COMPILATION
ET
THÉORIE DES LANGAGES

Dr. Abdelali El Gourari

Dr. Abdelali El Gourari

Doctor of Computer Science specializing in Artificial Intelligence and Embedded Systems at Cadi Ayyad University, Morocco. His primary research interests are applying artificial intelligence to adaptive education systems, particularly remote-controlled experiments.

mail

a.elgourari@emsi.ma
a.elgourari.ced@uca.ac.ma

Websites

ai-labs.uca.ma
elgourari.com

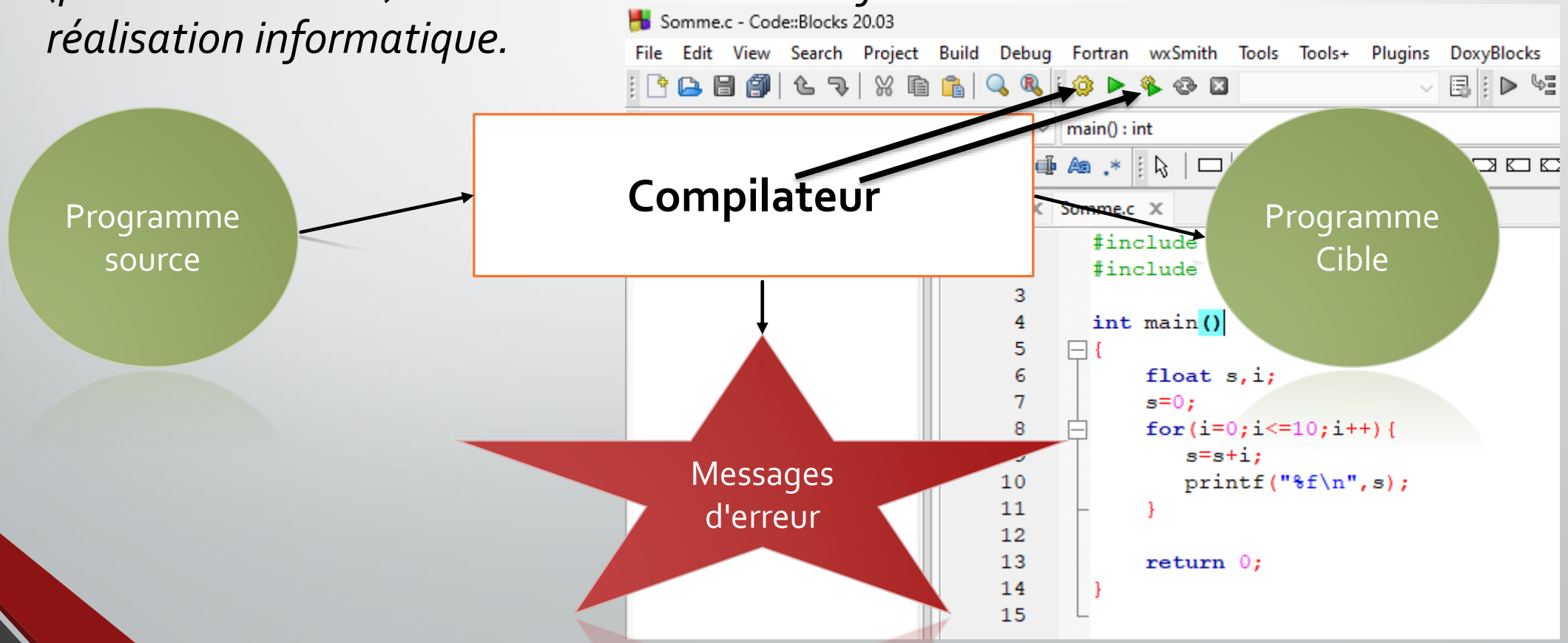
Présentez vous

Plan

- *Introduction*
- *Les objectives du Cours*
- *L'environnement d'un compilateur*
- *Cousins des compilateurs*
  - *Interpréteurs*
  - *Préprocesseurs*
  - *Langages Intermédiaire ou P-code*
  - *Éditeur de liens*
  - Le Chargeur
- *Langages d'assemblage*
- *Assemblage*
- *Langages machines*
- *Structure d'un compilateur*
- *Analyse lexicale*
- *Analyse syntaxique*
- *Analyse sémantique*
- *Génération de code*
- *Optimisation de code*

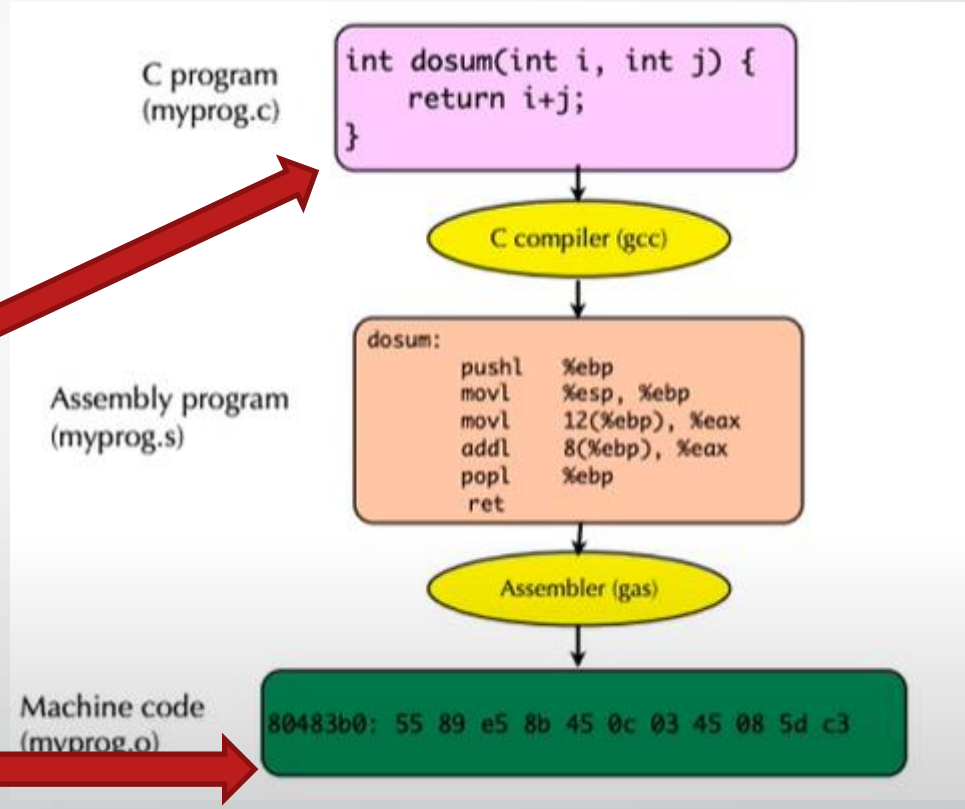
Introduction

*Un compilateur est un **logiciel particulier** qui **traduit** un programme écrit dans un **langage de haut niveau** (par le programmeur) en **instructions exécutables** (par un ordinateur). C'est donc l'instrument fondamentale à la base de tout réalisation informatique.*



Introduction

- *Premier compilateur : **compilateur Fortran** de J. Backus (1957)*
- *Langage source : **langage de haut niveau** (C, C++, Java, Pascal,…*
- *Langage cible : **langage de bas niveau** (assembleur, langage machine)*



Introduction

- ***La compilation** est la **traduction** d'un langage de programmation de **haut niveau** vers un autre langage de programmation de **haut niveau**.*
- *Une traduction de **Pascal** vers **C**, ou de **Java** vers **C++**, on parle plutôt de **traducteur***
- *La différence entre **traducteur** et **compilateur** : dans **un traducteur**, il n'y a pas de **perte d'informations** (on garde les commentaires, par exemple), alors que dans un **compilateur** il y **a perte** d'informations.*
- *La traduction d'un langage de programmation **de bas niveau** vers un autre langage de programmation de **haut niveau**. Par exemple pour retrouver le code C à partir d'un code compilé (**piratage**, **récupération de vieux logiciels**, ...)*

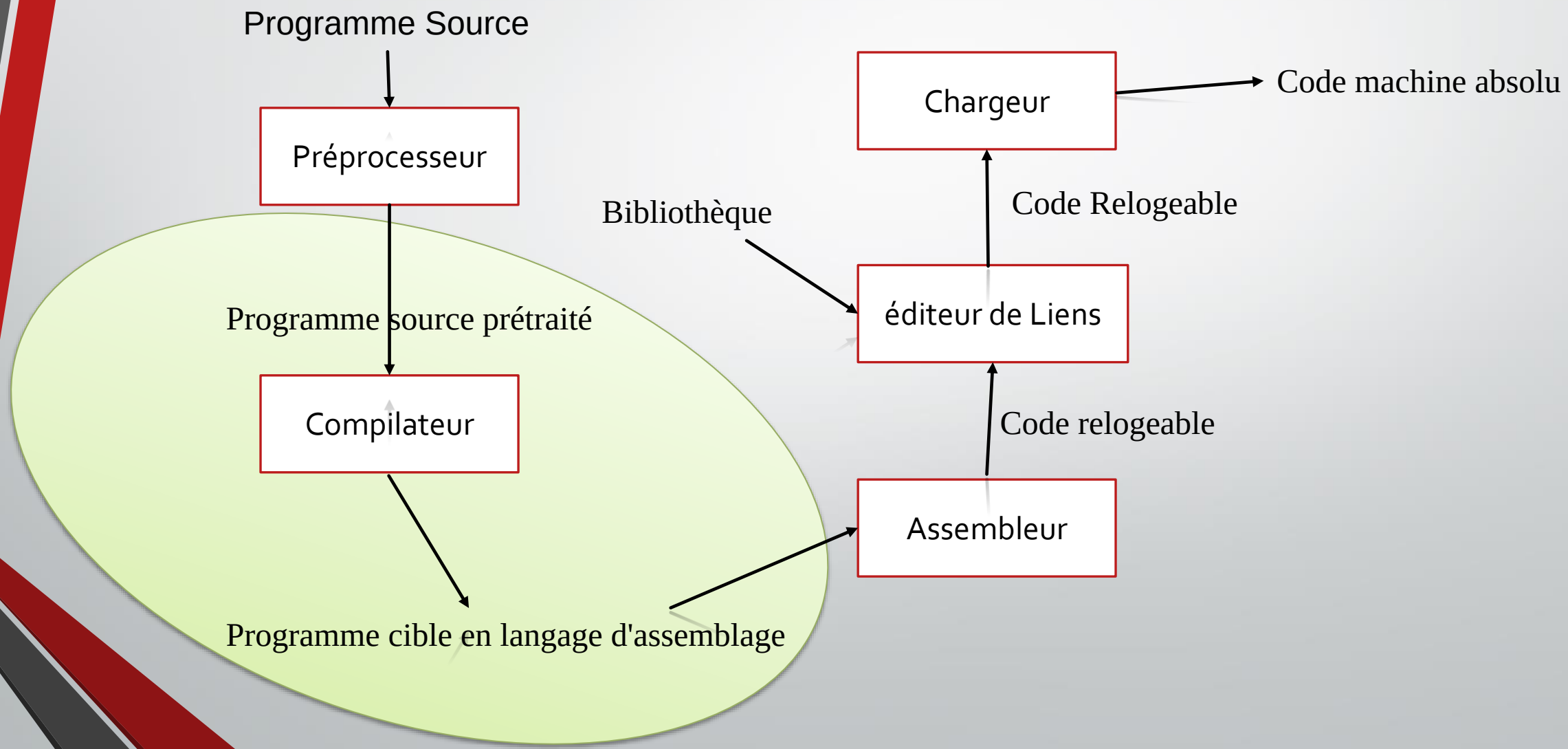
Les objectives du Cours

Le but de ce cours est de présenter ***les principes de base*** *inhérents à la* ***réalisation de compilateurs****.*

Les objectives du Cours

- *Les principes de base inhérents à la **réalisation** de compilateurs : **analyse lexicale, analyse syntaxique, analyse sémantique, génération de code**.*
- *Les outils fondamentaux utilisés pour effectuer ces analyses : **Grammaires**, **Automates à état fini**, **méthodes algorithmiques d'analyse**…*
- *Comprendre **comment est écrit un compilateur** permet de mieux comprendre les contraintes imposées par les différents langages lorsque l'on écrit un programme dans un langage de **haut niveau**.*

