Cours de compilation
1ère partie
Généralités

Jacques Ferber

1. Historique

Au début de l'informatique, on programmait directement les ordinateurs en langage machine. Cela s'est vite avéré fastidieux. On a très vite essayé d'utiliser les possibilités de l'informatique pour faciliter le travail de programmation.

En premier lieu on a donné des informations symboliques, ce que l'on appelle aussi des mnémoniques, aux instructions machines. Cela a été l'assembleur. Par exemple on a codé MOV A0, A1 une instruction qui était notée auparavant 0A43 en hexadécimal.

Ensuite on a utilisé des étiquettes pour exprimer les branchements. Cela a donné des instructions de la forme :

```
BRANCH boucle ;
```

à la place d'une instruction de la forme

```
BRANCH -24
```

où -24 est le déplacement qu'il faut effectuer dans le programme. Enfin, on a exprimé directement des places mémoires en leur donnant un nom symbolique à la place de l'adresse en hexadécimal.

L'étape suivante a consisté à s'affranchir complètement de la machine en élaborant ce que l'on appelle des langages de haut niveau. Ces langages ont introduit un certaine nombre de constructions qui n'existent pas dans le langage de la machine : expressions arithmétiques, variables locales, procédures et fonctions avec des paramètres qui retournent un résultat, structures de données (tableaux, énumération, record, objet,..), etc.

Pour cela il a fallu construire des programmes qui traduisent des énoncés exprimés dans le langage de haut niveau utilisé par les programmeurs, ce que l'on appelle le **langage source**, en instructions pour la **machine cible**. Les programmes effectuant ce genre d'opération s'appellent des **compilateurs** (ou des **interprètes**, nous verrons la différence par la suite, mais pour l'instant elle n'est pas fondamentale).

2. La compilation comme traduction

Un programme est la manifestation d'une pensée qui s'exprime sous la forme d'un texte. Ce texte désigne des opérations qui doivent être réalisées par une machine qui exécute ainsi le texte. Comme le dit J.F. Perrot dans son cours de compilation de Paris 6, « un ordinateur est une machine à convertir la pensée en actions ».

Un texte s'exprime dans un langage. Si ce langage est compris de la machine, cela signifie que la machine peut exécuter directement le texte qui lui est proposé. Dans le cas contraire, il est nécessaire de traduire le texte initial, le **texte source**, en un texte acceptable par la machine, le **texte cible**. L'opération conduisant à cette traduction s'appelle compilation. Le compilateur, qui est un programme est lui aussi écrit dans un certain langage. L'opération de compilation est bien une **traduction**, car elle doit conserver le sens initial du texte source.

Si on note $T^{S}{}_{L}{}^{C}$ l'opération de traduction consistant à traduire un texte écrit dans un langage source (S) en un texte écrit dans un langage cible (C) par un compilateur écrit dans un langage de traduction (L), alors on peut décrire plusieurs types de traduction :

- Assembleur : $T^{A}{}_{M}{}^{M}$ où A est le langage d'assemblage et M est le langage machine.
- Compilateur : $T^{S}{}_{M}{}^{M}$ où S est le langage source de haut niveau.

La compilation d'un programme écrit en S, ce que l'on nomme P_S, par un compilateur de S écrit dans un langage M pour une machine M' s'exprime ainsi : $P_S \mid T^{S}{}_{M}{}^{M'} = P_{M'}$. Cela signifie que le « dénominateur » de P_S et le « 1er numérateur » du compilateur s'éliminent.

On peut évidemment écrire les compilateurs en langage machine. Mais les compilateurs sont de gros programmes (le premier compilateur Fortran avait nécessité 18 hommes/années d'effort), qui sont pratiquement impossibles à réaliser en langage machine. De ce fait, on écrit les compilateurs en langage de haut niveau. Comment le fait-on ? Supposons que l'on dispose d'un compilateur PASCAL, c'est-à-dire d'un programme de traduction de la forme suivante : $T^{Pascal}{}_{M}{}^{M}$, et que l'on désire écrire un compilateur C, c'est-à-dire un programme de traduction de la forme $T^{C}{}_{M}{}^{M}$. Pour cela il suffit d'écrire le compilateur sous la forme d'un programme écrit en PASCAL ($T^{C}{}_{Pascal}{}^{M}$) et de le compiler à l'aide du compilateur PASCAL. Ce que l'on note ainsi : $T^{C}{}_{Pascal}{}^{M} \mid T^{Pascal}{}_{M}{}^{M}] = T^{C}{}_{Pascal}{}^{M}$. D'une manière générale lorsque le programme initial est un compilateur, c'est-à-dire un programme de la forme : $T^{L}{}_{S}{}^{M}$, la compilation s'exprime ainsi : $T^{L}{}_{S}{}^{M'} \mid T^{S}{}_{M}{}^{M''} = T^{L}{}_{M''}{}^{M'}$.

