Chapitre I.
Introduction aux compilateurs (Objectifs)

a.hettab@centre-univ-mila.dz

Qu'est-ce qu'un compilateur ?

- Un **compilateur** est un **traducteur** qui permet de transformer un programme écrit dans un langage **L1** en un autre programme écrit dans un langage **L2**.
- Le langage **L1** dans lequel on écrit le programme original est :
  - **le langage source ex: C, C++, Pascal …**
- Le langage **L2** généré à partir du langage **L1** est :
  - **le langage cible (ou langage objet) ex : EXE,…**
- En pratique on s'arrête souvent à un langage intermédiaire (assembleur ou encore un langage d'une machine abstraite)

Qu'est-ce qu'un programme?

- Un programme est un ensemble d'opérations qui décrivent comment produire des résultats à partir des entrées.
- Le compilateur rejette des programmes qui contiennent des erreurs (erreurs statiques),
- dans le cas contraire, le compilateur construit un nouveau programme (le programme objet) que la machine pourra l'exécuter sur différentes entrées.
- L'exécution du code objet sur une entrée particulière peut ne pas terminer ou échouer à produire un résultat (erreur dynamique).

Exemple de compilation

```
1  int SommeCarre (int a, int b)
2  {
3    return a*a+b*b;
4  }
```

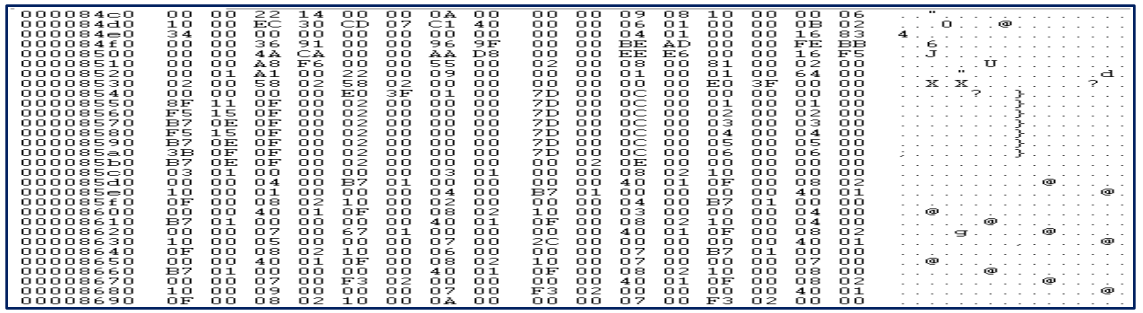
Programme source: langage C

Compilateur gcc

```
1  <SommeCarre>:
2  0:  0f  af  ff      imul   %edi,  %edi
3  3:  0f  af  f6      imul   %esi,  %esi
4  6:  8d  04  3e      lea    (%rsi, %rdi,1), %eax
5  9:  c3      retq
```

Programme Assembleur généré

Assembleur

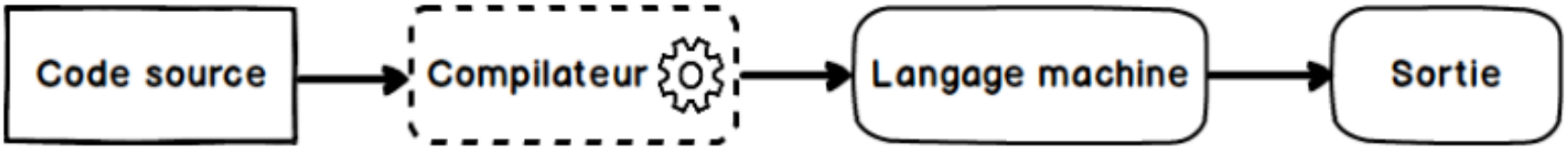


Code machine généré

Compilateur vs Interpréteur

Compilateur

- Un **compilateur** prend tout le programme et le convertit en code objet qui est généralement stocké dans un fichier. Le code objet est également référencé en tant que code binaire et peut être exécuté directement par la machine après la liaison.
- Exemple: **C**, **C++**, **Pascal …**



