Grammaire du projet (Fragment 1)

```
<programme>   ::=  <commande> | <commande> <programme>

<commande>    ::=  ;
                 | { <programme> }
                 | <expression> ;
                 | Si ( <expression> ) <commande>
                 | Si ( <expression> ) <commande> Sinon <commande>
                 | TantQue ( <expression> ) <commande>
                 | Faire <commande> TantQue ( <expression> )
                 | Pour ( <expression> ; <expression> ; <expression> ) <comma
                 | ecrire( <expression> ) ;

<expression>  ::=  <NOMBRE> | <BOOLEEN> | <IDENT>
                 | (<expression>)
                 | <op_unair> <expression>
                 | <expression> <op_binair> <expression>
                 | <expression> ? <expression> : <expression>
                 | <IDENT> = <expression>

<op_unair>    ::=  - | ! | ++ | -- | Typeof

<op_binair>   ::=  + | - | * | / | % | === | > | !== | && | ||
```

Grammaire du projet (Fragment 1)

<programme> ::= <commande> | <commande> <programme>

<commande> ::=
| Si (<expression>) <commande>
| Si (<expression>) <commande> Sinon <commande>
| TantQue (<expression>) <commande>
| Faire <commande> TantQue (<expression>)
| Pour (<expression> ; <expression> ; <expression>) <commande>
| ecrire(<expression>) ;

<expression> ::= <NOMBRE> | <BOOLEEN> | <IDENT>
| (<expression>)
| <op_unair> <expression>
| <expression> <op_binair> <expression>
| <expression> ? <expression> : <expression>
| <IDENT> = <expression>

<op_unair> ::= - | ! | ++ | -- | Typeof

<op_binair> ::= + | - | * | / | % | === | > | !== | && | ||

Non-Terminaux

ce sont les structures que l'on décrit

Terminaux

Ce sont les structures données en entrée.

Plusieurs types de terminaux

```
<programme>    ::=   <commande> | <commande> <programme>

<commande>     ::=   ;
                   | { <programme> }
                   | <expression> ;
                   | Si ( <expression> ) <commande>
                   | Si ( <expression> ) <commande> Sinon <commande>
                   | TantQue ( <expression> ) <commande>
                   | Faire <commande> TantQue ( <expression> )
                   | Pour ( <expression> ; <expression> ; <expression> ) <comma
                   | ecrire( <expression> ) ;

<expression>   ::=   <NOMBRE> | <BOOLEEN> | 
                   | (<expression>)
                   | <op_unair> <expression>
                   | <expression> <op_binair> <exp
                   | <expression> ? <expression> : <expression>
                   | <IDENT> = <expression>

<op_unair>     ::=   - | ! | ++ | -- | Typeof

<op_binair>    ::=   + | - | * | / | % | === | > | !== | && | ||
```

Token

Boite étiquetée
(fournie par le lexeur),
on ne regarde que son nom

Plusieurs types de terminaux

<programme> ::= <commande> | <commande> <programme>

<commande> ::= ;
| { <programme> }
| <expression> ;
| Si (<expression>) <commande>
| Si (<expression>) <commande> Sinon <commande>
| TantQue (<expression>) <commande>
| Faire <commande> TantQue (<expression>)
| Pour (<expression> ; <expression> ; <expression>) <comma
| ecrire(<expression>) ;

<expression> ::= <NOMBRE> | <BOOLEEN> |
| (<expression>)
| <op_unair> <expression>
| <expression> <op_binair> <exp
on> ? <expression> : <expression>
> = <expression>

<op_una | Typeof

<op_binair> ::= + | - | * | / | % | === | > | !== | && | ||

Mot clef

Correspond à un token vide

Symbole

Correspond à un token vide

Token

Boite étiquetée (fournie par le lexeur), on ne regarde que son nom

Symbole complexe

Correspond à UN token vide

Règles d'une grammaire

Notation informatique

```
<expression>  ::=  <NOMBRE>
                 | (<expression>)
                 | <expression> + <expression>
```

