Cours de compilation
4ème partie
Analyse syntaxique

Jacques Ferber

1. Introduction

L'analyse syntaxique est l'une des opérations majeures d'un compilateur qui consiste à indiquer si un texte est grammaticalement correct et à en tirer une représentation interne, que l'on appelle arbre syntaxique.

2. Grammaire

2.1 Introduction

Une grammaire est un ensemble de règles permettant de dire si une phrase, c'est-à-dire une suite de mots ou lexèmes (cf. l'analyse lexicale), est correcte ou non. Les règles qui nous intéressent en informatique sont celles qui donnent une description générative du langage, en indiquant comment précisément construire de telles phrase.

Par exemple une définition générative est la suivante :

> *Une phrase se compose d'une proposition sujet, d'un verbe et d'une proposition complément.*

Alors qu'une définition de la forme :

> *La subordonnée relative complète un nom ou un groupe nominal appartenant à la proposition principale*

n'est pas générative : elle ne nous dit pas comment effectivement construire une subordonnée relative.

2.2 Arbres et règles de grammaire

La structure d'une phrase est généralement récursive : par exemple, une phrase contient un groupe nominal et un groupe verbal, ce dernier étant composé lui-même d'un groupe nominal. De plus, un groupe nominal peut lui-même contenir des groupes nominaux sous la forme de subordonnées relatives.

L'expression de la structure d'une phrase s'exprime sous la forme d'un arbre, que l'on appelle *arbre syntaxique* (ou *arbre de dérivation*) d'une phrase. Par exemple, la phrase :

> *le chat qui est sur la commode regarde la souris*

peut être représentée par l'arbre suivant :

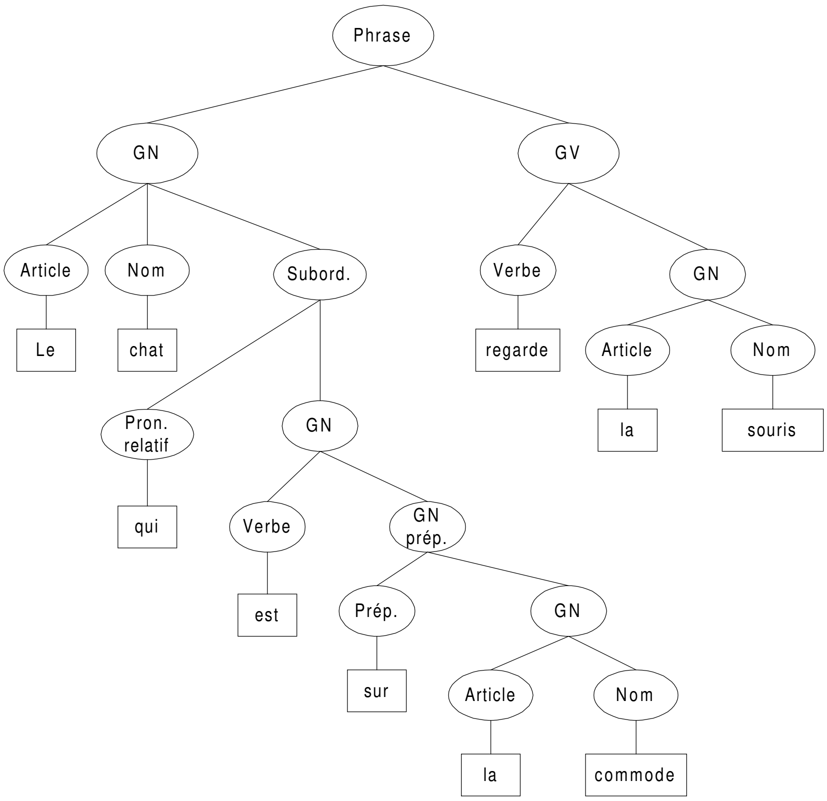


Figure 1. Arbre syntaxique ou de dérivation d'une phrase

Les feuilles de l'arbre correspondent aux mots de la phrase, et les noeuds non-terminaux aux catégories grammaticales.

