Analyse syntaxique

Sections 4.1 à 4.4

* Contenu *

- Introduction
  - Rôle et types d'analyseurs syntaxiques
  - Gestion des erreurs
- Langages hors-contexte
  - Définitions et conventions
  - Langages non hors-contexte
  - Vérification du langage généré par une grammaire
- Forme des grammaires
  - Élimination de l'ambiguité
  - Élimination des récursions à gauche
  - Factorisation à gauche
- Analyse syntaxique descendante
  - Analyse prédictive avec récursion
  - Analyse prédictive sans récursion
    - Les ensembles FIRST et FOLLOW
    - Création de la table d'analyse
    - Grammaires LL(1)

Analyse Syntaxique:

Introduction

Introduction

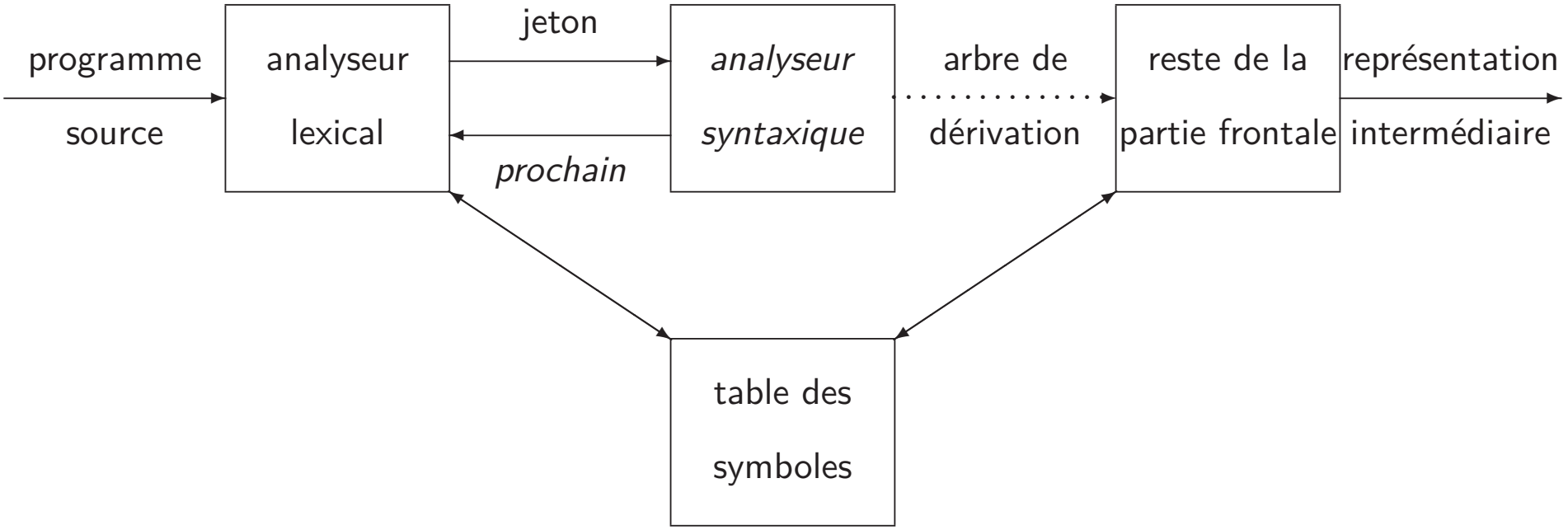
Les grammaires hors-contexte sont les outils que nous utiliserons pour spécifier la structure syntaxique des programmes.

Les grammaires hors-contexte ont plusieurs avantages:

- Elles constituent une spécification précise et claire de la syntaxe des programmes.
- On peut construire un analyseur syntaxique efficace directement à partir de certaines classes de grammaires.
- Elles offrent un cadre pour la définition de traducteurs orientés-syntaxe.
- Une définition et une implantation basées sur une grammaire hors-contexte rendent les modifications ultérieures du compilateur plus facile.

Cette portion du cours se concentre sur l'implantation à la main d'analyseurs syntaxiques et se rend jusqu'à la définition des grammaires LL(1).

Rôle des analyseurs syntaxiques



Types d'analyseurs syntaxiques

Il y trois grands types d'analyseurs syntaxiques:

- Les méthodes universelles: CYK, Earley.
  - Coûteuses en temps: $\Theta(n^3)$ et $\Omega(n) \cap O(n^3)$, respectivement.
  - S'accommodent de n'importe quelle grammaire.
- Les méthodes descendantes.
  - Rapides: temps dans $O(n)$.
  - Simples.
  - Ne s'accommodent que de certaines grammaires: celles d'une des classes LL(k).
  - Variante LL(1) étudiée dans ce cours.
- Les méthodes ascendantes.
  - Rapides: temps dans $O(n)$.
  - Plus complexes que les méthodes descendantes.
  - Ne s'accommodent que de certaines grammaires: celles d'une des classes LR(k).
  - Les classes LR(k) sont plus grandes que les classes LL(k).

Gestion et signalement des erreurs

Rappelons qu'on peut classer les erreurs en ces catégories: lexicales, syntaxiques et sémantiques.

Étant donné que peu d'erreurs peuvent être détectées au niveau lexical, l'analyseur syntaxique détecte la plupart des erreurs dans les deux autres catégories.

Objectifs du gestionnaire d'erreurs:

- Les erreurs doivent être signalées clairement (messages) et précisément (causes et coordonnées).
- Il devrait y avoir récupération après une erreur afin de continuer la compilation et la détection d'erreurs pour le reste du programme.
- La gestion des erreurs ne devrait pas ralentir la compilation des programmes corrects de façon importante.

Gestion et signalement des erreurs

Après une erreur, l'analyseur syntaxique peut s'arrêter ou sinon il peut tenter de récupérer de diverses façons:

- Mode panique: abandon des jetons jusqu'à un jeton dit *de synchronisation* (certains délimiteurs ou mots clés).
- Correction locale: modification d'un préfixe *court* de l'entrée pour permettre à l'analyse de se poursuivre.
- Correction globale: calcul d'une *plus petite distance d'édition* qui transforme le programme erroné en un programme syntaxiquement valide.
- Productions d'erreurs: utilisation de *productions d'erreur* qui captent certaines erreurs et fournissent une réaction adéquate.

Grammaires H-C versus expressions régulières

On sait que pour toute expression régulière, on peut créer une grammaire hors-contexte qui génère le même langage.

Donc, pourquoi tient-on à utiliser les expressions régulières pour définir la structure lexicale des langages de programmation?

- La structure lexicale des langages est habituellement très simple et ne nécessite pas l'usage des grammaires hors-contexte.
- Les expressions régulières sont plus compactes et plus claires.
- On peut produire des analyseurs lexicaux plus efficaces à partir d'expressions régulières qu'à partir de grammaires.
- La séparation en structure lexicale et structure syntaxique aide à modulariser le compilateur.

Analyse Syntaxique:

Langages hors-contexte

Conventions de notation

Pour minimiser le verbiage inutile, on adopte plusieurs conventions liées à la notation.

1. Les choses suivantes sont des terminaux:
   - des lettres minuscules du début de l'alphabet comme *a*, *b*, *c*;
   - des opérateurs comme $+$, $-$;
   - des signes de ponctuation;
   - les chiffres 0, . . . , 9;
   - des noms en gras comme **id**, **if**.
2. Les choses suivantes sont des non-terminaux:
   - des lettres majuscules du début de l'alphabet comme A, B, C;
   - la lettre S qui est souvent le symbole de départ;
   - des noms en minuscules et en italique comme *expr*, *stmt*.

