Théorie des Langages
Épisode 0 — Introduction

Thomas Pietrzak

Université Paul Verlaine — Metz

Quelques ouvrages de référence

- Alfred Aho, Ravi Sethi et Jeffrey Ullman. *Compilateurs : Principes, techniques et outils*, Dunod, Paris, 2000
- John Hopcroft et Jeffrey Ullman. *Introduction to automata theory, languages, and computations*, Addison Wesley, 2001

Sur le net

- `http://wwwdgeinew.insa-toulouse.fr/~lebotlan/cours/theorie-des-langages.pdf`
- `http://www.lri.fr/~vialette/Enseignements/2004-2005/LF/`

Sources

Cours de messieurs :

- Yurii Rogozhin
- Dieter Kratsch
- Dominique Cansell

Compilation

- permettre d'écrire des programmes complexes
- permettre d'identifier des erreurs dans le programme
- s'affranchir de la machine utilisée

Cours

- connaître les méthodes d'analyse de langage les plus courantes
- savoir comment est fait un outil peut aider à mieux l'utiliser

Autres applications

Les techniques de compilation ne servent pas qu'à créer des compilateurs. Elles sont très utiles dans d'autres cas, parmi :

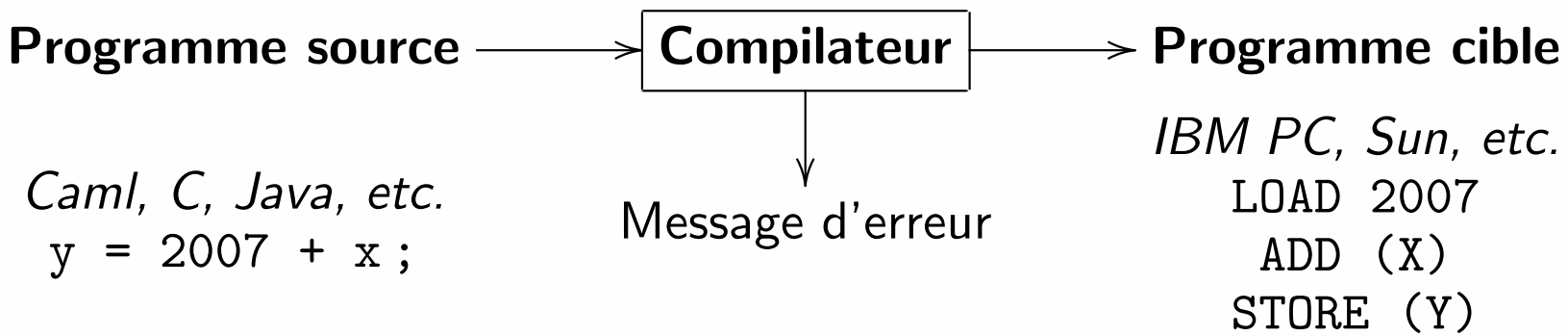
- Systèmes de composition de texte : certains passages du texte d'un éditeur peuvent constituer des commandes pour générer des figures, des structures mathématiques, de la mise en forme, etc.
- Compilateurs de silicium : proche d'un compilateur conventionnel, à ceci près que les variables ne représentent pas des des emplacements mémoire, mais des signaux logiques ou des groupes de signaux dans un circuit de commutation.
- Interpréteur de requêtes : lignes de commandes, requêtes dans une base de donnée, etc.

