Grammaires Formelles : jamais “vu” mais souvent croisé

Dans votre jeunesse

En cours de Français (ou autre langue)

À l'institut Galilée

- *N3* : les “free types” de *Spécif*,
- *N4* : les “algebraic data types” de OCaml et la grammaire du λ-calcule en *PF*,
- *N5* : grammaire vs automates à pile en *Automates*,
- *N6* : les ADT de OCaml en *Principes de prog*,

Un exemple de grammaire (projet compil)

⟨programme⟩ ::= ⟨commande⟩ | ⟨commande⟩ ⟨programme⟩

⟨commande⟩ ::= `if (` ⟨expression⟩ `)` ⟨commande⟩ `else` ⟨commande⟩
| ⟨expression⟩ `;`
| `while (` ⟨expression⟩ `)` ⟨commande⟩
| `for(`⟨expression⟩`;`⟨expression⟩`;`⟨expression⟩`)`⟨commande⟩
| `;`
| `{`⟨programme⟩`}`

⟨expression⟩ ::= ⟨NUMBER⟩ | ⟨BOOLEEN⟩ | ⟨IDENT⟩
| `(`⟨expression⟩`)`
| ⟨expression⟩ `+` ⟨expression⟩
| ⟨expression⟩ `-` ⟨expression⟩
| ⟨expression⟩ `*` ⟨expression⟩
| ⟨expression⟩ `/` ⟨expression⟩
| ⟨expression⟩ `==` ⟨expression⟩
| `-` ⟨expression⟩
| ⟨IDENT⟩ `=` ⟨expression⟩

Non-terminaux et Terminaux

⟨programme⟩ ::= ⟨commande⟩ | ⟨commande⟩ ⟨programme⟩

⟨commande⟩ ::= `if (` ⟨expression⟩ `)` ⟨commande⟩ `else` ⟨commande⟩
| ⟨expression⟩ `;`
| `while (` ⟨expression⟩ `)` ⟨commande⟩
| `for(`⟨expression⟩`;`⟨expression⟩`;`⟨expression⟩`)`⟨commande⟩
| `;`
| `{`⟨programme⟩`}`

⟨expression⟩ ::= ⟨NUMBER⟩ | ⟨BOOLEEN⟩ | ⟨IDENT⟩
| `(`⟨expression⟩`)`
| ⟨expression⟩ `+` ⟨expression⟩
| ⟨expression⟩ `-` ⟨expression⟩
| ⟨expression⟩ `*` ⟨expression⟩
| ⟨expression⟩ `/` ⟨expression⟩
| ⟨expression⟩ `==` ⟨expression⟩
| `-` ⟨expression⟩
| ⟨IDENT⟩ `=` ⟨expression⟩

Non-terminaux et Terminaux

Les Terminaux

Ce sont les (noms des) tokens issus du lexeur !

Exemples : NOMBRE, IDENTIFIANT, IF, STRING...

Lexeur et token

Seront vu en détail au second semestre.
C'est un moyen de formaliser la diférence entre des entiers, des noms de variables, ou des mots-clés par exemple.

Les Non-Terminaux

Ce sont les grandes classes structurant une phrase

Exemples : expressions, commandes, groupe nominal, COI...

Signification

⟨programme⟩ peut être ⟨commande⟩
⟨programme⟩ peut être ⟨commande⟩ ⟨programme⟩
⟨commande⟩ peut être `if (` ⟨expression⟩ `)` ⟨commande⟩ `else` ⟨commande
⟨commande⟩ peut être ⟨expression⟩`;`
⟨commande⟩ peut être `while (` ⟨expression⟩ `)` ⟨commande⟩
⟨commande⟩ peut être `for(`⟨expression⟩`;`⟨expression⟩`;`⟨expression⟩`)`⟨comm
⟨commande⟩ peut être `;`
⟨commande⟩ peut être `{`⟨programme⟩`}`
⟨expression⟩ peut être ⟨NUMBER⟩
⟨expression⟩ peut être ⟨BOOLEEN⟩
⟨expression⟩ peut être ⟨IDENT⟩
⟨expression⟩ peut être `(`⟨expression⟩`)`
⟨expression⟩ peut être ⟨expression⟩ `+` ⟨expression⟩
⟨expression⟩ peut être ⟨expression⟩ `-` ⟨expression⟩
⟨expression⟩ peut être ⟨expression⟩ `*` ⟨expression⟩
⟨expression⟩ peut être ⟨expression⟩ `/` ⟨expression⟩
⟨expression⟩ peut être ⟨expression⟩ `==` ⟨expression⟩
⟨expression⟩ peut être `-` ⟨expression⟩
⟨expression⟩ peut être ⟨IDENT⟩ `=` ⟨expression⟩

