Chapitre III.
Analyse syntaxique

a.hettab@centre-univ-mila.dz

Introduction

- **L'analyse syntaxique** à plusieurs objectifs :
  - déterminer si la suite de **tokens** (unité lexicales) est **conforme** à la **grammaire** définissant le **langage source** ;
  - repérer les **erreurs** de nature syntaxique ;
  - définir une structure hiérarchique (**un arbre syntaxique**).
- Les **grammaires algébriques** ou **non contextuelles** sont suffisamment puissantes pour décrire la partie principale de la syntaxe de la plupart des **langages de programmation**.
- Dans ce chapitre nous allons faire quelques rappels sur les grammaires **non-contextuelles.**

Les grammaires non-contextuelles

- une **grammaire algébrique**, ou **grammaire non contextuelle**, aussi appelée grammaire **hors-contexte** ou grammaire « **context-free** » est une grammaire formelle dans laquelle chaque règle de production est de la forme:

$$A \rightarrow \alpha$$

où **A** est un symbole non terminal et **α** est une chaîne composée de **terminaux** et/ou de **non-terminaux**.

- Le terme « **non contextuel** » provient du fait qu'un non terminal **A** peut être remplacé par **α**, sans tenir compte du contexte où il apparaît.

Les grammaires non-contextuelles

- Une grammaire non-contextuelle G est un quadruplet **<N,T,P,S>** où :
- **N** : ensemble fini non vide de symboles appelés symboles **non terminaux**,
- **T** : ensemble fini de symboles appelés symboles **terminaux**,
- **S** : S∈N, **axiome** de la grammaire,
- **P** : ensemble de **productions**,

Chaque production est de la forme:

$$A \rightarrow \alpha \quad \text{où}$$

$$A \in N \text{ et } \alpha \in (N \cup T)^*$$

Exemple

- La grammaire G décrivant les expressions arithmétiques :
- G = <{S}, {+,-,*,/,(,),a,b,c},P,S>
- P :

$$S \rightarrow S + S$$
$$S \rightarrow S - S$$
$$S \rightarrow S * S$$
$$S \rightarrow S / S$$
$$S \rightarrow (S)$$
$$S \rightarrow a$$
$$S \rightarrow b$$
$$S \rightarrow c$$

- autre manière :

$$P : S \rightarrow S + S \mid S - S \mid S * S \mid S / S \mid (S) \mid a \mid b \mid c$$

Dérivations et arbres de dérivation

- Il existe **deux façons** pour décrire l'appartenance d'un mot au language générée à partir d'une grammaire donnée :
- la première consiste à lister une suite des applications des règles (**suite de dérivation**).
- la deuxième synthétise cette liste en un arbre, appelé (**arbre de dérivation**).

Dérivation

- **Dérivation:** Le processus par lequel une grammaire définit un langage s'appelle ***dérivation*** :
  - Soit $G = (N, T, P, S)$ une grammaire non contextuelle,
  - $A \in N$ un symbole non terminal et $\gamma \in (N \cup T)^*$ une suite de symboles, tels qu'il existe dans P une production $A \longrightarrow \gamma$.
  - Quelles que soient les suites de symboles α et β, on dit que $\alpha A \beta$ *se dérive en une étape* en la suite $\alpha\gamma\beta$ ce qui s'écrit

$$\alpha A \beta \Rightarrow \alpha\gamma\beta$$

Suite de dérivations

- **Suite de dérivations :**
  - Si $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n$ on dit que α_0 se dérive en α_n en ***n* étapes**, et on écrit:

$$\alpha_0 \overset{n}{\Rightarrow} \alpha_n.$$

  - Si α se dérive en β en un nombre quelconque, éventuellement nul, d'étapes on dit simplement que α se dérive en β et on écrit:

$$\alpha \overset{*}{\Rightarrow} \beta.$$

  - Si α se dérive en β en un nombre quelconque, non nul, d'étapes on dit simplement que α se dérive en β et on écrit:

$$\alpha \overset{+}{\Rightarrow} \beta.$$

Exemple

- Soit la grammaire G_1 :

  expression → expression "+" terme | terme

  terme → terme "*" facteur | facteur

  facteur → nombre | identificateur | "(" expression ")"

