Spécification et traduction des Langages

Mathias Ettinger D'après les cours de L. Feraud

27 octobre 2009

Table des matières

Première partie
Spécification des langages

1 Approche générale du problème de la traduction

1.1 Introduction

Aux débuts de l'informatique, les premiers travaux étaient directement écrits en langage binaire. Pour palier au manque de convivialité de cette méthode, le langage assembleur a fait son apparition très vite (les premiers traducteurs aussi donc).

Ensuite, se sont développés les premiers langages universels (Fortran, Cobol, ...) qui nécessitent des traducteurs spécifiques pour chaque type de machine. Au débuts, cette traduction était faite à la main. Mais l'accroissement du "fossé sémantique" (distance entre le langage haut niveau et le langage machine) a vite rendu cette méthode très lourde. L'appel à des traductions automatiques est maintenant indispensable.

1.2 Deux approches de la traduction : interpréter ou compiler ?

Approche intuitive :

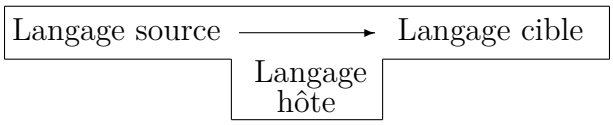
```
A:=2;
B:=3;
C:=4;
D:=A*B+C;
ECRIRE(D);
```

Lors de l'interprétation, une action est entreprise à chaque fois que c'est possible :
- 2 est mis dans une case mémoire correspondant à `A`.
- 3 est mis dans une case mémoire correspondant à `B`.
- 4 est mis dans une case mémoire correspondant à `C`.
- 10 est mis dans une case mémoire correspondant à `D`.
- 10 est affiché.

Pour la compilation, il s'agit d'une ré-écriture du code en langage machine :

```
Load = 2
Store A
Load = 3
Store B
Load = 4
Store C
Load A
Mult B
Add C
Store D
Print D
```

Lors d'une compilation, le langage à traduire est appelé langage source et le langage vers lequel on traduit est le langage cible. Le langage utilisé pour écrire le traducteur est le langage hôte.



1.3 Interprétation

Un interprète est un simulateur d'une machine pour le langage source. Un interprète contient un "évaluateur" qui fait le calcul des sous-parties du programme. L'interprète a un comportement itératif.

Boucle
- Lire un mot (sous-partie) du programme source.
- Evaluer mot.
- Imprimer résultat.

1.4 Compilation

Le but d'un compilateur est, en partant d'un programme `P` écrit en langage source, d'écrire un programme `P'` en langage cible qui a le même sens que `P`.

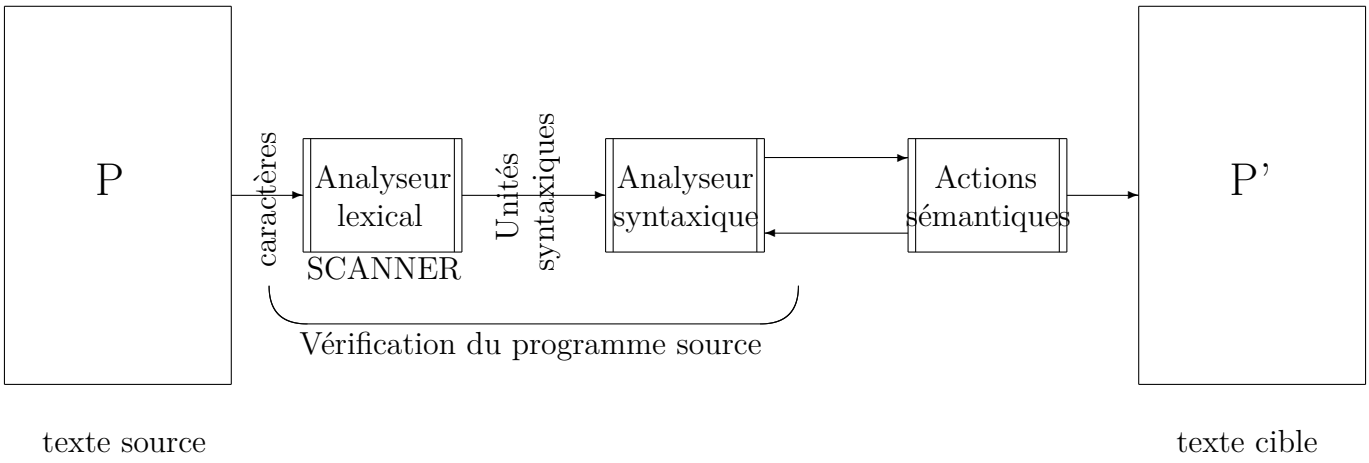


FIGURE 1 – Schématisation du processus de compilation

On peut aussi avoir recours à un optimiseur, en intéraction avec les actions sémantiques, pour améliorer les performances du programme cible.

Exemple d'un programme P

```
class example{
  int b[10,20];
  void spin(){
    int i;
    for(i=0; i<100; i++){_}
  }
}
```

- Analyse lexicale (reconnaissance des symboles et production de tokens)
  symboles : int, for, void, ..
  nombres : 10, 20, ..
  constructions du langage : ++, =, ..
- Analyse syntaxique (production d'un arbre syntaxique)

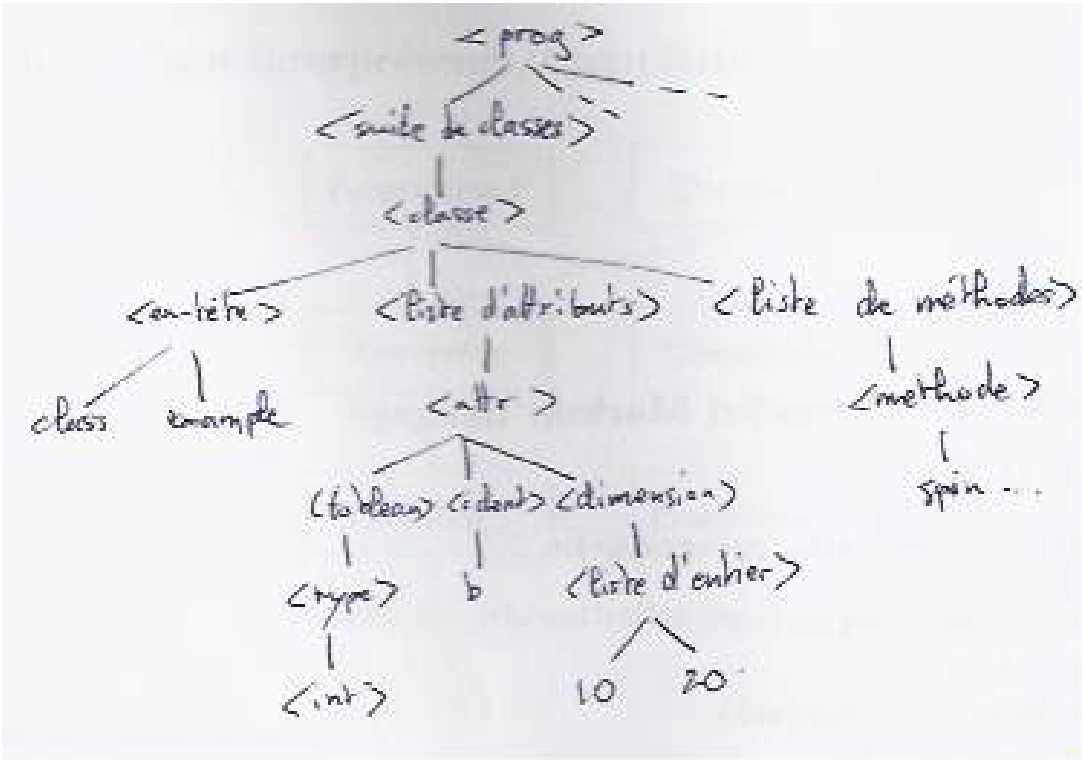


FIGURE 2 – Arbre syntaxique de la classe example

– Programme `P'` (bytecode)

```
 0 iconst 0    //empiler 0
 1 istore 1    //stocker 1 dans var. locale
 2 goto 8      //première exécution
 5 iincr1 1    //incrémenter de 1 la variable locale
 8 iload 1     //empiler la variable locale
 9 bipush 100 //empiler 100
11 if_cmplt 5 //comparer et boucler
14 return      //retourner vide
```

Exemple d'optimisation

```
for(i=0; i<100; i++){
  ...
  while(j<5){...}
}
```

sans optimisation

```
test 1: ...(i<100 ?)
        goto suite1
        ...
test 2: ...(j<5 ?)
        goto suite2
        ...
        goto test 2
suite2: goto test 1
suite1: ...
```

avec optimisation

```
test 1: ...(i<100 ?)
        goto suite1
        ...
test 2: ...(j<5 ?)
        goto test 1
        ...
        goto test 2
suite1: ...
```

1.5 Exemple du XML : eXtended Markup Langage

Data Type Definitions (DTD)

```
<! element livre (titre, ?preambule, paragraph+)>
<! element titre #PCDATA>
<! element paragraph (titre, section+)>
<! element section #PCDATA>
<! element preambule (section+, auteur)>
<! element auteur (nom, prenom)>
<! element nom #PCDATA>
<! element prenom #PCDATA>
```

Il s'agit en fait d'une grammaire :

```
LIVRE -> TITRE {PREAMBULE|λ} PARA
TITRE -> t
PARA -> TITRE SECTION {PARA|λ}
SECTION -> t' {SECTION|λ}
PREAMBULE -> SECTION AUTEUR
AUTEUR -> NOM PRENOM
NOM -> t''
PRENOM -> t'''
```

On représente ainsi tous les documents par un arbre :