Les règles de grammaire peuvent être données sous la forme de **règles de réécritures** (qu'on appelle aussi parfois productions) qui indiquent comment une catégorie grammaticale peut être transformée en une suite de catégories grammaticales ou de mots du vocabulaire.

Voici un exemple de grammaire qui a permis d'engendré la phrase précédente : (on notera $\alpha \rightarrow \beta \mid \delta$ à la place de $\alpha \rightarrow \beta, \alpha \rightarrow \delta$

<phrase> → <GN> <GV>
<GN> → <Article> <Nom> | <Article> <Nom> <Subord>
<Subord> → <Pron. relatif> <GV>
<GV> → <verbe> <GN> | <Verbe> <GN prep>
<GN Prep> → <Prep> <GN>
<Article> → **le** | **la** | **les ...**
<Nom> → **chat** | **commode** | **souris** | **...**
<Verbe> → **est** | **regarde**
<Prep> → **sur** | **sous** | **...**
<Pron. relatif> → **qui** | **que** | **dont** | **...**

Mais ces règles ne tiennent pas compte de certains critères grammaticaux tels que les accords. Par exemple, la phrase suivante peut être engendrée à partir de cette grammaire :

La chat que regarde sur le souris est les commode

qui est donc correcte par rapport à la grammaire, même si elle n'est pas correcte par rapport à la grammaire (complète) du français. Les problèmes d'accord sont traités en informatique par ce que l'on appelle vérification de type (aussi appelé parfois analyse sémantique, bien qu'il n'y ait pas de sémantique là-dedans au vrai sens du terme).

2.3 Quelques éléments grammaticaux de Pascal

Voici quelques règles correspondant à un sous-ensemble de Pascal (les expressions ne sont pas représentées, on les traitera plus loin) :

<programme> → <en-tête de prog.> <block>

<en-tête de prog.> → **program** <ident>

<block> → <declaration-list> <instruction-list>

<instruction-list> → <instruction> | <instruction> ; <instruction-list>

<instruction> → **begin** <instruction-list> **end** ...

{{à finir}}

2.4 Définitions

2.4.1 Grammaire

Une grammaire est donnée formellement sous la forme d'un quadruplet : G=<V, Σ, R, S> où :

- V est un vocabulaire (qu'on appelle aussi souvent alphabet), c'est à dire un ensemble fini de symboles.
- $\Sigma \subseteq V$ est l'ensemble des symboles terminaux, c'est à dire les mots du langage, et $V-\Sigma$ est l'ensemble des symboles non-terminaux, c'est-à-dire les catégories grammaticales.
- $R \subseteq (V^+ \times V^*)$ est un ensemble fini de règles de réécritures.
- $S \in V-\Sigma$ est le symbole de départ (Phrase dans l'exemple du chat) .

Notation : on fait commencer les catégories grammaticales d'une majuscule, les mots (terminaux) sont écrits en minuscules, les lettre grecques α,β,δ etc. représentent des suites de symboles (terminaux ou non-terminaux). Le mot vide s'écrit ε (comme pour les expressions régulières).

2.5 Types de grammaires

- **Type 0** : pas de restriction sur les règles.
- **Type 1** : grammaire dépendantes du contexte (context-sensitive). Les règles sont de la forme :

  $$\alpha \to \beta$$

  qui doivent satisfaire la condition : $|\alpha| \leq |\beta|$. De plus, on ne doit pas trouver le symbole S dans le membre droit des règles. Dans ce type de grammaire, on peut avoir des règles de la forme :

  $$\alpha A\beta \to \delta,$$

  c'est à dire des règles qui ne peuvent s'appliquer que si la catégorie grammaticale est entourée d'ensembles de symboles. C'est pourquoi on les nomme « dépendante du contexte ».

