Compilation : introduction et rappels

Arnaud Labourel

Section 1

Introduction

Organisation du cours

Volume horaire

- 5 séances de cours (9h)
- 5 séances de TD (9h)
- 6 séances de 2h de TP en salle machine (12h)

Évaluation

- un examen sur papier (50% de la note)
- un TP noté lors de la dernière séance de TP (25% de la note)
- Contrôle Continu : suivi de TP (25% de la note)

Généralités

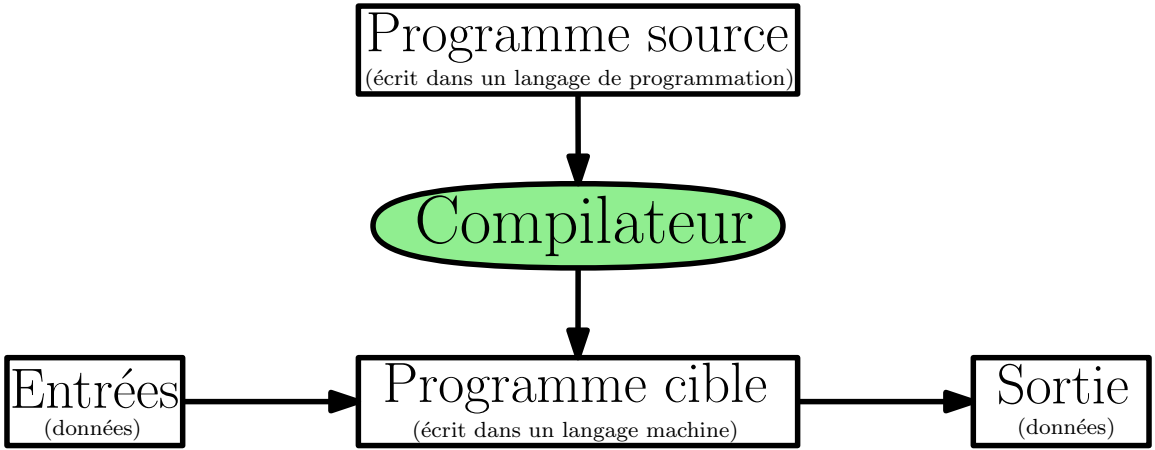
- Qu'est-ce qu'un langage de programmation ?
  - Un ensemble de notations pour décrire des calculs à des gens et à des machines
  - Un pont entre les programmeurs et les ordinateurs
- Problème : un ordinateur est généralement incapable d'exécuter directement un programme écrit dans un langage de programmation par un programmeur (langage de programmation haut niveau)

  $\rightarrow$ Nécessité de traduire les programmes dans une forme compréhensible par l'ordinateur : le rôle d'un compilateur

Compilateur

- Un compilateur est un programme :
  1. qui lit un programme rédigé dans un langage de programmation, appelé langage source
  2. et qui le traduit dans un autre langage, le langage cible (généralement en langage machine)
- Le compilateur a aussi pour rôle de reporter l'ensemble des erreurs rencontrées au cours du processus de traduction (non-respect de la grammaire du langage, problèmes de types pour les langages statiquement typé, ...)
- Lorsque le programme cible est un programme exécutable, en langage machine, l'utilisateur peut l'exécuter pour traiter des données et produire des résultats.