- La chaîne **w = nombre " * " identificateur " + " nombre** dérive de **expression** de la manière suivante:

- **expression** ⇒ expression "+" terme ⇒ terme "+" terme ⇒ terme "*" facteur "+" terme ⇒ facteur "*" facteur "+" terme ⇒ nombre "*" facteur "+" terme => nombre "*" identificateur "+" terme ⇒ nombre "*" identificateur "+" facteur ⇒ **nombre "*" identificateur "+" nombre**

Dérivation la plus à gauche et Dérivation la plus à droite

- **Dérivation la plus à gauche** est entièrement composée de dérivations en une étape dans lesquelles à chaque fois c'est le **non-terminal le plus à gauche** qui est **réécrit**.
- **Dérivation la plus à droite** est entièrement composée de dérivations en une étape où, à chaque étape, c'est **le non-terminal le plus à droite** qui est **réécrit**.

Exemple

- Soit la grammaire suivante :

  G = <{S,A}, {a,b},P,S>

  P : S → aAS | a

  A → SbA | SS | ba

- La chaine **aabbaa** dérive de l'axiome **S** en utilisant :
- **La dérivation la plus à gauche :**

  S ⇒ aAS ⇒ aSbAS ⇒ aabAS ⇒ aabbaS ⇒ aabbaa
- **La dérivation la plus à droite :**

  S ⇒ aAS ⇒ aAa ⇒ aSbAa ⇒ aSbbaa ⇒ aabbaa

langage engendré par une grammaire

- Soit $G = (N, T, P, S)$ une grammaire non contextuelle ; le **langage engendré par G** est l'ensemble des chaînes de symboles terminaux qui dérivent de S:

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

  - Si une chaîne $v \in L(G)$ on dit que v est une ***phrase*** de G.
  - Plus généralement, si $\alpha \in (T \cup N)^*$ est tel que $S \overset{*}{\Rightarrow} \alpha$ alors on dit que α est une ***proto-phrase*** de G.
  - Une **proto-phrase** dont tous les symboles sont terminaux est une **phrase**.

Exemple

Exemple1 :

- Soit la grammaire **G1=<N,T,P,S>** avec :

  N = {S} , T = {a,b} ,

  P = {S→aSb ; S→ε }

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

Exemple2 :

Soit la grammaire **G2=<N,T,P,S>** avec :

N = { S } , T = { a, b }

P = { **S → aaS | bbS | ε** }

$L(G) = \{(aa \mid bb)^*\}$

Arbre de dérivation

- Soit w une chaîne de symboles terminaux du langage $L(G)$; il existe donc une dérivation telle que $S \overset{*}{\Rightarrow} w$. Cette dérivation peut être représentée graphiquement par un arbre, appelé ***arbre de dérivation***, défini de la manière suivante :
- **la racine de l'arbre** est le symbole de départ **S**;
- les nœuds intérieurs sont étiquetés par des symboles **non terminaux** ;
- si un **nœud intérieur** e est étiqueté par le symbole A

  et $A \rightarrow A_1 A_2 \ldots A_k$ est une **production** de la grammaire alors les **fils** de e sont des nœuds étiquetés, de la gauche vers la droite, par $A_1, A_2 \ldots A_k$;
- les **feuilles** sont étiquetées par des **symboles terminaux**.