- **Type 2** : grammaires hors-contexte (context-free). C'est une grammaire dont toutes les productions sont de la forme

  $A \rightarrow \beta$

  où $A \in V-\Sigma$ et où il n'y a pas de restrictions sur β. β peut contenir n'importe quel type de symbole. On les appelle aussi grammaires algébriques.
- **Type 3** : grammaire régulières. Une grammaire est régulière si toutes ses production sont de la forme :

  $A \rightarrow \alpha B$

  $A \rightarrow \alpha$

  où A et B sont des catégories grammaticales (c'est-à-dire appartiennent à $V - \Sigma$) et où $\alpha \in \Sigma^*$. Cette grammaire correspond exactement aux expressions régulières.

De plus on a les propriétés suivantes :

Type 3 $\subset$ Type 2 $\subset$ Type 1 $\subset$ Type 0

En compilation on ne s'intéresse qu'aux grammaires de type 2 (pour l'analyse syntaxique) et 3 (pour l'analyse lexicale).

On note L(G) le langage engendré par la grammaire G.

2.6 Grammaires et arbres de dérivation

Les expressions arithmétiques ont été particulièrement étudiées et posent de nombreux problèmes que nous verrons plus loin dans le cours. Pour exprimer une expression arithmétique de la forme : (a + b) * (a * 2+3) on peut utiliser les grammaires suivantes :

GEA1 :

$E \rightarrow E + E \mid E * E \mid E \wedge E \mid (E) \mid Id \mid Cste$

GEA :

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow G \wedge F \mid G$

$G \rightarrow (E) \mid Id \mid Cste$

Dérivations équivalentes :

Pour obtenir l'expression (Id + Id) * Id du langage engendré par GEA on peut obtenir : (notation $\Rightarrow$ indique une réécriture, $\Rightarrow^*$ indique une dérivation (suite de réécriture) $\Rightarrow_i$ indique une réécriture par la règle i,

1. $E \Rightarrow T \Rightarrow T * F \Rightarrow T * G \Rightarrow T * Id \Rightarrow F * Id \Rightarrow (E) * Id \Rightarrow (E + T) * Id \Rightarrow^* (G + T)* Id \Rightarrow^* (G + G) * Id \Rightarrow (Id + G) * Id \Rightarrow (Id + Id) * Id$
2. $E \Rightarrow^* (G + G) * Id \Rightarrow (G + Id) * Id \Rightarrow (Id + Id) * Id$

Ces dérivations doivent être considérées comme équivalentes. La notation en arbre que l'on a vu pour un sous-ensemble du français est la plus appropriée, car on ne soucie pas de l'ordre des dérivations.

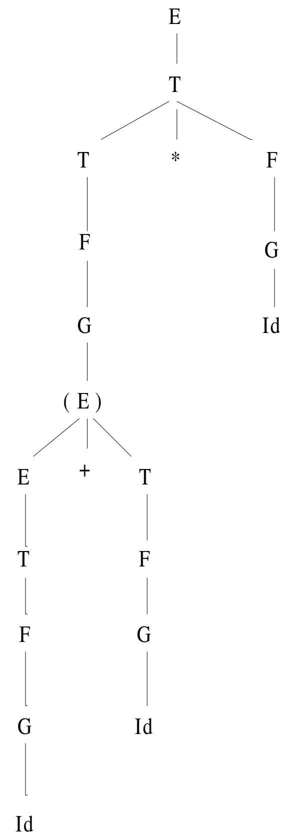


Figure 2. Arbre de dérivation d'une expression arithmétique

2.6.1 Dérivations gauches et droites :

- Dérivation gauche consiste à réécrire le symbole non-terminal le plus à gauche à chaque étape.
- Dérivation droite consiste à réécrire le symbole non-terminal le plus à droite à chaque étape.

2.6.2 Ambiguïté

Si une phrase d'un langage L(G) possède deux arbres de dérivations distincts, alors cette phrase est ambiguë et la grammaire associée est elle-aussi ambiguë.