L'environnement d'un compilateur



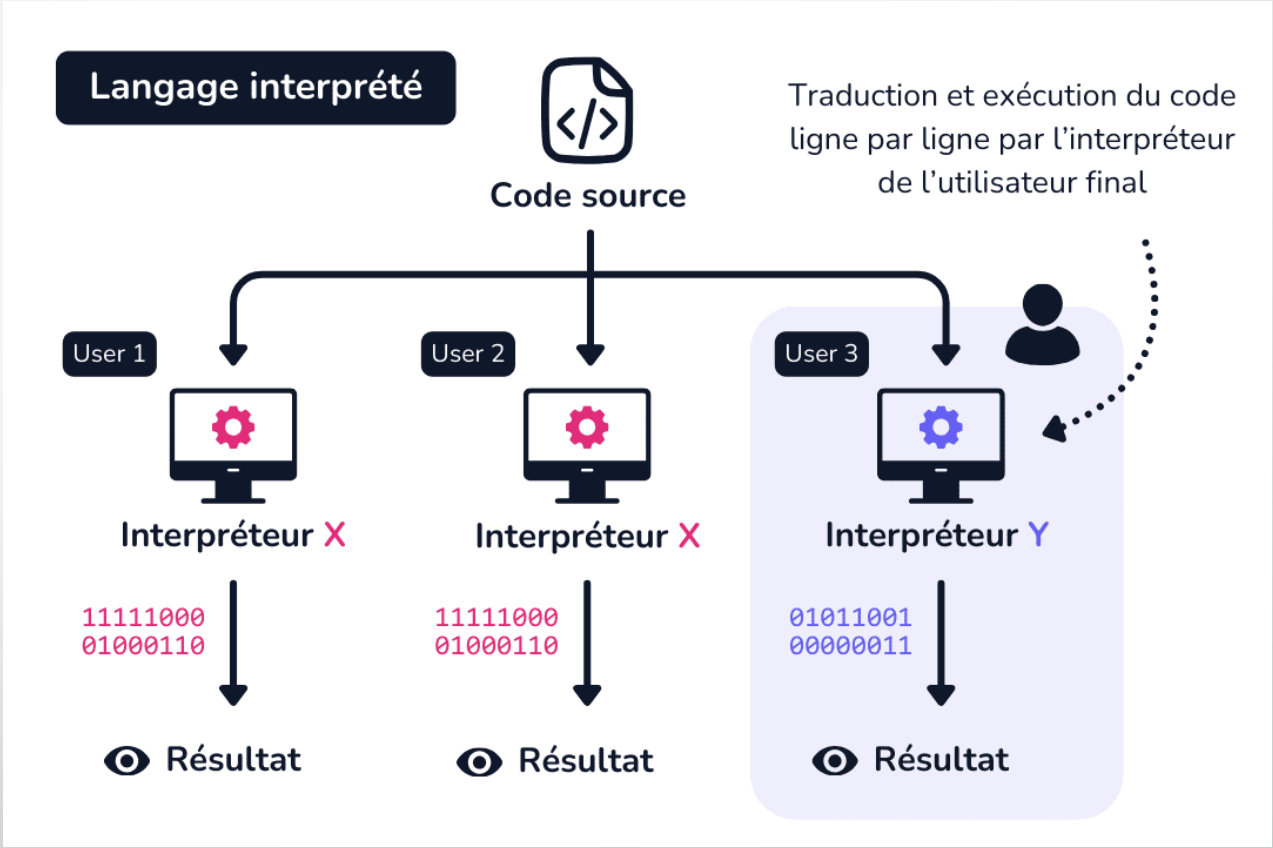
Cousins des compilateurs

- *Interpréteurs*
- *Préprocesseurs*
- *Langages Intermédiaire ou P-code*
- *Éditeur de liens*
- *Le chargeur*

Interpréteurs

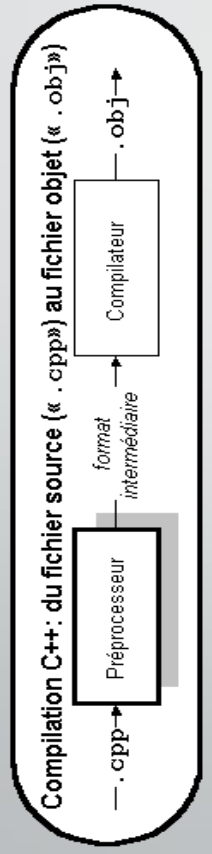
- ***L'interpréteur*** ➔ *analyse une instruction après l'autre puis l'exécute.*
- ***Un compilateur*** ➔ *Il travaille simultanément sur le programme et sur les données.*

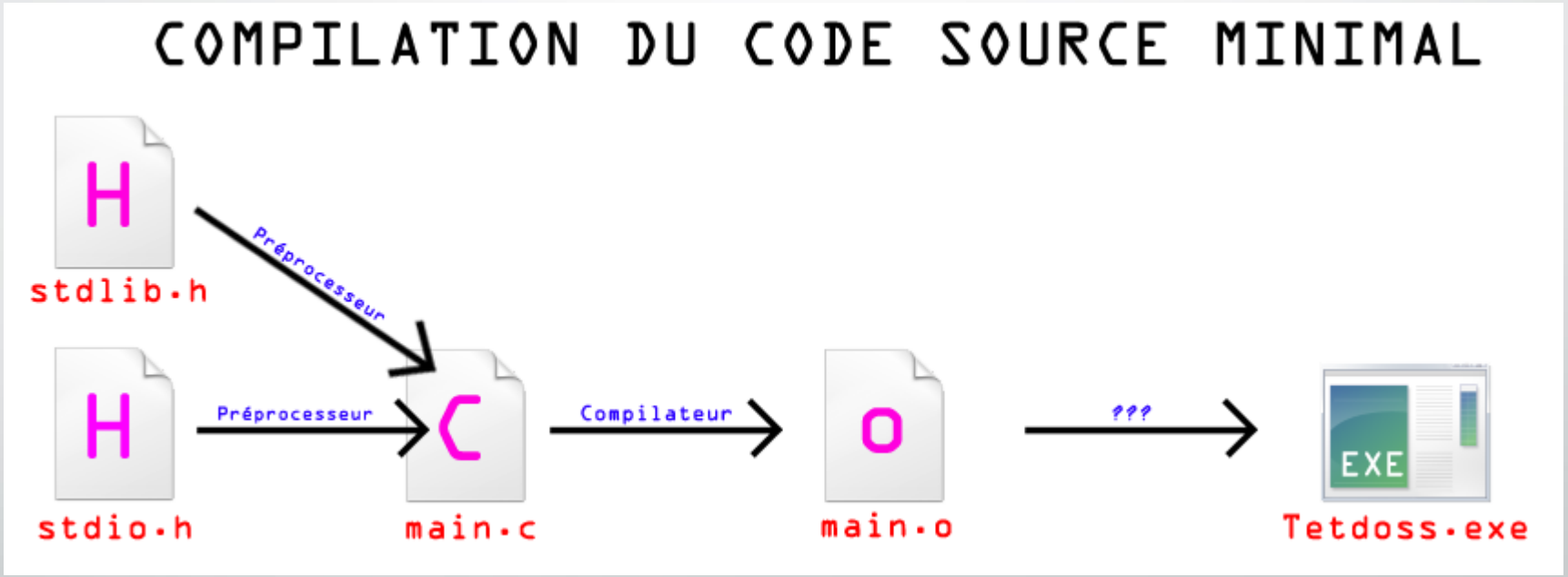
Exemples de langages interprétés : Python, BASIC, Perl, Prolog…



Préprocesseurs ➔ .cpp

*Le préprocesseur est un **programme** qui **analyse votre code source** et y effectue des **modifications** avant la compilation.*





https://www.google.com/url?sa=i&url=https%3A%2F%2Fyard.onl%2Fsitelycee%2Fcours%2Fc%2FCompilationSeparee.html&psig=AOvVaw0SKl4NAbvHgsrmRJPyX302&ust=1728408833048000&source=images&cd=vfe&opi=89978449&ved=0CBcQjhxqFwoTCNDojPfm_IgDFQAAAAAdAAAAABAb

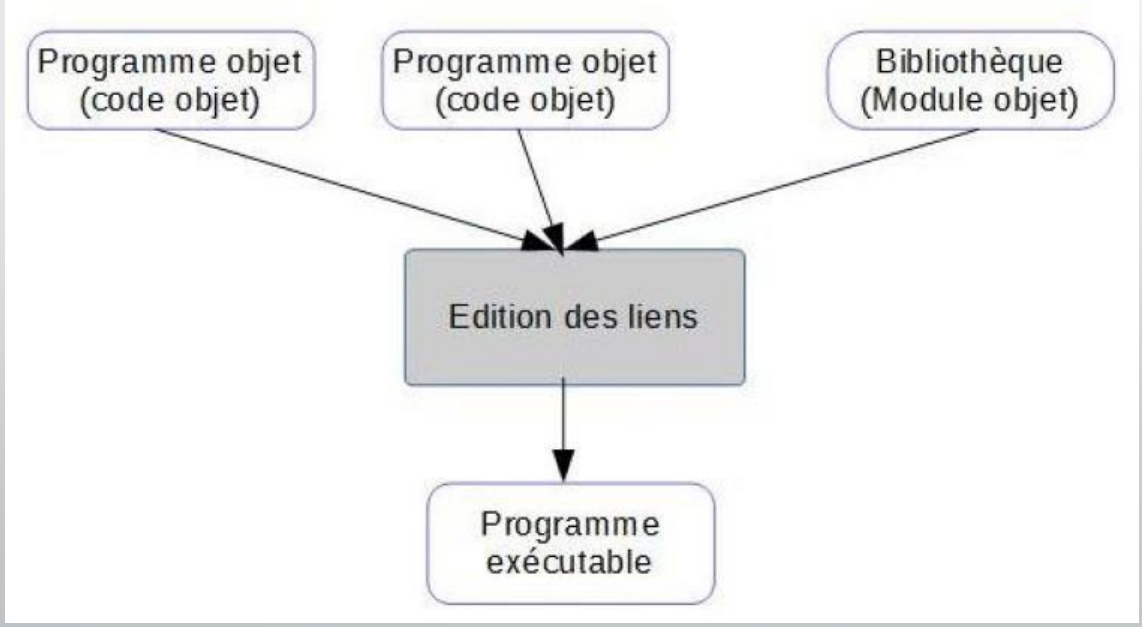
Langages Intermédiaire ou P-code

- *Les langages **P-code** ou **langages intermédiaires** sont à mi-chemin de l'**interprétation** et de la **compilation**.*
- *Le code source est **traduit** (compilé) dans une **forme binaire** compacte (du pseudo-code ou p-code) qui n'est pas encore du code machine. Ce **P-code** est interprété pour exécuter le programme.*
- *Par exemple en **Java**, le code source est compilé pour obtenir un fichier (**.class**) "byte code" qui sera **interprété** par une **machine virtuelle**.*

L'édition des liens (**Linker**)

- ***L'édition des liens*** *consiste à relier les codes de chaque fonction de bibliothèque utilisée avec le programme.*

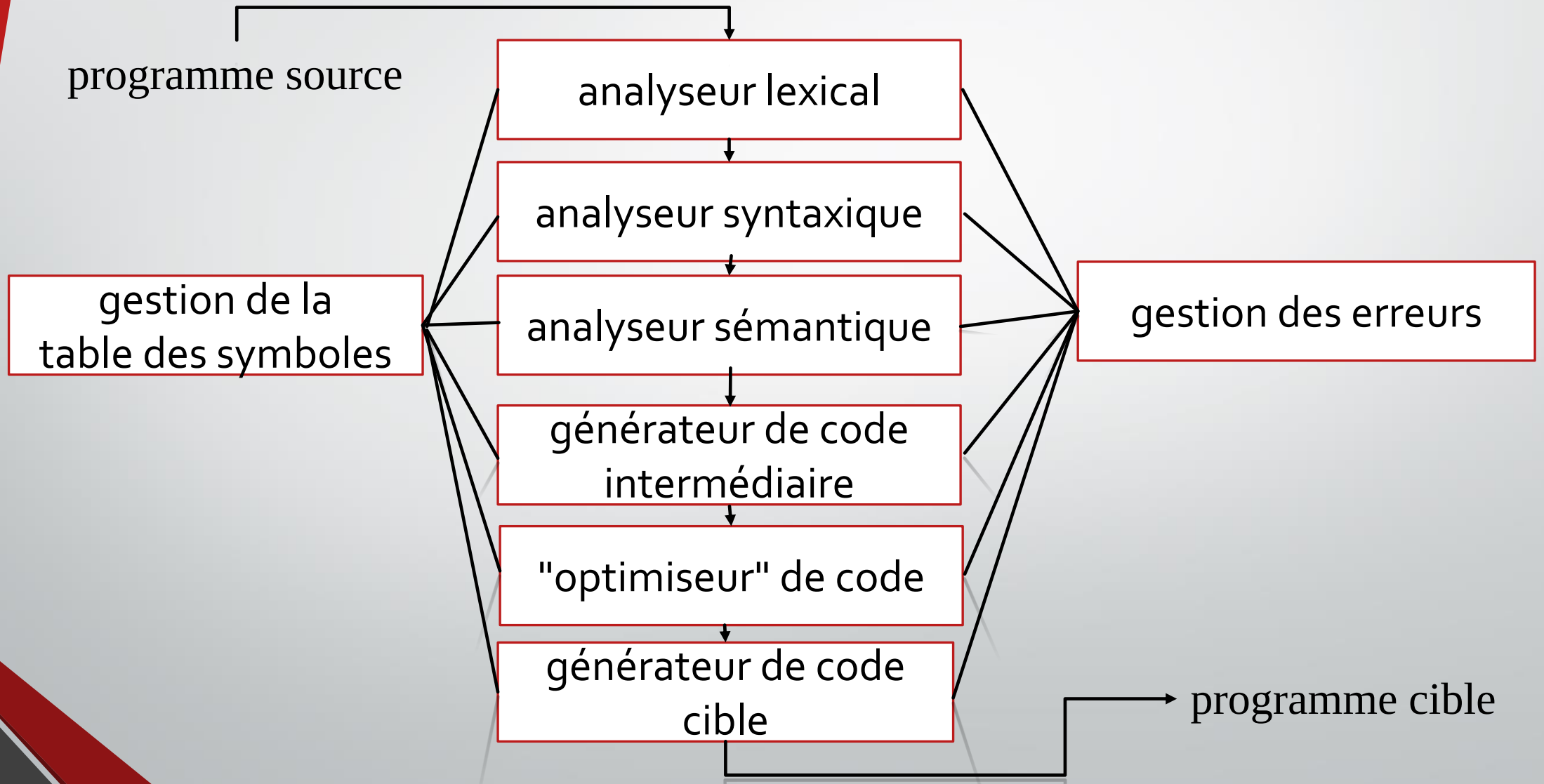
- *Le résultat de l'édition des liens est un programme exécutable,*



