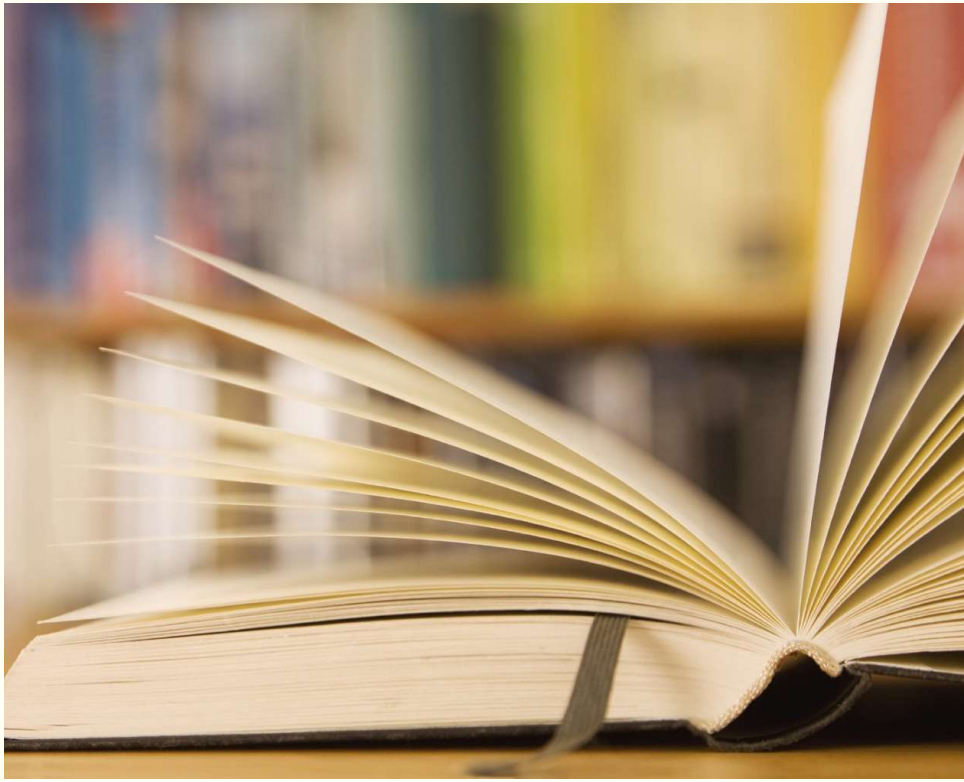
THEORIE DE LA COMPILATION

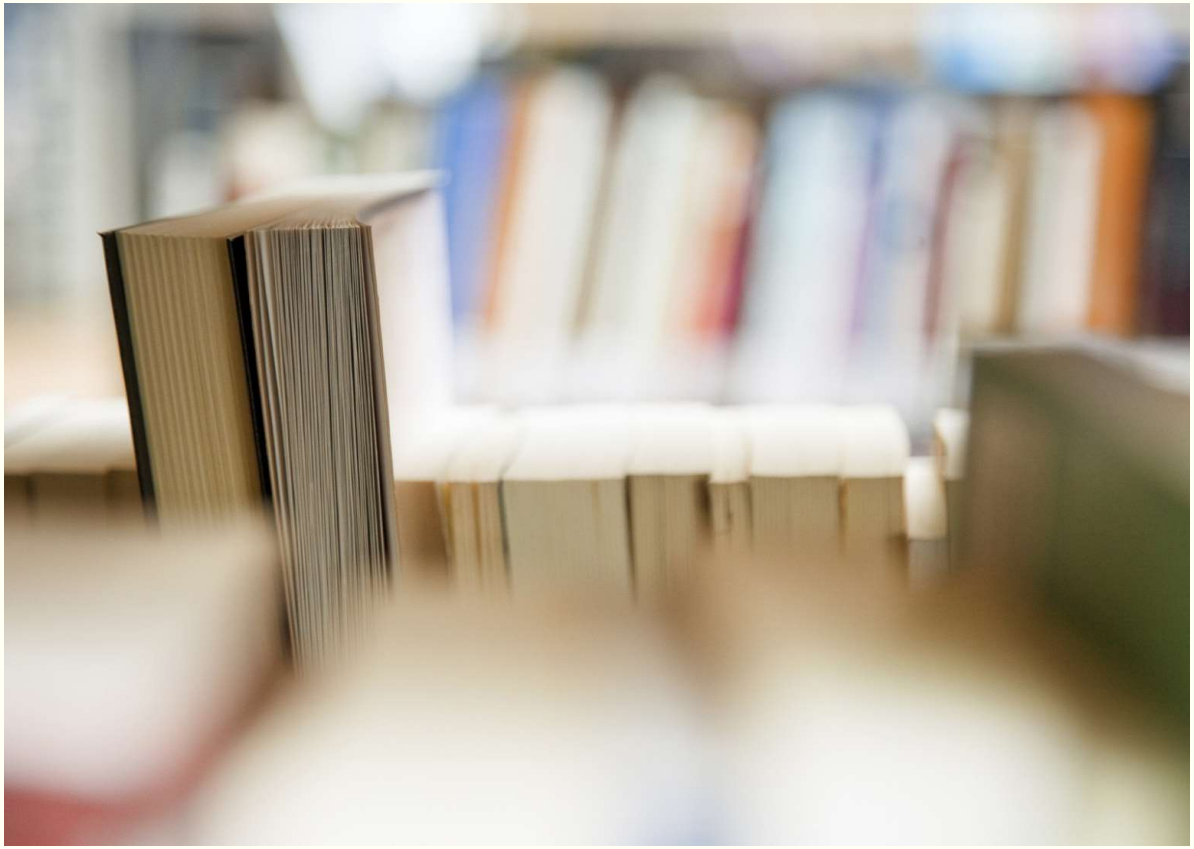
DESCRIPTION DE LA PHASE D'UN COMPILATEUR