On peut aussi écrire un compilateur d'un langage dans le langage lui-même. Par exemple écrire un compilateur C en C. Mais comment faire si l'on ne dispose pas d'un compilateur C initialement ? Il suffit de construire une suite de langages C0, C1, ... Cn = C, tels que C0 $\subset$ C1 $\subset$ C2 $\subset$ ... $\subset$ Cn, et d'écrire la suite de compilateurs suivants :

- $T^{C0}{}_{A}{}^{M} \mid T^{A}{}_{M}{}^{M} = T^{C0}{}_{M}{}^{M}$ où $T^{A}{}_{M}{}^{M}$ est l'assembleur.

- $T^{C1}{}_{C0}{}^{M} \mid T\ ^{C0}{}_{M}{}^{M} = T^{C1}{}_{M}{}^{M}$
- ...
- $T^{Cn}{}_{Cn-1}{}^{M} \mid T\ ^{Cn-1}{}_{M}{}^{M} = T^{Cn}{}_{M}{}^{M}$

et le tour est joué : on a bien notre compilateur qui transforme des textes écrits en langage Cn (c'est-à-dire C) en textes acceptable par la machine M.

Remarque 1 : souvent le compilateur transforme le texte source en assembleur, l'assembleur se chargeant ensuite de traduire ce texte en langage machine. On a ainsi la suite de transformations suivantes :

$$P_S \mid T\ ^{S}{}_{M}{}^{A} \mid T\ ^{A}{}_{M}{}^{M} = P_A \mid T\ ^{A}{}_{M}{}^{M} = P_M$$

Remarque 2 : On note que l'opération de traduction est associative à gauche: (A | B) | C =/= A | (B | C). En effet :

$$T^{S}{}_{X}{}^{Y} \mid T\ ^{X}{}_{Z}{}^{U} \mid T\ ^{U}{}_{A}{}^{M} = T^{S}{}_{U}{}^{Y} \mid T\ ^{U}{}_{A}{}^{M} = T^{S}{}_{M}{}^{Y}$$

A condition, évidemment, que le deuxième programme tourne sur une machine Z et le troisième sur une machine A.

D'autre part, la combinaison de deux programmes n'est pas équivalente à une traduction. Par exemple la combinaison de deux traducteurs suppose que le langage cible du premier est égal au langage source du second : $T^{S}{}_{M}{}^{Y}$ o $T^{Y}{}_{M}{}^{U}$ = $T^{S}{}_{M}{}^{U}$. Dans ce cas, on a la relation suivante :

$$(A \mid B) \mid C \quad =/= A \mid (B \text{ o } C)$$

En effet :

$$T^{S}{}_{X}{}^{Y} \mid T\ ^{X}{}_{Z}{}^{U} \mid T\ ^{U}{}_{A}{}^{M} = T^{S}{}_{M}{}^{Y}$$

$$T^{S}{}_{X}{}^{Y} \mid (T\ ^{X}{}_{Z}{}^{U} \text{ o } T^{U}{}_{A}{}^{M}) = T^{S}{}_{X}{}^{Y} \mid T\ ^{X}{}_{A}{}^{M} = T^{S}{}_{M}{}^{Y}$$

les conditions étant les mêmes que précédemment.

3. Phases de compilation

3.1 Principe

Un compilateur est découpé en plusieurs phases. Chaque phase constitue une partie de traduction en elle même.