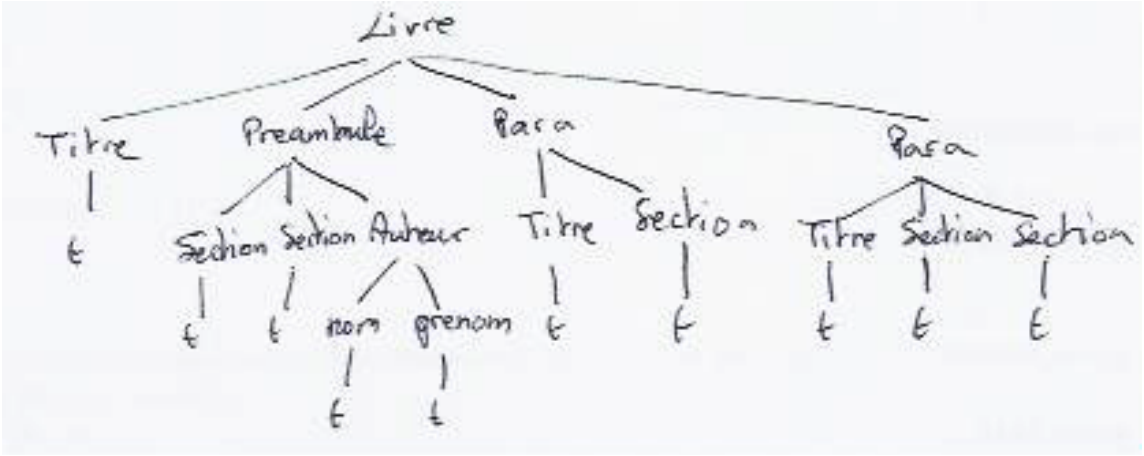


FIGURE 3 – Exemple de représentation d'un livre

Les "PARSERS" XML sont équivalents à des analyseurs syntaxiques agrémentés de quelques actions sémantiques.

1.6 Comparaison Interpréteurs/Compilateurs

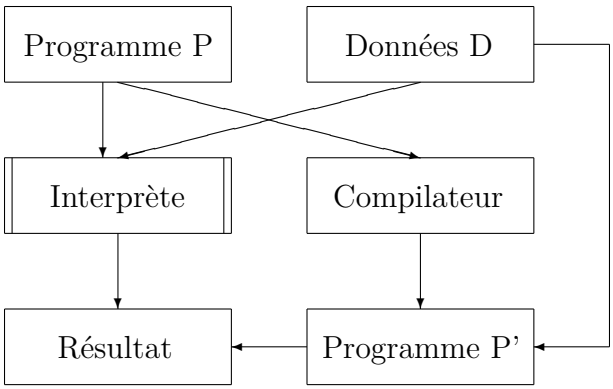


FIGURE 4 – Schématisation des deux processus

Avantages et Inconvénients

Les interprètes :
- ⊕ mise au point rapide.
- ⊕ convivial.
- ⊕ possibilité d'auto-modification du programme.
- ⊖ problème de place mémoire (programme + interprète).
- ⊖ pas de mise en facteur.

Les compilateurs :
- ⊕ meilleures performances.
- ⊕ mise en facteur.
- ⊕ optimisation sophistiquée.
- ⊖ peu d'interaction et de convivialité.
- ⊖ manque de souplesse.

2 Rappels sur les langages et grammaires (le problème de l'analyse syntaxique)

2.1 Quelques définitions

a. Soit un alphabet Σ.
Σ^* est l'ensemble des mots qu'on peut former sur Σ. L est un langage sur l'alphabet Σ ssi $L \subset \Sigma^*$.

b. Soit deux alphabets, Σ et N, un ensemble de règles P et un axiome S.
Une grammaire est $G=(\Sigma, N, P, S)$ ssi $\Sigma \cap N = \emptyset$ et $S \in N$ avec Σ l'alphabet des terminaux et N celui des non-terminaux.

Exemple :

$$\Sigma = \{a, b, c\} ; N = \{S, T\}$$
$$P: \begin{cases} S \to & aSb \\ S \to & cT \\ T \to & aTc \\ T \to & c \end{cases}$$

Conventions :
- Les lettres latines minuscules designent les terminaux ($\in \Sigma$).
- Les lettres latines majusculent désignent les non-terminaux ($\in$ N).
- Les lettres grecques désignent soit un terminal soit un non-terminal.

c. Dérivation :
· u se dérive directement en v ($u \mapsto v$) ssi $u = u_1 A v_1$, $v = u_1 \alpha v_1$ et $(A \to \alpha) \in P$.
· u se dérive en v ($u \overset{*}{\mapsto} v$) ssi soit u = v, soit $\exists u_1, u_2, \ldots, u_k$ tq $u = u_1$, $v = u_k$ et $\forall i < k, u_i \mapsto u_{i+1}$.

Exemple :

$$\begin{array}{ccc} & aTcS \overset{*}{\mapsto} accaccb & \\ \swarrow & & \searrow \\ accS \mapsto accaSb & & \mapsto accacTb \end{array}$$

d. Langage engendré par une grammaire :
Soit la grammaire G=(Σ, N, P, S). Le langage engendré par G est $L(G) = \{w \in \Sigma^* \ / \ S \overset{*}{\mapsto} w\}$

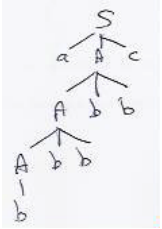
Exemple du langage Java :
Σ = {if, do, void, class, ..., ++, =, ...}
N = {<affectation>, <exp>, <classe>, <boucle>, ...}
Règles : <affectation> → <id> = <exp>
...

Remarque : Un langage peut être engendré par plusieurs grammaires

$$1: \begin{cases} S & \to aAc \\ A & \to Abb \\ A & \to b \end{cases} \qquad 2: \begin{cases} S & \to aAc \\ A & \to bAb \\ A & \to b \end{cases} \qquad 3: \begin{cases} S & \to aAc \\ A & \to bbA \\ A & \to b \end{cases}$$

e. Arbre syntaxique
Il s'agit de la trace de l'emploi des règles de grammaires pour engendrer un mot.

Exemple : D'après la grammaire 1, un arbre syntaxique de abbbbbc est



f. Équivalence grammaire/langage
Soit la grammaire G=(Σ, N, P, S). À chaque règle $A \to \alpha_1|\alpha_2|\cdots|\alpha_k$ de P est associée une équation de la forme $L_A = L_{\alpha_1} + L_{\alpha_2} + \ldots + L_{\alpha_k}$.
On a donc $L(G) = L_S$.

Exemple sur la grammaire 1 :

$$\begin{cases} L_S = & aL_Ac \\ L_A = & L_Abb + b \end{cases}$$

$L_A = L_A$bb + b est une équation linéaire soluble par Arden. Donc L_A = b(bb)* et L_S = ab(bb)*c.

Autre exemple :

$$\begin{cases} S & \to aSb \\ S & \to RS|cR \\ R & \to SaR|b \end{cases} \Leftrightarrow \begin{cases} L_S = aL_Sb + L_RL_S + cL_R \\ L_R = L_SaL_R + b \end{cases}$$

Y a-t-il une solution ? Comment la calculer ?

Théorème de Tarski sur l'existence d'un point fixe

Définitions :

(a) Soit (E, ≤) un ensemble ordonné et ⊥ son plus petit élément.
(b) Si toute partie (finie ou infinie) de E admet une borne supérieure, alors E est dit supcomplet.
(c) Soit φ: E → E, φ est monotone ssi $\forall a, b \in E, a \leq b \Rightarrow \varphi(a) \leq \varphi(b)$.
(d) Soit φ: E → E, φ est continue ssi $\forall E_i \in E, \varphi(\sup E_i) = \sup(\varphi(E_i))$.

Théorème : Si E est supcomplet et f: E → E est monotone et continue, alors f admet un plus petit point fixe x = f(x) = sup($f^n(\perp)$), n ∈ ℕ.

Démonstration : Soit x = sup($f^n(\perp)$) avec f continue et n ∈ ℕ.
f(x) = f(sup($f^n(\perp)$)) = sup($f^{n+1}(\perp)$) = sup($f^n(\perp)$) = x.

Unicité : Soit y un autre point fixe.

$$\begin{aligned} \perp = f^0(\perp) &\leq y \\ f^1(\perp) &\leq f(y) = y \\ f^2(\perp) &\leq f(y) = y \\ &\vdots \\ f^n(\perp) &\leq f(y) = y \end{aligned}$$

À la limite, x = sup($f^n(\perp)$) ≤ f(y) = y. Donc x est le plus petit point fixe.

Pour les langages, $\perp = \emptyset$. De plus, l'union des langages (+) est monotone et continue. On a donc $x = f(x) \Rightarrow x = \sup(f^n(\emptyset))$, $n \in \mathbb{N}$
Exemple : $L_A = f(L_A) = L_A bb + b$. Donc $L_A = \sup(f^n(\emptyset))$, $n \in \mathbb{N}$.

$$\begin{aligned} n = 1 &: f(\emptyset) = \emptyset bb + b = b \\ n = 2 &: f^2(\emptyset) = bbb + b \\ n = 3 &: f^3(\emptyset) = (bbb + b)bb + b = b^5 + b^3 + b = b(b^4 + b^2 + \lambda) \\ &\vdots \\ n &: f^n(\emptyset) = b((bb)^{n-1} + (bb)^{n-2} + \cdots + \lambda) \end{aligned}$$

Ainsi, par extrapolation, $L_A = b(bb)^*$

2.2 Analyse syntaxique

Faire l'analyse syntaxique de w avec la grammaire G=(Σ, N, P, S) c'est répondre aux deux questions :
– $w \in L(G)$?
– Quelle est la suite des dérivations utilisées pour engendrer w ?

2.2.1 Analyse syntaxique descendante

Le principe est de construire un arbre de dérivation depuis la racine vers les feuilles.

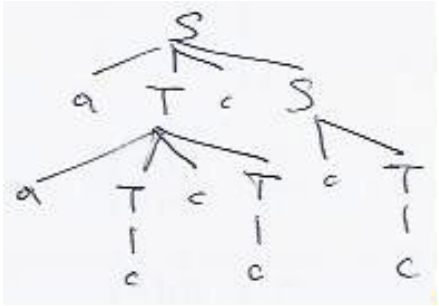
$$S \overset{*}{\mapsto} w : S \mapsto u_1 \mapsto u_2 \mapsto \cdots \mapsto u_n = w$$

Pour trouver u_{i+1} à partir de u_i, on va utiliser la dérivation la plus à gauche.