Dr Souissi Abdelmoghit



SOMMAIRE

1. Introduction à la théorie de compilation
2. Modèle en couche
3. Qu'est ce qu'un compilateur
4. Déroulement d'une compilation
5. Exemple code assembleur
6. Structure d'un compilateur
7. Outils théoriques et pratiques
8. Type de compilateurs
9. Analyse Lexicale
10. Token
11. Analyse Syntaxique
12. Analyse sémantique
13. Optimisation du code



Qu'est-ce que la Théorie de Compilation ?

Un compilateur est un programme informatique qui traduit un code source écrit dans un langage de programmation de haut niveau (comme C, Java, etc.) en code machine ou un langage intermédiaire compréhensible par l'ordinateur.

Ce processus est réalisé en plusieurs phases, chacune jouant un rôle spécifique dans la compilation. Voici une description des principales phases d'un compilateur :

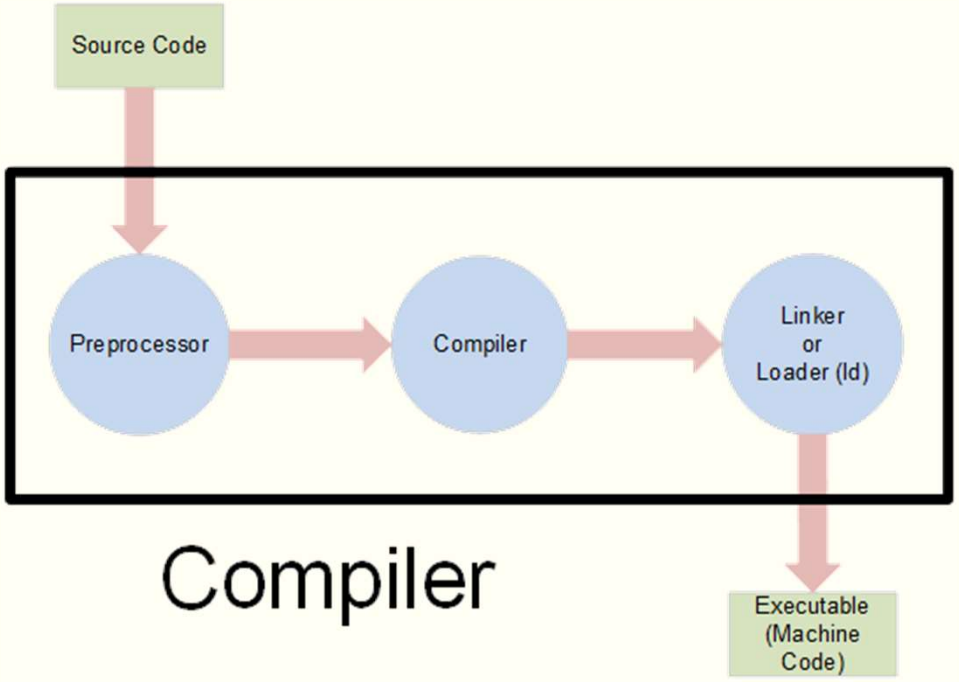
1. Analyse Lexicale : Divise le code source en tokens.
2. Analyse Syntaxique : Vérifie la structure grammaticale et crée un AST.
3. Analyse Sémantique : Vérifie les règles logiques (types, portée, etc.).
4. Génération de Code Intermédiaire : Produit un code intermédiaire abstrait.
5. Optimisation : Améliore l'efficacité du code sans changer son comportement.
6. Génération de Code : Traduit le code intermédiaire en code machine.
7. Assemblage et Linking : Génère le fichier exécutable final.

