



Compilation

Présenté par Yann Caron
skyguide

ENSG Géomatique

Plan du cours

Grammaire

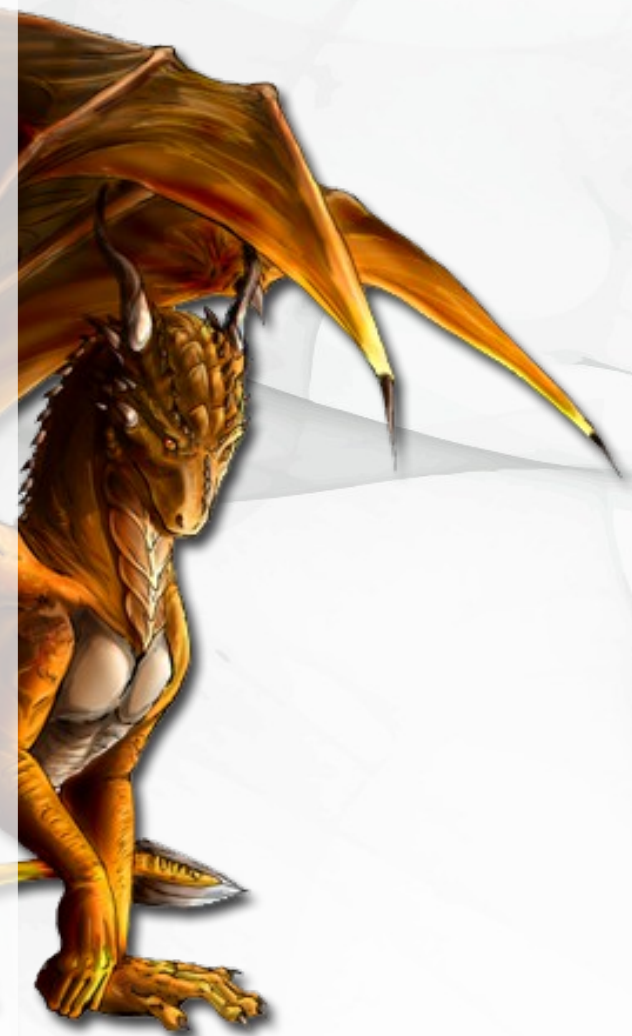
Analyse Lexicale

Algorithme LL(0)

Analyse Syntaxique

Algorithmes LL(k), LL(*) et LR

Arbre Syntaxique (AST)



Vue d'ensemble

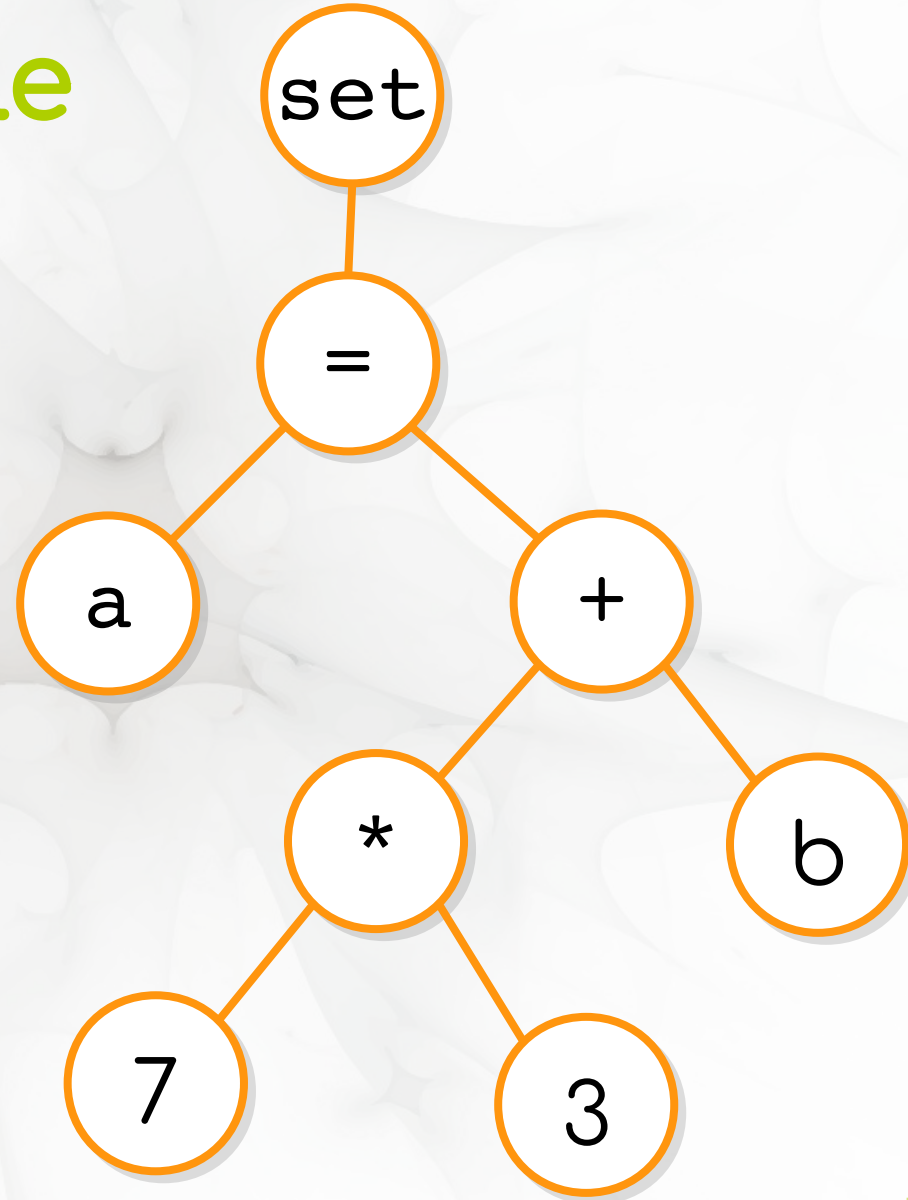
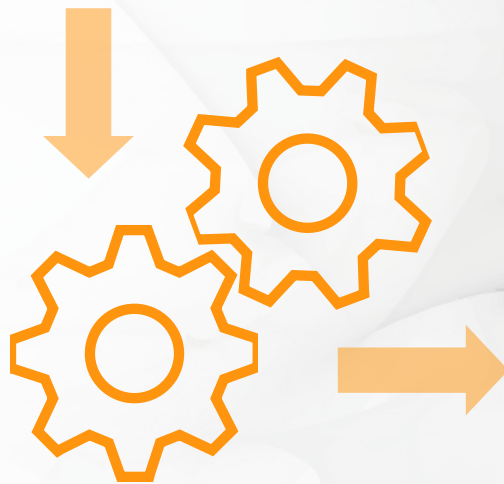
- ✓ La Syntaxe est l'étude de la façon dont les mots sont organisés pour construire des phrases
- ✓ La syntaxe ne s'intéresse pas à la sémantique des mots (les paradigmes)
- ✓ Par contre l'analyse syntaxique peut apporter de l'information pour l'analyse sémantique future

Vue d'ensemble

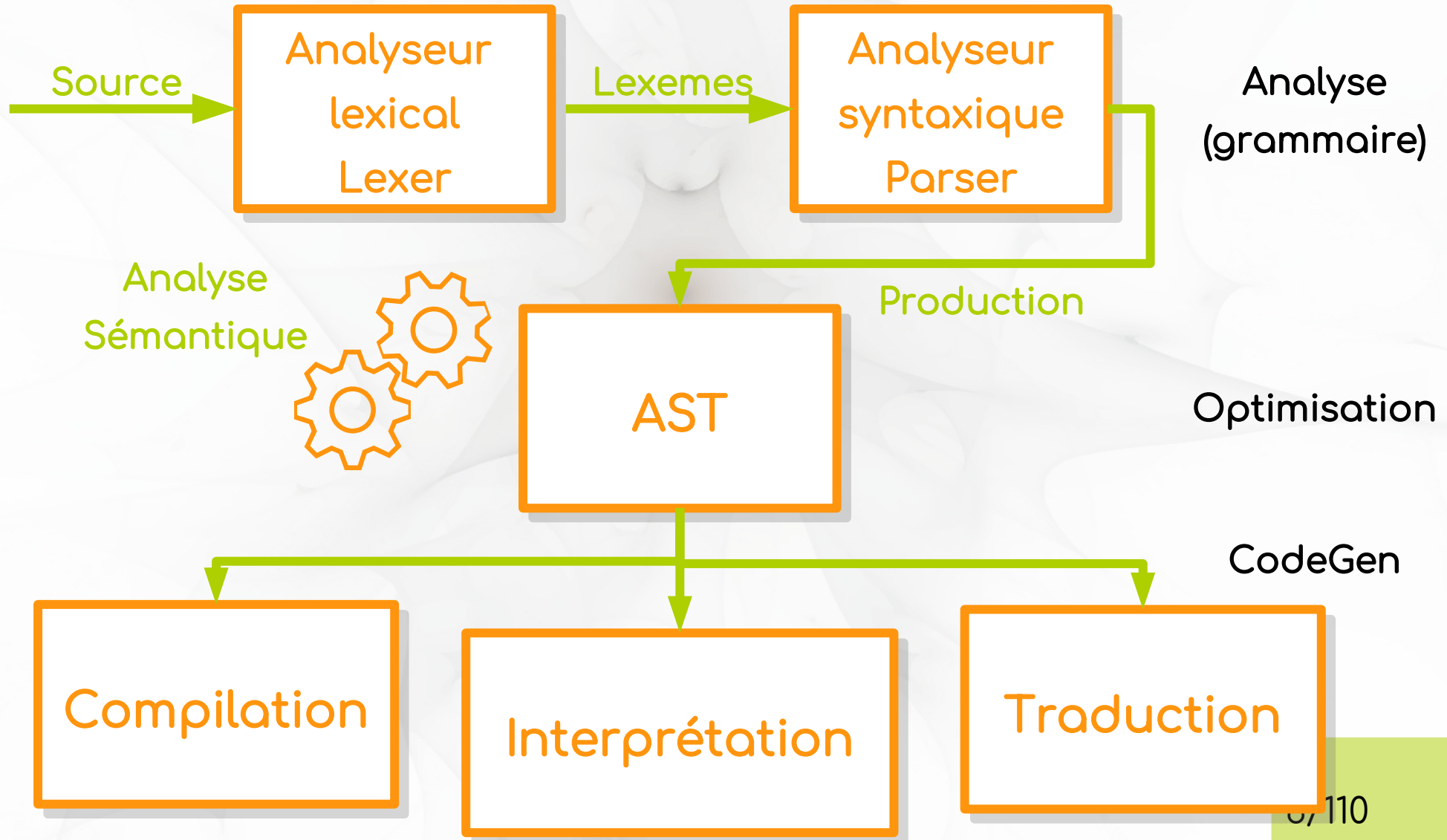
- ✓ Analyseur lexical : Transforme une chaîne de lettres en une chaîne de mots typés appelés Lexems (token)
- ✓ Ex : [87 (number), '+' (symbol), 3 (number)]
- ✓ Analyseur syntaxique: Transforme une chaîne de lexems en un arbre syntaxique (AST)

Vue d'ensemble

s e t a = 7 * 3 + b



Vue d'ensemble

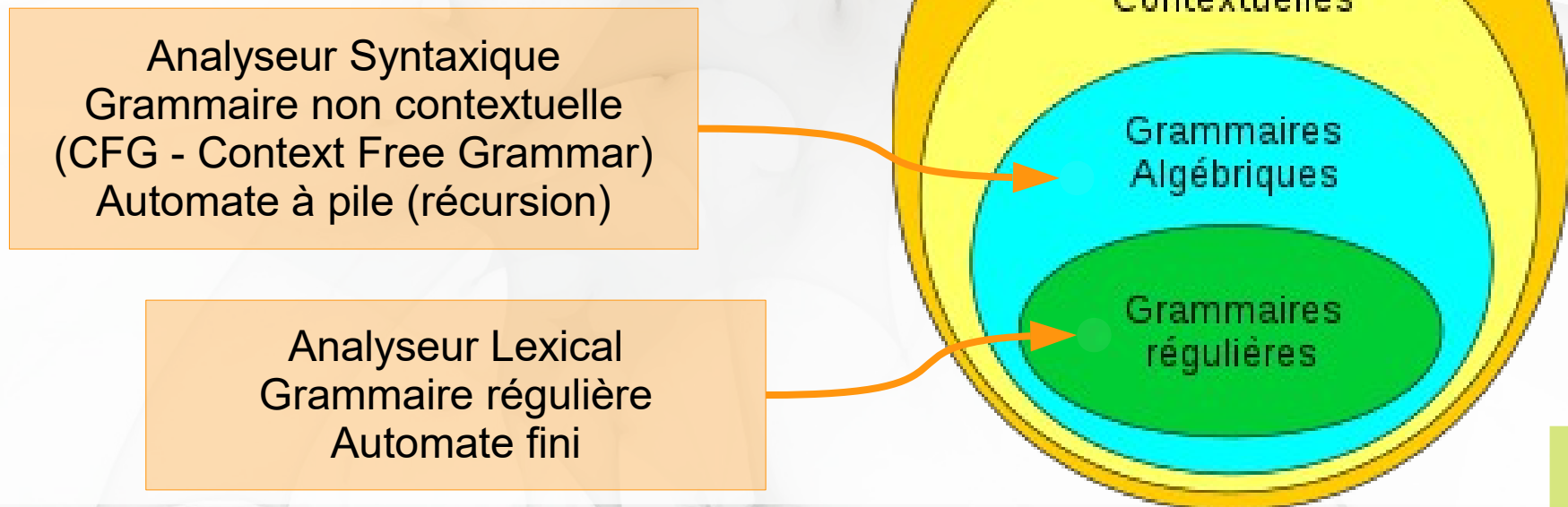


Vue d'ensemble

- ✓ Généralement :
- ✓ L'analyseur lexical utilise une grammaire régulière
 - ✓ Automate fini lit de gauche à droite
- ✓ L'analyseur syntaxique utilise une grammaire non-contextuelle (context free)
 - ✓ Encapsulation (exemple bien parenthésé)