Exemple d'une grammaire ambiguë (problème du if-then-else)

<instruction> → <conditionnelle> | ...

<conditionnelle> → **if** <condit> **then** <instruction> |
if <condit> **then** <instruction> **else** <instruction>

la phrase

if C1 **then** if C2 **then** S1 **else** S2

possède les deux arbres de dérivations suivants :

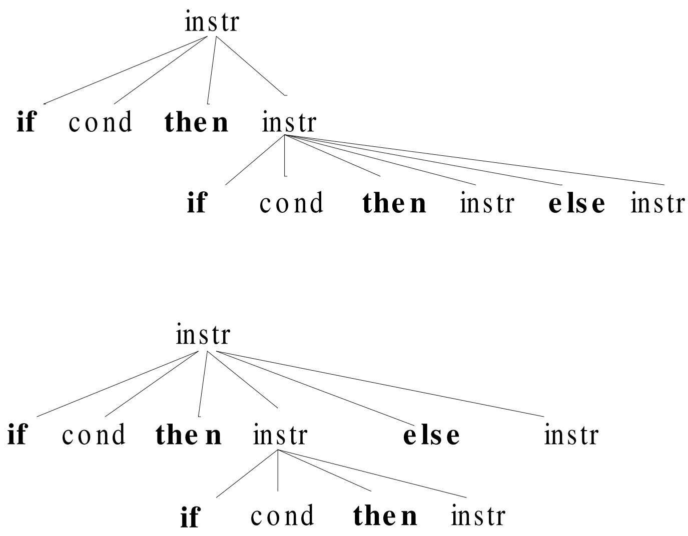


Figure 3. Une grammaire ambiguë

Mais il est possible de la désambiguiser : il suffit d'ajouter des règles de réécriture permettant d'appliquer le principe : « chaque else doit être associé avec le then le plus près ».

En incorporant ce principe dans la grammaire on obtient :

<instruction> → <instr. assoc.> | < instr. non-assoc>

<instr. assoc.> → **if** <condit> **then** <instr. assoc.> **else** <instr. assoc> |
<autres instr.>

<instr. non-assoc> → **if** <condit> **then** <instruction> |
if <condit> **then** <instr. assoc.> **else** <instr. non-assoc>

Il en est de même des expressions induites par la grammaire GEA1.

3. Processus d'analyse

Problème : reconstruire l'arbre syntaxique d'une phrase à partir d'une grammaire (non-ambiguë).

Il existe essentiellement deux types de processus :

1. l'analyse descendante, dans laquelle on construit l'arbre en descendant de la racine vers les feuilles. Cette analyse correspond à un parcours descendant gauche (parce qu'on lit de gauche à droite). Il s'agit de deviner à chaque étape quelle est la règle qui sert à engendrer le mot que l'on lit. Les grammaires GEA et GEA1 ne peuvent pas être appliquées, car on ne sait pas à chaque moment quelle règle appliquer. Cette analyse effectue un parcours préfixe de l'arbre.
2. l'analyse ascendante construit l'arbre syntaxique en montant des feuilles vers la racine. Elle correspond au parcours inverse (puisqu'on monte) du parcours descendant droit. Dans ce type d'analyse, on essaye d'appliquer le membre droit d'une règle et à le remplacer par le non-terminal gauche correspondant. L'opération est donc inverse de celle la dérivation. La principale difficulté consiste à trouver le bon membre droit à appliquer. Elle est à la fois plus puissante mais aussi plus complexe à mettre en œuvre que l'analyse descendante (sans retour arrière). Cette analyse effectue un parcours postfixe de l'arbre.

4. Analyse descendante

L'algorithme d'analyse est récursif. Son problème est celui de la terminaison : il ne peut s'arrêter s'il y a une infinité d'appels emboîtés sans lecture.