Definition

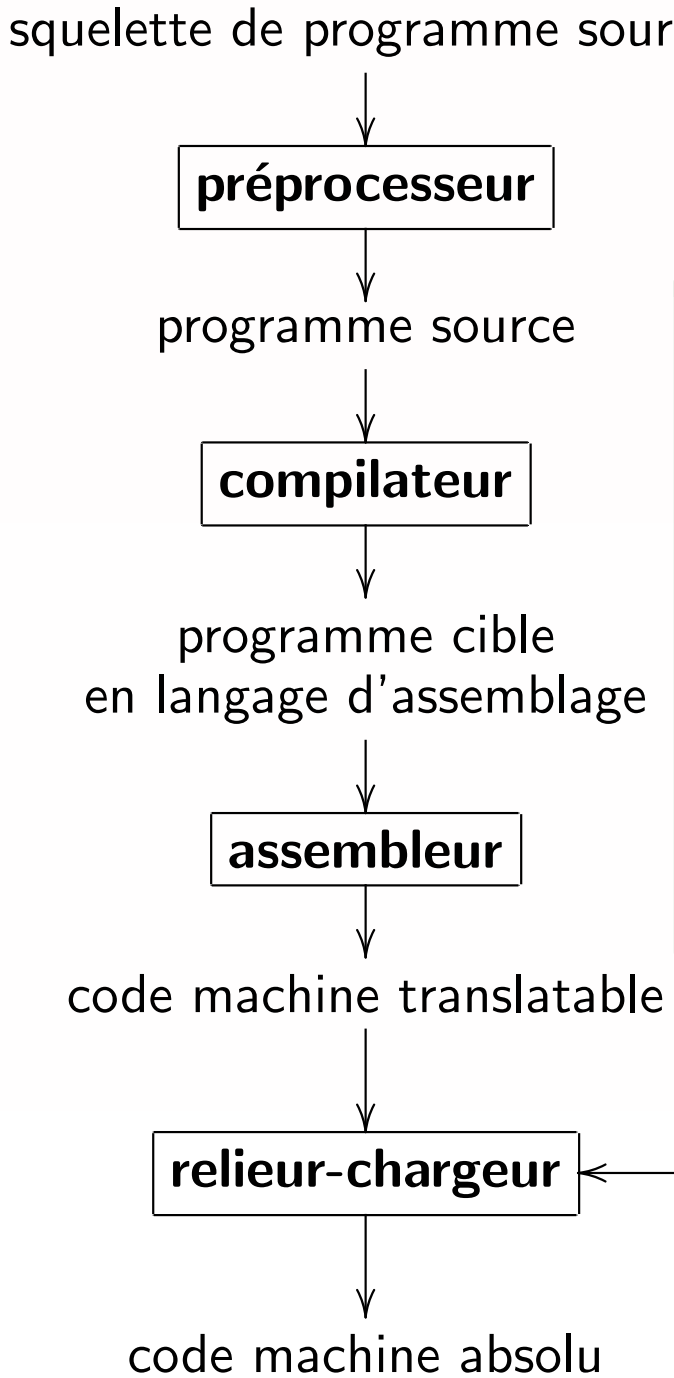
La compilation embrase les langages de programmation, l'architecture des machines, la théorie des langages, l'algorithmique et le génie logiciel.

Definition

Un compilateur est un programme qui lit un programme écrit dans un premier langage (le langage source) et le traduit en un programme équivalent écrit dans un autre langage (le langage cible).



La création d'un programme cible exécutable peut requérir, en plus du compilateur, plusieurs autres programmes.




```
gcc -c monfichier1.c
gcc -c monfichier2.c
gcc -c ...
```
préprocesseur + compilateur + assembleur

```
gcc -o monprogramme monfichier1.o
 monfichier2.o ...
```
relieur-chargeur

blibliothèques,
fichiers objet translatables

Préprocesseur

Un programme source peut être divisé en modules stockés dans des fichiers séparés. La tâche consistant à reconstruire le programme source est parfois confié à un programme distinct, appelé préprocesseur. Il peut réaliser diverses tâches :

- Macro-expansion : définition de « marcos-définitions » ou plus simplement « macros ». Exemple (C) : `#define MAX(X,Y) ((X)>(Y)?(X) :(Y))`.
- Inclusion de fichiers : inclusion d'en-têtes dans le texte d'un programme. Exemple (C) : `#include <stdio.h>`.
- Préprocesseurs rationnels : permet d'ajouter des instructions n'existant pas à la base dans un langage de programmation.
- Extensions de langages : extension des fonctionnalités d'un langage par l'intermédiaire de sortes de macros intrasèques. Exemple (Equel et C) : les instructions commençant par ## sont des instructions d'accès à une base de donnée.

Assembleur

Le compilateur produit du code en langage d'assemblage, qui est traduit par un assembleur en code machine.
Certains compilateurs produisent du code en langage d'assemblage qui doit être confié à un assembleur, alors que d'autres se chargent de cette phase et produient directement du code translatable.

