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*,
- *N7* : en pointillés dans *Fondements de la prog*.

Un exemple de grammaire (projet)

⟨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, IDENT, IF, STRING...

pour plus de lisibilité on utilise souvent les notations d'origines pour les singletons (mots-clefs, symboles), et des majuscules entre brackets ⟨VAR⟩ pour les autres.

Les Non-Terminaux

Ce sont les grandes classes structurant une phrase

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

pour plus de lisibilité on utilise des noms en minuscule entre brackets ⟨expression⟩

Forme alt. : séquence de règles

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

Forme alt. : séquence de règles

⟨programme⟩ ↦ ⟨commande⟩
⟨programme⟩ ↦ ⟨commande⟩ ⟨programme⟩
⟨commande⟩ ↦ `if (` ⟨expression⟩ `)` ⟨commande⟩ `else` ⟨commande⟩
⟨commande⟩ ↦ ⟨expression⟩`;`
⟨commande⟩ ↦ `while (` ⟨expression⟩ `)` ⟨commande⟩
⟨commande⟩ ↦ `for(`⟨expression⟩`;`⟨expression⟩`;`⟨expression⟩`)`⟨commande⟩
⟨commande⟩ ↦ `;`
⟨commande⟩ ↦ `{`⟨programme⟩`}`
⟨expression⟩ ↦
⟨expression⟩
⟨expression⟩
⟨expression⟩
⟨expression⟩ ↦
⟨expression⟩ ↦ ⟨expression⟩ `-` ⟨expression⟩
⟨expression⟩ ↦ ⟨expression⟩ `*` ⟨expression⟩
⟨expression⟩ ↦ ⟨expression⟩ `/` ⟨expression⟩
⟨expression⟩ ↦ ⟨expression⟩ `==` ⟨expression⟩
⟨expression⟩ ↦ `-` ⟨expression⟩
⟨expression⟩ ↦ ⟨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 $\mathcal{T}$.

Exemple très méta : la grammaire des expressions régulières

au tableau

Version plus compacte

On associe une regexp aux non terminaux

À chaque non terminal $N \in \mathcal{N}$ on associe une regexp :

$$\tau : \mathcal{N} \to \texttt{RegExp}(\mathcal{T} \uplus \mathcal{N})$$

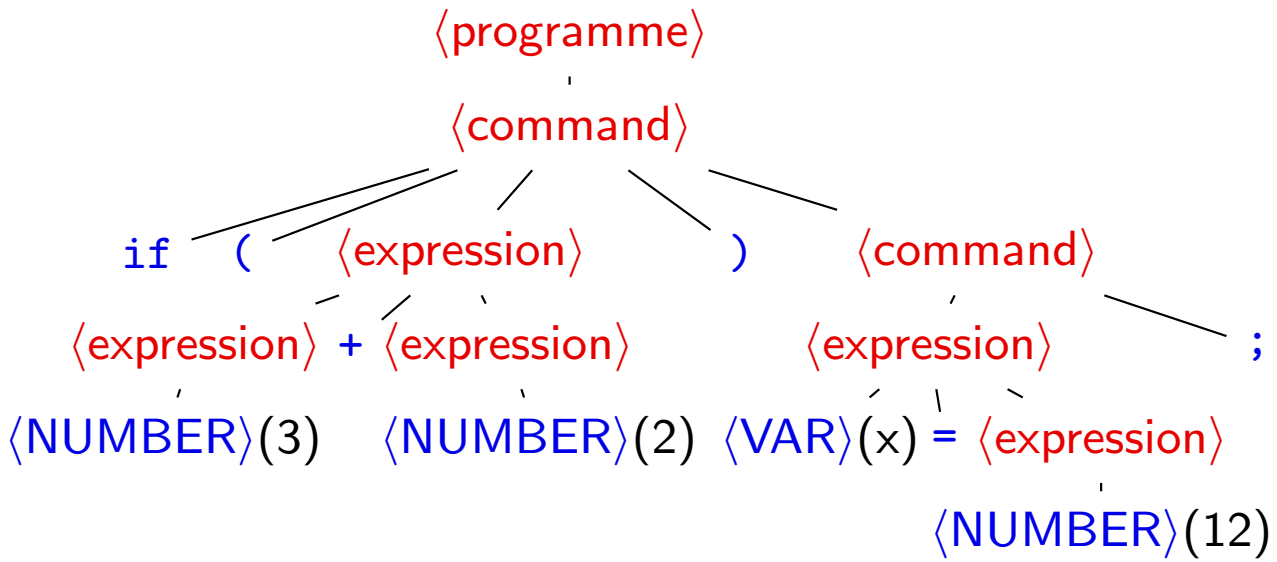
En faisant ça, on peut écrire des règles plus compactes :