Conventions de notation

(suite)

3. Les lettres majuscules de la fin de l'alphabet, comme X, Y et Z, représentent des symboles de la grammaire; c'est-à-dire, des terminaux et non-terminaux indifféremment.

4. Les lettres minuscules de la fin de l'alphabet, comme u, ..., z, représentent des chaînes faites de terminaux.

5. Les lettres grecques minuscules, comme α, β et γ, représentent des chaînes faites de symboles de la grammaire. Par ex., $A \to \alpha$ est la forme générale d'une production où A et α sont les membres gauche et droit, respectivement.

6. Si on a les productions $A \to \alpha_1$, ..., $A \to \alpha_k$, appelées A-productions, on peut écrire $A \to \alpha_1 \mid \ldots \mid \alpha_k$ et appeler les α_i les alternatives pour A.

7. À moins d'indications contraires, le membre gauche de la première production est le symbole de départ.

Définitions

Une *phrase* est une suite de terminaux (synonyme d'une chaîne).

Une *forme de phrase* est une suite de terminaux et de non-terminaux.

Une forme de phrase α *est dérivable en une étape* en une forme de phrase α',
qu'on dénote par $\alpha \Rightarrow \alpha'$,
si $\alpha = \beta A \gamma$, $\alpha' = \beta \delta \gamma$ et si $A \rightarrow \delta$ est une production.

On dit aussi que α_1 *est dérivable* en α_n,
qu'on dénote par $\alpha_1 \Rightarrow^* \alpha_n$,
si $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ et $n \geq 1$.
Si $n > 1$, on peut écrire $\alpha_1 \Rightarrow^+ \alpha_n$.

On a une *dérivation* de E en w si $E \Rightarrow^* w$.

Une chaîne w est *générée* par la grammaire si $S \Rightarrow^* w$ où S est le symbole de départ.

Un *arbre de dérivation* d'une chaîne w . . . (voir la définition au chapitre 2).

Définitions

(suite)

On dit que α *est dérivable à gauche d'abord en une étape* en α',
qu'on note $\alpha \underset{\mathsf{lm}}{\Rightarrow} \alpha'$,
si $\alpha = wA\beta$, $\alpha' = w\gamma\beta$ et $A \to \gamma$ est une production.

On dit aussi que α_1 *est dérivable à gauche d'abord* en α_n,
qu'on dénote par $\alpha_1 \underset{\mathsf{lm}}{\Rightarrow}^* \alpha_n$,
si $\alpha_1 \underset{\mathsf{lm}}{\Rightarrow} \alpha_2 \underset{\mathsf{lm}}{\Rightarrow} \cdots \underset{\mathsf{lm}}{\Rightarrow} \alpha_n$ et $n \geq 1$.
Si $n > 1$, on peut écrire $\alpha_1 \underset{\mathsf{lm}}{\Rightarrow}^+ \alpha_n$.

On a une *dérivation à gauche d'abord* de E en w si $E \underset{\mathsf{lm}}{\Rightarrow}^* w$.

De manière analogue, on dit que α *est dérivable à droite d'abord en une étape* en α',
qu'on note $\alpha \underset{\mathsf{rm}}{\Rightarrow} \alpha'$,
si $\alpha = \beta Aw$ et $\alpha' = \beta\gamma w$ et si $A \to \gamma$ est une production.

Les définitions de "dérivable à droite d'abord" et de "dérivation à droite d'abord" sont analogues.

Définitions

(suite)

Le *langage* généré par une grammaire G, dénoté $L(G)$, est l'ensemble des chaînes générées par G.

Un langage L est *hors-contexte* s'il existe une grammaire G dont le langage est L.

Deux grammaires sont *équivalentes* si elles génèrent le même langage.

Une grammaire est *ambiguë* s'il existe plus d'un arbre de dérivation pour un certain w.

De manière équivalente, une grammaire est *ambiguë* s'il existe plus d'une dérivation à gauche d'abord pour un certain w.

De manière équivalente, une grammaire est *ambiguë* s'il existe plus d'une dérivation à droite d'abord pour un certain w.

Note: on **ne** peut **pas** conclure qu'une grammaire est ambiguë simplement parce qu'il existe plus d'une dérivation pour un certain w.

Rappel: langages non hors-contexte

Les langages suivants ne sont pas hors-contexte:

- $L_1 = \{wcw \mid w \in \{a, b\}^*\}$
- $L_2 = \{a^n b^m c^n d^m \mid n \geq 1 \text{ et } m \geq 1\}$
- $L_3 = \{a^n b^n c^n \mid n \geq 0\}$

Pourtant, ils ressemblent beaucoup aux langages suivants qui, eux, sont hors-contexte:

- $L'_1 = \{wcw^R \mid w \in \{a, b\}^*\}$
- $L'_2 = \{a^n b^m c^m d^n \mid n \geq 1 \text{ et } m \geq 1\}$
- $L'_3 = \{a^n b^n \mid n \geq 0\}$

Vérification du langage généré

Le créateur d'un langage de programmation doit s'assurer que le langage généré par la grammaire qu'il emploie génère bien le bon langage.

Le degré de formalisme dans la vérification peut varier.

Aussi, il est rarement nécessaire de vérifier l'ensemble de la grammaire d'un coup.

Pour montrer qu'une grammaire G génère le bon langage L, il faut montrer que:

- **seulement** des chaînes appartenant à L sont générées par G (*correction*);
- **toutes** les chaînes appartenant à L sont générées par G (*exhaustivité*).

Vérification du langage généré (exemple)

Exemple 4.12: le langage des chaînes dans $\{(,)\}^*$ qui sont bien parenthésées est généré par la grammaire:

$$S \to (S)S \mid \epsilon$$

Démonstration:

1) Toute chaîne générée par S est bien balancée. Preuve par induction sur le nombre de dérivations:

- Cas de base: une dérivation de S en *une* étape est bien balancée. Preuve: La seule dérivation possible en une étape est $S \Rightarrow \epsilon$, et ϵ est une chaîne bien balancée.

- Pas d'induction: si toute dérivation en $n-1$ étapes ou moins produit une chaîne balancée, alors toute dérivation en n étapes produit aussi une chaîne balancée. Preuve: une dérivation en n étapes a nécessairement la forme $S \Rightarrow (S)S \Rightarrow^* (x)y$, où $S \Rightarrow^* x$ et $S \Rightarrow^* y$ sont des dérivations de moins de n étapes. Par hypothèse d'induction on sait donc que x et y sont bien balancées, et donc $(x)y$ est aussi bien balancée.

Vérification du langage généré (exemple)

2) Toute chaîne bien balancée est générée par S. Preuve par induction sur la longueur de la chaîne:

- Cas de base: la chaîne bien balancée de longueur 0 (laquelle est forcément ϵ) est générée par S.

- Pas d'induction: Si toutes les chaînes bien balancées de longueur inférieure à $2n$, pour $n \geq 1$, sont générées par S, alors toute chaîne w bien balancée de longueur $2n$ est aussi générée par S (notons que toutes les chaînes du langage sont de longueur paire). Preuve: w doit forcément commencer par une parenthèse ouvrante et cette parenthèse est appariée plus loin avec une parenthèse fermante correspondante; alors on peut écrire w sous la forme $w = (x)y$ où x et y sont bien balancés. Puisque x et y sont bien balancés et de longueur inférieure à $2n$, on sait par hypothèse d'induction que S peut les générer. On a donc la dérivation $S \Rightarrow (S)S \Rightarrow^* (x)S \Rightarrow^* (x)y = w$.