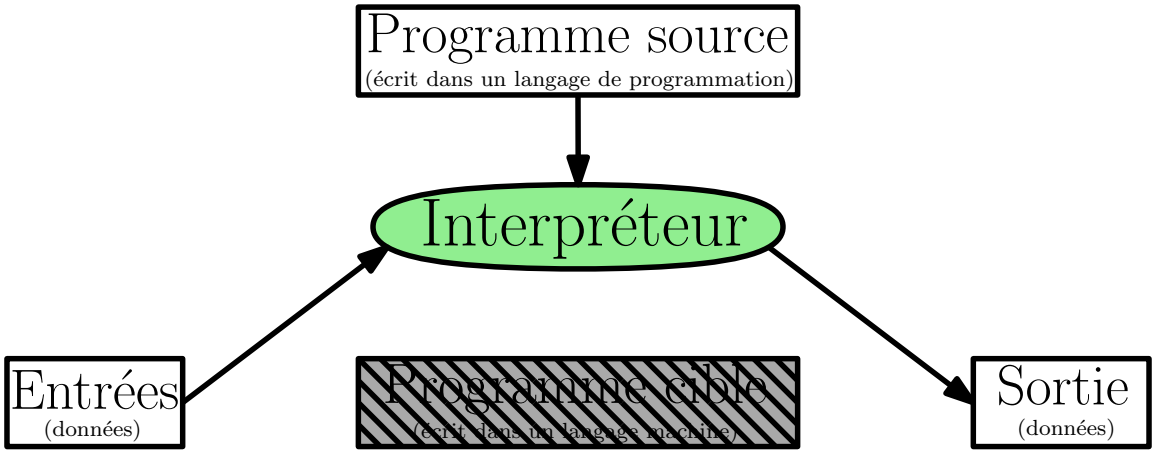
Compilateur



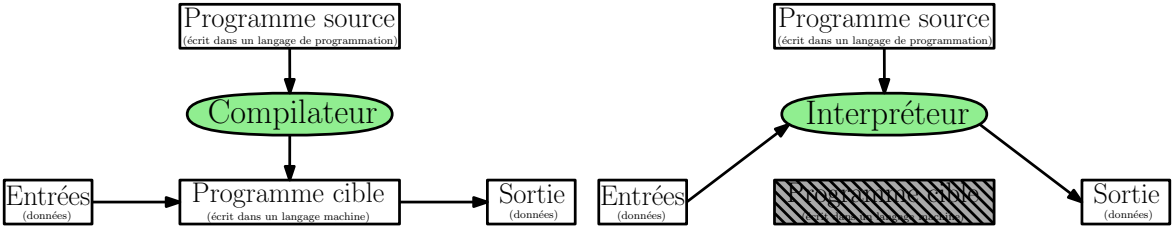
Interpréteur

- Un interpréteur est un programme :
  1. qui lit un programme rédigé dans un langage de programmation, appelé langage source
  2. qui exécute (interprète) directement les instructions du programme
- L'interpréteur doit reporter :
  - les erreurs de syntaxe (non-respect de la grammaire du langage)
  - les erreurs détectables qu'à l'exécution (par exemple appel d'une méthode sur une valeur `null`)
- L'interprétation évite l'étape de compilation, mais est généralement moins efficace lorsque le programme est utilisé de nombreuses fois.

Interpréteur



Compilateur vs interpréteur



Différences

- L'exécutable produit par un compilateur est souvent plus rapide et efficace que le traitement similaire opéré par un interpréteur.
- Un interpréteur permet d'exécuter plus facilement le code et rende plus facile sa mise à jour.

Exemple 1 : langage C → assembler

```c
#include <stdio.h>
  int main() {
  int a = 10;
  int b = 20;
  printf("%d\n", a+b);
  return 0;
}
```

```
_main:
    push    rbp
    mov rbp, rsp
    sub rsp, 16
    mov dword ptr [rbp - 4], 0
    mov dword ptr [rbp - 8], 10
    mov dword ptr [rbp - 12], 20
    mov esi, dword ptr [rbp - 8]
    add esi, dword ptr [rbp - 12]
    lea rdi, [rip + L_.str]
    mov al, 0
    call    _printf
    xor eax, eax
    add rsp, 16
```

Exemple 2 langage java → bytecode Java

```
outer:
for (int i = 2; i < 1000; i++){
    for (int j = 2; j < i; j++){
        if (i % j == 0)
            continue outer;
    }
    System.out.println (i);
}
```

```
0: iconst_2
1: istore_1
2: iload_1
3: sipush 1000
6: if_icmpge 44
9: iconst_2
10:istore_2
11:iload_2
12:iload_1
13:if_icmpge 31
16:iload_1
17:iload_2
18:irem
19:ifne 25
22:goto 38
25:iinc 2, 1
28:goto 11
31:getstatic #84;
34:iload_1
35:invokevirtual #85;
38:iinc 1, 1
41:goto 2
44:return
```

Compilation

La plupart du temps en compilation, on cherche à traduire

Programme source

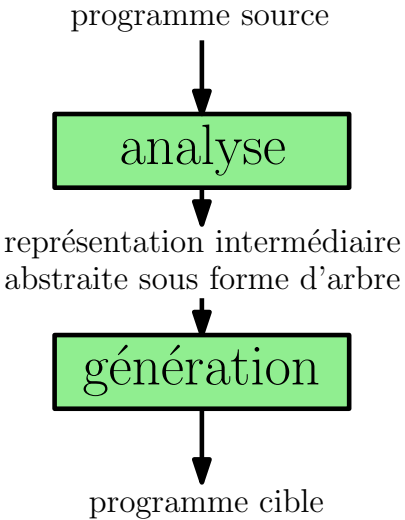
Programme en langage « haut niveau » comme le C, le JAVA, ...

en un

Programme cible

Programme en langage « bas niveau », à savoir un langage assembleur représentant le langage machine de manière lisible par un être humain, comme le bytecode Java, assembler x86, assembler ARM, ...

Étapes de compilation simplifiées



1. une phase d'analyse :
   - découpe le programme source en parties
   - détecte d'éventuelles erreurs lexicales
   - impose une structure grammaticale entre ces constituants
   - utilise cette structure comme représentation intermédiaire du programme
   - détecte d'éventuelles erreurs syntaxiques ou sémantiques
   - collecte et conserve les informations sur les fonctions et variables du programme source dans une structure de données : table des symboles
2. une phase de génération qui :
   - analyse la représentation intermédiaire et la table des symboles
   - construit le programme cible en fonction