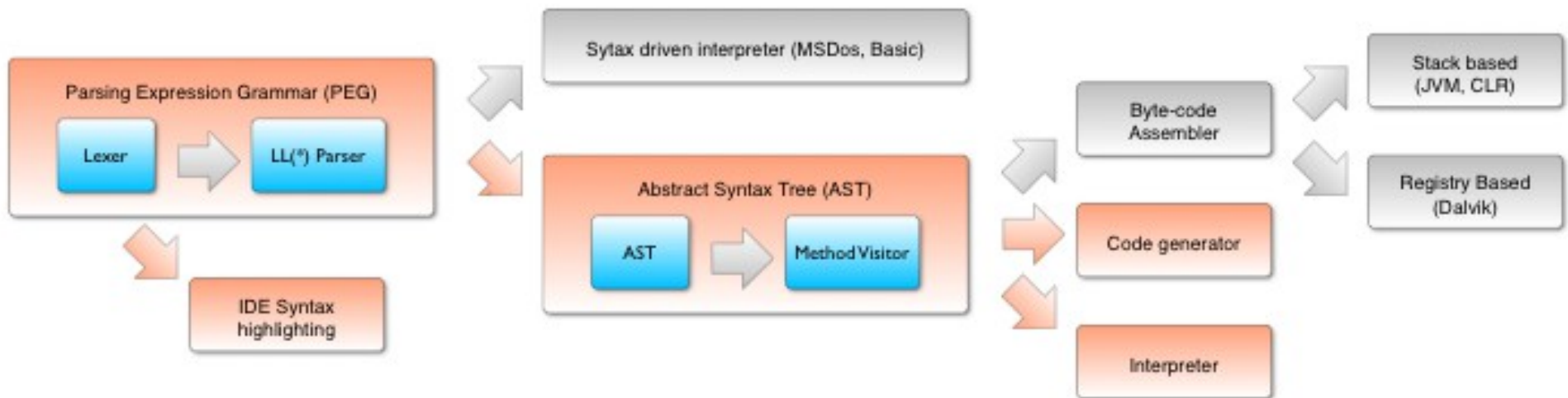
Hiérarchie de Chomsky

- ✓ voir le cours sur la théorie de langages
- ✓ langages formels
- ✓ 4 classes de langages



Vue d'ensemble

- ✓ Choix à plusieurs niveaux



Vue d'ensemble

- ✓ Le résultat de l'analyse lexicale est aussi utilisé pour la coloration syntaxique
- ✓ Pour décrire les grammaires des analyseurs lexicaux et syntaxiques, on utilise la Forme de Backus-Naur

Grammaire

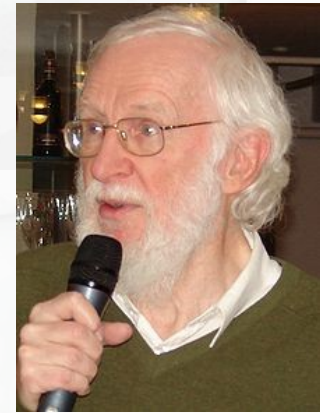


BNF

- ✓ Backus Naur Form
- ✓ Notation permettant de décrire les règles syntaxiques des langages de programmation
- ✓ Défini lors de la création de Algol60
- ✓ Successeur du Fortran originalement créé par John Backus



John Backus
1924 - 2007



Peter Naur
1928 - 2016

EBNF - syntaxe

- ✓ * répétition
- ✓ - absence
- ✓ , concaténation
- ✓ | choix
- ✓ = définition
- ✓ ; terminaison
- ✓ 'a' .. 'z' intervalle
- ✓ ' terminal ambigu '
- ✓ " terminal ambigu "
- ✓ (* commentaire *)
- ✓ (groupe)
- ✓ [groupe optionnel]
- ✓ { groupe répété }
- ✓ ? séquence spéciale ?

EBNF - exemple

```
null = ' ' | '\t' | '\n' | '\r';  
number = ( '0' .. '9' ) { '0' .. '9' };  
symbol = '(' | ')' | '+';  
keyword = ( 'if' | 'goto' );  
  
lexem = keyword | symbol | number |  
null;  
lexemList = (lexem) { lexem }
```

BNF - exercice

- ✓ Quelle est la grammaire du wkt ?
- ✓ POINT (10 20)
- ✓ POINT Z (17 15.5 14.7)
- ✓ MULTIPOINT((3.5 5.6), (4.8 10.5))
- ✓ MULTIPOINT Z((3.5 5.6 4.7), (4.8 10.5 7))

EBNF - exercise

✓ MULTIPOINT Z ((3.5 5.6 4.7), (4.8 10.5 7))

MULTIPOINT

Keyword

Dim? data

pointCoordList

pointCoord

coord

EBNF - réponse

```
multipoint = keyword dim? Data;
```

```
keyword = 'MULTIPOINT';
```

```
dim = 'Z' | 'M' | 'ZM';
```

```
data = '(' pcl ')';
```

```
pcl = pc { ',' pc };
```

```
pc = '(' number { ' ' number } ')';
```

```
number = [0..9] { [0..9] }
```

Pourquoi cette
syntaxe ?

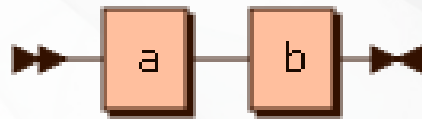
Attention destiné
au Lexer

Railroad Diagram

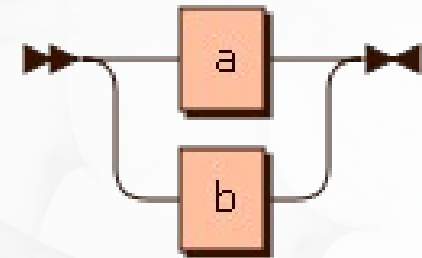
- ✓ Propose une représentation graphique de la grammaire
- ✓ <http://www.bottlecaps.de/rr/ui>
- ✓ Représente les séquences, les choix et les boucles
- ✓ Remarque : opérateurs nécessaires et suffisants
- ✓ cf : Grammar-lab.io

Railroad Diagram

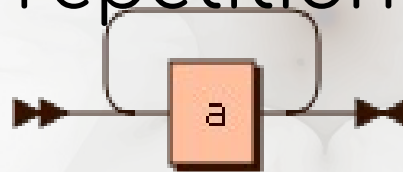
sequence $a\ b$



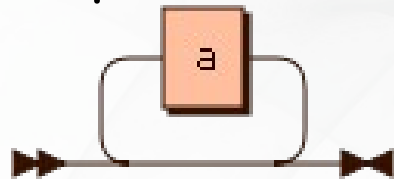
choix $a\ |\ b$



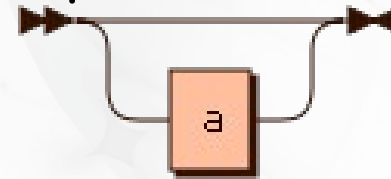
répétition 1-n a^+



répétition 0-n a^*

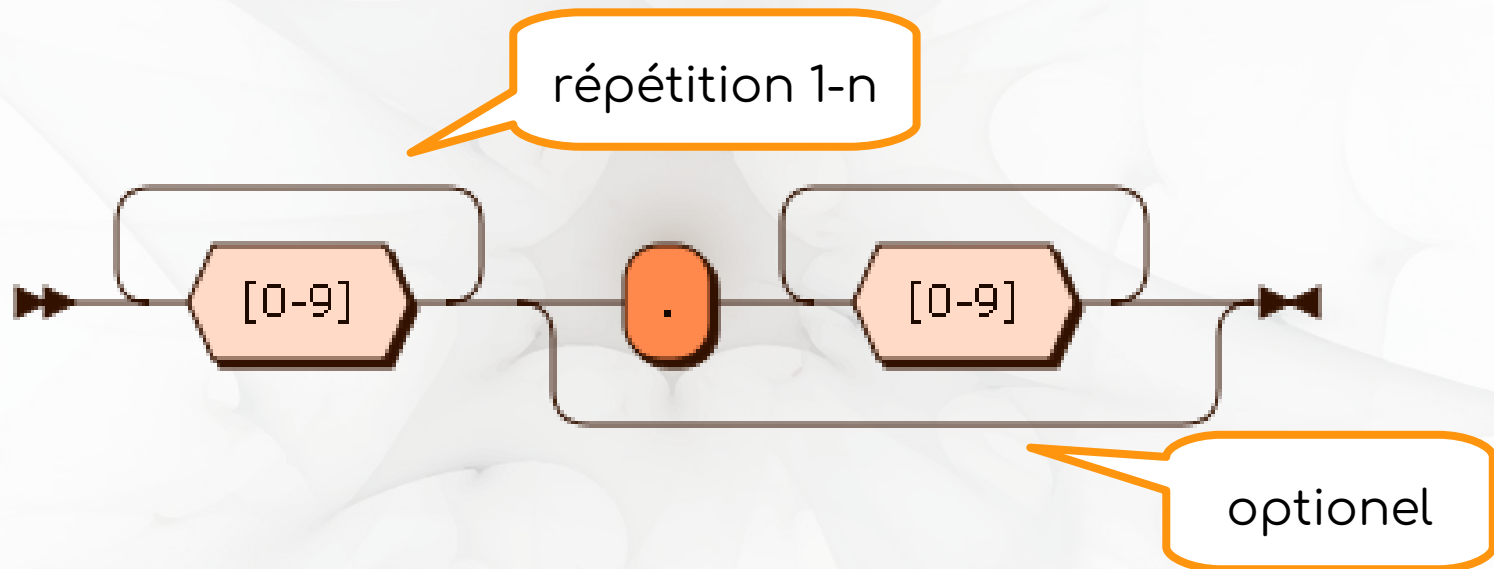


optionnel $a^?$



Railroad Diagram

$[0-9]^+ (\cdot [0-9]^+)?$



Équivalences Kleene / EBNF

- ✓ Optionnel :

$$a? \Leftrightarrow [a] \Leftrightarrow a \mid \varepsilon$$

- ✓ Répétition (0 .. n) :

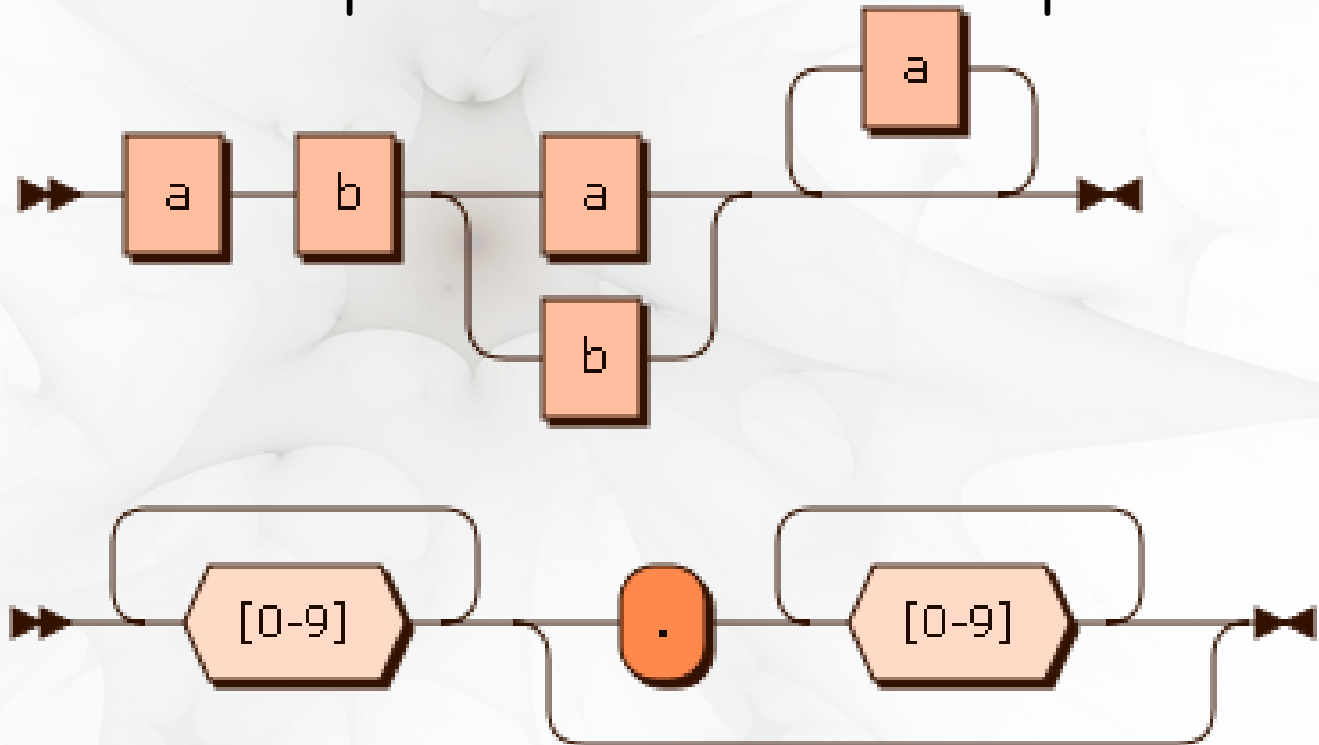
$$a^* \Leftrightarrow \{a\} \Leftrightarrow (a \mid \varepsilon)^+$$

- ✓ Répétition (1 .. n) :

$$a^+ \Leftrightarrow a \{a\}$$

Exercice

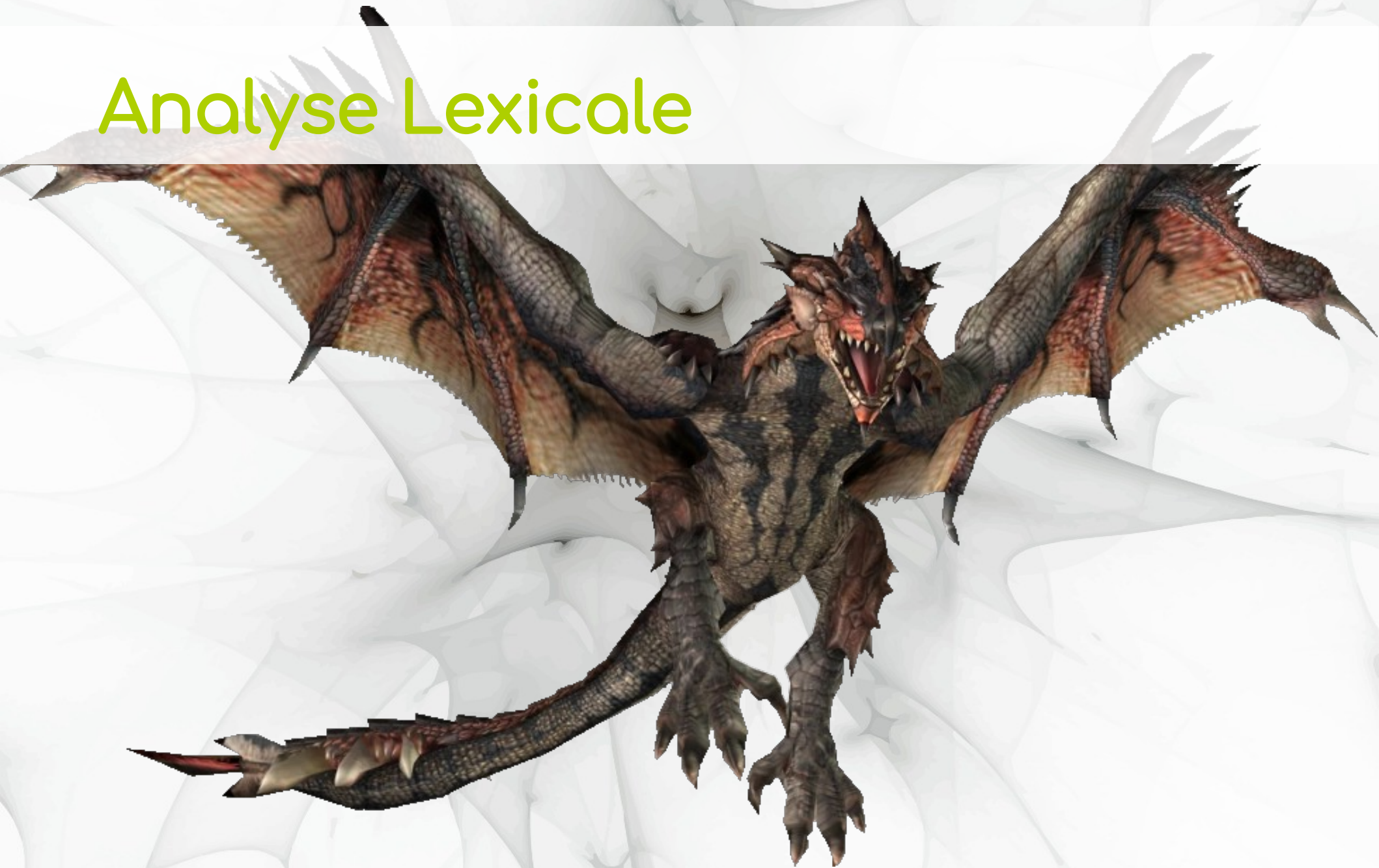
- ✓ Donner des exemples de mots acceptables pour :



Réponses

- ✓ aba, abb, abaaaa, abbaaaaa
- ✓ non acceptés : abbb, aaba, bab
- ✓ 0, 1, 50, 25.4, 29.7598
- ✓ non acceptés : .25, 2.2.2, 27.