3) Conclusion: S génère uniquement les chaînes du langage et toutes les chaînes du langage.

Analyse Syntaxique:

Forme des grammaires

Élimination de l'ambiguïté

La grammaire décrivant un langage doit être conçue de façon à être sans ambiguïté ou bien être convertie de façon à ce qu'elle ne comporte plus d'ambiguïté.

Par exemple, la grammaire synthétique suivante génère des énoncés mais elle est ambiguë:

$$
\begin{array}{lcl}
\textit{stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\
& | & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt} \\
& | & \textbf{other}
\end{array}
$$

Toutefois, on peut la modifier par la grammaire équivalente suivante, laquelle n'est pas ambiguë:

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textit{matched\_stmt} \\
& | & \textit{open\_stmt} \\
\textit{matched\_stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{matched\_stmt} \textbf{ else } \textit{matched\_stmt} \\
& | & \textbf{other} \\
\textit{open\_stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\
& | & \textbf{if } \textit{expr} \textbf{ then } \textit{matched\_stmt} \textbf{ else } \textit{open\_stmt}
\end{array}
$$

Élimination de la récursion à gauche

Considérons la grammaire récursive à gauche suivante:

$$A \quad \rightarrow \quad A\alpha \mid \beta$$

où β ne commence pas par A.

Cette grammaire est dangereuse car elle peut mener à une boucle à l'infini. Supposons que notre procédure de décision prescrive d'utiliser la production $A \rightarrow A\alpha$ lorsqu'on veut substituer A en voyant c sur l'entrée. Ainsi, à l'étape suivante, il faudra à nouveau décider comment substituer A en voyant c sur l'entrée. Une boucle infinie s'ensuit.

Analyser A:
    si *critère* **alors** analyser A; analyser α
    sinon analyser β;

On peut éliminer la récursion à gauche en remarquant que A ne peut mener qu'à des formes de phrases du genre $\beta\alpha^*$. Ainsi, la grammaire suivante génère le même langage mais sans utiliser la récursion à gauche.

$$\begin{array}{lcl} A & \rightarrow & \beta A' \\ A' & \rightarrow & \alpha A' \mid \epsilon \end{array}$$

Élimination de la récursion à gauche

Exemple 4.17. On peut éliminer la récursion à gauche de la grammaire suivante:

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

en la transformant en la grammaire suivante:

$$\begin{array}{lcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \mid \epsilon \\ T & \rightarrow & FT' \\ T' & \rightarrow & *FT' \mid \epsilon \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

Élimination de la récursion à gauche

La technique d'élimination de la récursion à gauche fonctionne peu importe le nombre de A-productions. À titre d'exemple, les A-productions suivantes peuvent se grouper en celles qui commencent avec A et celles qui ne commencent pas avec A:

$$A \quad \rightarrow \quad A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$$

On peut alors transformer la grammaire en une grammaire sans récursion à gauche:

$$\begin{array}{lcl} A & \rightarrow & \beta_1 A' \mid \ldots \mid \beta_n A' \\ A' & \rightarrow & \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \epsilon \end{array}$$

Ce genre de transformation élimine les récursions à gauche *directes*. Certains non-terminaux peuvent être récursifs à gauche indirectement. Par exemple, dans la grammaire suivante, S est récursif à gauche:

$$\begin{array}{lcl} S & \rightarrow & Aa \mid b \\ A & \rightarrow & Ac \mid Sd \mid \epsilon \end{array}$$

Élimination de la récursion à gauche

L'algorithme suivant élimine toute récursion à gauche d'une grammaire G, même la récursion indirecte. Il est garanti de fonctionner correctement lorsqu'il n'y a pas de cycles ni d'ϵ-productions dans G. **Définition**: Il y a un *cycle* dans G lorsqu'il existe un non-terminal A tel que $A \Rightarrow^+ A$.

Algorithme 4.19

Entrée: Une grammaire G sans cycles ni ϵ-productions.

Sortie: Une grammaire équivalente sans récursion à gauche.

Méthode:

1. Ordonner les non-terminaux par indices $A_1, \ldots, A_n$.
2. Pour $i := 1$ à n faire
   (a) Pour $j := 1$ à $i - 1$ faire
   Remplacer chaque production de la forme $A_i \to A_j\gamma$
   par les productions $A_i \to \delta_1\gamma \mid \ldots \mid \delta_k\gamma$,
   où $A_j \to \delta_1 \mid \ldots \mid \delta_k$ sont les A_j-productions actuelles
   (b) Éliminer les récursions immédiates à gauche des A_i-productions

Élimination de la récursion à gauche

Exemple 4.20. Éliminons la récursion à gauche de la grammaire:

$$\begin{array}{lcl} S & \to & Aa \mid b \\ A & \to & Ac \mid Sd \mid e \end{array}$$

Sommaire:

~~$S \to S \ldots$~~	$S \to A \ldots$
$A \to S \ldots$	$A \to A \ldots$

Nous ordonnons (arbitrairement) les non-terminaux ainsi en 1:

S, A

Dans une première étape ($i = 1$), nous éliminons toute récursion à gauche de S vers un non-terminal précédent en 2(a) (aucun, donc boucle sur j vide) et la récursion à gauche directe en 2(b) (inexistante).

Dans une deuxième étape ($i = 2$), nous éliminons toute récursion à gauche de A vers un non-terminal précédent en 2(a) ($j = 1$):

$$\begin{array}{lcl} S & \to & Aa \mid b \\ A & \to & Ac \mid Aad \mid bd \mid e \end{array}$$

Sommaire:

~~$S \to S \ldots$~~	$S \to A \ldots$
~~$A \to S \ldots$~~	$A \to A \ldots$

Finalement, nous éliminons la récursion à gauche directe en 2(b):

$$\begin{array}{lcl} S & \to & Aa \mid b \\ A & \to & bdA' \mid eA' \\ A' & \to & cA' \mid adA' \mid \epsilon \end{array}$$

Sommaire:

~~$S \to S \ldots$~~	$S \to A \ldots$
~~$A \to S \ldots$~~	~~$A \to A \ldots$~~

Exemple. Sur le site web, on peut retrouver une trace de l'algorithme 4.19 dans le cas général.

Factorisation à gauche

Dans une grammaire comme la suivante:

$$\begin{array}{lcl} \textit{stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt} \\ & | & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \end{array}$$

on ne peut choisir dès le départ laquelle des productions il faut utiliser. Toutefois, en mettant en commun les 4 premiers symboles on retarde la prise de décision jusqu'au point où on dispose des informations nécessaires:

$$\begin{array}{lcl} \textit{stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt}\ \textit{stmt}' \\ \textit{stmt}' & \rightarrow & \textbf{else } \textit{stmt} \mid \epsilon \end{array}$$

Plus généralement, on peut remplacer les productions:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

par:

$$\begin{array}{lcl} A & \rightarrow & \alpha A' \\ A' & \rightarrow & \beta_1 \mid \beta_2 \end{array}$$

Factorisation à gauche

Algorithme 4.21

Entrée: Une grammaire G.

Sortie: Une grammaire équivalente factorisée à gauche.

Méthode:

Pour chaque non-terminal A:

1. Trouver le plus long préfixe α commun à au moins deux A-productions.
2. Si $\alpha \neq \epsilon$, remplacer:

$$A \quad \rightarrow \quad \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m$$

   par:

$$\begin{array}{lcl} A & \rightarrow & \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_m \\ A' & \rightarrow & \beta_1 \mid \ldots \mid \beta_n \end{array}$$

   où A' est un nouveau non-terminal.
3. Si $\alpha \neq \epsilon$, retourner à 1.

