8. Introduction à la Compilation

Bernard ESPINASSE

Professeur à l'Université d'Aix-Marseille

1998

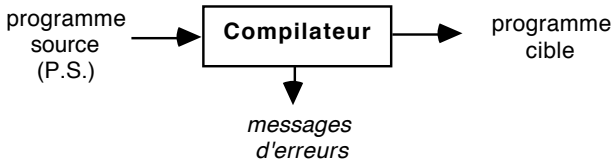
- **Introduction**
- **Exécution d'un programme - notion de traducteur**
- **Compilateur / Assembleur / Interpréteur / ...**
- **Les phases de la compilation**
- **Analyses lexicale, syntaxique, sémantique**
- **Génération et optimisation du code**

Introduction

Premiers compilateurs : **réalisation très difficile** (1°compilateur Fortran : 18 homme-années de travail - Backus &al.57).

De façon générale :

- **compilateur** = pgm **traduisant** un **pgm source (P.S.)** écrit dans un **langage source (L.S.)**, en un **pgm cible** équivalent écrit dans un **langage cible**
- lors de cette traduction, il doit signaler la présence **d'erreurs** dans le PS.



Importante variété de langages source:

- langages généraux (Fortran, Pascal, C, ...), langages spécialisés

Importante variété de langages cible:

- autre langage de programmation ou langage machine exécutable par un microprocesseur (ou un super calculateur).

Compilateurs

- Selon comment ils **fonctionnent** ou selon leur **finalité**:
  - compilateur **une passe**, **multi-passes**,
  - **compilateur-exécuteur**,
  - compilateur **de mise au point**,
  - compilateur **optimiseur**, ...
- Cependant tous les compilateurs doivent effectuer les **mêmes tâches** de base
- aussi, découverte :
  - de **techniques systématiques** pour traiter la plupart des tâches de base
  - de **langage d'implantation** (**LEX**) et
  - d'**environnement adaptés** au développement (**YACC**)

*.... qui ont rendu la **réalisation de compilateur bien plus facile**.*

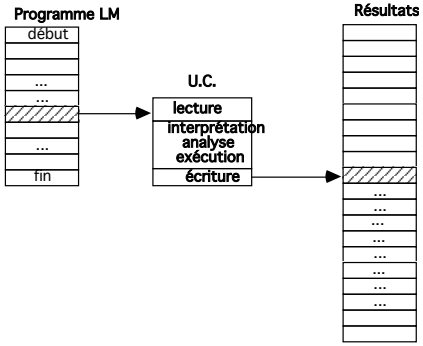
Exécution d'un programme

- Langage de base d'un ordinateur = **langage machine** (**LM**) défini sur l'alphabet **{0, 1}**
- En général toute instruction du **LM** (séquence de 0 et de 1) = code opération suivi de trois adresses d'opérandes : **(Cop, opérand_1, opérand_2, opérand_3)**
- Tout pgm en LM peut être **directement exécuté** par l'ordinateur, l'U.C. réalise pour chaque instruction du pgm :

a • sa **lecture**

b • son **interprétation** :
- **analyse** et **décodification** de l'instruction
- son **exécution** par activation des blocs de calcul selon cette analyse

c • **écriture** des résultats intermédiaires / définitifs.



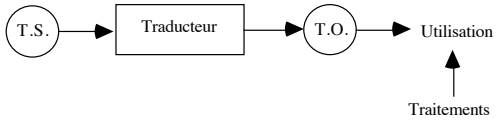
- "**Temps d'exécution**" (**Run-Time**) d'un pgm = durée d'exécution de ce pgm (durée d'immobilisation de l'UC)

Notion de traducteur

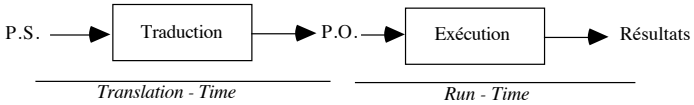
Si pgm écrit dans langage de prog. différent du LM => ne peut être exécuté par l'UC

*=> nécessaire de **traduire** ce programme en **LM**.*

- **Traducteur** = pgm (pour simplifier écrit en LM) qui traduit un **Texte source (TS)** en un texte équivalent appelé **Texte objet (TO)**.



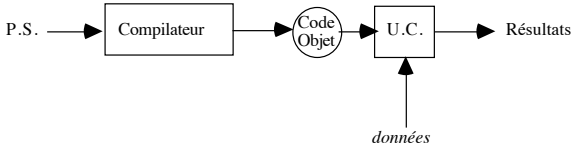
- **"Temps de traduction" (Translation-Time)** = durée passée à cette traduction



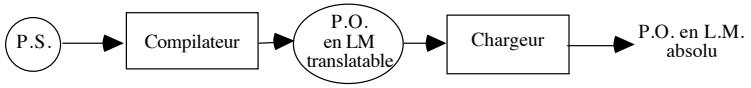
- les 2 phases **traduction** et **exécution** peuvent être imbriquées (**interpréteur**)

Les compilateurs

- **Compilateur d'un langage de prog.** en L.S. = **traducteur** qui substitue à tout pgm écrit dans ce langage, un pgm écrit **en langage machine** (LM).
- pgm objet obtenu appelé **Code-objet** (Object-code).