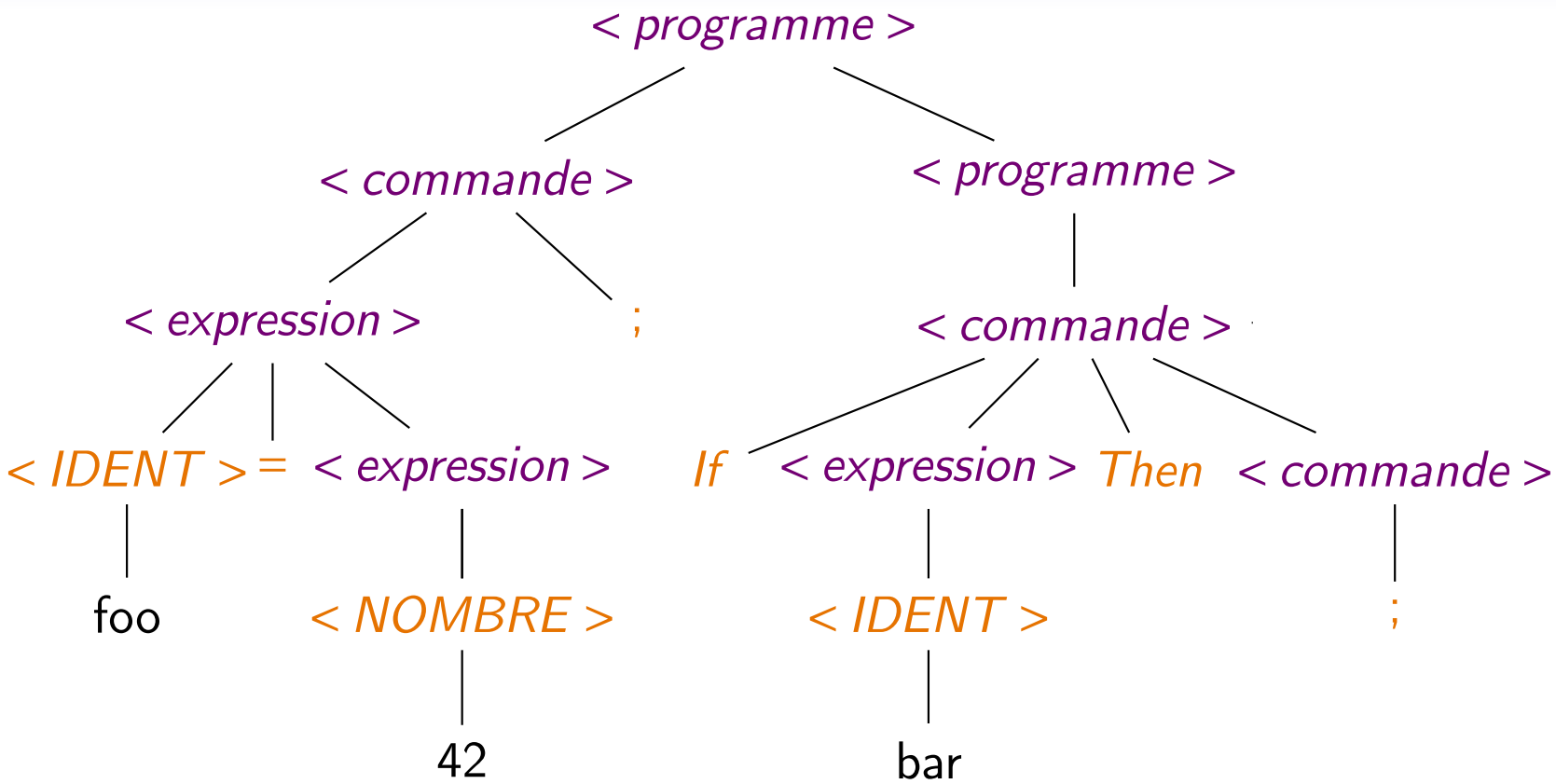
Notation mathématique

$$
\begin{aligned}
\langle \text{expression} \rangle &\mapsto \langle \text{NOMBRE} \rangle \\
\langle \text{expression} \rangle &\mapsto (\langle \text{expression} \rangle) \\
\langle \text{expression} \rangle &\mapsto \langle \text{expression} \rangle + \langle \text{expression} \rangle
\end{aligned}
$$

Ici, on signifie qu'une expression est

- ou bien un nombre,
- ou bien de la forme "(w)" où w est une expression,
- ou bien de la forme "$w_1 + w_2$" où w_1 et w_2 sont des expressions.