Modèle en couches

6	Programmes d'application *(Traitement de texte, PAO, Jeux, …)*
5	Langages de programmation *(Fortran, Cobol, C, C++, Java, …)*
4	Langage assembleur *(Langage natif symbolique de la machine)*
3	Noyau du système d'exploitation *(Gestion des taches, des ressources : mémoire, I/Os,…)*
2	Langage machine : jeu d'instructions *(Langage natif du processeur)*
1	Langage de microprogrammation
0	Logique numérique *(Couche matérielle : circuits logiques, électroniques)*

Qu'est-ce qu'un compilateur?

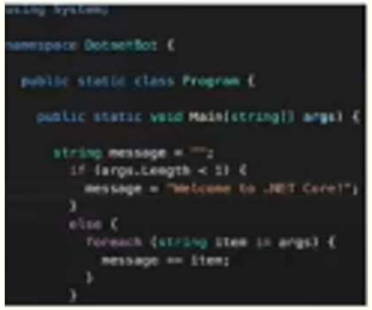
- Le reole de compilateur est la traduction de programmes d'un langage de haut niveau vers un langage de bas niveau (langage machine) exécutable par l'ordinateur.



Déroulement d'une compilation d'un programme écrit avec le langage haut niveau

Programmeur

Code en langage évolué



Traduction

Code en assembleur

```
section .data
  num1 db 5
  num2 db 10
  result db 0
section .text
      global _start
_start:
      mov al, [num1]
      mov bl, [num2]
```

Code en micro instruction

```
R1 ← MEM[num1]
R2 ← MEM[num2]
R3 ← R1 + R2 :
R3.MEM[result] ← R3 :
```

Linker

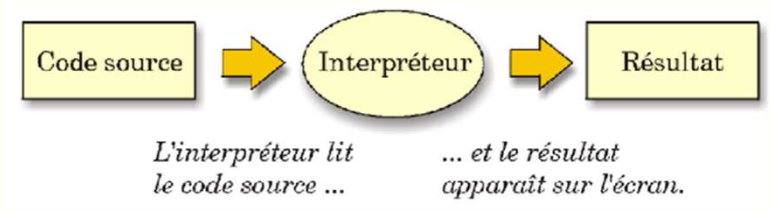
```
111011111111
100111111011
111111100011
100011111111
```

Exemple d’un programme écrit avec le langage assembleur

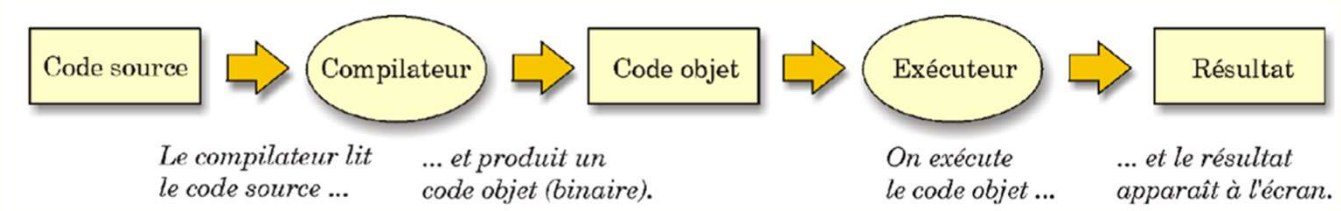
```
section .data                ; section des données
      num1 db 5                ; première variable, valeur 5
      num2 db 10               ; deuxième variable, valeur 10
      result db 0            ; résultat initialisé à 0

section .text               ; section de code
      global _start         ; point d'entrée pour l'exécution

_start:
      ; Charger les valeurs des variables dans les registres
      mov al, [num1]        ; charger la valeur de num1 dans le registre AL
      mov bl, [num2]        ; charger la valeur de num2 dans le registre BL

      ; Ajouter les deux registres
      add al, bl              ; AL = AL + BL (sommer num1 et num2)

      ; Stocker le résultat
      mov [result], al     ; stocker la somme dans la variable result

      ; Fin du programme
      mov eax, 60             ; appel système pour terminer le programme (code 60)
      xor edi, edi          ; statut de retour 0
      syscall                 ; exécuter l'appel système
```