Compilateur vs Interpréteur

interpréteur

- Un **interpréteur** exécute directement des instructions écrites dans un langage de programmation l'une après l'autre sans les convertir en un code objet ou un code machine.
- Exemple: **BASIC, Perl, Python, Ruby, PHP ….**

Compilateur vs Interpréteur

Avantage et inconvénient

	Avantage	Inconvénient
Interpréteurs	processus de développement simple (en particulier débogage)	processus de traduction peu efficace et vitesse d'exécution lente
Compilateurs	Transmet au processeur l'intégralité du code machine prêt à l'emploi et exécutable	Toute modification du code nécessite une nouvelle traduction (correction des erreurs, extension du logiciel, etc.)

Compilateur vs Interpréteur

compilateur à la volée

- La **compilation à la volée** est une solution hybride traduit le code du programme pendant le fonctionnement, à l'instar d'un interpréteur. Elle mélange les deux méthodes précédentes pour bénéficier de leurs avantages. De cette façon, la **vitesse d'exécution élevée** (permise par le compilateur) est complétée par un **processus de développement simplifié**.
- Exemple: **JAVA….**

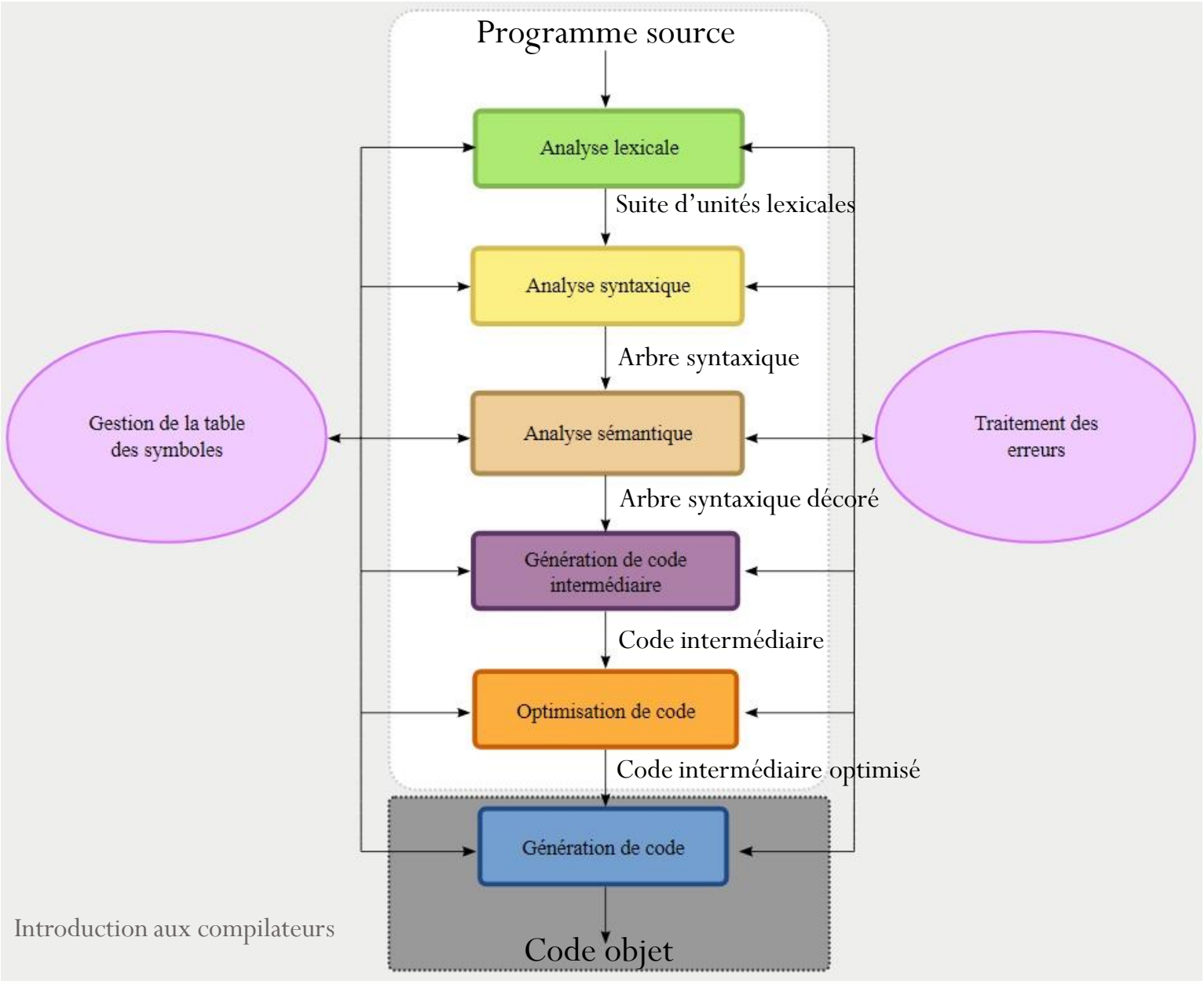
Qu'attend-on d'un compilateur ? (1)

