Traduction des Langages

Cours de Christine Maurel
Christine.Maurel@irit.fr

Mattijs Korpershoek
mattijs.korpershoek@univ-tlse3.fr
M1 Informatique, Université Paul Sabatier
Année 2012 - 2013

Ceci est un cours pris par moi-même.
Il a été relu mais peut quand même comporter des inexactitutes ou des erreurs.
En aucune façon, ce document ne prétend remplacer "le plaisir d'aller écouter le cours raconté par la dame même à 7h45 le vendredi"

Table des matières

Chapitre 1

Introduction

Le but est de nous faire comprendre comment fonctionne la traduction des langages haut niveau en langage machine. (ie compris par la machine, en binaire)

Combler le fossé sémantique. logiciels :

interprété P → interprete → résultat du calcul

compilé P → compilateur → Programme P'

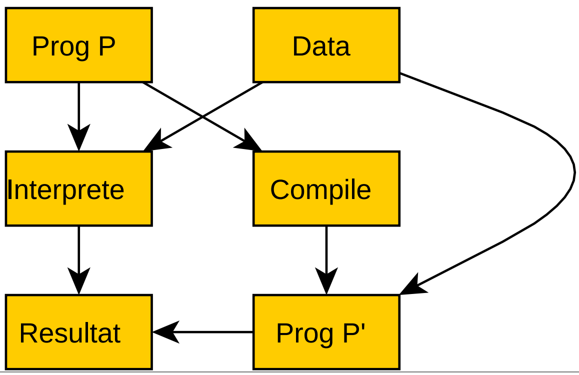
Interpreter, c'est Lire ; → Evaluer ; → Imprimer ;

1.1 Définitions

– Un interprete c'est *un programme qui simule la machine cible*
– Un compilateur c'est un *transformateur/traducteur* de programme

Note : il est important d'avoir *équivalence* entre P et P' , ça nécéssite de certifier la traduction. (voir 1.1)

FIGURE 1.1– Différences entre compilation et interpretation

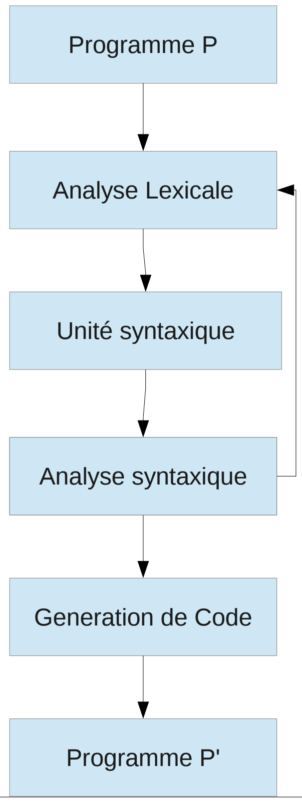


Interpreté Convivial, rapidité de la mise au point. Inconv : efficacité.
Compilé Efficacité, Inconv : lourd, rigide

1.2 Phases de la compilation

voir 1.2

FIGURE 1.2– Schéma compilation



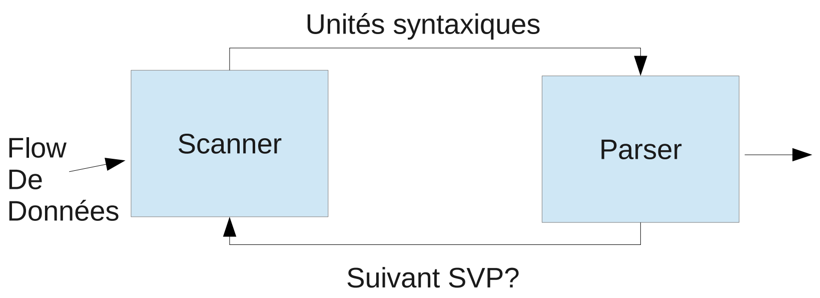
Analyse lexicale
Unité syntaxique détection d'erreurs syntaxiques
Analyse syntaxique
Génération de code

Chapitre 2

Analyse Lexicale

La première phase de la chaine de compilation. Il s'agit de passer de la *syntaxe concrete* à la *syntaxe abstraite*. C'est pour éliminer le "sucre syntaxique"

FIGURE 2.1– analyse lexicale



Scanner l'outil qui réalise l'analyse lexicale

Parser l'outil qui réalise l'analyse syntaxique

Exemple :

Listing 2.1– Exemple de code analysé

```
begin /* Retour a la ligne */
  X := 4;
end
```

Mots cléfs begin end if then else

Identificateur de variable X

Operateur :=

Constante numérique 4

Séparateurs ; retour à la ligne

Lexème

- Les identificateurs doivent commencer par une *lettre* suivie d'une suite eventuellement vide de lettres, chiffres et éventuellement de caractères spéciaux

2.0.1 Expressions régulières

$$l \in \{a, ..., z, A, ..., Z\}$$
$$c \in \{0, ..., 9\}$$

$l(l + c)^*$ $\rightarrow$ Théorème d'Arden $\rightarrow$ automate fini *deterministe.*

2.0.2 Constantes numériques

Suite *non vide* de chiffres. Rappel :

$$cc^* \equiv c^+$$

Automate fini déterministe.

2.0.3 Scanner

C'est un automate fini déterministe construit de façon rigoureuse avec le théorème d'Arden pour reconnaitre les différentes unités syntaxiques du langage compilé.

Il faut parfois avoir des informations associées aux unités syntaxiques, celles ci sont nommées les *attributs sémantiques.* Par exemple, la valeur décimale d'une constante numérique, la chaine de caractères pour un identificateur.

Pour calculer ces attributs, on décore les transitons de l'automate fini avec des *actions sémantiques.*

2.1 Exemples et rappels de L3

Exemple de construction de scanner pour reconnaitre les *identificateurs* et les *constantes entières.*

Alphabet :

$$
\begin{aligned}
X &= L \cup C \\
L &= \{a, b, ..., z\} \\
C &= \{0, 1, ..., 9\} \\
\text{mot } \omega &\in X^* \\
\text{mot vide} &= \lambda
\end{aligned}
$$

Langage Ensemble de mots compris dans X^*

Expressions régulières caracterisent un langage régulier

Expression régulière c'est λ ou voir ci dessous :

Pour L_0 :

$$L_0 = l.(l+c)^* + c.c^* + (+).c.c^* + (-).c.c^*$$
$$L_0 = l.L_1 + c.L_2 + (+).L_3 + (-).L_3$$

Pour L_1 :

$$L_1 = (l+c)^*.\lambda$$
$$\Rightarrow^{Arden}$$
$$L_1 = (l+c).L_1 + \lambda$$

Pour L_2 :

$$L_2 = c^* \equiv c^*\lambda$$
$$\Rightarrow^{Arden}$$
$$L_2 = c.L_2 + \lambda$$