4.1 Récursion gauche

Théorème : l'algorithme d'analyse descendante termine pour toute phrase en entrée (et tout choix des règles) d'un langage L(G) si et seulement si G n'est n'est pas *récursive gauche*, cad ne contient pas de règles de la forme :

$A \rightarrow^* A\alpha$

Par exemple, la grammaire suivante est récursive gauche :

GEA :

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid Id \mid Cste$

Néanmoins, il est possible d'éliminer la récursion gauche. La plus simple, est la récursivité gauche immédiate. Si l'on a une règle de la forme :

$A \rightarrow A\alpha \mid \beta$ où β ne commence pas par le symbole A, alors on peut éliminer la récursion gauche en remplaçant ces règles par :

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

Par exemple, la grammaire GEA peut être transformée en GEAL qui est la suivante :

GEAL

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid Id$

4.2 Principe de réalisation « à la main » d'un compilateur descendant

Idée la plus simple: considérer que chaque symbole non-terminal du langage représente une procédure, et implémenter directement cette procédure. De cette manière, la récursivité inhérente aux grammaires est directement réalisée par la récursivité du langage d'implémentation.

Supposons que l'on dispose d'une variable, `input-symbol`, qui représente le dernier symbole lu par l'analyseur lexical (représente en fait un index dans la table des symboles), et d'une procédure, `readToken`, qui lance l'analyse lexicale lit un mot, le place dans la table des symboles et retourne son index dans la table.

Dans ce cas, la grammaire GEAL peut être implémentée de la manière suivante:

4.2.1 Ecriture d'un analyseur pour expressions arithmétiques

E est décrit par Expression,

T est décrit par Terme

F est décrit par Facteur

```
Expression() {
   Terme();
   Exprime();
}

Exprime() {
   if (chaine(input_symbol)== "+") {
      readToken();
      Terme();
      Eprime();
   }
}

Terme () {
```

```
    Facteur();
    TermePrime();
}

TermePrime() {
    if (chaine(input_symbol == "*")) {
        readToken();
        Facteur();
        TermePrime();
    }
}

Facteur() {
    if (chaine(input_symbol == "id"))
        readToken(); // on a analyse un mot!
    else {
        if (type(input_symbol == LPAR) { // "("
            readToken();
            Expression();
            if (input_symbol == RPAR ) // ")"
                readToken();
            else erreur();
        } else erreur();
    }
}
```

4.2.2 Les problèmes rencontrés

Malheureusement, cette technique ne fonctionne pas toujours. En effet, pour que l'on puisse appliquer une analyse descendante récursive, il faut que toutes les règles de la forme:

$A \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$

puissent être telles que l'on sache toujours quelle règle choisir. Il faut que pour un non-terminal donné, toutes les règles associées commencent par un symbole terminal. Les règles doivent donc être de la forme:

$A \rightarrow a1\ \alpha_1 \mid a2\ \alpha_2 \mid ... \mid a3\ \alpha_n$

où

$A \rightarrow \alpha$ (pas d'autre règle ayant A comme membre gauche).

En pratique, les langage de programmation « bien écrits » des mots clés (ou des classes lexicales) permettant de distinguer parmi les règles sans problèmes. Par exemple, dans :

<instruction> → **for** <var> := <expr> **do** <instr> |
if <cond> **then** <expr> **else** <expr> |
while <cond> **do** <instruction> |
begin <instr-list> **end** |
<procedure>(< arg-list>) |

<ident> := < expr>

les mots clés permettent de reconnaître sans ambiguïté le type d'instruction. Il en est de même des mots clés **procedure** , **function** , **var** , **type** , **const** qui sont placés devant ce qu'ils introduisent. Les classes lexicales `<procedure>` et `<ident>` agissent comme des mots clés pour lever l'ambiguïté sur les règles.

Mais cela ne suffit pas toujours. Nous allons voir l'ensemble des opérations que l'on doit appliquer aux règles pour qu'elles aient le bon format.

Factorisation à gauche

Les règles de la forme : $A \rightarrow \alpha\beta \mid \alpha\delta$ sont transformées en :

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta \mid \delta$$

Expansion partielle

Principe: lorsque des règles commencent par des non-terminaux dans le membre droit, on expanse toutes ces règles jusqu'à obtenir des règles qui commencent par des terminaux, et ce de manière récursive.