- **Détection des erreurs**: Un compilateur doit pouvoir détecter les erreurs statiques comme:
  - Identificateurs mal formés,
  - commentaires non fermés . . .
  - Constructions syntaxiques incorrectes,
  - Identificateurs non déclarés,
  - Expressions mal typées if 3 then "toto" else 4.5,
  - Références non instanciées.

Qu'attend-on d'un compilateur ? (2)

- **Efficacité** : Un compilateur doit être si possible rapide.
- **Correction** : Le programme compilé doit représenter le même calcul que le programme original.
- **Modalité** : Utilisation de la compilation séparée lors de d'un gros développement.

Structure d'un compilateur



Analyse lexicale

- L'analyse lexicale consiste à transformer un code sous forme textuelle en une suite de token (unités lexicales), séparant ainsi les mots-clés, les variables, les entiers, etc.…
- Les commentaires et les blancs séparant les caractères formant les unités lexicales sont éliminés au cours de cette phase.
- L'analyseur lexical associe à chaque unité lexicale un code qui spécifie son type et une valeur (un pointeur vers **la table des symboles**).

Exemple

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

Table des symboles

- Une **table de symboles** est une centralisation des informations rattachées aux identificateurs d'un programme informatique.
- Dans une table des symboles, on retrouve des informations comme : le type, l'emplacement mémoire, etc.
- Généralement, la table est créée dynamiquement. Une première portion est créée au début de la compilation. Puis, de façon opportuniste, en fonction des besoins, elle est complétée.
- La première fois qu'un symbole est vu (au sens des règles de visibilité du langage), une entrée est créée dans la table.

Table des symboles

N°	Unité Lexicale	Type de l'unité lexicale
10	for	Mot clé
11	to	Mot clé
12	do	Mot clé
13	;	séparateur
…	……	……
100	:=	affectation
101	+	opérateur arithmétique
…	….	….
1000	i	Identificateur
1001	a	Identificateur
1002	vmax	Identificateur
….	…..	
5000	1	entier

Ensuite, l'énoncé précédent peut s'exprimer ainsi : 10, 1000, 100, 5000, 11, 1002, 12, 1001, 100, 1001, 101, 1000, 13