Étapes de compilation détaillées

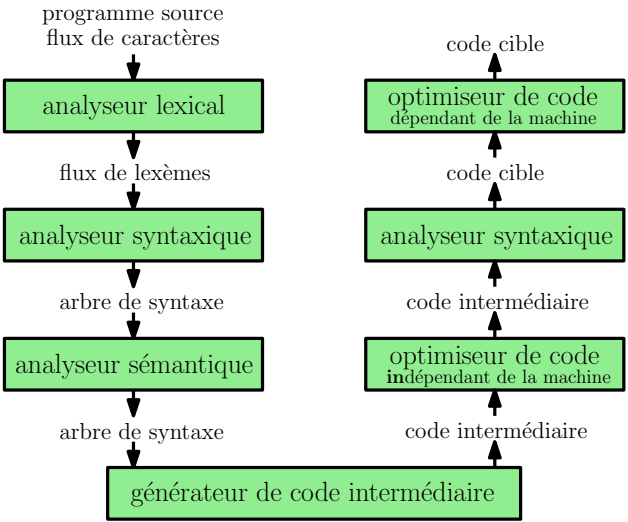


Table des symboles

L'analyse est composée de 3 parties :

- lexicale
- syntaxique
- sémantique

Dans les compilateurs modernes, il y a souvent un code intermédiaire et deux phases d'optimisation.

Étapes de compilation détaillées

1. **Analyse lexicale** : découpage du code en *tokens* (mot-clé, identifiant ou symbole)
2. **Analyse syntaxique** : construction de l'arbre de syntaxe issu des règles de la grammaire du langage
3. **Analyse sémantique** : création de la table des symboles (noms et types des variables), vérification des types, simplification de l'arbre de syntaxe, ...
4. **Génération code intermédiaire :** : code trois-adresse (séquence d'instructions avec au plus 3 opérandes)
5. **Optimisation indépendante de la machine (optionnelle)** : par exemple propagation de constante
6. **Génération de code** : attribution des registres, ...
7. **Optimisation dépendante de la machine (optionnelle)** : utilisation des instructions spécifique du processeur ciblé

Analyse lexicale

Principe

- Lit un flux de caractères et regroupe les caractères en séquences ayant un sens (identificateur de variable, nombre, opérateur, ...), appelées lexèmes
- Attribue à chaque lexème un *token* de la forme $\langle \texttt{nom}, \texttt{valeur} \rangle$ ou $\langle \texttt{nom} \rangle$, où `nom` correspond à la nature du lexème et `valeur` à sa référence dans la table des symboles (pour les variables et fonctions)

Exemple :

- Code : `p = 60 + y * x`
- Lexèmes : p, =, 60, +, y, *, x
- *Tokens* : $\langle \texttt{var,1} \rangle$ $\langle = \rangle$ $\langle \texttt{60,int} \rangle$ $\langle + \rangle$ $\langle \texttt{var,3} \rangle$ $\langle * \rangle$ $\langle \texttt{var,2} \rangle$

- Tables des symboles :

1	2	3	...
p	x	y	...
...	...	...	...

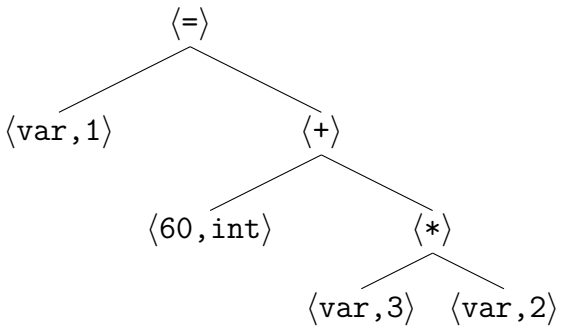
Analyse syntaxique : principes

Crée, à partir des tokens reçus de l'analyseur lexical, une représentation sous forme d'arbre illustrant la structure grammaticale du flux de lexèmes

- Arbre de syntaxe (ou de dérivation)
- Un nœud *n* représente un opérateur syntaxique
- Chaque fils de *n* représente un opérande de l'opération

Analyse syntaxique : exemple

- *Tokens* : ⟨var,1⟩ ⟨=⟩ ⟨60,int⟩ ⟨+⟩ ⟨var,3⟩ ⟨*⟩ ⟨var,2⟩
- Arbre de dérivation :



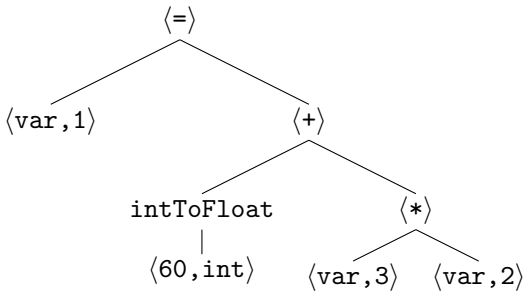
- Tables des symboles :

1	2	3	...
p	x	y	...
...	...	...	...

Analyse sémantique

Principe

- Utilise l'arbre de syntaxe et la table des symboles pour vérifier la cohérence sémantique du code par rapport à la définition du langage (sa grammaire)
- Vérification du typage des opérandes pour chaque opération (retour d'une erreur, changement de type...)