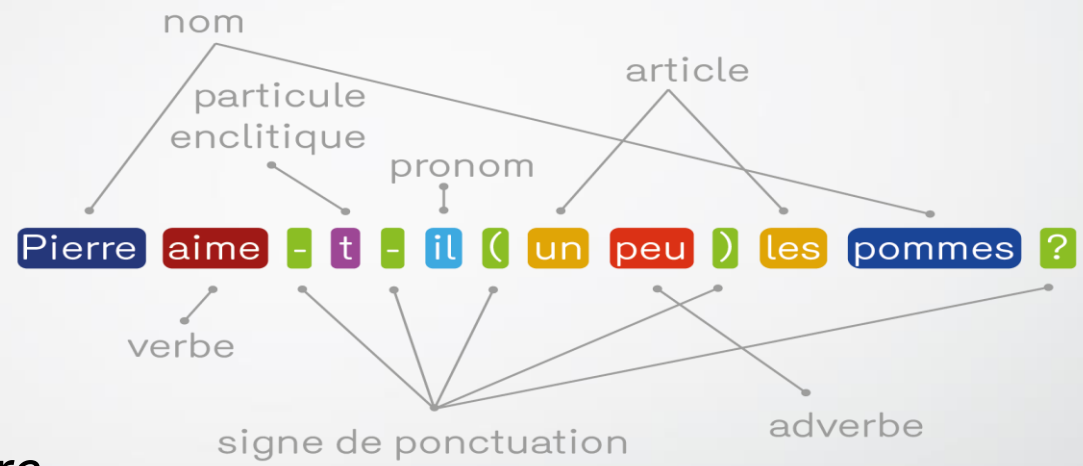
Le Chargeur

- *Le **programme objet**, obtenu après **l'édition de liens**, doit encore être **chargé en mémoire centrale** pour être **exécuté**. **Le chargeur** s'occupe de cette tâche.*
- *Dans les anciens systèmes (comme DOS), on pouvait **fixer les adresses** à l'avance et charger le programme à l'endroit spécifié. On utilisait donc un **chargeur absolu**.*
- *Aujourd'hui, les chargeurs s'occupent de reloger les programmes en mémoire centrale. Ce sont des **chargeurs relogeables (transformable)**.*

Structure d'un compilateur



Analyse lexicale(Scanning)

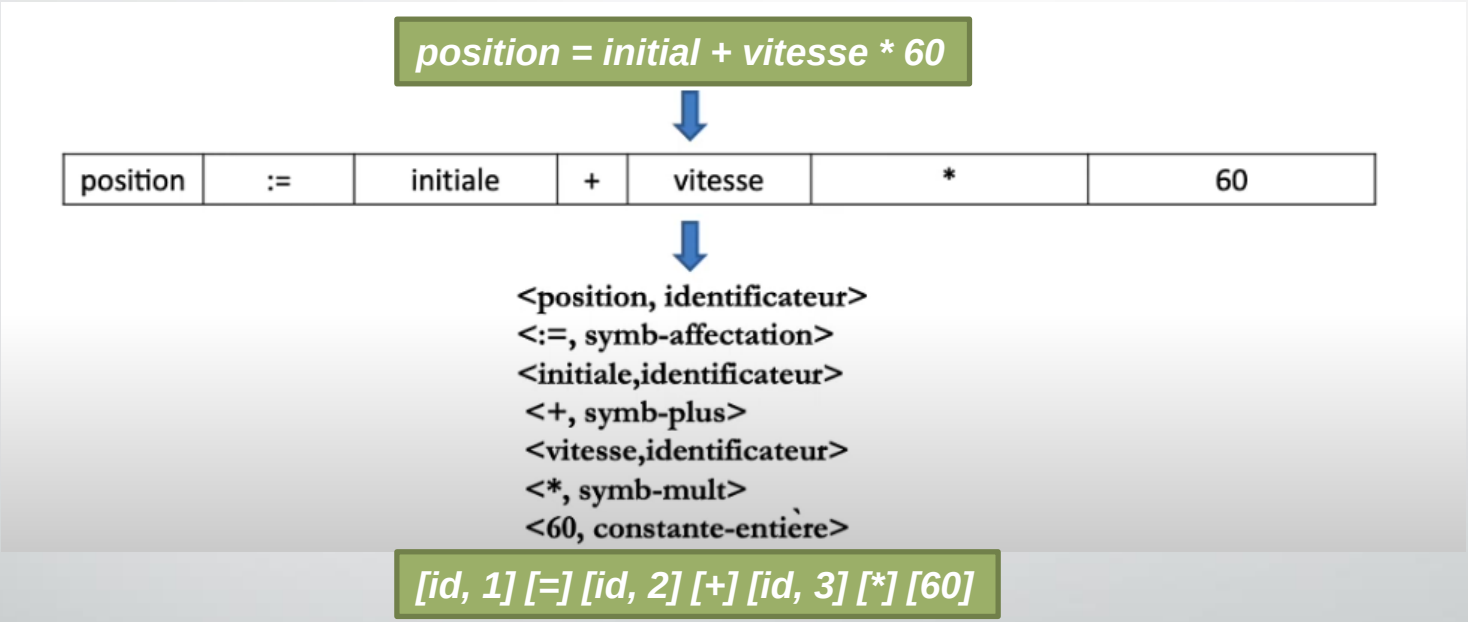


Appelée aussi Analyse linéaire

- ***Reconnaître** les types des **mots** lus. Pour cela, on lit le programme source de gauche à droite et les caractères sont regroupés en **unités lexicales**.*
- *Se charger **d'éliminer** les caractères superflus (**commentaires, espaces, …**)*
- ***Identifier** les parties du texte qui ne font pas partie à proprement parler du programme mais sont des directives pour le compilateur .*

Analyse lexicale(Scanning)

*Outils théoriques utilisés : **expressions régulières** et **automates à états finis***



*Les **identificateurs** rencontrés sont placés dans la **table des symboles**.*

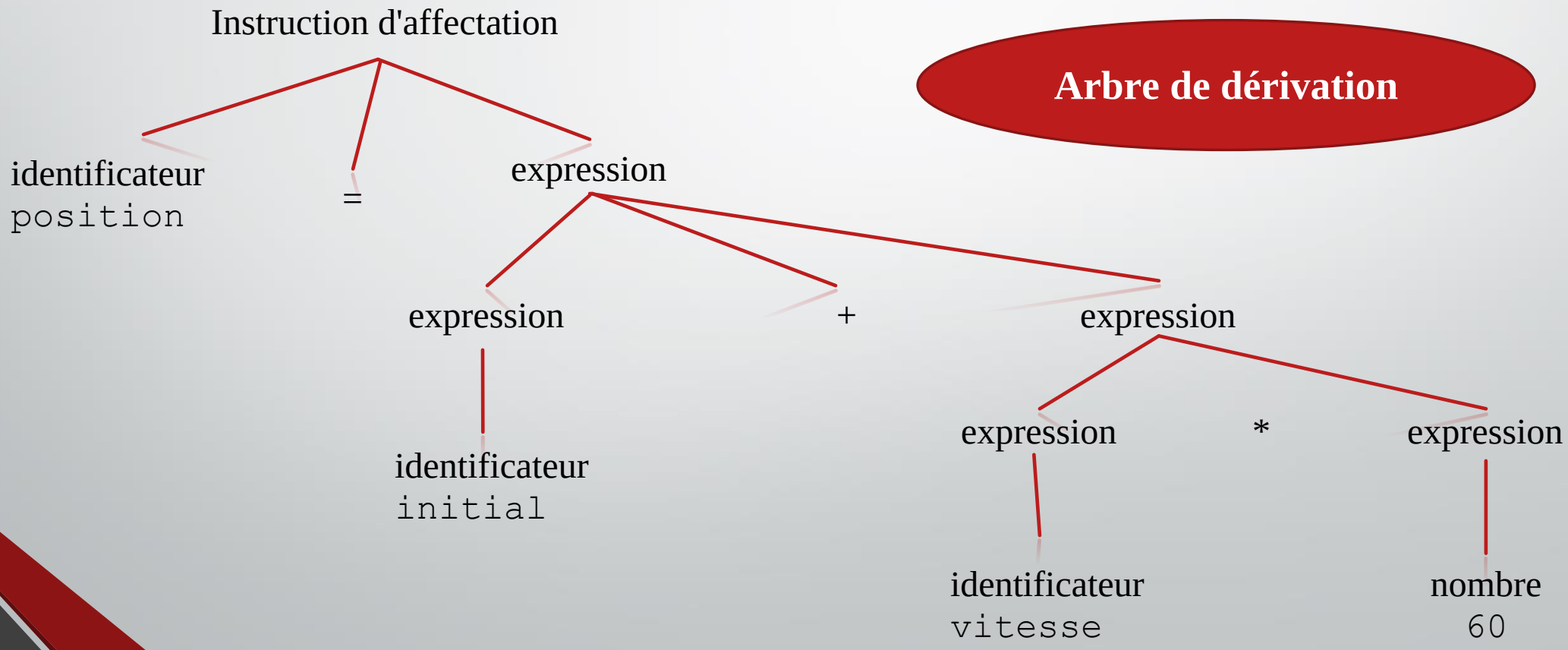
Les blancs et les commentaires sont éliminés.

Analyse syntaxique

- Appelée aussi *analyse* ***grammaticale***
- Il s'agit de **regrouper** les **unités lexicales** en **structures grammaticales**, de découvrir la structure du programme.
- L'analyseur syntaxique sait comment doivent être construites les **expressions**, les **instructions**, les **déclarations de variables**, les **appels de fonctions**, …

Analyse syntaxique

On reconstruit la **structure syntaxique** de la suite d'unités lexicales fournie par **l'analyseur lexical**.

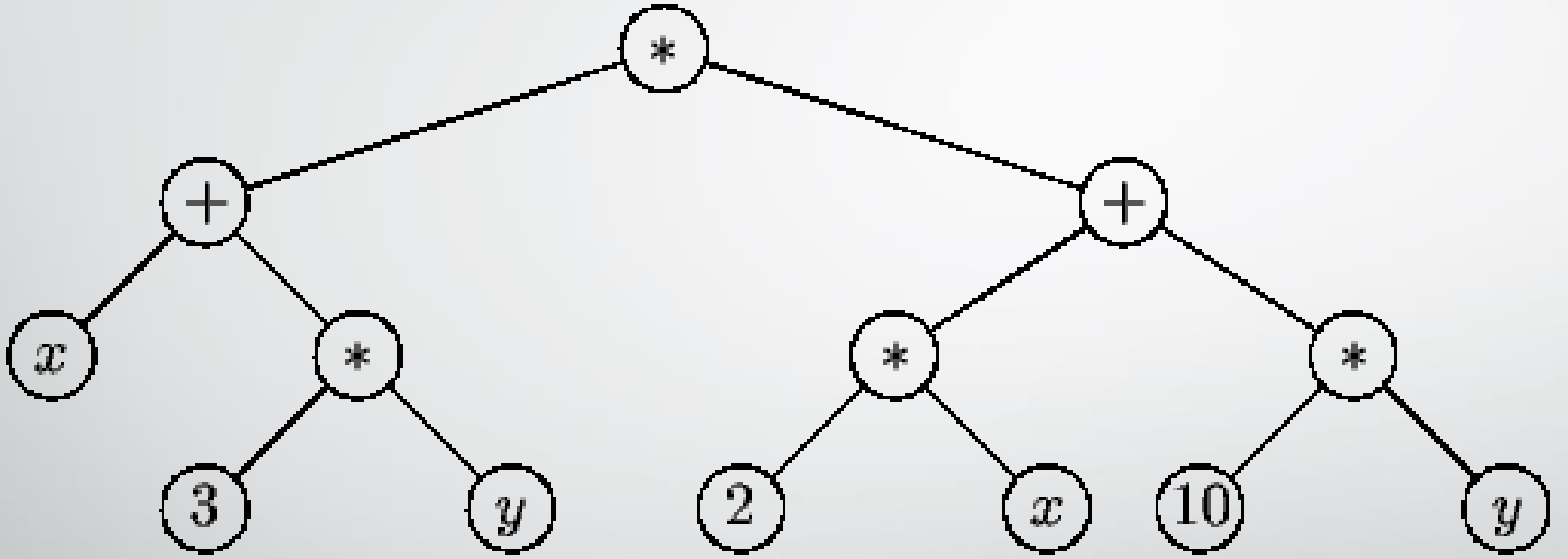


Analyse syntaxique

```
while (b != o) {
        if (a > b)
                a = a - b;
        else
                b = b - a;
            }
return a;
```

Donner l'arbre de dérivation?