Fonctionnement des compilateurs vs interpréteurs

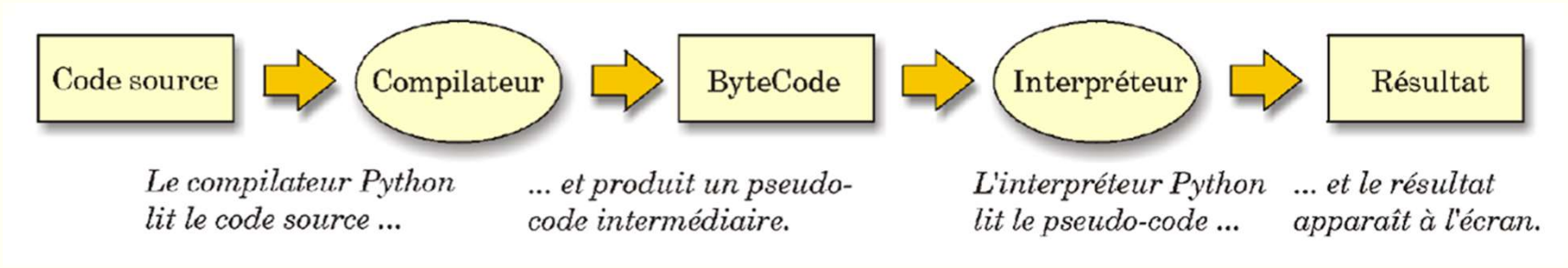
Interpréteur



Compilateur



Compilateur avec machine virtuelle



Compilateur Vs Traducteur

Compilateur: Il analyse tout le code source en une seule fois, puis le compile en un fichier binaire exécutable. Ensuite,

Avantages : Meilleure performance lors de l'exécution, car le programme est déjà traduit en code machine.

Le code peut être optimisé pour améliorer les performances.

Inconvénients : Temps de compilation plus long.

En cas d'erreur, il faut recompiler le programme après correction.

Exemples : Compilateur GCC (C/C++), javac (Java).

Interprète (ou traducteur) : Un interprète est un programme qui traduit et exécute le code source ligne par ligne, sans le transformer en un fichier exécutable indépendant.

Fonctionnement : Il lit chaque ligne du code source, la traduit en instructions machine, puis l'exécute immédiatement avant de passer à la ligne suivante.

Avantages :Facilite la détection rapide des erreurs, car elles sont détectées ligne par ligne pendant l'exécution.

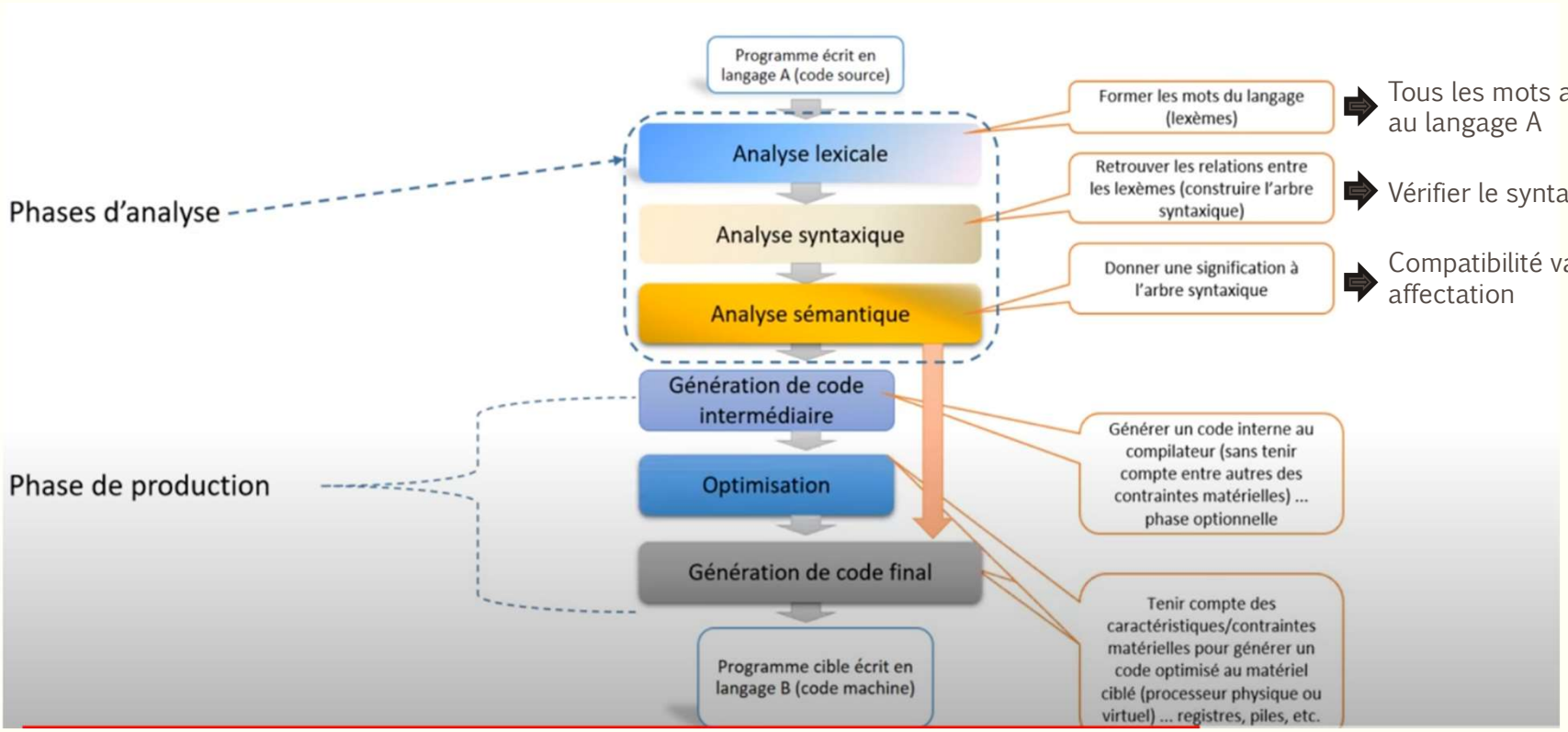
Le code peut être modifié et exécuté immédiatement sans recompiler l'ensemble.