Ainsi, les règles de la forme:

$$A \rightarrow B\alpha \mid C\beta$$
$$B \rightarrow aB' \mid B''$$
$$B'' \rightarrow bD \mid D'$$
$$D' \rightarrow eD''$$
$$C \rightarrow cC' \mid dC''$$

doivent être transformées en:

$$A \rightarrow aB'\alpha \mid bD\alpha \mid eD''\alpha \mid cC'\beta \mid dC''\beta$$

et on élimine les règles:

$$B \rightarrow aB' \mid B''$$
$$B'' \rightarrow bD \mid D'$$
$$D' \rightarrow eD''$$
$$C \rightarrow cC' \mid dC''$$

Par exemple, les règles suivantes:

<instruction> → <instr-if> < instr-for> < instr-while> <affect> < proc-call>

<instr-if> → **if** <cond> **then** <expr> **else** <expr>

<instr-for> → **for** <var> := <expr> **do** <instr>

<instr-while> → **while** <cond> **do** <instruction>

<instr-bloc> → **begin** <instr-list> **end**

<instr-callproc> → <procedure>(< arg-list>) |

<instr-affect> → <ident> := < expr>

doivent être transformées en l'instruction supérieure.

<instruction> → **for** <var> := <expr> **do** <instr> |
if <cond> **then** <expr> **else** <expr> |
while <cond> **do** <instruction> |
begin <instr-list> **end** |
<procedure>(< arg-list>) |
<ident> := < expr>

Traitement des règles vides

Les règles de type A → ε posent un problème particulier, car il faut tenir compte, lors de l'expansion partielle de ce qui suit ces non-terminaux.

Par exemple une grammaire de la forme

A → BB' | cD
B → ε | dE
B' → mM

se transforme en :

A → cD | dE | mM // cette dernière à cause de la règle B → ε.

Récursion

Rien de spécial à signaler:

Ex:

A → aB | CB'
C → bA | c
B → (A)
B' → d

se transforme en

A → a(A) | bA | c(A) | c'd

Plus complexe:

E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | Id

se transforme en:

E → (E) T'E' | Id T'E'
E' → +(E)T'E' | ε | + Id T'E'
T' → *(E) T' | ε | * Id T'

4.2.3 Application au if

On a vu qu'il était possible de transformer le if en une règle un peu plus complexe. Malheureusement, cette transformation n'est pas applicable dans le cas d'une analyse descendante récursive, car il n'existe pas de mécanisme d'expansion partielle et de factorisation à gauche permettant de lever l'ambiguïté.

<instruction> → <instr. assoc.> | < instr. non-assoc>

<instr. assoc.> → **if** <condit> **then** <instr. assoc.> **else** <instr. assoc> |
<autres instr.>

<instr. non-assoc> → **if** <condit> **then** <instruction> |
if <condit> **then** <instr. assoc.> **else** <instr. non-assoc>

La technique utilisée est en général assez simple: elle revient tout simplement à traiter l'instruction if de la manière suivante:

<instr-if> → **if** <cond> **then** <expr> **else** <expr> | **if** <cond> **then** <expr>

puis par factorisation à gauche d'obtenir:

<instr-if> → **if** <cond> **then** <expr> < suite-if>

<suite-if> → **else** <expr> | ε

puis de considérer, directement dans l'analyseur, que le < suite-if> se rattache au dernier < instr-if> rencontré. Cela n'est pas très propre, mais on fait de bons compilateurs avec de telles techniques. En fait, cela n'est pas trop grave du fait qu'on fait attention à ne traiter que des grammaires non-ambiguë sauf dans le cas du if qui est traité à part.

4.3 Analyseurs prédictifs

Il s'agit d'appliquer ce que nous avons vu à des analyseurs fonctionnant de manière tabulaire, c'est-à-dire sans utiliser la récursivité du langage d'implémentation de l'analyseur syntaxique.