Signification

⟨programme⟩ peut être ⟨commande⟩
⟨programme⟩ peut être ⟨commande⟩ ⟨programme⟩
⟨commande⟩ peut être `if (` ⟨expression⟩ `)` ⟨commande⟩ `else` ⟨commande
⟨commande⟩ peut être ⟨expression⟩`;`
⟨commande⟩ peut être `while (` ⟨expression⟩ `)` ⟨commande⟩
⟨commande⟩ peut être `for(`⟨expression⟩`;`⟨expression⟩`;`⟨expression⟩`)`⟨comm
⟨commande⟩ peut être `;`
⟨commande⟩ peut être `{`⟨programme⟩`}`
⟨expression⟩ peut
⟨expression⟩
⟨expression⟩
⟨expression⟩
⟨expression⟩ peut
⟨expression⟩ peut être ⟨expression⟩ `-` ⟨expression⟩
⟨expression⟩ peut être ⟨expression⟩ `*` ⟨expression⟩
⟨expression⟩ peut être ⟨expression⟩ `/` ⟨expression⟩
⟨expression⟩ peut être ⟨expression⟩ `==` ⟨expression⟩
⟨expression⟩ peut être `-` ⟨expression⟩
⟨expression⟩ peut être ⟨IDENT⟩ `=` ⟨expression⟩

Associe à un non-terminal
un mot composé de
terminaux et non-terminaux

Définition formelle

Une grammaire est donnée par

- Un ensemble $\mathcal{T}$ de terminaux,
- Un ensemble $\mathcal{N}$ de non-terminaux,
- Un non-terminal principal $S \in \mathcal{N}$,
- Une relation $(\mapsto) \subseteq \mathcal{N} \times (\mathcal{T} \uplus \mathcal{N})^*$ décrivant les règles.

Attention au mot vide

Une règle $\langle \text{arguments} \rangle \mapsto \epsilon$
associe le non terminal $\langle \text{arguments} \rangle$ au mot vide (et non au caractère ϵ !)

Les règles ne regardent que le nom des non-terminaux

Les noms terminaux $\mathcal{T}$ sont des tokens, mais les dans les règles on ne regarde que leur noms $\mathcal{T}$.

En FdlP : beaucoup plus informelles

Type	$\tau ::=$	`Num`	(nombres)
		`Str`	(textes)
		$\tau_1 \to \tau_2$	(fonctions)
		$(\tau_1, \ldots, \tau_n)$	(produit)
		`Unit`	(produit vide)

Expresion	$e ::=$	x	(variable)
		n	(nombre littéral)
		$e_1 + e_2$	(ajouter)
		$e_1 * e_2$	(multiplier)
		$(e_1, \ldots, e_k)$	(tuple)
		`let x=`e_1 `in` e_2	(définition)

Token implicites

On ne va pas expliciter les tokens (en part. mots-clés et opérateurs)

Les "..."

Si la version formelle est longue et évidente, on simplifie...