Inconvénients :L'exécution est généralement plus lente, car chaque ligne de code doit être traduite à chaque fois qu'elle est exécutée.

Le programme ne peut pas être exécuté sans l'interprète.

Exemples : Python, Ruby, PHP

Structure d'un compilateur/phase de compilation

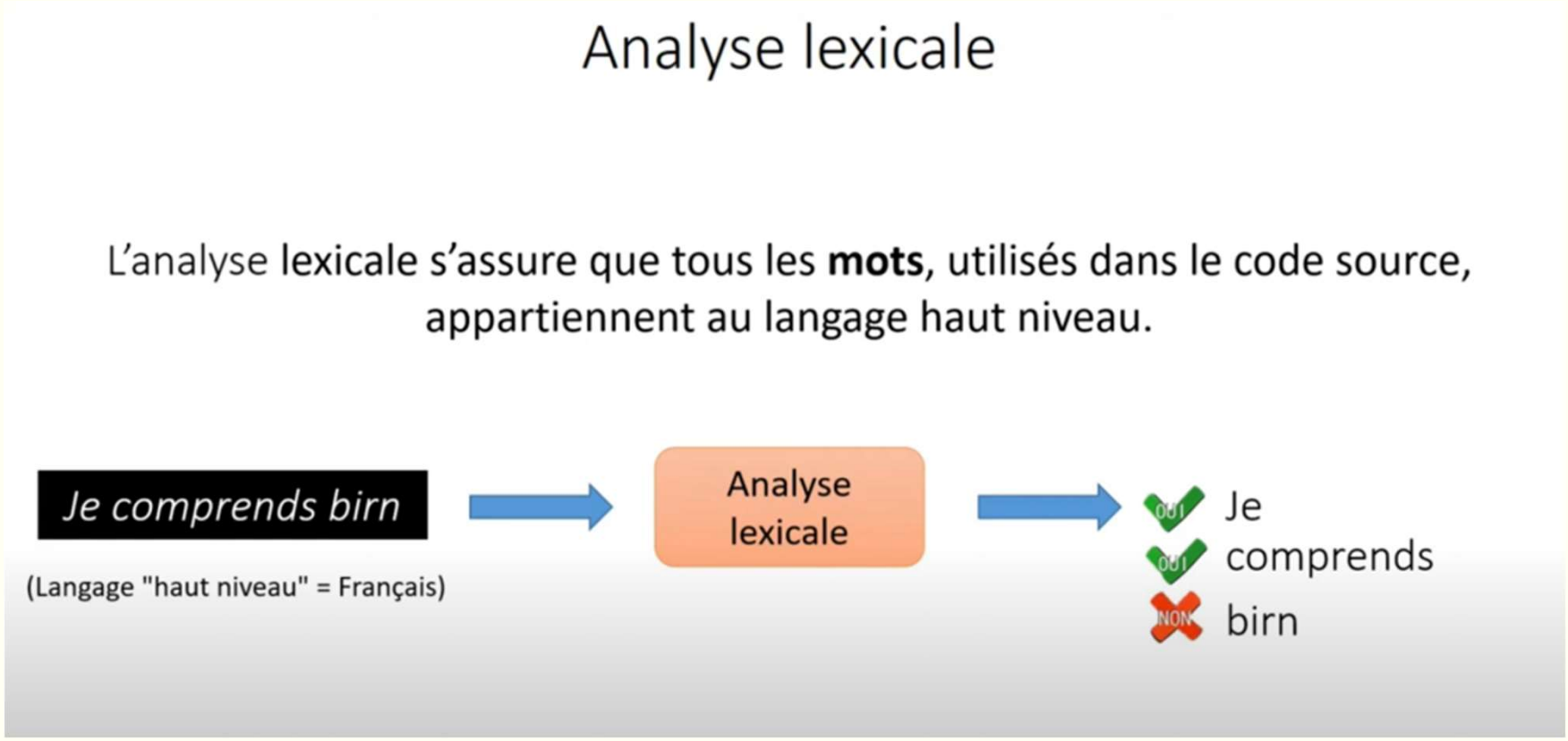


Outils théoriques et pratiques

Phase de la compilation		Outils théoriques
Phases d'analyse	Analyse lexicale scanning (scanning)	Expressions régulières Automates à états finis
	Analyse syntaxique (parsing)	Grammaires
	Analyse sémantiques	Traduction dirigée par la syntaxe
	Génération de code	Traduction dirigée par la syntaxe