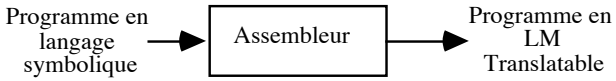
- on suppose ici **code objet exprimé en LM absolu**, c.a.d. adresses des opérandes calculées pour être directement interprétables par l'U.C.
- pour meilleure utilisation du système d'exploitation des ordinateurs, **code objet en LM translatable** : adresses des opérandes déterminées relativement à la tête du pgm.
- le **chargeur** a pour objet d'implanter de manière absolue le programme objet en mémoire centrale.



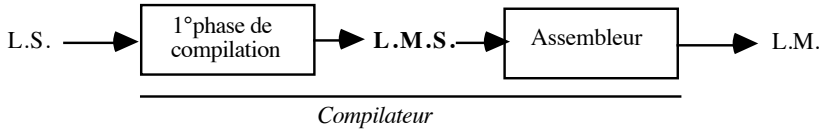
Les Assembleurs

- **Assembleur** = compilateur particulier qui **traduit** tout texte écrit en **LM symbolique** en un texte écrit en **L.M. translatable**

- **LM symbolique** = langage proche du LM dans lequel les différents éléments de la phrase LM sont représentés à l'aide de codes mnémotechniques exprimés dans l'alphabet latin.

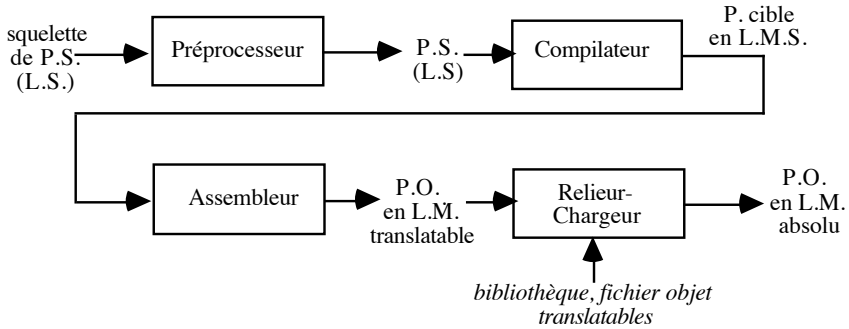


- Certains **compilateurs** utilisent **l'assembleur** pour générer le **code-objet :**



Environnement d'un compilateur

- **création d'un programme exécutable** peut nécessiter l'usage d'**autres programmes** :



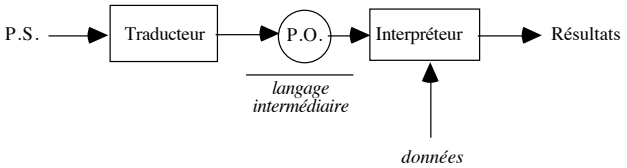
- un **PS** peut être **fractionné** en modules (fichiers distincts) : **reconstitution du PS** peut être confié à un pgm spécifique appelé **"préprocesseur" ou "précompilateur"**.

- le pgm cible créé par compilateur peut nécessiter pour être exécutable passage par un **assembleur** et ensuite le **reliage** (link) avec des routines de bibliothèque

Les interpréteurs

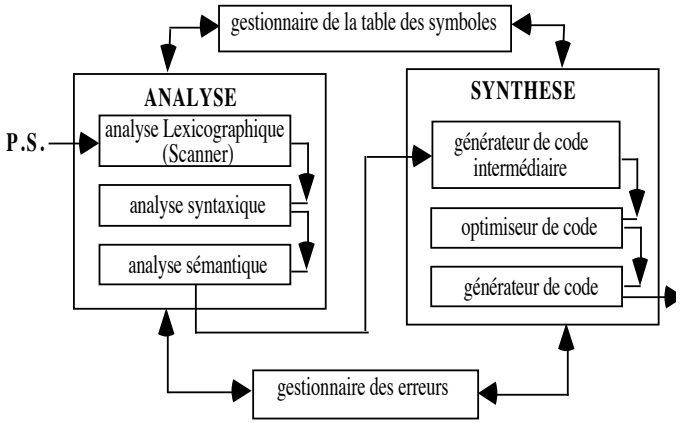
- Dans un PS existent des éléments dont définition et/ou analyse dépend du mode d'exploitation du compilateur et/ou de l'exécution du pgm: **traduction directe en LM pas possible** :
  - **compilateur conversationnel** : toute l'information nécessaire pour la génération du code machine pas dans PS
  - **sous-programme récursif** : si nombre d'appel dépend de son exécution, demande place mémoire dont la taille connue lors de l'exécution
  - **structures de données complexes et leurs opérations associées** : nécessitent des réservations dynamiques de place mémoire, ou des définitions dynamiques de composition d'opérations, etc

    => ***traduction du texte source dans un langage intermédiaire différent du L.M.***
- L'exécution du P.O. intermédiaire pas réalisée par l'U.C, mais par autre programme = **l'interpréteur** :



Les phases de la compilation

- Un compilateur doit réaliser **2 phases essentielles** : **ANALYSE & SYNTHESE**



- il a besoin des informations locales du programme source : il construit durant la phase **d'analyse** différentes **tables d'information ou des symboles**.