Analyse syntaxique



?

Exercice

Analyse Lexicale

Créez un analyseur lexical pour un langage simple avec les règles suivantes :

- Les mots-clés sont if, else, while, return.
- Les identificateurs commencent par une lettre et peuvent contenir des lettres et des chiffres.
- Les nombres sont des entiers composés uniquement de chiffres (ex: 123).
- Les opérateurs sont : +, -, *, /, =, ==, !=.
- Les espaces et les tabulations doivent être ignorés.

1. **Écrivez des règles lexicales qui décrivent comment votre analyseur identifierait les tokens dans la phrase suivante :**

```
if (x == 10) return x + 1;
```

Analyse Syntaxique

2. **Construisez l'arbre syntaxique pour l'expression suivante :**

```
if (x + 1 == y) return x * 2;
```

Exercice

Analyse Sémantique

Supposons que vous ayez les règles de typage suivantes pour un langage de programmation simple :

- ✓ Les identificateurs doivent être déclarés avant utilisation.
- ✓ Les opérandes des opérateurs arithmétiques doivent être des entiers.
- ✓ Les expressions booléennes doivent retourner des booléens (**true** ou **false**).

Pour le code suivant, identifiez et expliquez les erreurs sémantiques éventuelles :

```
int x;
x = 5;
if (x + true)
return 1;
```

Sol_Exercice

Analyse Lexicale

```
if          -> MOT_CLÉ
(           -> PARENTHÈSE_OUVRANTE
x           -> IDENTIFICATEUR
==          -> OP_COMPARAISON
10          -> ENTIER
)           -> PARENTHÈSE_FERMANTE
return      -> MOT_CLÉ
x           -> IDENTIFICATEUR
+           -> OP_ARITHMÉTIQUE
1           -> ENTIER
;           -> POINT_VIRGULE
```

Analyse Syntaxique

Analyse Syntaxique

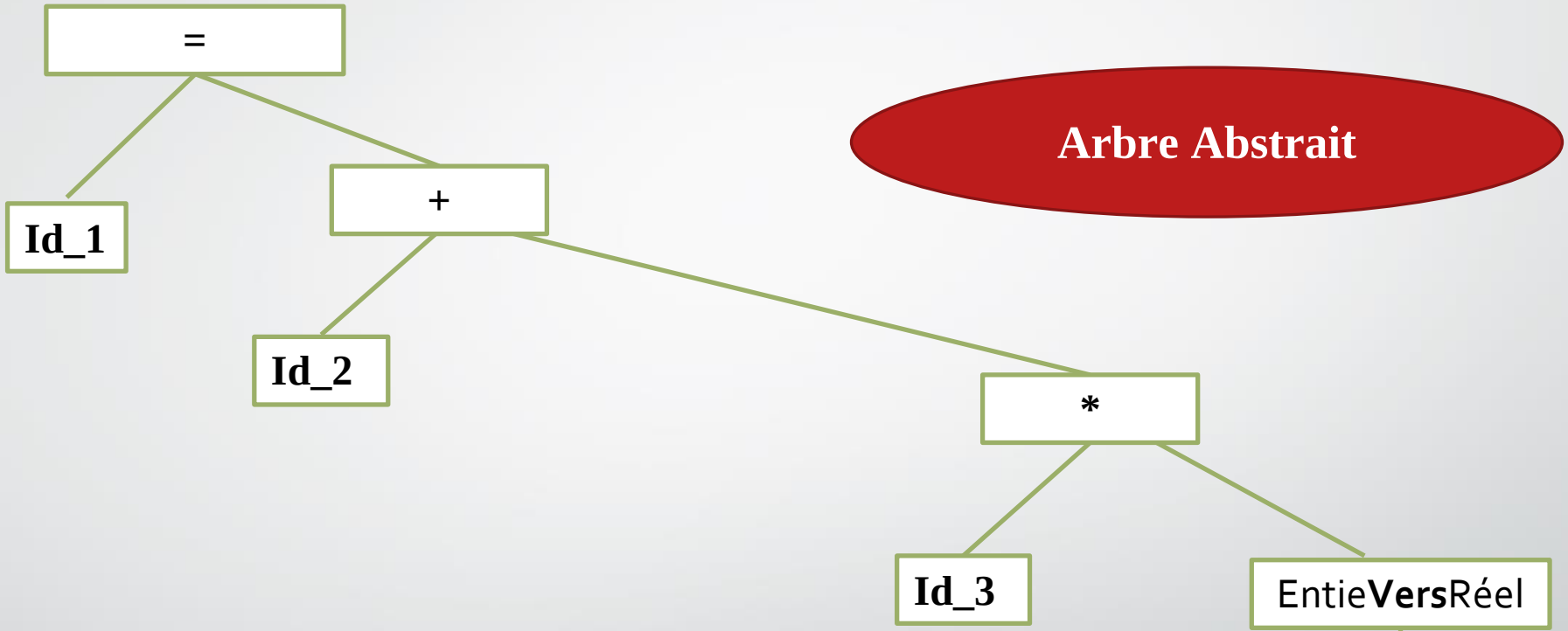
L'opérateur **+** ne peut être appliqué qu'entre des entiers. Cependant, dans l'expression **x + true**, **x** est un entier et **true** est un **booléen**, ce qui provoque une erreur de **typage**.

Analyse Sémantique

Appelée aussi analyse **contextuelle**

- Dans cette phase, on opère certains contrôles (contrôles de type, par exemple) afin de **vérifier** que **l'assemblage des constituants du programme** a un **sens.**
- On ne peut pas, par exemple, **additionner un réel avec une chaîne de caractères**, ou **affecter une variable à un nombre**, …
- Outil théorique utilisé : **schéma de traduction dirigée par la syntaxe**

Analyse Sémantique



Arbre Abstrait

Dans cet exemple **d'analyse sémantique**, la conversion de `**60**` en réel (`**60.0**`) est nécessaire pour assurer la cohérence des types dans l'expression entière, et ce processus de conversion est un élément essentiel de l'analyse sémantique. Pourquoi de cette conversion et de ses effets :

Cohérence des types: Si `**initial**` et `**vitesse**` sont des **réels** (flottants), le produit `**vitesse * 60**` doit aussi être un **réel** pour éviter une incompatibilité de types. Dans ce contexte, l'entier `60` est converti implicitement en réel (`**60.0**`) pour que le type soit homogène dans l'expression.

Génération de code

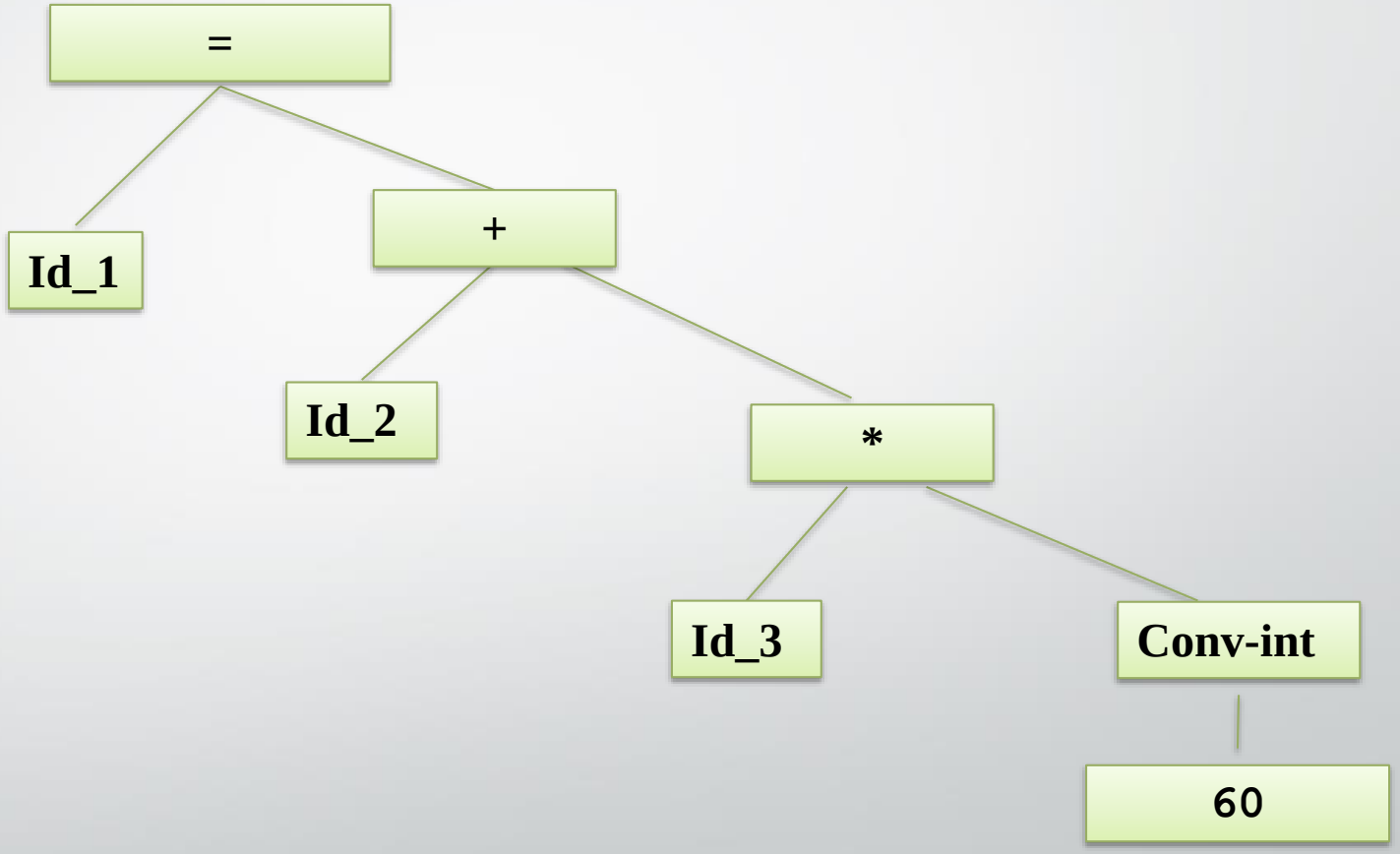
Code à trois adresses

t1 := int**to**real(60)

t2 := id3 * t1

t3 := id2 + t2

id1 := t3



Optimisation de code

Elimination des opérations inutiles pour produire du code plus efficace.

t1 := inttoreal(60)

t2 := id3 * t1

t3 := id2 + t2

id1 := t3

Les variables t2 et t3 sont éliminées

t1 := id3 *60.0

id1 := id2 + t1

- La constante est traduite en réel flottant à la compilation, et non à l'exécution

Cela permet **d'optimiser les performances** puisque le travail de conversion est fait une seule fois (à la compilation) plutôt qu'à chaque exécution du programme. C'est une technique classique d'optimisation qui fait partie des optimisations à la compilation.

Génération de code cible

- La dernière phase produit du code en langage d'assemblage
- Un point important est l'utilisation des registres

```
t1 := id3 * 60.0
id1 := id2 + t1
```



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

- F = flottant.
- La première instruction transfère le contenu de id3 dans le registre R2
- La seconde multiplie le contenu du registre R2 par la constant 60.0

Exercice : Génération et optimisation de code

- ***Générateur de code***

Écrire un générateur de code pour une petite machine virtuelle. Le programme doit être capable de traduire une expression en code à **trois adresses**, puis d'**optimiser** ce code en particulier le nombre d'instructions.