$$\begin{array}{lll} \langle \text{commande} \rangle & ::= & \mid \texttt{function}\ \langle \text{IDENT} \rangle\ (\langle \text{arguments} \rangle)\ \{\ \langle \text{command} \rangle^*\} \\ & & \mid \texttt{Class}\ \langle \text{IDENT} \rangle\ (\texttt{extends}\ \langle \text{expression} \rangle)^?\ \langle \texttt{methode} \rangle^* \end{array}$$

Utilisé par JavaCC ou Antler par exemple

Avantage : Plus facile à écrire.
Défaut : Priorités/associativités plus difficiles à définir.

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 :

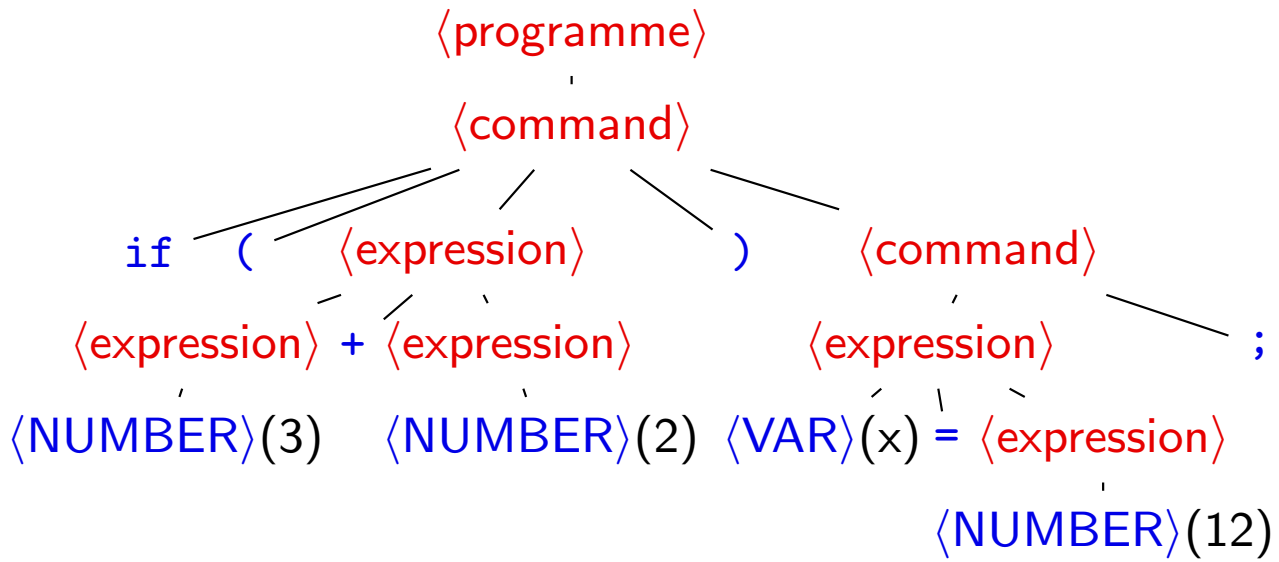
- Ensemble : $\mathcal{N} \subseteq \mathcal{N} \times (\mathcal{T} \uplus \mathcal{N})^*$,
- Initialisation : $\mathcal{N} = \emptyset$,
- Pas :