Arbres syntaxiques

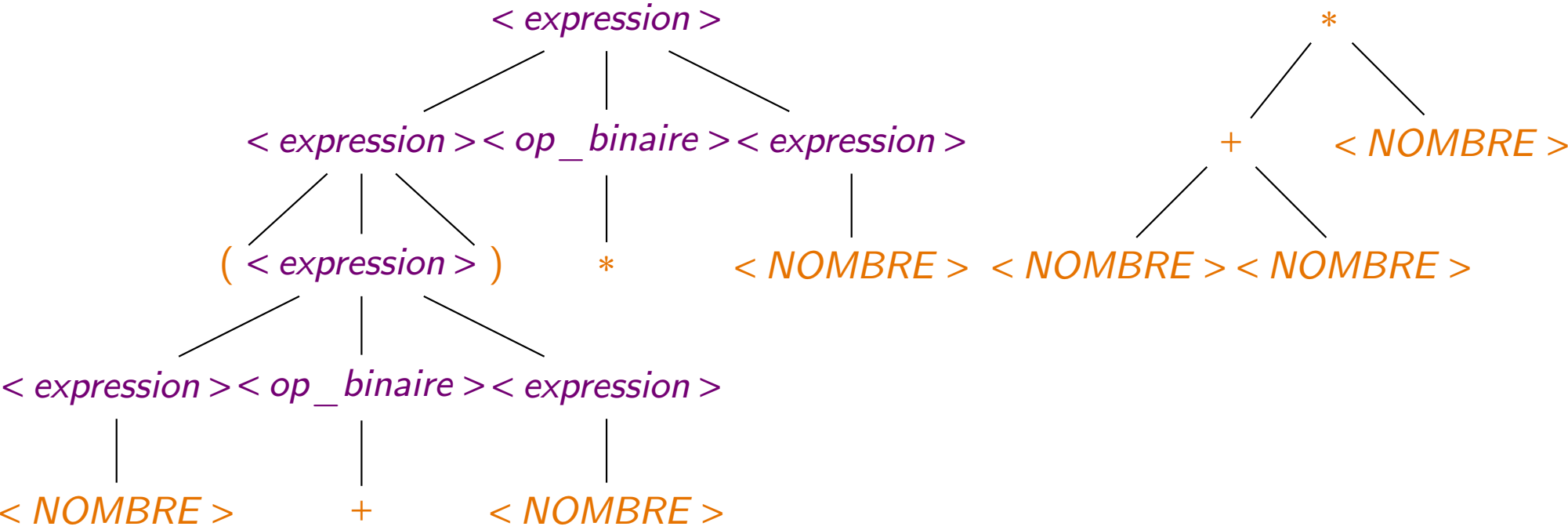


Une grammaire permet de définir un ensemble d'arbres syntaxiques dont les noeuds sont des non terminaux et les feuilles des terminaux.

Attention le contenu des tokens ne fait pas vraiment partie de l'arbre syntaxique, ils sont gérés en amont.

Arbres syntaxiques abstrait

L'arbre syntaxique est souvent lourd et plein de redondance, on va généralement définir une opération de simplification :



Il n'y a pas de définition uniforme d'abstraction d'un arbre, cela dépend de ce que vous en faites.

Langage d'une grammaire

Langage reconnu par une grammaire

Le mot formé par les feuilles d'un arbre syntaxique est appellé le réduit de l'arbre.

Langage reconnu par une grammaire

Le langage reconnu par une grammaire est le langage des réduits de ses arbres syntaxiques.

Par exemple,

```
foo = 42 ; If bar Then ;
```

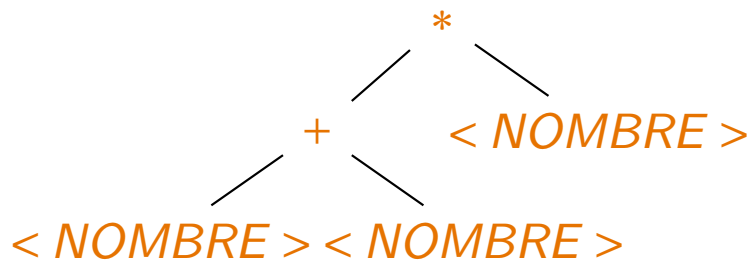
est reconnu

Analyse syntaxique

Bute de Parseur

Construire l'arbre sémantique à partir d'un mot de tokens reconnu par la grammaire.

ex : [(,<NOMBRE>,+,<NOMBRE>,),*,<NOMBRE>] devient



Problème 1

Que se passe-t-il s'il y a plusieurs arbres pour un mot ?

ex : 1 + 2 + 3

Problème 2

Comment trouver cet arbre efficacement

Grammaire ambigüe

Definition

Une grammaire est ambigüe s'il existe deux arbres syntaxiques de même réduit.

