

Introduction à la compilation

Polytech'Paris-Saclay – 4^{ème} année –

Bases théoriques des compilateurs: Language Formelles

usr.lmf.cnrs.fr/~wolff/teach-material/2025-2026/ET4-Compil/

Burkhart Wolff (& Frederic Voisin)

Bases théoriques des compilateurs: Language Formelles

Burkhart Wolff, (Frederic Voisin)

Foundations Theoretiques

- 1950 + : Concept de “grammaire” pour décrire un ensemble de “mots” c’est a dire, une “Language” (naturelle ou pas) qui peut être dérivé par la *grammaire*. Les concepts fondateurs étaient étudié par Noam Chomsky (linguiste) ...
- ... mais est devenu une branche classique de l’informatique théorique, permettant l’étude des algorithmes d’analyse (parsing) et de synthèse (pretty-printing, par exemple)
- ... est toujours de relevance pour la construction de compilateurs et la conception des languages de programmation

Foundations Theoretiques

- Définition : une *grammaire* est un quadruplet

$$G = (V_t, V_n, P, S), \quad \text{où } V_t \cap V_n = \emptyset$$

- V_t est un ensemble fini de symboles dits *terminaux*,
- V_n est un ensemble fini de symboles dits *non-terminaux*,
- P est un ensemble fini de *règles*, notées $\alpha ::= \beta$ avec $\alpha \in A^+$ et $\beta \in A^*$, où $A = V_t \cup V_n$ est l'ensemble des symboles de la grammaire. On impose que α contienne au moins un élément de V_n ,
- $S \in V_n$ est appelé «racine» (root) de la grammaire.

Foundations Theoretiques

- Terminologie: *règle* = *production*

- Notation:

$$\alpha ::= \beta_1 \mid \dots \mid \beta_n$$

est equivalent:

$$\{\alpha ::= \beta_1, \dots, \alpha ::= \beta_n\}$$

- Notation: lettres minuscules a,b,c sont terminaux, lettres minuscules u,v,w des sequences des terminaux, lettres A,B,C des non terminaux.

Foundations Theoretiques

- Definition: Formes Sequentielles s est une *forme sequentielle* est une séquence des symboles terminaux et non terminaux, i.e. $s \in (V_t \cup V_n)^*$. Une forme sequentielle s consistant uniquement des terminaux (i.e. $s \in (V_t)^*$) est nommé un *mot*.
- Definition: Le mot vide (la 'liste' vide) est dénoté par le caractère ε .
- Definition: Les *dérivations* d'une forme sequentielle dans une grammaire sur une grammaire (écrit $D(G)$) sont définit inductivement comme suivant:
 - $G = (V_t, V_n, P, S)$, alors $S \in D(G)$
 - $G = (V_t, V_n, P, S)$ et " $\alpha_1\beta\alpha_2$ " $\in D(G)$ et " $\beta ::= \gamma$ " $\in P$, alors " $\alpha_1\gamma\alpha_2$ " $\in D(G)$. On écrit aussi $\alpha_1\beta\alpha_2 \rightarrow \alpha_1\gamma\alpha_2$

Foundations Theoretiques

- Definition: La **dérivation**, notée $\rightarrow^*$, denote la fermeture réflexive et transitive de la relation $\rightarrow$.
- Definition: Le *language généré* par une grammaire G (écrit $L(G)$) est le sous-ensemble de tous les *mots* dans $D(G)$.

$$L(G) = \{ w \in V_t^* \mid S \rightarrow^* w \}$$

Examples

- $G1 = (\{0,1\}, \{S\}, \{ S ::= 0, S ::= 1 \}, S)$
- $G2 = (\{a,b\}, \{S,T\}, \{ S ::= a S, S ::= T, T ::= b T, T ::= \varepsilon \}, S)$
- $G2' = (\{a,b\}, \{S,T\}, \{ S ::= a S, S ::= T, T ::= T b, T ::= \varepsilon \}, S)$
- $G3 = (\{ (,), [,] \}, \{S\}, \{ S ::= (S) S, S ::= [S] S, S ::= \varepsilon \}, S)$
- $G4 = (\{a,b\}, \{S\}, \{ S ::= a S b, S ::= \varepsilon \}, S)$

Foundations Theoretiques

- Noam Chomsky a introduit une classification des langages, suivant la structure des règles. Il se trouve que ses classes correspondent a la complexité des algorithmes qui décident le problème

$$s \in L(G)$$

(algorithme d'appartenance (word decision problem)).

