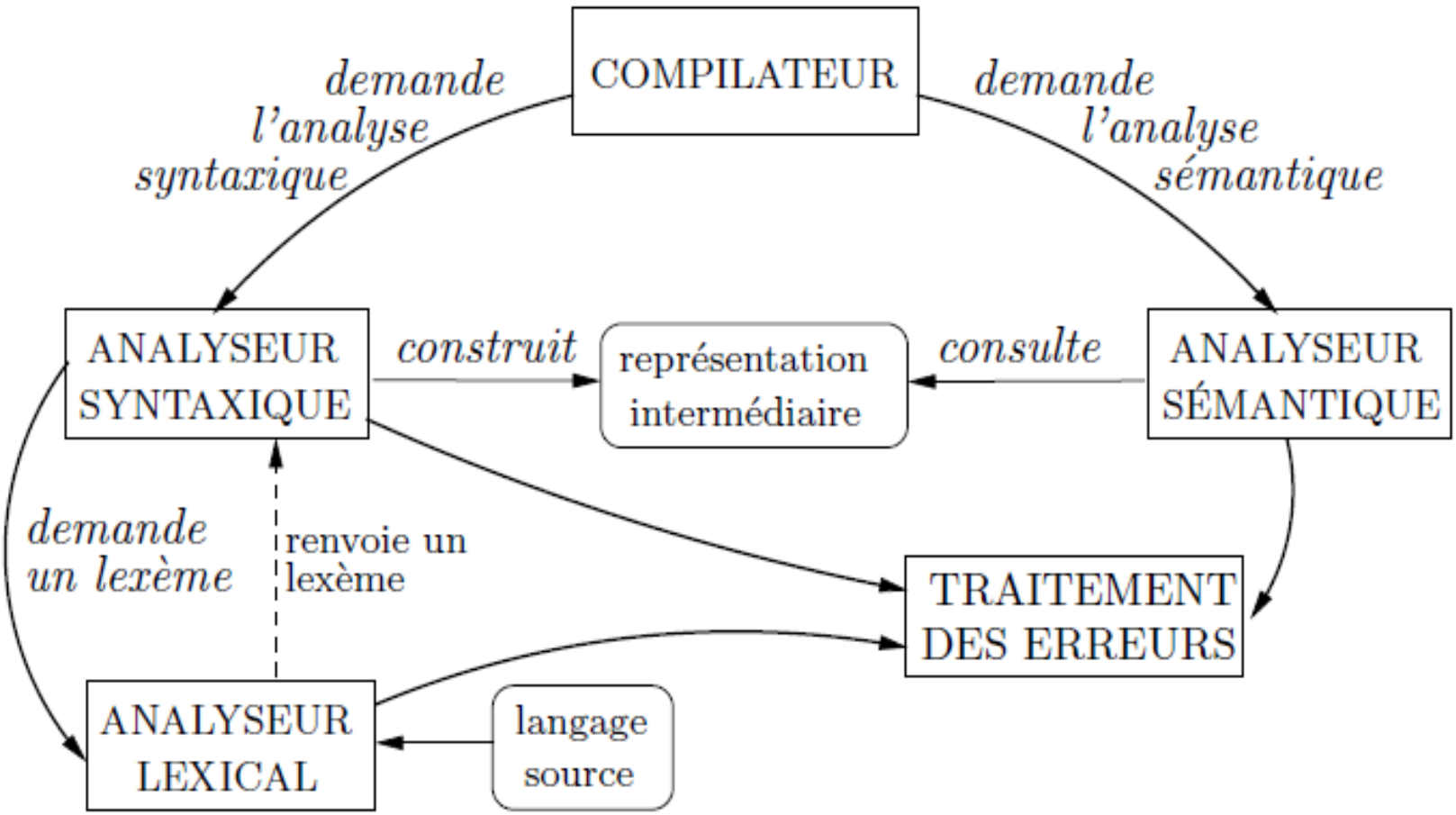


Schéma synthétique de la partie avant



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

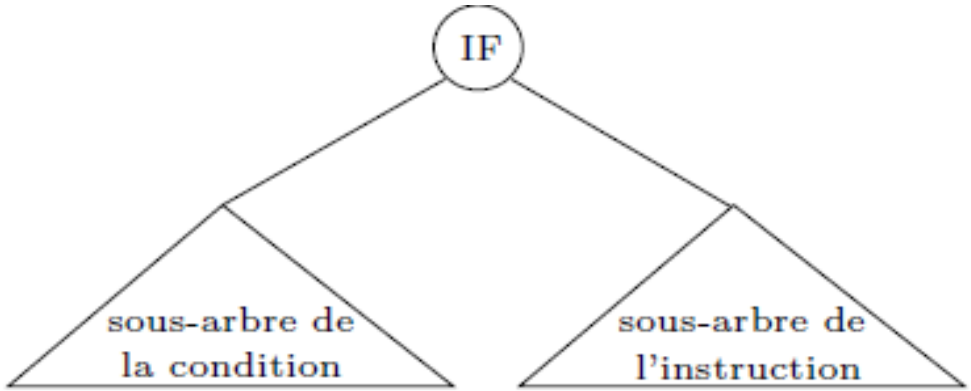
Concepts et structures de données utilisés

- **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-cléh 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 et représentation intermédiaire : analyser le texte selon la règle de grammaire :
  instruction ! MOT-IF (condition) instruction
  et construire (par exemple) un sous-arbre comme



Un exemple

- Analyse sémantique
  - analyse de nom : vérifier que i et ifou sont bien déclarés, et déclarés comme des variables (ou des attributs) ; met en oeuvre la table des symboles, qui permet de mémoriser les symboles déclarés (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 oeuvre des équations de type du genre int == int ! bool

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)
- Indiquer de fa¸con 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