PRINCIPE DE L'ANALYSE LEXICALE

Décompose le code source en symboles (tokens) représentant les éléments syntaxiques du langage.



Le rôle d'un analyseur lexical:

L'analyseur lexical ou, aussi appelée « scanner» lit le code source caractère par caractère et le divise en « tokens », qui sont des séquences de caractères avec une signification particulière dans le langage (comme des mots-clés, des identifiants, des opérateurs, etc.).

- éliminer les blancs (espaces, tabulations, fin de lignes) et les commentaires.
- détecter les erreurs et associer des messages d'erreurs.

- Par exemple, pour le code `int a = 5;`, les tokens seraient : `int`, `a`, `=`, `5`, et `;`.
- Le résultat est un flux de tokens qui est transmis à la phase suivante.

Exemple :
int a = 5;
`

Tokens générés : `int`, `a`, `=`, `5`, `;`.

EXEMPLE D'UN LEXEME

Un léxème est une séquence finie de symboles de l'alphabet du langage
Quels sont les lexèmes du code C suivant?

While (ip <z)
ip++;

- *While*
- *(*
- *Ip*
- *<*
- *Z*
- *)*
- *Ip*
- *++*
- *;*

PRINCIPE DE JETON (TOKEN)

Jeton (token)

- La deuxième étape de l'analyse lexicale consiste à valider si chaque lexème appartient ou non au langage.
- Il est impossible d'énumérer tous les lexèmes possibles d'un langage (ip, z, …)

- La validation s'effectue en définissant toutes les parties logiques du fichier source (Jeton), puis associer chaque lexème à un jeton.

  (les lexèmes ip, z sont associés au jeton "variable" qui peut être défini comme : un ensemble de symboles de l'alphabet commençant par une lettre).

- Le lexème qui ne peut être associé à aucun jeton n'appartient donc pas au langage.

EXEMPLE D'UN JETON (token)

Jeton (token)

- Un jeton représente une partie logique du fichier source:
  - Un mot clé : if, while …
  - Une variable : ip, z …
  - Un opérateur : +, < …
  - Un littéral : true, "Canada" …
  - …
- Chaque jeton peut avoir des attributs optionnels qui sont des informations supplémentaires tirées du texte :
  - Par exemple une valeur numérique: les trois caractères "137" sont un lexème associé au jeton "entier" qui a une valeur numérique égale à 137

Analyse Syntaxique (Syntax Analysis)

Objectif : Vérifier que les tokens suivent la grammaire du langage de programmation.

- Cette phase, aussi appelée « analyseur syntaxique » ou « parser », organise les tokens obtenus dans l'étape précédente en une « structure arborescente » appelée « arbre syntaxique abstrait (AST) ».

- L'analyseur vérifie que les tokens respectent les règles grammaticales du langage. Si la syntaxe est incorrecte, des erreurs de syntaxe sont générées.

- L'AST représente la structure hiérarchique du programme et reflète la façon dont les instructions doivent être exécutées. Exemple :`int a = 2+3*4;`
AST produit : un nœud représentant une déclaration de variable `a` de type `int` avec l'expression facteur + facteur * facteur

Analyse Sémantique (Semantic Analysis)

Objectif : Vérifier que les constructions syntaxiques ont un sens logique.

- Dans cette phase, le compilateur vérifie les « règles sémantiques » du langage, comme la validité des types de données, la portée des variables et la compatibilité des opérateurs.
- Par exemple, l'analyseur sémantique vérifiera si une variable est utilisée avant d'être déclarée ou si un opérateur est utilisé avec des types de données incorrects (par exemple, additionner un entier avec une chaîne de caractères).

Exemple :

```
int a = "test"; //  Erreur sémantique
```

Erreur détectée : Incompatibilité de type (`int` ne peut pas être assigné à une chaîne de caractères).

Rôle de la table de symboles

La table de symboles est créée et maintenue par le compilateur afin :

Enregistrer les déclarations : Quand un identificateur est déclaré (comme une variable ou une fonction), le compilateur stocke ses informations dans la table.

Assurer la portée et la visibilité : Elle aide à gérer les portées (scopes) des identificateurs, garantissant qu'ils sont utilisés uniquement là où ils sont visibles.