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

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

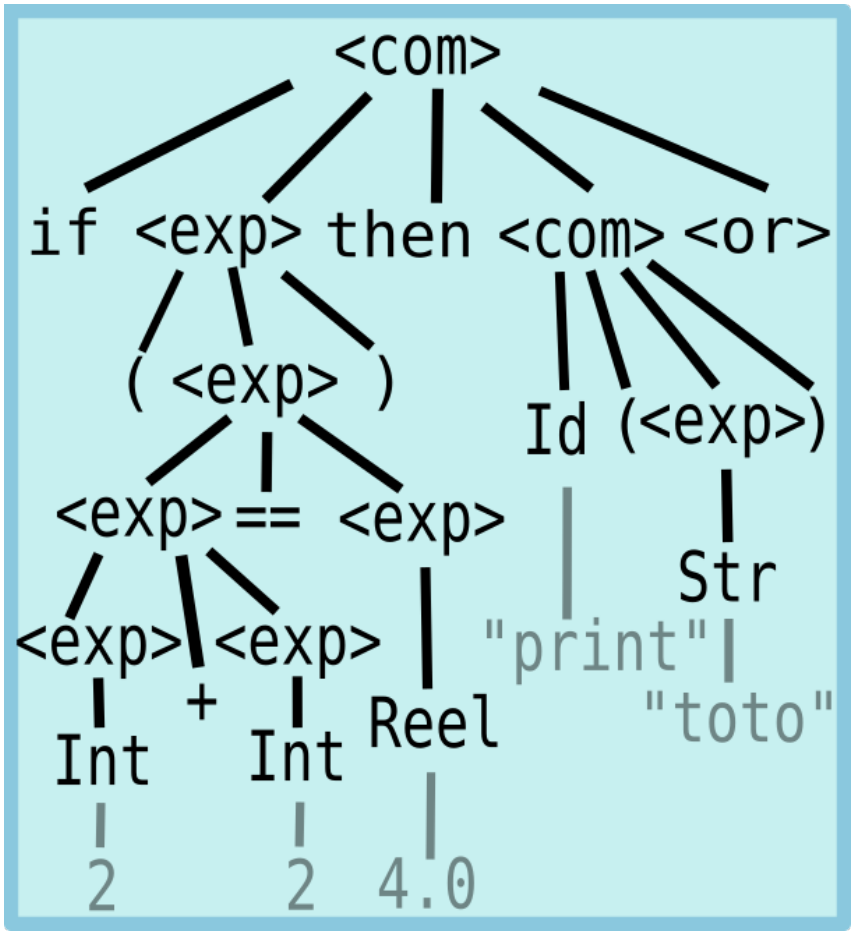
Mot reconnu par un arbre syntactique

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



Reconnait :
`if` `(` ⟨NUMBER⟩(3) ⟨NUMBER⟩(2) `)` ⟨VAR⟩(x) ⟨=⟩ ⟨NUMBER⟩(12) `;`

Mot reconnu par un arbre syntactique

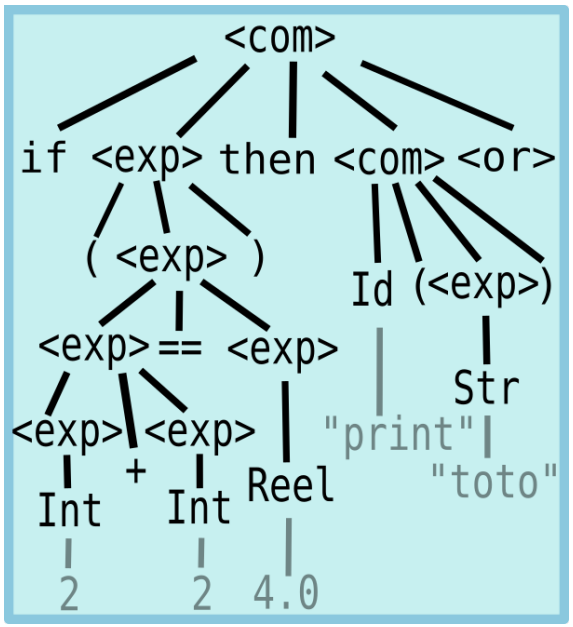


Reconnait :
IF, INT(2), +, INT(2), ==, REEL(4.),THEN, ID(“print”), ’(’, STR(“toto”)