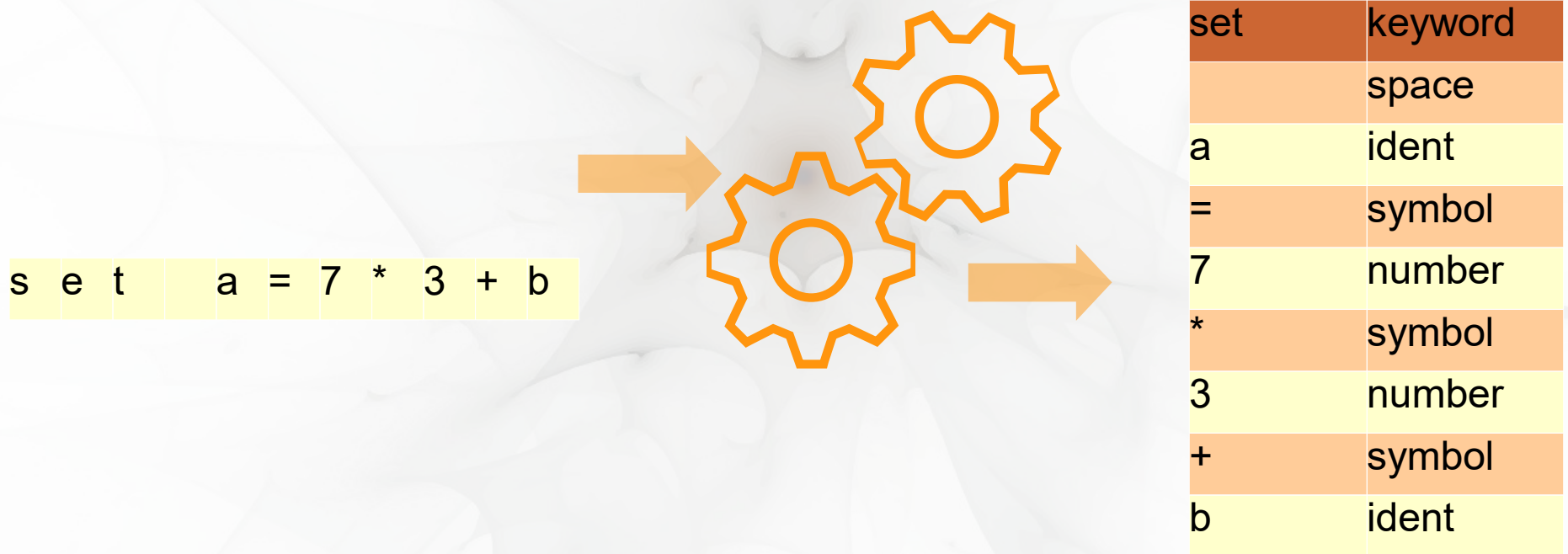
Analyse Lexicale



Généralités

- ✓ But :
 - ✓ Lire le code source
 - ✓ Reconnaître les mots
 - ✓ Distinguer les types de mots (chiffres, identifiants, chaînes de caractères, symboles, espaces)
- ✓ Entrée :
 - ✓ Une liste de caractères
- ✓ Sortie :
 - ✓ Une liste de mots typés (lexèmes)

Lexer - définition



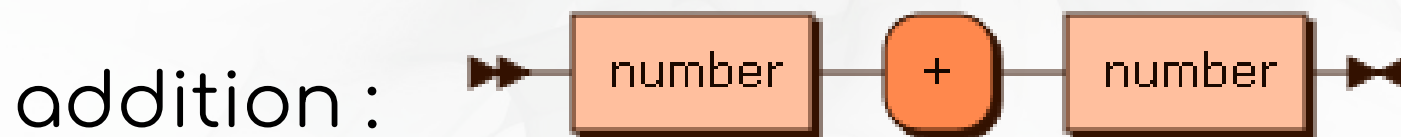
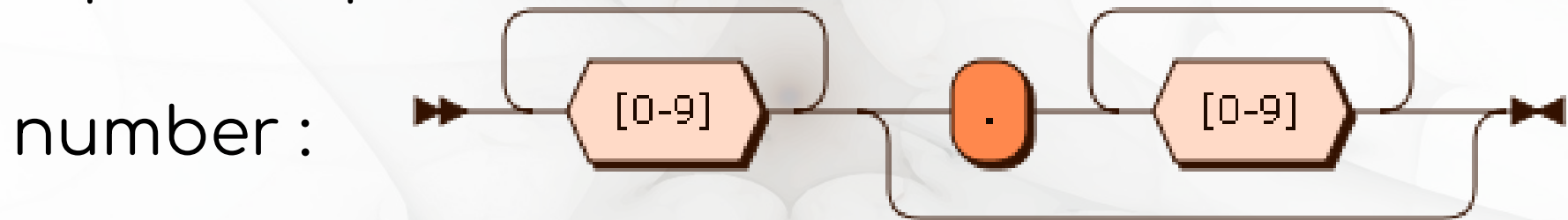
Remarques

- ✓ Consommé par l'analyseur syntaxique
- ✓ Mais pas obligatoire (algorithmes similaires entre lexer et parser)
- ✓ Utilise généralement un algorithme LL(1)
- ✓ Également utilisé lors de la coloration syntaxique

```
void main() {  
    set a = 7 * 3 + b  
}
```

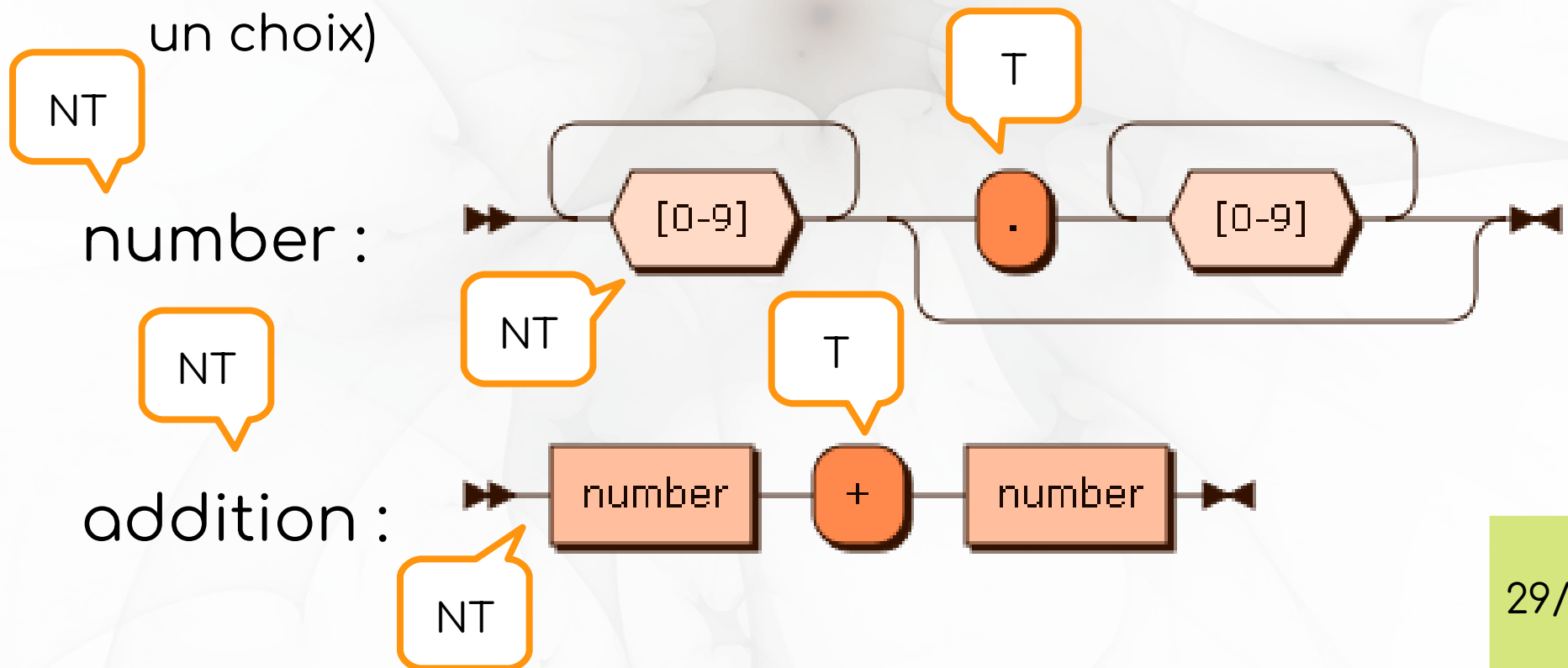
Question ?

- ✓ Différence entre grammaire pour le lexer et pour le parser



Terminal / Non Terminal

- ✓ Et dit terminal ssi aucune règle n'existe pour le transformer en autre chose
- ✓ Et dit non terminal ssi une règle existe (dés qu'il existe un choix)



Remarques

- ✓ Ces mots typés sont appelés Lexèmes ou Tokens
- ✓ Types généralement utilisés :
 - ✓ SPACE, COMMENT - /* */ \n
 - ✓ SYMBOL - () { } + - * / %
 - ✓ NUMBER, STRING
 - ✓ KEYWORD – if, else, while, do
 - ✓ IDENT – Test sur le type uniquement

Exercice

- ✓ Donner la définition d'un analyseur lexical capable de parcourir ce script :

```
if (_var0 < 0.1) { /* test */  
    print ("too small")  
} else {  
    _var0 += 1  
}
```

Réponse

```
digit ::= [0-9]
alpha ::= [a-z] | [A-Z] | '_'
alphaNum ::= digit | alpha
character ::= alpha | num | Space
```

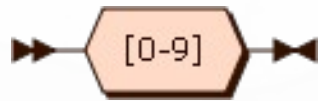
```
SPACE ::= ' ' | '\n' | '\t'
COMMENT ::= '/*' character* '*/'
SYMBOL ::= '+=' | '=' | '+' | '(' | ')' | '{' | '}'
NUMBER ::= digit+ ('.' digit+)?
STRING ::= '"' character* '"'
KEYWORD ::= 'if' | 'then'
IDENT ::= alpha alphaNum*
```

L'ordre est il important ?

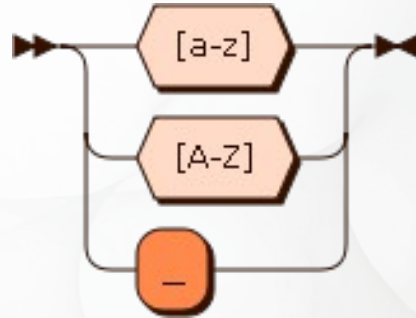
Production d'un
lexème + type

Pourquoi pas
alphaNum+ ?