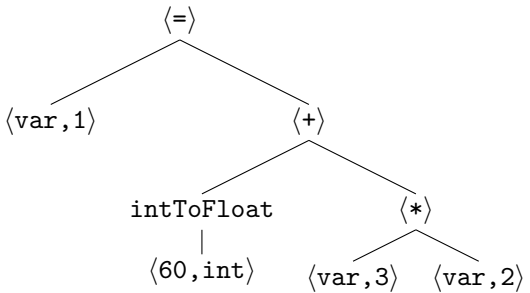
Coercition d'`int` en `float`

1	2	3	...
p	x	y	...
`float`	`float`	`float`	...
...	...	...	...

Génération de code intermédiaire

Principe

Génération d'une représentation bas niveau du programme source (code intermédiaire facile à produire et facile à traduire en assembleur)
$\Rightarrow$ code à trois adresses généré à partir de l'arbre de syntaxe et de la table des symboles




```
t1 = intToFloat(60)
t2 = var3 * var2
t3 = t1 + t2
var1 = t3
```

Optimisation du code intermédiaire

Principe

Amélioration du code intermédiaire pour obtenir un meilleur programme cible (plus efficace/rapide)

```
t1 = intToFloat(60)
t2 = var3 * var2
t3 = t1 + t2
var1 = t3
```

$\rightarrow$

```
t1 = var3 * var2
t2 = 60.0 + t1
var1 = t2
```

Génération de code cible

Principes

- Registres (zone de mémoire centrale à accès extrêmement rapide $\sim$ 1 ns) réservés pour les variables les plus utilisées
- Mémoire physique (zone de mémoire à accès rapide $\sim$ 100 ns) réservés pour toutes les autres variables

$\Rightarrow$ bien allouer les registres (en nombre très limité) lors de la traduction en assembleur

```
t1 = var3 * var2
t2 = 60.0 + t1
var1 = t2
```

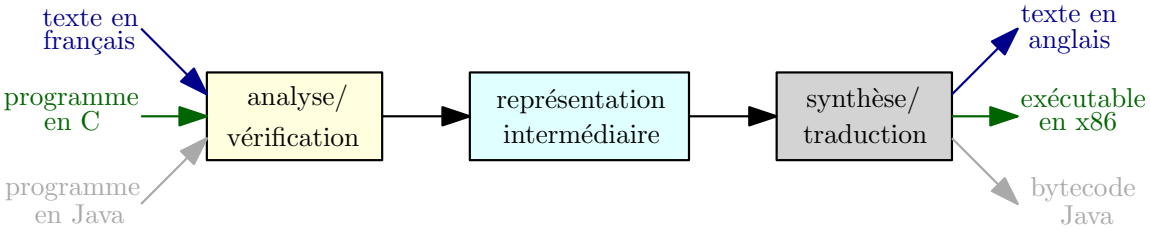
```
mov eax, [var3] ; a <- var3
mov ebx, [var2] ; b <- var2
mul eax ebx     ; a <- a * b
add eax 60.0    ; a <- a + 60.0
mov [var1] eax  ; var1 <- a
```

Compilateur en résumé

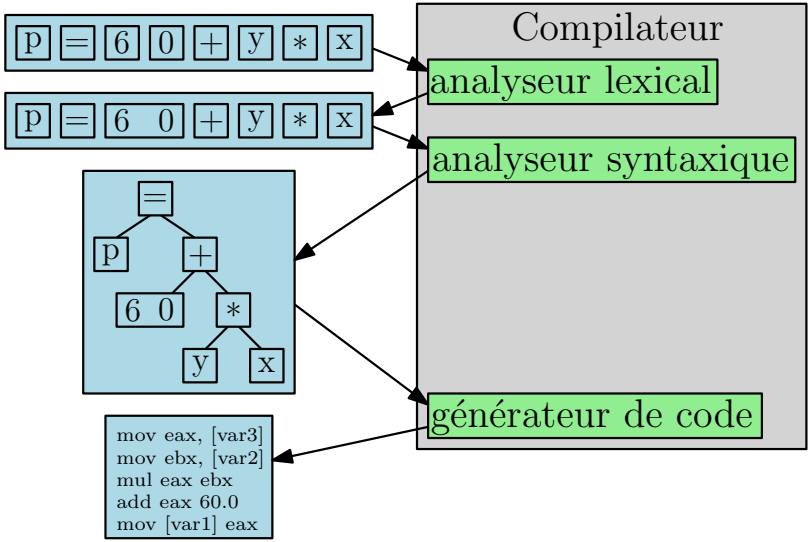
Résumé

Un **COMPILATEUR** :

- c'est un **VÉRIFICATEUR** et
- c'est un **TRADUCTEUR**



Étapes principales de compilation



Section 2

Théorie des langages : introduction/rappels

Rappels terminologie langages formels

- Les *symboles* sont des éléments indivisibles qui vont servir de briques de base pour construire des mots.
- Un *alphabet* est un ensemble fini de symboles noté Σ.
- Une suite (finie) de symboles, appartenant à un alphabet Σ, mis bout à bout est appelé un *mot* sur Σ.
- On note $|m|$ la longueur du mot m (le nombre de symboles qui le composent) et $|m|_s$ le nombre de symboles s que possède le mot m.