Exemple

$b := a + 2$

```
MOV a, R1
ADD #2, R1
MOV R1, b
```

Note : certains langages d'assemblages proposent des macros telles que ceux décrits précédemment.

Relieur-chargeur

Un programme cible peut être constitué de plusieurs fichiers objet translatables, et nécessite parfois des bibliothèques. Le rôle du Relieur-chargeur est de les assembler pour créer le programme cible exécutable.

Chargement

Le chargement consiste à prendre du code machine translatable, à modifier les adresses translatables et à placer les instructions et les données ainsi modifiées en mémoire, aux emplacements appropriés.

Relieur

Le relieur permet de constituer un unique programme à partir de plusieurs fichiers contenant du code machine translatable. Ces fichiers peuvent avoir été produits par plusieurs compilations séparées, et un ou plusieurs d'entre eux peuvent être des fichiers ou des routines de bibliothèque, fournis par le système et displonibles pour tout programme qui en a besoin.

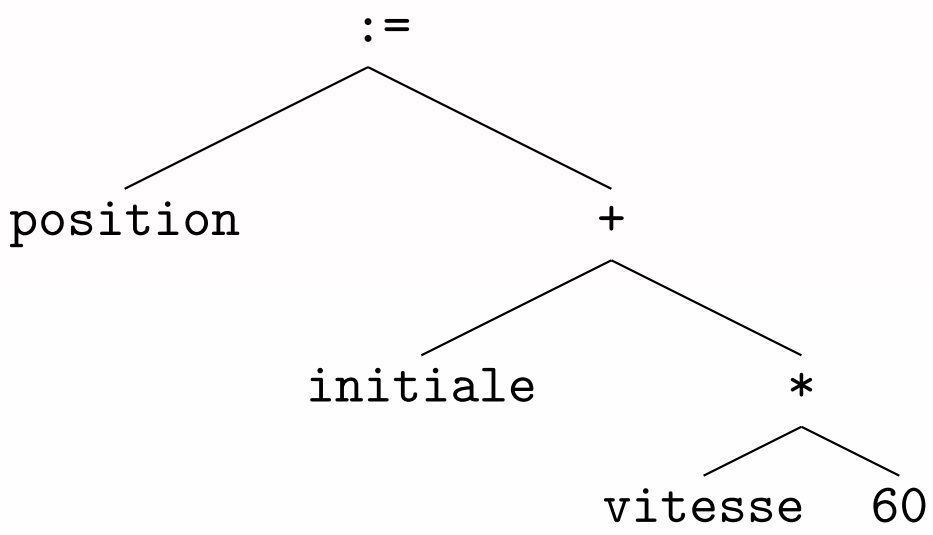
```
gcc -lmath -o monprogramme monfichier.o
```

Compilation

La compilation se divise en deux étapes :

- L'analyse qui partitionne le pogramme source en ses constituants et en crée une représentation intermédiaire.
- La synthèse qui construit le programme cible à partir de la représentation intermédiaire.

La théorie des langages intervient au niveau de l'analyse.



Arbre abstrait pour l'opération :
`position := initiale + vitesse * 60`

Pendant l'analyse, les opérations spécifiées par le programme source sont déterminées et conservées dans une structure hiérarchique appelée arbre. On utilise souvent un genre d'arbre spécial, appelé arbre abstrait, dans lequel chaque noeud représente une opération et les fils de ce noeud représentent les arguments de cette opération.

Les différentes phases de compilation sont souvent réunies en dex grandes parties : la partie frontale et la partie finale.

Partie frontale

La partie frontale est constituée des phases de compilation qui dépendent principalement du langage source et qui sont en grande partie indépendantes de la machine cible. Elle comprend l'analyse lexicale, syntaxique et sémantique, ainsi que la génération du code intermédiaire. Une partie de l'optimisation peut en faire partie

Partie finale

La partie finale comprend les phases de compilation qui dépendent de la machine cible, c'est à dire l'optimisation et la génération du code. Celles-ci ne dépendent en général pas du langage source, mais seulement du langage intermédiaire.