Arbre syntaxique abstrait (AST)

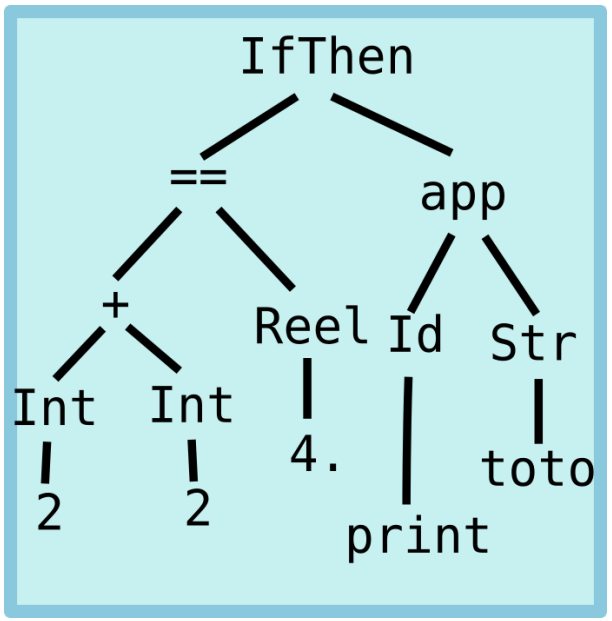
C'est une simplification arbitraire de l'arbre syntaxique qui enlève le superflu :

- suppression des parenthèses,
- remplacer les nom de non-terminaux par le nom de la règle utilisée
- suppression des tokens sans contenus.



Arbre Syntaxique

Devient



AST

Rappel : Analyse Syntaxique / Parsing

Stream de Tokens

```
IF,ParG,
Int(2),Plus,
Int(2),Eg,
Reel(4.0),
ParD,THEN,
Id(print),
Str(toto)
```

Analyse Syntaxique

Construction de l'arbre syntaxique

Deux types de parseurs

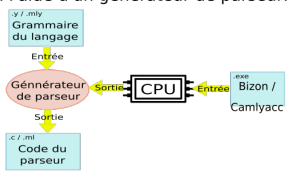
LL:
"devine" le noeud que l'on va lire

LR :
"devine" le noeud que l'on a lu

Simplification / abstraction

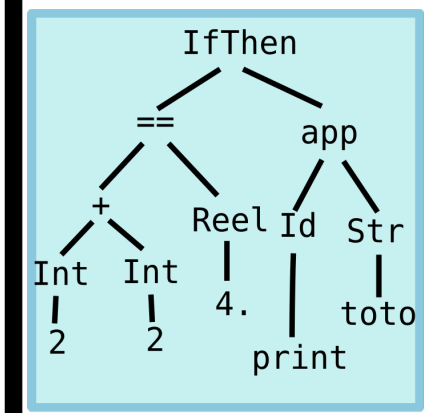
Comment faire un parseur ?

- à la main...
- à l'aide d'un générateur de parseur:



Parseur

Arbre syntaxique abstrait (AST)



De la théorie (TD) à la pratique (TP)

TD : pas d'AST

En TD, on se concentre sur la partie complexe; le décodage de le ST.
On considère donc que l'arbre généré est le ST pour simplifier l'algo.

