Langages et Compilation

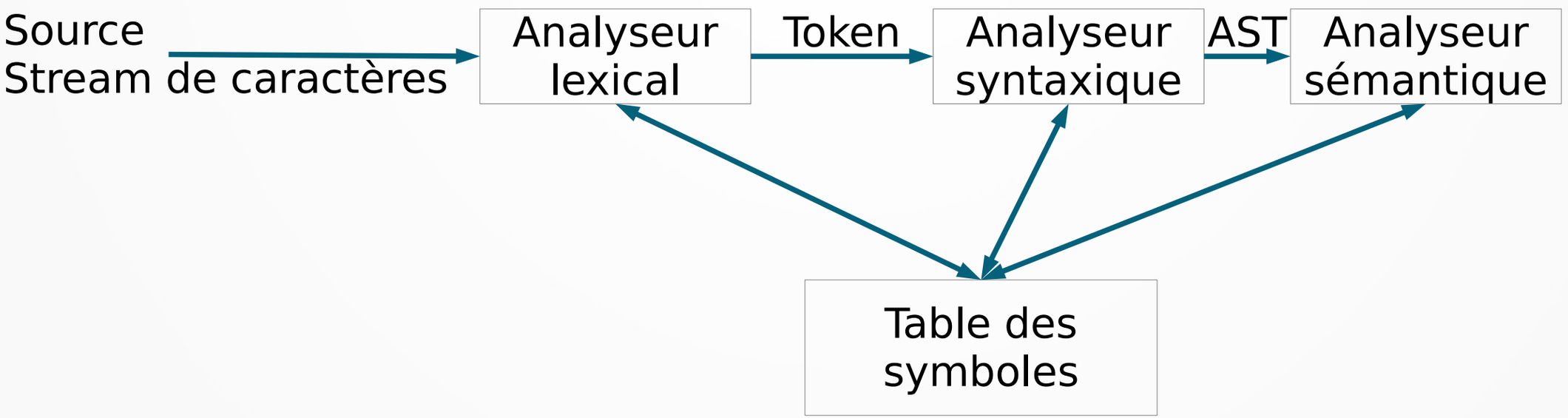
L3: Analyse Syntaxique

T. Goubier

L3A

2019/2020

L3: Analyse Syntaxique



L3: Analyse Syntaxique

- Analyse lexicale → convertir du texte en mots
- Analyse syntaxique → organiser les mots en phrases
- Approche hiérarchique
  - Définir des documents comme des phrases composées de mots
  - Correspondant aux langages de programmation
    - Un fichier contient des déclarations et définitions de fonction
    - Une définition de fonction ce sont un jeu de paramètres et un corps (body)
    - Le body contient des déclarations et des statements
    - Un statement peut contenir un body

L3: Analyse Syntaxique

- Principe d'une grammaire
  - Définir de manière récursive comment un texte se décompose ou se structure en mots (i.e. des phrases se composant de mots)
  - Principe important:
    - la récursivité permet de décrire de manière très compacte une structure complexe
    - Mais elle rend l'analyse complexe
  - Mode de description: règles de production
    - stmt → if (expr) stmt else stmt
      - Un statement peut être un if/else contenant une expression et deux statements

L3: Analyse Syntaxique

- Principe d’une grammaire
  - La méthode la plus générale et puissante pour décrire un ensemble
- Hiérarchie de Noam Chomsky (1956)
  - Type 0 : presque aucune restriction
    - $\alpha \rightarrow \beta$ (où α contient au moins un non terminal)
  - Type 1 : contextuelle
    - $\alpha\ A\ \beta \rightarrow \alpha\ \gamma\ \beta$
  - Type 2 : hors contexte
    - $A \rightarrow \alpha$
  - Type 3 : expressions régulières

L3: Analyse Syntaxique

- Principe d’une grammaire
  - La méthode la plus générale et puissante pour décrire un ensemble
- Hiérarchie de Noam Chomsky (1956)
  - Type 0 : presque aucune restriction
    - $\alpha \rightarrow \beta$ (où α contient au moins un non terminal)
  - Type 1 : contextuelle
    - $\alpha\ A\ \beta \rightarrow \alpha\ \gamma\ \beta$
  - Type 2 : hors contexte
    - $A \rightarrow \alpha$
  - Type 3 : expressions régulières

L3: Analyse Syntaxique

- Utilisation
  - Grammaires pour les langues naturelles
    - L'origine des grammaires
  - Grammaires pour les langages de programmation
    - Beaucoup de progrès sur les implémentations et normalisations
  - Grammaires pour les formats de fichiers / données
    - Correctement spécifier les entrées / sorties et les échanges entre programmes
- Histoire
  - La plus vieille grammaire hors contexte connue est celle du Sanskrit
    - Panini, *Ashtadhyayi*, ~IV^{eme} siècle av. J.C.
    - ~4000 règles.

L3: Analyse Syntaxique

- Importance ‘Calcul’
  - Équivalence entre un type de grammaire et une structure de calcul
  - Type 3 → automates à états finis

    → Programmes avec des boucles / sans récursion
  - Type 2 → automates à piles non-déterministes
    - Avec un sous-ensemble ‘déterministes’

      → Programmes avec récursion
  - Type 1 et Type 0 → machine de turing

L3: Analyse Syntaxique

- Une grammaire (dite hors contexte / context free)
  - Un ensemble de terminaux
    - symboles correspondant aux mots et ponctuations de notre langage
  - Un ensemble de non-terminaux
    - Chaque non-terminal correspond à une phrase de notre langage
  - Un symbole de départ
    - Un non-terminal qui correspond à un document (un ensemble de phrases)
  - Des productions, règles de production
    - non-terminal → [terminal | non-terminal] *
    - Un non-terminal peut se décomposer en une séquence de terminaux / non-terminaux
    - Il peut aussi être vide

L3: Analyse Syntaxique

- Exemple de grammaire
  - Pour exprimer toutes les expressions
    - 3 + 4
    - 3 + 4 * 5
    - (7 + 3) * 6
    - ...
  - Sous une forme récursive
    - Une expression est
      - une addition de deux nombres
      - une multiplication de deux nombres
      - une addition d'un nombre avec une multiplication de deux nombres
      - une addition d'un nombre avec une addition de deux nombres