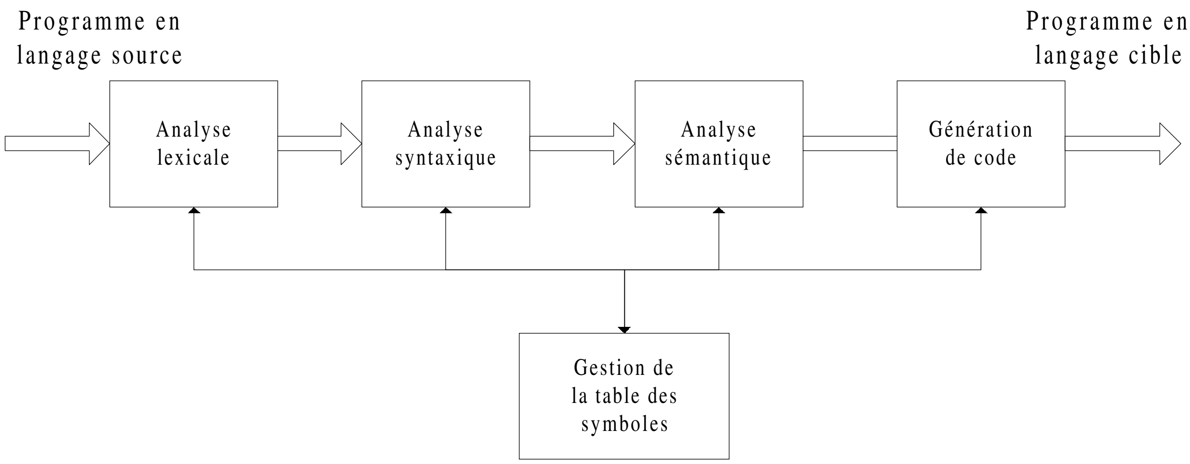

Figure 1. Les différentes phases de compilation

3.2 Analyse lexicale

Consiste à récupérer les mots, que l'on appelle « **tokens** », à partir d'une suite de caractères. Par exemple déterminer, à partir de l'énoncé suivant :

```
for i :=1 to vmax do a :=a+i;
```

on peut dégager la suite de tokens suivante :

```
for  : mot clé
i    : identificateur
:=   : affectation
1    : entier
to   : mot clé
vmax : identificateur
do   : mot clé
a    : identificateur
:=   : affectation
a    : identificateur
+    : opérateur arithmétique
i    : identificateur
;    : séparateur
```

et que l'on peut construire la table des symboles suivante :

Numéro de symbole	**Token**	**Type de token**	**Type de variable**
10	for	mot clé	
11	to	mot clé	
12	do	mot clé	
13	;	séparateur	
	...	...	
100	:=	affectation	
101	+		
	...		

1000	i	ident	
1001	a	ident	
1002	vmax	ident	
	...		
5001	1	entier	
	...		

Ensuite, l'énoncé précédent peut s'exprimer ainsi :

```
10, 1000, 100, 5001, 11, 1002, 12, 1001, 100, 1001, 101,
1000, 13
for   i   :=   1   to   vmax   do      a    +   a   +     i
;
```

c'est-à-dire comme une suite de référence à la table des symboles.

3.3 Analyse syntaxique

Lors de l'analyse syntaxique, on vérifie que l'ordre des tokens correspond à l'ordre définit pour le langage. On dit que l'on vérifie la syntaxe du langage à partir de la définition de sa grammaire. Cette phase produit une représentation sous forme d'arbre de la suite des tokens obtenus lors de la phase précédente. Par exemple, l'arbre suivant représente la structure de la phrase :

```
for i :=1 to vmax do a :=a+i
```