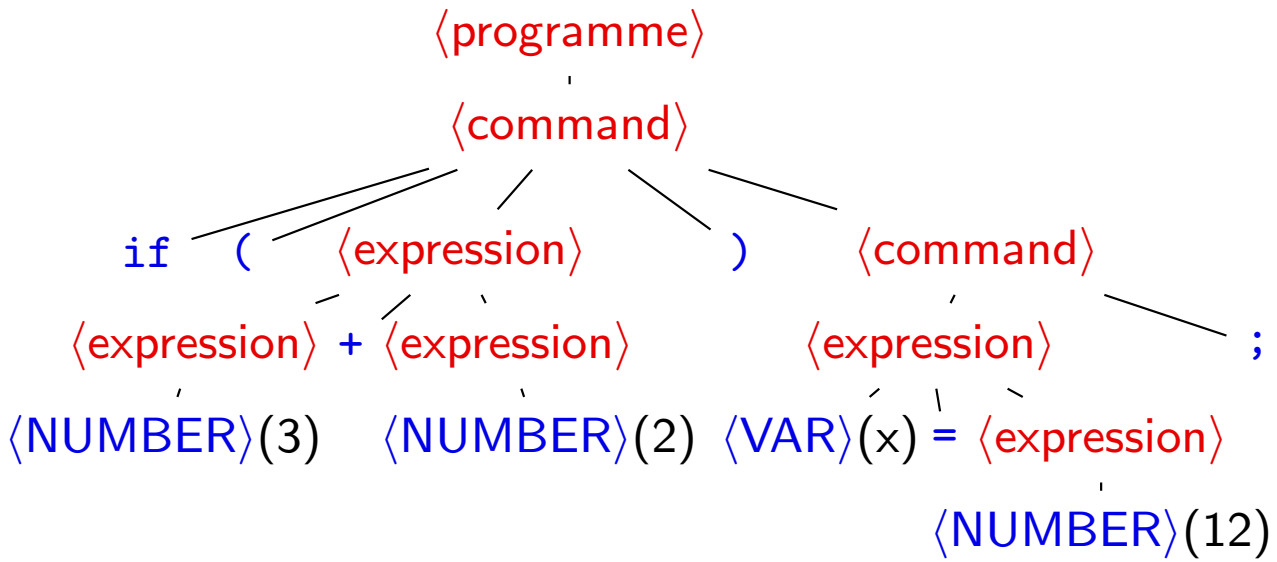
TP : Abstraction à la volée

En pratique, le parseur décode bien l'arbre syntaxique mais crée directement l'AST. (Les actions de l'automate sont plus complexe.)

Nous nous concentrons ici sur la théorie.

Mot reconnu par un arbre syntactique

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



Reconnait :
`if` `(` ⟨NUMBER⟩(3) ⟨NUMBER⟩(2) `)` ⟨VAR⟩(x) ⟨=⟩ ⟨NUMBER⟩(12) `;`

Le but d'un parseur

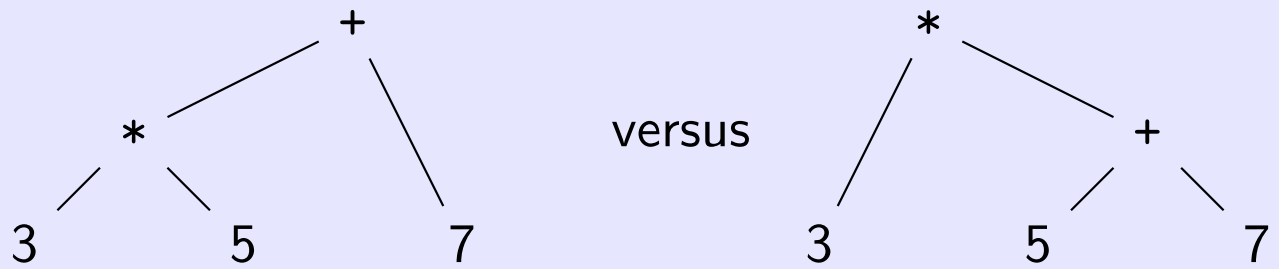
Trouver l'arbre qui reconnaît le mot en entrée