Analyse lexicographique - *Scanner*

- permet de **reconnaître**, **d'organiser** et de **coder** les **unités syntaxiques**, à partir des caractères et des mots définissant le programme source :
  - **mots particuliers du langage** - ***mots-clés*** - (BEGIN, MOVE, IF, WHILE, ...)
  - **mots créés par le programmeur** - ***identificateurs*** - (variables, étiquettes,...
  - **symboles** (opérateurs, séparateurs,...)
  - **constantes arithmétiques et logiques**

Exemple : soit l'instruction

```
position := initiale + vitesse * 60
```

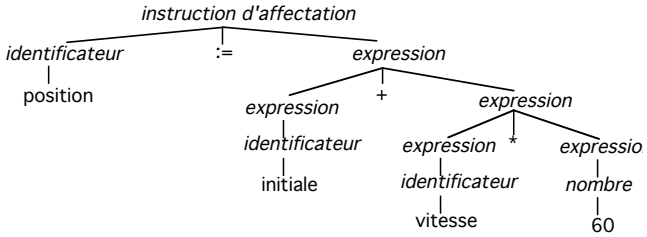
- Après analyse lexicale, **unités lexicales** associées à cette instruction d'affectation :
  - 1.L'identificateur **`position`**;
  - 2.Le symbole d'affectation **`:=;`**
  - 3.L'identificateur **`initiale`**;
  - 4.Le signe plus **`"+"`**;
  - 5.L'identificateur **`vitesse`**;
  - 6.Le signe de multiplication **`"*"`**;
  - 7.Le nombre **`60`**

Analyse syntaxique (ou hiérarchique ou grammaticale) - *Parser*

- **Regrouper les unités lexicales** du programme source en **structures grammaticales** qui seront employées par le compilateur pour la **synthèse**
- Ces structures grammaticales sont **représentées par un arbre syntaxique** :

Exemple : Arbre syntaxique pour l'instruction

```
position := initiale + vitesse * 60
```



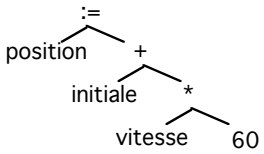
- la structure hiérarchique d'un programme est généralement exprimée par des règles récursives
- à toute valeur lexicale est associée un identificateur **id** pointant vers la **table des symboles** :

```
id1 := id2 + id3 * 60
```

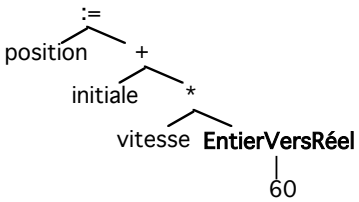
Analyse sémantique

- **Opère** des **contrôles** : s'assurer que l'**assemblage des constituants du programme a un sens**
  - **contrôle de type** : vérifier que les opérandes de chaque opérateur sont conformes aux spécifications du L.S.
  - ...
- Utilise la **structure hiérarchique** définie par l'analyse syntaxique pour identifier opérateurs et opérandes des expressions, ainsi que les instructions.

Exemple : analyse sémantique insérant une **conversion d'entier en réel**



=> convertir l'entier en un réel



Supposons tous les identificateurs déclarés comme **réels**, et que **60** soit a priori un entier.
Le contrôle de type révèle que "*" a pour opérandes un réel, `vitesse`, et un entier, `60`

=> Création d'un noeud supplémentaire pour l'**opérateur EntierVersRéel**, qui convertit explicitement un entier en un réel

Analyse

position := initiale + vitesse * 60

↓

analyse lexicale

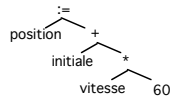
↓

id := id2 + id3 * 6o

↓

analyse syntaxique

↓



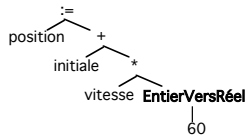
↓

analyse sémantique

↓

Table de symboles

1	position	...
2	initiale	...
3	vitesse	...
...		



↓

Synthèse

analyse

↓

générateur de code intermédiaire

↓

```
temp1 := EntierVersRéel(60)
temp2 := idS * temp1
temp3 := id2 + temp2
id1 := temp3
```

↓

optimiseur de code

↓

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

↓

générateur de code

↓

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```