L3: Analyse Syntaxique

- Exemple de grammaire
  - Pour exprimer toutes les expressions
    - 3 + 4
    - 3 + 4 * 5
    - (7 + 3) * 6
    - ...
  - Sous une forme récursive
    - Une expression est
      - une addition de deux nombres → expression
      - une multiplication de deux nombres → expression
      - une addition d'un nombre avec une expression → expression
      - une addition d'un nombre avec une expression → expression

L3: Analyse Syntaxique

- Exemple de grammaire
  - Pour exprimer toutes les expressions
    - 3 + 4
    - 3 + 4 * 5
    - (7 + 3) * 6
    - ...
  - Sous une forme récursive
    - Une expression est
      - un nombre → expression
      - une addition de deux expressions → expression
      - une multiplication de deux expressions → expression
      - une addition d'une expression avec une expression → expression

L3: Analyse Syntaxique

- Grammaire

  Expression → Expression + Expression

  Expression → Expression * Expression

  Expression → (Expression)

  Expression → <number>

- Terminaux: { + , * , (,) , <number> }
- Non-terminaux: { Expression }
- Départ: Expression

L3: Analyse Syntaxique

- Grammaire

  Expression → Expression + Expression

  Expression → Expression * Expression

  Expression → (Expression)

  Expression → <number>

- Terminaux: { + , * , (,) , <number> }
- Non-terminaux: { Expression }

- Forme compacte

  E → E + E | E * E

  E → (E) | <number>

L3: Analyse Syntaxique

- Dérivation
  - Partir d'un non-terminal, appliquer une production
    - poursuivre jusqu'à n'obtenir que des symboles terminaux (ou le vide)
  - Point de départ: le non-terminal de départ
    - Poursuivre jusqu'à obtenir l'entrée visée (ex: 3 + 4)
  - Cascade:
    - si $\alpha \rightarrow \beta \rightarrow \delta$, alors on a:
      - $\alpha \Rightarrow \beta$
      - $\alpha \Rightarrow^* \delta$

L3: Analyse Syntaxique

- Point de départ: 3 + 4

  E → E + E

  → <number> + E

  → <number> + <number>

- 3 + 4 * 5

  E → E + E

  → <number> + E

  → <number> + E * E

  → <number> + <number> * E

  → <number> + <number> * <number>

L3: Analyse Syntaxique

- Dérivation à droite / à gauche
  - À droite : je dérive le non-terminal le plus à droite dans ma phrase

    E + E * E → E + E * <number>
  - À gauche : je dérive le non-terminal le plus à gauche dans ma phrase

    E + E * E → <number> + E * E
  - Chaque type de parser a tendance à faire des dérivations dans un sens ou dans un autre

L3: Analyse Syntaxique

- Arbre de dérivation
  - Construire un arbre à partir des productions
- Exemple:
  - 3 + 4

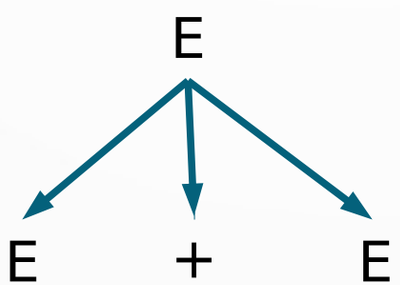
    $E \rightarrow E + E$

E

L3: Analyse Syntaxique

- Arbre de dérivation
  - Construire un arbre à partir des productions
- Exemple:
  - 3 + 4

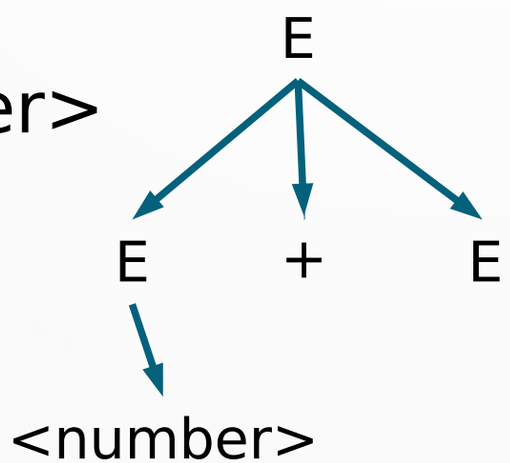
    E → <number>



L3: Analyse Syntaxique

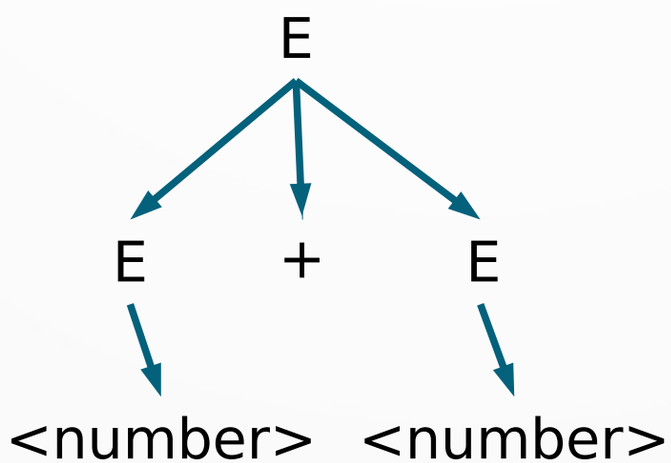
- Arbre de dérivation
  - Construire un arbre à partir des productions
- Exemple:
  - 3 + 4

    E → <number>



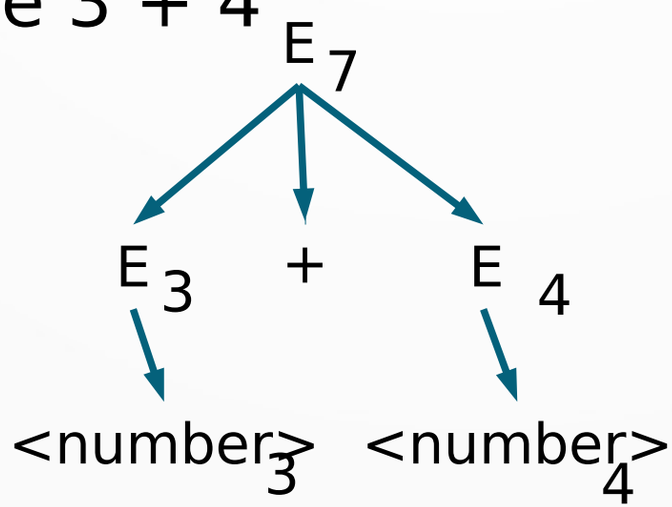
L3: Analyse Syntaxique

- Arbre de dérivation
  - Construire un arbre à partir des productions
- Exemple:
  - 3 + 4