Problème : Ambigüité

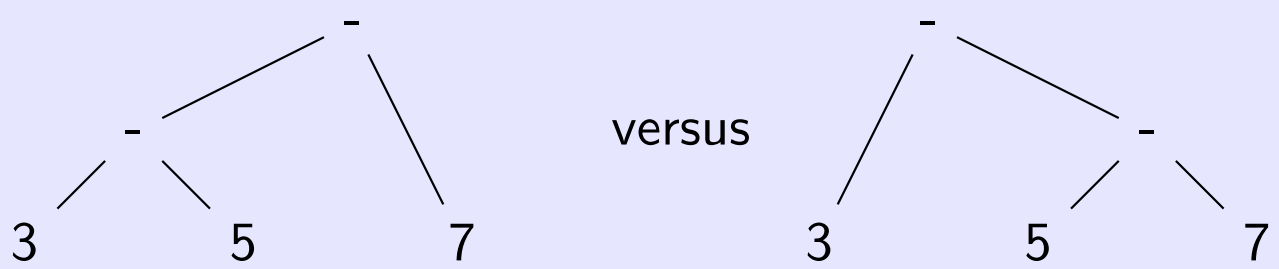
Si plusieurs arbres le reconnaissent, que fait-on ?

Les cas d'ambigüité classiques (AST)