Par contre :

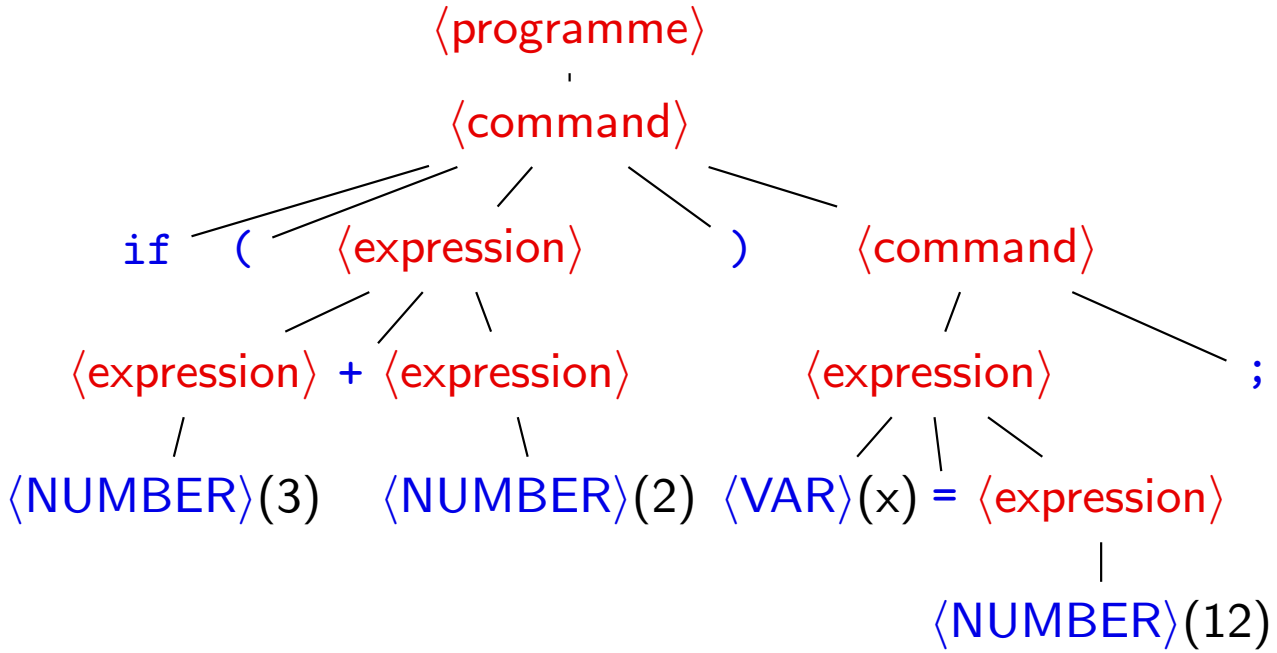
Méta-variables

On utilise des méta-variables et des indexes

Noms (rêgles et NTs)

On donne des noms aux rêgles et non-terminaux

Arbre Syntaxique (def informelle)



Un arbre syntaxique est un arbre :

- Dont les noeuds internes sont des non-terminaux
- Dont les feuilles sont des terminaux
- Dont les fils d'un non-terminal N correspondent à une règle sur N.

Attention, avec cette définition, un noeud interne peut ne pas avoir de fils (règle vers le mot vide) et ne pas être une feuille.

Arbre Syntaxique (def formelle)

Soit $\mathcal{G} = (\mathcal{T}, \mathcal{N}, S, \tau)$ une grammaire.

On définit les arbres syntactiques de $\mathcal{G}$ comme

Un ensemble $\mathcal{N}$ de tokens/noeuds de noms $\mathcal{N}$ définit par point fixe :