Pour L_3 :

$$L_3 = cc^*$$
$$L_3 = c.L_2$$

2.2 Décorer l'automate fini avec des actions sémantiques

Pour calculer :
- La valeur demandée du nombre lu en entrée.
- La chaine de caractères constituant l'identificateur reconnu

val fonction qui donne la valeur décimale correspondant au caractère lu.

carcour variable qui contient le caractère courant lu en entrée.

valeur variable qui doit contenir la valeur décimale de l'entrée lue.

$$A_1 = \{valeur := val(carcour);\}$$
$$A_2 = \{valeur := valeur * 10 + val(carcour);\}$$
$$A_3 = \{\text{si } carcour = - \text{ alors } \{signe := -1\} \text{ sinon } \{signe := 1\}\}$$
$$A_4 = \{valeur := valeur * signe;\}$$

2.3 Rappels

Il faut découper le texte en lexèmes(token, unités syntaxiques) pour que le scanner donne ça au *parser* afin de voir si ça forme une phrase correcte.

Un lexème ça peut être :

- identificateur
- mots cléfs réservés
- constantes numériques
- opérations

Il ne doit pas tenir compte des blancs, des retours à la ligne, des commentaires.

Le scanner c'est un *automate fini* construit rigoureusement à partir d'expressions régulières avec Arden. Il faut ensuite ajouter les *actions sémantiques* sur les transitions pour calculer les attributs.

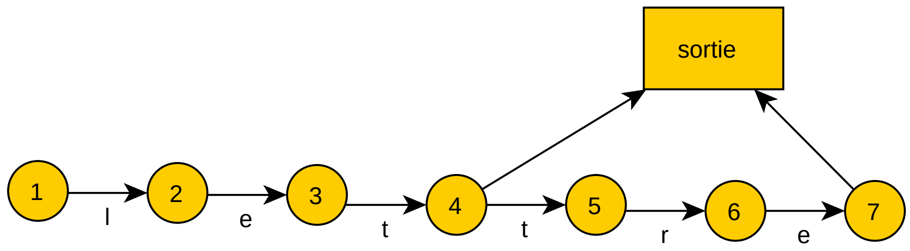
Parfois on a besoin d'informations supplémentaires ; pour les identificateurs c'est la chaine de caractères qui correspond au nom de variable, pour les constantes numériques c'est leur valeur.

2.4 Quelques problèmes à régler

2.4.1 Problèmes de reconnaissance

Si on a "let" en mot clé et on a un identificateur "lettre". On donne la *priorité* à l'unité syntaxique la plus longue. voir 2.2

FIGURE 2.2– Problème de reconnaissance



2.4.2 Table des Symboles (TDS)

Une Unité Syntaxique(US) est identifiée par un code et éventuellement "des infos". La table des symboles sert à ranger ces infos. L'entrée dans la TDS est calculée par une fonction de hashcode.

Exemple de hascode :
$f('abc') = (code(a) + code(b) + code(c)) \mod taille('abc') = @1$

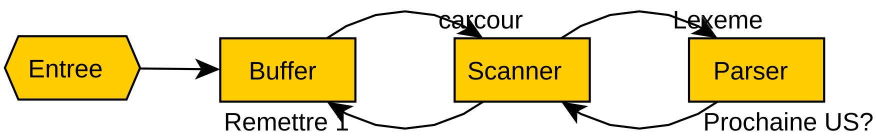
Mais cette fonction de calcul de hash peut entrainer des *collisions* :
$f('abc') = f('bac')$

2.4.3 Problème du recul

Si on a $<$, on sait que l'unité syntaxique c'est $<$. Mais si on tombe sur $<=$ on a une autre unité syntaxique ; $\leq$.

C'est pas un problème car on a vu qu'on privilegie l'US la plus longue. Si on tombe sur <1 , il faut remettre 1 dans le flot d'entrée.

FIGURE 2.3– Problème du recul



2.5 La notion de surlangage

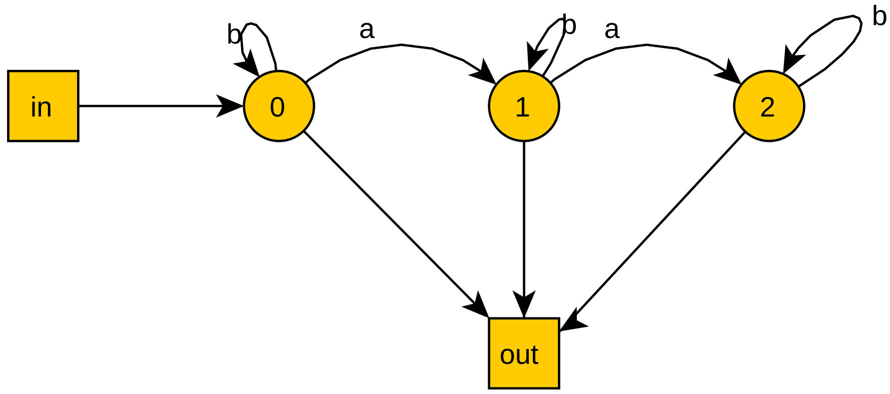
Parfois, il faut trouver un compromis entre automate et actions sémantiques

$$
\begin{aligned}
X &= \{a, b\} \\
L &= \{w \in X^*/|w|_a \leq 2\} \\
&= \{w \in X^*/|w|_a = 0\} \cup \{w \in X^*/|w|_a = 1\} \cup \{w \in X^*/|w|_a = 2\} \\
&= b^* + b^*ab^* + b^*ab^*ab^*
\end{aligned}
$$

Rappel : + est l'operateur d'union sur les expressions régulières

$$
\begin{aligned}
&\text{Pour } L_0 : \\
&\qquad L_0 = b^* + b^*ab^* + b^*ab^*ab^* \\
&\qquad = b^*(\lambda + ab^* + ab^*ab^*) \\
&\qquad = r1^*r2 \\
&\Rightarrow^{Arden} L_0 = bL_0 + \lambda + ab^* + ab^*ab^* \\
&\qquad = bL_0 + aL_1 + \lambda \\
&\text{Pour } L_1 : \\
&\qquad L_1 = b^* + b^*ab^* \\
&\qquad = b^*(\lambda + ab^*) \\
&\qquad = r1^*(\lambda + ab^*) \\
&\Rightarrow^{Arden} L_1 = bL_1 + \lambda + ab^* \\
&\qquad L_1 = bL_1 + aL_2 + \lambda \\
&\text{Pour } L_2 : \\
&\qquad L_2 = b^*.\lambda \\
&\qquad L_2 = r1^*.r2 \\
&\Rightarrow^{Arden} L_2 = bL_2 + \lambda
\end{aligned}
$$