Foundations : Chomsky Hierarchy

- Grammaires de “type 0” :
aucune restriction sur la forme des productions.
- Cela correspond à la classe des langages dits “ récursivement énumérables ”, pour lesquels on dispose d’un algorithme pour engendrer les mots du langage.
- Ceci ne veut pas dire qu’on dispose d’un algorithme pour décider si un mot appartient au langage : si le langage a un nombre infini de mots, on ne peut pas se servir de cet énumérateur pour vérifier si le mot cible est ou non dans le langage !

Foundations : Chomsky Hierarchy

- Grammaires de “type 1” :
il existe deux caractérisations équivalentes :
 - « Grammaires monotones » : toutes les règles sont “ croissantes ”, de la forme $\alpha ::= \beta$ avec $|\alpha| \leq |\beta|$, en notant $|\cdot|$ l’opérateur qui renvoie la taille d’une chaîne.
 - « Grammaires avec contexte » : les règles sont de la forme $\gamma X \delta ::= \gamma \beta \delta$ avec $\gamma \in (V_t \cup V_n)^*$, $\beta \in (V_t \cup V_n)^+$ et X un non-terminal. On ne remplace X par β que dans un contexte où l’occurrence de X est encadrée par les chaînes γ et δ qui sont laissées inchangées par la règle. La partie β ne peut être vide donc les grammaires avec contexte sont des cas particuliers de grammaires monotones.

Foundations : Chomsky Hierarchy

- Grammaires de “type 2” (ou “hors contexte” ou “context-free”) : la partie gauche de chaque production est réduite à un **unique** non-terminal.

Pour ces grammaires, il existe deux algorithmes classiques (dus à Cocke-Younger-Kasami et Earley) pour décider de l'appartenance,

- complexité en temps cubique en la taille du mot à reconnaître. Pour des langages de programmation cette complexité est normalement trop élevée
- On utilisera des méthodes donc normalement des sous-classes de grammaires permettant la reconnaissance en temps linéaire.

Foundations : Chomsky Hierarchy

- Grammaires de “type 3” : c’est une grammaire hors-contexte dans laquelle chaque partie droite de règle comporte **au plus un non-terminal**.
- Si cette symbole est toujours le plus à droite (gauche) de la règle, on parle de “grammaires linéaires droites (gauches)”.
- La complexité de reconnaissance est lineare.
Des algorithmes optimisés permettent de parser des giga-octets de donnés dans qq. secondes ...
- Les langages qui peuvent être reconnus par automates à états finis (et donc décrits par des expressions régulières)

QQ Resultats théoriques

- Type de G si $L(G) = \{a^n b^n \mid n \in \mathbb{N}\}$?
- Type de G si $L(G) = \{a^n b^n c^n \mid n \in \mathbb{N}\}$?
- Type de G si $L(G) = \{a^n b^m c^n d^m \mid n, m \in \mathbb{N}\}$?
- Type de G si $L(G) = \{ww \mid w \in V_t^*\}$?
- Type de G_1, G_2 pour lequel on peut décider $L(G_1) = L(G_2)$?
- Type de G pour lequel on a un algorithme qui décide: $w \in L(G)$?