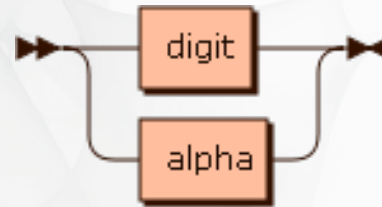
digit



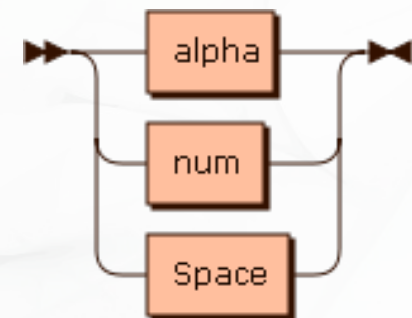
alpha



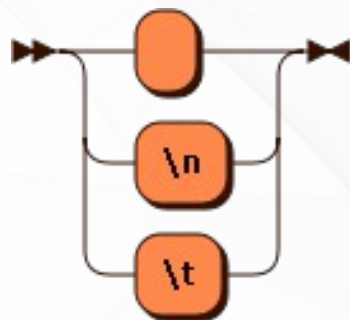
alphaNum



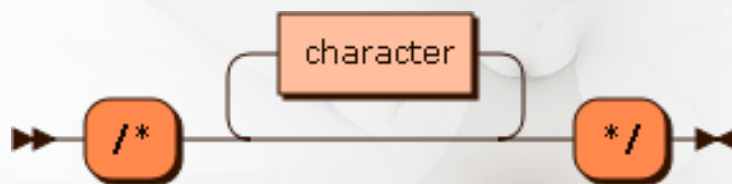
character



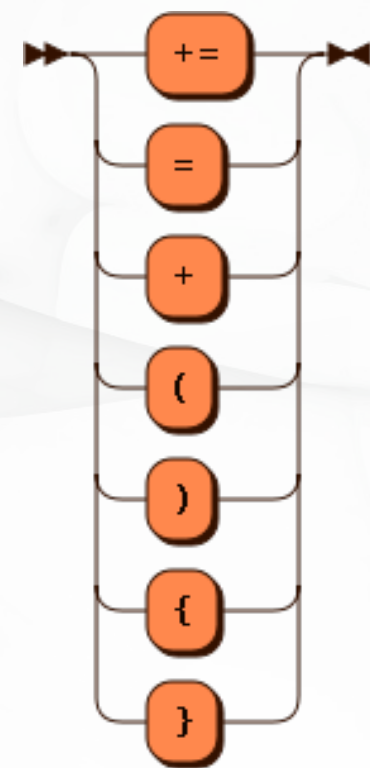
SPACE



COMMENT



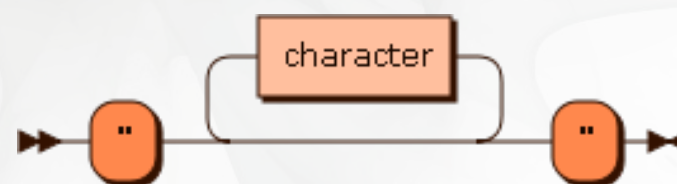
SYMBOL



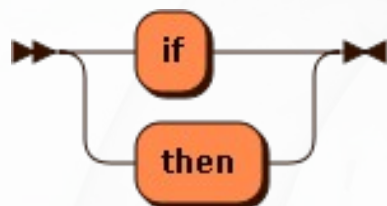
NUMBER



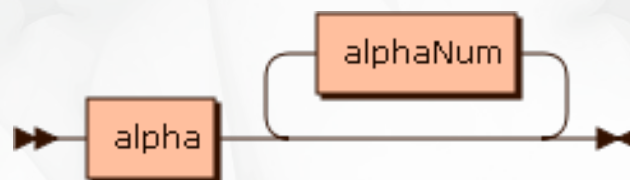
STRING



KEYWORD



IDENT

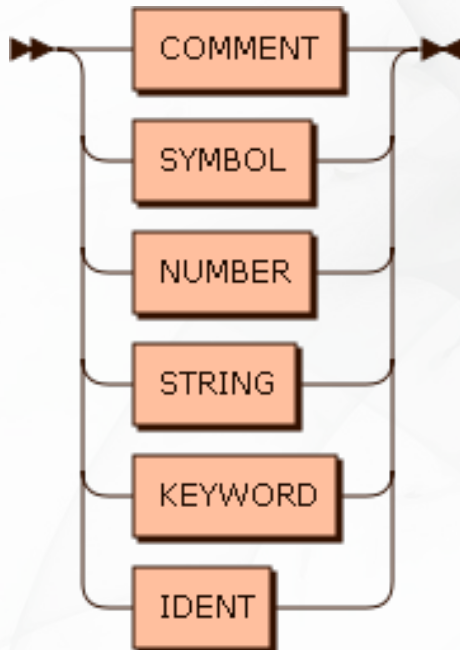


Lexème global

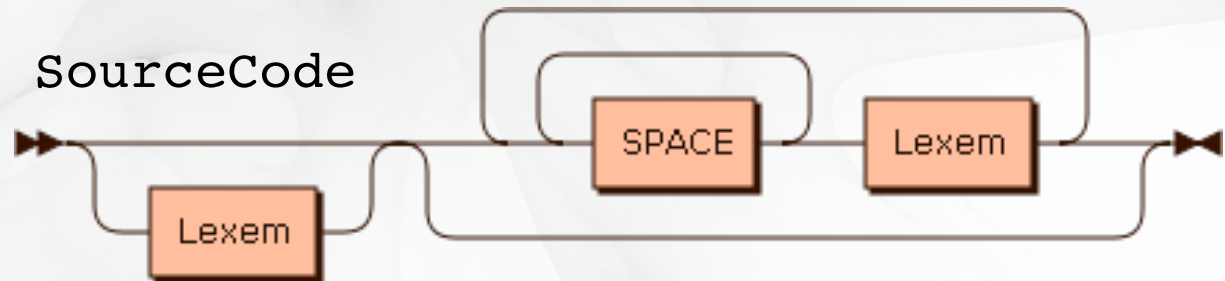
- ✓ Un programme est une liste de mots

```
Lexem ::= COMMENT | SYMBOL | NUMBER |  
        STRING | KEYWORD | IDENT  
SourceCode ::= Lexem? (SPACE Lexem)*
```

Lexem



SourceCode



Algorithme LL(0)



Algorithme LL(0)

- ✓ Pour transformer la chaîne de caractère en liste de lexèmes, il faut :
- ✓ Lire les caractères uns a uns
- ✓ De gauche à droite
- ✓ Déterminer le type de production
- ✓ Stocker le résultat dans une liste

LL(0) Recursive descent parser

abbaaaaa

consumed ↑↑

```
function testA() {  
    if (current == "a") return true;  
    return false;  
}
```

```
function consumeA() {  
    // do stuffs  
    next()  
}
```

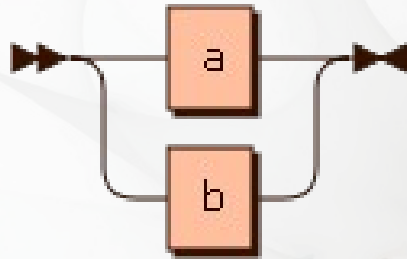
Implémentation typique



```
function sequence() {  
  if (testA()) consumeA()  
  else return false  
  
  if (testB()) consumeB()  
  else return false  
  
  return true  
}
```

Implémentation typique

choix



```
function choice() {  
  
    if (testA()) consumeA()  
    else if (testB()) consumeB()  
    else return false  
  
    return true  
}
```

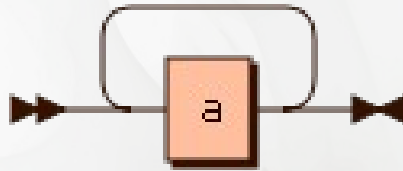

Implémentation typique



```
function optional() {  
    if (testA()) consumeA()  
    return true  
}
```

Implémentation typique

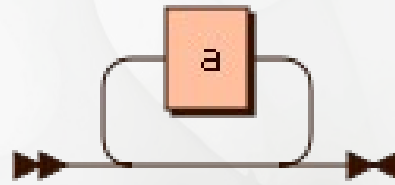
répétition + :



```
function repeatOneOrMore() {  
    if (!testA()) return false  
    do {  
        consumeA()  
    } while (testA())  
  
    return true  
}
```

Implémentation typique

répétition * :

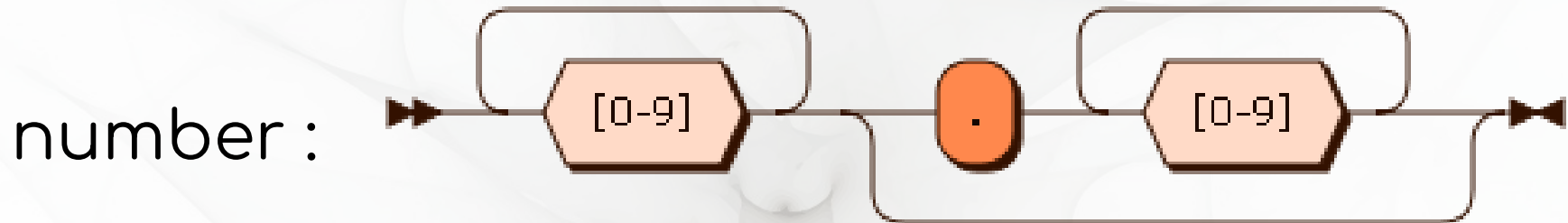


```
function repeatZeroOrMore() {  
    while (testA()) {  
        consumeA()  
    }  
    return true  
}
```

Algorithme LL(0)

- ✓ Left to Right
- ✓ Left most derivation
 - ✓ L'élément de gauche détermine quelle règle grammaticale appliquer
- ✓ Look ahead de 0
 - ✓ L'algorithme ne teste que le caractère courant
- ✓ Algorithme typiquement utilisé pour l'analyseur lexical
- ✓ Implémentation Recursive Descent Parser

Exemple



```
function parseNumChar() {  
  
    if (!testNum()) return false  
    do {  
        consume()  
    } while (testNum())  
  
    return true  
}
```

Répétition +

```
function parseNumber() {  
  
    parseNumChar()  
    if (testDot()) {  
        consume()  
        parseNumChar()  
    }  
}
```

Optionnel ?

Limitations

- ✓ Le premier symbole (Left most derivation) est déterminant pour la production
- ✓ Convient pour le Basic, le Pascal, Le LISP ou l'analyse lexicale
- ✓ Pourquoi les identifiants doivent commencer par une lettre (C like, Python, Pascal, Basic ...)
- ✓ `String str2 = « myString »` est valide
- ✓ `String 2dStr = « myString »` ne l'est pas !

Limitations - solution

- ✓ Pour augmenter le nombre de grammaire reconnus on peut augmenter le “look ahead”
- ✓ Avant la production on regarde les éléments suivant
- ✓ L'analyseur LL(k)
- ✓ k détermine le nombre d'éléments à regarder
- ✓ LL(0) est un cas particulier du LL(k)
- ✓ LL(*) lorsque k est illimité (pré-analyse sans production)

LL(k)

2dStr = "Hello"



look ahead

- ✓ Un look ahead de 2 permettrait de préfixer les identifiants avec un chiffre
- ✓ Ainsi de distinguer un identifiant d'un nombre de manière certaine
- ✓ Ne fonctionne plus si le nombre est plus grand : 27dString
- ✓ Une solution ?

LL(*)

```
2dStr = "Hello"
```



look ahead

- ✓ Un look ahead infini permet de déterminer à coup sûr de la grammaire qui va suivre
- ✓ Intervient lors d'un choix entre deux productions
- ✓ Algorithme Backtrack (parcours récursivement la structure)
- ✓ A suivre

Analyse Syntaxique

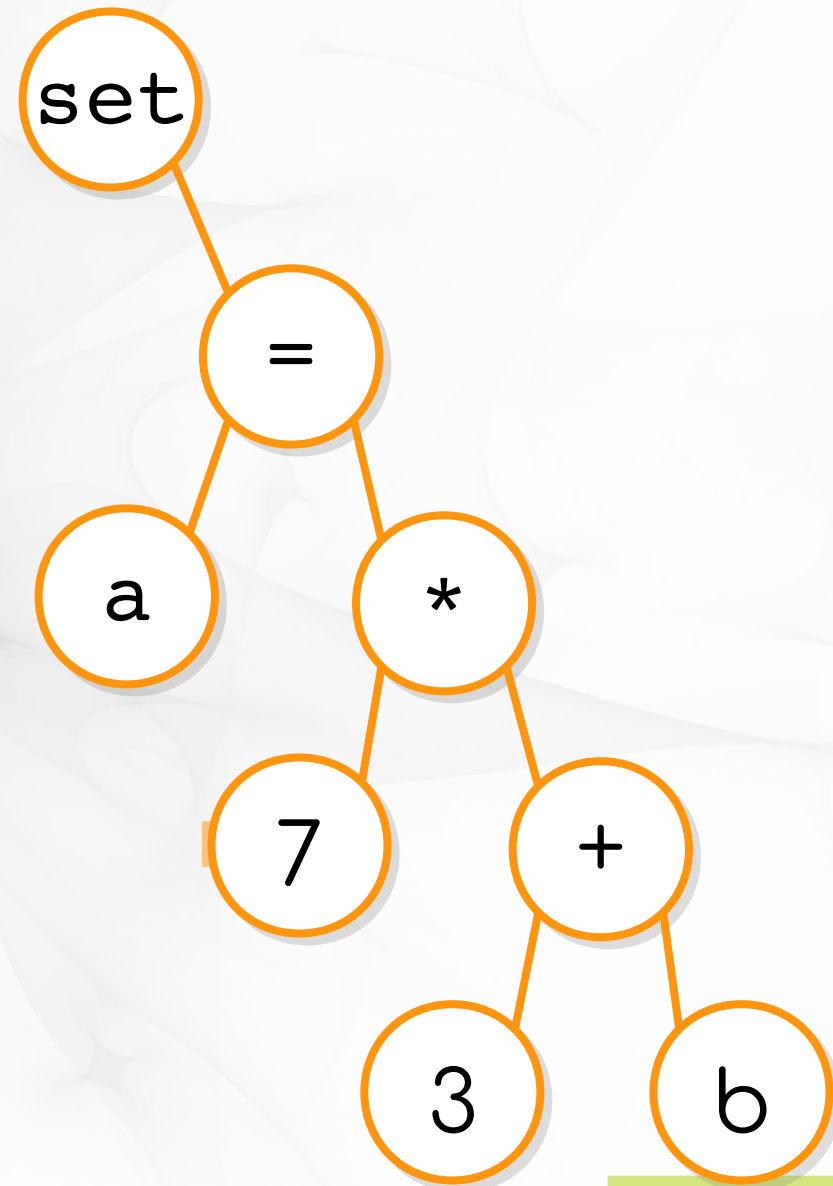
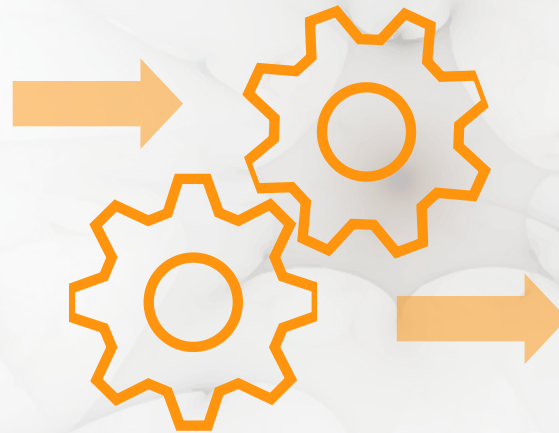


Généralités

- ✓ But :
 - ✓ Lire les lexèmes
 - ✓ Reconnaître les grammaires
 - ✓ Produire un AST (arbre de syntaxe)
- ✓ Entrée :
 - ✓ Une liste de lexèmes
- ✓ Sortie :
 - ✓ Un AST, arbre représentant le programme

Parser - définition

set	keyword
	space
a	ident
=	symbol
7	number
*	symbol
3	number
+	symbol
b	ident



Remarques

- ✓ Consomme le résultat de l'analyseur lexical
- ✓ Utilise généralement un algorithme plus complexe que le LL(0)
- ✓ LL(0) utilisé par BASIC, PASCAL, MsDOS
- ✓ Consommé par l'analyseur sémantique et/ou le codegen
- ✓ Algorithme Récursif : Recursive Descent Parser

Algorithme LL(1)