Exemple :

$$\begin{cases} S & \to aTcS \\ S & \to cT \\ T & \to aTcT \\ T & \to c \end{cases}$$

Reconnaissance de aacccccc : S ↦ aTcS ↦ aaTcTcS ↦ aaccTcS ↦ aaccccS ↦ aacccccT ↦ aacccccc.



Cette analyse peut se faire par un automate à pile : DESC={ {q}, Σ, Σ ∪ N, δ, q, S, ∅ }.

Rappel : AP = {ensemble des états, alphabet d'entrée, alphabet de pile, fonction de transition, état initial, initial dans la pile, états finals}

Le principe d'un automate par pile vide traduisant une grammaire est :

1. δ(q, λ, A) contient (q, α) s'il existe une règle A $\to \alpha$.
2. $\forall a \in \Sigma$, δ(q, a, a) = (q, λ).

La fonction de transition pour $\begin{cases} \mathrm{S} & \to \mathrm{T}a \\ \mathrm{T} & \to ab\mathrm{T}|\lambda \end{cases}$ est $\delta =$

1. δ(q, λ, S) = (q, Ta).
2. δ(q, λ, T) = { (q, abT), (q, λ) }.
3. δ(q, a, a) = (q, λ).
4. δ(q, b, b) = (q, λ).

Reconnaissance de ababa : (q, ababa, S) $\overset{1}{\vdash}$ (q, ababa, Ta) $\overset{2.1}{\vdash}$ (q, ababa, abTa) $\overset{3}{\vdash}$ (q, baba, bTa) $\overset{4}{\vdash}$ (q, aba, Ta) $\overset{2.1}{\vdash}$ (q, aba, abTa) $\overset{3}{\vdash}$ (q, ba, bTa) $\overset{4}{\vdash}$ (q, a, Ta) $\overset{2.2}{\vdash}$ (q, a, a) $\overset{3}{\vdash}$ (q, λ, λ).

2.2.2 Analyse syntaxique ascendante

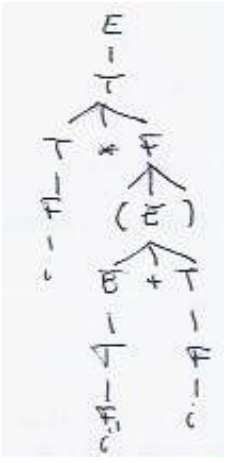
Le but est de construire un arbre de dérivation des feuilles vers la racine.

$$\mathrm{S} \overset{*}{\mapsto} \mathrm{w} : \mathrm{S} \mapsto \mathrm{u}_1 \mapsto \mathrm{u}_2 \mapsto \cdots \mapsto \mathrm{u}_n = \mathrm{w}$$

Pour déterminer u_i à partir de u_{i+1}, on va utiliser les réductions les plus à gauche (réduire α en A si une règle A $\to \alpha$ existe). Le réduction la plus à gauche est équivalente à la dérivation la plus à droite.

Exemple : Reconnaissance de i*(i+i) $\in$ L(E) où $\begin{cases} \mathrm{E} & \to \mathrm{E}+\mathrm{T}|\mathrm{T} \\ \mathrm{T} & \to \mathrm{T}*\mathrm{F}|\mathrm{F} \\ \mathrm{F} & \to i|(\mathrm{E}) \end{cases}$

E $\Rightarrow$ T $\Rightarrow$ T*F $\Rightarrow$ T*(E) $\Rightarrow$ T*(E+T) $\Rightarrow$ T*(E+F) $\Rightarrow$ T*(E+i) $\Rightarrow$ T*(T+i) $\Rightarrow$ T*(F+i) $\Rightarrow$ T*(i+i) $\Rightarrow$ F*(i+i) $\Rightarrow$ i*(i+i).



Cette analyse peut aussi se faire par un automate à pile : ASC={ {q, r}, Σ, $\Sigma \cup$ N$\cup${\$}, δ, q, \$, r } [1].

1. Pour ce genre d'automates, le fond de pile est à gauche et on empile vers la droite.

Le principe d'un automate à pile pour une analyse ascendante est :

1. $\forall a \in \Sigma$, δ(q, a, λ) = (q, a).
2. δ(q, λ, α) contient (q, A) pour chaque règle A $\rightarrow \alpha$.
3. δ(q, λ, \$S) = (r, λ).

La fonction de transition pour $\begin{cases} \mathrm{S} & \rightarrow \mathrm{T}a \\ \mathrm{T} & \rightarrow ab\mathrm{T}|\lambda \end{cases}$ est donc :

1. δ(q, a, λ) = (q, a).
2. δ(q, b, λ) = (q, b).
3. δ(q, λ, abT) = (q, T).
4. δ(q, λ, λ) = (q, T).
5. δ(q, λ, Ta) = (q, S).
6. δ(q, λ, \$S) = (r, λ).

Reconnaissance de ababa : (q, ababa, \$) $\overset{1}{\vdash}$ (q, baba, \$a) $\overset{2}{\vdash}$ (q, aba, \$ab) $\overset{1}{\vdash}$ (q, ba, \$aba) $\overset{2}{\vdash}$ (q, a, \$abab) $\overset{4}{\vdash}$ (q, a, \$ababT) $\overset{3}{\vdash}$ (q, a, \$abT) $\overset{3}{\vdash}$ (q, a, \$T) $\overset{1}{\vdash}$ (q, λ, \$Ta) $\overset{5}{\vdash}$ (q, λ, \$S) $\overset{6}{\vdash}$ (r, λ, λ).

2.2.3 Le problème du déterminisme

Quelle que soit la forme choisie pour l'analyse syntaxique, les automates à piles généraux sont indéterministes (il y a donc possibilité de retours arrières). Cela pose d'énormes problèmes de performances. Il faut donc utiliser des automates (à piles) qui sont K-prédictifs.

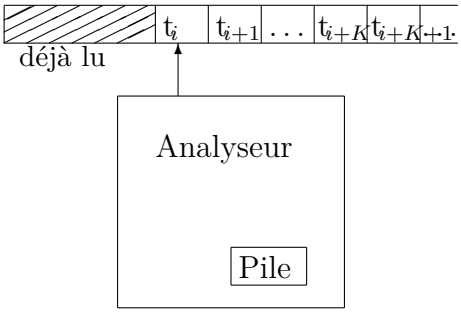


FIGURE 5 – Automate à pile

Un automate à pile K-prédictif permet, à partir d'une pile d'analyse et de K symboles sur le ruban, d'avoir un comportement déterministe.

2.3 Simplification des grammaires

2.3.1 Élimination des symboles improductif

Pour une grammaire G=(Σ, N, P, S), un symbole X $\in$ N est improductif ssi $\nexists \alpha \in \Sigma^*, \mathrm{X} \overset{*}{\mapsto} \alpha$. Pour éliminer les symboles improductifs, on va utiliser un algorithme de construction de l'ensemble des symboles productifs.

début
$i := 0$;
$W_i := \{\mathrm{X} \in \mathrm{N}/\mathrm{X} \mapsto \alpha, \alpha \in \Sigma^*\}$;
répéter
$i++$;
$W_i := W_{i-1} \cup \{\mathrm{X} \in \mathrm{N}/\mathrm{X} \mapsto \alpha, \alpha \in (\Sigma \cup W_{i-1})^*\}$;
jusqu'à$(W_i == W_{i-1})$
fin

Exemple : La construction successive des symboles productifs de la grammaire $\begin{cases} \mathrm{S} & \to a|A \\ \mathrm{A} & \to B|bD|Ec|F \\ \mathrm{B} & \to bA \\ \mathrm{D} & \to a|c \\ \mathrm{E} & \to d \end{cases}$ donne :

- $\mathrm{W}_0 = \{\mathrm{S}, \mathrm{D}, \mathrm{E}\}$
- $\mathrm{W}_1 = \{\mathrm{S}, \mathrm{D}, \mathrm{E}\} \cup \{\mathrm{A}\} = \{\mathrm{S}, \mathrm{A}, \mathrm{D}, \mathrm{E}\}$
- $\mathrm{W}_2 = \{\mathrm{S}, \mathrm{A}, \mathrm{D}, \mathrm{E}\} \cup \{\mathrm{B}\} = \{\mathrm{S}, \mathrm{A}, \mathrm{B}, \mathrm{D}, \mathrm{E}\}$
- $\mathrm{W}_3 = \{\mathrm{S}, \mathrm{A}, \mathrm{B}, \mathrm{D}, \mathrm{E}\}$

On peut prouver que L(G) = L(G') avec G'=(Σ, N $\cap$ W_i, P', S).

2.3.2 Élimination des symboles inaccessibles à partir de l'axiome

Un symbole X $\in \Sigma \cup$N est inaccessible à partir de l'axiome S ssi il n'existe pas de dérivation à partir de S telle que $\mathrm{S} \overset{*}{\mapsto} \alpha \mathrm{X} \beta$ avec $\alpha, \beta \in (\Sigma \cup \mathrm{N})^*$.