Factorisation à gauche

Exemple

$$\begin{array}{lcl} E & \rightarrow & T \Rightarrow E \\ & | & T \otimes E \\ & | & T \end{array}$$

Factorisation à gauche

Exemple

$$\begin{array}{lll} E & \rightarrow & T \Rightarrow E \\ & | & T \otimes E \\ & | & T \end{array}$$

Devient:

$$\begin{array}{lll} E & \rightarrow & T\ E' \\ E' & \rightarrow & \Rightarrow E \mid \otimes E \mid \epsilon \end{array}$$

Factorisation à gauche

Exemple

$$
\begin{array}{rcl}
S & \rightarrow & \textbf{do } S \textbf{ while } E \\
 & | & \textbf{do } S \textbf{ until } E
\end{array}
$$

Factorisation à gauche

Exemple

$$\begin{array}{rcl} S & \rightarrow & \textbf{do } S \textbf{ while } E \\ & | & \textbf{do } S \textbf{ until } E \end{array}$$

Devient:

$$\begin{array}{rcl} S & \rightarrow & \textbf{do } S\ S' \\ S' & \rightarrow & \textbf{while } E \mid \textbf{until } E \end{array}$$

Factorisation à gauche

Exemple 4.22

$$
\begin{array}{lcl}
S & \rightarrow & \textbf{if } E \textbf{ then } S \\
 & | & \textbf{if } E \textbf{ then } S \textbf{ else } S \\
 & | & \texttt{a} \\
E & \rightarrow & \texttt{b}
\end{array}
$$

Factorisation à gauche

Exemple 4.22

$$
\begin{array}{lcl}
S & \rightarrow & \textbf{if } E \textbf{ then } S \\
 & | & \textbf{if } E \textbf{ then } S \textbf{ else } S \\
 & | & \texttt{a} \\
E & \rightarrow & \texttt{b}
\end{array}
$$

Devient:

$$
\begin{array}{lcl}
S & \rightarrow & \textbf{if } E \textbf{ then } S\ S' \mid \texttt{a} \\
S' & \rightarrow & \textbf{else } S \mid \epsilon \\
E & \rightarrow & \texttt{b}
\end{array}
$$

Factorisation à gauche

Une erreur qui est fréquemment commise consiste à faire la transformation suivante, laquelle **n**'est **pas** une factorisation à gauche.

Remplacer:

$$A \quad \rightarrow \quad \alpha\beta_1 \mid \alpha\beta_2$$

par:

$$\begin{array}{lcl} A & \rightarrow & A'\beta_1 \mid A'\beta_2 \\ A' & \rightarrow & \alpha \end{array}$$

En effet, il y a toujours deux alternatives commençant de la même façon.

Analyse Syntaxique:

Analyse descendante prédictive *avec* récursion

Section 4.4

Analyse syntaxique descendante

L'analyse syntaxique descendante peut être vue comme étant:

- une tentative de trouver une dérivation à gauche d'abord pour la chaîne d'entrée; ou
- une tentative de construire un arbre de dérivation pour la chaîne d'entrée en construisant les noeuds de l'arbre en pré-ordre.

En règle générale, l'analyse syntaxique descendante demande de faire du *rebroussement*.

Ce n'est pas souhaitable en compilation car, pour certaines grammaires, l'analyse peut prendre un temps exponentiel en pire cas.

Analyse syntaxique descendante

Exemple 4.29

Simulons l'analyse descendante de la chaîne $w = cad$ en considérant la grammaire:

$$\begin{array}{lcl} S & \rightarrow & cAd \\ A & \rightarrow & ab \mid a \end{array}$$

Étape	Forme de phrase	Production choisie
1	S	$S \rightarrow cAd$
2	cAd	$A \rightarrow ab$ *(choix 1)*
3	$cabd$	**Échec**, retournons en arrière
2	cAd	$A \rightarrow a$ *(choix 2)*
3′	cad	**Succès**

Nécessite un rebroussement!

Analyse syntaxique descendante

Lorsqu'il y a rebroussement, l'analyse syntaxique peut requérir, en pire cas, un temps exponentiel.

Exemple:

L'entrée $a^n b^n$ pourrait demander un temps exponentiel à analyser avec la grammaire suivante:

$$\begin{aligned} A &\rightarrow a\ a\ A\ b \\ A &\rightarrow a\ A\ b \\ A &\rightarrow \epsilon \end{aligned}$$

Analyseurs prédictifs

Dans certains cas, en concevant bien notre grammaire, en éliminant les ambiguïtés et les récursions à gauche, on obtient un analyseur syntaxique descendant qui n'a pas besoin d'effectuer de rebroussement, c'est-à-dire un *analyseur prédictif*.

Pour obtenir un analyseur prédictif, il faut (entre autres) respecter la condition suivante:

- pour tout non-terminal A à substituer,
- pour tout terminal a qui doit être consommé sur l'entrée,
- il doit y avoir au plus une alternative parmi $A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$ qui est capable de générer une chaîne commençant par a.

Par exemple, en ce qui concerne le non-terminal *stmt* dans la grammaire suivante, la condition est respectée:

$$
\begin{array}{lcl}
\textit{stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt} \\
& \mid & \textbf{while } \textit{expr} \textbf{ do } \textit{stmt} \\
& \mid & \textbf{begin } \textit{stmt\_list} \textbf{ end}
\end{array}
$$

Analyse prédictive avec récursion

On peut facilement écrire un analyseur prédictif avec récursion:

- Créer une fonction pour chaque symbole non-terminal.
- Dans la fonction correspondant au non-terminal A:
  - Si A a plus d'une production, installer une procédure qui détermine d'abord quelle production est appropriée en se basant sur le prochain jeton.
  - Pour chaque membre droit α_i et pour chaque symbole $X_{i,j}$ dans α_i:
    - Si $X_{i,j}$ est un terminal, s'assurer que le prochain jeton de l'entrée est identique à $X_{i,j}$ et consommer ce jeton de l'entrée (sinon, lancer une exception).
    - Si $X_{i,j}$ est un non-terminal, appeler la fonction correspondant à $X_{i,j}$.
- Installer un appel à la fonction correspondant au symbole de départ.
- Vérifier que toute l'entrée a été lue et qu'aucune exception n'a été levée.

Note: *nous verrons au prochain chapitre comment instrumenter ces fonctions afin de générer des données utilisables pour la suite de la compilation.*

Analyse prédictive avec récursion

Exemple. Soit la grammaire suivante:

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \ \mid\ -\ T\ E' \ \mid\ \epsilon \\
T & \rightarrow & (\ E\ ) \ \mid\ \textbf{id}\ T' \\
T' & \rightarrow & \epsilon \ \mid\ [\ E\ ]
\end{array}
$$

Dans le pseudo-code que nous présentons, nous utilisons les fonctions suivantes.

Supposons que '*peekToken*' retourne le prochain jeton sans le consommer.

Supposons que '*readToken*' reçoit un type de jeton en argument et lit un jeton en s'assurant qu'il a le type souhaité.