✓ $A \rightarrow (B)$

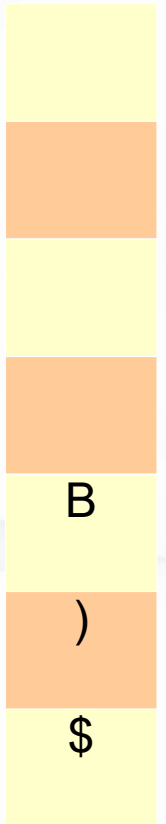
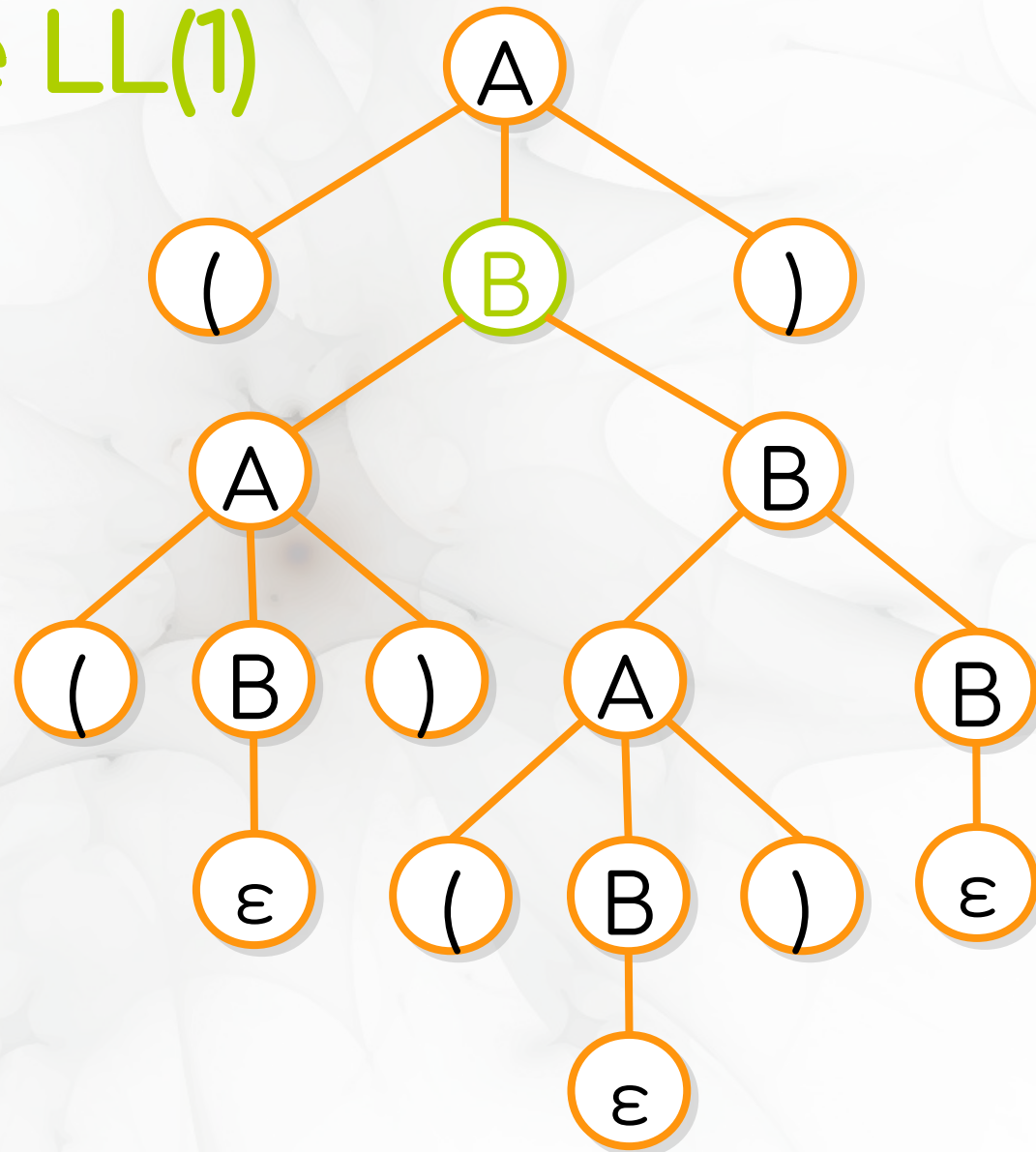
✓ $B \rightarrow AB$

✓ $B \rightarrow \epsilon$

↓

	()	\$
A	$A \rightarrow (B)$	-	-
B	$B \rightarrow AB$	$B \rightarrow \epsilon$	$B \rightarrow \epsilon$

→

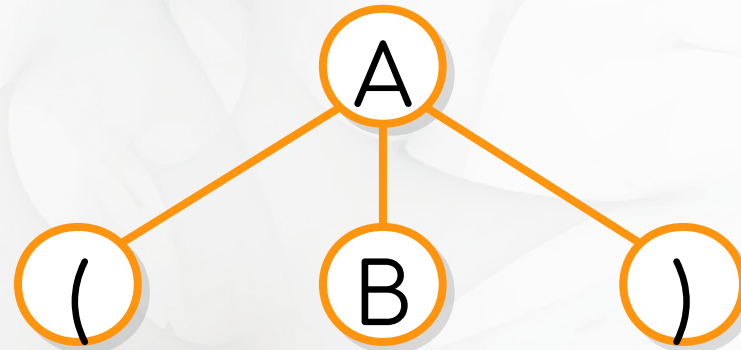


✓ Exemple : ((()())

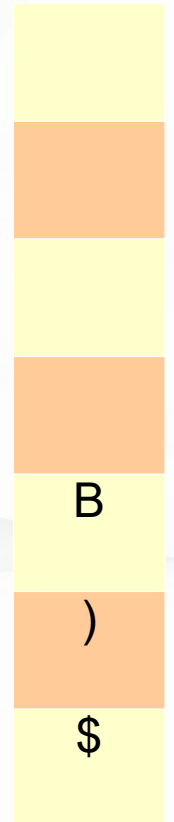
✓ $A \rightarrow (B)$

✓ $B \rightarrow AB$

✓ $B \rightarrow \epsilon$



	()	\$
A	$A \rightarrow (B)$	-	-
B	$B \rightarrow AB$	$B \rightarrow \epsilon$	$B \rightarrow \epsilon$

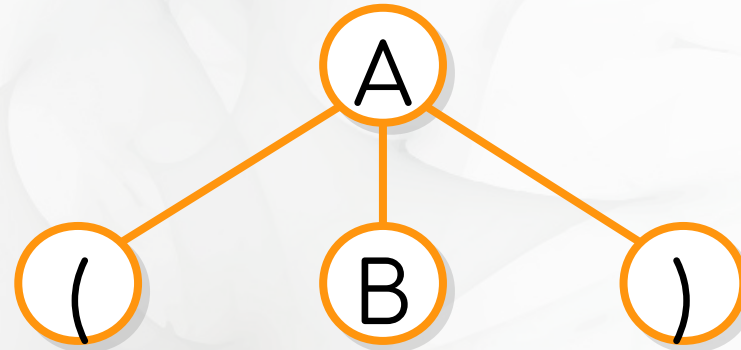


✓ Exemple : $((()(($

✓ $A \rightarrow (B)$

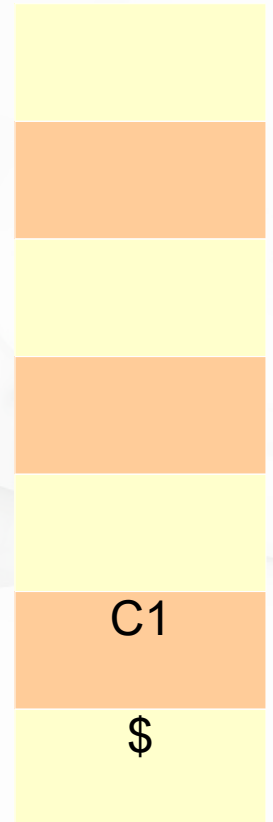
✓ $B \rightarrow AB$

✓ $B \rightarrow \varepsilon$



```

function parseA() {
  parse '('
  parseB() // c1
  parse ')'
}
function parseB() {
  if '(' {
    parseA() // c2
    parseB() // c3
  } else return true
}
    
```



✓ Exemple : $((()()))$

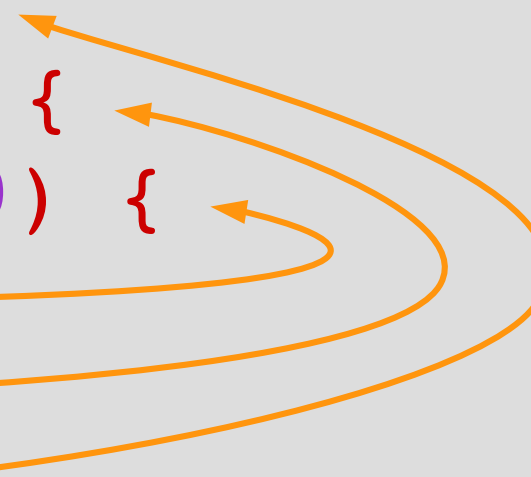
Remarques

- ✓ Tout algorithme récursif peut-être converti en machine à pile
- ✓ Et réciproquement
- ✓ Dans un algorithme récursif, on utilise la pile d'appel des fonctions

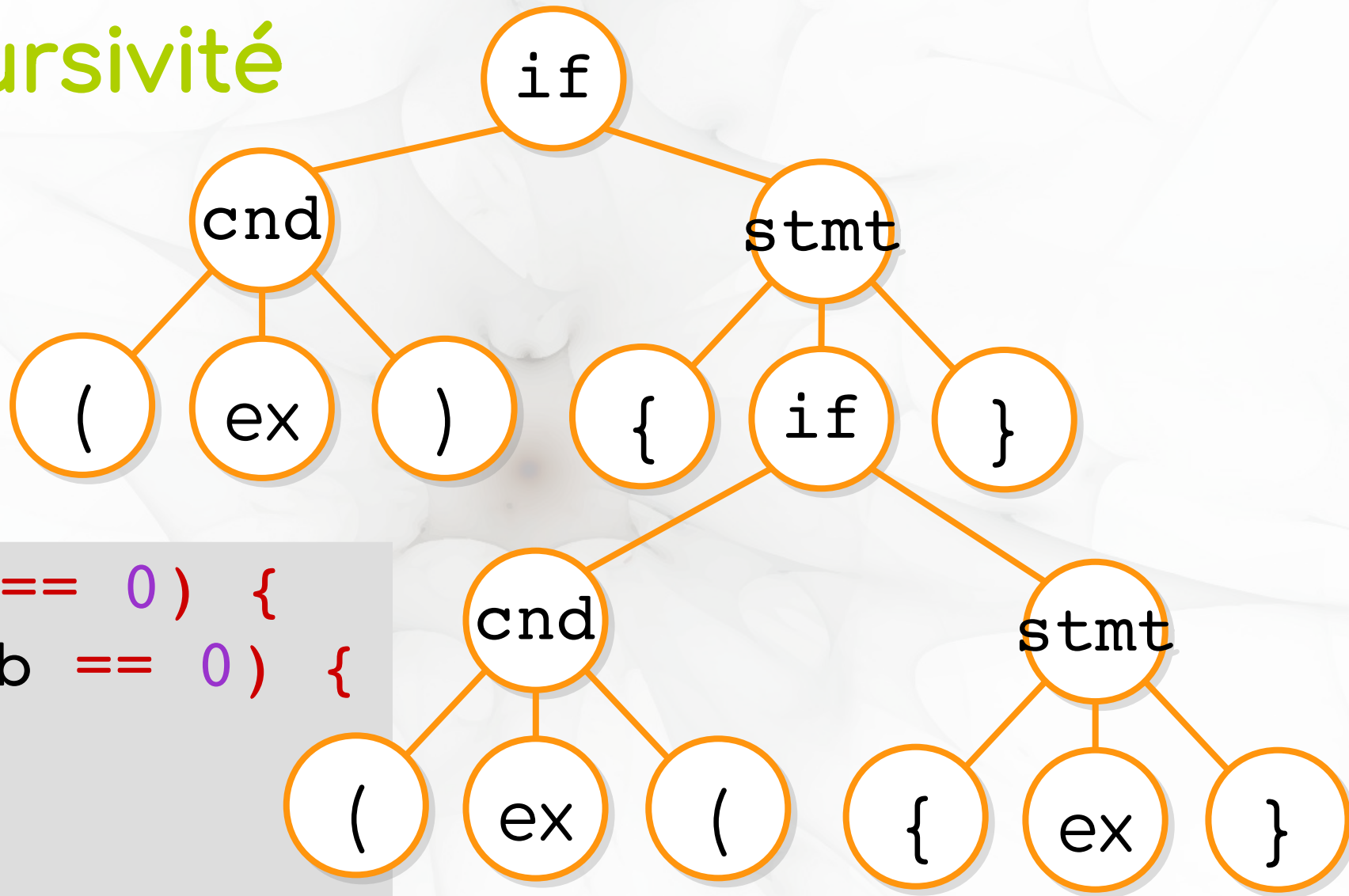
Récurtivité

- ✓ Exemple "bien parenthésé"

```
if (a == 0) {  
    if (b == 0) {  
        if (c == 0) {  
            }  
        }  
    }  
}
```



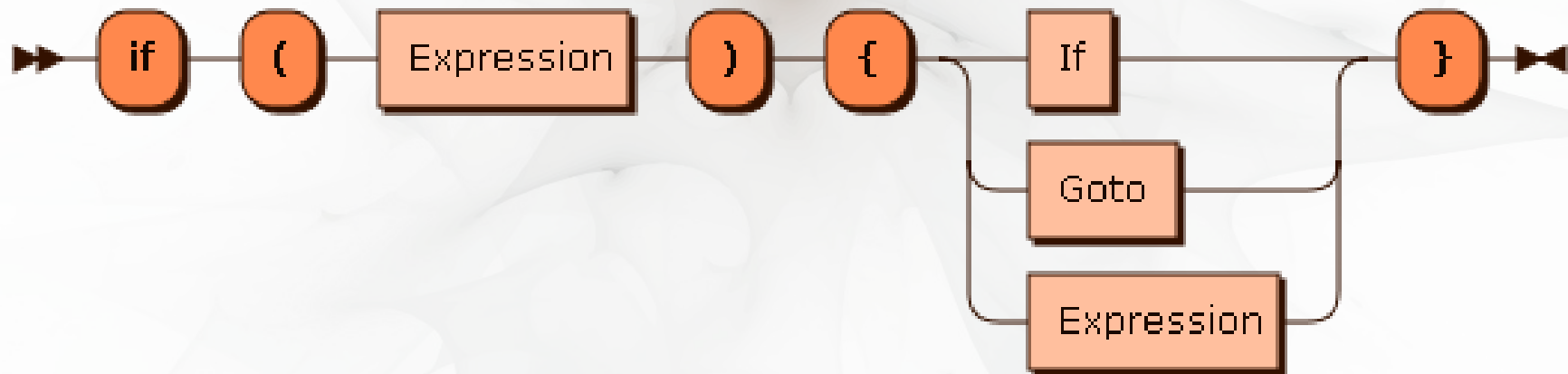
Récurtivité



```
if (a == 0) {  
    if (b == 0) {  
    }  
}
```

Récurtivité

- ✓ Récuratif dans les cas de : structures de contrôle, expressions arithmétiques, blocs de code, appels de fonctions etc....