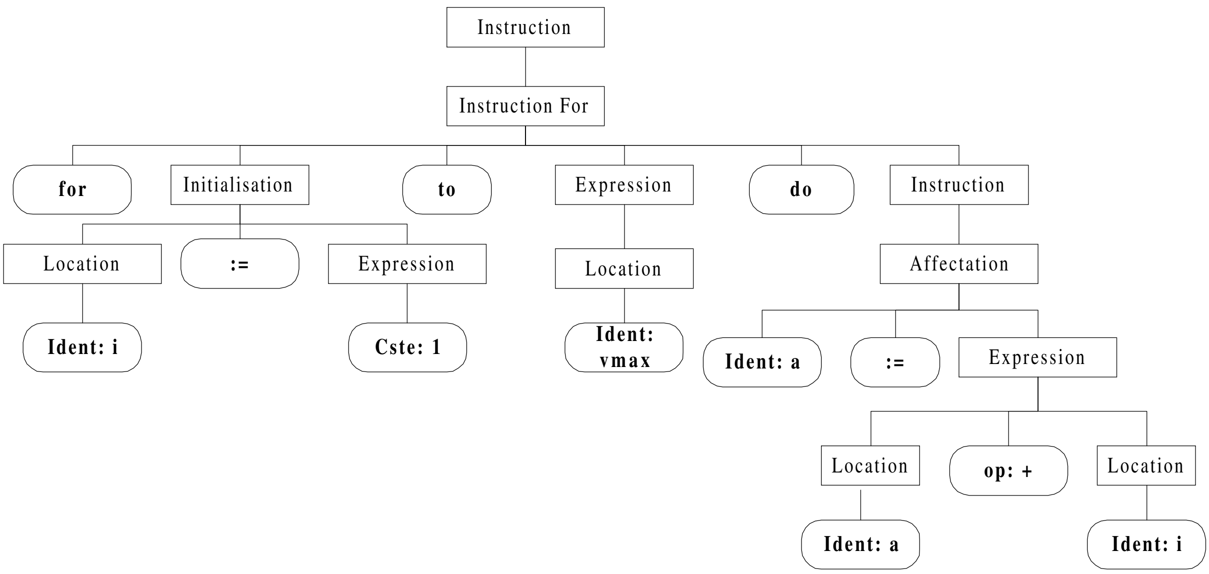

Figure 2. L'arbre syntaxique obtenu après analyse syntaxique

3.4 Analyse sémantique ou vérification de type

Dans cette phase on vérifie que les variables ont un type correct. Par exemple, il faut vérifier que la variable '`i`' possède bien le type 'entier', et que la variable '`a`' est bien un nombre. Cette opération s'effectue en parcourant l'arbre syntaxique et en vérifiant à chaque niveau que les opérations sont correctes.

3.5 Génération de code

On génère le code pour du langage machine ou un langage d'assemblage. Voici un exemple de code généré pour une (pseudo) machine.

```
   var_a          A0000          ;    les    étiquettes    des
variables
   var_i          A0001
   var_vmax       A0002

   ...                           ; le code du programme
   mov var_i,1
loop:
   mov A0, (var_i)               ; comparaison i >= vmax
   jge A0, (var_vmax), finFor    ; si vrai aller en finFor
   mov A0, (var_a)               ; calcul de a+i
   add A0, A0, (var_i)
   mov var_a,A0                  ; a := a+i
   mov A0, (var_i)               ; on incrémente i
   add A0, A0, 1
   mov var_i, 1
   jmp loop                      ; et on continue la boucle
finFor:
   ...
```

4. Organisation du cours

4.1 Langage source et langage cible

On choisira comme langage cible un langage issu d'une machine virtuelle. Une machine virtuelle ressemble à une machine réelle, mais cette machine n'existe pas en tant que telle et doit être simulée (interprétée) par une machine réelle. L'avantage des machines virtuelles est d'être portables (cela tourne n'importe où) et d'éliminer certains détails d'implémentation inutiles.

La machine virtuelle que l'on utilisera s'appelle VCMC97. C'est une adaptation de la machine VCMC94 de J.F. Perrot (proposée en cours de Lisp dans le DESS GLA de Paris 6 en 1994) qui est elle même adaptée de la machine VCMC2 de Jérôme Chailloux et Patrick Greussay, utilisée pour l'implémentation du langage VLisp puis Le_Lisp.

On utilisera deux langages sources : dans un premier temps, on prendra comme langage Source un sous-ensemble du langage SCHEME (lui-même l'un des dialectes de Lisp). L'intérêt de SCHEME est que l'on peut se concentrer tout de suite sur la génération de code en faisant abstraction de l'analyse lexicale, l'analyse syntaxique et l'analyse sémantique, puisque :

1. L'analyse lexicale est faite par la fonction de lecture de SCHEME.

2. L'analyse syntaxique est pratiquement inexistante (puisque les expressions SCHEME correspondent pratiquement à un arbre syntaxique).
3. L'analyse sémantique est inutile puisque SCHEME est un langage faiblement typé (les variables et les paramètres des fonctions ne sont pas typées en SCHEME).