  Le mot de longueur zéro est noté ε.
- L'ensemble de tous les mots sur un alphabet Σ est noté Σ^*.
- Un *langage* sur un alphabet Σ est un ensemble de mots construits sur Σ et donc une *partie* de Σ^*.

Exemple de langages

- $\Sigma_1 = \{a\}, L_1 = (\Sigma_1)^* = \{\varepsilon, a, aa, aaa, \ldots\}$
- $\Sigma_2 = \{a, b\}, L_2 = \{\varepsilon, ab, aabb, aaabbb, aaaabbbb, \ldots\} = \{a^n b^n \mid n \in \mathbb{N}\}$
- $\Sigma_2 = \{a, b\}, L_3 = \{\varepsilon, aa, bb, aaaa, abba, baab, bbbb, \ldots\}$

  $L_3 = \{w_1 w_2 w_3 \ldots w_n \mid n\%2 = 0 \text{ et } \forall 1 \leq i \leq n, w_i = w_{n-i+1}\}$
- $\Sigma_3 = \{a, b, c\},\ L_4 = \{\varepsilon, abc, aabbcc, aaabbbccc, \ldots\} = \{a^n b^n c^n \mid n \in \mathbb{N}\}$

Opérations sur les langages

- Union : $L_1 \cup L_2$, $\{x \mid x \in L_1 \text{ ou } x \in L_2\}$
- Intersection : $L_1 \cap L_2$, $\{x \mid x \in L_1 \text{ et } x \in L_2\}$
- Différence : $L_1 - L_2$, $\{x \mid x \in L_1 \text{ et } x \notin L_2\}$
- Complément : $\bar{L}$, $\{x \in \Sigma^* \mid x \notin L\}$
- Concaténation : $L_1 L_2$, $\{xy \mid x \in L_1 \text{ et } y \in L_2\}$
- Auto concaténation : $\overbrace{L \dots L}^{n}$, L^n
- Fermeture de Kleene : L^*, $\bigcup_{k \geq 0} L^k$

Comment décrire un langage ?

- **Énumération**

  $L_2 = \{\varepsilon, ab, aabb, aaabbb, aaaabbbb, \ldots\}$

- **Description littéraire**

  Ensemble des mots construits sur l'alphabet $\{a, b\}$, commençant par des a et se terminant par des b et tel que le nombre de a et le nombre de b soit égal

- **Grammaire de réécriture**

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

Grammaires de réécriture

Une grammaire de réécriture est un 4-uplet $\langle N, \Sigma, P, S \rangle$ où :

- N est un ensemble de *symboles non terminaux*, appelé l'*alphabet non-terminal*.
- Σ est un ensemble de *symboles terminaux*, appelé l'*alphabet terminal*, tel que N et Σ soient disjoints.
- P est un ensemble *fini* de *règles de production* ou *règles de réécriture*.

  Une règle est de la forme $\alpha \to \beta$

  - α est appelé partie gauche de la règle
  - β est appelé partie droite de la règle
  - α et β peuvent prendre des formes différentes mais on a toujours :
    - $\alpha \in (N \cup \Sigma)^* N (N \cup \Sigma)^*$
    - $\beta \in (N \cup \Sigma)^*$
- S est un élément de N appelé l'*axiome* de la grammaire.

Notation

Pour alléger les notations, on note:

$$\alpha \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

les n règles :

$$\alpha \to \beta_1, \alpha \to \beta_2, \ldots, \alpha \to \beta_n$$

Quelques exemples de grammaires

- $\langle \{S\}, \{a\}, \{S \to Sa \mid \varepsilon\}, S \rangle$
- $\langle \{S\}, \{a, b\}, \{S \to aSb \mid \varepsilon\}, S \rangle$
- $\langle \{S\}, \{a, b\}, \{S \to aSa \mid bSb \mid aa \mid bb\}, S \rangle$
- $\langle \{S, S_1, S_2\}, \{a, b, c\}, \left\{ \begin{array}{lcllcl} S & \to & aS_1c, & S_1 & \to & b \mid SS_2, \\ cS_2 & \to & S_2c, & bS_2 & \to & bb \end{array} \right\}, S \rangle$

Proto-mots d'une grammaire

Les *proto-mots* d'une grammaire $G = \langle N, \Sigma, P, S \rangle$ sont des mots construits sur l'alphabet $\Sigma \cup N$, on les définit récursivement de la façon suivante :

- S est un proto-mot de G
- si $\alpha\beta\gamma$ est un proto-mot de G et $\beta \to \delta \in P$ alors $\alpha\delta\gamma$ est un proto-mot de G.

Un proto-mot de G ne contenant aucun symbole non-terminal est appelé un *mot généré* par G.

Le *langage généré par* G, noté $L(G)$ est l'ensemble des mots générés par G.