Exercice

- ✓ Donner la définition d'un analyseur syntaxique capable d'interpréter ce script :

```
if (_var0 < 0.1) { /* test */  
    print ("too small")  
} else {  
    _var0 += 1  
}
```

Réponse

Récuratif

If ::= 'if' '(' Expr ')' Statement ('else' Statement)?

Parameter ::= Call | Expr

Pourquoi?

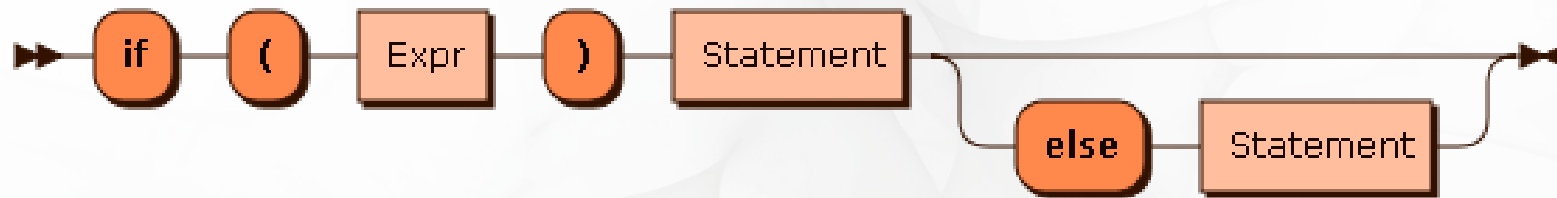
Call ::= Ident '(' (Parameter (',' Parameter)*)? ')'

Statement ::= If | Call | Expr | Block

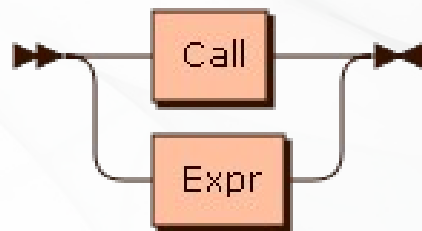
Block ::= '{' Statement* '}'

Recuratif

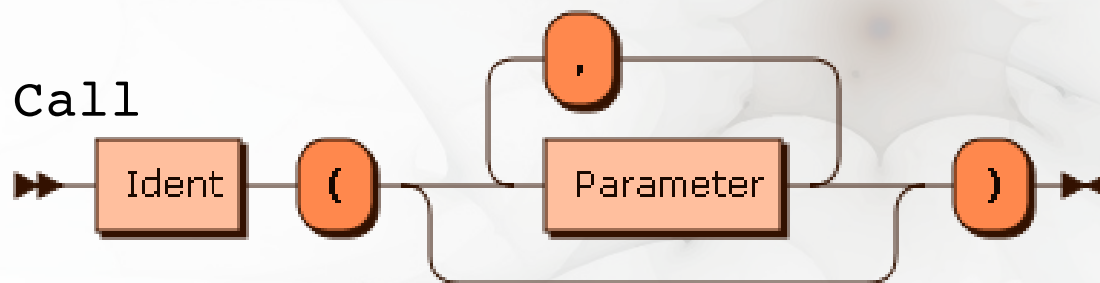
If



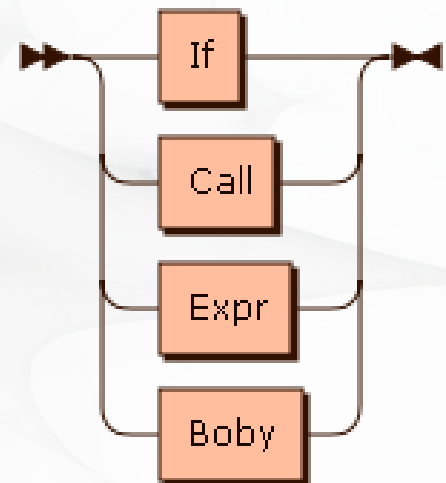
Parameter



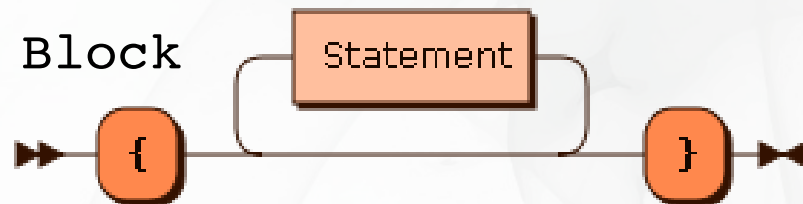
Call



Statement



Block



Algorithmes LL(k) et LL(*)



Limitations

- ✓ Rappel : Les algorithmes LL(0) sont limités
- ✓ Le symbole le plus à gauche (Left most dérivation) est déterminant pour la production
- ✓ Convient à la majeure partie des structures (if, set/let, while, do)
- ✓ Mais pas pour l'arithmétique !

LL1

10 + 10



- ✓ L'analyseur n'est pas capable de savoir, à cet endroit là :
 - ✓ S'il est en présence d'un chiffre seul
 - ✓ Ou si ce chiffre fait partie d'une opération
- ✓ **Grammaire ambiguë**
- ✓ $S \rightarrow n$
- ✓ $S \rightarrow S + S$
- ✓ ici S commence par un non terminal)

Impossible !

$S \rightarrow S + S$

$S \rightarrow n$

```
function parseSum() {  
  try {  
    parseSum()  
    if (testPlus()) consumePlus()  
    return false  
    parseSum()  
  } catch {  
    if (testNumber()) consumeNumber()  
    return false  
  }  
  return true  
}
```

Pas mieux !

$S \rightarrow n$

$S \rightarrow S + S$

```
function parseSum() {  
    if (testNumber()) consumeNumber()  
    else {  
        parseSum()  
        parsePlus()  
        parseSum()  
    }  
}
```

Solution 1 : Modifier le langage

(+ 10 10)



- ✓ Cas du LISP (John McCarthy 1958)
- ✓ Pas d'ambiguïté, l'opération commence par l'opérateur
- ✓ $S \rightarrow (+ S S)$
- ✓ $S \rightarrow n$

Mieux !

$S \rightarrow + S S$

$S \rightarrow n$

```
function parseSum() {  
    if (testPlus()) {  
        consumePlus()  
        parseSum()  
        parseSum()  
    } else if (testNumber())  
        consumeNumber()  
    else return false  
    return true  
}
```


Solution 2 : Modifier la grammaire

10 + 10



- ✓ $S \rightarrow n F$
- ✓ $F \rightarrow + S$
- ✓ $F \rightarrow \varepsilon$
- ✓ Suppression de la récursivité à gauche

Mieux !

$$S \rightarrow n F$$
$$F \rightarrow + S$$
$$F \rightarrow \varepsilon$$

```
function parseSum() {  
  if (testNumber()) consumeNumber()  
  else return false  
  
  if (testPlus()) {  
    consumePlus()  
    parseSum()  
  }  
  return true  
}
```

Cas Général

- ✓ Suppression de la récursivité à gauche
 - ✓ Immédiate : $A \rightarrow A\alpha, \alpha \in (T \cup N)^+$
 - ✓ Générale : $A \rightarrow ^*A\alpha, \alpha \in (T \cup N)^+$
- ✓ Il est possible de supprimer les deux cas par transformation de la grammaire

Récurtivité immédiate

✓ On remplace les règles de la forme :

✓ $X \rightarrow X\alpha_1 \mid \dots \mid X\alpha_n \mid \beta_1 \mid \dots \mid \beta_m$

✓ Par les règles :

✓ $X \rightarrow \beta_1 X' \mid \dots \mid \beta_m X'$

✓ $X' \rightarrow \alpha_1 X' \mid \dots \mid \alpha_n X' \mid \varepsilon$

Exercice

✓ Supprimer la récursivité à gauche de :

✓ $E \rightarrow E + T \mid T$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow i \mid (E)$$

✓ $X \rightarrow X\alpha_1 \mid \dots \mid X\alpha_n \mid \beta_1 \mid \dots \mid \beta_m$

✓ $X \rightarrow \beta_1 X' \mid \dots \mid \beta_m X'$

✓ $X' \rightarrow \alpha_1 X' \mid \dots \mid \alpha_n X' \mid \varepsilon$

Solution

✓ $E \rightarrow T E'$

$$E' \rightarrow + T E' \mid \varepsilon \quad (\text{ou } E' \rightarrow + \textcolor{brown}{E} \mid \varepsilon)$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid \varepsilon \quad (\text{ou } T' \rightarrow + \textcolor{brown}{T} \mid \varepsilon)$$

$$F \rightarrow i \mid (E)$$

✓ $X \rightarrow \beta_1 X' \mid \dots \mid \beta_m X'$

$$X' \rightarrow \alpha_1 X' \mid \dots \mid \alpha_n X' \mid \varepsilon$$

Précédence des opérateurs

✓ $E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

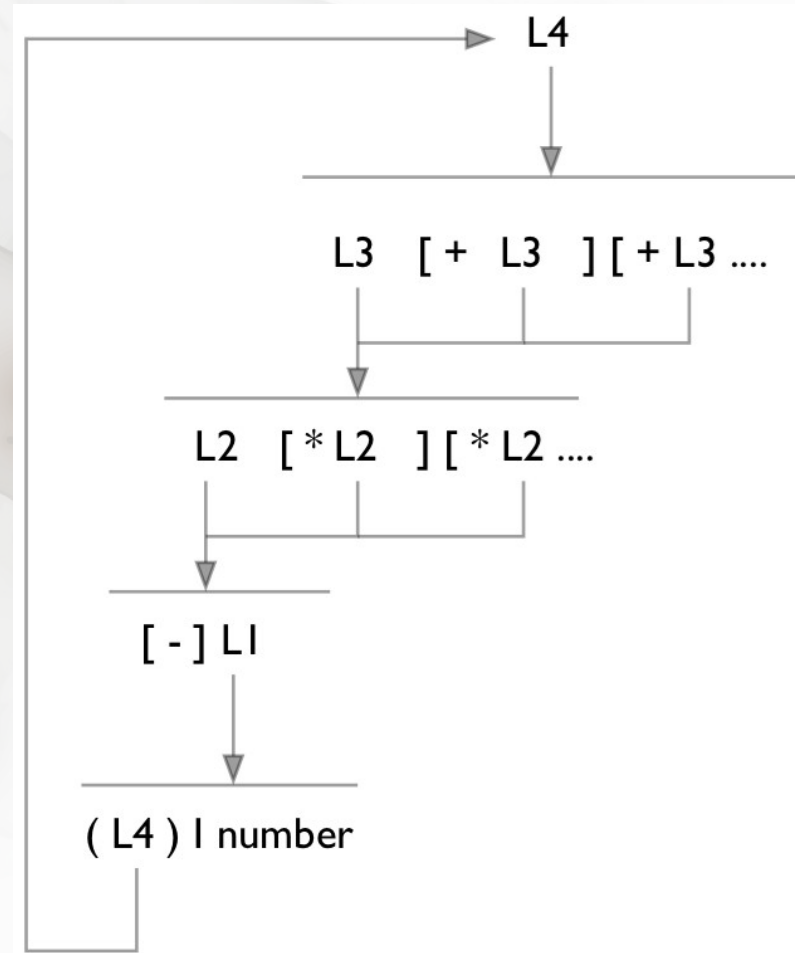
$F \rightarrow (E) \mid i$

✓ ex : $(3 + 7 * 2) + 4$

$E \rightarrow E + T$

$E \rightarrow T, T \rightarrow F, F \rightarrow (E)$

$E \rightarrow E + T, T \rightarrow T * F$



Solution 3 : Lookahead

10 + 10



lookahead

- ✓ Un lookahead de 2 permettrait de définir la production (chiffre ou addition)
- ✓ Lorsque l'algorithme est face à un choix de production, il lance une prédiction
- ✓ $S \rightarrow n (+ S) ^ *$

Mieux !

10 + 10



lookahead

✓ $S \rightarrow n (+ S)^*$

```
function parseSum() {  
    if (!testNumber()) return  
    moveForewardLL()  
    if (!testPlus()) return  
    moveBackwardLL()  
  
    parseNumber()  
    parsePlus()  
    parseSum()  
}
```

Algorithmes LL(k), LL(*) et LR



Solution 3 : Lookahead

10 + 10



lookahead

- ✓ Un look ahead de 2 permettrait de définir la production (chiffre ou addition)
- ✓ Lorsque l'algorithme est face à un choix de production, il lance une prédiction
- ✓ $S \rightarrow n + S$
- ✓ $S \rightarrow n$

Mieux !

10 + 10



lookahead

✓ $S \rightarrow n + S$

✓ $S \rightarrow S$

```
function parseSum() {  
    if (!testNumber()) return  
    moveForwardLL()  
    if (!testPlus()) return  
    moveBackwardLL()  
  
    parseNumber()  
    parsePlus()  
    parseSum()  
}
```

LL(k) et LL(*)

- ✓ L'algorithme LL(0) est un cas particulier de LL(k)
 - ✓ $k = 0$, pas de look ahead
- ✓ LL(k) est un cas particulier de LL(*)
 - ✓ k est fini alors que $*$ est infini
- ✓ antLR v1 \rightarrow LL(1)
- ✓ JavaC \rightarrow LL(k)
- ✓ antLR v3 \rightarrow LL(*)