Expression à traiter : result = (a + b) * (a - b) + (a * b)

Étape 1 : Génération du code à trois adresses

- Construire un code intermédiaire **à trois adresses** pour l'expression donnée.
- Écrivez chaque étape de calcul dans l'ordre correct en utilisant des **variables temporaires** pour chaque **sous-expression**.

- ***Optimisation du code***

Appliquer des optimisations au code à **trois adresses** générées pour réduire le nombre d'instructions.

_Solution_Exercice_

Techniques d'optimisation suggérées :

- Élimination des expressions communes : Par exemple, si **a * b** est calculé plusieurs fois, faites-le une seule fois et réutilisez le résultat.
- Propagation des constantes : Si certaines valeurs sont des constantes, remplacez les variables par leurs valeurs.
- Élimination des instructions mortes : Supprimez les calculs de variables temporaires inutilisées.

```
result = (a + b) * (a - b) + (a * b)
```

Génération du code à trois adresses

```
T1 := a + b
T2 := a - b
T3 := T1 * T2
T4 := a * b
T5 := T3 + T4
result := T5
```

Code optimisé

```
T1 := a + b
T2 := a - b
T3 := T1 * T2
T4 := a * b
result := T3 + T4
```

```
T1 := a + b      ; 3 adresses : T1, a, et b
T2 := a - b      ; 3 adresses : T2, a, et b
T3 := T1 * T2    ; 3 adresses : T3, T1, et T2
T4 := a * b      ; 3 adresses : T4, a, et b
T5 := T3 + T4    ; 3 adresses : T5, T3, et T4
result := T5     ; 2 adresses : result et T5
```

Chaque instruction suit la règle du code à trois adresses (**destination + deux opérandes**), même si plusieurs variables temporaires sont utiles.

Génération et optimisation de code pou les structures conditionnelles et les boucles

Pour traduire un code contenant les **structures conditionnelles** et les **boucles** en instructions intermédiaires en **trois adresses**, nous devons :

- ✓ Créer des **instructions conditionnelles** pour simuler les boucles (**while**, **for**, **do while**).
- ✓ Créer des **labels** pour gérer les sauts de contrôle dans la boucle.
- ✓ Utiliser des **variables temporaires** pour stocker les résultats des opérations.

Génération et optimisation de code

- ***Génération de Code Intermédiaire en Trois Adresses***

```
if (x == 10) return x + 1;
```

```
L1:   if x != 10 goto L2       // Si x n'est pas égal à 10, sauter à L2
      t1 = x + 1               // Sinon, calculer x + 1
      return t1                // Retourner le résultat
L2:   // Rien (instruction vide si condition non satisfaite)
```

Explication

- ✓ **L1** : Vérifie si x est égal à 10. Si ce n'est pas le cas, on passe à **L2** sans exécuter le return.
- ✓ **Calcul de t1** : Si la condition **x == 10** est vraie, on calcule **t1 = x + 1**.
- ✓ **Retour** : Si la condition est vérifiée, on **retourne t1**.

Génération et optimisation de code

Optimisation de Code

Pour optimiser ce code, il n'y a pas beaucoup de modifications à faire étant donné la simplicité de la logique, mais on peut :

Supprimer le label L2 si nous savons qu'aucune action n'est nécessaire quand x n'est pas égal à 10.

Cela donne le code optimisé suivant :

```
if x != 10 goto END        // Si x n'est pas égal à 10, fin de l'exécution
t1 = x + 1                 // Sinon, calculer x + 1
return t1                  // Retourner le résultat
END:
```

Exercice : Génération et optimisation de code

```
while (b != 0) {
    if (a > b)
        a = a - b;
    else
        b = b - a;
}
return a;
```

Génération de Code Intermédiaire en Trois Adresses

Pour traduire ce code en instructions intermédiaires en trois adresses, nous devons :

- ✓ Créer des **instructions conditionnelles** pour simuler le **while**.
- ✓ Créer des **labels** pour gérer les sauts de contrôle dans la boucle.
- ✓ Utiliser des **variables temporaires** pour stocker les résultats des opérations.

Exercice : Génération et optimisation de code

```
while (b != 0) {
    if (a > b)
        a = a - b;
    else
        b = b - a;
}
return a;
```

```
L1:   if b == 0 goto L4        // Condition d'arrêt de la boucle : si b est 0, on sort
      if a <= b goto L3        // Vérifie si a est plus grand que b, passe à L2
      t1 = a - b               //  on calcule a - b
      a = t1                   // On met à jour a avec a - b
      goto L1                  // Retourne au début de la boucle pour vérifier la condition
L3:   t2 = b - a               //  on calcule b - a
      b = t2                   // On met à jour a avec b – a
      goto L4                  // Retourne au début de la boucle pour vérifier la condition
L4:   return a                 // Renvoie le résultat a
```

Exercice : Génération et optimisation de code pour une boucle **for**

```
for ( i = 0; i < N; i++) {
    A[i] = 2 * i + 5;
}
```

- ***Génération de Code Intermédiaire en Trois Adresses***

```
i = 0                   // initialisation de i à 0
L1:  if i >= N goto L2  // vérifie si la condition de boucle est satisfaite, si i >= N,
sort de la boucle
t1 = i * 2              // calcule 2 * i et stocke le résultat dans t1
t2 = t1 + 5             // ajoute 5 à t1 et stocke le résultat dans t2
A[i] = t2               // affecte le résultat t2 dans A[i]
i = i + 1               // incrémente i de 1
goto L1                 // retourne au début de la boucle
L2:                     // fin de la boucle
```

Exercice : Génération et optimisation de code pour une boucle **for**

- ***Optimisation du code intermédiaire***

Plusieurs optimisations sont possibles :

- Élimination des calculs invariants : Puisque **2 * i** est recalculé à chaque itération, une optimisation pourrait consister à l'exprimer par i et à manipuler **t1** directement.

- Propagation de constante : si 5 est constant, on pourrait simplifier l'expression à chaque itération.

- Déroulement de boucle (si N est petit) : Si N est une constante de faible valeur, on pourrait dérouler la boucle pour réduire le nombre de branchements.

```
i = 0
L1:  if i >= N goto L2
t1 = i * 2
A[i] = t1 + 5
i = i + 1
goto L1
L2:
```

THÉORIE DES LANGAGES

Un langage ?

- Pour nous, un langage est simplement un ensemble de suites de lettres…
- Un langage formel = un ensemble de mots.
- on dit aussi: **chaînes de caractères**

Un langage ?

Exemples

- {0, 01, 10, 00, 11, 000, 001, 010, 100, 011, 101, 110, 111} est un langage
- {la, lala, lalala, lalalala, lalalalala, …, lalalala…la, ….} est un langage
- {T, ҒT, Ғ Ғ T, Ғ Ғ Ғ T, …} est un langage
- L'ensemble des mots définis dans un dictionnaire.
- L'ensemble des phrases que l'on peut écrire en français.

Alphabet

- **Alphabet** (vocabulaire, lexique) : est un ensemble **fini** de symboles.
  - **alphabet =** lettres, espace et symboles de ponctuation, . . .

- **A={0, 1} ➔ l'alphabet binaire**
- **B={a, b, c…..z, é, ç, …..} ➔ l'alphabet de la langue français**
- **C={int, if, else, printf} ➔ un alphabet du langage C**
- **C={BONJOUR,ab, ça, 20, $} ➔ un alphabet de 5 symboles**

sont des alphabets couramment utilisés

Mots

Un **mot** est une **suite finie** de symboles :

–> **0010111100** est un mot sur {**0, 1**}

–>**aaabbb** est un mot sur {**a, b**}

–> **acababbc** est un mot sur {**a, b, c**}

–> **BONJOUR** est un mot sur {**A, …, Z**}

–> **μξπρσστυ** est un mot sur {**α**, …, **ξ**}

✓ Longueur d'un mot

- La longueur d'un mot **w**, notée **|w|**, est le nombre de symboles qui le composent.
- **BONJOUR** est un mot de longueur **7** → **|BONJOUR| = 7**

Mot

Notations:

- Le **mot vide**, noté par **ε**, est le mot qui ne contient aucun élément d'un alphabet **X**.
- L'ensemble des mots construits sur un alphabet **X** est un ensemble infini noté **X***.
- L'ensemble des mots construits sur un alphabet X qui ne contient pas le mot vide est un ensemble infini noté $\mathbf{X^+}$. On écrit :

$$\mathbf{X^* = X^+ \cup \{\varepsilon\}}$$

Stratification

Parmi tous les mots sur A, il y a:

- Les mots de **longueur nulle**, formant l'ensemble {ε}
- Les mots de longueur 1, donnant simplement l'ensemble A
- Les mots de longueur 2, formant A^2
- Les mots de longueur n, formant A^n

$A^* = \{\varepsilon\} \cup A \cup A^2 \cup A^3 \ldots \cup A^n \ldots$

Ou:

$$A^* = \bigcup_{n=0}^{\infty} A^n$$

avec $A^0 = \varepsilon$

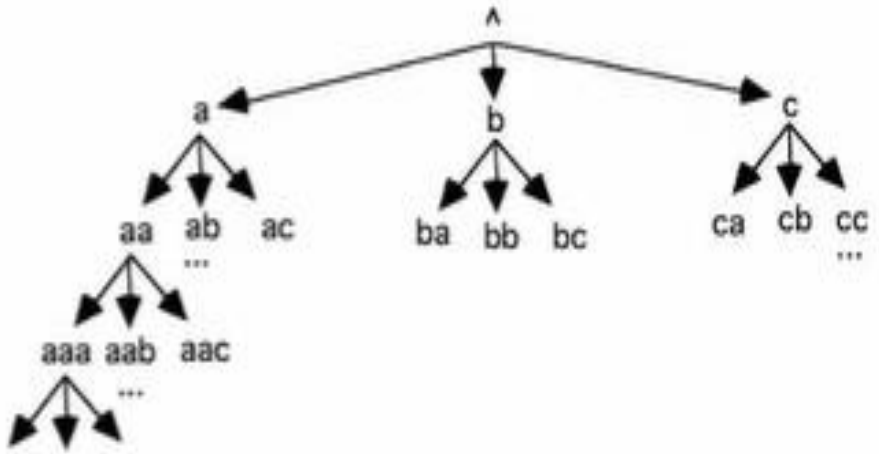
Mot

Exemple

Soit l'alphabet X= {a, b, c}, donc X* définit sur X est :

X*={ε, a, b,c, aa,bb,cc,ac, ab,ba,bc,ca,cb, aba, abb,.....}

X*={ε}+{a, b}+{aa, ab, bb,ba,cc,..}+{aaa, bbb, cccc, aab,...}+...+-



Si on note: X^i l'ensemble des mots de longueur i construits sur l'alphabet X, donc on a:

$$X^* = X^0 + \underbrace{X^1 + X^2 + \ldots\ldots\ldots\ldots\ldots + X^{+\infty}}_{X^+} = \sum_{i=0}^{\infty} X^i$$

Concaténation

- **Soit deux mots S et S′ sur A**
- **S′ils sont définis par : S = $S_1 \ldots S_n$ et S′ = $S'_1 \ldots S'_m$,**
- **On peut définir un nouveau mot, qu'on notera S.S′ (ou SS′) par:**
- **S.S′ = $S_1 \ldots S_n S'_1 \ldots S'_m$**
- **Il s'agit de la concaténation de S et de S′**

Sur X={a, b, c}

- **S=abbac** est un mot, qui est désigné par **S**
- **S′=bcca** est un autre mot, qui est désigné par **S′**

- On vérifiera que leur **concaténation S.S′** correspond au mot **abbacbcca**

S.S′ = abbacbcca

Propriétés

La concaténation est associative…mais non commutative…

- **Un élément neutre : ε**

  ε.x = x. ε = x

- **Conséquence : la concaténation d'une suite vide de mots est le mot vide**

Puissance d'un mot

Soit un alphabet X , w ∈ X* et n ∈ IN , la puissance de W est donnée comme suit :

$$w^n = \begin{cases} \varepsilon & \text{si } n = 0 \\ w & \text{si } n = 1 \\ ww^{n-1} = w^{n-1}w & \text{si } n > 1 \end{cases}$$

Exemple

Soit X= {a, b} et w = aba

- $w^0=\varepsilon$
- $w^1=w.\ \varepsilon = w=aba$
- $w^2=ww^1=ww=abaaba$
- $w^3=ww^2=www=$ abaabaaba

Factorisation d'un mot

Soit un alphabet X et w, u ∈ X* :

- ✓ u est un facteur gauche (**préfixe**) de w ⇔ ∃ v ∈ **X*** tel que w = uv
- ✓ u est un facteur droit (**suffixe**) de w ⇔ ∃ v ∈ **X*** tel que w = vu
- ✓ u est un **préfixe propre** de w ⇔ ∃ v ∈ $\mathbf{X^+}$ tel que w = uv
- ✓ u est un **suffixe propre** de w ⇔ ∃ v ∈ $\mathbf{X^+}$ tel que w = vu

Exemple:

Soit l'alphabet X= {a, b}, et le mot w = babb :

- Les préfixes de w sont, b, ba, bab, babb
- Les suffixes de w sont, b, bb, abb, babb
- Les préfixes propres de w sont: b, ba , bab
- Les suffixes propres de w sont: b , bb , abb

Inverse d’un mot ou miroir

Le miroir d'un mot $\mathbf{w = a_1a_2...a_n}$ est le mot noté $\mathbf{w^R = a_n...a_2a_1}$ obtenu en inversant les symboles de **w**.