Analyse syntaxique impossible

Contrairement à l'annalyse lexicale, on ne peut pas fair de choix "par défaut", lorsqu'une grammaire est ambigüe, on refuse de la parser.

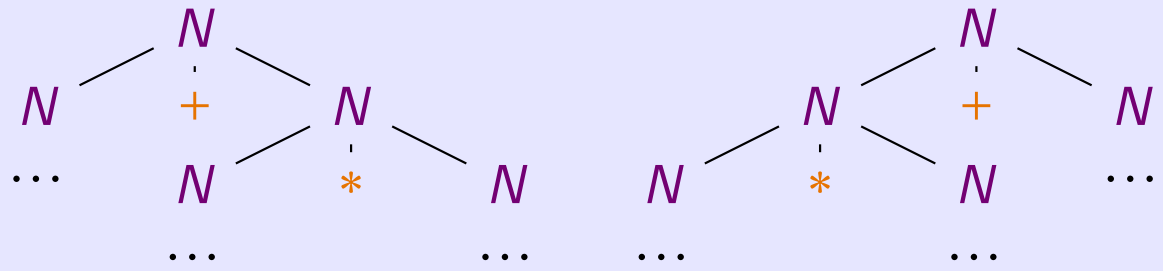
Priorités et associativités

Il y a tout de même un moyen de désambigüiser la grande majorité des cas, ce sont les rêgles de priorité et d'associativité.
mais celles-ci ne suffisent pas toujours

Priorité

Idée

Si une rêgle $N \mapsto N + N$ est prioritaire sur une rêgle $N \mapsto N * N$, alors les mots $N + N * N$ et $N * N + N$ n'ont plus qu'un arbre accepté qui est celui qui met le $*$ en dessous :



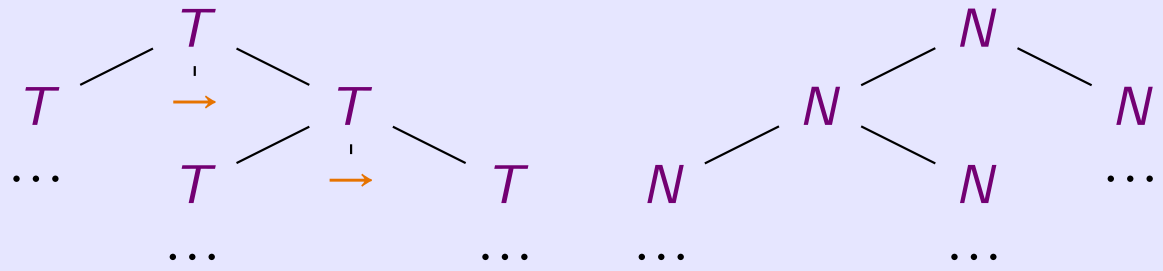
Peut se généraliser

- (<expr> $\mapsto$ $-$<expr>) prioritaire sur (<expr> $\mapsto$ <expr> $-$ <expr>)
- (<com> $\mapsto$ `If <expr> then <prog> else <prog>`) prioritaire sur (<com> $\mapsto$ `If <expr> then <prog>`)

Associativité

Idée

Si $T \mapsto T \to T$ est associative à droite, et $N \mapsto NN$ est associative à gauche , alors les mots $T \to T \to T$ et NNN n'ont plus qu'un arbre accepté qui est celui qui penche à droite pour le premier et à gauche pour le second :



Systématique sur tous les opérateurs infixes

- Pour les opérateurs associatif (ex : l'addition), peu importe le choix entre gauche et droite, mais il faut en faire un.
- On peut imaginer des opérateurs 3,4...-aires infixes avec d'autres types d'associativité, mais ça n'arrive pas en pratique.