L'algorithme de construction de l'ensemble des symboles accessibles à partir de l'axiome est :

début
$i := 0$;
$W_i := \{\mathrm{S}\}$;
répéter
$i++$;
$W_i = W_{i-1} \cup \{\mathrm{X} \in (\Sigma \cup \mathrm{N})/\exists \mathrm{A} \in W_{i-1}$ et $\mathrm{A} \to \alpha \mathrm{X} \beta$ avec $\alpha, \beta \in (\Sigma \cup \mathrm{N})^*\}$;
jusqu'à$(W_i == W_{i-1})$
fin

Exemple : La construction successive des symboles accessibles à partir de l'axiome de la grammaire $\begin{cases} \mathrm{S} & \to a|A \\ \mathrm{A} & \to AB \\ \mathrm{B} & \to b \\ \mathrm{C} & \to c|d \end{cases}$ donne :

- $\mathrm{W}_0 = \{\mathrm{S}\}$
- $\mathrm{W}_1 = \{\mathrm{S}\} \cup \{\mathrm{A}, \mathrm{a}\} = \{\mathrm{S}, \mathrm{A}, \mathrm{a}\}$
- $\mathrm{W}_2 = \{\mathrm{S}, \mathrm{A}, \mathrm{a}\} \cup \{\mathrm{B}\} = \{\mathrm{S}, \mathrm{A}, \mathrm{B}, \mathrm{a}\}$
- $\mathrm{W}_3 = \{\mathrm{S}, \mathrm{A}, \mathrm{B}, \mathrm{a}\} \cup \{\mathrm{b}\} = \{\mathrm{S}, \mathrm{A}, \mathrm{B}, \mathrm{a}, \mathrm{b}\}$
- $\mathrm{W}_4 = \{\mathrm{S}, \mathrm{A}, \mathrm{B}, \mathrm{a}, \mathrm{b}\}$

On peut prouver que L(G) = L(G') avec G'=($\Sigma \cap W_i$, N $\cap$ W_i, P', S).

2.3.3 Grammaires augmentées

Soit G=(Σ, N, P, S) un grammaire. La grammaire augmentée à partir de G est G'=(Σ, N, P $\cup$ $\{S' \to S\$^K\}$, S').

Cela permet, dans le cadre des automates K-prédictifs, de toujours pouvoir lire K symboles. Désormais, toutes les grammaires seront (implicitement) augmentées.

3 Grammaires LL(K)

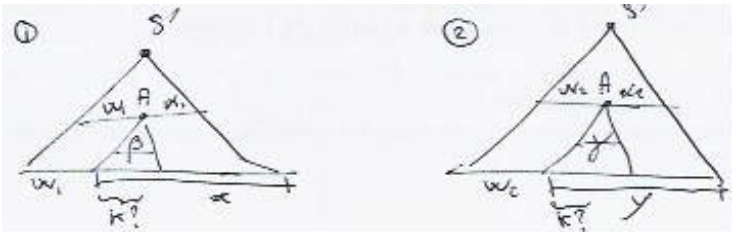
3.1 Introduction

On se place dans le cadre d'une analyse syntaxique descendante (construction d'un arbre syntaxique depuis la racine vers les feuilles).

On a donc une grammaire augmentée G=(Σ, N, P, S') et un mot w sur l'alphabet Σ (w $\in \Sigma^*$). L'on désire savoir si w $\in$ L(G) et, si c'est le cas, dresser son arbre syntaxique.

Exemple : reconnaissance de aabb avec la grammaire $\{S \to aSbS|\lambda$
$S \mapsto aSbS \mapsto aaSbSbS \mapsto aabSbS \mapsto aabbS \mapsto aabb$.

Les grammaires LL(K) [2] vont fonctionner avec des automates K-prédictifs. Le principe est le suivant :
Pour une règle de production $A \to \beta|\gamma$ ($\beta \neq \gamma$) Suivant la taille de K, on peut observer sur le ruban



les premiers symboles de β ou γ, voire les symboles qui suivent A. Ces deux ensembles de symboles sont nommés :

- first_K pour les K premiers symboles de β ou γ
- follow_K pour les K premiers symboles suivant A.

3.2 Définition et calcul de first_K et follow_K

3.2.1 First_K

Définition :

$$\text{first}_K(\alpha) = \{\ w \in \Sigma^*\ /\ |w| \leq k \text{ et } \begin{cases} \alpha \overset{*}{\mapsto} w & \text{si } |w| < k \\ \alpha \overset{*}{\mapsto} w\beta, \beta \in (\Sigma \cup N)^* & \text{si } |w| = k \end{cases} \}$$

$\text{first}_K(\lambda) = \{\lambda\}$

$\forall\ t \in \Sigma,\ \text{first}_K(t) = \{t\}$

$$\forall\ u \in \Sigma^*,\ \text{first}_K(u) = \begin{cases} \{u\} & \text{si } |u| \leq K \\ \{w\} & \text{si } u = wx \text{ avec } |w| = K \end{cases}$$

2. pour : Left (lecture de gauche à droite) Left (dérivation la plus à gauche) et K symboles lus à la fois.

Exemple : $G_1 = (\{a, b\}, \{S, S', A, B\}, P, S')$ avec

$$P : \begin{cases} S' & \to S\$^3 \\ S & \to aAaB \\ S & \to bAbB \\ A & \to a \\ A & \to ab \\ B & \to aB \\ B & \to a \end{cases}$$

- $first_1(aAaB) = \{a\}$
- $first_2(aB) = \{aa\}$
- $first_3(aAaB) = \{aaa, aab\}$
- $first_3(a) = \{a\}$

3.2.2 Follow$_K$

Définition :
Soit $A \in N$, $follow_K(A) = \{ w \in \Sigma^* \ / \ S \overset{*}{\mapsto} uA\alpha$ avec $u \in \Sigma^*$, $\alpha \in (\Sigma \cup N)^*$ et $w \in first_K(\alpha)\}$

Exemples : $G_2 = (\{a, b\}, \{S, S'\}, P, S')$ avec

$$P : \begin{cases} S' & \to S\$^2 \\ S & \to aaSbb \\ S & \to a \\ S & \to \lambda \end{cases}$$

- $follow_1(S) = \{\$, b\}$
- $first_2(aB) = \{\$\$, bb\}$

Pour G_1 :
- $follow_1(A) = \{a, b\}$
- $follow_1(B) = \{\$\}$
- $follow_2(A) = \{aa, ba\}$
- $follow_3(A) = \{aa\$, aaa, ba\$, baa\}$

3.3 Les grammaires fortement LL(K)

Avec des grammaires LL(K) il est possible de réécrire A de manière déterministe en connaissant les K symboles à venir sur le ruban. Si elles sont fortement LL(K), on peut le faire quel que soit le contexte.

Définitions *cf* polycopié.

Exemple avec G_2 :
- 2-lookahead(S→aaSbb) = $first_2(aaSbb\, follow_2(S)) = \{aa\}$
- 2-lookahead(S→a) = $first_2(a\, follow_2(S)) = first_2(a.\{\$\$, bb\}) = \{a\$, ab\}$
- 2-lookahead(S→ λ) = $first_2(follow_2(S)) = \{\$\$, bb\}$

3.4 Analyseurs fortement LL(K)

L'analyseur est un automate à pile piloté par une matrice M (table d'analyse) qui, à un symbole à venir et au sommet de pile associe une unique action.

Les lignes de M correspondent aux terminaux et aux non-terminaux, les colones sont les mots appartenant aux ensembles K-lookahead.

Les actions associées sont :
- empiler β pour M(w,A) si w $\in$ K-lookahead(A$\rightarrow \beta$)
- dépiler a (pop) pour M(w,a) si a est préfixe de w
- accepter le mot pour M(λ,λ)

Exemple :

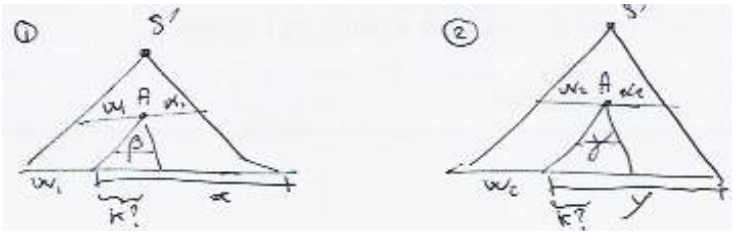
	aa	ab	a$	bb	$$	λ	a	b	$
S	aaSbb	a	λ	a	λ				
a	pop	pop	pop				pop		
b				pop				pop	
$					pop				pop
λ						Acc.			

TABLE 1 – Table d'analyse reconnaissant L(G_2)

Reconnaissance de aabb (le troisième champ est la trace des règles utilisées)
(aaabb$$, S$$, λ) $\vdash$ (aaabb$$, aaSbb$$, 1) $\vdash$ (aabb$$, aSbb$$, 1) $\vdash$ (abb$$, Sbb$$, 1) $\vdash$ (abb$$, abb$$, 1-2) $\vdash$ (bb$$, bb$$, 1-2) $\vdash$ (b$$, b$$, 1-2) $\vdash$ ($$, $$, 1-2) $\vdash$ ($, $, 1-2) $\vdash$ (λ, λ, 1-2) $\Rightarrow$ Accepté.

3.5 Grammaires faiblement LL(K)

Avec des grammaires LL(K) il est possible de réécrire A de manière déterministe en connaissant les K symboles à venir sur le ruban. Si elles sont faiblement LL(K), on peut le faire si on sait d'où on vient (dans un même contexte).



Définitions *c.f.* polycopié.

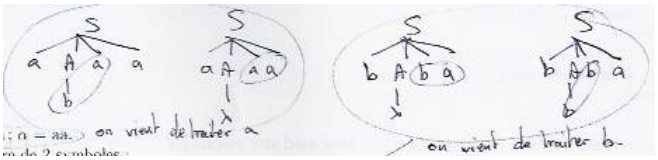
On peut légitiment se poser la question de savoir s'il y a un lien entre les grammaires fortement LL(K) et les grammaires faiblement LL(K).

Théorème Toute grammaire fortement LL(K) est faiblement LL(K).
On a donc, pour K fixé, fortement LL(K) $\subset$ faiblement LL(K).

Exemple : La grammaire G_1 dont les règles de production sont $\begin{cases} S' & \to S\$\$ \\ S & \to aAaa \\ S & \to bAba \\ A & \to b \\ A & \to \lambda \end{cases}$ est elle LL(2) ?