Faciliter la vérification des types : Lorsqu'une variable ou une fonction est utilisée, la table permet de vérifier que le type et l'utilisation correspondent à la déclaration.

Aider à la génération de code : Les informations sur les identificateurs sont utilisées lors de la génération du code machine ou du bytecode.

Structure de la table de symboles

La table de symboles est généralement implémentée comme une table de hachage ou un arbre, où :

Les clés sont les noms des identificateurs (variables, fonctions, types).

Les valeurs sont des ensembles d'informations associées à chaque identificateur, comme :

- **Le type** de l'identificateur (par exemple, int, float, string).
- **La localisation** en mémoire (ou adresse) pour une variable.
- **La portée** (scope) ou le contexte dans lequel l'identificateur est valide.
- **Les attributs** supplémentaires comme les paramètres d'une fonction, les dimensions d'un tableau, etc.

Exemples

```
int x;float y;
int main() {
int z;     x = 10;     y = 3.14;
}
```

Identificateur	Type	Portée	Adresse/Emplacement	Autres attributs
`x`	`int`	global	adresse1	-
`y`	`float`	global	adresse2	-
`z`	`int`	main()	adresse3	-

```
int x;int main() {
int x;   // Une autre variable x, différente de la première
}
```

Dans ce cas, la table de symboles doit avoir une hiérarchie pour comprendre que x dans le bloc main() est différent de x dans le contexte global.

Pour une fonction, la table de symboles stocke généralement :

- Le type de retour.
- La liste des paramètres avec leur type.
- Le bloc ou la portée dans laquelle la fonction est définie.

```
int add(int a, int b) {
    return a + b;
}
```

Identificateur	Type	Portée	Adresse/Emplacement	Autres attributs
`add`	`int (int, int)`	global	adresse4	Paramètres : `a: int`, `b: int`
`a`	`int`	add()	adresse5	-
`b`	`int`	add()	adresse6	-

Une grammaire context-free

(grammaire hors-contexte ou CFG, pour Context-Free Grammar en anglais) est un concept central en théorie des langages formels et en compilation.
Il s'agit d'un type de grammaire formelle qui est utilisé pour définir la syntaxe des langages de programmation ou des langages formels en général.
Voici une explication des concepts clés associés :

Définition Une grammaire context-free est une collection de règles de réécriture qui produisent des chaînes de symboles. Elle est composée de quatre éléments :

1. N : Un ensemble de non-terminaux (ou variables), qui représentent des catégories syntaxiques (par exemple, une expression arithmétique, une fonction).
2. Σ : Un ensemble de terminaux, c'est-à-dire les symboles du langage (par exemple, des mots-clés, des opérateurs).
3. P : Un ensemble de règles de production, de la forme $A \rightarrow \alpha$, où A est un non-terminal et α est une chaîne de terminaux et/ou de non-terminaux.
4. S : Un symbole de départ (un non-terminal particulier), à partir duquel la génération de la syntaxe commence.

Règles de production

Les règles de production d'une grammaire context-free permettent de remplacer un non-terminal par une séquence de symboles (terminaux et/ou non-terminaux). Le caractère "hors-contexte" signifie que la réécriture d'un non-terminal ne dépend pas du contexte dans lequel il apparaît, mais uniquement du non-terminal lui-même.
Une règle de production typique ressemble à : $A \rightarrow B\ C \mid a$. Cela signifie que le non-terminal A peut être réécrit soit comme la séquence des non-terminaux B C, soit comme le terminal a.
Exemple :

Terminaux (Σ) :

- $+, *, (,), \text{number}$

Non-terminaux (N) :

- $Expr$ (expression)
- $Term$ (terme)
- $Factor$ (facteur)

Symbole de départ (S) :

- $Expr$

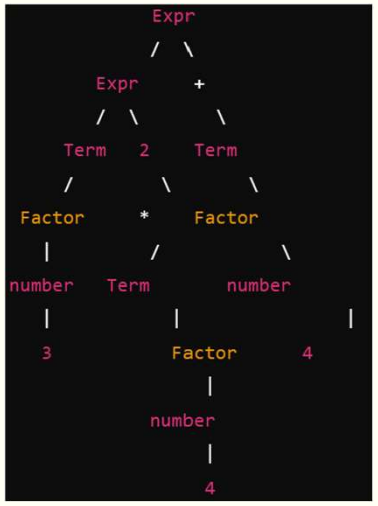
Règles de production (P) :

1. $Expr \rightarrow Expr + Term$
2. $Expr \rightarrow Term$
3. $Term \rightarrow Term * Factor$
4. $Term \rightarrow Factor$
5. $Factor \rightarrow (Expr)$
6. $Factor \rightarrow \text{number}$

Parse Tree ou arbre syntaxique concret

Un parse tree (ou arbre de dérivation, ou encore arbre syntaxique concret) est une structure de données arborescente qui représente la manière dont une chaîne d'entrée (comme une instruction ou un programme) est dérivée à partir des règles d'une grammaire.