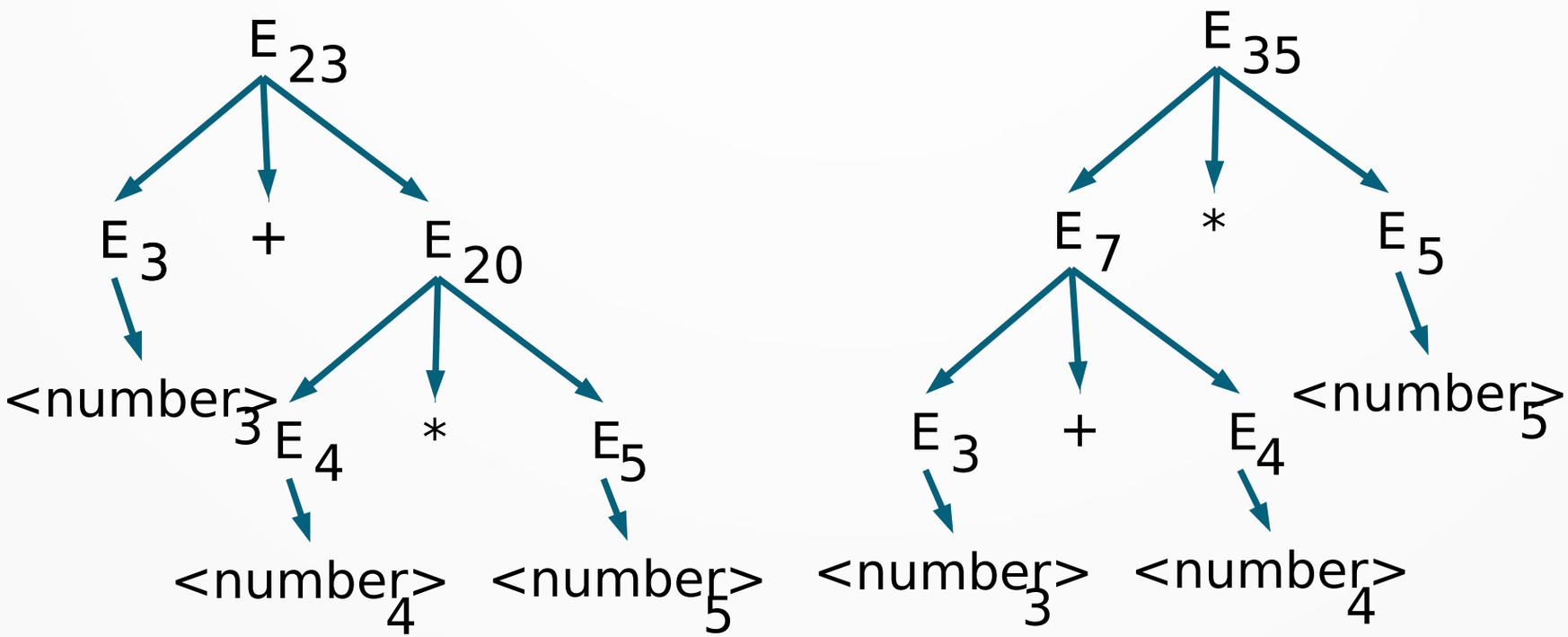
L3: Analyse Syntaxique

- Arbre de dérivation
  - Construire un arbre à partir des productions
- Utilisation
  - évaluation de 3 + 4



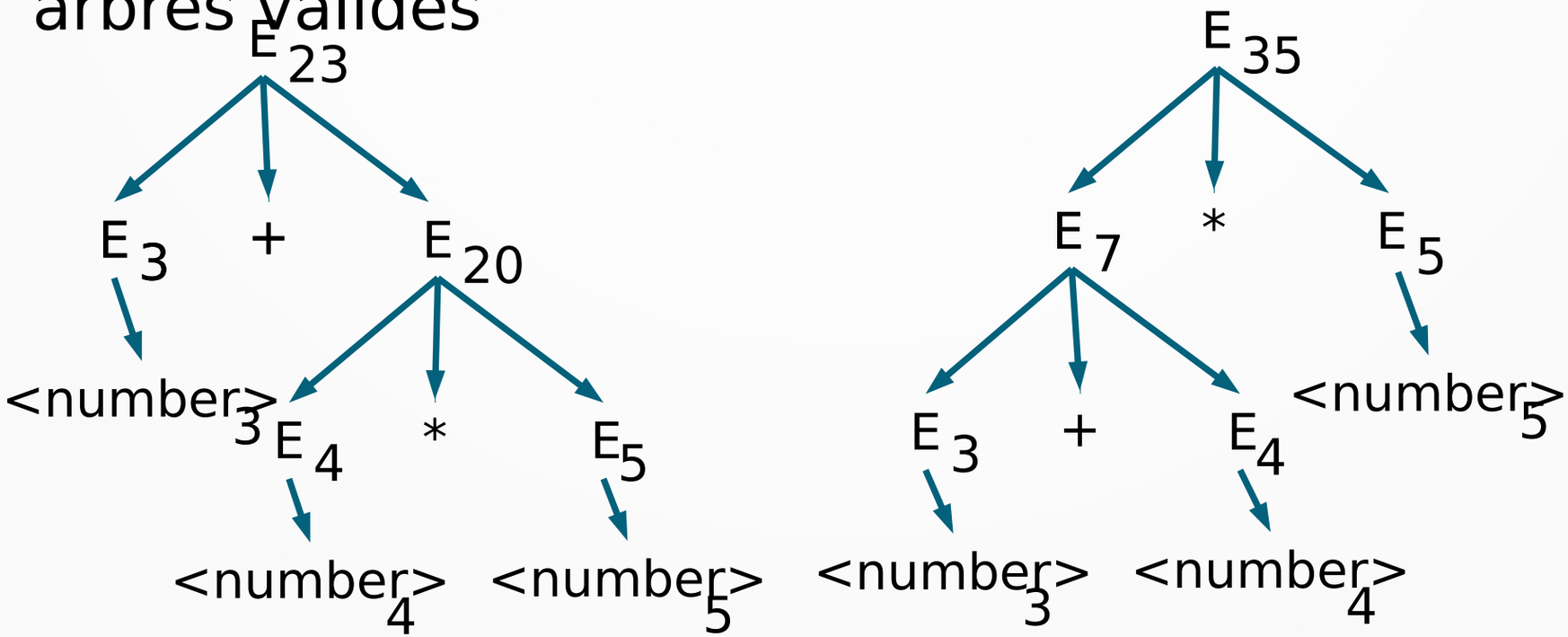
L3: Analyse Syntaxique

- Ambiguïté
  - 3 + 4 * 5 ... E → E + E ou E → E * E.