Application: Arithm. Expressions

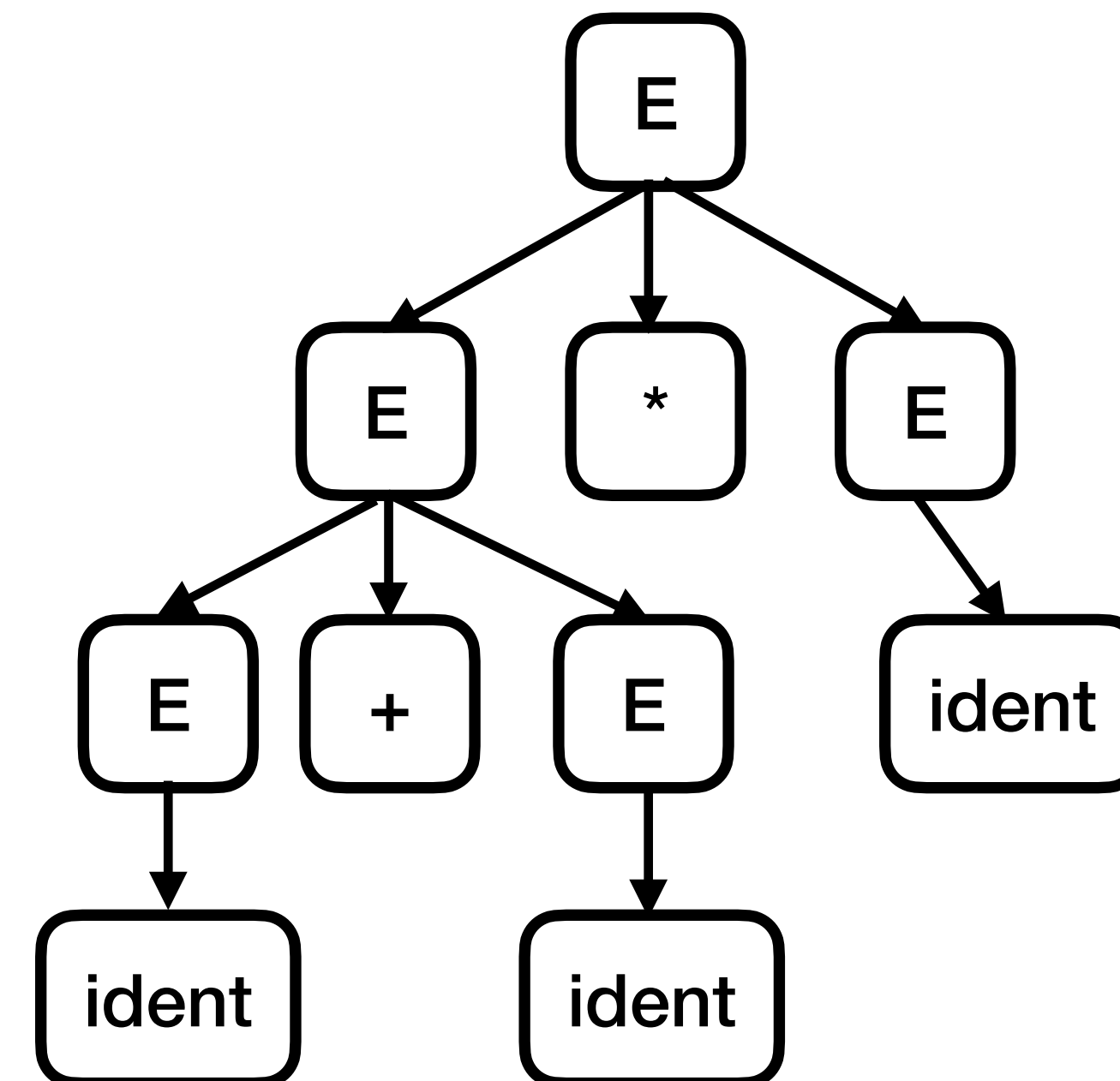
