Compilation

Analyse syntaxique

SI4 — 2018-2019

Erick Gallesio

Introduction (1 / 2)

Les **langages rationnels** (ou réguliers):

- très utiles pour représenter les lexèmes des langages de programmation (entre autres);
- constituent la classe des langage formels les moins puissants;
- ne permettent pas d'analyser les langages du type $\{(^n \ldots)^n \mid n \geq 0 \}$

Or ces formes sont très présentes dans les langages de programmation:

- expressions parenthésées: `(3 * (2 + 4))`
- structures de contrôle: `if x then if y then z else fi else w fi`
- blocs: `{ int x; .. { int y ; .... } ...}`

→ Utilisation de **langages algébriques**.

Introduction (2 / 2)

L'analyse syntaxique

- reçoit une suite de lexèmes de l'analyseur lexical
- doit vérifier que cette suite peut être engendrée par la grammaire du langage.

Donc, étant donnés

- une grammaire G
- un mot *m* (un programme)

le problème consiste à déterminer si $m \in L(G)$, le langage généré par G.

Les **grammaires algébriques** permettent de décrire facilement les langages de programmation.

Grammaire algébrique (1 / 3)

Définition:

Une grammaire algébrique est le quadruplet `G=(T, N, S, P)` où

- **T** est un ensemble de symboles terminaux
- **N** est un ensemble de symboles non terminaux
- **S** est un symbole particulier de **N**, appelé *axiome*
- **P** est un ensemble de règles de production de la forme $X \rightarrow \phi_1 \ldots \phi_n$ où:
- $X \in N$
- $\phi_i \in N \cup T \cup \{ \varepsilon \}$

Grammaire algébrique (2 / 3)

Dérivation:

c'est l'application d'une ou plusieurs règles de production à partir d'un mot appartenant à $(T \cup N)^+$.

On note:

- $m \rightarrow m'$ une dérivation
  - *m est de la forme* $\alpha R \beta$,
  - *m' est de la forme* $\alpha \delta_1 \ldots \delta_n \beta$
  - *et il existe une règle de* ***P*** *telle que* $R \rightarrow \delta_1 \ldots \delta_n$
- $m \overset{*}{\rightarrow} m'$ une suite dérivations obtenue par l'application de n règles de **P**

Langage engendré:

$$\{ a_1 \ldots a_n \mid \forall\, i\ a_i \in T \wedge S \overset{*}{\rightarrow} a_1 \ldots a_n \}$$

Grammaire algébrique (3 / 3)

Langage algébrique:

le langage **L** est dit algébrique (ou *context free*) si il existe une grammaire algébrique telle que **L** = *L(G)*

Soit la grammaire G1 des expressions ({var, cst, +, *}, { EXP }, EXP, **P**),
où **P** =

```
EXP → EXP + EXP
EXP → EXP * EXP
EXP → var
EXP → cst
```

Exemples de dérivations:

EXP → EXP + EXP → EXP * EXP + EXP
EXP *→ EXP * EXP + EXP
EXP *→ cst * var + cst

Notes:

- **L(G1) est rationnel:** (var | cst)((+ | *) (var | cst))*
- Avec l'ajout de la règle `EXP → ( EXP )`, **il ne le serait plus**.

Grammaires & langages de programmation

- L'alphabet **terminal** correspond aux lexèmes renvoyés par le lexical:
  - *mots-clés*
  - *identificateurs,*
  - *constantes,*
  - …
- L'alphabet **non terminal** correspond
  - *à des constructions sémantiques du langage: DÉCLARATION, EXPRESSION, ÉNONCÉ, …*
  - *à des constructions syntaxiques: LISTE, SUITE, …*
- L'axiome correspond à une construction autonome compilable:
  - *PROGRAMME*
  - *MODULE*
  - *CLASSE*

Forme de Backus Naur (BNF)

Pour simplifier, on utilise souvent des méta-opérateurs pour décrire les grammaires des langages de programmation.

Alternative:

instruction ::= instruction_if | instruction_while ⇔
instruction → instruction_if
instruction → instruction_while

Elément optionnel:

instruction_return ::= 'return' [expression] ';' ⇔
instruction_return → 'return' expression ';'
instruction_return → 'return' ';'

Répétition 0→n fois:

terme ::= facteur { oper_mult facteur } ⇔
terme → facteur oper_mult facteur
terme → facteur

Exemple for de C/Java:

```
instr_for ::= 'for' '(' [init] ';' [cond] ';' [update] ')' instr
```

Arbre de dérivation (1 / 4)

Un **arbre de dérivation** (ou arbre syntaxique) est un arbre tel que pour la grammaire
G = (**T**, **N**, **S**, **P**)