Analyse prédictive avec récursion

Exemple. (suite)

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \ \mid\ -\ T\ E' \ \mid\ \epsilon \\
T & \rightarrow & (\ E\ ) \ \mid\ \textbf{id}\ T' \\
T' & \rightarrow & \epsilon \ \mid\ [\ E\ ]
\end{array}
$$

Voici le pseudo-code d'un analyseur récursif qui reconnait cette grammaire:

```
void E()                                    /* E -> T E' */
{
  T(); Eprime();
}
```

Analyse prédictive avec récursion

Exemple. (suite)

$$
\begin{array}{rcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \ \mid\ -\ T\ E' \ \mid\ \epsilon \\
T & \rightarrow & (\ E\ ) \ \mid\ \textbf{id}\ T' \\
T' & \rightarrow & \epsilon \ \mid\ [\ E\ ]
\end{array}
$$

Voici la suite du pseudo-code:

```
void Eprime()
{
  if      (peekToken().type == '+')     /* E' -> + T E' */
    { readToken('+'); T(); Eprime(); }
  else if (peekToken().type == '-')     /* E' -> - T E' */
    { readToken('-'); T(); Eprime(); }
  else                                  /* E' -> ε */
    { }
}
```

Analyse prédictive avec récursion

Exemple. (suite)

$$
\begin{array}{rcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \ \mid\ -\ T\ E' \ \mid\ \epsilon \\
T & \rightarrow & (\ E\ ) \ \mid\ \textbf{id}\ T' \\
T' & \rightarrow & \epsilon \ \mid\ [\ E\ ]
\end{array}
$$

Voici la suite du pseudo-code:

```
void T()
{
  if   (peekToken().type == '(')        /* T -> ( E ) */
    { readToken('('); E(); readToken(')'); }
  else                                  /* T -> id T' */
    { readToken('id'); Tprime(); }
}
```

Analyse prédictive avec récursion

Exemple. (suite)

$$
\begin{array}{rcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \ \mid\ -\ T\ E' \ \mid\ \epsilon \\
T & \rightarrow & (\ E\ ) \ \mid\ \textbf{id}\ T' \\
T' & \rightarrow & \epsilon \ \mid\ [\ E\ ]
\end{array}
$$

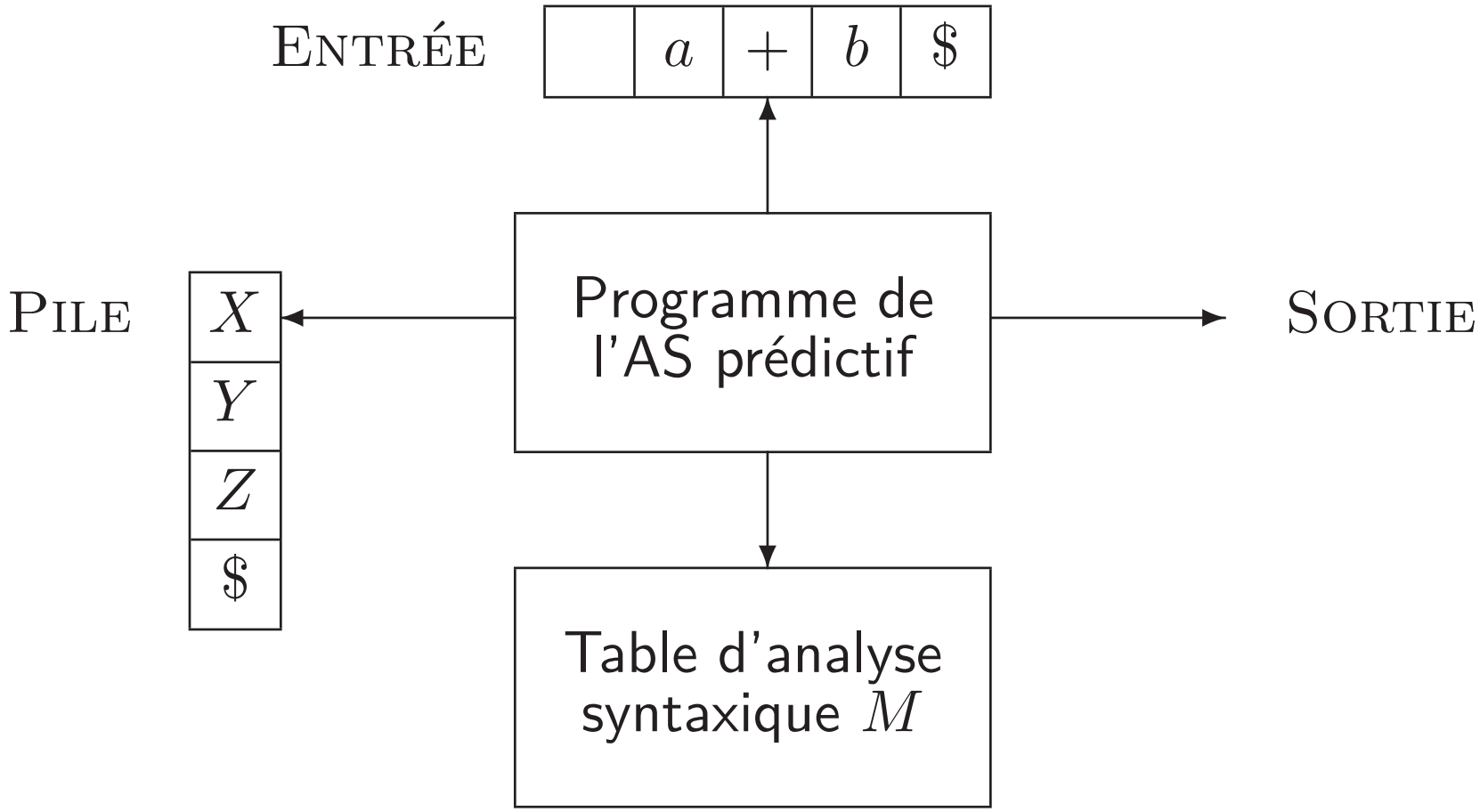
Voici la suite du pseudo-code:

```
void Tprime()
{
  if   (peekToken().type == '[')        /* T' -> [ E ] */
    { readToken('['); E(); readToken(']'); }
  else                                  /* T' -> ϵ */
    { }
}
```

Analyse Syntaxique:

Analyse descendante prédictive *sans* récursion

Analyse prédictive sans récursion



Analyse prédictive sans récursion

L'analyseur syntaxique prédictif:

- comporte un tampon d'entrée, initialisé avec la chaîne d'entrée suivie du symbole $\$$;
- comporte une pile, initialisée avec le symbole de départ par-dessus le symbole $\$$;
- utilise une table d'analyse M, où $M[A, a]$ indique quelle production utiliser si le symbole sur le dessus de la pile est A et que le prochain symbole en entrée est a.

Dans une configuration où X est sur le dessus de la pile et a est le prochain symbole dans l'entrée, l'analyseur effectue une des actions parmi les suivantes:

- Si $X = a = \$$, l'analyseur arrête ses opérations et annonce une analyse réussie.
- Si $X = a \neq \$$, l'analyseur dépile X et fait avancer le pointeur de l'entrée.
- Si X est un non-terminal, le programme de l'analyseur consulte la table d'analyse en position $M[X, a]$. La case consultée fournit soit une production à utiliser, soit une indication d'erreur. Si, par exemple, $M[X, a] = \{X \rightarrow UVW\}$, alors l'analyseur dépile X et empile W, V et U, dans l'ordre. En cas d'erreur, l'analyseur s'arrête et signale l'erreur.

La sortie de l'analyseur consiste en la suite de productions utilisées.

Analyse prédictive sans récursion

Algorithme 4.34

Entrée: Une chaîne w et la table d'analyse M associée à une grammaire G.

Sortie: Une dérivation à gauche d'abord de w si $w \in L(G)$ et le signalement d'une erreur sinon.