- $follow_2(A) = \{aa, ba\}$
- 2-lookahead($A \to b$) = $first_2(b \cdot \{aa, ba\}) = \{ba, bb\}$
- 2-lookahead($A \to \lambda$) = $first_2(follow_2(A)) = \{aa, ba\}$

L'intersection des 2-lookahead des règles de productions issues de A n'est pas vide. G_1 n'est donc pas fortement LL(2).



w = a ; α = aa.
Lecture de 2 symboles :
– aa : A→ λ
– ba : A→b
w = b ; α = ba.
Lecture de 2 symboles :
– ba : A→ λ
– bb : A→b

3.6 Le cas K = 1

Théorème Toute grammaire faiblement LL(1) est fortement LL(1).

Démonstration par l'absurde
Hypothèse : Soit G faiblement LL(1) et non fortement LL(1).
Donc, $\exists A \to \beta$, $A \to \gamma$, $\beta \neq \gamma$ w_1,w_2,α_1,α_2 tels que :

1. $S' \overset{*}{\mapsto} w_1 A \alpha_1 \mapsto w_1 \beta \alpha_1 \overset{*}{\mapsto} w_1 t_1 \alpha_1 \overset{*}{\mapsto} w_1 t_1 u_1 = w_1 x$
2. $S' \overset{*}{\mapsto} w_2 A \alpha_2 \mapsto w_2 \gamma \alpha_2 \overset{*}{\mapsto} w_2 t_2 \alpha_2 \overset{*}{\mapsto} w_2 t_2 u_2 = w_2 y$
3. $first_1(t_1 u_1) = first_1(t_2 u_2)$

a) $t_1 = t_2 = \lambda$ est impossible sinon G serait ambigüe.
On suppose donc $t_1 \neq \lambda$. Ainsi $first_1(t_1 u_1) = first_1(t_1)$.

b) · $S' \overset{*}{\mapsto} w_2 A \alpha_2 \mapsto w_2 \beta \alpha_2 \overset{*}{\mapsto} w_2 t_1 \alpha_2 \overset{*}{\mapsto} w_2 t_1 u_2$
· $S' \overset{*}{\mapsto} w_2 A \alpha_2 \mapsto w_2 \gamma \alpha_2 \overset{*}{\mapsto} w_2 t_2 \alpha_2 \overset{*}{\mapsto} w_2 t_2 u_2$
$first_1(t_1 u_2) = first_1(t_1) = first_1(t_1 u_1) =^3 first_1(t_2 u_2)$
Or $first_1(t_1 u_2) = first_1(t_1 u_1) \Rightarrow^4 \beta = \gamma$
$\Rightarrow$ Contradiction.

3. par hypothèse non fortement LL(1)
4. par hypothèse faiblement LL(1)

4 Mise en œuvre de l'analyse LL(1) par un programme de descente récursive

Principe :
À un non-terminal, on associe une procédure en langage récursif qui est "image de la règle".

Outils :
- Analyseur lexical : SCAN
- Unité syntaxique courante : NEXT_S
- Analogue de "pop" : SKIP

```
procedure SKIP(t : u.s.)
begin
  if NEXT_S = t then SCAN
  else ERROR endif
end
```

4.1 Construction de l'analyseur descendant récursif

À chaque règle de production A$\rightarrow \beta$ on associe :

```
procedure A
begin
```

image de β

```
end
```

Pour construire l'image de β, il suffit de :
- Associer SKIP(t) à tout t $\in \Sigma$;
- Associer l'appel de la procédure A à tout A $\in$ N ;
- Associer l'instruction vide NULL à λ.

Note : "L'image" de $S_1S_2\cdots S_n$ est associée à image(S_1)·image(S_2)$\cdots$image(S_n).

Exemple avec la règle de production A$\rightarrow t_1t_2BCt_3$:

```
procedure A
begin
  SKIP(t1); SKIP(t2); B; C; SKIP(t3)
end
```

Les règles de productions de la forme A$\rightarrow \beta_1|\cdots|\beta_n$ sont associées à :

```
switch(NEXT_S)
```

1-lookahead(A$\rightarrow \beta_1$) : *image de* β_1
$\vdots$
1-lookahead(A$\rightarrow \beta_n$) : *image de* β_n

```
  others: ERROR
endswitch
```

Exemple avec les règles de production $\begin{cases} S & \to aAS \\ S & \to b \\ A & \to a \\ A & \to bSaA \end{cases}$

```
procedure S
begin
  switch(NEXT_S)
    a: SKIP(a); A; S;
    b: SKIP(b);
    others: ERROR
  endswitch
end

procedure A
begin
  switch(NEXT_S)
    a: SKIP(a);
    b: SKIP(b); S; SKIP(a); A;
    others: ERROR
  endswitch
end
```

Le programme principal permet d'armer l'analyseur qui va mettre la première lettre du mot dans NEXT_S puis va lancer la reconnaissance du mot :

```
procedure AnalSynt
begin
  SCAN; S
end
```

Exemple :

$$\begin{cases} < programme > & \to \text{program} < suitedecl > \text{begin} < suiteinst > \text{end} \\ < suiteinst > & \to \ < inst > | < inst > \ ; < suiteinst > \\ < inst > & \to \text{if} < exp > \text{then} < suiteinst > \text{else} < suiteinst > \text{endif}| \\ & \ \text{while} < exp > \text{loop} < suiteinst > \text{endloop}|\text{repeat} < suiteinst > \text{until} < exp > \text{endloop} \\ < suitedecl > & \to \cdots \\ < exp > & \to \cdots \end{cases}$$

Les règles de production de $< suiteinst >$ induisent une ambiguïté. Levons la :
$< suiteinst > = < inst > + < inst >;< suiteinst > = < inst >< inst' >$
où $< inst' > = \lambda + ;< suiteinst > \Rightarrow < inst' > \to \lambda| \, ; < suiteinst >$.

Vérifions que la grammaire est toujours LL(1) :

- 1-lookahead($< inst' > \to \lambda$) = follow$_1$($< inst' >$) = follow$_1$($< suiteinst >$) = {end, else, endif, endloop, until}.
- 1-lookahead($< inst' > \to ;< suiteinst >$) = { ;}.

L'intersection des 1-lookahead des règles de production provenant de $< inst' >$ est vide, on a donc toujours une grammaire LL(1).

```
procedure PROGRAMME
begin
  SKIP(program); SUITE_DECL; SKIP(begin); SUITE_INST; SKIP(end)
end

procedure SUITE_INST
begin
  INST; INST_PRIME
end

procedure INST_PRIME
begin
  switch(NEXT_S)
    ';': SKIP(';'); SUITE_INST;
    end, else, endif, endloop, until: NULL;
    others: ERROR
  endswitch
end

procedure INST
begin
  switch(NEXT_S)
    if: SKIP(if); EXP; SKIP(then); SUITE_INST; SKIP(else); SUITE_INST; SKIP(endif);
    while: SKIP(while); EXP; SKIP(loop); SUITE_INST; SKIP(endloop);
    repeat: SKIP(repeat); SUITE_INST; SKIP(until); EXP; SKIP(endloop);
    others: ERROR
  endswitch
end
```

5 Grammaires LR(K)

5.1 Introduction

On se place dans le cadre d'une analyse ascendante (construction d'un arbre des feuilles vers la racine).

Le fonctionnement va être symétrique à ce qu'on a vu pour les grammaires LL(K) : on utilise les parties droites de règles pour réduire un manche au sommet de pile en la partie gauche.

Le processus de la réduction du manche le plus à gauche est équivalent à la notation de la dérivation la plus à droite.

Exemple avec la grammaire des expressions G_{EXP} $\begin{cases} E & \rightarrow E + T|T \\ T & \rightarrow T * F|F \\ F & \rightarrow i|(E) \end{cases}$

$E \Rightarrow E+T \Rightarrow E+F \Rightarrow E+i \Rightarrow T+i \Rightarrow T*F+i \Rightarrow T*i+i \Rightarrow F*i+i \Rightarrow i*i+i$

Pour traiter ces grammaires, on va utiliser un analyseur K-prédictif LR(K) [5].

En fonction de K symboles lus, on ne peut exécuter qu'une unique action qui est :

– décalage (shift) : lire 1 symbole dans le mot et l'empiler.

5. pour : Left (lecture de gauche à droite) Right (dérivation la plus à droite) et K symboles lus à la fois.

– réduire la règle n (réduce) : en sommet de pile il y a un manche qui est partie droite de n. On remplace le manche par la partie gauche.
– accepter (accept) : dernière réduction pour l'axiome.

Contre exemple avec la grammaire non LR(K) suivante $\begin{cases} \text{S'} & \to \text{S\$}^K \\ \text{S} & \to \text{Ac}|\text{Bd} \\ \text{A} & \to \text{Aa}|\lambda \\ \text{B} & \to \text{Ba}|\lambda \end{cases}$

S $\$^K \Rightarrow$ Ac$\$^K \overset{*}{\Rightarrow}$ Aaaaa...ac$\$^K \Rightarrow \lambda$aaa...ac$\$^K$
S $\$^K \Rightarrow$ Bd$\$^K \overset{*}{\Rightarrow}$ Baaaa...ad$\$^K \Rightarrow \lambda$aaa...ad$\$\$^K$

Le chois de réduire λ en A ou en B dépend de la fin du mot. Or on peut toujours trouver un mot dont le nombre de a est strictement supérieur à K.

5.2 Définition des grammaires LR(K) en terme de réductions gauches / dérivations droite

Soit un grammaire G = (Σ, N, P, S) et un mot w $\in \Sigma^*$.
Intuitivement, G sera LR(K) si à chaque étape i :
S $\Rightarrow \ldots \Rightarrow \alpha_{i-1} \Rightarrow \alpha_i \Rightarrow \ldots \Rightarrow \alpha_{m-1} \Rightarrow \alpha_m =$ w
on peut isoler un manche en parcourant α_i de gauche à droite en ne dépassant pas K symboles.

On suppose que l'on réduit A $\to \beta$ avec w $\in \Sigma^*$, α,$\beta \in (\Sigma \cup \text{N})^*$. La réduction $\alpha_{i-1} \Rightarrow \alpha_i$ est donc αAw $\Rightarrow \alpha\beta$w où |w| $\leq$ K (β est le manche).