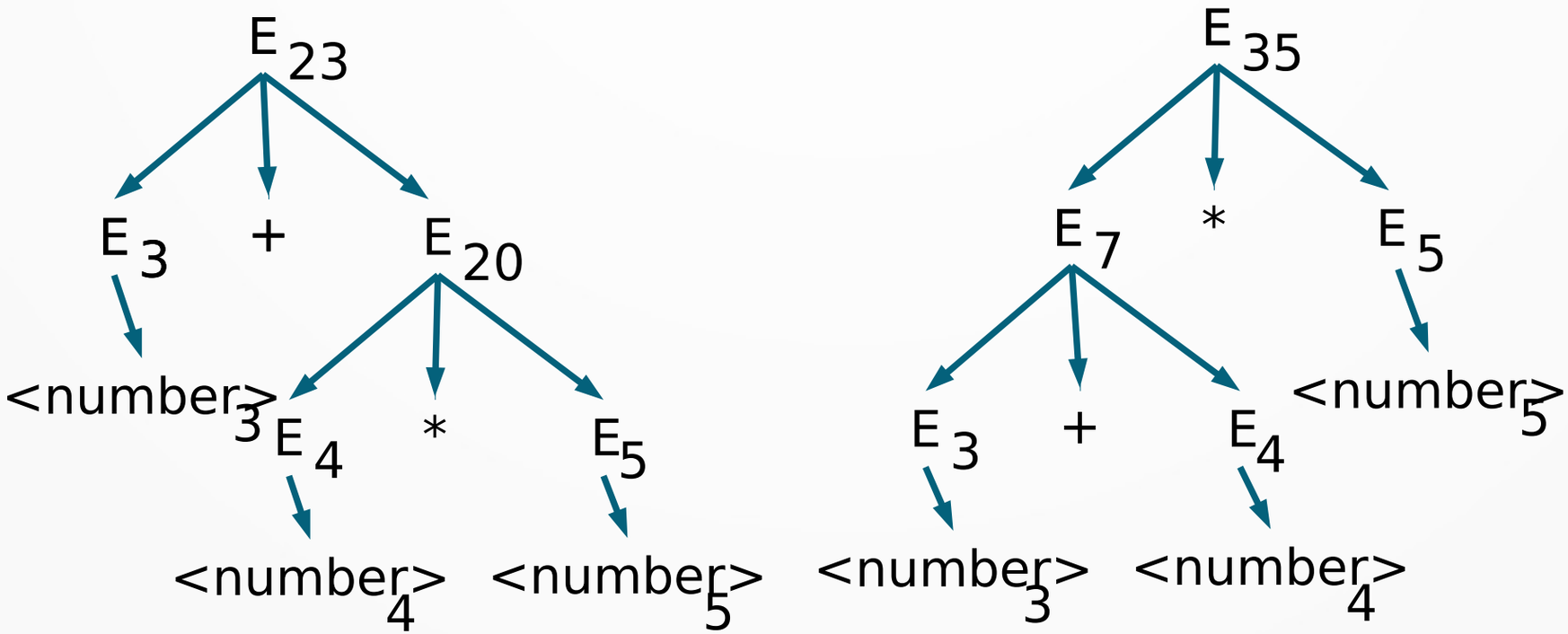
L3: Analyse Syntaxique

- Ambiguïté
  - Une grammaire est ambiguë si elle autorise plusieurs arbres valides



L3: Analyse Syntaxique

- Ambiguïté
  - Ici, ça correspond à un non-respect de la précédence



L3: Analyse Syntaxique

- Ambiguïté:
  - Écrire une grammaire non ambiguë

    E → E + T | T

    T → T * F | F

    F → (E) | <number>
    - On décompose E (Expression) en T (Term) et F (Factor).
  - Cette grammaire assure la précédence des opérateurs
  - Exercice : vérifier que c’est le cas sur les expressions précédentes

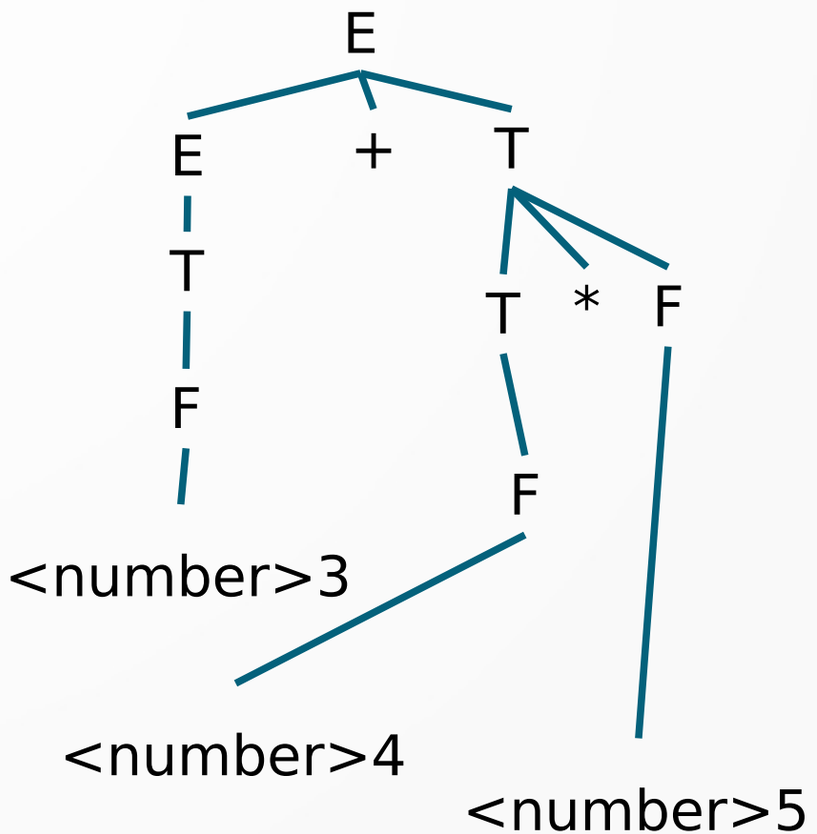
L3: Analyse Syntaxique

- Ambiguïté:
  - Écrire une grammaire non ambiguë

    E → E + T | T

    T → T * F | F

    F → (E) | <number>



L3: Analyse Syntaxique

- Ambiguïté:
  - Dans les langages de programmation
  - if/else en C:

    stmt → if (expr) stmt else stmt

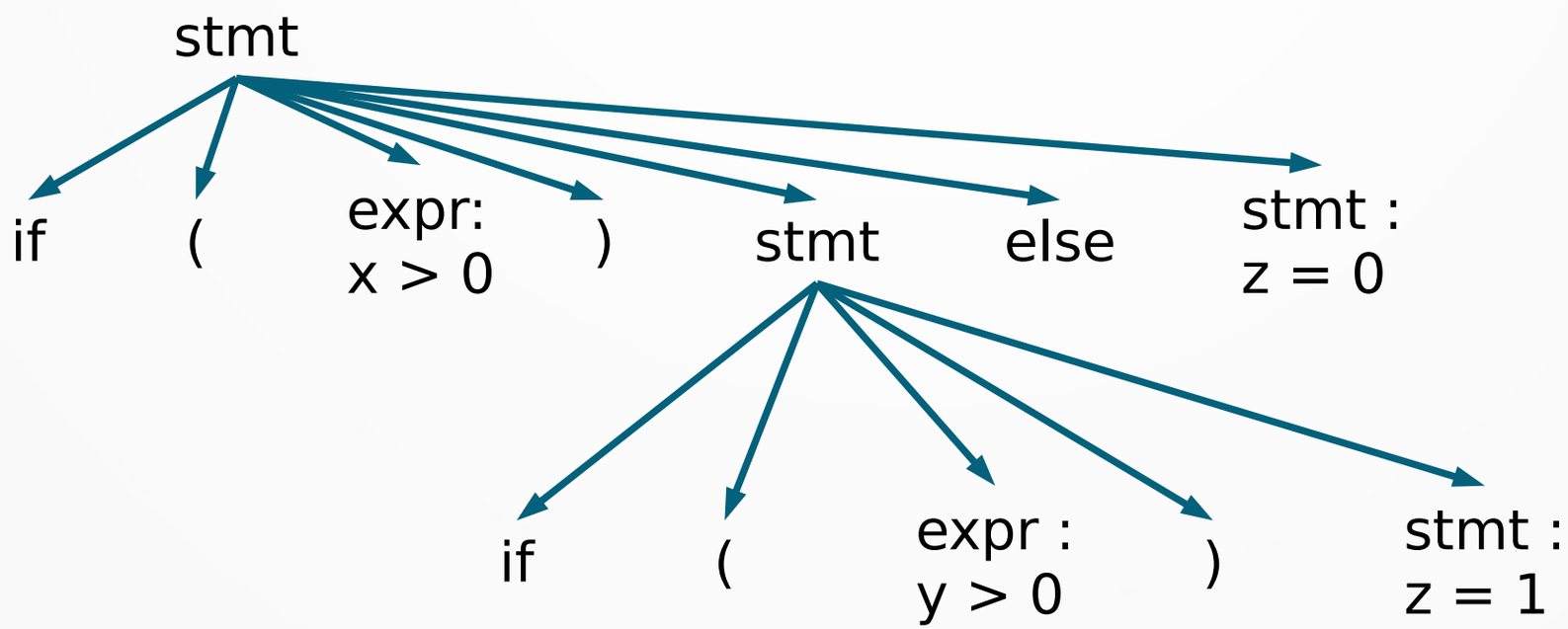
    stmt → if (expr) stmt
  - Construire l'arbre de dérivation de:

```
if (x > 0)
    if (y > 0)
        z = 1
    else
        z = 0
```

L3: Analyse Syntaxique

- if/else en C

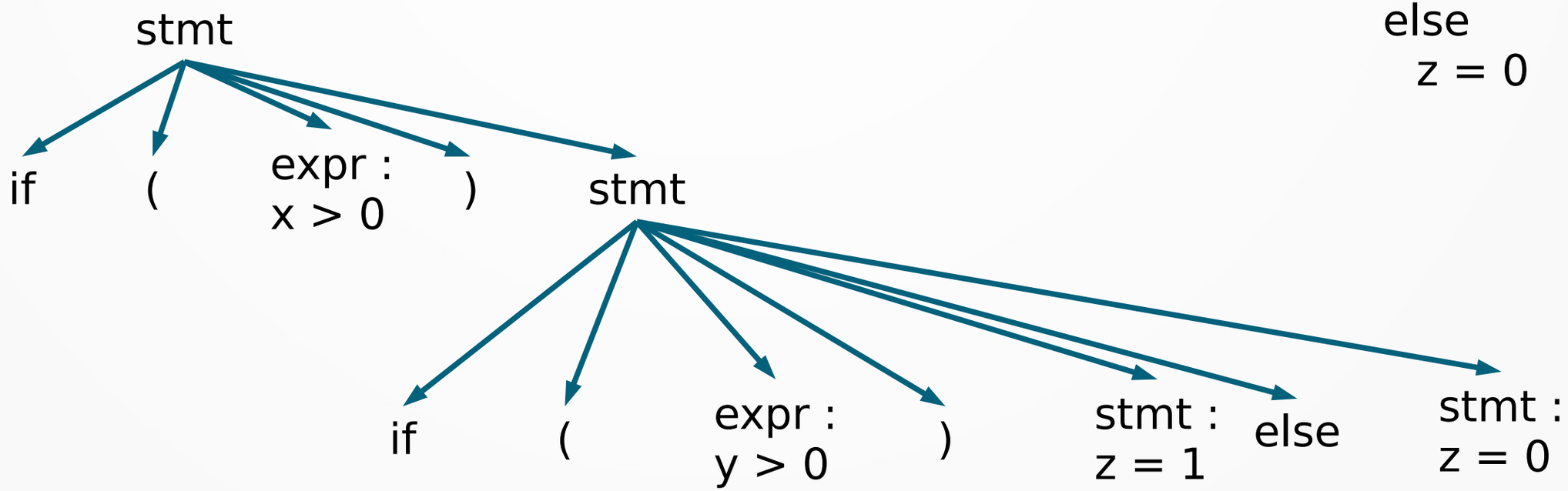
```
if (x > 0)
  if (y > 0)
    z = 1
  else
    z = 0
```