Dérivation

- L'opération qui consiste à générer un proto-mot $\alpha\delta\gamma$ à partir d'un proto-mot $\alpha\beta\gamma$ et d'une règle de production r de la forme $\beta \to \delta$ est appelée l'opération de dérivation. Elle se note à l'aide d'une double flèche :

  $$\alpha\beta\gamma \Rightarrow \alpha\delta\gamma$$

- On note $\alpha \overset{k}{\Rightarrow} \beta$ pour indiquer que β se dérive de α en k étapes.
- On définit aussi les deux notations $\overset{+}{\Rightarrow}$ et $\overset{*}{\Rightarrow}$ de la façon suivante :
  - $\alpha \overset{+}{\Rightarrow} \beta \equiv \alpha \overset{k}{\Rightarrow} \beta$ avec $k > 0$
  - $\alpha \overset{*}{\Rightarrow} \beta \equiv \alpha \overset{k}{\Rightarrow} \beta$ avec $k \geq 0$

Attention

Les symboles $\Rightarrow$ et $\to$ ne représentent pas la même chose.

Conventions pour la définition de grammaire

- Les symboles non terminaux appartenant à N sont représentés par des lettres latines majuscules : $A, B, C, S, E, T \ldots$
- Les symboles terminaux appartenant à Σ sont représentés par des lettres latines minuscules : $a, b, c, d \ldots$
- Les proto-mots appartenant à $(N \cup \Sigma)^*$ sont représentés par des lettres grecques minuscules : $\alpha, \beta, \gamma, \varepsilon \ldots$
- L'axiome est représenté par le non-terminal S et constitue la partie gauche de la première règle de production

Langage généré par une grammaire

- $L(G)$ est défini de la façon suivante :

$$L(G) = \{m \in \Sigma^* \mid S \overset{+}{\Rightarrow} m\}$$

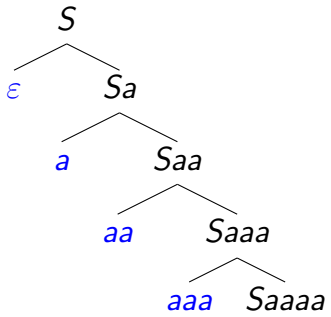
- Deux grammaires G et G' sont équivalentes si $L(G) = L(G')$.

$$G = \langle \{S\}, \{a, b\}, \{S \to aSb \mid \varepsilon\}, S \rangle$$

$L_1 = \{\varepsilon, a, aa, aaa, \ldots\}$

$G = \langle \{S\}, \{a, b\}, \{S \to Sa \mid \varepsilon\}, S \rangle$

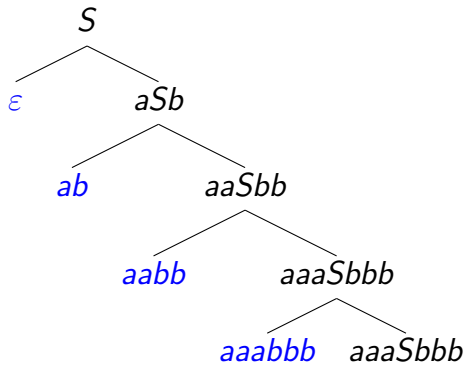
Sous-ensemble des proto-mots et mots générés de G



$L_2 = \{\varepsilon, ab, aabb, aaabbb, aaaabbbb, \ldots\}$

$G = \langle \{S\}, \{a, b\}, \{S \to aSb \mid \varepsilon\}, S \rangle$

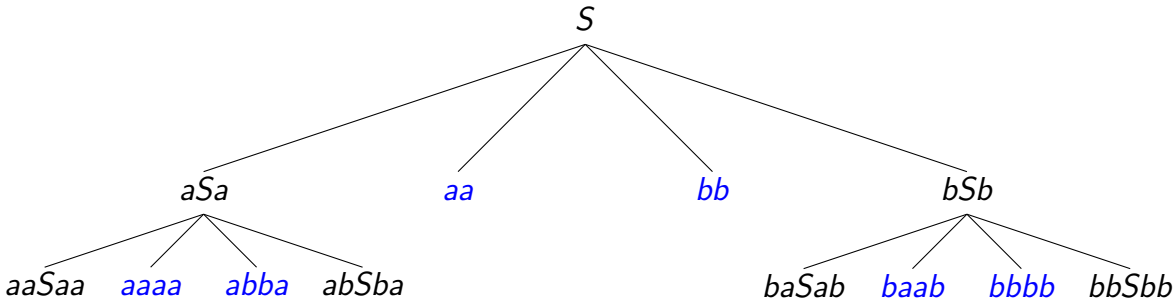
Sous-ensemble des proto-mots et mots générés de G