Méthode:

initialiser la pile avec S par-dessus \$ et le tampon d'entrée avec w\$;
faire pointer ip sur le premier symbole de l'entrée;
répéter
  soit X le symbole du dessus de pile et a le symbole pointé par ip;
  si X est un terminal ou \$ **alors**
    si $X = a$ **alors**
      dépiler X et incrémenter ip
    sinon *erreur*()
  sinon /* X est un non-terminal */
    si $M[X, a] = X \to Y_1 \dots Y_k$ **alors**
      dépiler X;
      empiler Y_k, ..., Y_1, dans cet ordre;
      afficher la production $X \to Y_1 \dots Y_k$
    sinon *erreur*()
jusqu'à ce que $X =$ \$ /* la pile est vide */

Analyse prédictive sans récursion

Exemple 4.32

À la grammaire:

$$\begin{array}{lcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \mid \epsilon \\ T & \rightarrow & FT' \\ T' & \rightarrow & *FT' \mid \epsilon \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

on associe la table d'analyse suivante:

Non-terminal	Symbole d'entrée					
	id	$+$	$*$	$($	$)$	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		

Analyse prédictive sans récursion

Analyse prédictive de la chaîne $\mathbf{id} + \mathbf{id} * \mathbf{id}$.

Non-terminal	Symbole d'entrée					
	id	$+$	$*$	$($	$)$	\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

Pile	Entrée	Sortie
E \$	**id** + **id** * **id** \$	$E \to TE'$
$T\ E'$ \$	**id** + **id** * **id** \$	$T \to FT'$
$F\ T'\ E'$ \$	**id** + **id** * **id** \$	$F \to$ **id**
id $T'\ E'$ \$	**id** + **id** * **id** \$	(Consommer le terminal **id**)
$T'\ E'$ \$	+ **id** * **id** \$	$T' \to \epsilon$
E' \$	+ **id** * **id** \$	$E' \to +\ T\ E'$
$+\ T\ E'$ \$	+ **id** * **id** \$	(Consommer le terminal $+$)
$T\ E'$ \$	**id** * **id** \$	$T \to F\ T'$
$F\ T'\ E'$ \$	**id** * **id** \$	$F \to$ **id**
id $T'\ E'$ \$	**id** * **id** \$	(Consommer le terminal **id**)
$T'\ E'$ \$	* **id** \$	$T' \to *\ F\ T'$
$*\ F\ T'\ E'$ \$	* **id** \$	(Consommer le terminal *)
$F\ T'\ E'$ \$	**id** \$	$F \to$ **id**
id $T'\ E'$ \$	**id** \$	(Consommer le terminal **id**)
$T'\ E'$ \$	\$	$T' \to \epsilon$
E' \$	\$	$E' \to \epsilon$
\$	\$	(Succès)

PREDICT, FIRST et FOLLOW

Les ensembles PREDICT nous aident à choisir quelles productions utiliser.

Supposons:

- qu'on cherche à substituer un non-terminal A,
- qu'il existe une production $A \to \alpha$ et
- qu'on observe que b est le prochain jeton de l'entrée,

alors c'est *envisageable* d'utiliser $A \to \alpha$ si $b \in \mathrm{PREDICT}(A \to \alpha)$.

PREDICT, FIRST et FOLLOW

Exemple

Considérons cette grammaire:

$$\begin{array}{rcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \mid \epsilon \\ T & \rightarrow & FT' \\ T' & \rightarrow & *FT' \mid \epsilon \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

En sachant que

$$\text{PREDICT}(F \rightarrow (E)) = \{(\}$$

et

$$\text{PREDICT}(F \rightarrow \mathbf{id}) = \{\mathbf{id}\},$$

on peut décider quelle production utiliser en observant le prochain jeton de l'entrée.

Pareillement, en sachant que

$$\text{PREDICT}(T' \rightarrow *FT') = \{*\}$$

et

$$\text{PREDICT}(T' \rightarrow \epsilon) = \{+,), \$\}$$

(où \$ est un terminal spécial représentant la fin de l'entrée),
on peut décider quelle production utiliser en observant le prochain jeton de l'entrée.

PREDICT, FIRST et FOLLOW

Exemple

Considérons cette grammaire:

$$\begin{array}{lcl} S & \rightarrow & \textbf{if } E \textbf{ then } S\ S' \mid \texttt{a} \\ S' & \rightarrow & \textbf{else } S \mid \epsilon \\ E & \rightarrow & \texttt{b} \end{array}$$

En sachant que

$$\text{PREDICT}(S' \rightarrow \textbf{else } S) = \{\textbf{else}\}$$

et

$$\text{PREDICT}(S' \rightarrow \epsilon) = \{\textbf{else}, \$\},$$

on **ne** peut **pas** décider quelle production utiliser si le prochain jeton de l'entrée est **else**; c'est-à-dire que les deux productions sont envisageables.

PREDICT, FIRST et FOLLOW

Les ensembles PREDICT sont calculés à partir des ensembles FIRST et FOLLOW.

$$\mathrm{PREDICT}(A \to \alpha) = \begin{cases} (\mathrm{FIRST}(\alpha) - \{\epsilon\}) \cup \mathrm{FOLLOW}(A), & \text{si } \epsilon \in \mathrm{FIRST}(\alpha) \\ \mathrm{FIRST}(\alpha), & \text{sinon} \end{cases}$$

On définit l'ensemble $\mathrm{FIRST}(\alpha)$ comme étant l'ensemble des terminaux qui débutent les chaînes générées à partir de α. De plus, on ajoute ϵ à $\mathrm{FIRST}(\alpha)$ si α peut générer ϵ.

On définit l'ensemble $\mathrm{FOLLOW}(A)$ comme étant l'ensemble des terminaux qui peuvent apparaître immédiatement à droite de A dans une forme de phrase générée par la grammaire. De plus, on ajoute $\$$ à $\mathrm{FOLLOW}(A)$ si A peut apparaître complètement à droite dans une forme de phrase.

PREDICT, FIRST et FOLLOW

Mathématiquement: $\mathrm{FIRST}(\alpha) = \{a \mid \alpha \Rightarrow^* aw\} \cup \{\epsilon \mid \alpha \Rightarrow^* \epsilon\}$

Opérationnellement, on peut calculer des ensembles FIRST à l'aide des contraintes suivantes:

- $\mathrm{FIRST}(\epsilon) = \{\epsilon\}$
- $\mathrm{FIRST}(a\alpha) = \{a\}$
- $\mathrm{FIRST}(A) = \mathrm{FIRST}(\alpha_1) \cup \ldots \cup \mathrm{FIRST}(\alpha_n)$,
  où $A \to \alpha_1 \mid \ldots \mid \alpha_n$ sont les A-productions,
- $\mathrm{FIRST}(A\alpha) = \begin{cases} (\mathrm{FIRST}(A) - \{\epsilon\}) \cup \mathrm{FIRST}(\alpha), & \text{si } \epsilon \in \mathrm{FIRST}(A) \\ \mathrm{FIRST}(A), & \text{sinon} \end{cases}$

et en résolvant les contraintes pour obtenir la *plus petite solution*.

PREDICT, FIRST et FOLLOW

Mathématiquement, $\mathrm{FOLLOW}(A) = \{a \mid S \Rightarrow^* \alpha A a \beta\} \cup \{\$ \mid S \Rightarrow^* \alpha A\}$

Opérationnellement, on peut calculer les ensembles FOLLOW de tous les non-terminaux d'une grammaire en posant les contraintes suivantes.