Prédictibilité

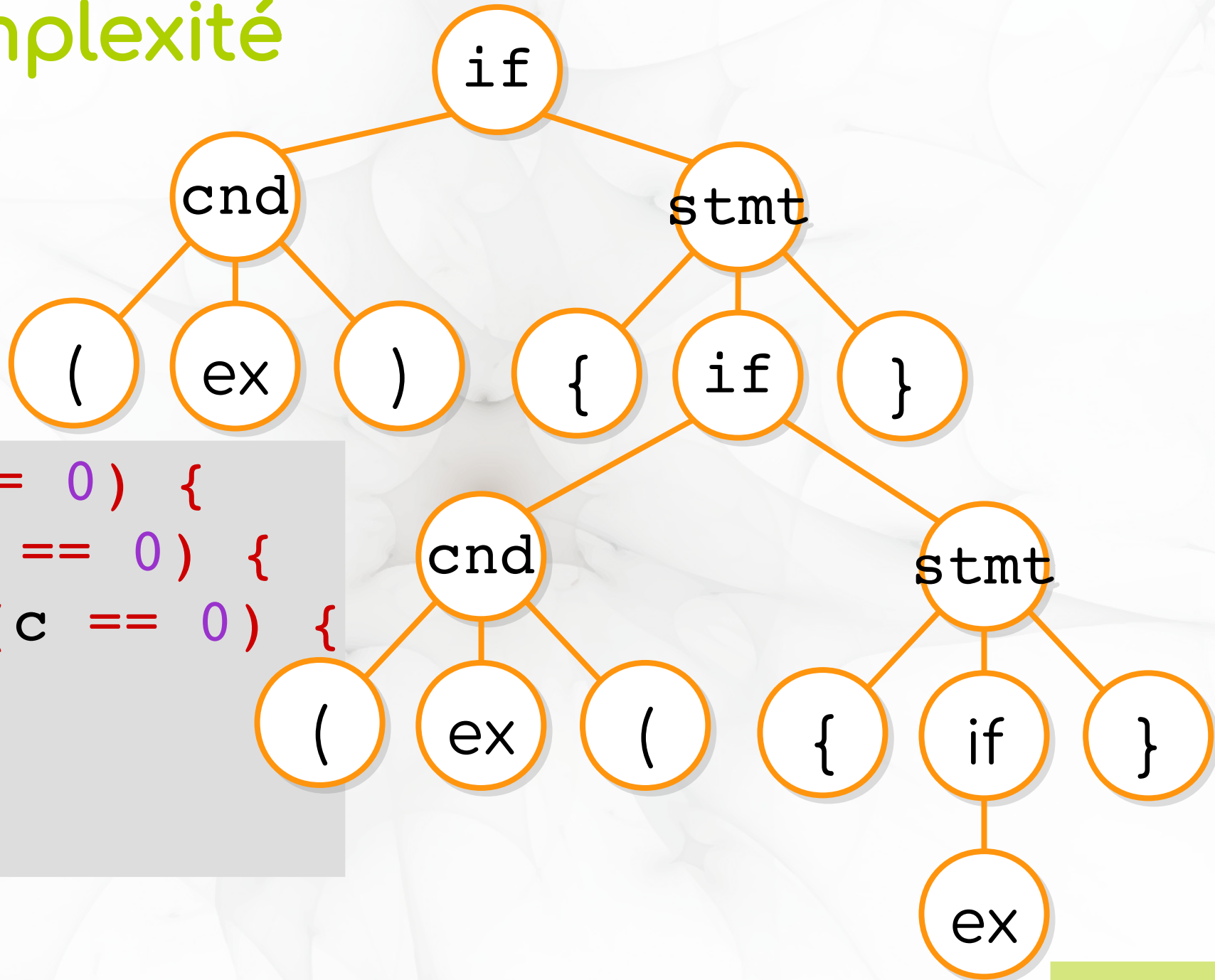
- ✓ Principe : lors d'une décision, l'algorithme lance une prédiction
- ✓ c'est-à-dire un algorithme récursif qui ne consomme pas et ne produit pas

```
function predictSum() {  
    if (testNumber()) {  
        moveForeward()  
  
        if (testPlus()) {  
            moveForeward()  
  
            if (predictSum())  
                return true  
  
            moveBackward()  
        }  
        moveBackward()  
    }  
    return false  
}
```

Limitations

- ✓ L'algorithme $LL(*)$ refuse toujours la récursivité à gauche.
- ✓ De plus sa complexité est exponentielle

Complexité



Solution : Packrat

- ✓ Utiliser un cache.
- ✓ Pour chaque décision, stocker une paire clé (position), valeur (décision)
- ✓ Utiliser le cache lors du prochain passage
- ✓ Garantie que l'évaluation n'aura lieu qu'une seule fois
- ✓ Algorithme **Packrat**
- ✓ Complexité presque linéaire

Analysér LR



Généralités

- ✓ Inventé par Donald Knuth en Juillet 1965
- ✓ Left To Right, Right most derivation
- ✓ LL : Approche Top Down
- ✓ LR : Approche Bottom Up

Analyseur LR



Donald Knuth
1938

- ✓ Inventé par Donald Knuth
- ✓ en Juillet 1965
- ✓ Left To Right, Right most derivation
- ✓ LL : Approche Top Down
- ✓ LR : Approche Bottom Up

Analyseur LR

- ✓ 2 structures :
 - ✓ Flot d'entrée
 - ✓ Pile
- ✓ 4 opérations :
 - ✓ **Shift** : transfert du flot d'entrée vers la pile
 - ✓ **Reduce** : On reconnaît sur le sommet de la pile une partie droite d'une production, on la remplace par la partie gauche
 - ✓ **Erreur** : Arrêt et signalement d'une erreur
 - ✓ **Accept** : Arrêt, phrase reconnu

Exemple : SLR

- ✓ Simple LR parser ou LR(0)
- ✓ Soit le flot d'entrée : $10 + 10 + 10$
- ✓ Et la grammaire :

$$E \rightarrow E + T \mid T$$

$$T \rightarrow n$$

$$E \rightarrow E + T \mid T$$

$$T \rightarrow n$$

	10 + 10 + 10	shift
10	+ 10 + 10	reduce
T	+ 10 + 10	reduce
E	+ 10 + 10	shift
E +	10 + 10	shift
E + 10	+ 10	reduce
E + T	+ 10	reduce
E	+ 10	shift
E +	10	shift
E + 10		reduce
E + T		reduce
E		accept

LALR

- ✓ LALR – Look-Ahead LR Parser
- ✓ Analyseur avec anticipation
- ✓ Permet un ensemble de grammaire plus grand
- ✓ Fonctionnement de YaCC
- ✓ Permet de résoudre l'ambiguïté :
- ✓ $S \rightarrow L \bullet = R$ (Curseur •)
- ✓ $R \rightarrow L \bullet$

$$E \rightarrow E + T \mid T$$

$$T \rightarrow n$$

	10 + 10 + 10	shift
10	+ 10 + 10	reduce
T	+ 10 + 10	shift
T +	10 + 10	shift
T + 10	+ 10	reduce
T + T	+ 10	shift (*)
T + T +	10	shift
T + T + 10		reduce
T + T + T		reduce (*)
T + T + E		reduce
T + E		reduce
E		accept

Compilateurs de compilateurs

- ✓ Les analyseurs LR sont trop complexes à écrire à la main
- ✓ Généralement construits par des générateurs d'analyseur (compilateurs de compilateurs)
- ✓ Crée une table d'analyse
 - ✓ Action, les actions à faire en fonction des symboles rencontrés
 - ✓ Transition, les branchements

Bootstrapping - Auto-hébergé

- ✓ Le compilateur du langage écrit dans le langage lui-même
- ✓ Imaginé pour le LISP, idée poursuivie pour C et Pascal
- ✓ Paradoxe de l'œuf et de la poule (pourquoi et comment)
- ✓ Comment : premier compilateur minimal en assembleur, puis suite de versions de plus en plus complexe / haut niveau
- ✓ Pourquoi : plus simple que d'écrire un compilateur complet en assembleur

Les algorithmes

- ✓ Deux grandes familles d'algorithmes :
- ✓ $LL \rightarrow LL(0), LL(k), LL(*), \text{Packrat}, \text{PEG (Parsing Expression Grammar)}$
- ✓ $LR \rightarrow \text{SLR (Simple LR)}, \text{LALR (Look Ahead LR)}, \text{GLR}$

Arbre syntaxique abstrait (AST)

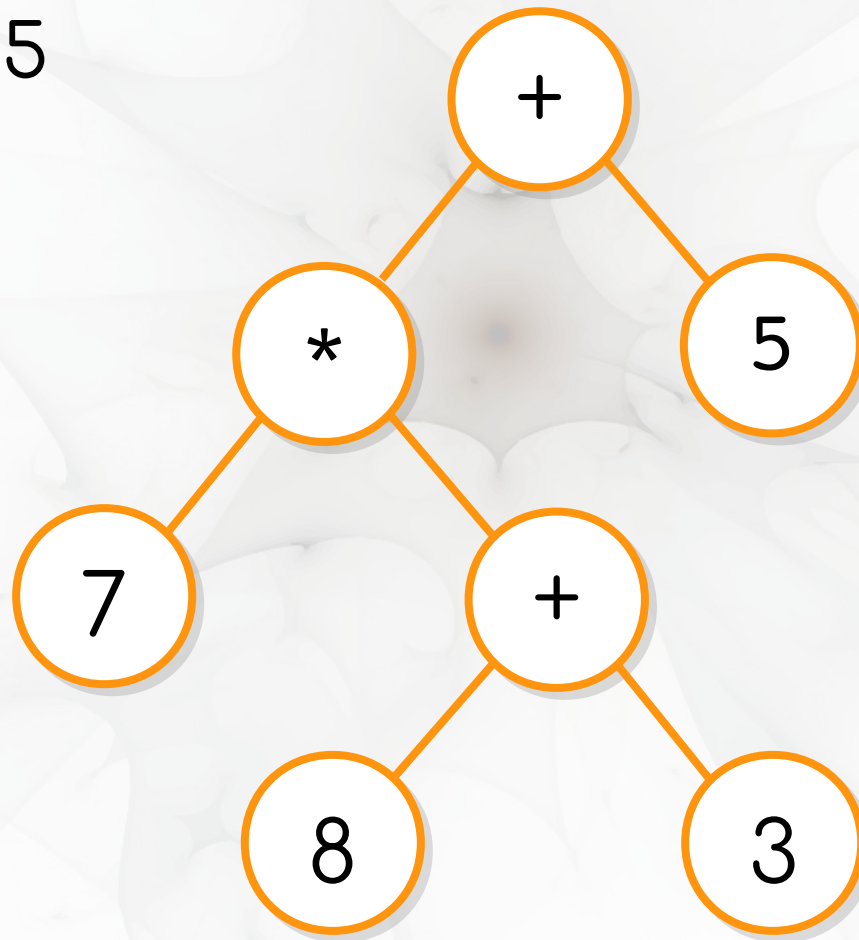


AST : Arbre syntaxique

- ✓ En : Abstract Syntax Tree
- ✓ Arbre logique qui représente la structure du programme
- ✓ Ses nœuds sont des opérateurs et ses feuilles des opérandes (variables ou constantes)
- ✓ Une production grammaticale produit un nœud de l'arbre

Exemple

$$7 * (8 + 3) + 5$$



AST - Parcours

- ✓ CodeGen arithmétique → LISP

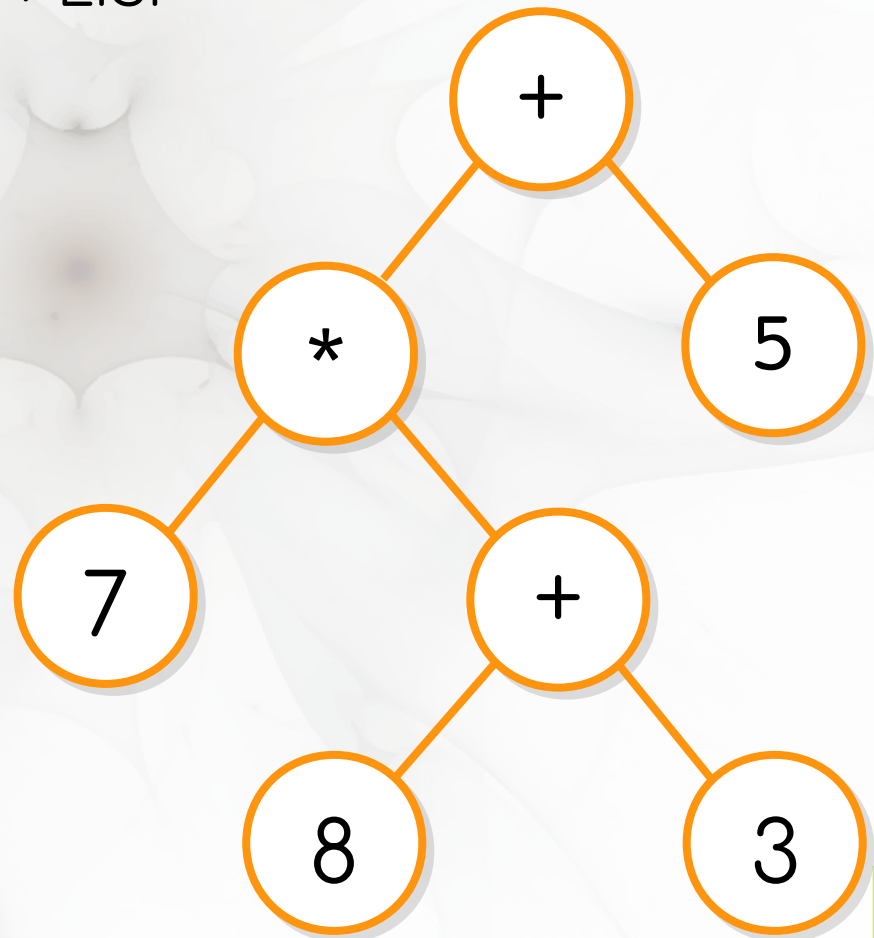
- ✓ $7 * (8 + 3) + 5$

- ✓ LISP :