$L_3 = \{aa, bb, aaaa, abba, baab, bbbb, \ldots\}$

$G = \langle \{S\}, \{a, b\}, \{S \to aSa \mid bSb \mid aa \mid bb\}, S \rangle$

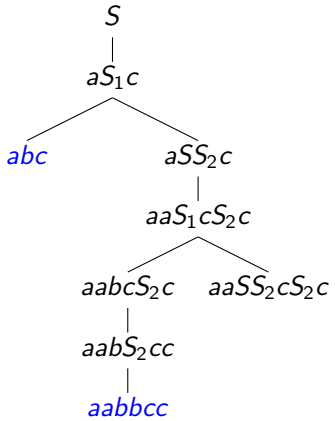
Sous-ensemble des proto-mots et mots générés de G



$L_4 = \{\varepsilon, abc, aabbcc, aaabbbccc, \ldots\}$

$$\langle \{S, S_1, S_2\}, \{a, b, c\}, \left\{ \begin{array}{rclrcl} S & \to & aS_1c, & S_1 & \to & b \mid SS_2, \\ cS_2 & \to & S_2c, & bS_2 & \to & bb \end{array} \right\}, S \rangle$$

Sous-ensemble des proto-mots et mots générés de G



Plusieurs règles peuvent s'appliquer

$$G = \langle \{E, T, F\}, \{+, *, a\}, \{E \to T + E \mid T, T \to F * T \mid F, F \to a\}, E \rangle$$

Les proto-mots générés lors d'une dérivation peuvent comporter plus d'un symbole non-terminal :

$$\begin{aligned} E &\Rightarrow T + E \\ &\Rightarrow T + T \\ &\Rightarrow F + T \\ &\Rightarrow F + F * T \\ &\Rightarrow F + a * T \\ &\Rightarrow F + a * F \\ &\Rightarrow a + a * F \\ &\Rightarrow a + a * a \end{aligned}$$

Sens de dérivation

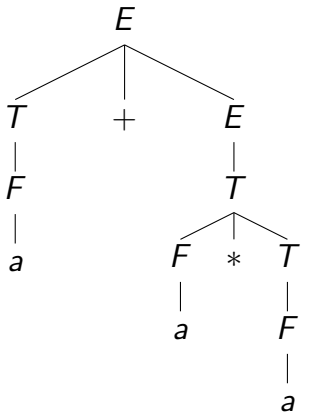
Dérivation gauche : on réécrit le non-terminal le plus à gauche :

$$
\begin{aligned}
E &\Rightarrow T + E \\
&\Rightarrow F + E \\
&\Rightarrow a + E \\
&\Rightarrow a + T \\
&\Rightarrow a + F * T \\
&\Rightarrow a + a * T \\
&\Rightarrow a + a * F \\
&\Rightarrow a + a * a
\end{aligned}
$$

Dérivation droite : on réécrit le non-terminal le plus à droite :

$$
\begin{aligned}
E &\Rightarrow T + E \\
&\Rightarrow T + T \\
&\Rightarrow T + F * T \\
&\Rightarrow T + F * F \\
&\Rightarrow T + F * a \\
&\Rightarrow T + a * a \\
&\Rightarrow F + a * a \\
&\Rightarrow a + a * a
\end{aligned}
$$

Arbre de dérivation

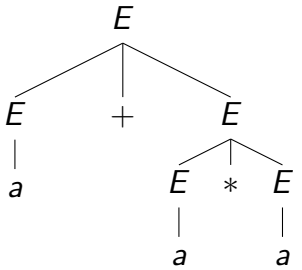


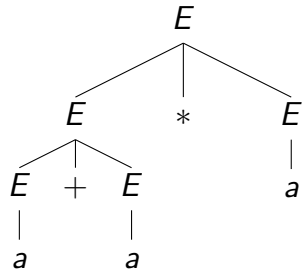
- Un arbre de dérivation pour G ($G = \langle N, \Sigma, P, S \rangle$) est un arbre ordonné et étiqueté dont les étiquettes appartiennent à l'ensemble $N \cup \Sigma \cup \{\varepsilon\}$.
- Si un nœud de l'arbre est étiqueté par le non-terminal A et ses fils sont étiquetés $X_1, X_2, ..., X_n$ alors la règle $A \to X_1, X_2, ..., X_n$ appartient à P.
- Un arbre de dérivation indique les règles qui ont été utilisées dans une dérivation, mais pas l'ordre dans lequel elles ont été utilisées.
- À un arbre de dérivation correspondent une seule dérivation droite et une seule dérivation gauche.

Grammaire ambigüe

Une grammaire G est *ambiguë* s'il existe au moins un mot m dans $L(G)$ auquel correspond plus d'un arbre de dérivation.

Exemple : règles : $E \to E + E \mid E * E \mid a$, mot $a + a * a$





On évite généralement de travailler sur ce type de grammaire en raison du non-déterminisme induit dans l'analyse.