Priorité

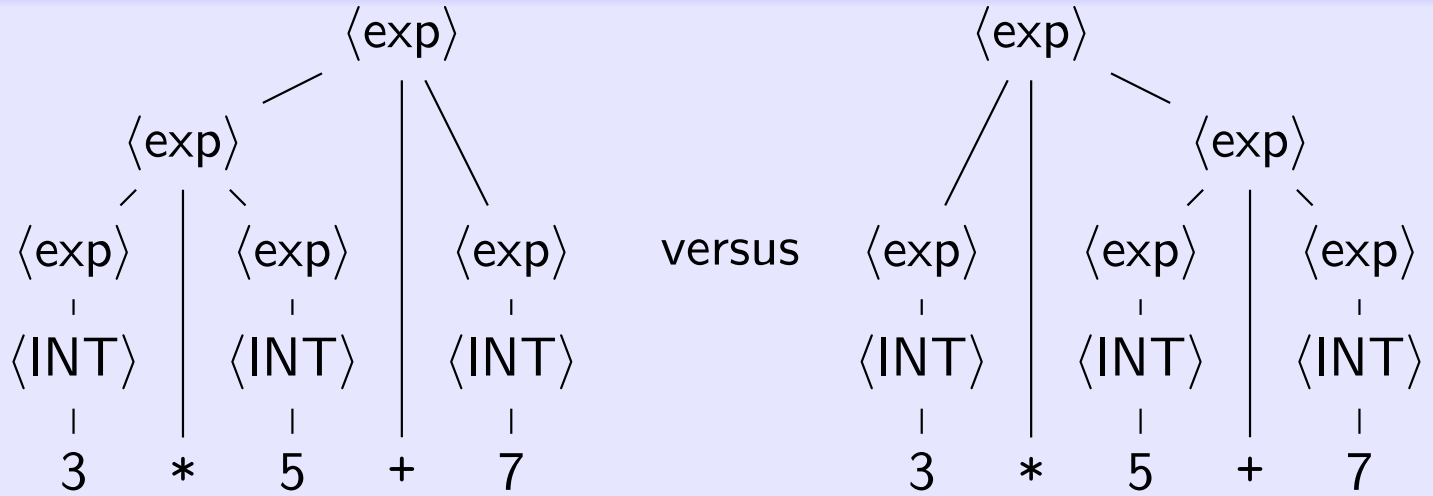


Associativité

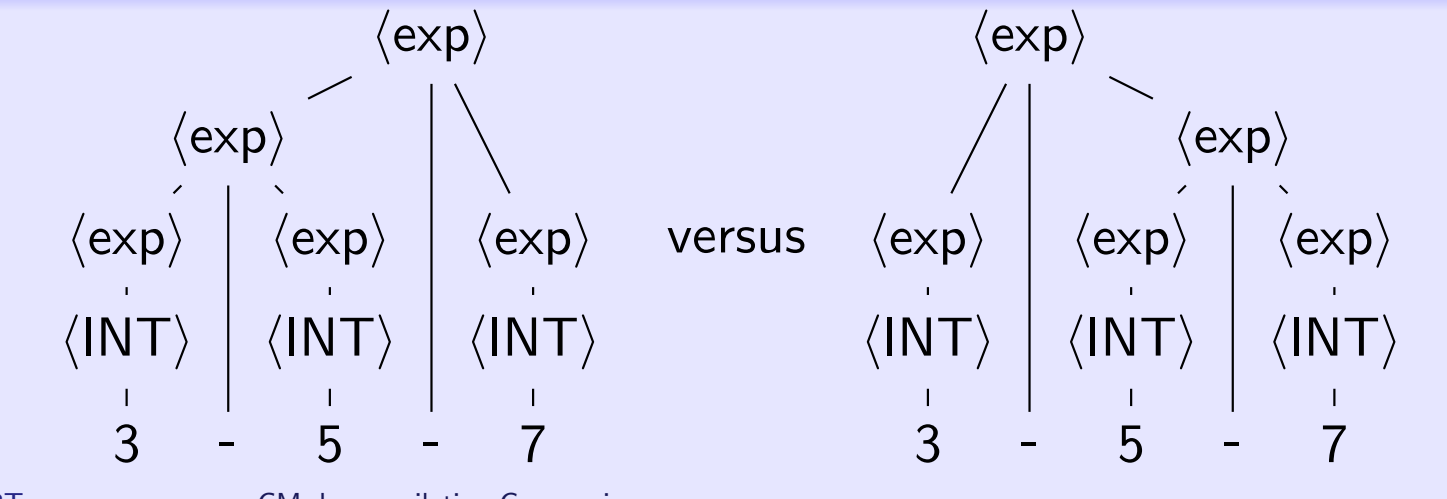


Les cas d'ambigüité classiques (ST)

Priorité



Associativité



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).

Attention, dans le projet, le + et le * ne sont pas associatifs.

On peut faire pareil avec des opérateurs d'autres arités

```
!b&&c,
if b then if c then ; else ;
a?b:c?e:f
```

Régler les problèmes de priorité

Priorité

∗ prioritaire sur +

\+ prioritaire sur ∗

Solution 1

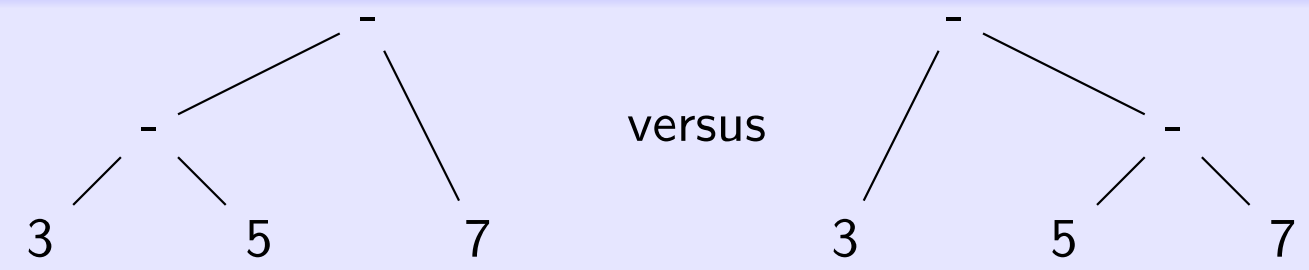
Changer la grammaire, avec un nouveau non-terminal par niveau de priorité (voire exo 1 du TP).

Solution 2

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

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).