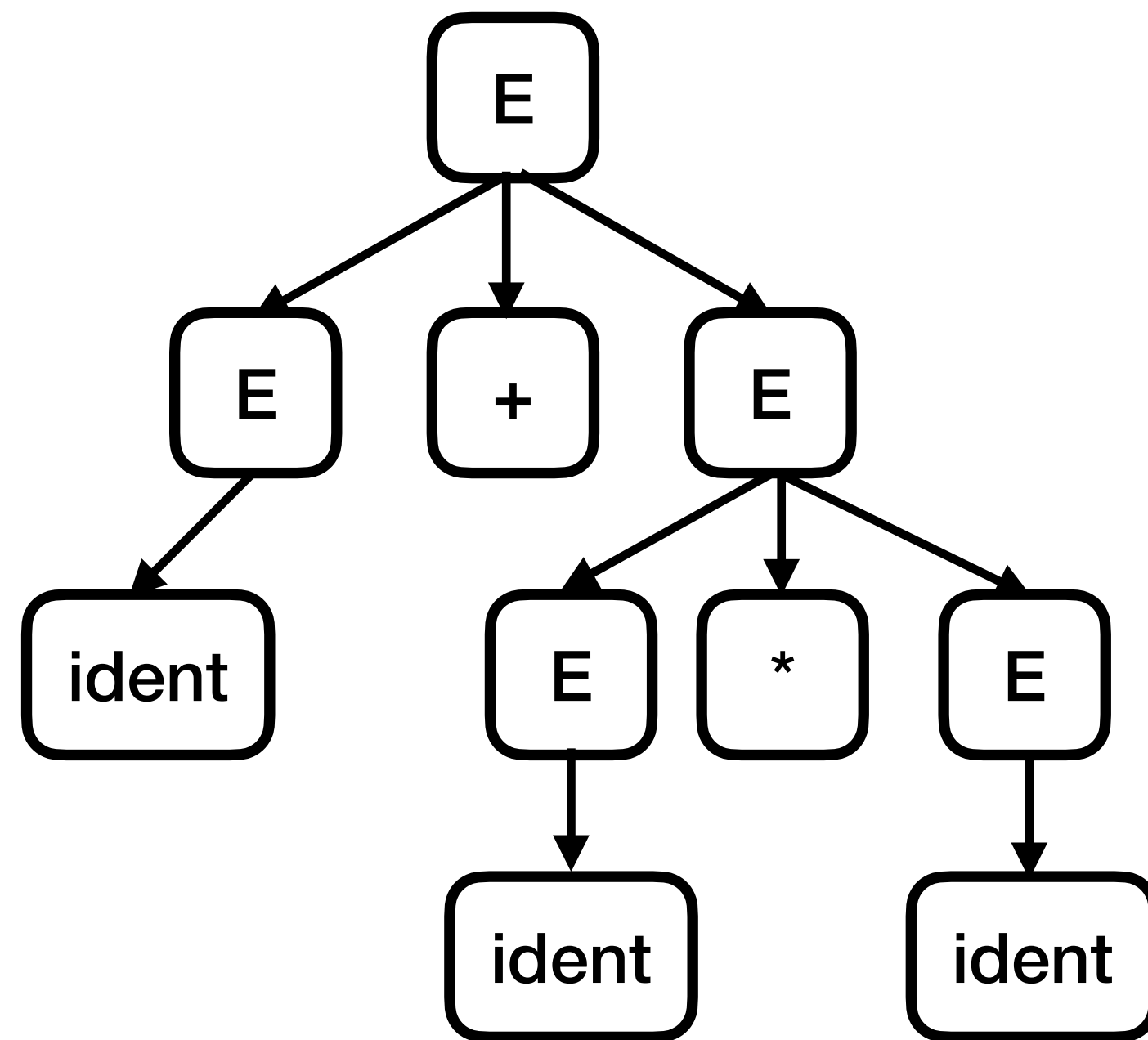
$P(G_1) :$
 $E ::= E + E$
 $| E * E$
 $| \text{ident}$
 $| (E)$