Exemple

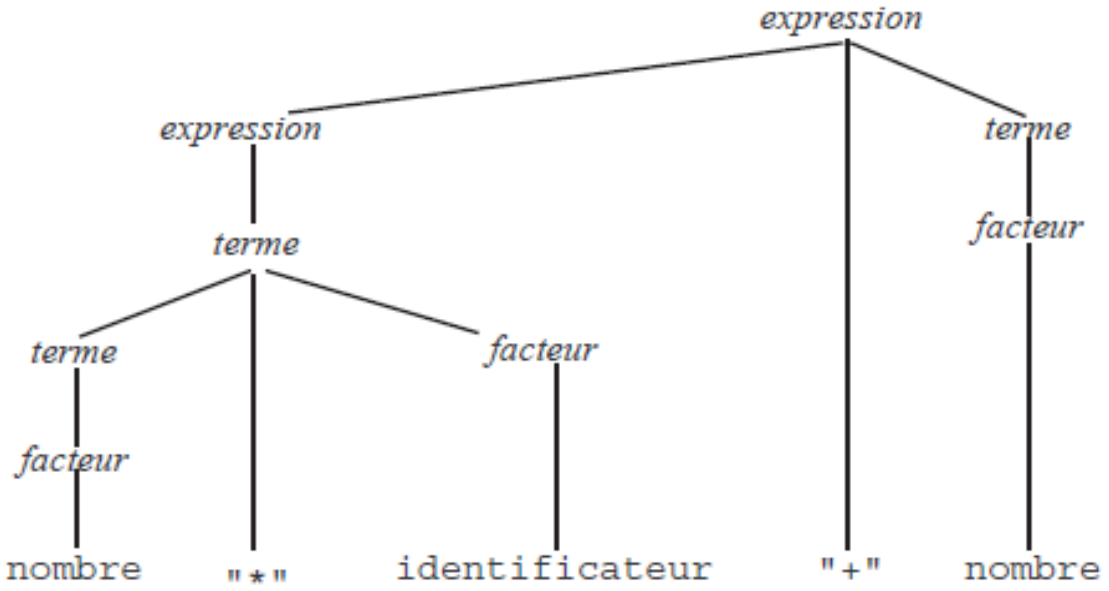
L'arbre de dérivation pour la chaîne: **nombre " * " identificateur " + " nombre**

Les productions de la grammaire:

expression → expression "+" terme | terme

terme → terme "*" facteur | facteur

facteur → nombre | identificateur | "(" expression ")"



Grammaire ambiguë

- Une grammaire est **ambiguë** s'il existe **plusieurs dérivations gauches (ou droite) différentes** pour une même chaîne de terminaux.
- Une grammaire est **ambiguë** lorsqu'au moins un mot engendré par la grammaire possède au moins **deux arbres de dérivation distincts**.
- **On notera** que l'ordre des dérivations (gauche, droite) ne se voit pas sur l'arbre de dérivation.