5.2.1 Définitions

La grammaire G = (Σ, N, P, S) est LR(K) ssi
Si $\forall$ A $\to \beta$, B $\to \delta \in$ P, α,β,γ,$\delta \in (\Sigma \cup \text{N})^*$, w,x,y $\in \Sigma^*$:

1. S' $\overset{*}{\Rightarrow} \alpha$Aw $\Rightarrow \alpha\beta$w
2. S' $\overset{*}{\Rightarrow} \alpha$Aw $\Rightarrow \alpha\beta$y $= \gamma\delta$x
3. first_K(w) = first_K(y)

Alors $\alpha = \gamma$, $\beta = \delta$, x = y, A = B (On fait la même réduction au même endroit).

Exemple : Avec les mêmes symboles, on fait la même chose au même endroit.

1. S' $\overset{*}{\Rightarrow}$ T*F+i$\$ \Rightarrow$ T*i+i$\$ \Rightarrow$ F*i+i$\$ \Rightarrow$ i*i+i$\$
2. S' $\overset{*}{\Rightarrow}$ T*F+i*i$\$ \Rightarrow$ T*i+i*i$\$ \Rightarrow$ F*i+i*i$\$ \Rightarrow$ i*i+i*i$\$

Contre exemple

1. S' $\overset{*}{\Rightarrow}$ Aa^{K}c$\$^K \Rightarrow \lambda\lambda$a^{K}c$\$^K$ (α=λ, β=λ, w=a^{K}c$\$^K$)
2. S' $\overset{*}{\Rightarrow}$ Ba^{K}d$\$^K \Rightarrow \lambda\lambda$a^{K}d$\$^K$ (α=λ, β=λ, y=a^{K}d$\$^K$)

On a bien first_K(w) = first_K(y) = a^K. Si la grammaire était LR(K), on aurait réduit β=λ de la même façon dans les deux cas (en A ou en B, mais pas les deux).

5.2.2 Propriétés des grammaires LR(K)

Les langages à grammaires LR(K) sont la classe la plus large à analyser en un passage de façon déterministe.

Une grammaire LR(K) est non ambigüe.

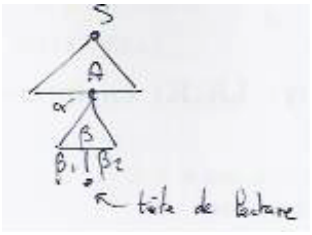
Pour un K fixé, on sait déterminer si la grammaire est LR(K).

Si G est LR(K), $\exists$ G' qui est LR(1) tel que L(G') = L(G).

5.3 Définition constructive des grammaires LR(K) (ensembles d'items)

5.3.1 Préfixe viable

Lors de la construction de l'arbre syntaxique, on n'a pas encore totalement parcouru β pour réduire la règle A$\rightarrow \beta$: β_1 a été parcouru et la pile contient $(\alpha\beta_1)$. $\gamma = \alpha\beta_1$ est dit préfixe viable.



Définition :
$\gamma \in (\Sigma \cup N)^*$ est un préfixe viable de G ssi γ est préfixe d'un mot dans une réduction la plus à gauche.

i.e. S' $\overset{*}{\Rightarrow} \alpha$Aw $\Rightarrow \alpha\beta$w, $\gamma = \alpha\beta_1$ est préfixe de $\alpha\beta$ si $\beta = \beta_1 \cdot \beta_2$.

5.3.2 Items LR(K)

Un item est un "photographie" d'une situation d'analyse. On peut être dans une situation de réduction ou une situation de décalage.

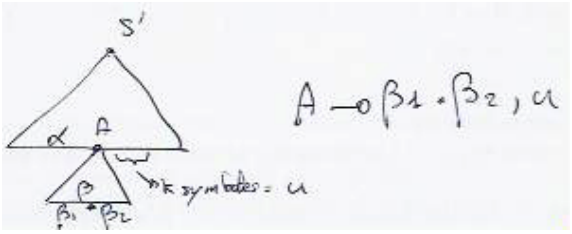


FIGURE 6 – Situation de décalage

Lors d'une situation de décalage, il faut faire progresser le point (la tête de lecture) pour arriver au bout d'un manche.

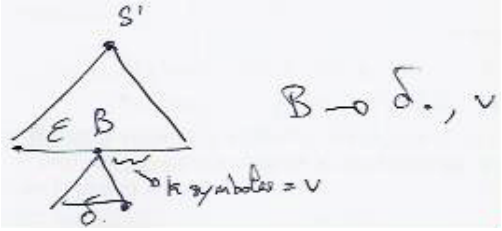


FIGURE 7 – Situation de réduction

Lors d'une situation de réduction, il on est arrivé au bout d'un manche. Il faut le réduire en le non-terminal correspondant.

Définition :
Un item LR(K) est A→ $\beta_1 \cdot \beta_2$,u où A→ $\beta_1 \cdot \beta_2$ est le cœur et u le contexte.

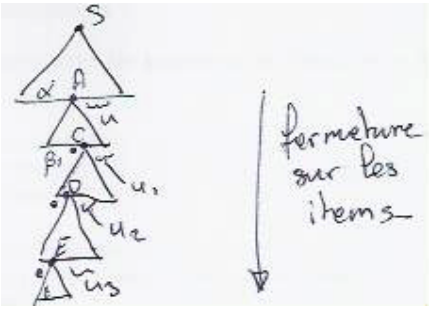
Cet item est valide (utile) pour le préfixe viable $\gamma = \alpha\beta_1$ de G ssi, pour A ∈ N, u,w ∈ Σ^* avec |u|=K et α,β,β_1,β_2 ∈ $(\Sigma \cup N)^*$
∃ une règle A→ $\beta_1 \cdot \beta_2 = \beta$ et S' $\stackrel{*}{\Rightarrow}$ αAw ⇒ $\alpha\beta_1\beta_2$w avec u = first_K(w).

Exemple
S' ⇒ E$ ⇒ T$ ⇒ T*F$ ⇒ T*(E)$ ⇒ T*(E+T)$ ⇒ T*(E+F)$ ⇒ T*(E+i)$ ⇒ T*(T+i)$ ⇒ T*(F+i)$ ⇒ T*(i+i)$ ⇒ F*(i+i)$ ⇒ i*(i+i)$

Les items suivant sont valides :
- E→E+.T,) pour γ = T*(E+ (ici β_1 = E+ et β_2 = T)
- F→.i,* pour $\gamma = \lambda$ (ici $\beta_1 = \lambda$ et β_2 = i)

Propriété de consistance des items :

Il peut arriver que $\alpha\beta_1 = \epsilon\delta$ (*cf* Schémas 6 et 7 pour la représentation des symboles). On s'attend donc à trouver K symboles différents pour exécuter deux actions différentes. Il faut donc que v ∉ $\text{first}_K(\beta_2 u)$.



Définition de la consistance sur deux items : B→ δ.,v et A→ β_1.tβ_2,u avec t ∈ Σ sont consistantes si v ∉ first_K(tβ_2u)

6 Construction des ensembles d'items pour une grammaire G

Le principe de cette opération est de :
- prendre un préfixe viable de G : $\gamma = X_1X_2 \ldots X_n$
- construire l'ensemble $V_K(\gamma)$ des items valides pour γ.

Pour ce faire, on va procéder par approximations successives en construisant $V_K(\lambda)$ puis $V_K(X_1)$ puis ... puis $V_K(X_1X_2 \ldots X_n) = V_K(\gamma) = (A)$. On appelle GOTO((A), X) l'ensemble $V_K(\gamma X)$.

6.1 Algorithme de construction des ensembles d'items LR(K)

Argument : une grammaire augmentée G'
Sortie : C = collection canonique d'items

function : fermeture(I : ensemble d'items)
début
répéter
pour chaque item A $\to \alpha$.Bφ,u $\in$ I faire
pour chaque production B $\to \delta$ de G' faire
pour chaque v $\in$ first$_K(\varphi$u) tel que B $\to$.δ,v $\notin$ I faire
I $\leftarrow$ I $\cup$ {B $\to$.δ,v } ;
finpour
finpour
finpour
jusqu'à stabilité
retourner I
fin

function : Goto(I : ensemble d'items, X $\in (\Sigma \cup$ N))
début
Soit J = l'ensemble des items A $\to \alpha$X.φ,u tels que A $\to \alpha$.Xφ,u $\in$ I
retourner fermeture(J)
fin

procedure : Items(G' : grammaire)
début
C = fermeture (S' $\to$ .S,\$^{K}) ;
répéter
pour chaque ensemble I de C, et chaque X $\in (\Sigma \cup$ N) tels que Goto(I,X) $\neq \emptyset$ et Goto(I,X) $\notin$ C faire
ajouter Goto(I,X) à C
finpour
jusqu'à stabilité
retourner C
fin

Théorème :
G' est LR(K) ssi tous les ensembles d'items canoniques sont consistants.

6.2 Exemple d'analyseurs LR(K)

K = 0 : Soit la grammaire G'$_1$:
$$\begin{cases} {}_{(0)}\ \text{S'} & \to \text{S\$}^0 \\ {}_{(1)}\ \text{S} & \to \text{aAc} \\ {}_{(2)}\ \text{A} & \to \text{Abb} \\ {}_{(3)}\ \text{A} & \to \text{b} \end{cases}$$
C = { I$_0$ = { S' $\to$ .S,λ ; S $\to$ .aAc,λ } ; shift
I$_1$ = Goto(I$_0$,S) = V$_0$(S) = { S' $\to$ S.,λ } ; accept
I$_2$ = Goto(I$_0$,a) = V$_0$(a) = { S $\to$ a.Ac,λ ; A $\to$ .Abb,λ ; A $\to$ .b,λ } ; shift

$I_3 = \text{GOTO}(I_2,A) = V_0(aA) = \{ S \to aA.c,\lambda\ ;\ A \to A.bb,\lambda \}$; shift/shift
$I_4 = \text{GOTO}(I_2,b) = V_0(ab) = \{ A \to b.,\lambda \}$; reduce (3)
$I_5 = \text{GOTO}(I_3,c) = V_0(aAc) = \{ S \to aAc.,\lambda \}$; reduce (1)
$I_6 = \text{GOTO}(I_3,b) = V_0(aAb) = \{ A \to Ab.b,\lambda \}$; shift
$I_7 = \text{GOTO}(I_6,b) = V_0(aAbb) = \{ A \to Abb.,\lambda \}\ \}$ reduce (2)

Tous les ensembles d'items (les I_i) sont consistants $\Rightarrow$ G'$_1$ est LR(0).

