# Chapitre 01 : Introduction

La compilation est une transformation que l'on fait subir à un programme écrit dans un langage évoluée pour le rendre exécutable. Fondamentalement, c'est une traduction : un texte écrit en Pascal, C, Java, etc., exprime un algorithme et il s'agit de produire un autre texte, spécifiant le même algorithme dans le langage d'une machine que nous cherchons à programmer.

Une autre phase importante qui intervient après la compilation pour obtenir un exécutable est la phase d'éditions de liens. Un éditeur de liens résout entre autres les références à des appels de routines dont le code est conservé dans des librairies. En général, un compilateur comprend une partie éditeur de liens.

La "compilation" n'est pas limitée à la traduction d'un programme informatique écrit dans un langage de haut niveau en un programme directement exécutable par une machine, cela peut aussi être : la traduction d'un langage de programmation de haut niveau vers un autre langage de programmation de haut niveau. Par exemple une traduction de Pascal vers C, ou de Java vers C++,... etc. On parle plutôt de **traducteur** c-à-d La traduction d'un langage quelconque vers un autre langage quelconque (ie pas forcement de programmation) : word vers html, pdf vers ps, ...

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 les utilisera très souvent au cours de sa carrière : traitement de données, moteurs de recherche, etc. ..

Comprendre comment est écrit un compilateur permet de mieux comprendre les "contraintes" imposées par les différents langages lorsque l'on écrit un programme dans un langage de haut niveau.

## Compilateur et interprète

Un compilateur est un programme de traduction automatique d'un programme écrit dans un langage source en un programme écrit dans un langage cible.

**Exemples :**

Pascal, C, C++, ADA, Fortran, Cobol

Au lieu de produire un programme cible comme dans le cas d'un compilateur, un interprète exécute lui même au fur et à mesure les opérations spécifiées par le programme source. Il analyse une instruction après l'autre puis l'exécute immédiatement. A l'inverse d'un compilateur, il travaille simultanément sur le programme et sur les données. Généralement les interpréteurs sont assez petits. L'interpréteur doit être présent sur le système à chaque fois que le programme est exécuté, ce qui n'est pas le cas avec un compilateur. Autre inconvénient : on ne peut pas cacher le code, toute personne ayant accès au programme peut le consulter et le modifier comme il le veut. Par contre, les langages interprétés sont souvent plus simples à utiliser et tolèrent plus d'erreurs de codage que les langages compilés.

**Exemples:**

BASIC, CaML, Tcl, LISP, Perl, Prolog…

Il existe des langages qui sont à mi-chemin de l'interprétation et de la compilation. On les appelle langages P-code ou langages intermédiaires. Le code source est traduit (compilé) dans une forme binaire compacte (du pseudo-code ou p-code) qui n'est pas encore du code machine. Lorsque l'on exécute le programme, ce P-code est interprété. Par exemple en Java, la source est compilée pour obtenir un fichier (.class) "byte code" qui sera interprété par une machine virtuelle. Autre langage p-code : Python.

Les interpréteurs de p-code peuvent être relativement petits et rapides, si bien que le p-code peut s'exécuter presque aussi rapidement que du binaire compilé. En outre les langages p-code peuvent garder la flexibilité et la puissance des langages interprétés.

# Structure d'un compilateur

La compilation se décompose en deux phases :

- Une phase d'analyses, qui va reconnaître les variables, les instructions, les opérateurs et élaborer la structure syntaxique du programme ainsi que certaines propriétés sémantiques
- Une phase de synthèse et de production qui devra produire le code cible:

**Une phase d'analyses**

1. **Analyse lexicale** (appelée aussi Analyse linéaire)

Dans cette étape, il s'agit de reconnaître les "types" des "mots" lus. Pour cela, on lit le programme source de gauche à droite et les caractères sont regroupés en unités lexicales. L'analyse lexicale se charge de

- éliminer les caractères superflus (commentaires, espaces, ...)
- identifier les parties du texte qui ne font pas partie à proprement parler du programme mais sont des directives pour le compilateur
- identifier les symboles qui représentent des identificateurs, des constantes réelles, entière, chaînes de caractères, des opérateurs (affectation, addition, ...), des séparateurs (parenthèses, points virgules, ...), les mots clefs du langage, ... C'est cela que l'on appelle des unités lexicales.

Outils théoriques utilisés : **expressions régulières** et **automates à états finis**

2. **Analyse syntaxique** (appelée aussi Analyse hiérarchique ou Analyse grammaticale)

Il s'agit de regrouper les unités lexicales en structures grammaticales, de découvrir la structure du programme. L'analyseur syntaxique sait comment doivent être construites les expressions, les instructions, les déclarations de variables, les appels de fonctions, ...