$P(G_2) :$
 $E ::= T E'$
 $E' ::= + T E' \mid \varepsilon$
 $T ::= F T'$
 $T' ::= * F T' \mid \varepsilon$
 $F ::= (E) \mid \text{ident}$

$P(G_3)$
 $E ::= E + T \mid T$
 $T ::= T * F \mid F$
 $F ::= (E) \mid \text{ident}$

Arithm. Expressions Syntax Trees

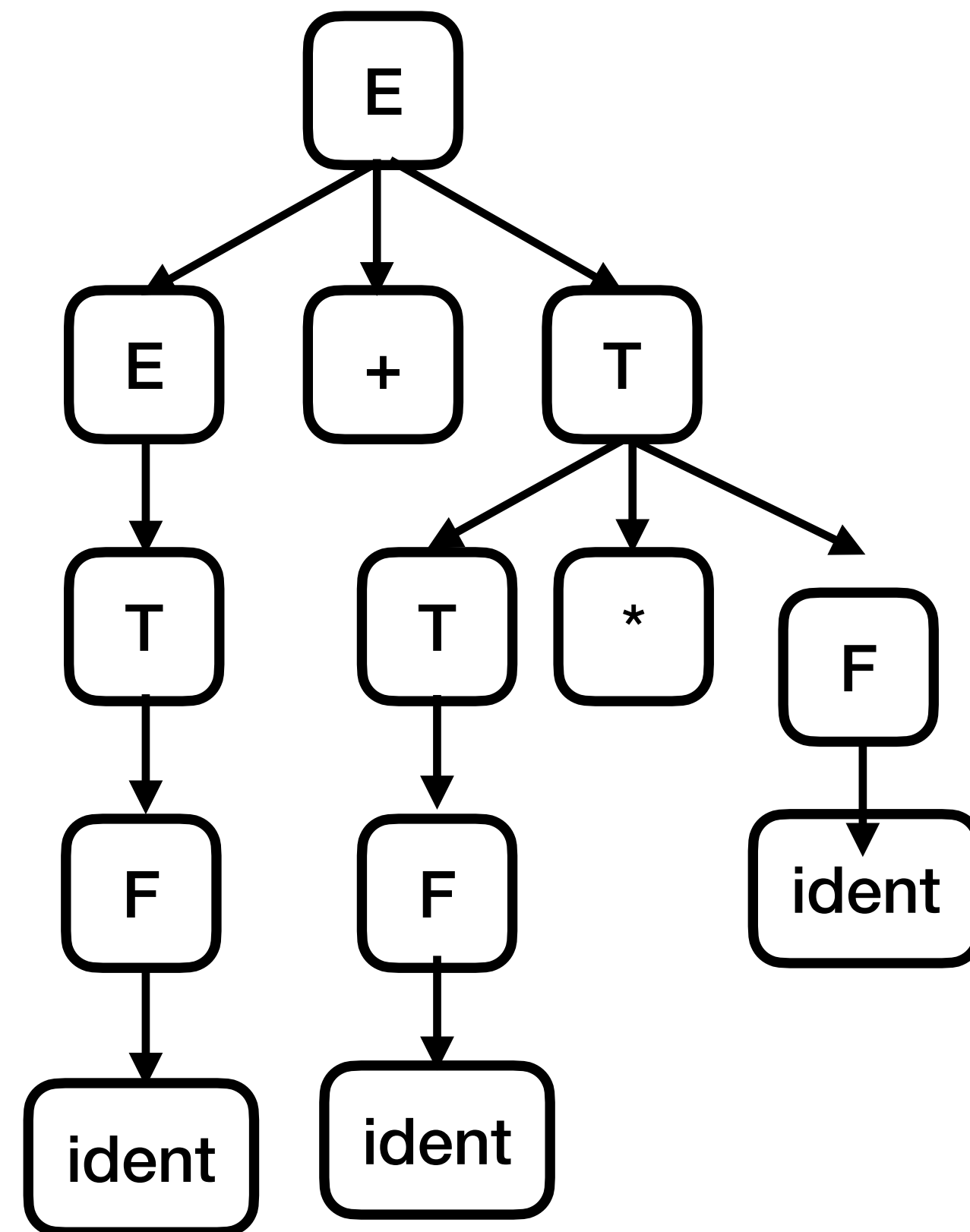
- Une autre représentation d'une dérivation: arbres syntaxiques ... pour G_1



- ... la grammaire G_1 est ambiguë - ce qu'on ne veut pas, si on a le choix, dans la conception des langages de programmation ...
(ça arrive, par contre, dans des langages naturels et, par exemple, dans des notations mathématiques ...-> systèmes de preuves comme Coq, Isabelle,...)

Arithm. Expressions Syntax Trees

- Pour G_3 , l'arbre syntaxique pour `ident + ident * ident` donne:



- ... compliqué ... mais au moins non-ambigue ...
(généralisation ? priorités ... associativités a gauche et a droite ...)

Application: Arithm. Expressions

- Une grammaire avec addition, soustraction (meme priorité), multiplication et division (priorité supérieure), exponentiation (priorité maximale):

$G3'$:

$E ::= E + T \mid E - T \mid T$

$T ::= T * F \mid T / F \mid F$

$F ::= G ^ F \mid G$

$G ::= (E) \mid \text{ident}$

$G3''$:

$E_0 ::= E_0 + E_1 \mid E_0 - E_1 \mid E_1$

$E_1 ::= E_1 * E_2 \mid E_1 / E_2 \mid E_2$

$E_2 ::= E_3 ^ E_2 \mid E_3$

$E_3 ::= (E_0) \mid \text{ident}$

Application: Arithm. Expressions

- Avec les priorités, il est relativement simple de disambiguer les opérateurs binaires associatives avec des principes assez haut niveau.

Soit $\oplus$ un opérateur de priorité p .

Soit “ $E ::= E \oplus E$ ” la production qu’on souhaite disambiguer.