Reconnaissance de abbbc :

Les états sont de la forme (w, P, r) où w est le ruban, P est l'état de la pile où chaque symbole est suivi de l'état dans lequel il est apparu, et r est la trace des règles utilisées pour les réductions.

(abbbc, \$0, λ) $\vdash$ (bbbc, \$0a2, λ) $_{[\text{GOTO}(I_0,a) = I_2]}$ $\vdash$ (bbc, \$0a2b4, λ) $_{[\text{GOTO}(I_2,b) = I_4]}$ $\vdash$ (bbc, \$0a2A3, 3) $_{[\text{GOTO}(I_2,A) = I_3]}$ $\vdash$ (bc, \$0a2A3b6, 3) $\vdash$ (c, \$0a2A3b6b7, 3) $\vdash$ (c, \$0a2A3, 32) $\vdash$ (λ, \$0a2A3c5, 32) $\vdash$ (λ, \$0S1, 321) $\vdash$ Accepter.

K = 1 : Soit la grammaire G'$_2$:
$$\begin{cases} {}_{(0)}\ S' & \to S\$^1 \\ {}_{(1)}\ S & \to Sc \\ {}_{(2)}\ S & \to T \\ {}_{(3)}\ T & \to aTb \\ {}_{(4)}\ T & \to \lambda \end{cases}$$

$I_0 = V_1(\lambda) = \{ S' \to .S,\$\ ;\ S \to .Sc,\$\ ;\ S \to .T,\$\ ;\ S \to .Sc,c\ ;\ S \to .T,c\ ;\ T \to .aTb,\$\ ;\ T \to .,\$\ ;\ T \to .aTb,c\ ;\ T \to .,c \}$

Notation pour simplifier l'écriture :

$I_0 = \{ S' \to .S,\$\ ;\ S \to .Sc,\$/c\ ;\ S \to .T,\$/c\ ;\ T \to .aTb,\$/c\ ;\ T \to .,\$/c \}$ shift/reduce (4)

$\mathtt{C} = \{ I_0\ ;\ I_1 = \text{GOTO}(I_0,S) = V_1(S) = \{ S' \to S.,\$\ ;\ S \to S.c,\$/c \}$; accept/shift
$I_2 = \text{GOTO}(I_0,T) = V_1(T) = \{ S \to T.,\$/c \}$; reduce (2)
$I_3 = \text{GOTO}(I_0,a) = V_1(a) = \{ T \to a.Tb,\$/c\ ;\ T \to .aTb,b\ ;\ T \to .,b \}$; shift/reduce (4)
$I_4 = \text{GOTO}(I_1,c) = V_1(Sc) = \{ S \to Sc.,\$/c \}$; reduce (1)
$I_5 = \text{GOTO}(I_3,T) = V_1(aT) = \{ T \to aT.b,\$/c \}$; shift
$I_6 = \text{GOTO}(I_3,a) = V_1(aa) = \{ T \to a.Tb,b\ ;\ T \to .aTb,b\ ;\ T \to .,b \}$; shift/reduce (4)
$I_7 = \text{GOTO}(I_5,b) = V_1(aTb) = \{ T \to aTb.,\$/c \}\ \}$; reduce (3)
$I_8 = \text{GOTO}(I_6,T) = V_1(aaT) = \{ T \to aT.b,b \}$; shift
[$\text{GOTO}(I_6,a) = I_6 \Rightarrow$ On ne le met pas]
$I_9 = \text{GOTO}(I_8,b) = V_1(aaTb) = \{ T \to aTb.,b \}\ \}$ reduce (3)

Reconnaissance de abc :

(abc\$, \$0, λ) $\vdash$ (bc\$, \$0a3, λ) $\vdash$ (bc\$, \$0a3T5, 4) $\vdash$ (c\$, \$0a3T5b7, 4) $\vdash$ (c\$, \$0T2, 43) $\vdash$ (c\$, \$0S1, 432) $\vdash$ (\$, \$0S1c4, 432) $\vdash$ (\$, \$0S1, 4321) $\vdash$ Accepter.

7 Les optimisations des analyseurs LR(K)

Le principe est d'essayer de rassembler les items pour diminuer le nombre d'états.

7.1 Les grammaires LALR(K)

LALR(K) signifie Look-ahead Augmented LR(K).

Méthode

1. $\mathtt{C} = \{ \mathrm{I}_0 ; \mathrm{I}_1 ; \ldots ; \mathrm{I}_n \}$ les ensembles d'items de G.
   Si échec LR(K) $\Rightarrow$ échec LALR(K).
2. Pour chaque cœur présent dans les ensembles d'items LR(K), trouver tous les états ayant des items de même cœur et remplacer les contextes par l'union des contextes.

   Exemple

$$\mathrm{I}_m = \{ \mathrm{A} \to \beta_1.\beta_2,\mathrm{u}_1 ; \; \mathrm{B} \to \delta.,\mathrm{v}_1 \} \quad \mathrm{I}_r = \{ \mathrm{A} \to \beta_1.\beta_2,\mathrm{u}_2 ; \; \mathrm{B} \to \delta.,\mathrm{v}_2 \} \quad \mathrm{I}_s = \{ \mathrm{A} \to \beta_1.\beta_2,\mathrm{u}_3 ; \; \mathrm{B} \to \delta.,\mathrm{v}_3 \}$$

   $\Rightarrow \mathrm{I}_{mrs} = \{ \mathrm{A} \to \beta_1.\beta_2,\mathrm{u}_1/\mathrm{u}_2/\mathrm{u}_3 ; \mathrm{B} \to \delta.,\mathrm{v}_1/\mathrm{v}_2/\mathrm{v}_3 \}$
3. $\mathtt{C}' = \{ \mathrm{J}_0 ; \mathrm{J}_1 ; \ldots ; \mathrm{J}_m \}$
   Si tous les J_j sont consistant, alors G' est LALR(K). Sinon G' n'est pas LALR(K).

Exemple G'$_2$ est-elle LALR(1) ?
I_3 et I_6 donnent $\mathrm{I}_{36} = \{ \mathrm{T} \to \mathrm{a.Tb},\$/\mathrm{c}/\mathrm{b} ; \mathrm{T} \to .\mathrm{aTb},\mathrm{b} ; \mathrm{T} \to .,\mathrm{b} \}$
I_7 et I_9 donnent $\mathrm{I}_{79} = \{ \mathrm{T} \to \mathrm{aTb}.,\$/\mathrm{c}/\mathrm{b} \}$
I_5 et I_8 donnent $\mathrm{I}_{58} = \{ \mathrm{T} \to \mathrm{aT.b},\$/\mathrm{c}/\mathrm{b} \}$

7.2 Les grammaires SLR(K)

SLR(K) signifie Simple LR(K).

Méthode

1. $\mathtt{C} = \{ \mathrm{I}_0 ; \mathrm{I}_1 ; \ldots ; \mathrm{I}_n \}$ des items LR(0).
   Si tous les éléments sont consistants, alors G' est LR(0) : *Stop.*
2. S'il existe des conflits LR(0) alors construire $\mathtt{C}' = \{ \mathrm{J}_0 ; \mathrm{J}_1 ; \ldots ; \mathrm{J}_m \}$ sur la base $\mathrm{A} \to \beta_1.\beta_2,\mathrm{follow}_K(\mathrm{A})$ où :
   – $\mathrm{A} \to \beta_1.\beta_2,\lambda$ est un item LR(0) de $\mathtt{C}$.
   – $\mathrm{follow}_K(\mathrm{A})$ est considéré comme le contexte de l'item.
3. Si $\mathtt{C}'$ contient des ensembles d'items consistants, alors G' est SLR(K). Sinon G' n'est pas SLR(K).

8 Relations entre les classes de grammaires

Pour K fixé, les grammaires LALR(K) sont un sous-ensemble des grammaires LR(K) et les grammaires SLR(K) sont un sous-ensemble des grammaires LALR(K) (Le contexte étant maximal pour les grammaires SRL(K)).

Théorème :
Toute grammaire LL(K) est LR(K).

On ne peut, par contre, pas comparer les grammaires LALR(K) et SLR(K) avec les grammaires LL(K) : certaines grammaires SLR(K) et LALR(K) sont LL(K) mais d'autres non ; et certaines grammaires LL(K) sont LALR(K) ou SLR(K) et d'autres non.

Deuxième partie
Traduction des langages

On va se baser sur un traducteur particulier : le compilateur.