FIGURE 2.4– Automate résultat



Rappel théoreme d'Arden :

$$\begin{aligned} & X = r1.X + r2 \\ \Leftrightarrow\ & X = r1^*r2 \\ & \text{Si } \lambda \notin r1 \text{ la solution est unique.} \end{aligned}$$

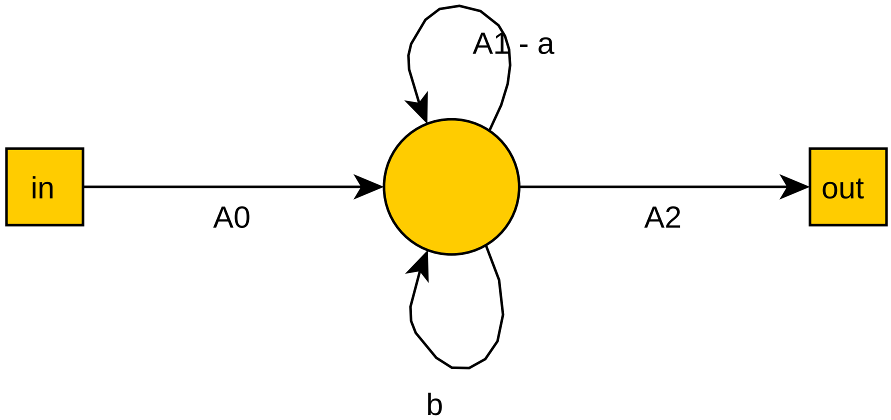
Imaginons maintenant qu'on veuille reconnaitre ce langage là :

$$L = \{w \in X^* / |w|_a \leq 2012\}$$

On peut le reconnaitre grace a un surlangage L' tel que $L \subseteq L'$ puis on contraint à L par des actions sémantiques.

$$\begin{aligned} L' &= X^* = (a+b)^* \\ L' &= (a+b)^*.\lambda \\ \Rightarrow^{Arden}\ L' &= (a+b)L' + \lambda \\ &= aL' + bL' + \lambda \end{aligned}$$

Figure 2.5– Automate avec surlangage



Description des actions :

Listing 2.2– Action semantique AO

```
NB_a := 0; //on initialise le nombre de 'a' a  0
```

Listing 2.3– Action semantique A1

```
NB_a++; //on a trouve un 'a'
```

Listing 2.4– Action semantique A2

```
if (NB_a <= 2012) then
  return (NB_a);
else
  ERROR ("Trop de a");
```

2.5.1 Pour automatiser les scanners

Il existe des outils pour générer automatiquement des scanners.

- lex
- flex
- ocamllex

On donne l'expression regulière et les actions sémantiques écrites en C ou en Ocaml

2.6 Rappels sur les grammaires

Un langage, plusieurs grammaires ? Voici quelques exemples :

$$
\begin{aligned}
&G_1 : \\
&\quad S \rightarrow aAc \\
&\quad A \rightarrow Abb \\
&\quad A \rightarrow b \\
&G_2 : \\
&\quad S \rightarrow aAc \\
&\quad A \rightarrow bAb \\
&\quad A \rightarrow b \\
&G_3 : \\
&\quad S \rightarrow aAc \\
&\quad A \rightarrow bbA \\
&\quad A \rightarrow b
\end{aligned}
$$

Par intuition, on peut trouver les 3 expressions régulières :
- $L(G_1) = ab(bb)^*c$
- $L(G_2) = a(b^n bb^n)c$ avec $n \geq 0$
- $L(G_3) = a(bb)^*bc$

On peut en conclure que $L(G_1) = L(G_2) = L(G_3) = ab^{2n+1}c$
Formellement on sait résoudre des équations de langages.
- Passer de la grammaire à un système d'équations de langage.

$$A \rightarrow \alpha_1|\alpha_2|...\alpha_n \equiv A = \alpha_1 + \alpha_2 + ... + \alpha_n$$

– Résoudre un système d'équations de langage :

$$\begin{aligned}
&\text{Pour } G_1 : \\
&\quad G_1 \equiv S = aAc \\
&\qquad\quad A = Abb + b \\
\Rightarrow^{Ardenbis}\ &\qquad\quad A = b(bb)^* \\
&\qquad\quad S = ab(bb)^*c \\
&\text{Pour } G_2 : \\
&\quad G_2 \equiv S = aAc \\
&\qquad\quad A = bAb + b \\
\Rightarrow^{A^nB^n}\ &\qquad\quad A = b^n b b^n (A^nB^n) \\
&\qquad\quad S = ab^n b b^n c \\
&\text{Pour } G_3 : \\
&\quad G_3 \equiv S = aAc \\
&\qquad\quad A = bbA + b \\
\Rightarrow^{Arden}\ &\qquad\quad A = (bb)^*b \\
&\qquad\quad S = a(bb)^*bc
\end{aligned}$$

Rappels Arden :

$$X = r_1X + r_2 \Leftrightarrow X = {r_1}^* r_2$$

Rappels ArdenBis :

$$X = Xr_1 + r_2 \Leftrightarrow X = r_2{r_1}^*$$

Rappels A^nB^n :

$$X = YXZ + U \Leftrightarrow X = Y^nUZ^n, n \geq 0$$

2.7 Approximation successive

Mais quelques fois, pour trouver le langage engendré, on doit procéder par approximations successives. Prenons par exemple :

$$\begin{aligned}
&S \to aSR|RS|cR \\
&R \to SaR|b
\end{aligned}$$

$L(G) = S =?$ Il faut trouver *le plus petit point fixe* qui est solution du système d'équations.

On a une équation de langage $X = f(X)$, il faut trouver X qui soit solution. On part de $\perp$ ie $\perp \equiv \emptyset$. Ensuite, on applique par itération la fonction f.

$$\begin{aligned} f^1(\emptyset) &= E_1 \\ f^2(\emptyset) &= f(E_1) = E_2 \\ f^{n+1}(\emptyset) &= f(f^n(\emptyset)) = f(E_n) = E_{n+1} \end{aligned}$$

Exemple pour $A = bAb + b$:

$$\begin{aligned} &i = 0 : \emptyset \\ &i = 1 : f(\emptyset) = b.\{\emptyset\}.b + b = \{b\} = E_1 \\ &i = 2 : f^2(\emptyset) = f(E_1) = f(b) = bbb + b = b^3 + b = E_2 \\ &i = 3 : f^3(\emptyset) = f(f^2(\emptyset) = f(b^3 + b) = b(b^3 + b)b + b = b^5 + b^3 + b = E_3 \end{aligned}$$

Appliqué à l'exemple ci dessus, ça donne :
$< S, R >=< \emptyset, \emptyset >$
Ce qui implique que $f(< S, R >) =< aSR + RS + cR, SaR + b >$
Ca fait donc :

$$\begin{aligned} &i = 0 :< \emptyset, \emptyset > \\ &i = 1 : f^1 < \emptyset, \emptyset >=< \emptyset, b > \\ &i = 2 : f^2 < \emptyset, \emptyset >= f < \emptyset, b >=< cb, b > \\ &i = 3 : f^3 < \emptyset, \emptyset >= f < cb, b >=< acbb + bcb + cb, cbab + b > \end{aligned}$$

Rappel :

$$\forall L \subseteq X^*, L.\emptyset = \emptyset$$

Chapitre 3

Notion d'analyse syntaxique

Il s'agit de déterminer si un *mot* $\in L(G)$ et si oui, avec quelles règles de grammaire il est construit.