- la racine est l'axiome **S**
- les nœuds de l'arbre sont des non terminaux de **N**
- les feuilles de l'arbre sont des terminaux de **T**
- les fils d'un nœud A de l'arbre sont $\alpha_0 \ldots \alpha_n$ ssi la règle $A \rightarrow \alpha_0 \ldots \alpha_n \in \mathbf{P}$

Prenons la grammaire G1:

```
E → E + E
  | E * E
  | (E)
  | id
```

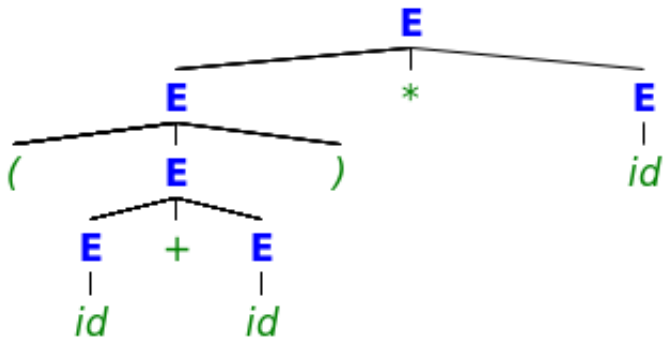
Par exemple, l'expression **(id + id) * id** est une phrase valide.

Arbre de dérivation (2 / 4)

Pour l'expression **(id + id) * id** :

Dérivations gauche	Dérivations droite
E * E	E * E
(E) * E	E * id
(E + E) * E	(E) + id
(id + E) * E	(E + E) * id
(id + id) * E	(E + id) * id
(id + id) * id	(id + id) * id

Les deux dérivations sont **différentes** mais elles ont un arbre de dérivation **identique**.

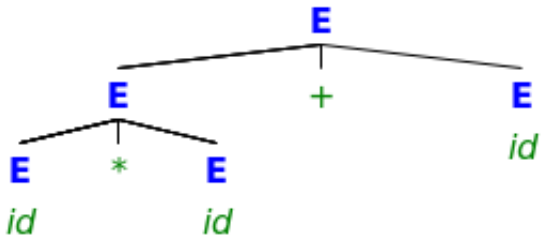


Arbre de dérivation (3 / 4)

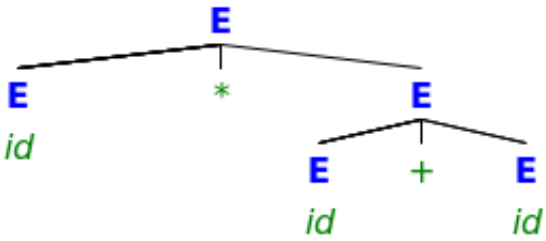
Grammaire G1 : **E → E + E | E * E | (E) | id**

Considérons la phrase: **id * id + id**

Nous pouvons construire l'arbre de dérivation (*à droite*):



Mais nous pouvons aussi construire l'arbre de dérivation (*à gauche*):

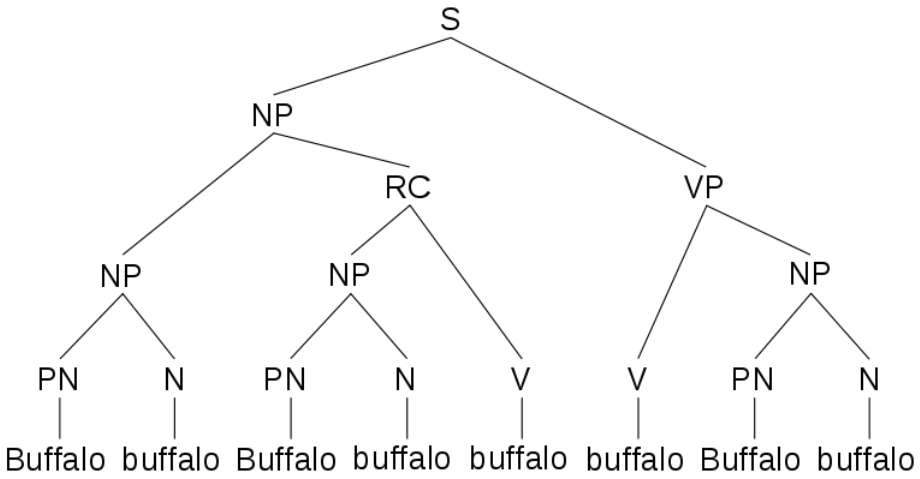


$\Longrightarrow$ deux **dérivations différentes** qui conduisent à deux **arbres différents**.