- Si S est le symbole de départ, alors $\$ \in \mathrm{FOLLOW}(S)$.
- Si $A \to \alpha B \beta$ est une production,
  alors $\mathrm{FIRST}(\beta) - \{\epsilon\} \subseteq \mathrm{FOLLOW}(B)$.
- Si $A \to \alpha B$ est une production ou
  si $A \to \alpha B \beta$ est une production et que $\epsilon \in \mathrm{FIRST}(\beta)$,
  alors $\mathrm{FOLLOW}(A) \subseteq \mathrm{FOLLOW}(B)$.

Ensuite, il faut résoudre les contraintes pour obtenir la *plus petite solution*.

PREDICT, FIRST et FOLLOW

Le calcul de la plus petite solution pour FIRST et FOLLOW se fait comme suit.

- Initialiser à $\emptyset$ tous les ensembles sujets aux contraintes.
- Tant qu'il y a une contrainte qui n'est pas respectée,
  ajouter les éléments manquants
  à l'ensemble qui est du côté droit de l'opérateur $\subseteq$
  (ou, symétriquement, du côté gauche de l'opérateur $\supseteq$).
- Lorsque toutes les contraintes sont respectées,
  on a la plus petite solution aux contraintes.

Il ne faut pas tenter de *retirer* des éléments du côté gauche de l'opérateur $\subseteq$.

PREDICT, FIRST et FOLLOW (exemple)

Exemple 4.30

Considérons (une fois encore) cette grammaire:

$$\begin{array}{rcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \mid \epsilon \\ T & \rightarrow & FT' \\ T' & \rightarrow & *FT' \mid \epsilon \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

Calcul des ensembles FIRST.

$$\begin{array}{rl} & \mathrm{FIRST}(E) \\ = & \langle\ E \rightarrow TE'\ \rangle \\ & \mathrm{FIRST}(TE') \\ = & \langle\ \epsilon \notin \mathrm{FIRST}(T)\ \rangle \\ & \mathrm{FIRST}(T) \end{array}$$

$$\begin{array}{rl} & \mathrm{FIRST}(E') \\ = & \langle\ E' \rightarrow +TE' \mid \epsilon\ \rangle \\ & \mathrm{FIRST}(+TE') \cup \mathrm{FIRST}(\epsilon) \\ = & \\ & \{+\} \cup \{\epsilon\} \\ = & \\ & \{+, \epsilon\} \end{array}$$

PREDICT, FIRST et FOLLOW (exemple)

Calcul des ensembles FIRST. (suite)

$$
\begin{array}{ll}
 & \mathrm{FIRST}(T) \\
= & \langle\ T \to FT'\ \rangle \\
 & \mathrm{FIRST}(FT') \\
= & \langle\ \epsilon \notin \mathrm{FIRST}(F)\ \rangle \\
 & \mathrm{FIRST}(F)
\end{array}
\qquad
\begin{array}{ll}
 & \mathrm{FIRST}(T') \\
= & \langle\ T' \to *FT' \mid \epsilon\ \rangle \\
 & \mathrm{FIRST}(*FT') \cup \mathrm{FIRST}(\epsilon) \\
= & \\
 & \{*\} \cup \{\epsilon\} \\
= & \\
 & \{*, \epsilon\}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathrm{FIRST}(F) \\
= & \langle\ F \to (E) \mid \mathbf{id}\ \rangle \\
 & \mathrm{FIRST}((E)) \cup \mathrm{FIRST}(\mathbf{id}) \\
= & \\
 & \{(\} \cup \{\mathbf{id}\} \\
= & \\
 & \{(, \mathbf{id}\}
\end{array}
$$

Plus petite solution:

N-T	FIRST
E	$\{(, \mathbf{id}\}$
E'	$\{+, \epsilon\}$
T	$\{(, \mathbf{id}\}$
T'	$\{*, \epsilon\}$
F	$\{(, \mathbf{id}\}$

PREDICT, FIRST et FOLLOW (exemple)

Calcul des ensembles FOLLOW.

On commence par appliquer la première règle: $\text{FOLLOW}(E) \supseteq \{\$\}$.

Appliquons la deuxième règle, symbole par symbole:

Symbole	Contraintes identifiées
$E \to \underline{T}E'$	$\text{FOLLOW}(T) \supseteq \text{FIRST}(E') - \{\epsilon\} = \{+, \epsilon\} - \{\epsilon\} = \{+\}$
$E \to T\underline{E'}$	$\text{FOLLOW}(E') \supseteq \text{FIRST}(\epsilon) - \{\epsilon\} = \{\epsilon\} - \{\epsilon\} = \{\}$
$E' \to +\underline{T}E'$	$\text{FOLLOW}(T) \supseteq \text{FIRST}(E') - \{\epsilon\} = \{+, \epsilon\} - \{\epsilon\} = \{+\}$
$E' \to +T\underline{E'}$	$\text{FOLLOW}(E') \supseteq \text{FIRST}(\epsilon) - \{\epsilon\} = \{\epsilon\} - \{\epsilon\} = \{\}$
$T \to \underline{F}T'$	$\text{FOLLOW}(F) \supseteq \text{FIRST}(T') - \{\epsilon\} = \{*, \epsilon\} - \{\epsilon\} = \{*\}$
$T \to F\underline{T'}$	$\text{FOLLOW}(T') \supseteq \text{FIRST}(\epsilon) - \{\epsilon\} = \{\epsilon\} - \{\epsilon\} = \{\}$
$T' \to *\underline{F}T'$	$\text{FOLLOW}(F) \supseteq \text{FIRST}(T') - \{\epsilon\} = \{*, \epsilon\} - \{\epsilon\} = \{*\}$
$T' \to *F\underline{T'}$	$\text{FOLLOW}(T') \supseteq \text{FIRST}(\epsilon) - \{\epsilon\} = \{\epsilon\} - \{\epsilon\} = \{\}$
$F \to (\underline{E})$	$\text{FOLLOW}(E) \supseteq \text{FIRST}()) - \{\epsilon\} = \{)\} - \{\epsilon\} = \{)\}$

PREDICT, FIRST et FOLLOW (exemple)

Calcul des ensembles FOLLOW. (suite)

Appliquons la troisième règle, symbole par symbole.

Symbole	Justification	Contraintes identifiées
$E \to \underline{T}E'$	suivi d'un symbole annulable	$\text{FOLLOW}(T) \supseteq \text{FOLLOW}(E)$
$E \to T\underline{E'}$	situé à la fin	$\text{FOLLOW}(E') \supseteq \text{FOLLOW}(E)$
$E' \to +\underline{T}E'$	suivi d'un symbole annulable	$\text{FOLLOW}(T) \supseteq \text{FOLLOW}(E')$
$E' \to +T\underline{E'}$	situé à la fin	$\text{FOLLOW}(E') \supseteq \text{FOLLOW}(E')$
$T \to \underline{F}T'$	suivi d'un symbole annulable	$\text{FOLLOW}(F) \supseteq \text{FOLLOW}(T)$
$T \to F\underline{T'}$	situé à la fin	$\text{FOLLOW}(T') \supseteq \text{FOLLOW}(T)$
$T' \to *\underline{F}T'$	suivi d'un symbole annulable	$\text{FOLLOW}(F) \supseteq \text{FOLLOW}(T')$
$T' \to *F\underline{T'}$	situé à la fin	$\text{FOLLOW}(T') \supseteq \text{FOLLOW}(T')$
$F \to (\underline{E})$	–	–

PREDICT, FIRST et FOLLOW (exemple)

Calcul des ensembles FOLLOW. (suite)

Plus petite solution:

N-T	FOLLOW
E	$\{), \$\}$
E'	$\{), \$\}$
T	$\{+,), \$\}$
T'	$\{+,), \$\}$
F	$\{*, +,), \$\}$

Construction de tables d'analyse

Algorithme 4.31

Entrée: Une grammaire G.