Soit G une grammaire, il existe un automate à pile qui *simule* G.
Etant donnée une grammaire $G =< N, X, P, S >$, on sait construire un *automate à pile* de façon automatique qui reconnaît $L(G)$, avec *un seul état*, souvent *non déterministe*.

$$G =< N, X, P, S >$$
$$P = A \to \beta, A \in N, \beta \in (N \cup X)^*$$

Automate à pile :

- q
- X : alphabet d'entrée
- $N \cup X$: alphabet de pile
- q : état initial
- S : fond de pile ; axiome
- $\emptyset$: ensemble d'états finaux
- δ : fonction de transition :
  - $\delta(q, \lambda, A) = (q, B) \forall A \to B \in P$
  - $\delta(q, x, x) = (q, \lambda) \forall x \in X$

Exemple :
Soit G_2 tel que :

$$S \to aAc \tag{3.1}$$
$$A \to bbA \tag{3.2}$$
$$A \to b \tag{3.3}$$
$$N = S, A \tag{3.4}$$
$$X = a, b, c \tag{3.5}$$

Avec δ tel que :

$$\delta(q, \lambda, S) = (q, aAc) \tag{3.6}$$
$$\delta(q, \lambda, A) = (q, bbA) \tag{3.7}$$
$$\delta(q, \lambda, A) = (q, b) \tag{3.8}$$
$$\delta(q, a, a) = (q, \lambda) \tag{3.9}$$
$$\delta(q, b, b) = (q, \lambda) \tag{3.10}$$
$$\delta(q, c, c) = (q, \lambda) \tag{3.11}$$

Analyse de la séquence *abbbbbc* :

ruban	pile	regle
λabbbbbc	S	3.6
abbbbbc	aAc	3.9
λbbbbbc	Ac	3.7
λbbbbbc	bbAc	3.10
bbbbc	bAc	3.10
bbbc	Ac	3.7
bbbc	bbAc	3.10
bbc	bAc	3.10
λbc	Ac	3.8
bc	bc	3.10
c	c	3.11
mot lu	*pile vide*	

Donc le mot est reconnu.

L'analyse doit se faire en *une seule lecture* du ruban et de façon *déterministe*. Pour cela, on regarde k symboles sur le ruban pour pouvoir décider de *façon unique* de la règle à appliquer. Cette analyse efficace est appellée *analyse k-prédictive*

Rappel G_2 :

$$S \rightarrow aAc$$
$$A \rightarrow bbA$$
$$A \rightarrow b$$

On peut écrire un algo déterministe pour la reconnaissance de $\omega \in L(G_2)$

Listing 3.1– Algo reconnaissance de ω

```
Utiliser 3.6;
Utiliser 3.9 pour depiler "a";
while on voit "bb" sur le ruban, do:
  Utiliser 3.7;
  Utiliser deux fois 3.10 pour depiler les 2 "b";
end
```

```
Utiliser la regle 3.8 si on lit "bc";
Utiliser la regle 3.10 pour depiler "b";
Utiliser la regle 3.11 pour depiler "c";
```

On utilise des grammaires dites *augmentées* pour être sur de pouvoir regarder k symboles. On choisit \$ comme symbole de fin de mot et on ajoute la règle suivante :

$$S' \to S\$^k$$

On prendra S' comme axiome.

On a fait de l'analyse *déscendante* : on part de l'axiome jusqu'aux feuilles. C'est à dire, on dérive le non terminal le *plus à gauche*.

$$\begin{aligned} S' \to S\$^k \to aAc \to & \, abbAc\$^k \\ \to & \, abbbbAc\$^k \\ \to & \, abbbbc\$^k \end{aligned}$$

On lit le mot de *gauche à droite*. Ceci est appelé l'analyse déscendante ~$LL(k)$ (Left)

Il existe aussi l'analyse ascendante : on part du mot pour retrouver l'axiome, c'est à dire *on réduit* le terminal le plus à droite. Dans tout les cas, $A \to \alpha$ se lit "A produit α" ou "α se réduit en A".

Exemple d'analyse ascendante ~$LR(k)$:

$$\begin{gathered} abbbbbc\$^k \\ \nabla \\ abbbbAc\$^k \\ \nabla \\ abbAc\$^k \\ \nabla \\ aAc\$^k \\ \nabla \\ S\$^k \end{gathered}$$

3.1 Analyse descendante

Reconnaître(ou pas) un mot $u \in X^*$ sur le ruban, On part de S' et on a la pile.

A un instant donné, on a travaillé et on a reconnu un préfixe ω de u, $\omega \in X^*$. Pour ça on a mis $A\alpha$ dans la pile, $A \in N$

A est le sommet de pile et $\alpha \in (N \cup X)^*$
Quelle règle faut-t-il appliquer ? Choisir de *façon déterministe*

$$\begin{aligned}
S \rightarrow^* wA\alpha \\
A \rightarrow \beta \in P \\
\omega\beta\alpha \rightarrow^* wx = u \\
\omega \text{ c'est le ruban, A}\alpha \text{ c'est la pile} \\
u \in X^* \\
\omega \in X^* \\
x \in X^* \\
\alpha \in (N \cup X)^*
\end{aligned}$$

Donc :

$$\begin{aligned}
first_k(x) \in first_k(\beta\alpha) \\
first_k(x) \notin first_k(\gamma\alpha)\forall A \rightarrow \gamma \neq A \rightarrow \beta
\end{aligned}$$

Une grammaire est $LL(k)$ ssi $\forall$ regles :

$$\begin{gathered}
A \rightarrow \beta_1 \in P \\
A \rightarrow \beta_2 \in P \\
\vdots \\
A \rightarrow \beta_n \in P \\
k.lookahead(A \rightarrow \beta_i) \cap k.lookahead(A \rightarrow \beta_j) = \emptyset \\
\forall i, j (i \neq j)
\end{gathered}$$