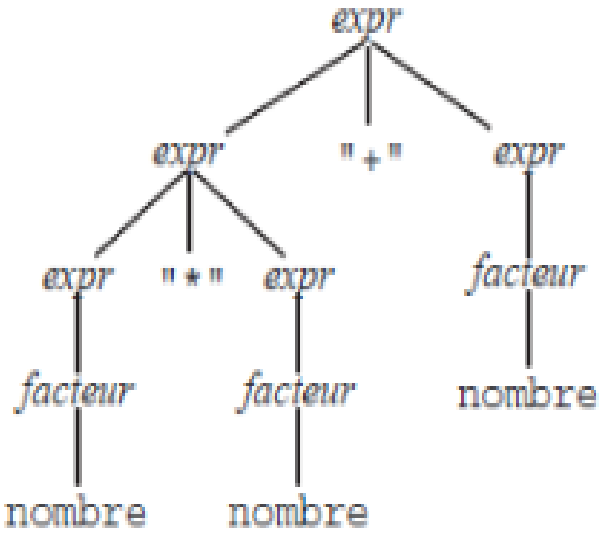
Exemple

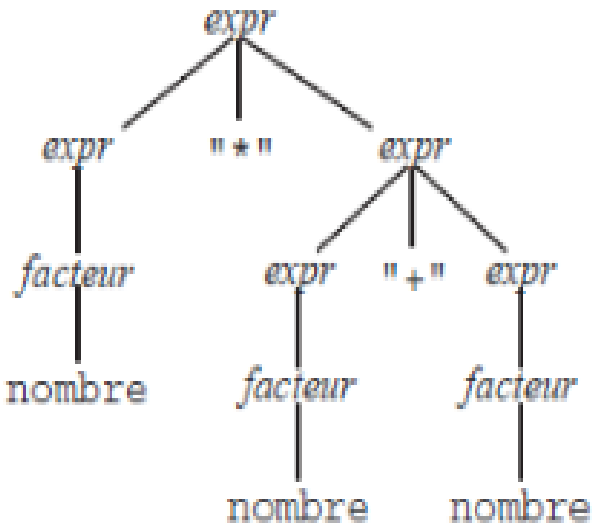
- Soit la grammaire G_1 :

  exp → exp "+" exp | exp "*" exp | facteur

  facteur → nombre | ident | "(" exp ")"

- Cette grammaire est **ambiguë** car elle possède **deux arbres de dérivation distincts pour la chaîne: nombre "*" nombre "+" nombre**





Analyses descendantes et ascendantes

- La plupart des méthodes d'analyse syntaxique tombent dans **deux classes** :
- **les méthodes d'analyse descendantes** (**top-down parsing en anglais**). Elles sont les plus populaires car les analyses correspondants sont **faciles** à mettre en œuvre à la main ;
- **les méthodes d'analyse ascendantes** (**bottom-up parsing en anglais**). Elles sont les plus **puissantes** et sont celles utilisées par les outils qui génèrent automatiquement un analyseur à partir d'une grammaire (exemple : **yacc**).

Analyse descendante

- Dans les analyses **descendantes** :
  - l'arbre d'analyse est construit en partant de la **racine** et en allant jusqu'au **feuilles** ;
  - l'analyse s'appuie sur une **dérivation à gauche** de la chaîne, c'est-à-dire qu'on remplace toujours en premier le non-terminal le **plus à gauche**.

Analyse descendante

Exemple:

Soit la grammaire **G=<N,T,P,S>** avec :

N = {**E,T,F**} ,T = {**+,*,),(,i**} ,

P = {

E → E + T | T

T → T × F | F

F → (E) | i

}

- On présente ci-dessous la dérivation de la chaîne **(i + i) * i** depuis la racine de l'arbre d'analyse.

Analyse descendante

Exemple (suite):

E (Étape 0)

E ⇒ T (Étape 1)

⇒ T * F (Étape 2)

⇒ (E) * F (Étape 3)

⇒ (E+T) * F (Étape 4)

⇒ (T+T) * F (Étape 5)

⇒ (F+T) * F (Étape 6)

⇒ (i+T) * F (Étape 7)

⇒ (i+F) * F (Étape 8)

⇒ (i+i) * F (Étape 9)

⇒ **(i+i) * i** (Étape 10)

Analyse ascendante

- Dans les analyses **ascendante** on agit **inversement** :
- l'arbre est construit en **remontant** à la **racine** **depuis** les **feuilles** en utilisant des **dérivations inverses** ;
- l'analyse s'appuie sur une **dérivation à droite** de la chaîne c'est-à-dire qu'on remplace toujours en premier le non-terminal le plus à droite.

Analyse ascendante

Exemple:

Soit la grammaire **G=<N,T,P,S>** avec :

N = {**E,T,F**} ,T = {**+,*,),(,i**} ,

P = {

E → E + T | T

T → T × F | F

F → (E) | i

}

- On présente ci-dessous la dérivation inverse de la chaîne **(i + i) * i** depuis les feuilles de l'arbre d'analyse.