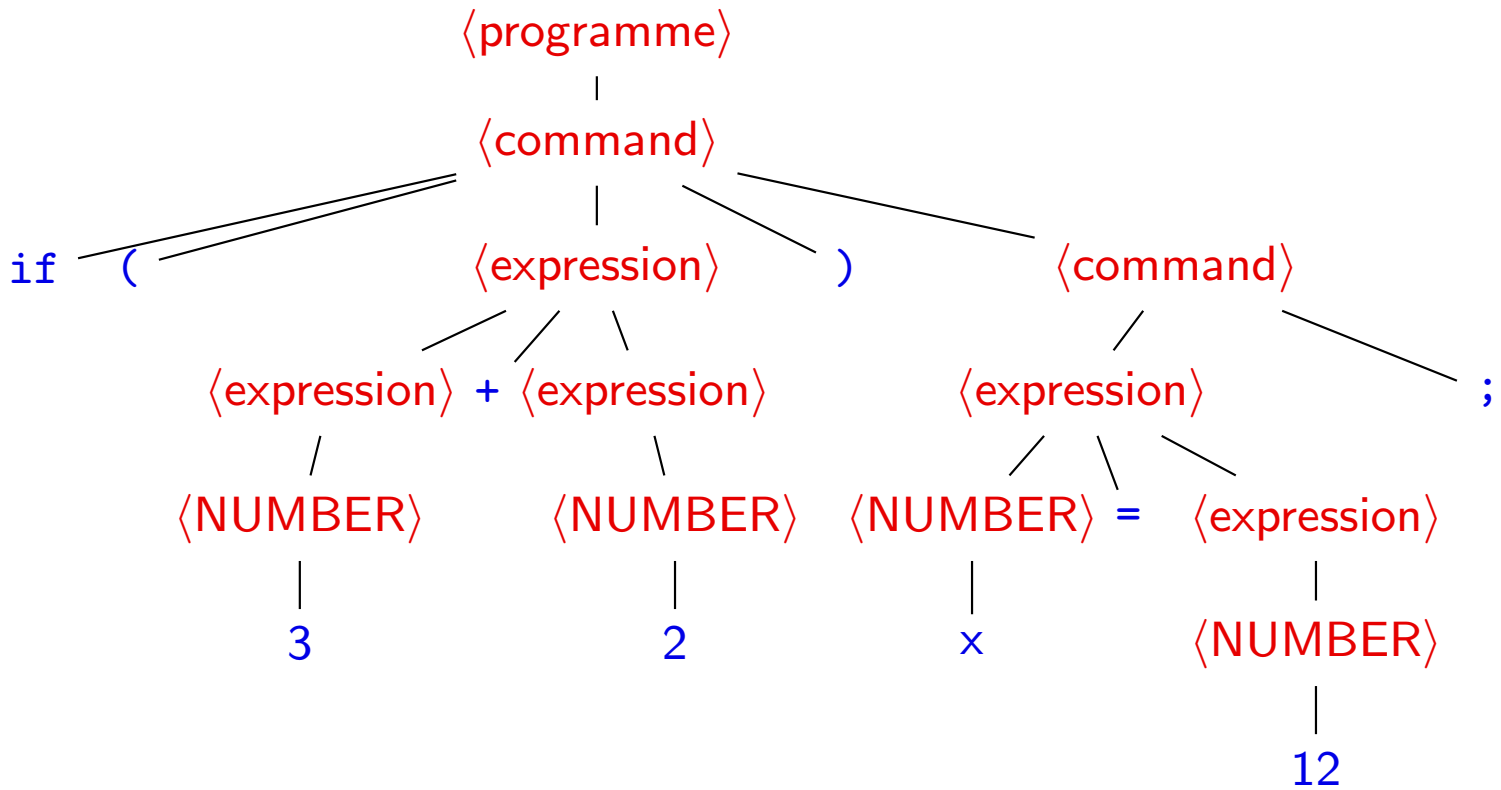
- Ensemble : $\mathcal{N} \subseteq \mathcal{N} \times (\mathcal{T} \uplus \mathcal{N})^*$,
- Initialisation : $\mathcal{N} = \emptyset$,
- Pas :

$$\mathcal{N} := \bigcup_{N \mapsto \underline{c_1} \dots \underline{c_n}} \{(N, \underline{c_1} \dots \underline{c_n}) \mid \forall i,\ \underline{c_i} \in \mathcal{T} \quad \text{et} \quad \underline{c_i} \in \mathcal{N}\}$$

On appelle arbre syntactique de $\mathcal{G}$ un arbre syntactique partant de de S.

Mot reconnu par un arbre syntactique

C'est le mot lu sur les feuilles de gauche à droite.