Types de règles

Les grammaires peuvent être classées en fonction de la forme de leurs règles de production. On définit cinq types de règles de production :

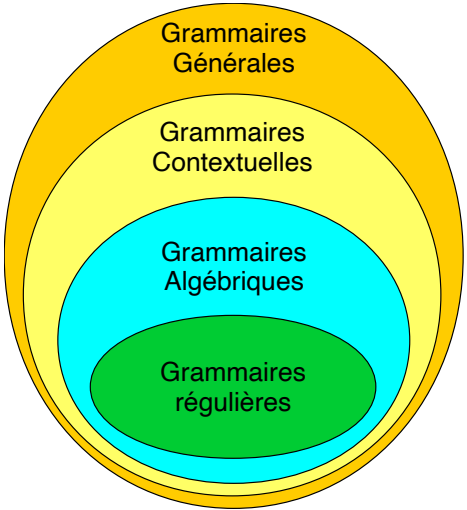
- Règle est *régulière à gauche* si et seulement si $A \to xB$ ou $A \to x$ ou $A \to \varepsilon$ avec $A, B \in N$ et $x \in \Sigma$.
- Règle *régulière à droite* si et seulement si $A \to Bx$ ou $A \to x$ ou $A \to \varepsilon$ avec $A, B \in N$ et $x \in \Sigma$.
- Règle $A \to \alpha$ est *hors-contexte* si et seulement si $A \in N$ et $\alpha \in (N \cup \Sigma)^*$
- Règle $\alpha \to \beta$ est *contextuelle* si et seulement si $\alpha = \gamma A \delta$ et $\beta = \gamma B \delta$ avec $\gamma, \delta, B \in (N \cup \Sigma)^*$ et $A \in N$. Le nom "contextuelle" provient du fait que A se réécrit B uniquement dans le contexte γ_δ.
- Une règle $\alpha \to \beta$ est *sans restriction* si elle n'est pas contextuelle.

Type d'une grammaire

Une grammaire est :

- **régulière** (de type 3) si elle est régulière à droite ou régulière à gauche. Une grammaire est régulière à gauche si *toutes* ses règles sont régulières à gauche et une grammaire est régulière à droite si *toutes* ses règles sont régulières à droite.
- **hors contexte/algébrique** (de type 2) si toutes ses règles de production sont hors contexte.
- **contextuelle** (de type 1) si toutes ses règles de production sont *contextuelles*.
- **générale** (de type 0) si toutes ses règles de production sont sans restrictions.

Hiérarchie de Chomsky



Type d'un langage

Grammaire	Règles	Langage	Machine
3	$A \to aB, \quad A \to a$	Régulier/rationnel	automate fini
2	$A \to \gamma$	Hors contexte/algébrique	automate à pile non-déterministe
1	$\alpha A \beta \to \alpha \gamma \beta$	Contextuel	Automate linéairement borné
0	$\alpha \to \beta$	récursivement énumérable	Machine de Turing

Exemples de langages réguliers

- $L_1 = \{m \in \{a,b\}^* \}$, $G_1 = \langle \{S\}, \{a,b\}, \{S \to aS \mid bS \mid \varepsilon\}, S \rangle$
- $L_2 = \{m \in \{a,b\}^* \mid |m|_a \mod 2 = 0\}$

  $G_2 = \langle \{S,T\}, \{a,b\}, \left\{ \begin{array}{lcl} S & \to & aT \mid bS \mid \varepsilon, \\ T & \to & aS \mid bT \end{array} \right\}, S \rangle$
- $L_3 = \{m \in \{a,b\}^* \mid m = xaaa \text{ avec } x \in \{a,b\}^*\}$

  $G_3 = \langle \{S,T,U\}, \{a,b\}, \left\{ \begin{array}{lcl} S & \to & aS \mid bS \mid aT, \\ T & \to & aU, \\ U & \to & a \end{array} \right\}, S \rangle$
- $L_4 = \{m \in \{a,b\}^* \mid |m|_a \mod 2 = 0 \text{ et} |m|_b \mod 2 = 0\}$

  $G_4 = \langle \{S,T,U,V\}, \{a,b\}, \left\{ \begin{array}{lcllcl} S & \to & aT \mid bU \mid \varepsilon, & T & \to & aS \mid bV, \\ V & \to & aU \mid bT, & U & \to & aV \mid bS \end{array} \right\}, S \rangle$

Exemples de langages hors-contexte

- $L_1 = \{a^n b^n \mid n \geq 0\}$

  $G_1 = \langle \{S\}, \{a, b\}, \{S \to aSb \mid \varepsilon\}, S \rangle$

- $L_2 = \{mm^{-1} \mid m \in \{a, b\}^*\}$ (langage miroir - palindromes pairs)

  $G_2 = \langle \{S\}, \{a, b\}, \{S \to aSa \mid bSb \mid \varepsilon\}, S \rangle$

Exemples de langages contextuels

$$L_1 = \{a^n b^n c^n \mid n \geq 0\}$$

$$G_1 = \langle \{S, B, \bar{B}, C\}, \{a, b, c\}, \left\{ \begin{array}{rclrcl} S & \to & aSBC \mid \varepsilon, & C\bar{B} & \to & \bar{B}B, \\ \bar{B}B & \to & \bar{B}C, & \bar{B}\bar{C} & \to & BC, \\ aB & \to & ab, & bB & \to & bb, \\ bC & \to & bc, & cC & \to & cc \end{array} \right\}, S \rangle$$