Analyse syntaxique

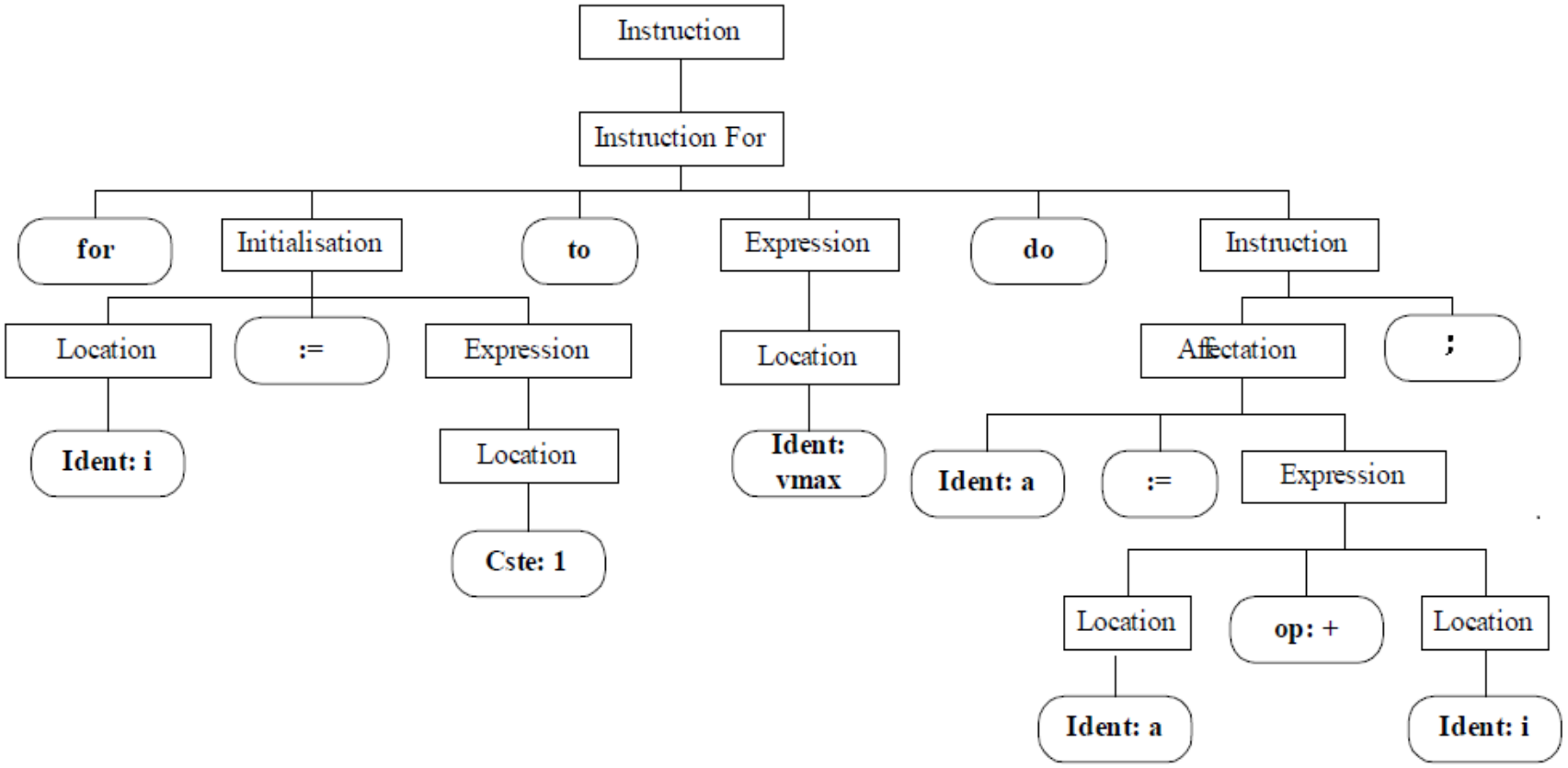
- Cette phase consiste à regrouper les unités lexicales du programme source en structures grammaticales (i.e. vérifier si un programme est correctement écrit selon la grammaire qui spécifie la structure syntaxique du langage).
- En général, cette analyse est représentée par un arbre.