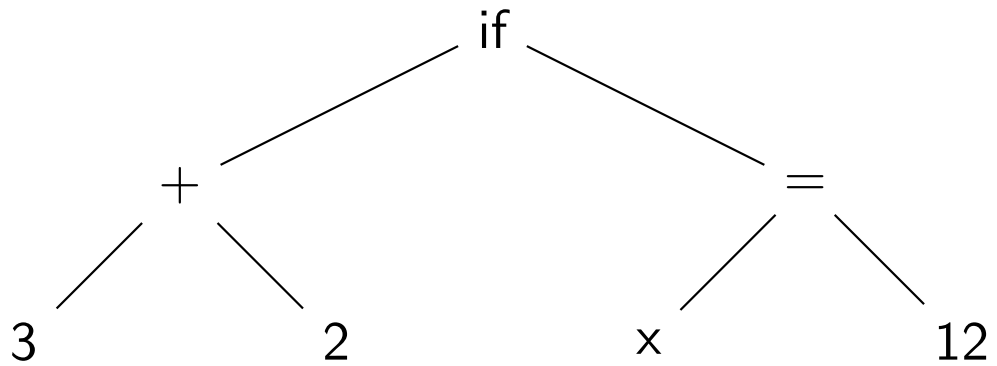
Reconnait :

```
if ( 3 + 2) x = 12 ;
```

Arbre syntaxique abstrait (AST)

C'est une simplification arbitraire de l'arbre syntaxique qui enlève les noeuds superflu

- suppression des parenthèses,
- remplacer les nom de non-terminaux par le nom de la règle utilisée
- suppression des tokens sans contenus.
- (plus en cours de compil)

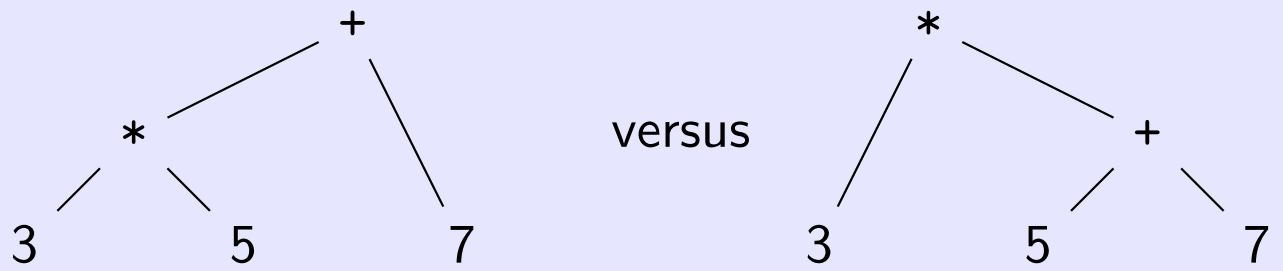


À partir de maintenant : on ne considèrera que des ASTs

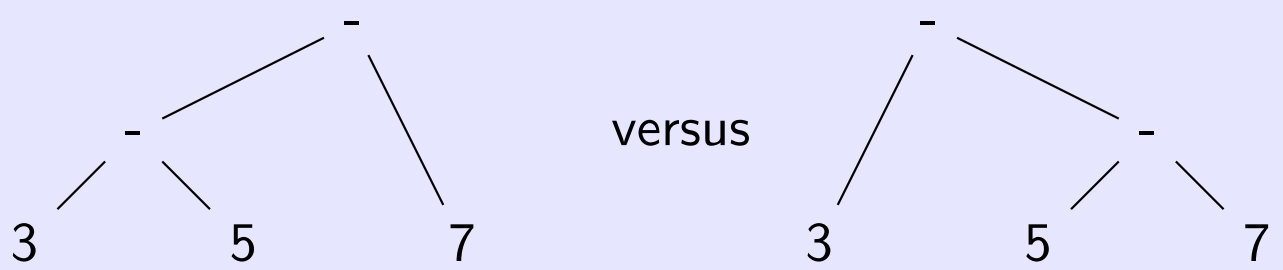
Les problèmes d'ambiguïté (AST)