Entrée à analyser :2+3*4
Voici le parse tree pour cette expression
selon la grammaire donnée :

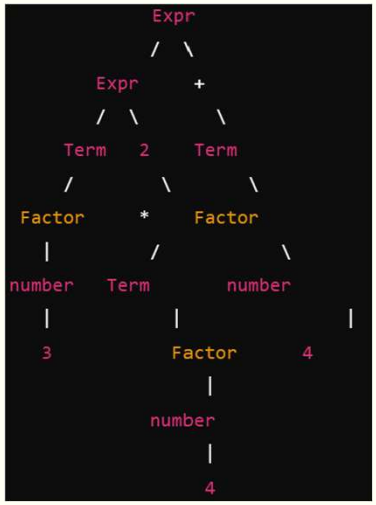


Les nœuds internes correspondent à des non-terminaux de la grammaire.
Les feuilles (les nœuds les plus bas) représentent les symboles terminaux du langage (comme des opérateurs, des mots-clés, des variables, etc.).
La racine de l'arbre est généralement le symbole de départ de la grammaire.
Les branches de l'arbre illustrent l'application des règles de production de la grammaire.

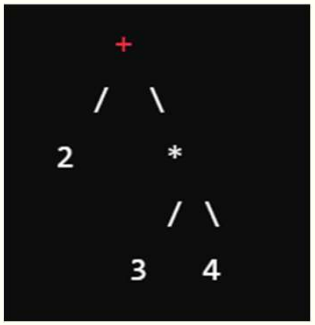
Parse Tree vs l'arbre de syntaxe abstraite (AST),

Facilite l'analyse syntaxique : Le parse tree montre explicitement la structure hiérarchique d'une expression ou d'une instruction en fonction des règles de la grammaire.
Aide à la détection d'erreurs : En analysant la structure du parse tree, le compilateur peut identifier rapidement des erreurs syntaxiques dans le programme source.
Base pour la construction d'autres structures :
Le parse tree est souvent une première étape pour construire l'AST, qui est utilisé pour des analyses plus poussées et pour la génération du code.

Parse Tree



AST



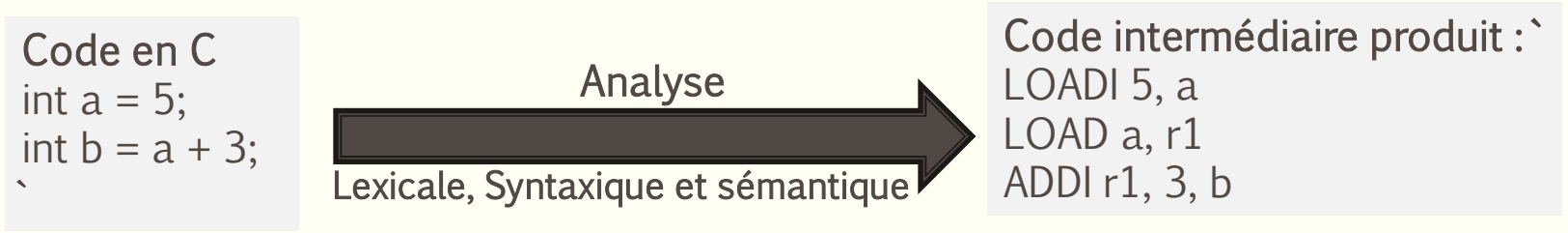
Optimisation Intermédiaire (Intermediate Code Generation)

Objectif : Traduire le code source en un code intermédiaire.

- Le compilateur convertit le programme en un « langage intermédiaire » ou « code intermédiaire » qui est plus simple et proche du code machine, mais indépendant de la plateforme matérielle.

- Ce code est souvent plus abstrait et simplifié, et il facilite les optimisations.
- Ce code intermédiaire peut ensuite être optimisé avant d'être traduit en code machine.

Exemple



Optimisation (Optimization)

Objectif : Améliorer l'efficacité du code sans changer son comportement.

- Le compilateur tente de rendre le code plus rapide, plus efficace ou moins gourmand en mémoire.
- Il peut s'agir de la « réduction des redondances », de la « réutilisation des calculs », de « l 'élimination du code mort » (instructions inutiles), ou encore de la « fusion de boucles ».
- Les optimisations peuvent être de haut niveau (sur le code source) ou de bas niveau (sur le code machine).

Exemple d'optimisation :

```
int a = 2 * 3; // Devient simplement
int a = 6;
```

Disposition de titre et de contenu avec graphique SmartArt