La création d'un programme cible exécutable peut requérir, en plus du compilateur, plusieurs autres programmes.

Analyseur lexical

L'analyseur lexical lit les caractères formant le programme source et les groupe en un flot d'unités lexicales, dont chacune représente une suite de caractères formant un tout logiquement cohérent, comme un identificateur, un mot clé, un caractère de ponctuation, ou un opérateur composé de plusieurs caractères comme `:=`. La suite de caractères composant une unité lexicale est appelée son lexème.

Exemple

L'expression : `position := initiale + vitesse * 60` devient :
$\texttt{id}_1$ `:=` $\texttt{id}_2$ `+` $\texttt{id}_3$ `* 60`
La table des symboles est créée :

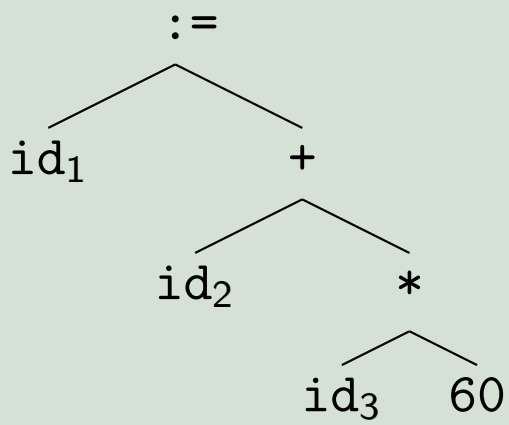
1	position	...
2	initiale	...
3	vitesse	...
4		

Analyseur syntaxique

L'analyseur syntaxique impose une structure hiérarchique sur le flot d'unités lexicales. Elle est représentée par un arbre abstrait.

Exemple

L'expression $\texttt{id}_1$ `:=` $\texttt{id}_2$ `+` $\texttt{id}_3$ `*` `60` devient un arbre :



La table des symboles reste la même.