Ambiguïté : 2 arbres pour un même terme

Priorité



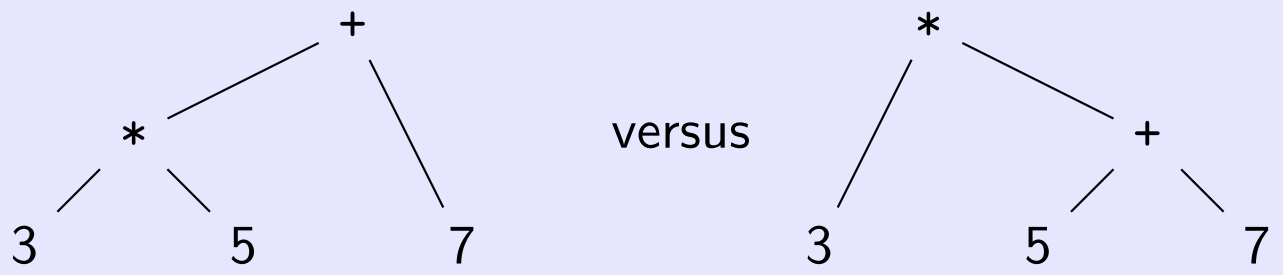
Associativité



Attention, ont peut être ambiguë d'autres manières...

Régler les problèmes de priorité

Priorité



$*$ prioritaire sur $+$ | $+$ prioritaire sur $*$

Solution 1

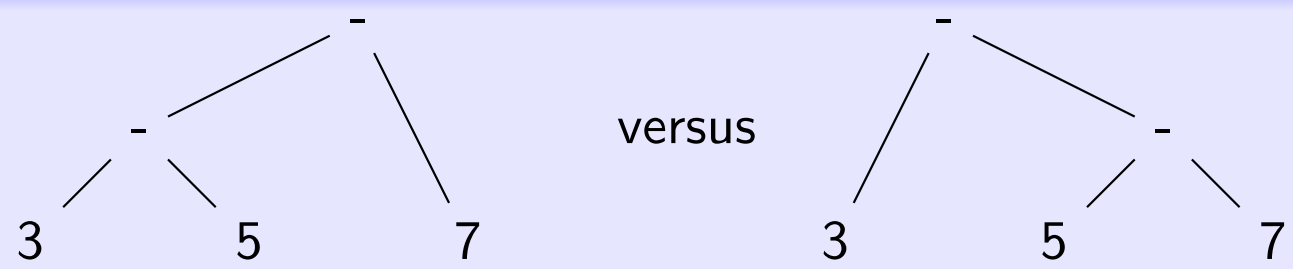
Changer la grammaire, avec un nouveau non-terminal par niveau de priorité (voire exo 1 du TP).

Solution 2

Utiliser un générateur de parseur dans lequel on peut indiquer la priorité (voir exo 3 du TP1 OCaml ou C en compil)

Régler les problèmes d'associativité

Associativité



(-) associatif à gauche

(-) associatif à droite

Solution 1

Changer la grammaire, avec un non-terminal intermédiaire (cf TP1 exo1)

Solution 2

Utiliser un générateur de parseur dans lequel on peut indiquer l'associativité gauche ou droite (voir exo 3 du TP1 OCaml ou C).

Quelques remarques

Même si l'opération est associative, le compilo ne le sait pas

Il faut bien construire un arbre...
Associativité gauche par défaut pour les générateurs LL (JavaCC).

Dans le projet de compil, le + et le * ne sont pas associatifs...

On peut faire pareil avec des opérateurs d'autres arités

`!b==c`
$\longrightarrow$! prioritaire sur ==

`if b then if c then ; else ;`
$\longrightarrow$ `if_then_else` prioritaire sur `if_then`

`true?false:true?true:true`
$\longrightarrow$ `_?_:_` associatif à droite

En FdIP : On précisera la première fois que l'on voit l'opérateur si ce n'est pas évident...

Lieur de variable et Abstract binding trees (ABT)

Au tableau