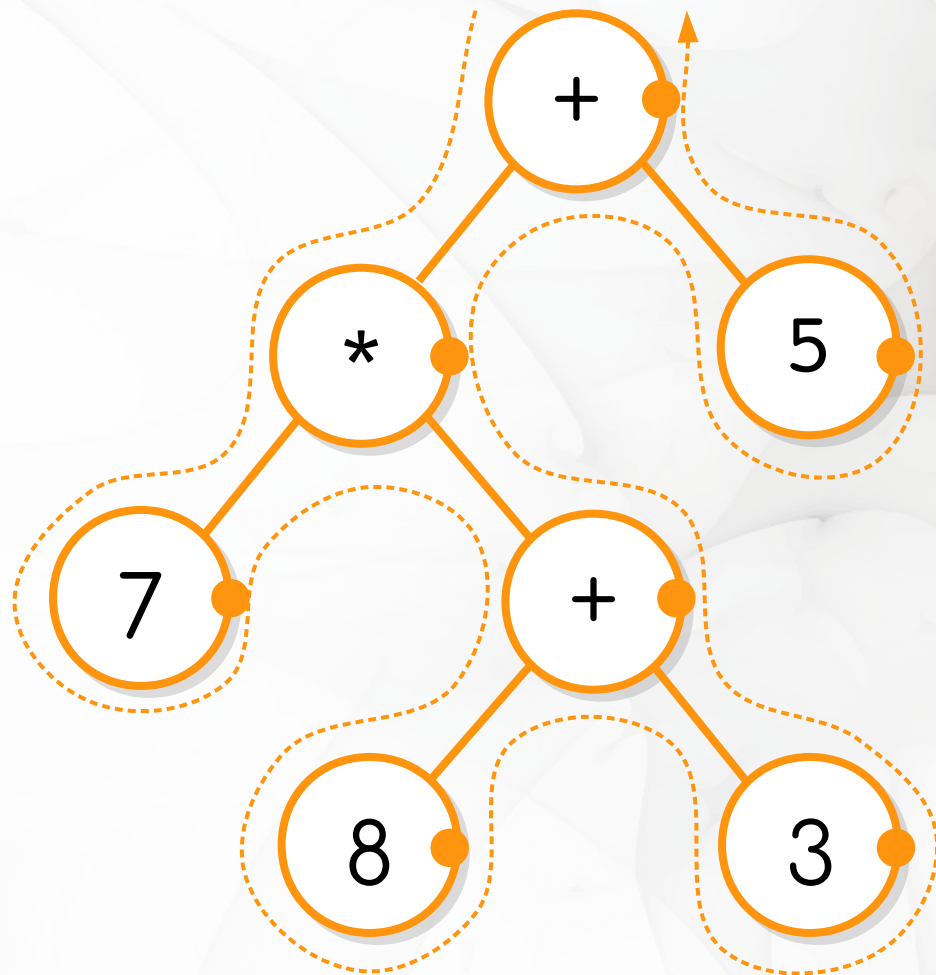
- ✓ $(+ (* 7 (+ 8 3)) 5)$

- ✓ Cf : Parcours d'arbres

- ✓ Cf : Demo Algo



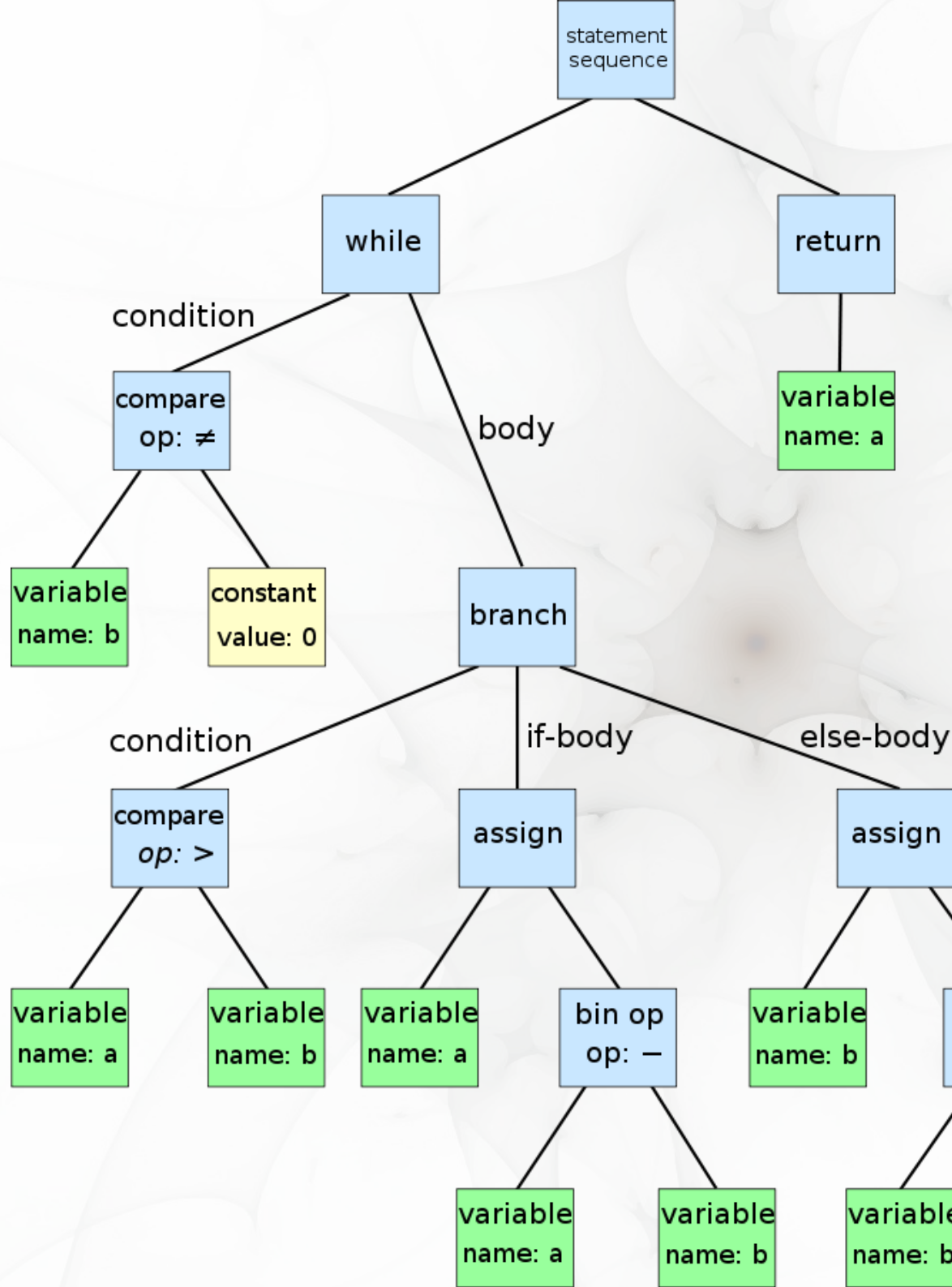
AST - Parcours



Pile

			+					
		3	3		*		+	
	8	8	8	11	11		5	
7	7	7	7	7	7	77	77	82

Temps



```
while b  $\neq$  0
  if a > b
    a := a - b
  else
    b := b - a
return a
```


Grammaires ambiguës

- ✓ Définition : une grammaire ambiguë c'est lorsque qu'une grammaire peut engendrer deux arbres différents
- ✓ Exemple : **if** - **if** – **else** (à quel **if** correspond le **else**)
- ✓ Solution : le Parsing Expression Grammar
 - ✓ Dans le cas d'une grammaire ambiguë, le PEG va choisir la première qui correspond
 - ✓ En réalité proche du programme et non de la théorie

Production – simple

$E \rightarrow E + E$

$E \rightarrow (E)$

$E \rightarrow \text{int}$

7 + 8 + 3



Précédence – contre exemple

$E \rightarrow E * E$

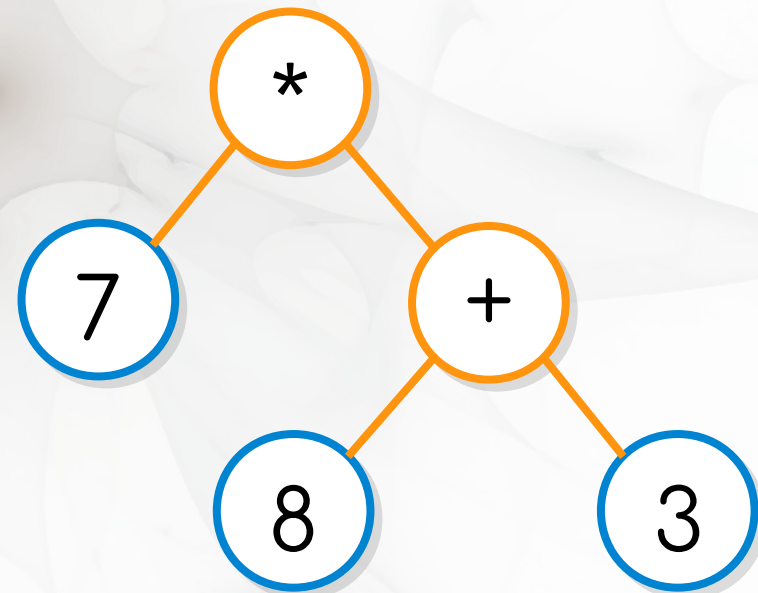
$E \rightarrow E + E$

$E \rightarrow (E)$

$E \rightarrow \text{int}$

$7 * 8 + 3$

$= 77 \text{ ???}$



$A \rightarrow M (+ A) ?$

$M \rightarrow E (* M) ?$

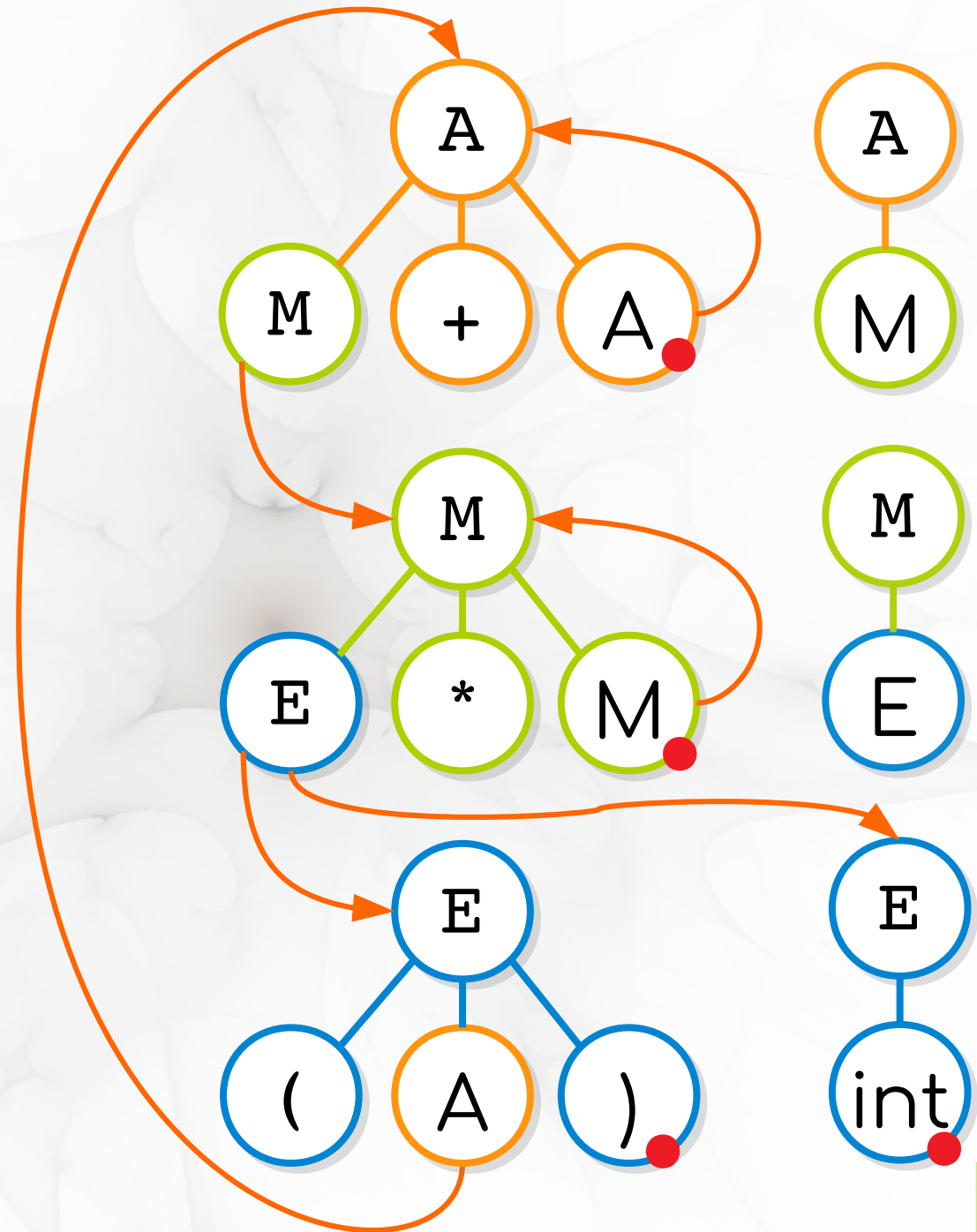
$E \rightarrow (A) \mid \text{int}$

$3 * 2 + 5$

$3 + 2 * 5$

$3 * (2 + 5)$

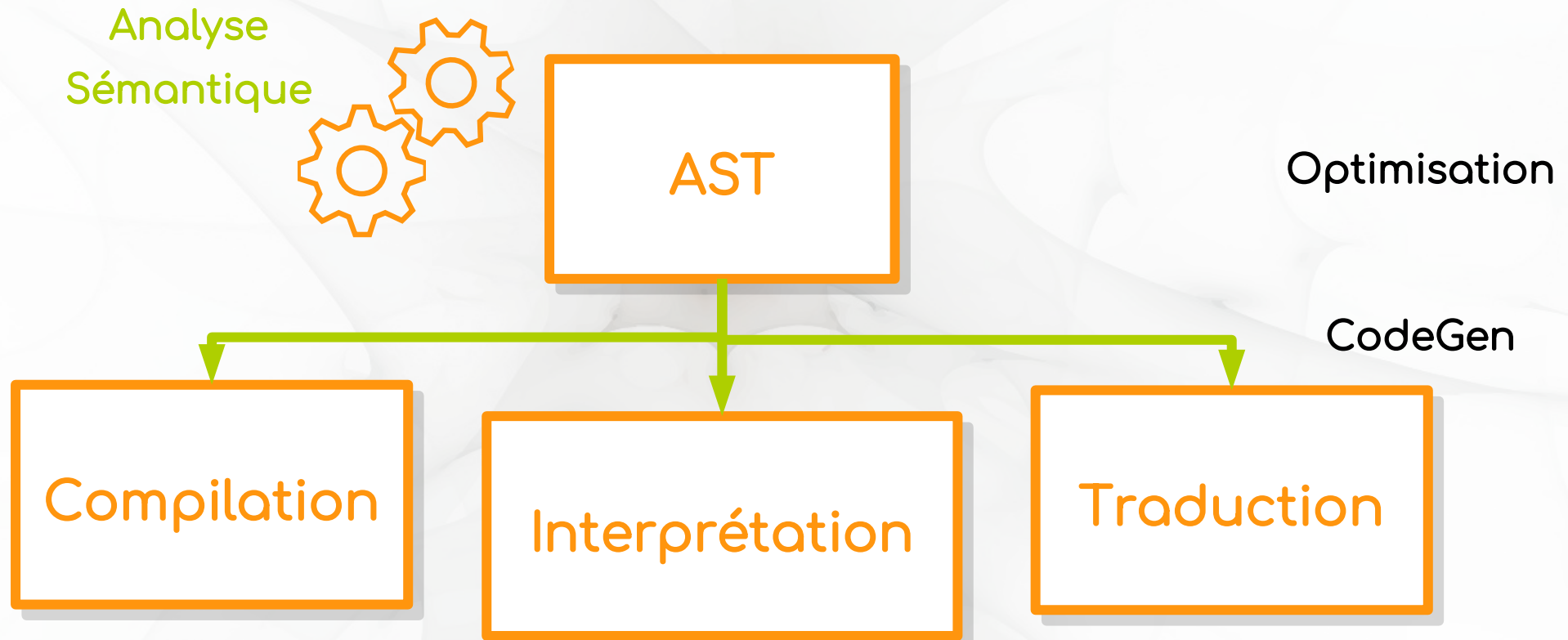
$(3 * 2) + 5$



Pourquoi un AST ?

- ✓ L'interprétation directe oblige de ré-analyser lors de chaque itération
- ✓ Offre l'opportunité d'optimisations (analyse sémantique)
- ✓ Offre l'opportunité de vérifications
- ✓ Peut être réutilisé
- ✓ Plusieurs cibles
 - ✓ Compilation
 - ✓ Interprétation
 - ✓ Traduction

Et après



Merci de votre attention