Un analyseur prédictif se compose d'une pile (pour contenir les symbole en cours) et d'une table qui contient les relations entre terminaux courants, symboles terminaux en entrée et règles à appliquer.

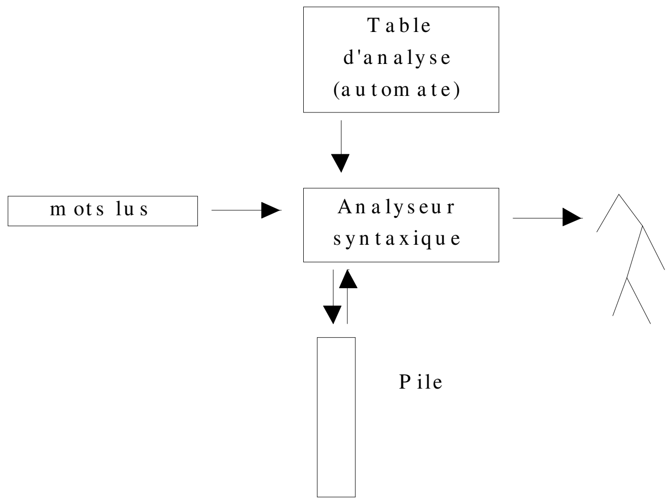


Figure 4. Un analyseur prédictif

4.3.1 Fonctionnement de l'analyseur

On suppose que l'analyseur lexical retourne le symbole '$' lorsque tous les caractères ont été lus (c'est le symbole de fin de fichier "réifié" comme un mot).

Mettre le symbole de démarrage sur la pile (appelons le '$' pour simplifier).
Tant que la pile n'est pas vide faire {
  soit X le sommet de la pile et *a* la valeur du symbole d'entrée.
  Si X est un symbole terminal (ou le symbole de démarrage)
    si X == a alors pop X et readToken()
    sinon erreur
  sinon // X est un symbole non-terminal
    si TableAnalyse[X,a]== $X \to Y_1 \ldots Y_k$ alors
      pop X
      push Y_k, Y_{k-1}, ... Y_1 // Y1 doit être sur le dessus
    sinon erreur
}

4.3.2 Construction de la table

Calculs de FIRST et FOLLOW

On définit les procédures FIRST et FOLLOW qui retournent les premiers non-terminaux associés à un symbole de la manière suivante:

FIRST(a) = {a}

FIRST(X) = $\{a \mid X \to a\alpha\} \cup$
$\{\text{FIRST}(Y_i) \mid X \to Y_1,..Y_k \text{ et } Y_1,..,Y_{i-1} \Rightarrow^* \varepsilon\} \cup$
$\{\varepsilon\}$ si $X \to \varepsilon$ ou si $X \to Y_1,..Y_k$ et $\varepsilon \in \text{FIRST}(Y_i)\ \forall\, Y_i$

FOLLOW(X) = FIRST(β) si $Y \to \alpha X \beta$ et $\beta \not\Rightarrow^* \varepsilon\ \cup$

FOLLOW(Y) si Y $\rightarrow \alpha X$ ou (Y $\rightarrow \alpha X\beta$ et $\varepsilon \in$ FIRST(β)) $\cup$
{\$} si X = S (le symbole initial)

Construction de la table d'analyse

Pour chaque production de la forme A $\rightarrow \alpha$
si a $\in$ FIRST(α) alors insérer A $\rightarrow \alpha$ dans TableAnalyse[A, a]
si $\varepsilon \in$ FIRST(α) alors insérer A $\rightarrow \alpha$ dans TableAnalyse[A, b]
pour chaque b $\in$ FOLLOW(A).