Arbre de dérivation (4 / 4)

Récréation:

Arbre de dérivation de la phrase
«Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo»
(en français : « Les bisons de Buffalo que des bisons de Buffalo intimident intimident des bisons de Buffalo. »)




```
PN : nom propre (proper noun)
N : nom commun (noun)
V : verbe (verb)
RC : proposition relative (relative clause)
NP : syntagme nominal (noun phrase)
VP : syntagme verbal (verb phrase)
S : phrase (sentence)
```

Arbre provenant de Wikipedia

Ambiguïté des grammaires (1/2)

Définition 1:

> Une grammaire algébrique G est ambiguë si il existe deux arbres de dérivation distincts pour un même mot du langage *L(G)*.

Définition 2:

> un langage **L** est ambigu si il n'existe aucune grammaire G non ambiguë,
> telle que *L = L(G)*.

Noter que l'on peut avoir une grammaire ambiguë alors que le langage ne l'est pas.

En général, l'ambiguïté d'une grammaire est une *mauvaise* propriété car elle permet d'associer **deux sémantiques différentes** à une même *phrase*.

Ainsi, **3 * 2 + 5** s'évaluerait

- en 6 + 5 = 11 avec les dérivations gauches
- en 3 * 7 = 21 avec les dérivations droites

Ambiguïté des grammaires (2/2)

On peut lever l'ambiguïté de **G1** en la réécrivant en **G2**:

```
E → E + T | T
T → T * id | T * (E) | id | (E)
```

Cette écriture rend «*» plus prioritaire que «+»:

En fait, on a:

- E → T + T + T …..
- T -> id * id * id * (…) * id

donc les produits sont toujours dans un arbre de racine **T** et **E** consiste en une somme de produits.

Cette écriture de la grammaire (**G2**) correspond à la sémantique classique des expressions arithmétiques.

L'énoncé if (1/2)

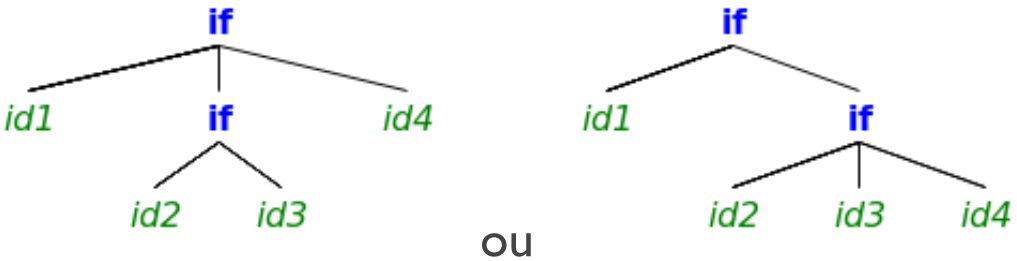
Soit la grammaire:

```
E → if E then E else E
  | if E then E
  | id
```

Considérons maintenant la phrase

```
if id1 then if id2 then id3 else id4
```

Nous avons deux arbres possibles:



En général, on préfère la forme de droite (le `else` associé au `if` le plus proche)

L'énoncé if (2/2)

Pour lever l'ambiguïté, on peut réécrire cette grammaire en passant par deux nouveaux non terminaux:

- IFE qui n'engendre que des `if` avec `else`
- IF qui peut engrender les deux types de `if`

```
E   → IFE
    | IF

IFE → if E then IFE else IFE
    |  id

IF  → if E then E
    | if E then IFE else IF
```

Constat:

c'est compliqué!

Gestion des ambiguïtés

- Pas de méthode pour convertir automatiquement une grammaire ambiguë en une grammaire non ambiguë.
- une grammaire non ambiguë équivalente est souvent plus compliquée/lourde

Parfois, on préfère

- conserver l'ambiguïté pour ne pas alourdir la grammaire et
- utiliser un mécanisme annexe (priorité, associativité, …) pour lever l'ambiguïté.

On verra plus tard comment gérer les ambiguïtés classiques avec *yacc/bison*.

Mise en œuvre de l'analyse syntaxique

L'analyseur syntaxique essaye de construire un arbre de dérivation pour savoir si une phrase (programme) est correcte.

- s'il réussit à construire l'arbre → la phrase est correcte
- s'il échoue → la phrase est incorrecte

***Deux méthodes d'analyse*:**

Méthode descendante:

en partant de l'axiome de la grammaire, on essaie de «tomber» sur la phrase à analyser.

Méthode asccendante:

on avance sur la phrase et on essaye de trouver les règles qui ont été appliquées. On remonte donc dans l'arbre. Si on remonte jusqu'à la racine, la phrase est correcte.