De ce fait, on pourra directement voir comment générer du code à partir des expressions données dans le langage.

Par la suite on introduira comme langage source un sous-ensemble du langage Pascal, et donc les problèmes liés aux analyses lexicales, syntaxiques et sémantiques.

4.2 Bibliographie

Le livre de référence dans le domaine est :

- A. Aho, J. Ullman, *Principles of Compiler Design,* Addisson Wesley, traduit en france par InterEditions (?)

On s'inspirera aussi de :

- H. Abelson, G. Sussman , *Structure et interprétation des programmes informatiques* , InterEditions, qui donne des éléments de compilation (et d'interprétation) des programmes fonctionnels de type Scheme.

5. La machine VCMC97

5.1 Présentation générale

La machine VCMC97 est une machine à pile et à registre. Elle comprend 3 registres généraux , `A0`, `A1` et `A2` plus un registre de pointeur de pile, `SP` (stack pointer), et un registre de pointeur de bloc `FP` (frame pointer).

5.2 Les instructions

Un registre est noté Rx. Une expression constante, une variable globale ou un registre est noté Vx. Un registre ou une variable globale est noté Loc.

Une variable globale est écrit ainsi : (loc V) où V est une variable globale. Les variables globales doivent être décrites dans la zone DATA de la mémoire.

5.2.1 LES DEPLACEMENTS

```
(MOV Loc VS)            ; RB <- VS
```

5.2.2 LES TESTS ET SAUTS

```
(JMP etiq)          ; saut inconditionnel
(JIP VE)            ; saut inconditionnel indirect
(JEQ V1 V2 etiq)    ; saut si =
(JNE V1 V2 etiq)    ; saut si NEQ
(JEG V1 V2 etiq)    ; saut si egalite NUMERIQUE
(JGT V1 V2 etiq)    ; saut si V1 > V2
(JGE V1 V2 etiq)    ; saut si V1 >= V2
```

```
(JLT V1 V2 etiq)  ; saut si V1 < V2
(JLE V1 V2 etiq)  ; saut si V1 <= V2
(JNI VS etiq)     ; saut si NIL
(JNN VS etiq)     ; saut si not NIL
```

5.2.3 LES MANIPULATIONS DE PILES ET APPELS A SOUS-PROGRAMMES

```
(PUSH VS)
(POP RB)
(JSR etiq)        ; appel de sous-programme (l'adresse de retour
                  ; est placée sur la pile
(RTN)             ; retour d'un sous-programme. Arrête la
                  ; machine si rien à depiler
(NTH R1 NS R2)    ; met dans R1 le NS-eme element de la pile
                  ; par rapport a R2. ex: (NTH A0 3 FP)
(SAV)             ; sauve l'environnement (avant un appel) et
                  ; lie un bloc (utilise FP)
(REST)            ; restore l'environnement (apres un appel) en
                  ; déliant un bloc
(FENTRY  nom)         ; aucune utilite directe.
                  ; indique simplement le debut d'une procédure
(CALL SYM)        ; appel la primitive de nom SYM,
                  ; le bon nombre d'arguments doit se trouver
                  ; sur la pile
```

5.2.4 LES OPERATIONS PRIMITIVES

```
(ADD RB V1 V2)    ; RB <- V1 + V2
(MUL RB V1 V2)    ; RB <- V1 * V2
(SUB RB V1 V2)    ; RB <- V1 - V2
(DIV RB V1 V2)    ; RB <- V1 / V2 (partie entière)
(MOD RB V1 V2)    ; RB <- V1 mod V2 (reste de la division)

(CONS RB V1 V2)   ; RB <- (cons V1 V2)
(CAR RB V1)       ; RB <- (car V1)
(CDR RB V1)       ; RB <- (cdr V1)
```

5.2.5 LA MANIPULATION DES VARIABLES

```
(LOCAL RB n)      ; RB <- @(FP+n)
                  ; recupere le nieme element sur la pile
                  ; par rapport a FP
(SETLOCAL RB n)   ; (FP+n) <- RB
                  ; affecte le nieme element de la pile à n
```

5.2.6 DIVERS

```
(DATA (var1 val1) ... (varn valn)) ; déclaration des variables
                  ; globales. Ne permet de décrire que des
                  ; variables simples (pas des tableaux)
```