- Si on veut exprimer que l’opérateur est associatif vers la gauche, on a dans la grammaire transformé

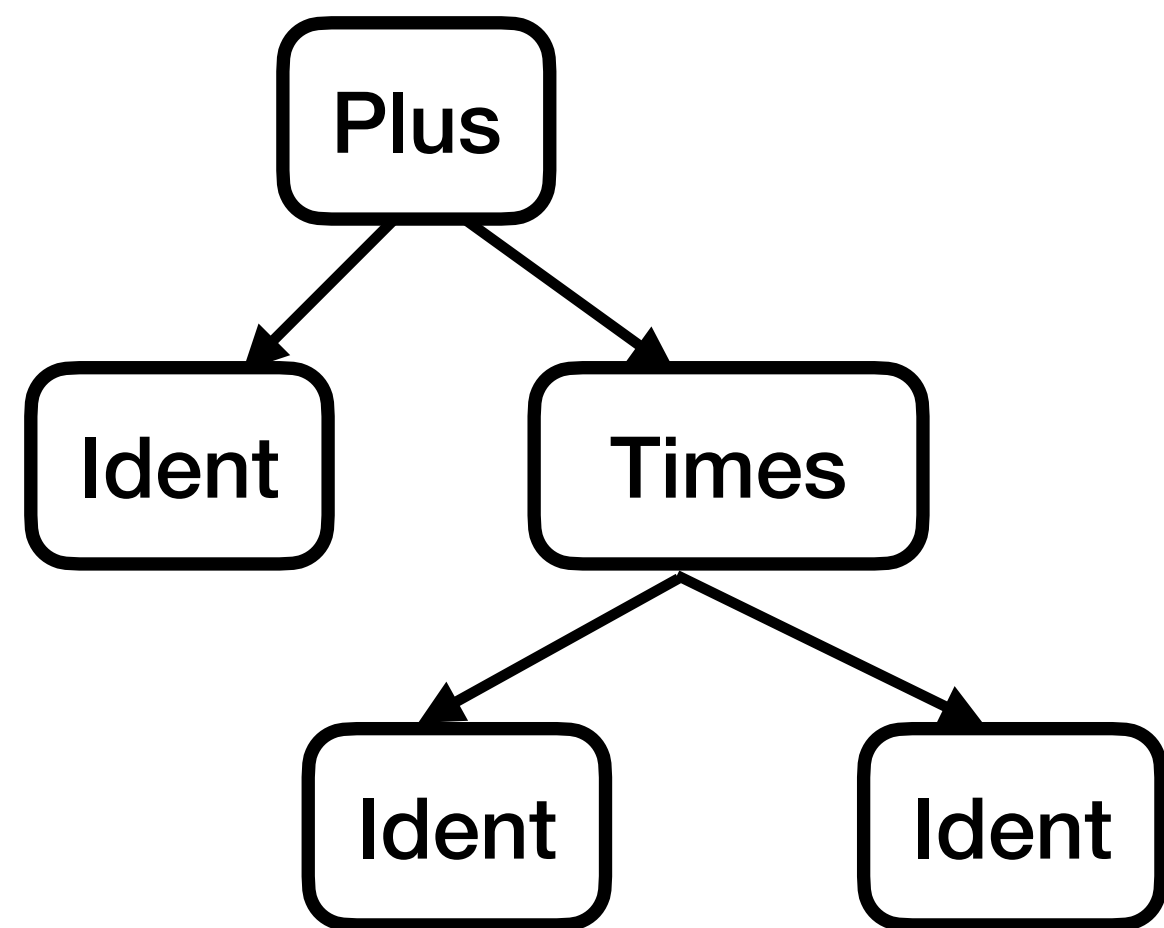
$$E_p ::= E_p \oplus E_{p+1}$$

- Si on veut exprimer que l’opérateur est associatif vers la droite, on a dans la grammaire transformé

$$E_p ::= E_{p+1} \oplus E_p$$

Application: Arithm. Expressions

- Pour traiter un arbre syntaxique, on a un intérêt de “purifier” la représentation vers une “arbre syntaxique abstraite”:
 - enlever les elements structurants comme parenthèses (et), commentaires ...
 - supprimer des non-terminaux représentant de priorité ...
 - enlever le “sucre syntaxique” comme “if_then” et “if_then_else” ...



```
type expr =  
  Id of string  
| Plus of expr* expr  
| Times of expr* expr
```

“x + y * z”



```
Plus (Id "x",  
      Times (Id "y",  
              Id "z"))
```

Conclusion

- La théorie des langages syntaxiques est une base théorique pour la construction des compilateurs
- On s'intéresse particulièrement pour les langages type 3 et des langages type 2 déterministes.
- Cela motive la séparation dans une composante “Lexer” (type 3) et “Parser” (type 2), qui génère des arbres syntaxiques
- Le génie de construction des compilateurs exige la construction des arbres syntaxique abstraites (ou d'autres langages intermédiaires dans le processus de compilation