Exemple:

Soit **w = abbc** un mot de l'alphabet **X = {a, b}**, le miroir de w est $\mathbf{w^R = cbba}$.

Remarque:

- ✓ Le miroir de n'importe quel mot composé d'un seul symbole est le mot lui-même.
- ✓ Le miroir du mot vide est lui-même : $\mathbf{\varepsilon^R = \varepsilon}$.
- ✓ Un mot est un palindrome si $\mathbf{w^R = w}$, ex : $\mathbf{(aba)^R = aba}$.

Exercice

1. Soit l'alphabet suivant : $\Sigma = \{a, b, c\}$.
   - Listez tous les mots possibles de longueur 2.
   - Combien de mots de longueur 3 peut-on former avec cet alphabet ? Expliquez.
2. Donnez trois exemples d'alphabets différents utilisés dans les applications suivantes :
   - Les mots d'une langue naturelle.
   - Le code binaire.

3. Soit l'alphabet $\Sigma = \{x, y, z\}$ et le mot $w = xyzxyz$:
   - Quelle est la longueur du mot w ?
   - Identifiez les sous-mots possibles de w.

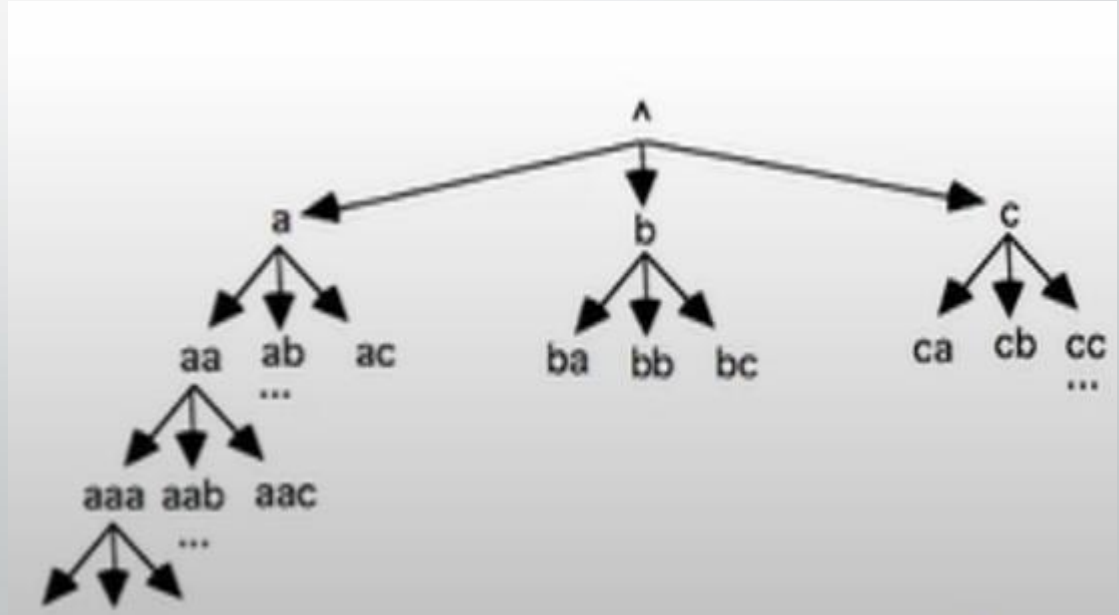
Relation d’ordre

On définit dans X* plusieurs relations d'ordre :

- **Ordre préfixiel** = ordre partiel défini par (**U < V**) si et seulement si (**U**) est un facteur gauche de (**V**).
- **Ordre lexicographique** (**ordre du dictionnaire**) = ordre total qui étend l'ordre préfixiel.
- **Ordre hiérarchique** = est un ordre total. Les mots sont classés d'abord par **longueur**, puis pour les mots de **même longueur** par ordre lexicographique (suppose que l'on a ordonné les lettres de l'alphabet).

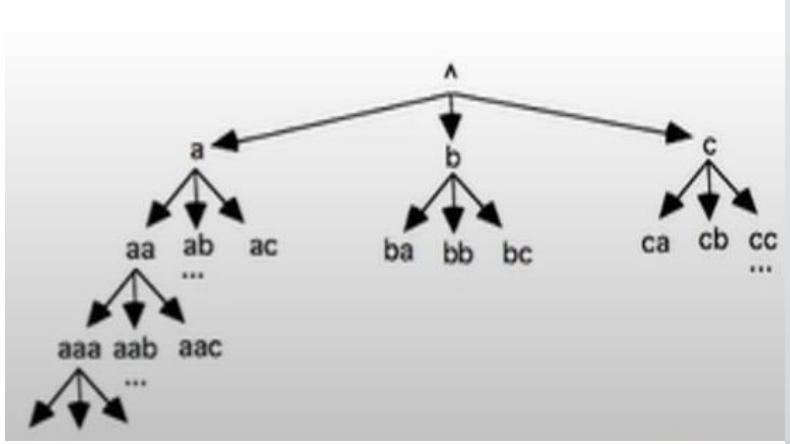
-> Sur la représentation graphique

Un mot précède un autre dans un ordre hiérarchique s'il se trouve à un niveau plus élevé ou bien, étant sur le même niveau, s'il se trouve plus à gauche que celui-ci



Exercice

Considérez l'alphabet $\Sigma = \{a, b\}$ et les mots sur cet alphabet.



1. **Ordre Préfixiel**
   a. Déterminez si $"ab" < "abc"$ selon l'ordre préfixiel.
   b. Trouvez tous les mots de Σ^* qui sont des préfixes de $"abab"$.
2. **Ordre Lexicographique**
   a. Classez les mots suivants selon l'ordre lexicographique : $\{"ba", "aab", "ab", "b"\}$.
   b. Quel mot vient immédiatement après $"ab"$ dans cet ordre ?
3. **Ordre Hiérarchique**
   a. Classez les mots suivants selon l'ordre hiérarchique : $\{"a", "ba", "b", "aa", "ab"\}$.
   b. Dans une représentation graphique (comme dans la figure de l'énoncé), tracez les premiers niveaux de l'arbre pour $\Sigma = \{a, b\}$.

Langage

📌 **Définition**

Un langage sur un alphabet X est une partie de X^*. Donc un langage est un ensemble de mots. On note $L \subseteq X^*$ ou $L \in P(X^*)$.

📌 **Exemple**

- Soit l'alphabet $X = \{a, b\}$
- $\emptyset$ est le langage vide, il ne contient aucun mot.
- $\{\varepsilon\}$ est un langage.
- $\{a, b, aa, bb, aba\}$ est un langage
- $\{w \in X^* \mid w = a^n \text{ tel que } n > 0\}$ est un langage

📌 **Remarque**

- Un langage sur un alphabet X peut être fini ou infini.
- $\emptyset$ est un langage défini sur n'importe quel alphabet.

Langages Formels

Un langage formel sur un alphabet X sera défini simplement comme une partie de X*

Donc parmi les langages possibles sur X:

- Ø ---→(langage vide)
- {ε} ---→(langage réduit au mot vide)
- X* ---→ (langage plein)

Les opérations sur les langages

Les langages étant des ensembles, toutes les opérations ensemblistes « classiques » leur sont donc applicables.

Soient $L_1, L_2 \subseteq X^*$:

$L_1 \cup L_2 = \{w \in X^* \mid w \in L_1 \text{ ou } w \in L_2\}$

📌 **Union**

L'union de L_1 et L_2 est l'ensemble des mots de L_1 et L_2 :

Elle est :

- Associative.
- Commutative.
- Élément neutre, le langage vide $\emptyset$: $L \cup \emptyset = \emptyset \cup L = L$
- Élément absorbant, le vocabulaire X^* : $L \cup X^* = X^* \cup L = X^*$
- Notée $+$ dans la théorie des langages. On écrit aussi :
  $L_1 + L_2 = \{w \in X^* \mid w \in L_1 \text{ ou } w \in L_2\}$

Les opérations sur les langages

❑ Intersection

L'intersection de L1 et L2 est l'ensemble des mots qui appartiennent à la fois à L1 et L2 :

$$L_1 \cap L_2 = L_2 \cap L_1 = \{w \in X^* / w \in L1 \text{ et } w \in L2\}$$

Elle est :

- Associative
- Commutative
- Son élément neutre est X^*
- Son élément absorbant est $\emptyset$ (puisque " $\forall L : \emptyset \cap L = L \cap \emptyset = \emptyset$)

Les opérations sur les langages

❑ **Théorème**

Le produit de langages est distributif par rapport à l'union

$$\forall L_1, L_2, L_3 \subseteq X^*$$

$$L_1.(L_2 \cup L_3) = (L_1.L_2) \cup (L_1.L_3)$$

$$(L_2 \cup L_3).L_1 = (L_2.L_1) \cup (L_3.L_1)$$

De manière générale, $\forall A, L_i \subseteq X^*$

$$A.(\bigcup_{i=1}^{\infty} L_i) = \bigcup_{i=1}^{\infty} (A.L_i)$$

$$(\bigcup_{i=1}^{\infty} L_i).A = \bigcup_{i=1}^{\infty} (L_i.A)$$

Attention ! : Le produit de langages n'est pas distributif par rapport à l'intersection.

Les opérations sur les langages

- **Puissance**

  La puissance ou l'itération d'un langage est définit comme suit :

$$L^n = \begin{cases} \{\varepsilon\} & \text{si } n = 0 \\ L & \text{si } n = 1 \\ LL^{n-1} = L^{n-1}L & \text{si } n > 1 \end{cases}$$

Attention ! Ne pas confondre L^n avec le langage contenant les puissances nièmes des mots de L et qui serait défini par $\{u \in X^*, \exists v \in L, u = v^n\}$.

Les opérations sur les langages

- **La fermeture positive**

  La fermeture positive de L noté L^+ et on écrit : $L^+ = \bigcup_{i=1}^{\infty} L$

- **La fermeture de Kleene**

  La fermeture étoile de L nommée aussi la fermeture de Kleene et notée L* est définit par :

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

$$L^* = L^0 + L^1 + + L^*$$

$$= \{\varepsilon\} + \bigcup_{i=1}^{\infty} L^i = \{\varepsilon\} + L^+$$

- $L^* = L^+ \cup \{\varepsilon\}$ et $L^+ = L.L^*$

Les opérations sur les langages

- **Exemple**

Soit X= {0,1}, L_1= {00, 11, 01} et L_2= {01, 10}

- $L_1 + L_2$= {00, 11, 01, 10}
- $L_1 \cap L_2$= {01}
- $L_1.L_2$ = {0001, 0010, 1101, 1110, 0101, 0110} ≠ $L_2.L_1$= {0100, 0111, 0101, 1000, 1011, 1001}
- L_2^*= {ε} +{L_2} + {L_2. L_2} + {L_2. L_2.L_2} +..... +{L_2. L_2.L_2..... L_2. L_2.L_2}

Exercice

Soit l'alphabet $A = \{a, b\}$

Etant donnés les mots $u = aa$ et $v = bab$

1. Ecrire les mots uv, $(uv)^2$ et u^3v.
2. Enoncer tous les mots de longueur 2 définis sur A.
3. Définir les ensembles suivants:

$E_1 = \{u.v / u \in A^+, v \in A^+\}$, $E_2 = \{u.v / u \in A^+, v \in A^*\}$, $E_3 = \{u.v / u \in A^*, v \in A^*\}$

Solution

Soit l'alphabet A = {a, b} et les mots u = aa et v = bab

1. uv = aabab, $(uv)^2$ =uvuv=aababaabab et u^3v = uuuv=aaaaaabab.
2. Mots de longueur 2 = {aa, ab, ba, bb}
3. $E_1 = \{u.v / u \in A^+, v \in A^+\} = \{u \in A^* / |u| \geq 2\}$ = ensemble des mots d'au moins 2 symboles

   $E_2 = \{u.v / u \in A^+, v \in A^*\} = A^+ = \{u \in A^* / |u| \geq 1\}$ = ensemble des mots d'au moins 1 symbole

   $E_3 = \{u.v / u \in A^*, v \in A^*\} = A^*$

Exercice

Soit $\Sigma = \{a, b\}$ et les langages $L_1 = \{a, ab\}$ et $L_2 = \{b, bb\}$:

- Trouvez $L_1 \cup L_2$.
- Trouvez $L_1 \cdot L_2$ (concatenation des langages).
- Trouvez L_1^* (fermeture de Kleene de L_1).