3.2 $First_k$

Définition : soit $\alpha \in (N \cup X)^*$

$$\begin{aligned}
first_k(\alpha) = \{\omega \in X^* / |\omega| < k \text{ et } \alpha \rightarrow^* \omega \\
\text{ou } \alpha \rightarrow^* \omega\beta, \beta \in (N \cup X)^* \text{ si } |\omega| = k\}
\end{aligned}$$

Calcul ;

$$\begin{aligned} & first_0(\alpha) = \{\lambda\}, \forall \alpha \in (N \cup X)^* \\ k \geq 1 : \; & first_k(\lambda) = \{\lambda\} \\ & first_k(x) = \{x\}, \forall x \in X \\ & first_k(u) = u \text{ si } u \in X^* \text{ et } |u| \leq k \\ & \qquad\quad\;\; = x \text{ si } u \in X^* \text{ et } u = xv, |x| = k \\ & first_k(\alpha) = first_k(X_1)first_k(X_2)...first_k(X_n) \end{aligned}$$

si $\alpha = X_1X_2...X_n \forall X_i \in (X \cup N)$

Exemple pour $L(G_1) = a^n c b^n \$^k, n \geq 0$

$$\begin{aligned} S' &\to S\$^k \\ S &\to aSb \\ S &\to c \end{aligned}$$

Calcul de $first_1$:

$$\begin{aligned} first_1(S') &= first_1(S\$^k) \\ &= first_1(first_1(S)first_1(\$^k)) \\ &= first_1(S) \\ first_1(S) &= first_1(aSb) + first_1(c) \\ &= \{a, c\} \\ &= \{a\} + \{c\} \end{aligned}$$

Calcul de $first_2$:

$$\begin{aligned} first_2(S') &= first_2(S\$^k) \\ &= first_2(first_2(S)first_2(\$^k)) \\ first_2(S) &= first_2(aSb) + first_2(c) \\ &= first_2(first_2(a)first_2(S)first_2(b)) + \{c\} \\ &= afirst_1(first_2(S)first_2(b)) + \{c\} \\ &= a(\{a\} + \{c\}) + \{c\} \\ &= \{aa, ac, c\} \end{aligned}$$

Calcul de $first_3$:

$$first_3(S) = \{c, acb, aaa, aac\}$$

3.3 $Follow_k$

Definition :
Soit $A \in N$,
$follow_k(A) = \{\omega \in X^*/S' \rightarrow^* uA\alpha, u \in X^*, \alpha \in (N \cup X)^*, \omega \in first_k(\alpha)\}$

$$follow_k(S') = \{\lambda\} \forall k \geq 0$$
$$follow_k(Y) = \bigcup_{X \rightarrow \gamma Y \delta} first_k(\delta follow_k(X))$$

Exemple pour $L(G_1) = a^n c b^n \$^k, n \geq 0$

$$S' \rightarrow S\$^k$$
$$S \rightarrow aSb$$
$$S \rightarrow c$$

Calcul de $follow_1$:

$$
\begin{aligned}
follow_1(S') &= \{\lambda\} \\
follow_1(S) &= \\
\text{règle0 : } S' &\to S\$^k \\
follow_1(S) &= first_1(\$^k follow_1(S')) \\
&= \{\$\} \\
\text{règle1 : } S &\to aSb \\
follow_1(S) &= first_1(b follow_1(S)) \\
&= \{b\} \\
\text{Donc : } follow_1(S) &= \{\$, b\}
\end{aligned}
$$

Calcul de $follow_2$:

$$
\begin{aligned}
follow_2(S) &= \\
\text{règle0 : } S' &\to S\$^k \\
follow_2(S) &= first_2(\$^k ...) \\
&= \{\$^2\} \\
\text{regle1 : } S &\to aSb \\
follow_2(S) &= first_2(b follow_2(S)) \\
&= b first_1(follow_2(S)) \\
&= \{bb, b\$\}
\end{aligned}
$$

3.4 *k.lookahead*

Définition :

$$
\begin{aligned}
&\forall A \in N, \\
&\forall A \to \beta \in P, \\
&\beta \in (N \cup X)^* \\
k.lookahead(A \to \beta) &= first_k(\beta follow_k(A))
\end{aligned}
$$

Exemple sur G_1 :

$$1.lookahead(S \to aSb) = first_1(aSbfollow_1(S)) = \{a\}$$
$$1.lookahead(S \to C) = first_1(cfollow_1(S)) = \{c\}$$

3.5 $LL(k)$

Soit $G =< N, X, P, S >$ et k un entier quelconque fixé.

$$\begin{gathered} G \text{ est } LL(k) \\ \Leftrightarrow \\ \forall A \in N, \forall A \to \beta \in P \\ \text{et } \forall A \to \gamma \in P, \\ \beta, \gamma \in (N \cup X)^* \\ k.lookahead(A \to \beta) \cap k.lookahead(A \to \gamma) = \emptyset \end{gathered}$$

G_1 est $LL(1)$

Exemple : sur G_2 :

$$\begin{aligned} S' &\to S\$^k \\ S &\to aAc \\ A &\to bbA \\ A &\to b \end{aligned}$$

$G_2 LL(?)$

$L(G_2) = a(bb)^*bc\$^k = ab^{2n+1}c\$^k, n \geq 0$

G_2 n'est surement pas $LL(0)$, est-elle $LL(1)$?

Montrons que G_2 n'est pas $LL(1)$ avec calcul des 1.lookhead. On ne s'intéresse qu'à A :

$$\begin{aligned} 1.lookahead(A \to bbA) &= first_1(bbAfollow_1(A)) \\ &= \{b\} = E_1 \\ 1.lookahead(A \to b) &= first_1(bfollow_1(A)) \\ &= \{b\} = E_2 \end{aligned}$$

$E_1 \cap E_2 \neq \emptyset \Rightarrow G_2$ n'est pas $LL(1)$

Regardons si G_2 est $LL(2)$? On ne s'intéresse qu'à A :

$$\begin{aligned}
2.lookahead(A \to bbA) &= first_2(bbAfollow_2(A)) \\
&= \{bb\} = E_1 \\
2.lookahead(A \to b) &= first_2(bfollow_2(A)) \\
&= bfirst_1(follow_2(A)) = E_2
\end{aligned}$$

$Follow_2(A)? \Rightarrow$ règles 1 et 2

$$\begin{aligned}
follow_2(A) &= first_2(cfollow_2(S)) \\
&= cfirst_1(...) \\
\text{Donc } 2.lookahead(A \to b) &= \{bc\} \\
E_1 \cap E_2 &= \emptyset \\
&\Rightarrow G_2 \text{ est } LL(2)
\end{aligned}$$

3.6 Table d'analyse d'une grammaire $LL(k)$

On a une grammaire $G =< N, X, P, S' >$
On a montré que G est $LL(R)$ en calculant $k.lookahead(A \to \beta) \forall A, \forall A \to \beta$
Maintenant on construit une *table d'analyse* M tq $M(mot, sommetdepile) =$ une *seule* action.