Solution 3

Utiliser un générateur de parseur avec grammaire compacte (TP1 java)

Exemple de parseur (OCamlyacc)

```
%token NOMBRE PLUS MOINS FOIS DIV GPAREN DPAREN EOL
%type <unit> main expression terme facteur
%start main
%%
main:
    expression EOL                  {}              ;
expression:
    expression PLUS terme           {}
  | expression MOINS terme          {}
  | terme                           {}              ;
terme:
    terme FOIS facteur              {}
  | facteur                         {}              ;
facteur:
    GPAREN expression DPAREN        {}
  | MOINS facteur                   {}
  | NOMBRE                          {}              ;
```

Exemple de parseur (OCamlyacc)

```
%token NOMBRE PLUS MOINS FOIS GPAREN DPAREN EOL
%left PLUS MOINS
%left FOIS DIV
%nonassoc UMOINS
%type <unit> main expression
%start main
%%
main:
   expression EOL                    {}                    ;
expression:
   expression PLUS expression        {}
 | expression MOINS expression       {}
 | expression FOIS expression        {}
 | GPAREN expression DPAREN          {}
 | MOINS expression %prec UMOINS     {}
 | NOMBRE                            {}                    ;
```

Exemple de parseur (OCamlyacc)

```
%token <int> NOMBRE
%token PLUS MOINS FOIS GPAREN DPAREN EOL
%left PLUS MOINS
%left FOIS DIV
%nonassoc UMOINS
%type <AST.expression_a> main expression
%start main
%%
main:
   expression EOL                  { $1 }              ;
expression:
   expression PLUS expression      { Plus (($1,$3) }
 | expression MOINS expression     { Moins($1,$3) }
 | expression FOIS expression      { Mult (($1,$3) }
 | GPAREN expression DPAREN        { $2 }
 | MOINS expression %prec UMOINS   { Neg $2 }
 | NOMBRE                          { Num $1 }          ;
```

Exemple de parseur (Bison)

```
%parse-param {struct ExpressionA* ast}
%union { struct ExpressionA* expA; int num; }
%type <expA> expression
%token <num> NOMBRE
%left '+' '-'
%left '*' '/'
%nonassoc MU
%%
resultat:    expression          {*ast = *$1;}
expression:
  expression '+' expression {$$=newExpression('+',$1,$3,0);}
| expression '-' expression {$$=newExpression('-',$1,$3,0);}
| expression '*' expression {$$=newExpression('*',$1,$3,0);}
| expression '/' expression {$$=newExpression('/',$1,$3,0);}
| '(' expression ')'        {$$=$2;}
| '-' expression %prec MU   {$$=newExpression('-',$2,NULL,0);}
| NOMBRE                    {$$=newExpression('0',NULL,NULL,$1);
%%
```

Exemple de parseur (JavaCC)

```
SKIP : { " " | "\t" }
TOKEN : { < NOMBRE: ["1"-"9"] (["0"-"9"])* > | < EOL: "\n" > }

AST.ExpressionA mainNT () :
{ AST.ExpressionA e; }
{
    e=expression() <EOL>          {return e;}
}
AST.ExpressionA expression () :
{ AST.ExpressionA exp;
  AST.ExpressionA e;
}
{
    e=terme()                     {exp = e;}
    ( "+" (e=terme())             {exp = new AST.Plus(exp,e);}
    | "-" (e=terme())             {exp = new AST.Moins(exp,e);}
    )*                            {return exp;}
}
```

Suite

```
AST.ExpressionA terme () :
{ AST.ExpressionA exp;
  AST.ExpressionA e;
}
{
    e=facteur()                    {exp = e;}
    ( "*" (e=facteur())            {exp = new AST.Mult(exp,e);}
    | "/" (e=facteur())            {exp = new AST.Div(exp,e);}
    )*                             {return exp;}
}
AST.ExpressionA facteur () :
{ AST.ExpressionA e;
  Token t;
}
{
    "(" (e=expression()) ")"       {return e;}
  | "-" (e=facteur())              {return new AST.Neg(e);}
  | t=<NOMBRE>                     {return new AST.Num(Integer.parse
}
```