Sortie: Une table d'analyse M.

Méthode:

1. Initialiser chaque entrée de M à l'ensemble vide.
2. Pour chaque production $A \to \alpha$, faire:
   (a) Pour chaque terminal $a \in \text{FIRST}(\alpha)$, ajouter $A \to \alpha$ à $M[A, a]$;
   (b) Si $\epsilon \in \text{FIRST}(\alpha)$, ajouter $A \to \alpha$ à $M[A, b]$ pour chaque symbole $b \in \text{FOLLOW}(A)$. b est un terminal ou \$.
3. Chaque case de M qui est restée vide correspond à une erreur.

Note: Cet algorithme insère la production $A \to \alpha$ dans les cases $M[A, b]$, pour tous les $b \in \text{PREDICT}(A \to \alpha)$.

Construction de tables d'analyse (exemple 4.32)

Production	Étape	Information	Table d'analyse
$E \to TE'$	2a	$\text{FIRST}(TE') = \{(, \mathbf{id}\}$	$M[E, \mathbf{id}] \ni E \to TE'$
	2b	$\epsilon \notin \text{FIRST}(TE')$ N/A	$M[E, (] \ni E \to TE'$
$E' \to +TE'$	2a	$\text{FIRST}(+TE') = \{+\}$	$M[E', +] \ni E' \to +TE'$
	2b	$\epsilon \notin \text{FIRST}(+TE')$ N/A	
$E' \to \epsilon$	2a	$\text{FIRST}(\epsilon) = \{\epsilon\}$ N/A	$M[E',)] \ni E' \to \epsilon$
	2b	$\text{FOLLOW}(E') = \{), \$\}$	$M[E', \$] \ni E' \to \epsilon$
$T \to FT'$	2a	$\text{FIRST}(FT') = \{(, \mathbf{id}\}$	$M[T, \mathbf{id}] \ni T \to FT'$
	2b	$\epsilon \notin \text{FIRST}(FT')$ N/A	$M[T, (] \ni T \to FT'$
$T' \to *FT'$	2a	$\text{FIRST}(*FT') = \{*\}$	$M[T', *] \ni T' \to *FT'$
	2b	$\epsilon \notin \text{FIRST}(*FT')$ N/A	
$T' \to \epsilon$	2a	$\text{FIRST}(\epsilon) = \{\epsilon\}$ N/A	$M[T', +] \ni T' \to \epsilon$
	2b	$\text{FOLLOW}(T') = \{+,), \$\}$	$M[T',)] \ni T' \to \epsilon$
			$M[T', \$] \ni T' \to \epsilon$
$F \to (E)$	2a	$\text{FIRST}((E)) = \{(\}$	$M[F, (] \ni F \to (E)$
	2b	$\epsilon \notin \text{FIRST}((E))$ N/A	
$F \to \mathbf{id}$	2a	$\text{FIRST}(\mathbf{id}) = \{\mathbf{id}\}$	$M[F, \mathbf{id}] \ni F \to \mathbf{id}$
	2b	$\epsilon \notin \text{FIRST}(\mathbf{id})$ N/A	

Construction de tables d'analyse (exemple 4.32)

La table obtenue est:

	id	$+$	$*$	$($	$)$	\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**			$F \to (E)$		

Construction de tables d'analyse

Exemple 4.33: À la grammaire suivante (laquelle symbolise le cas du *if-then-else*):

$$\begin{array}{lcl} S & \rightarrow & iEtSS' \mid a \\ S' & \rightarrow & eS \mid \epsilon \\ E & \rightarrow & b \end{array}$$

on associe la table d'analyse suivante:

Non-terminal	Symbole d'entrée					
	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

Conclusion: cette grammaire n'est pas appropriée pour l'analyse prédictive sans récursion (elle n'est pas *LL(1)*; voir page 70).

Construction de tables d'analyse

Analyse prédictive de la chaîne $i\,b\,t\,i\,b\,t\,a\,e\,a$ à l'aide de la table que l'on a obtenue.

Non-terminal	Symbole d'entrée					
	a	b	e	i	t	\$
S	$S \to a$			$S \to iEtSS'$		
S'			$S' \to \epsilon$ $S' \to eS$			$S' \to \epsilon$
E		$E \to b$				

Pile	Entrée	Sortie
S \$	$i\ b\ t\ i\ b\ t\ a\ e\ a$ \$	$S \to iEtSS'$
$i\ E\ t\ S\ S'$ \$	$i\ b\ t\ i\ b\ t\ a\ e\ a$ \$	(Consommer le terminal i)
$E\ t\ S\ S'$ \$	$b\ t\ i\ b\ t\ a\ e\ a$ \$	$E \to b$
$b\ t\ S\ S'$ \$	$b\ t\ i\ b\ t\ a\ e\ a$ \$	(Consommer le terminal b)
$t\ S\ S'$ \$	$t\ i\ b\ t\ a\ e\ a$ \$	(Consommer le terminal t)
$S\ S'$ \$	$i\ b\ t\ a\ e\ a$ \$	$S \to iEtSS'$
$i\ E\ t\ S\ S'\ S'$ \$	$i\ b\ t\ a\ e\ a$ \$	(Consommer le terminal i)
$E\ t\ S\ S'\ S'$ \$	$b\ t\ a\ e\ a$ \$	$E \to b$
$b\ t\ S\ S'\ S'$ \$	$b\ t\ a\ e\ a$ \$	(Consommer le terminal b)
$t\ S\ S'\ S'$ \$	$t\ a\ e\ a$ \$	(Consommer le terminal t)
$S\ S'\ S'$ \$	$a\ e\ a$ \$	$S \to a$
$a\ S'\ S'$ \$	$a\ e\ a$ \$	(Consommer le terminal a)
$S'\ S'$ \$	$e\ a$ \$	**Que faire???**

Grammaires LL(1)

On dit qu'une grammaire est *LL(1)* si sa table d'analyse ne comporte aucune case qui contient plus d'une production.

On dit qu'un langage L est *LL(1)* s'il existe une grammaire LL(1) qui génère L.

L'abréviation LL(1) a la signification suivante:

- le premier L indique que l'entrée (la suite de jetons) est lue de gauche à droite (*from Left to right*);
- le second L indique que la technique recherche une dérivation à gauche d'abord (*Leftmost derivation*);
- le 1 indique qu'un seul jeton constitue l'horizon de l'analyse.

Grammaires LL(1)

Les grammaires LL(1) possèdent plusieurs propriétés intéressantes:

- Aucune grammaire ambiguë ou comportant une récursion à gauche n'est LL(1).
- S'il existe deux productions $A \rightarrow \alpha$ et $A \rightarrow \beta$ ($\alpha \neq \beta$) dans une grammaire LL(1), on a que:
  - il n'existe pas de terminal a tel que α et β génèrent tous deux des chaînes qui commencent par a;
  - au plus un de α et β peut générer la chaîne ϵ;
  - si $\beta \rightarrow^* \epsilon$, alors α ne peut pas générer une chaîne dont le premier terminal est élément de $\mathrm{FOLLOW}(A)$.

On peut créer un analyseur prédictif directement à partir d'une grammaire G ssi G est LL(1).