L3: Analyse Syntaxique

- if/else en C (2)

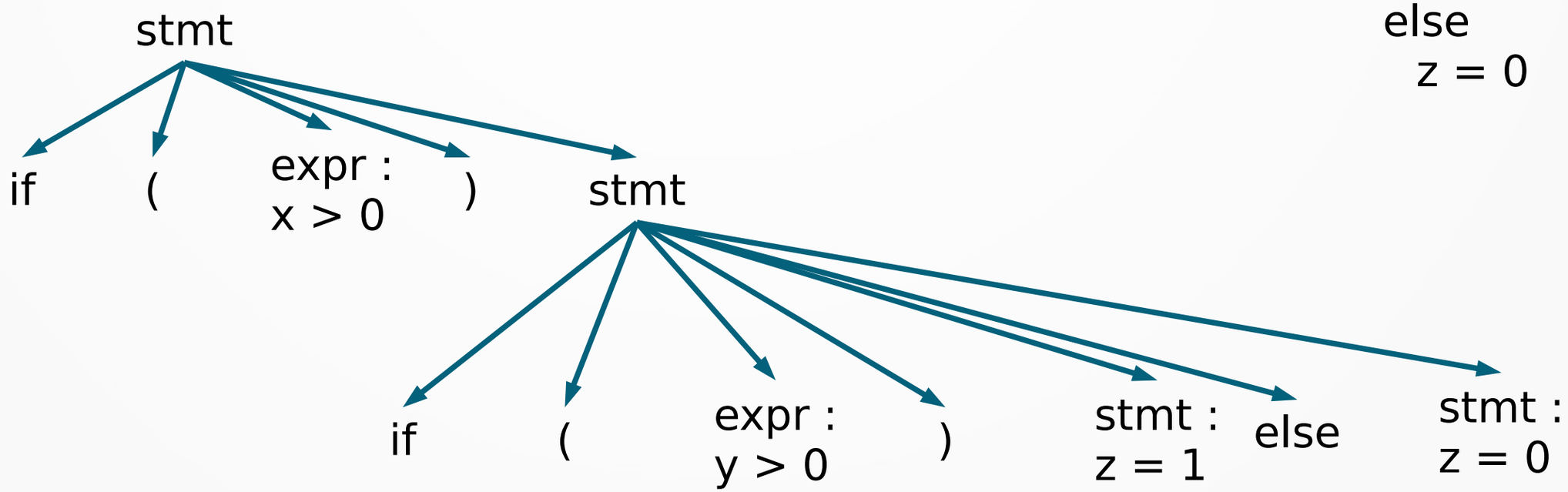
```
if (x > 0)
  if (y > 0)
    z = 1
  else
    z = 0
```



L3: Analyse Syntaxique

- if/else en C (2)
  - Arbre de dérivation correct:

```
if (x > 0)
  if (y > 0)
    z = 1
  else
    z = 0
```

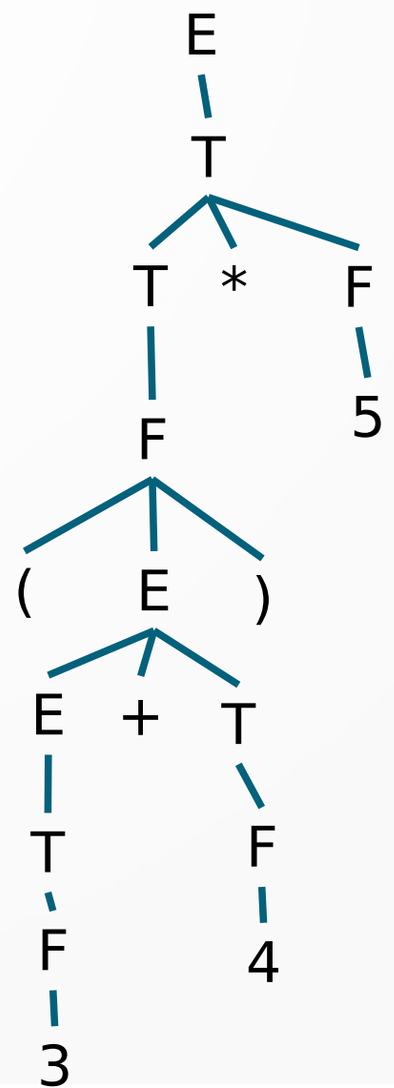


  - On ne corrige pas dans la grammaire
  - On repose sur les particularités du parser C pour résoudre ça

L3: Analyse Syntaxique

- Exercices
  - Bien maîtriser les arbres de dérivation
    - C’est essentiel pour comprendre les grammaires
  - Faire les arbres pour
    - (3) + 4 * 5
    - (3 + 4) * 5
    - 3 * 4 + (5)
    - 3 * 4 * 5

E → E + T | T
T → T * F | F
F → (E) | <number>



L3: Analyse Syntaxique

- Grammaires et expressions régulières
  - Toute expression régulière peut être représentée par une grammaire
  - L'inverse n'est pas vrai
- Exemples
  - S → (S) S | ε
    - Le langage L des parenthèses allant par paires (par ex: (()())())
  - L = {$a^n b^n$ | n >= 1 }
    - S → a S b | ε