Exemple

- Considérons la grammaire suivante pour la boucle for:
- **<Instruction>→ < Affectation > ;| <Instruction for>**
- **< Affectation > → ident := < Expression>**
- **<Instruction for> → for < Initialization> to < Expression> do <Instruction>**
- **< Initialization> → <Location> := < Expression >|<Location>**
- **<Location> → ident|const**
- **< Expression>→ < Location>|< Location > op < Location >**

Arbre syntaxique

- Après l'analyse syntaxique on obtient l'arbre suivant



Analyse sémantique

- L'analyse sémantique sert à préciser la nature des calculs représentés par un programme en vérifiant que les opérandes de chaque opérateur sont conformes aux spécifications du langage source.
- En général, cette analyse est représentée par un arbre syntaxique décoré.

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..

Génération de code intermédiaire

- Après les phases d'analyse sémantique, certains compilateurs produisent une représentation intermédiaire, une sorte de code pour une machine abstraite.
- La forme intermédiaire "code à trois adresses" est largement utilisée.
- Le passage par le code intermédiaire apporte certains avantages:
  - il est relativement facile de changer le code intermédiaire en code objet;
  - la représentation intermédiaire permet d'appliquer des optimisations indépendantes du langage cible (code objet).

Exemple: **JAVA** utilise une forme intermédiaire spécifique: Les **bytecode.** Le bytecode est exécuté sur n'importe quelle plateforme à l'aide de la machine virtuelle JAVA (**JVM**).

Optimisation du code intermédiaire (1)

- L'optimisation cherche à améliorer le code intermédiaire afin d'améliorer ses performances en terme de réduire le temps d'exécution ou l'occupation mémoire.
- Quelques unes des opérations d'optimisation sont :
  - **Déplacement de code invariant :**

    Le but est d'extraire du corps de la boucle des parties du code qui sont des invariants (nécessitent une seule exécution). Le nombre total d'instructions exécutées est diminué.

Exemples. Soit le code intermédiaire suivant:

```
for (int i = 0; i < n; i++) {
 x = y + z;
 a[i] = 6 * i + x * x; }
```

Après le déplacement du code invariant, on obtient:

```
x = y + z;
temp1 = x * x;
for (int i = 0; i < n; i++) {
 a[i] = 6 * i + temp1; }
```

Optimisation du code intermédiaire (2)

- Quelques unes des opérations d'optimisation sont :
  - **Elimination du code mort :**

  Le compilateur sait reconnaitre les parties du code qui sont morts (la raison peut être une erreur de programmation).

Exemple . Des instructions qui sont inaccessibles à cause d'instructions de saut sont considérées comme du code mort et peuvent être éliminées:

```
if 1 <> 0 goto label
i := a[k]
k := k + 1
label:
```

Optimisation du code intermédiaire (3)

- Quelques unes des opérations d'optimisation sont :
  - **Elimination des sous-expressions communes**

  Dans le cas où la valeur d'une expression est calculée en plusieurs endroits du programme, l'optimiseur gère le résultat de cette expression et le réutilise plutôt que refaire le calcul.

Exemple . Soit le code intermédiaire suivant:

```
t6 = 4 * i
x = a[t6]
t7 = 4 * i
t8 = 4 * j
t9 = a[t8]
a[t7] = t9
t10 = 4 * j
a[t10] = x
```

Après l'élimination de sous-expressions communes (et quelques autres optimisations), on obtient:

```
t6 = 4 * i
x = a[t6]
t8 = 4 * j
t9 = a[t8]
a[t6] = t9
a[t8] = x
```

Génération de code objet

- Cette phase produit du code objet en:
  - choisissant les emplacements mémoires pour les données ;
  - sélectionnant le code machine pour implémenter les instructions du code intermédiaire ;
  - allouant les registres.