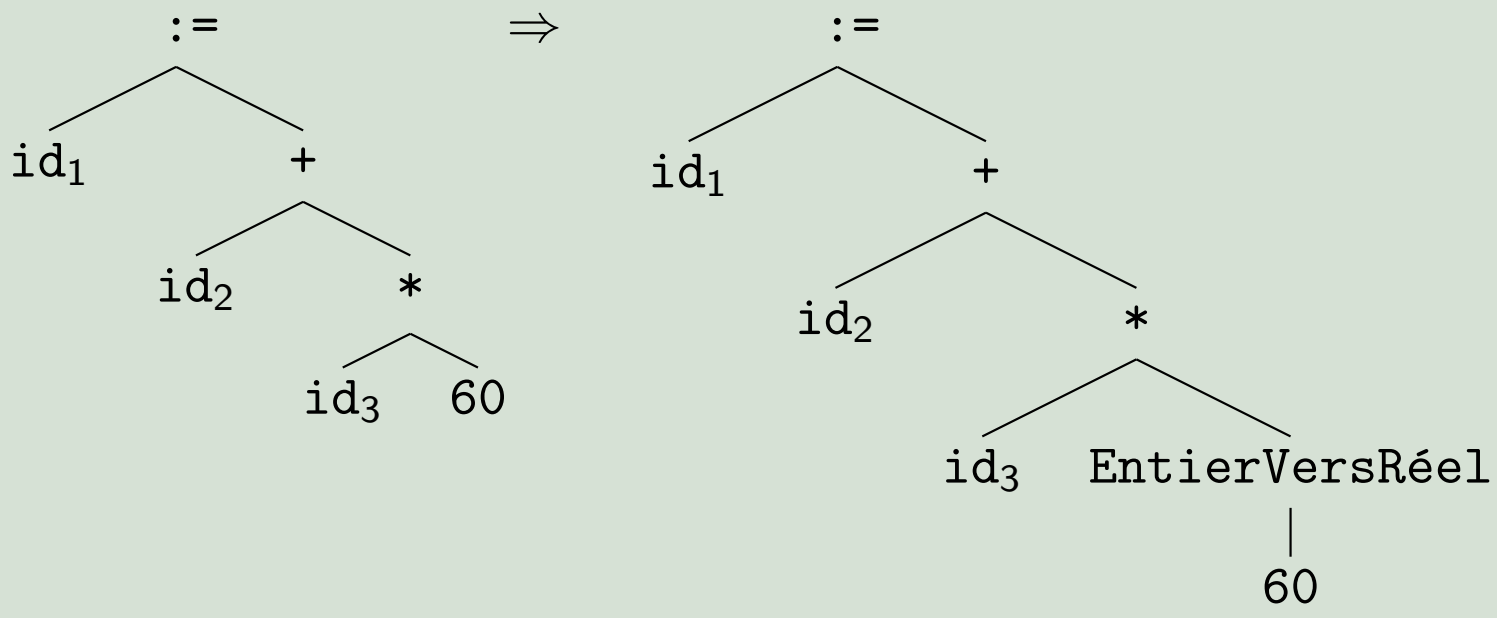
Analyseur sémantique

L'analyseur sémantique contrôle le code pour s'assurer que le code est conforme aux spécifications du langage. Il contrôle entre autres :

- le typage
- le flot d'exécution : branchements après un break
- unicité des objets, des étiquettes, etc.
- ...

Exemple

Si on considère que les opérations se font sur les réels et non sur les entiers, il faut modifier l'arbre :



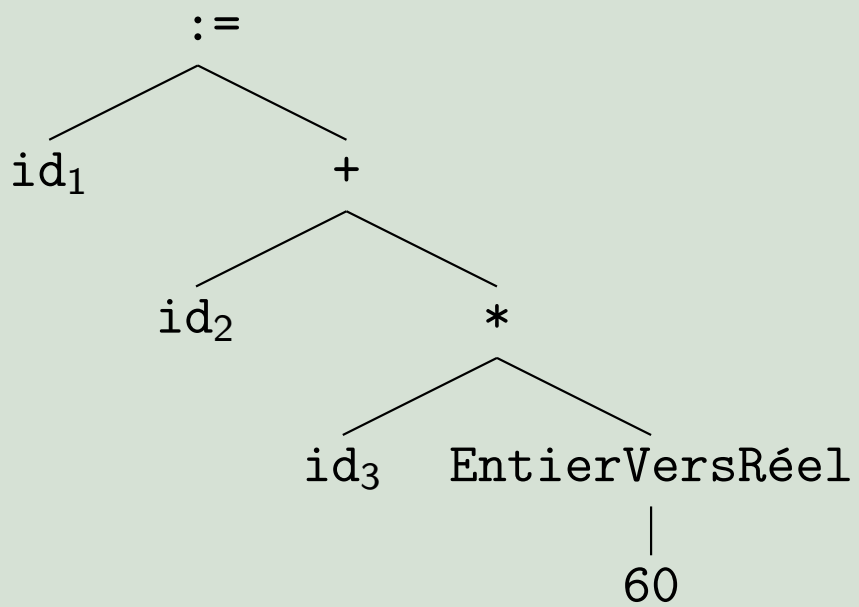
La constante entière 60 est convertie en réel.

Générateur de code intermédiaire

Le générateur de code intermédiaire contruit explicitement une représentation intermédiaire du programme source. Cette représentation intermédiaire doit être facile à produire et facile à traduire en langage cible.

Exemple

Un code intermédiaire est produit à partir de l'arbre précédent :




```
temp1 := EntierVersRéel(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

Optimiseur de code

L'optimiseur de code tente d'améliorer le code intermédiaire de façon que le code machine résultant s'exécute plus rapidement.

Exemple

Le code précédent est optimisé vu que :

- la constante 60 peut être convertie à la compilation
- la variable `temp3` n'est utilisée qu'un fois, pour transmettre sa valeur à `id1`

```
temp1 := EntierVersRéel(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