Si $\omega \in k.lookahead(A \to \alpha)$
G_2 est $LL(2)$

$2.lookahead(A \to bba) = \{bb\} = E_1$

$2.lookahead(A \to b) = \{bc\} = E_2$

$2.lookahead(S' \to S\$^k) = first_2(S\$^k follow_2(S')) = \{ab\}$

$2.lookahead(S \to aAc) = first_2(aAcfollow_2(S)) = afirst_1(A...) = \{ab\}$

Voici la table d'analyse correspondante :

	bb	bc	ab	$\$^2$
S'			S$,0	
S			aAc,1	
A	bbA,2	b,3		
a			pop	
b	pop	pop		
c				
$				accept

Analysons un mot $\in L(G_2)$ avec la table :
le mot choisi est : *abbbbbc$*

ruban	pile	action
abbbbbc$	S'	0
abbbbbc$	S$	1
abbbbbc$	aAc$	pop
bbbbbc$	Ac$	
bbbbc$	bbAc$	
bbbc$	Ac$	2*pop
bbbc$	bbAc$	2
bc$	Ac$	2*pop
bc$	bc$	3
$	$	accept

Accept $\Rightarrow$ mot *reconnu*

3.7 Résultats théoriques

- Toute grammaire $LL(k)$ n'est *pas* ambigüe $\Leftrightarrow$ Grammaire ambigüe n'est pas $LL(K), \forall k \geq 0$
- Toute grammaire contenant *au moins* une règle *récursive à gauche* n'est pas $LL(k), \forall k$
- Une grammaire $LL(0)$ engendre un langage fini
- Tout *langage régulier* peut être engendré par une grammaire $LL(1)$

3.8 Procédures de descente récursive

3.8.1 Définitions

G grammaire $LL(1)$:
- Table d'analyse
- Procédures de descente récursive. Celles-ci doivent être dans un langage qui *supporte la récursivité.* $\Rightarrow$ La pile de l'automate sera remplacée par la pile des appels récursifs.

Analyseur syntaxique $LL(1) \equiv$ ensemble de procédures.
Outils :

scan procédure pour appeler le scanner.

nexts *variable* qui contient l'unité syntaxique détectée par le scanner.

skip t : unité syntaxique.

Listing 3.2– procedure de skip

```
procedure SKIP(t:unite syntaxique) is
begin
```

```
    if (NEXTS == t) then SCAN
    else ERREUR();
  end
```

A chaque non terminal $A \in N$, on lui fait correspondre une procédure.

Listing 3.3– procedure non terminal

```
procedure A is
  begin
    image(beta)
  end
```

telle que si $A \to \beta$, $\beta \in (N \cup X^*)$, $A \in N$ avec :

si $\beta = t \in X \Rightarrow image(\beta) = SKIP('t')$
si $\beta = B \in N \Rightarrow image(\beta) =$ appel à la procédure B
si $\beta = \lambda \Rightarrow image(\beta) =$ instruction vide $= NULL$
si $\beta = \beta_1\beta_2...\beta_n \Rightarrow image(\beta) = image(\beta_1); image(\beta_2); ...; image(\beta_n);$

si :

$$
\begin{aligned}
A &\to \beta_1 \\
A &\to \beta_2 \\
&... \\
A &\to \beta_n
\end{aligned}
$$

Listing 3.4– procedure A

```
Procedure A is
  begin
    switch NEXTS
    1.lookahead(A -> beta_1): image(beta_1);
    1.lookahead(A -> beta_2): image(beta_2);
    ...;
    1.lookahead(A -> beta_n): image(beta_n);
    others: ERREUR;
  end
```

3.8.2 Exemple

$GN = \{S', S, A\}$

$$\begin{aligned} &X = \{a, b\} \\ &S' \text{axiome} \\ &S' \to S\$ \\ &S \to aAs \\ &S \to b \\ &A \to a \\ &A \to bSaA \end{aligned}$$

LL(0) ? LL(1) ?

G est elles $LL(0)$?
Non car on a un choix pour S et pour A.
G est elles $LL(1)$?
oui.
Preuve pour S :

$$\begin{aligned} 1.lookahead(S \to aAS) &= \{a\} \\ 1.lookahead(S \to b) &= \{b\} \\ \{a\} \cap \{b\} &= \emptyset \end{aligned}$$

Preuve pour A :

$$\begin{aligned} 1.lookahead(A \to a) &= \{a\} \\ 1.lookahead(A \to bSaA) &= \{b\} \\ \{a\} \cap \{b\} &= \emptyset \end{aligned}$$

Procédures de descente récursive

Listing 3.5– procedure S'

```
procedure S' is
  begin
    SCAN;
    S;
    SKIP('$');
  end
```

Listing 3.6– procedure S

```
procedure S is
  begin
    switch NEXTS
      a: SKIP('a'); A; S;
      b: SKIP('b');
      others: ERREUR();
  end
```

Listing 3.7– procedure A

```
procedure A is
  begin
    switch NEXTS
      a: SKIP('a');
      b: SKIP('b'); S; SKIP('a'); A;
      others: ERREUR();
  end
```

3.9 Eliminer la récursivité à gauche

Grammaire augmentée

Grammaire réduite on a éliminé tous les non terminaux inaccessibles depuis S' et tous les non terminaux improductifs

exemple :

$$S' \to S\$ \tag{3.12}$$
$$S \to aAbB \tag{3.13}$$
$$A \to aA \tag{3.14}$$
$$B \to bB|\lambda \tag{3.15}$$
$$C \to ac|\lambda \tag{3.16}$$

Ici, 3.15 est improductif et C est inaccessible depuis S'.

G récursive à gauche si il existe *au moins une règle* de P de la forme $A \to A\alpha$ donc on a aussi $A \to \beta$ pour arrêter.

Or une grammaire récursive à gauche n'est *PAS* $LL(k), \forall k$, en particulier *PAS* $LL(1)$. Si on écrit la procédure correspondant au non terminal A.

$$A \to A\alpha$$
$$A \to \beta$$

Listing 3.8– procedure A

```
procedure A is
  begin
    switch ...
        A // ca va boucler
  ...
  end
```

3.9.1 Pour essayer d'éliminer la récursivité à gauche

$$A \to A\alpha$$
$$A \to \beta$$
$$\Leftrightarrow$$

Pour A :

$$A = A\alpha + \beta \Rightarrow^{Arden} A = \beta\alpha^*$$
$$A = \beta A'$$

Pour A' :

$$A' = \alpha^* \Rightarrow^{Arden} A' = \alpha A' + \lambda$$

Donc :

$$A = A\alpha + \beta \Leftrightarrow A = \beta A' \text{ et } A' = \alpha A' + \lambda$$

Ce qui donne :

$$A \to \beta A'$$
$$A \to \alpha A' | \lambda$$

Cette grammaire n'est *plus* récursive à gauche. Est-elle $LL(1)$? Si oui, on applique la procédure de descente récursive.