Soit $L = \{a, b\}$. Trouvez les premiers 5 éléments de L^* (fermeture de Kleene de L).

Soit $L_1 = \{a, ab\}$ et $L_2 = \{b, bb\}$.

Calculez $L_1 \cdot L_2$ (produit des langages).

Soit $L = \{ab, ba\}$.

1. Trouvez les 4 premiers mots de L^+ (fermeture positive).
2. Calculez $L^+ \cap \{a, b, abba, abab\}$.

Exercice

Soit $L_1 = \{a, ab\}$ et $L_2 = \{b, bb\}$.

Calculez :

1. $(L_1 \cup L_2)^*$.
2. $(L_1 \cap L_2)^*$.

Soit $L = \{w \in \{a, b\}^* \mid w \text{ commence et finit par 'a'}\}$.

1. Parmi les mots suivants, lesquels appartiennent à L ?
   (a) aba, (b) baa, (c) $abab$, (d) aa.

Soit $L = \{w \in \{a, b\}^* \mid |w| \text{ est impair}\}$.

1. Trouvez les 5 premiers mots de L.
2. Déterminez si $w = abaab \in L$.

Soit $\Sigma = \{a, b\}$ et $L = \{w \in \Sigma^* \mid w \text{ contient au moins une fois 'ab'}\}$.

1. Trouvez $\overline{L}$, le complément de L.
2. Donnez 3 exemples de mots dans $\overline{L}$.

Grammaire

- **Introduction**

Au sens littéraire du terme, les grammaires désignent pour les langues naturelles, un ensemble de règles syntaxiques conventionnelles qui déterminent un emploi correct de la langue parlée et écrite. Le même concept s'applique aussi pour la théorie des langages, en suivant les règles de production d'une grammaire on peut engendrer un langage.

Grammaire

Définition

Une grammaire est un ensemble de règles de production qui sont utilisées pour engendrer un langage.

Exemple

Dans la grammaire de la langue française on peut formuler une phrase en respectant la séquence suivante : **'Sujet' 'Verbe' 'COD'.** On peut formuler un nombre bien défini des phrases en respectant les règles de production suivantes :

PHRASE----------→SUJET VERBE CO
SUJET----------→je I il
VERBE----------→lis I conduit
COD----------→DETERMINANT NOM
DETERMINANT----------→un I la
NOM----------→ livre I voiture

Grammaire

Exemple: A partir de ces règles syntaxiques on construit $2^4=16$ phrases syntaxiquement correct mais pas forcément sémantiquement correctes.

PHRASE----------→SUJET VERBE COD
SUJET----------→je I il
VERBE----------→lis I conduit
COD----------→DETERMINANT NOM
DETERMINANT----------→un I la
NOM----------→ livre I voiture

Parmi lesquelles, on cite les deux phrases suivantes :

SUJET-----→ Je
VERBE----→lis
DETERMINANT---→ un
NOM-----→ livre

et

SUJET----→ il
VERBE----→ conduit
DETERMINANT----→ la
NOM----→ voiture

→

PHRASE --→ Je lis un livre
PHRASE ---→ Il conduit la voiture

Grammaire

Définition formelle:

Une grammaire G est un quadruplet (**N**, **T**, **P**, **S**) tels que :

N : ensemble fini de symboles non terminaux.
T : ensemble fini de symboles terminaux.
N ∩ T=∅
S : symbole non terminal de départ (axiome).
P : ensemble fini de règles de production de la forme

$$\alpha \longrightarrow \beta \text{ tel que } \alpha \in (N \cup T)^{+} - T^{+} \text{ et } \beta \in (N \cup T)^{*}$$

Exemple:

Dans l'exemple précédant, la grammaire est définie comme suit :

- **N**= {PHRASE, SUJET, VERBE, COD, DETERMINANT}
- **T**= {je, il, lire, conduire, un, la, livre, voiture}
- **S**= {PHRASE}
- **P**= {PHRASE →SUJET VERBE COD, COD→DETERMINANT NOM, SUJET→je I il, VERBE→lis I conduit, DETERMINANT→un I la, NOM → livre I voiture }

Grammaire

Notations:

Dans les règles formelles d'une grammaire :

- Les symboles de l'alphabet sont écrits en minuscule et appelés les symboles des t**erminaux**.
- Les autres symboles (sauf le mot vide) sont écrits en majuscule et appelés les symboles **non terminaux**.
- La génération des mots commence toujours à partir d'un **symbole non terminal** appelé **l'axiome** (dans notre exemple l'axiome = PHRASE).

Dérivation

❑ **Dérivation directe**

Un mot w' dérive directement d'un mot w qu'on note w => w' si une règle de G est appliquée **une fois** pour passer de w à w'.

C'est-à-dire : il existe une règle $\alpha \rightarrow \beta$ dans P telle que : $w=u\alpha v$ et $w'=u\beta v$ avec avec $u, v \in (N \cup T)^*$.

❑ **Exemple**

Soit la grammaire: $G = (\{S\},\{a, b\}, \{S \rightarrow aSb, S \rightarrow \varepsilon\}, S)$, nous avons :

1. aSb dérive directement de S et on écrit : S => aSb car il existe une règle $S \rightarrow aSb \in P$ telle que :

   $w = S$ et $w' = aSb$ avec $u=v=\varepsilon \in (N \cup T)^*$

2. ab dérive directement de aSb et on écrit

   aSb => ab car il existe une règle $S \rightarrow \varepsilon$ telle que :

   w = aSb et w' = ab avec u=a et v=b

Exemple

Soit la grammaire: G = ({S},{a, b}, {S -> aSb, S->ε }, S) , nous avons :

X={a,b} l'aphabet

N={S}, T={a,b}, S={S}, **P= {S -> aSb,S->ε }**

S-> aSb-> ab

S-> aSb->aaSbb-> aabb

S-> aSb->aaSbb->aaaSbbb-> aaabbb L(G)={ε,ab, aabb, aaabbb, aaaabbbb,.....,a^n b^n }

Exemple

Soit la grammaire: G = ({S},{a, b}, {S -> aSb, S->ε }, S) , nous avons :

X={a,b} l'aphabet

N={S}, T={a,b}, S={S}, **P= {S -> aSb, S->ε },**

S->ε

S-> aSb -> aεb->ab

S-> aSb->aaSbb-> aabb

S-> aSb->aaSbb->aaaSbbb-> aaabbb L(G)={ε,ab, aabb, aaabbb, aaaabbbb,.....,a^n b^n }

Langage engendré par une grammaire

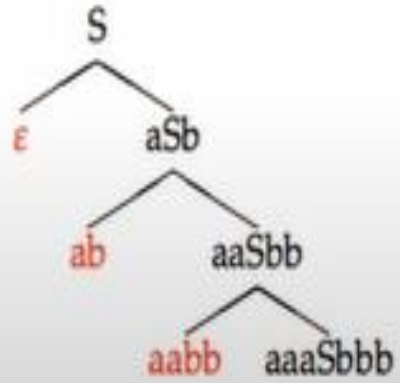
❑ Définition

Un langage engendré par une grammaire G= (N, T, P, S) est l'ensemble des mots obtenus en appliquant des séquences de dérivations à partir de l'axiome S. on note :

$$L(G)=\{ w \in T^* / S \Rightarrow^*_G w \}$$

❑ Exemples

1. Pour la grammaire G = ({S},{a, b}, {S -> aSb, S->ε }, S) , nous avons :



- Mot minimal: ε

- La forme générale : $L(G)=\{ a^n b^n / n \geq 0 \}$

$$S=>aSb=>aaSbb => aaaSbbb.....=> a^n \varepsilon b^n => a^n b^n$$

❑ Remarque

Une grammaire définit un seul langage par contre un même langage peut être engendré par plusieurs grammaires différentes.

Grammaire équivalentes

- **Exemple**

  Deux grammaires sont équivalentes si elles engendrent le même langage.

  $$G \text{ équivalente à } G' \Leftrightarrow L(G) = L(G')$$

- **Exemple**

G = ({S, A, B},{a, b},{S ->aS/ ABb, A ->Aa/a, B ->b}, S)

G′ = ({S, A, B},{a, b},{S ->aS/aA, A ->aA/bB, B->b}, S)
On trouve que L(G)=L(G′) = $\{a^k b^2 / k \geq 1\}$

Classification des grammaire

Hiérarchie de Chomsky

Selon la classification de Chomsky, les grammaires sont regroupées en quatre types en fonction de la forme de leurs règles de production.

Soit une grammaire G = (N, T, P, S)

- **Grammaire Syntagmatique – Type 0**
- **Grammaire Monotone- Type 1**
- **Grammaire Algébrique (hors contexte)- Type 2**
- **Grammaire Régulière- Type 3**

Classification des grammaire

❑ Grammaire Syntagmatique – Type 0

G est dite grammaire de type 0 dite aussi grammaire sans restriction (grammaire générale) : si toute ses règles sont de la forme générale suivante :

$$\alpha \rightarrow \beta \text{ avec } \alpha \in (N \cup T)^{+} - T^{+} \text{ et } \beta \in (N \cup T)^{*}$$

Exemple :

$$G = (\{S\}, \{a,b,c\}, \{S \rightarrow aS/Sb/c, aSb \rightarrow Sa/bS\}, S)$$

Classification des grammaire

❑ Grammaire monotone- Type 1

G est dite grammaire de type 1 dite aussi grammaire monotone : si toutes ses règles sont de la forme :

$$\alpha \to \beta \text{ avec } |\alpha| \leq |\beta| \text{ tels que } \alpha \in (N \cup T)^{+} - T^{+} \text{ et } \beta \in (N \cup T)^{*}$$

Exception : axiome $\longrightarrow \varepsilon$ peut appartenir à P

Exemple :

$$G = \left(\{S, R, T\}, \{a, b, c\}, \{S \to \varepsilon / aRbc / abc, R \to aRTb / aTb, Tb \to bT, Tc \to cc\}, S\right)$$

Classification des grammaire

❑ Grammaire Algébrique (hors contexte)- Type 2

G est dite grammaire de type 2 dite aussi grammaire algébrique (hors contexte) : si toutes ses règles sont de la forme.

$$A \longrightarrow \beta \text{ avec } A \in N \text{ et } \beta \in (N \cup T)^*$$

Exemple :

1. $G = (\{S\}, \{a, b\}, \{S \rightarrow aSb / \varepsilon\}, S)$
   Mots : ε, ab, aabb, aaabbb,....., $a^n b^n$
2. $G = (\{S\}, \{0, 1\}, \{S \rightarrow 0S0 / 1S1 / \varepsilon / 0 / 1\}, S)$
   Mots : 00, 11, 010 , 101, 000,111,101101....
   C'est la grammaire des Palindromes

Arbre Syntaxique

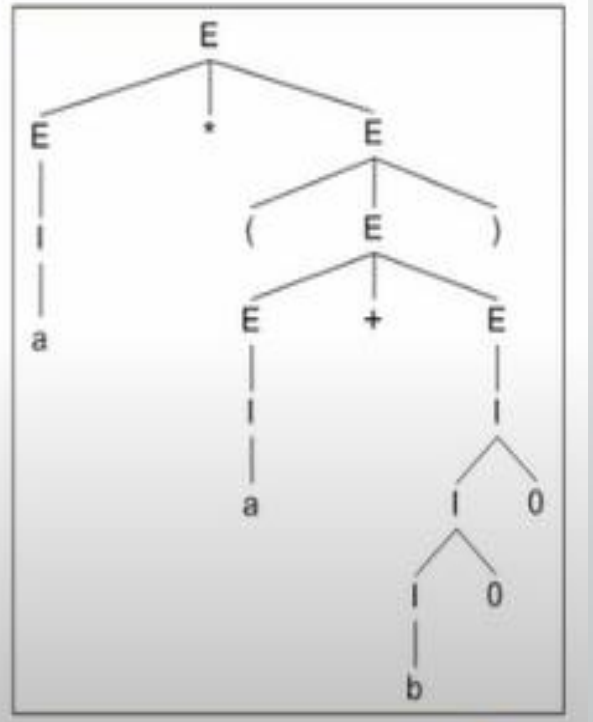
❑ **Exemple**

Soit la grammaire suivante : T={a, b, 0, 1, +, *, (,)}, N={E, I} avec E est l'axiome, et les règles de production suivantes :

$$E \rightarrow I \mid E*E \mid E+E \mid (E)$$

$$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

La figure ci-coté représente l'arbre de dérivation de l'expression : a * (a + b00).
La dérivation la plus à gauche et la plus à droite fournissent le même arbre.



$$E \rightarrow E * E$$
$$\rightarrow I * E$$
$$\rightarrow a * (E)$$
$$\rightarrow a * (E+E)$$
$$\rightarrow a * (I+E)$$
$$\rightarrow a * (a+E) \rightarrow a * (a+I) \rightarrow a * (a+I0) \rightarrow a * (a+I00) \rightarrow a * (a+b00)$$

Arbre Syntaxique

❑ **Exemple**

Soit la grammaire suivante : T={a, b, 0, 1, +, *, (,)}, N={E, I} avec E est l'axiome, et les règles de production suivantes :

$$E \rightarrow I / E*E / E+E / (E)$$

$$I \rightarrow a / b / Ia / Ib / I0 / I1$$

- **Dérivation la plus à gauche de : a*(b+a0)**

$E \Rightarrow E*E \Rightarrow I*E \Rightarrow a*E \Rightarrow a*(E) \Rightarrow a*(E+E) \Rightarrow a*(I+E) \Rightarrow a*(b+E) \Rightarrow a*(b+I) \Rightarrow a*(b+I0) \Rightarrow a*(b+a0)$

- **Dérivation la plus à droite de : a*(b+a0)**

$E \Rightarrow E*E \Rightarrow E*(E) \Rightarrow E*(E+E) \Rightarrow E*(E+I) \Rightarrow E*(E+I0) \Rightarrow E*(E+a0) \Rightarrow E*(I+a0) \Rightarrow E*(b+a0) \Rightarrow I*(b+a0) \Rightarrow a*(b+a0)$

Classification des grammaire

❑ **Grammaire Algébrique (hors contexte)- Type 3**

G est dite grammaire de type 3 dite aussi grammaire régulière :si elle est régulière à gauche ou bien à droite.