Outils théoriques utilisés : grammaires et automates à pile

3. **Analyse sémantique** (appelée aussi analyse contextuelle)

Dans cette phase, on opère certains contrôles (contrôles de type, par exemple) afin de vérifier que l'assemblage des constituants du programme a un sens. Outil théorique utilisé : schéma de traduction dirigée par la syntaxe

- les identificateurs apparaissant dans les expressions ont-ils été déclarés ?
- les opérandes ont-ils les types requis par les opérateurs ?
- les opérandes sont-ils compatibles ? n'y a-t-il pas des conversions à insérer ?
- les arguments des appels de fonctions ont-ils le nombre et le type requis ?
- etc.

**Une phase de synthèse**

4. **Génération du code intermédiaire** Il s'agit de produire les instructions en langage cible.

En règle générale, le programmeur dispose d'un calculateur concret (cartes équipées de processeurs, puces de mémoire, ...). Le langage cible est dans ce cas défini par le type de processeur utilisé.

Mais si l'on écrit un compilateur pour un processeur donné, il n'est alors pas évident de porter ce compilateur (ce programme) sur une autre machine cible. C'est pourquoi on introduit des machines dites abstraites qui font abstraction des architectures réelles existantes. Ainsi, on s'attache plus aux principes de traduction, aux concepts des langages, qu'à l'architecture des machines.

En général, on produira dans un premier temps des instructions pour une machine abstraite (virtuelle). Puis ensuite on fera la traduction de ces instructions en des instructions directement exécutables par la machine réelle sur laquelle on veut que le compilateur s'exécute. Ainsi, le portage du compilateur sera facilité, car la traduction en code cible virtuel sera faite une fois pour toutes, indépendamment de la machine cible réelle. Il ne reste plus ensuite qu'à étudier les problèmes spécifiques à la machine cible, et non plus les problèmes de reconnaissance du programme (cf Java)..

5. **Optimisation du code** Cette phase tente d'améliorer le code produit de telle sorte que le programme résultant soit plus rapide. Par exemple
   - détecter l'inutilité de recalculer des expressions dont la valeur est déjà connue,
   - transporter à l'extérieur des boucles des expressions et sous-expressions dont les opérandes ont la même valeur à toutes les itérations
   - détecter, et supprimer, les expressions inutiles
   - etc.
6. **Génération du code machine.** Cette phase nécessite la connaissance de la machine cible (réelle, virtuelle ou abstraite), et notamment de ses possibilités en matière de registres, piles, etc.

**Phase parallèle**

7. **Gestion de la table des symboles** La table des symboles est la structure de données utilisée servant à stocker les informations qui concernent les identificateurs du programme source (par exemple leur type, leur emplacement mémoire, leur portée, visibilité, nombre et type et mode de passage des paramètres d'une fonction, ...)

   Le remplissage de cette table a lieu lors des phases d'analyse. Les informations contenues dans la table sont nécessaires lors des analyses syntaxique et sémantique, ainsi que lors de la génération de code.
8. **Gestion des erreurs** Chaque phase peut rencontrer des erreurs. Cependant, après avoir détecté une erreur, une phase doit la traiter de telle façon que la compilation puisse continuer et que d'autres erreurs dans le programme source puissent être détectées. Un compilateur qui s'arrête à la première erreur n'est pas non plus très performant. Bien sûr, il y a des limites à ne pas dépasser et certaines erreurs (ou un trop grand nombre d'erreurs) peuvent entrainer l'arrêt de l'exécution du compilateur.

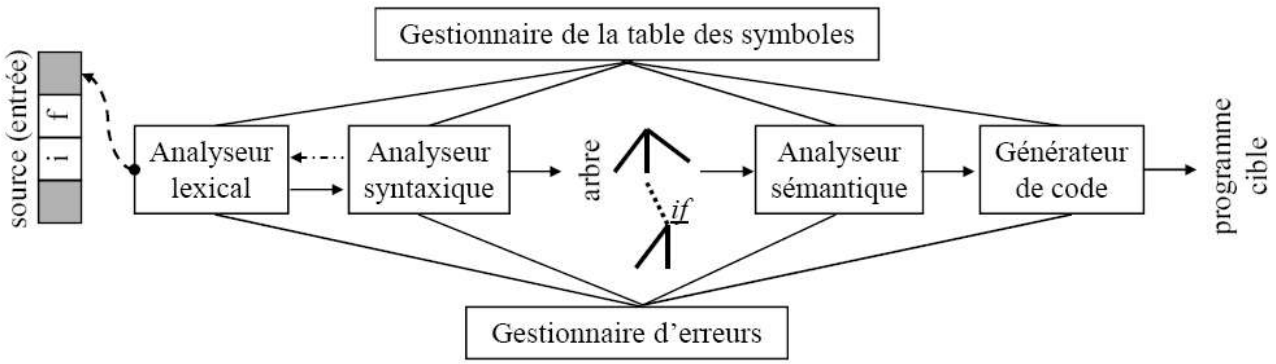


**Figure 01:** les étapes de la compilation