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

Générateur de code

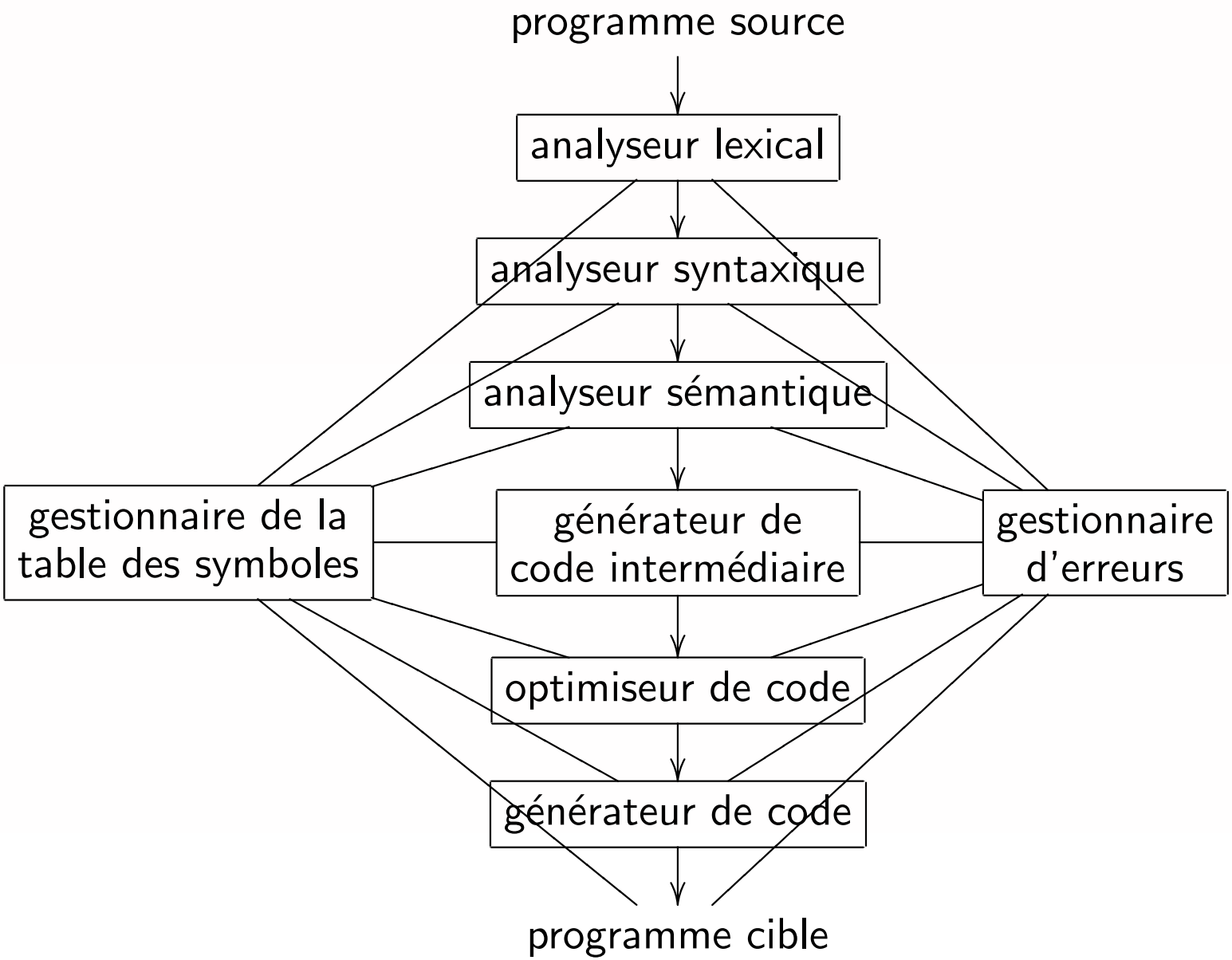
Le générateur de code produit le code cible, c'est à dire du code machine translatable ou du code en langage d'assemblage. Les emplacements mémoire pour chacune des variables utilisées dans le programme sont sélectionnées. Les instructions intermédiaires sont traduites en une suite d'instructions machine qui effectuent la même tâche. Un aspect crucial de ce processus est l'assignation de variables aux registres.

Exemple

Le code optimisé est traduit en langage d'assemblage. On utilise les registres.

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

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


programme source
analyseur lexical
analyseur syntaxique
analyseur sémantique
gestionnaire de la table des symboles
générateur de code intermédiaire
gestionnaire d'erreurs
optimiseur de code
générateur de code
programme cible