- Une grammaire G est dite régulière à gauche si toutes ses règles sont de la forme :

  $A \longrightarrow Bw$ ou $A \longrightarrow w$ avec $A, B \in N$ et $w \in T^*$

- Une grammaire G est dite régulière à droite si toutes ses règles sont :

  de la forme $A \longrightarrow wB$ ou $A \longrightarrow w$ avec $A, B \in N$ et $w \in T^*$

Classification des grammaire

❑ Grammaire Régulière - Type 3

- Une grammaire G est dite régulière à gauche si toutes ses règles sont de la forme :

  $A \longrightarrow Bw$ ou $A \longrightarrow w$ avec $A, B \in N$ et $w \in T^*$

- Une grammaire G est dite régulière à droite si toutes ses règles sont :

  de la forme $A \longrightarrow wB$ ou $A \longrightarrow w$ avec $A, B \in N$ et $w \in T^*$

Exemple :

- $G_1 = (\{S,A\},\{a,b\},\{S \to Sb/Ab, A \to Aa/a\},S)$ grammaire régulière à gauche.
- $G_2 = (\{S,A\},\{a,b\},\{S \to aS/aA,\ A \to bA/b\},S)$ grammaire régulière à droite.
- $G_3 = (\{S,A\},\{a,b\},\{S \to aS/aA, A \to Ab/b\},S)$ grammaire n'est pas régulière ; puisque n'est ni régulière à gauche ni régulière à droite.

Classification des grammaire

❑ Remarques

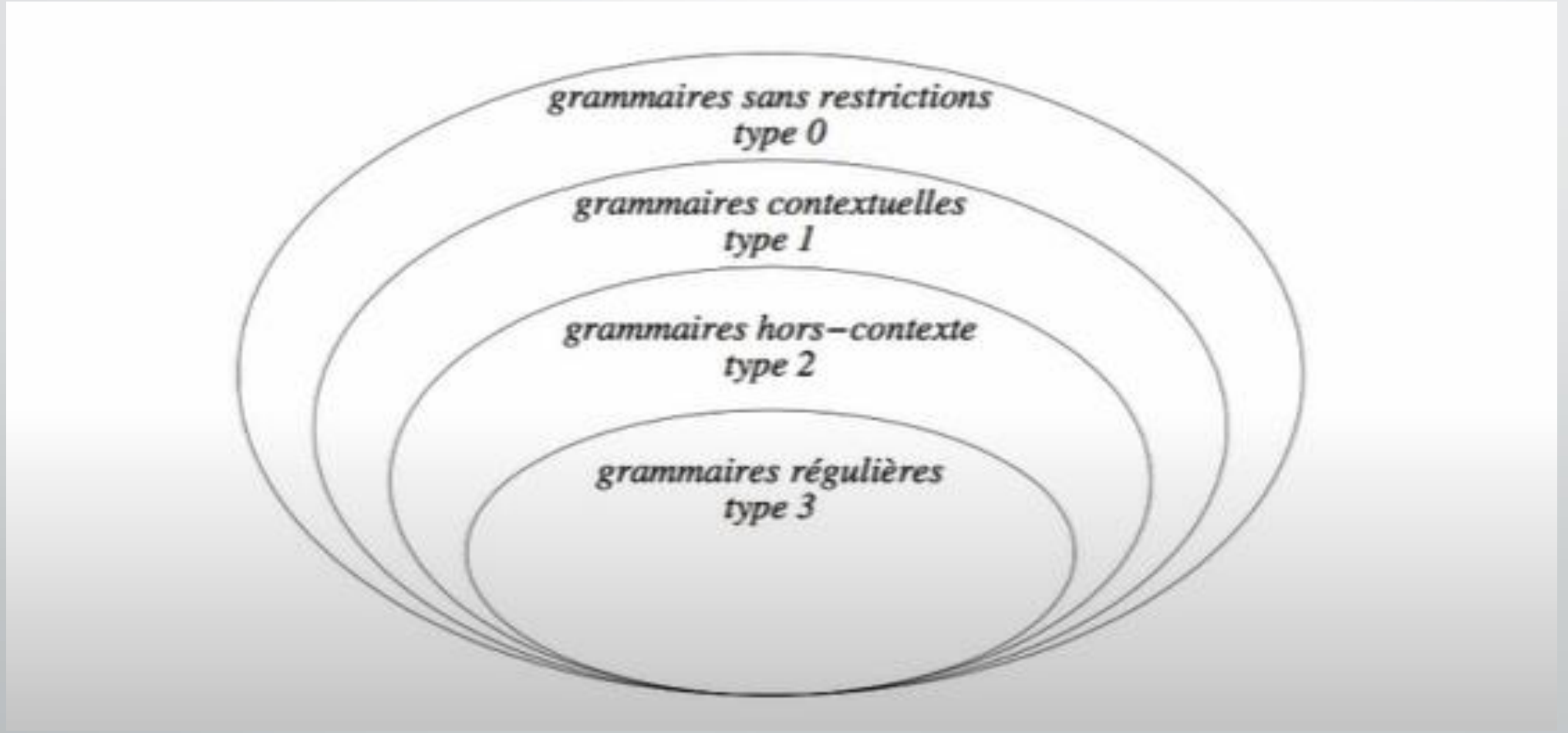
1. On peut trouver que G1 et G2 engendrent le même langage. G1 génère les mots de ce langage de la droite vers la gauche mais G2 génère les mots de ce langage de la gauche vers la droite.

2. Il y a une relation d'inclusion stricte entre les 4 types des grammaires c'est à dire :
Une grammaire de type i est aussi de type inférieur à i ($1 \leq i \leq 3$)

$$\text{type } 3 \subset \text{type } 2 \subset \text{type } 1 \subset \text{type } 0$$

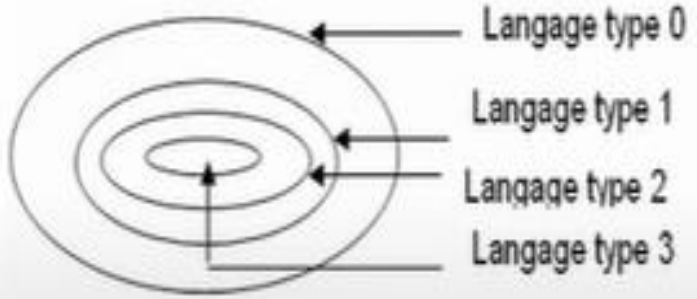
3. Le type retenu pour une grammaire est le type maximum de la grammaire qui vérifie ses règles.

Classification des grammaire



Classification des langages

La classification des grammaires va permettre de classer les langages selon le type maximum de la grammaire qui l'engendre (puisqu'un langage peut être engendré par plusieurs grammaires de types différents). Il y a une relation d'inclusion stricte entre les 4 types des langages.



On dit qu'un langage est de type i s'il est engendré par une grammaire de type i et pas par une grammaire d'un type supérieur.

Classification des langages

❑ **Exemples**

- **Langage de Type 0**

  L={ac, acb, ca…}

  $G = (\{S\},\{a,b,c\},\{S \rightarrow aS/Sb/c, aSb \rightarrow Sa/bS\},S)$

- **Langages Contextuel - Type 1 :**

  $L=\{a^n b^n c^n \mid n \geq 0\}$

  $G = \langle\{S,B,W,X\},\{a, b, c\},\{ S \rightarrow abc, S \rightarrow aSBc, cB \rightarrow WB, WB \rightarrow WX, WX \rightarrow BX, BX \rightarrow BC, bB \rightarrow bb\}, S\rangle$

Classification des langages

❑ **Exemples**

- **Langages hors Contexte- Type 2:**

  $L = \{a^n b^n \mid n \geq 0\}$

  $G = \langle \{S\}, \{a, b\}, \{S \rightarrow aSb|\varepsilon\}, S \rangle$

- **Langages Rationnel (Régulier)- Type 3**

  $L = \{m \in \{a, b\}*\}$

  $G = \{\{S\}, \{a, b\}, \{S \rightarrow aS \mid bS \mid \varepsilon\}, S\}$

Exercice 1 et 2

Exercice 1:

On considère la grammaire G = (T, N, S, R) où
T = { b, c }
N = { S }
R = { S → bS | cc }

1. Quel est le type de G.
2. Déterminer L(G).

Exercice 2:

Construire une grammaire pour le langage L = $\{ab^n a / n \in N\}$.

Sol_Exercice_1

1. On considère la grammaire G = (T, N, S, R) où

   $T = \{ b, c \}$

   $N = \{ S \}$

   $R = \{ S \rightarrow bS \mid cc \}$

G est une grammaire régulière à droite. Car on les règles de production: $S \rightarrow bS$ et $S \rightarrow cc$

2. $L(G) = \{b^n cc / n \in N\}$

En effet, partant de l'axiome S, toute dérivation commencera nécessairement par appliquer 0, 1 ou plusieurs fois la première règle puis se terminera en appliquant la deuxième règle. On représentera cela en écrivant le schéma de dérivation suivant :

Sol_Exercice_2

Exercice 2 :

Soit le langage $L = \{ab^n a / n \in N\}$.

On considère la grammaire G = (T, N, S, P) où
T = { a, b }
N = { S, T }
P= { $S \rightarrow aTa$, $T \rightarrow bT \mid \varepsilon$ }

S =>aTa=>aa
S=>aTa=>abTa=>aba
S=>aTa=>abTa=>abbTa=>abba
S=>aTa=>abTa=>abbTa=>abbbTa=>.....=>ab^nTa=>$ab^n a$

Exercice 3

Soient les grammaires Gi = ({a, b, c} , {S, A, B, R, T} , S , Pi), (i=1,..,6) ; où les Pi sont :

1) **P1 : S → aA | bB ; A → a | ab ; B → b | cb**
2) **P2 : S → bA ; A → aA | ε**
3) **P3 : S → aSc | A A → bA | b**
4) **P4 : S → aSbS | ε**
5) **P5 : S → aRbc | abc ; R → aRTb | aTb ; Tb → bT ; Tc → cc**
6) **P6 : S → aAb | ε A → aSb ; Ab → ε**

- ✓ Pour chacune des grammaires Gi (i=1,..,6) ; donner le type de celle-ci, puis trouver le langage engendré par chacune d'elles.
- ✓ Vérifier que G2 n'est pas de type 1 ; mais que L(G2) est de type 1.
- ✓ Montrer que L(G6) est de type 2 en trouvant une grammaire de type 2 qui l'engendre.

Exercice 4

Soit la grammaire G dont les règles de production sont :

$S \rightarrow RD$

$R \rightarrow aRb \mid A$

$Ab \rightarrow bbA$

$AD \rightarrow \varepsilon$

1) Déterminer L(G).
2) Construire une grammaire de type 2 équivalente à G.

Sol_Exercice_3

II) G2 n’est pas de type 1 car elle contient la règle : A → ε ; or dans les grammaires de type 1, le seul symbole qui peut produire la chaîne vide est S. Cependant, on peut écrire une grammaire de type 1 équivalente à G2 : G2’ a pour règles de production : S → Sa | b ; ce qui veut dire que L(G2) est de type 1.

III) Une grammaire de type 2 équivalente à G6 : G6’ a pour règles de production : S → aaSbb | a | ε

Sol_Exercice_4

1) L(G) = { anb2n / n > 0} ;

2) Grammaire de type 2 équivalente à G : G’ = ({a, b}, {S}, S, P’)

où P’ : S → aSbb | ε