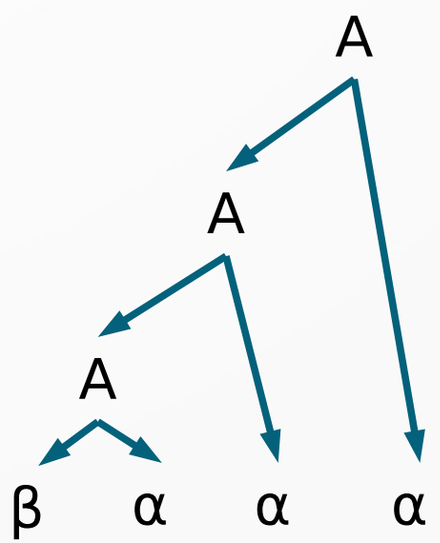
L3: Analyse Syntaxique

- Récursivité à gauche:
  - Certains parseurs, les plus simples, ne la supportent pas
- Définition
  - Une grammaire est récursive à gauche si, pour un non-terminal A, Il existe une dérivation $A \Rightarrow^{+} A\,\alpha$ pour une chaîne α .
  - La récursivité à gauche est directe si nous avons une production:
    - $A \rightarrow A\,\alpha$
  - Nous pouvons l'éliminer

L3: Analyse Syntaxique

- Élimination de la récursivité à gauche
  - Simple:

    $A \rightarrow A\ \alpha \mid \beta$
  - Construction d'un arbre en récursion à gauche
  - Commence toujours par β

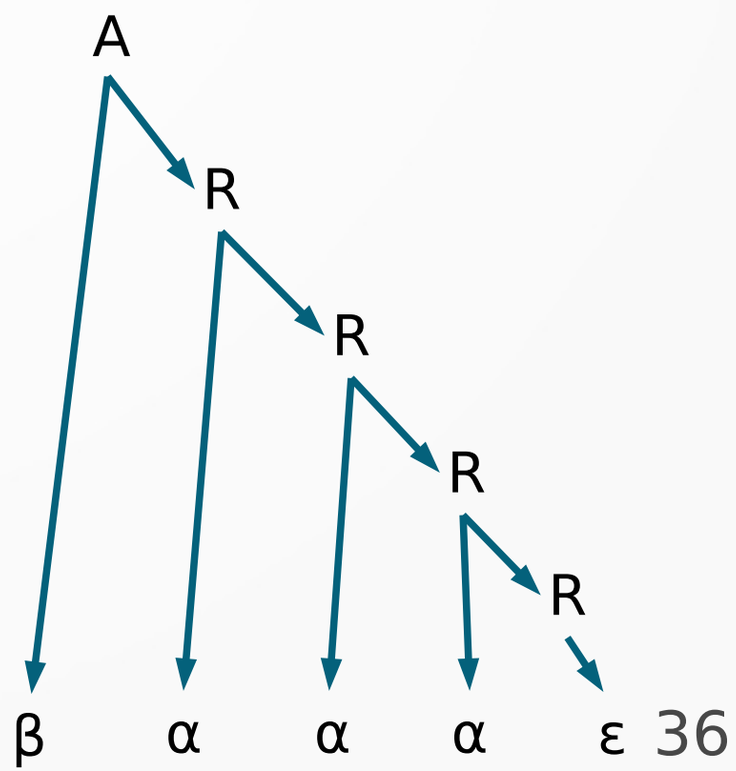


L3: Analyse Syntaxique

- Élimination de la récursivité à gauche
  - Remplacement par une construction par la droite

    $A \rightarrow \beta\ R$

    $R \rightarrow \alpha\ R \mid \varepsilon$



L3: Analyse Syntaxique

- Élimination de la récursivité à gauche
  - Sur la grammaire avec précédence:

    E → E + T | T

    T → T * F | F

    F → (E) | <number>
  - Règle:

    A → A α | β
    - devient

    A → β R

    R → α R | ε

    E → E + T | T
  - devient

    E → T E’

    E’ → + T E’ | ε

    T → T * F | F
  - devient:

    T → F T’

    T’ → * F T’ | ε

L3: Analyse Syntaxique

- Élimination de la récursivité à gauche
  - Sur la grammaire avec précédence:

    E → E + T | T

    T → T * F | F

    F → (E) | <number>
  - Règle:

    A → A α | β
    - devient

    A → β R

    R → α R | ε

E → T E’

E’ → + T E’ | ε

T → F T’

T’ → * F T’ | ε

F → (E) | <number>

L3: Analyse Syntaxique

- Factorisation à gauche
  - Éviter les productions qui commencent par le même préfixe
  - Difficile de les distinguer:

    stmt → if (expr) stmt else stmt

    stmt → if (expr) stmt
  - Factoriser en créant une règle pour le préfixe commun

    stmt → if (expr) stmt if_else

    if_else → else stmt | ε

L3: Analyse Syntaxique

- Factorisation à gauche
  - Définition:
  - Si

    $A \rightarrow \alpha\ \beta_1 \mid \alpha\ \beta_2$
  - Alors on remplace par:

    $A \rightarrow \alpha\ A'$

    $A' \rightarrow \beta_1 \mid \beta_2$

L3: Analyse Syntaxique

- Analyse syntaxique top-down
  - La plus simple: consiste à construire un arbre de dérivation à gauche.
  - Se code à la main comme un ensemble de fonctions récursives
  - Ne supporte pas la récursivité à gauche
  - Restreint à une classe de grammaire appelée LL(k)

L3: Analyse Syntaxique

ast A() {

choisir une production, $A \rightarrow X_1 X_2 \ldots X_k$

pour i de 1 à k {

si (X_i est un non-terminal)

appelle procédure X_i ()

sinon si (X_i est égal au token courant *a*)

ajoute *a* aux tokens reconnus, et demande le token suivant

sinon

erreur de syntaxe

}

L3: Analyse Syntaxique

- Construction d'arbre
  - Chacune des fonctions retourne l'arborescence qu'elle a créée
- Exécution
  - Chacune des fonctions retourne la valeur qu'elle a calculée
- Problème avec ce code
  - le choix de la production $A \rightarrow X_1 X_2 \ldots$ est non déterministe
- Solution possible
  - Savoir avec quel terminal la séquence $X_1 X_2 \ldots$ peut commencer.
  - Faire du back-tracking (retourner en arrière si la production est erronée)

L3: Analyse Syntaxique

- Exemple avec E’→ +TE’ | ε

```
ast E’() {
   if (current token is +) {
         add +
         get next token
         call T()
         call E’()
      }
   else
      return ()
}
```