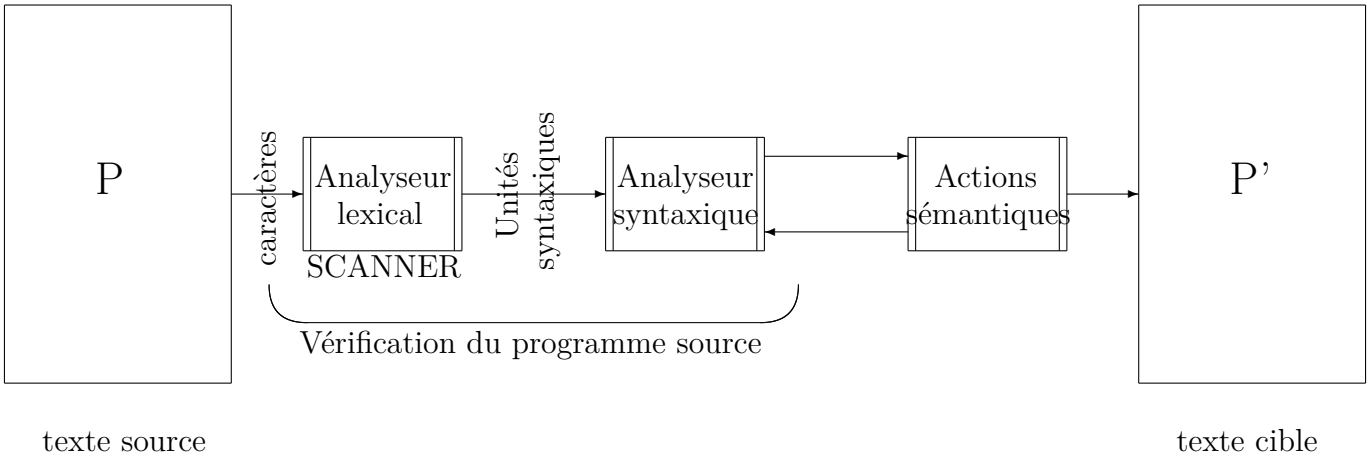


FIGURE 8 – Schématisation du processus de compilation

9 Table des symboles, code intermédiaire

9.1 La table des symboles, TDS

Que deviennent les symboles introduits par le programmeur ?
Que devient une expression telle que A := B+C ?

a) Interprétation
   - Les types des différents termes sont-ils compatibles ?
   - Quelles sont les valeurs de B et C (pour pouvoir faire le calcul) ?

b) Compilation
   - Les types des différents termes sont-ils compatibles ?
   - Quelles sont les adresses correspondant à A, B et C ?

Définition :
La TDS d'un traducteur mémorise toutes les informations nécessaires à la traduction. C'est une tables dont les clefs sont les symboles à retenir. Les opérations applicables sur la table sont :
- Créer une entrée
- Effacer une entrée
- Rechercher un objet

La composition d'une table des symboles n'est pas normalisée, elle dépend des besoins de chaque traducteur (de la façon dont il est implémenté).

```
class Personne{
    int date_de_naissance;
    char* prenom[15];
    struct adresse { int num;
                     string rue;
                     int code;
                     string ville; }
    void f(t1 p1; t2 p2){_}
}
```

FIGURE 9 – Déclaration Java

```
'personne'          (class)
'date_de_naissance' (entier)
'prenom'            (tableau, (1, 15), pt char)
'adresse'           (struct, (num, entier),
                             (rue, chaine),
                             (code, entier),
                             (ville, chaine))
'f'                 (function, (('p1', 't1'),
                               ('p2', 't2')),
                               point_entrée,
                               taille_local_data,
                               ...)
```

FIGURE 10 – TDS associée

9.2 Code intermédiaire

Les besoins de produire un code intermédiaire vient plus de l'économie que d'une réelle contrainte algorithmique.

On a donc besoin de deux traducteur : le premier pour traduire le code source vers le langage intermédiaire et le second pour traduire le code intermédiaire vers le langage cible.

Le code intermédiaire est :
- un langage assez peu riche
- indépendant d'une architecture
- facile à traduire pour une architecture donnée
- généralement inusité (exception du P-code provenant du PASCAL et du Byte-Code provenant du JAVA)

Exemple du langage des quadruplets

Il s'agit d'un langage pour machine fictive à 3 adresses.

`operateur, opérande1, opérande2, resultat`

Il n'y a pas de notion de registre. Chaque quadruplet a un numéro qui est son adresse où on peut se brancher. Tous les arguments sont des chaînes de caractères.

Les différentes opérations possibles sont :

1. Affectation : affecter op1 à res.
   :=, op1, nil, res
2. Arithmétique : utilise l'opération (+, -, *, /) sur a et b et stocker le résultat dans temp.
   opération, a, b, temp
3. Branchement inconditionnel : se rendre à une adresse donnée.
   goto, nil, nil, adresse
4. Branchement conditionnel : utilise le test ($=?$, $/=?$, $<?$, $<=?$, $>?$, $>=?$) sur a et b pour, s'il réussit, se rendre à une adresse donnée.
   test, a, b, adresse
5. Conversion : pour transformer un entier en chaîne de caractère on utilise la fonction str.
   str(45) renvoie '45'

Exemple de traduction d'une expression vers le langage des quadruplets.

```
t := x * t + w * k - z
```

(1) *, x, t, $temp_1$
(2) *, w, k, $temp_2$
(3) +, $temp_1$, $temp_2$, $temp_3$
(4) -, $temp_3$, z, $temp_4$
(5) :=, $temp_4$, nil, t

```
a or b and c
```

(1) = ?, a, 1, EttiquetteVrai
(2) = ?, b, 0, EttiquetteFaux
(3) = ?, c, 1, EttiquetteVrai
(4) goto, nil, nil, EttiquetteFaux

10 Traduction dirigée par la syntaxe

L'analyseur syntaxique, en association avec les actions sémantiques, va piloter le processus de traduction.

10.1 Traduction couplée avec un analyseur descendant

10.1.1 Introduction

Définition Une action sémantique est une exécution de code par le traducteur qui :
- engendre le code objet
- mémorise des informations
- modifie des informations

Ici, on va utiliser un analyseur descendant qui effectue une descente récursive LL(1).

10.1.2 Notations

SCAN est l'analyseur syntaxique.
NEXTS est la prochaine unité syntaxique.
SKIP(t : US) est une procedure qui consomme une unité syntaxique et place la suivante dans NEXTS.
GEN(s1, s2, s3, s4 : string) est une procédure qui construit un quadruplet du langage intermédiaire.
NEXTQUAD est le numéro du prochain quadruplet à générer.
NEWTEMP est une fonction qui créé un nouveau nom de variable temporaire.

10.1.3 Mise en œuvre

Problème : Traduire des expressions arithmétiques en quadruplets. $E \rightarrow T\{+T\}^*$

```
procedure E
début
  T
  while NEXTS = '+' do
    SCAN
    T
  endwhile
fin
```

Avec cette méthode il n'est pas possible de récupérer la valeur finale du calcul. Il va donc falloir paramétrer les différents appels à T pour leur fournir les valeurs déjà calculées ainsi que l'appel à E pour pouvoir récupérer la valeur finale.

L'action sémantique a donc pour support l'analyseur syntaxique LL(1) auquel va être adjoint des paramètres et des variables locales.

Exemple

$$\begin{cases} E & \rightarrow T\{+T\}^* \\ T & \rightarrow F\{*F\}^* \\ F & \rightarrow i|(E) \end{cases}$$

```
procedure E(r : string)
 u, t : string
début
  T(r)
  while NEXTS = '+' do
    SKIP(+);
    T(t);
    u := NEWTEMP;
    GEN(+, r, t, u);
    r := u;
  endwhile
fin

procedure T(r : string)
 u, t : string
début
  F(r)
  while NEXTS = '*' do
    SKIP(*);
    F(t);
    u := NEWTEMP;
    GEN(*, r, t, u);
    r := u;
  endwhile
fin
```

```
procedure F(r : string)
début
⌈ T(r)
| if NEXTS = i then r := i.string ; SCAN ;
⌊ else SKIP('(') ; E(r) ; SKIP(')') ;
fin
```

10.1.4 Quelques principes méthodologiques

<instr_cond> → Si <exp> alors <inst$_1$ > sinon <inst$_2$ >

a) Faire un schéma équivalent en quadruplets
   Quelques quadruplets pour <exp>
   TEST : = ?, <exp>.result, 0, PF
   Quelques quadruplets pour <inst$_1$ >
   goto, nil, nil, SUITE
   PF : ...
   Quelques quadruplets pour <inst$_2$ >
   SUITE : ...

b) Inventaire des problèmes
   – d'action sémantique
   – vérifier type <exp> → CHECKTYPE
   – référencement en avant (pour PF et SUITE)
     (a) mémoriser l'adresse du quadruplet qui pointe vers une adresse non encore créée
     (b) générer ce quadruplet incomplet
     (c) mettre à jour le champ incomplet quand l'information est disponible
         ⇒ BACKPATCH

   On a donc besoin des procédures de travail CHECKTYPE(s : string, t : type) et BACKPATCH(L : liste de quad, q : integer)

c) Écriture de la procédure d'analyse avec les actions sémantiques

```
procedure INSTCOND
 TEST, AD : integer, r : string
début
⌈SKIP('si');
|EXP(r);
|CHECKTYPE(r, booleen);
|SKIP('alors');
|TEST := NEXTQUAD;
|GEN(= ?, r, 0, nil);
|INST;
|AD := NEXTQUAD;
|GEN(goto, nil, nil, nil);
|BACKPATCH({TEST}, NEXTQUAD);
|SKIP('sinon');
|INST;
⌊BACKPATCH({AD}, NEXTQUAD);
fin
```

<boucle> → tantque <exp> faire <inst>

a) Faire un schéma équivalent en quadruplets
   DEBUT : ...
   Quelques quadruplets pour <exp>
   TEST : = ?, <exp>.result, 0, SUITE
   Quelques quadruplets pour $<\text{inst}_1>$
   goto, nil, nil, DEBUT
   SUITE : ...

b) Inventaire des problèmes
   - d'action sémantique
   - vérifier type <exp> → CHECKTYPE
   - référencement en avant (pour SUITE)
   - référencement en arrière (pour DEBUT)

c) Écriture de la procédure d'analyse avec les actions sémantiques

```
procedure BOUCLE
 DEBUT, TEST : integer, r : string
début
⌈SKIP('tantque');
|DEBUT := NEXTQUAD;
|EXP(r);
|CHECKTYPE(r, booleen);
|SKIP('faire');
|TEST := NEXTQUAD;
|GEN(=?, r, 0, nil);
|INST;
|GEN(goto, nil, nil, str(DEBUT));
⌊BACKPATCH({TEST}, NEXTQUAD);
fin
```

Troisième partie
Lambda-calcul

11 ??