Analyse ascendante

Exemple (suite):

(i+i)*i (Étape 0)

⇐(F+i)*i (Étape 1)

⇐(T+i)*i (Étape 2)

⇐(E+i)*i (Étape 3)

⇐(E+F)*i (Étape 4)

⇐(E+T)*i (Étape 5)

⇐(E)*i (Étape 6)

⇐F*i (Étape 7)

⇐T*i (Étape 8)

⇐T*F (Étape 9)

⇐T (Étape 10)

⇐**E** (Étape 11)

Traitement des erreurs syntaxiques

- En général, **plus grande** part du traitement des **erreurs** est faite par le **parser** car une grande partie des **erreurs** effectuées sont par nature **syntaxiques.**
- Un **gestionnaire des erreurs** dans un **Parser** doit:
  - indiquer la présence d'erreurs de façon claire et précise le **lieu** et la **cause** (**le lieu**: **exemple** : visualiser la **ligne** erronée, avec marqueur désignant **la position de l'erreur.** **la cause**: - **parenthèse fermante** en plus - **point-virgule** manquant)
  - traiter chaque erreur rapidement
  - ne pas ralentir de façon significative la compilation d'un programme correct

les stratégies de récupération sur erreur

- Il existe plusieurs techniques de récupération sur erreur.

  En particulier :

  - **Mode panique,**
  - **Correction locale**
  - **Productions d'erreurs**
  - **Correction globale**

Récupération sur erreur en mode panique

- La plus simple à **implanter**, applicable dans la **plupart** des méthodes d'analyse

Pendant le déroulement de l'analyse syntaxique, un ensemble d'unités lexicales (**symboles**) **de synchronisation** est continuellement mis à jour. Exemples (le symbole **End if** dans une structure conditionnelle **if**. **le point-virgule** dans une **instruction d'affectation**).

- Lorsque une erreur est détectée, on saute (ignore) tous les symboles donnés par le **scanner** jusqu'à ce qu'on rencontre un **symbole de synchronisation**

Récupération sur erreur en mode panique

- **Avantages**
  - Simple
  - Ne boucle pas à l'infini
- **Inconvénients**
  - Une partie considérable du programme **peut être sautée** sans vérifier sa validité
- **Conclusion**
  - **La récupération en mode panique** est très adéquate dans le cas où les erreurs multiples sont rares dans une même instruction

Récupération sur erreur en mode correction locale

- Lorsque une erreur est détectée, des **corrections locales** sont effectués en **modifiant** le **préfixe** du texte source restant de l'instruction en question. (**Exemples** : -remplacement d'une virgule par un point-virgule. -Suppression d'un point-virgule. - insertion d'un point-virgule manquant)
- Le **remplacement** choisi ne doit pas conduire à **une boucle infinie** (**Exemple**: si on insère toujours quelque chose devant le symbole courant de la chaîne)
- **Inconvénient** :
- Difficulté à gérer les situations dans lesquelles l'erreur réelle s'est produite avant le point de détection.

Récupération sur erreur en mode production d'erreurs

- L'utilisation **de règles de production d'erreurs** est une méthode de **récupération sur erreur**.
- Elle consiste à ajouter à la grammaire d'un langage **des productions contenant la notion d'erreur**.
- Cette technique est notamment utilisée par **Yacc**.

Récupération sur erreur en mode production d'erreurs

Exemple: Soit la grammaire **G=<N,T,P,S>** avec :

N = {*Expr*, *Op*} ,T = {+,-,*,/,id} , **P** = { :

Expr → ***Expr Op Expr*** **|(*Expr*) |- *Expr* | id**

Op → **+ | - | * | / }**

Pour détecter une parenthèse fermante mal placée, il suffit d'ajouter les règles:

Erreur → Op) | Expr)

- **Inconvénient** : Il faut un grand nombre de règles pour traiter un grand nombre d'erreurs.

Récupération sur erreur en mode correction globale

- Lorsqu'une chaîne d'entrée erronée (**instruction X**) , l'analyseur crée un arbre d'analyse pour une **instruction Y** sans erreur la plus proche de **X**. Cela peut permettre à l'analyseur d'apporter des modifications minimes dans le code source en remplaçant **X** par **Y.**

- **Inconvénient:**

Méthode trop coûteuse en mémoire et en temps. Pour l'instant, elle n'a pas encore été mise en œuvre dans la pratique.