Exemple : soit G, récursive à gauche :

$$N = \{S', S, A\}$$
$$X = \{a, b\}$$
$$\text{axiome} : S'$$
$$P : \{$$
$$S' \to S\$$$
$$S \to SaA | A$$
$$A \to b$$
$$\}$$

$$L(A) = b$$
$$S = Sab + b \Rightarrow^{ArdenBis} L(S) = b(ab)^*$$

G' équivalente à G (càd $L(G) = L(G')$) et G' n'est plus récursive à gauche.

$$S = SaA + A = A(aA)^* \Rightarrow S = AS_1$$
$$S_1 = (aA)^* \Rightarrow^{Arden} S_1 = aAS_1 + \lambda$$

Donc G' donne :

$$\begin{aligned} S' &\to S\$ \\ S &\to AS_1 \\ S_1 &\to aAS_1|\lambda \\ A &\to b \end{aligned}$$
$$N' = \{S', S, A, S_1\}$$

G' est bien définie tel que $L(G) = L(G')$ et non récursive à gauche. G' est elle $LL(1)$? Pas de problème pour S', S et A.

Calcul des $1.lookahead$ pour S_1 :

$$1.lookahead(S_1 \to aAS_1) = \{a\}$$
$$\begin{aligned} 1.lookahead(S_1 \to \lambda) &= first_1(\lambda follow_1(S_1)) \\ &= \{\$\} = E_2 \end{aligned}$$
On a bien $\{a\} \cap \{\$\} = \emptyset$

Donc G' est $LL(1)$

Ecrivons les procédures de descente récursive pour G' :

Listing 3.9– procedure S'

```
procedure S' is
  begin
    SCAN();
    S;
    SKIP('$');
  end
```

Listing 3.10– procedure S

```
procedure S is
  begin
    A;
    S_1;
  end
```

Listing 3.11– procedure S_1

```
procedure S_1 is
  begin
    switch NEXTS
      a: SKIP('a'); A; S_1;
      $: NULL
      others: ERREUR();
  end
```

Listing 3.12– procedure A

```
procedure A is
  begin
    SKIP('b');
  end
```

Chapitre 4

Génération de code

4.1 Introduction

On veut traduire les instructions du langage de haut niveau L_1 dans un langage intermédiaire *plus proche* du langage cible. On a 3 langages.

- Pour L_1 c'est un langage impératif composé de l'affectation, et toutes les structures de contrôle(if, else, case, switch, ...) et des boucles (for, while, repeat, ...)
- Pour L_2 (langage intermédiaire) on prend un langage de quadruplets. (*choix*)
- Pour σ, on va utiliser un langage *impératif* style ADA, Pascal

Traduction σ tq $\forall$ programme P, $\forall$ donnée $PD \equiv \sigma(P)D$

4.2 Langage intermédiaire des quadruplets

Langage cible intermédiaire, ne ressemblant pas à vraiment assembleur. Les opérations se font directement en mémoire, pas de registres. Un quadruplet = instructions à 4 champs dont 3 adresses mémoire. operation, opérande1, operande2, resultat.

On peut faire l'affectation, les operations arithmetiques, les branchements conditionnels ou inconditionnels.

Affectation (:=, d, nil, e)

Opération arithmétique (+, a, b, c)

Branchement inconditionnel (goto, nil, nil @)

Listing 4.1– Exemple de quadruplets

```
@i    +, a ,b , t1  // t1 := a + b
@i+1  *, t1,c , t2  // t2 := t1 * c
@i+2  :=,t2 ,nil, d   // d:= t2
```

```
@i+3  >?,a,b,alpha1 // si a > b alors aller en alpha1
   sinon faire suivant
@i+4 ...
alpha2 goto,nil,nil,@i+3
```

4.3 Actions sémantiques couplées à l'analyseur descendant

L'analyseur syntaxique c'est le chef d'orchestre. Il est *descendant* et $LL(1)$ Il s'occupe de l'AL (Analyse Lexicale) et de faire les actions sémantiques.

- Il va engendrer un programme *équivalent* en *quadruplets*. Il n'y a *PAS D'EXECUTION*.
- Pour ça on va avoir besoin d'informations à *mémoriser* ou à *modifier*.
- Les procédures de descente récursive vont avoir besoin de paramètres en entrée ou en sortie et de variables locales.

Exemple :
Les expressions arithmétiques définies par la grammaire suivante :

$$E \to T\{+T\}^*$$
$$T \to F\{*F\}^*$$
$$F \to \text{ident}|(E)$$

Notons que le symbole * sert ici pour le langage et pour définir l'opérateur de multiplication.

Listing 4.2– Procedure E

```
Procedure E is
  begin
    T;
    while NEXTS = '+' loop
      SKIP('+');
      T;
      //* => engendre un quadruplet boite a outils
      GEN(QUAD: String);
      //engendre le prochain quadruplet (A,B,C,D)
      GEN("+,?,?,?");
    endloop
  end // E
```

Il faut ajouter des arguments aux procédures E et T.
$E \to T\{+T\}^*$

Listing 4.3– Procedure E avec arguments

```
Procedure E (out r :String) is
  u, t :String;
  begin
    T(r);
    while NEXTS = '+' loop
      SKIP('+');
      T(t);    //2e operande
      u := NEWTEMP;
      GEN("+","^r^","^t^","^u); // ^ est la
         concatenation de chaines
      r := u;
    endloop
  end //E
```

NEWTEMP permet de créer une variable temporaire.

$T \rightarrow F\{*F\}^*$

Listing 4.4– Procedure T avec arguments

```
Procedure T (out r :String) is
  t :String;
  begin
    F(r); //1e operande
    while NEXTS = '*' loop
      SKIP('*');
      F(t); //2e operande
      u := NEWTEMP;
      GEN("*,"^ r ^","^ t ^","^ u);
      r := u;
    endloop
  end //T
```

$F \rightarrow \text{ident}|(E)$

Listing 4.5– Procedure F

```
Procedure F (out r :String) is
  begin
    switch NEXTS:
      'ident':
        r:= ident.string;
        SCAN;
      '(':
```

```
    SKIP('(');
    E(r);
    SKIP(')');

  others:
    ERREUR();
end
```

L'analyseur lexical a trouvé comme prochaine US un 'ident' et il a construit la chaine de caractères qui constitue l'identificateur : cet attribut sémantique est accessible à *ident.string*.