Exemple sur la grammaire GEAL:

E $\rightarrow$ TE'
E' $\rightarrow$ +TE' | ε
T $\rightarrow$ FT'
T' $\rightarrow$ *FT' | ε
F $\rightarrow$ (E) | Id
FIRST(E) = FIRST(T) = FIRST(F) = { (, Id}
FIRST(E') = {+, ε}
FIRST(T') = {*,ε}
FOLLOW(E) = FOLLOW(E') = {),\$}
FOLLOW(T) = FOLLOW(T') = {+,),\$}
FOLLOW(F) = {+, *,), \$}

Le tableau obtenu est le suivant:

	Id	**+**	*****	**(**	**)**	**$**
E	E $\rightarrow$ TE'			E $\rightarrow$ TE'		
E'		E' $\rightarrow$ +TE'			E' $\rightarrow \varepsilon$	E' $\rightarrow \varepsilon$
T	T $\rightarrow$ FT'			T $\rightarrow$ FT'		
T'		T' $\rightarrow \varepsilon$	T' $\rightarrow$ *FT'		T' $\rightarrow \varepsilon$	T' $\rightarrow \varepsilon$
F	F $\rightarrow$ Id			F $\rightarrow$ (E)		

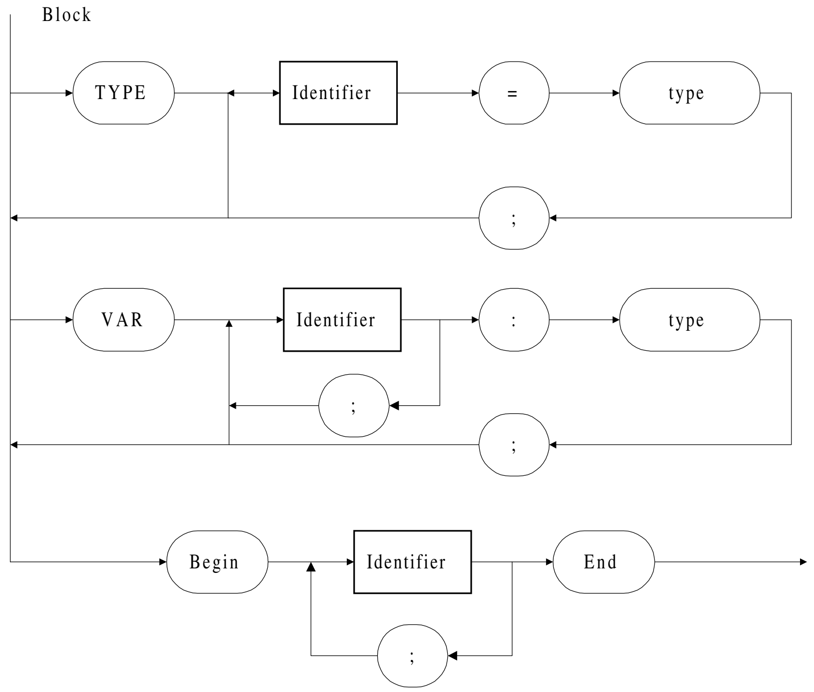
I l n'y a plus qu'à montrer que l'analyseur fonctionne sur une expression de la forme: (Id * Id + Id) * Id.

5. Diagrammes syntaxiques

Popularisés par N. Wirth avec Pascal, les diagrammes syntaxiques sont beaucoup plus pratiques à utiliser que les règles, tout en étant équivalent aux règles.

Les diagrammes s'expriment comme des ensembles de petits automates emboités.

Exemple:



Le diagramme correspond aux règles suivantes:

<block> → <type-bloc> | < var-bloc> | < inst-bloc>

<type-bloc> → Type < type-list> | ε

<type-list> → <type-decl> | < type-decl> < type-list>

<type-decl> → <id> = <type> ;

<var-bloc> → Var < var-decl-list> | ε

<var-decl-list> → <var-decl> | < var-decl> < var-decl-list>

<var-decl> → <id> : <type> ;

<ins-bloc> → Begin < inst-list> End

<inst-list> → <instruction> < inst-list> | ε

On peut