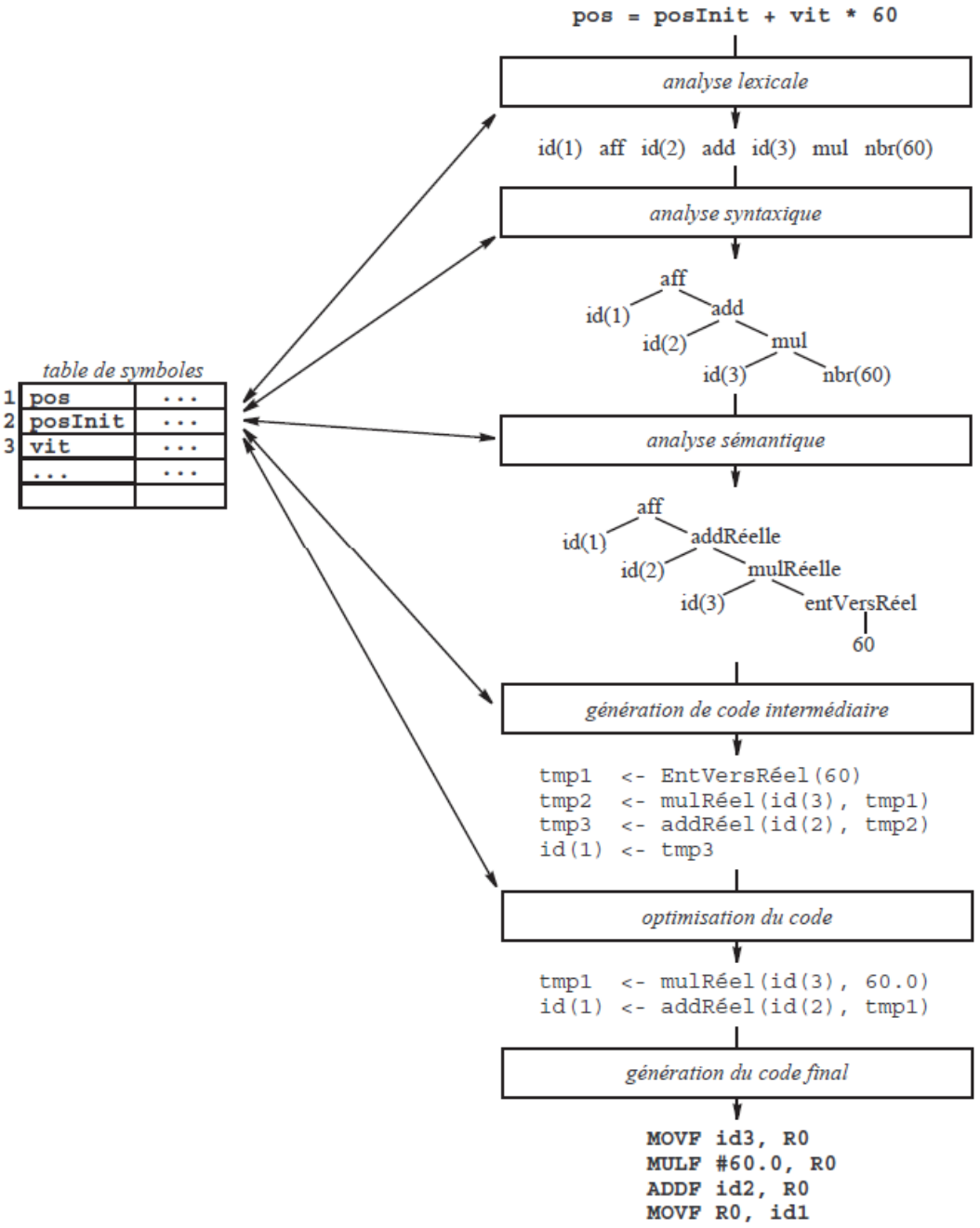
Exemple: voici le code intermédiaire optimisé suivant:

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

Voici le code objet généré en utilisant les registres R1 et R2:

```
MOVF id3, R2
MULF #600, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

Phases logiques de la compilation d'une instruction



Références

- **Compilateurs Principes, techniques et outils**. Alfred V.Aho, Ravi Sethi, Jeffrey D.Ullman. Pearson. Paris France. 2007.