L3: Analyse Syntaxique

- Exemple avec E→ E+T | ε

```
ast E() {
   if ( ? test pour première production) {
         call E()
         get next token
         if ( current token!= +) syntax error
         call T()
      }
   else if ( ? test pour deuxième )  {
      call T()
   }
}
```

L3: Analyse Syntaxique

- Classe de grammaire:
- LL(k)
  - premier L : Left to right (traite l'entrée de gauche à droite)
  - deuxième L : Leftmost (dérivation à gauche)
  - k : nombre de tokens de look-ahead
- Les exemples de code marchent avec un look-ahead de 1
  - LL(1)
- Une grammaire LL(1) n'a pas besoin de backtracking

L3: Analyse Syntaxique

- Analyse d'une grammaire:
  - Les ensembles FIRST et FOLLOW
  - FIRST :
    - pour un non terminal A, FIRST est l'ensemble des terminaux qui peuvent être en première position dans une dérivation de A
  - FOLLOW :
    - pour un non-terminal A, FOLLOW est l'ensemble des terminaux qui peuvent être juste après A dans une dérivation.
      - Spécial: le terminal $ représente la fin de l'entrée

L3: Analyse Syntaxique

- Exemples:
- E’ → + T E’ | ε
  - E’ peut commencer par +, ou être vide
    - Donc FIRST(E’) = { +, ε }
- F → (E) | <number>
- E → TE’
  - Dans notre grammaire, E est le symbole de départ, donc E peut être suivi par le terminal fin de l’entrée.
  - Ensuite, F → (E) nous dit que E peut être suivi par)
  - FOLLOW(E) = { $,) }

L3: Analyse Syntaxique

- Algorithme pour FIRST:
  - si X est un terminal, FIRST(X) = X
  - si X → ε, alors ε est dans FIRST(X)
  - si X → X_1 X_2 ... alors tous les éléments de FIRST(X_1) (sauf ε) sont dans FIRST(X)
    - Si ε est dans FIRST(X_1), alors tous les éléments de FIRST(X_2) sauf ε sont dans FIRST(X)
    - si ε est dans tous les FIRST(X_i), alors ε est dans FIRST(X)

L3: Analyse Syntaxique

- Algorithme pour FOLLOW:
  - Mettre $ dans FOLLOW(S) (S : symbole de départ)
  - Si on a A → α B β, alors tout ce qui est dans FIRST(β) est dans FOLLOW(B)
  - Si on a A → α B ou A → α B β avec ε dans FIRST(β), alors tout ce qui est dans FOLLOW(A) est dans FOLLOW(B)
- Recommandation:
  - Attention à l'ordre dans lequel on calcule les FOLLOW.

L3: Analyse Syntaxique

- Sur notre grammaire

  E → T E’

  E’ → + T E’ | ε

  T → F T’

  T’ → * F T’ | ε

  F → (E) | <number>

- FIRST(F) = { (, <number> }
- FIRST(T) = FIRST(F)
- FIRST(E) = FIRST(T)
- FIRST(T’) = { * , ε }
- FIRST(E’) = { + , ε }

L3: Analyse Syntaxique

- Sur notre grammaire

  E → T E’

  E’ → + T E’ | ε

  T → F T’

  T’ → * F T’ | ε

  F → (E) | <number>

- FOLLOW(E) = { $,) }
- FOLLOW(E’) = FOLLOW(E)
- FOLLOW(T) = { +,), $ }
- FOLLOW(T’) = FOLLOW(T)
- FOLLOW(F) = { *, +,) , $ }

L3: Analyse Syntaxique

- Définition
  - Une grammaire G est LL(1) si et seulement si, pour A → α | β deux productions distinctes de G on a:
  - Pour aucun terminal a, α et β peuvent dériver des chaînes commençant par a.
    - FIRST(α) et FIRST (β) ont une intersection vide
  - De α et β , au plus l'un des deux peut dériver la chaîne vide
    - Cf ci-dessus (si ε est dans FIRST(α), il ne peut être dans FIRST (β))
  - Si β =>* ε , alors α ne dérive aucune chaîne commençant par un terminal de FOLLOW(A) (même chose si α =>* ε)
    - FIRST(α) (FIRST(β)) et FOLLOW(A) sont disjoints

L3: Analyse Syntaxique

- Predictive parsing
  - Utiliser FIRST et FOLLOW pour construire un automate capable de parser.
  - Principe:
    - Si on a A → α β et que nous sommes dans A(), alors
      - si le token courant a est dans FIRST(α) alors on prend A → α β
        - et on exécute α ()
        - si α est un terminal (donc égal à a), alors on absorbe le token.
    - Si on a A → ε et que nous sommes dans A(), alors
      - si le token courant a est dans FOLLOW(A), alors on termine A()
        - on applique en fait A → ε

L3: Analyse Syntaxique

- Construction de la table du parseur LL(1)
  - Table M [A , *a*] (non-terminal / terminal)
  - Pour chaque production A → α
    - Pour tout *a* dans FIRST(α), ajouter A → α en M [A , *a*]
    - Si ε ∈ FIRST(α), alors ∀ *b* ∈ FOLLOW(A), ajouter A → α à M [A, *b*] (y compris M [A , $] si $ ∈ FOLLOW(A))
      - S'applique bien entendu si A → ε (i.e. FIRST(ε) = { ε })

L3: Analyse Syntaxique

- Pour notre grammaire, cela donne:
-

Non Terminal	Token / terminal					
	<number>	+	*	(	)	$
E	E → T E’			E → T E’		
E’		E’ → +T E’			E’ → ε	E’ → ε
T	T → F T’			T → F T’		
T’		T’ → ε	T’ → * FT’		T’ → ε	T’ → ε
F	F → <number>			F → (E)		

L3: Analyse Syntaxique

- Utilisation sur (3 + 4)
  - tokens: (3 + 4) $

E → T E’ #entrée (

T → F T’

F → (E)

(⇒ absorbe (, entrée 3 (<number>)

E → T E’

T → F T’

F → <number>

<number> ⇒ absorbe 3, entrée +

T’ → ε

E’ → + T E’

+ ⇒ absorbe +, entrée 4 (<number>)

T → F T’

F → <number>

<number> ⇒ absorbe 4, entrée)

T’ → ε

E’ → ε

) ⇒ absorbe), entrée $

T’ → ε

E’ → ε

- Fini !