4.4 Methodologie pour la traduction des structures de controle

Traduction dirigée par la syntaxe, en *un seul* passage(lecture) du programme. Pour le moment, l'analyseur est descendant $LL(1)$.
Il y a 3 étapes :

- Donner le schéma en quadruplets correspond à la structure de controle traduite. Donner la sémantique de la structure de controle en quadruplets. ie, ce qu'on *veut obtenir*
- Donner l'inventaire des *problèmes* qui peuvent apparaitre et leurs *solutions*
- Ecrire les *procédures de traduction* pour obtenir le schéma en quadruplets, ie comment on fait pour y arriver.

Pour ça, on a la boite à outils.

4.4.1 Rappels

Listing 4.6– Procedures connues

```
<statlist>  ::= <inst> <suite>
<suite>     ::= ; <statlist> | lambda
<inst>      ::= <ifstat> | <whilestat> | <repeatstat>
   | <casestat> | ...
```

On a les procédures STATLIST, SUITE, INST. INST est un gros aiguillage suivant NEXTS.

Par exemple :

if appel de IFSTAT

while appel de WHILESTAT

⋮ ⋮

case appel de CASESTAT

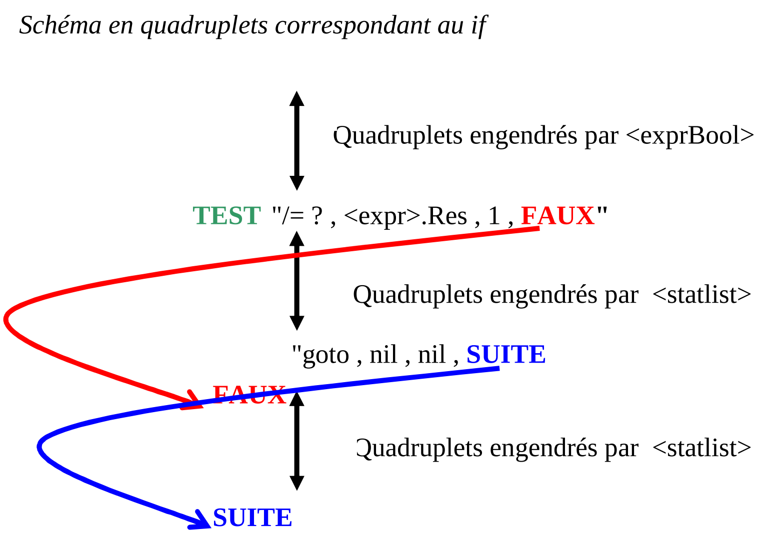
4.4.2 Exemple de traduction sur ifstat

Listing 4.7– ifstat

```
<ifstat> ::= if <exprBool> then <statlist> else <
   statlist> endif
```

Schema en quadruplets équivalent à <ifstat>

FIGURE 4.1– Schéma en quadruplets du if



Problèmes et solutions

- Il faut verifier le type. Verifier que la variable qui contiendra le résultat de <expr> est de type *booleen*
  ⇒ solution : procédure CHECKTYPE (S : string, T : type)
- Réference en avant à FAUX et à SUITE. On engendre un quadruplet *incomplet* en mettant nil à la place de l'adresse(@). Ensuite, on mémorise l'@ du quadruplet incomplet dans une *variable*. Une fois qu'on connait la bonne adresse, on met à jour le champ @ du quadruplet

incomplet. Pour mettre à jour, on utilise BACKPATCH(L : liste de quadruplets, A : int) ou A est une adresse de quadruplet.

Ecriture des procédures de traduction

Listing 4.8– procedure IFSTAT

```
procedure IFSTAT is
  Res :String;
  Test: Integer;  //@ de quadruplet
  begin
    SKIP('if');
    EXP(Res);
    CHECKTYPE(Res, Boolean);
    SKIP('then');
    Test := NEXTQUAD; //memoriser l'@ du quad
       incomplet
    GEN("/=?, "^ Res ^","^ 1 ^", nil"); //engendre
       quad incomplet
    STATLIST;
    Suite := NEXTQUAD; //memorise @ du quad incomplet
    GEN("goto, nil, nil, nil"); //engendre 2e quad
       incomplet
    SKIP('else');
    BACKPATCH([Test], NEXTQUAD) // mise a jour du
       champ @ du quad incomplet Test avec l'@ du
       prochain quadruplet
    STATLIST;
    BACKPATCH([SUITE], NEXTQUAD) // maj du champs @ du
        quad incomplet Suite avec l'@ du prochain
       quadruplet.
    SKIP('endif');

  end //IFSTAT
```

Traduction – Dirigée par la syntaxe. L'analyseur $LL(1)$ coordonne.
- en *un seul* passage
- On obtient un programme en quadruplets

Methodologie – Schéma équivalent en quadruplets
- Problèmes pour les actions sémantiques de la solution
- Mise en oeuvre avec les procédures de traduction(boite à outils)

4.4.3 Exemple de traduction sur whilestat

Listing 4.9– whilestat

```
<whilestat> ::= while <exprBool> loop <statlist>
   endloop
```

Schéma équivalent

$$
\begin{array}{l}
DEBUT \\
\uparrow \\
\vdots \text{ quadruplets pour } < exp > \\
\downarrow \\
=?, < expr > .res, 0, ?, FIN \leftarrow \text{ne pas rentrer dans la boucle} \\
\uparrow \\
\vdots \text{ quadruplets pour } < statlist > \\
\downarrow \\
goto, nil, nil, DEBUT \\
FIN
\end{array}
$$

Problèmes liés à la solution

Verification de type verifier que $< expr > .res$ est de type Boolean. On peut utiliser $CHECKTYPE$ pour ça.

Référence en avant à FIN – Engendrer un quadruplet *incomplet* c'est à dire qu'on met nil dans le champs adresse.
- Mémoriser l'adresse de ce quadruplet incomplet dans une variable $TEST$
- Mettre à jour le champ adresse du quad incomplet $TEST$ qui connait l'adresse FIN. On utilise $BACKPATCH$.

Référence en arrière à $DEBUT$ – Mémoriser l'adresse de quadruplet $DEBUT$ avant de l'engendrer dans une variable $DEBUT$

Procédure de traduction

Listing 4.10– procedure WHILESTAT

```
procedure WHILESTAT is
  DEBUT, TEST: Integer;
  Res: String;
  begin
```

```
    SKIP('while');
    DEBUT := NEXTQUAD; //memoriser @ debut
    EXP(Res);
    CHECKTYPE(Res, bool); //verif que expr.res est
       bool
    SKIP('loop');
    TEST := NEXTQUAD; //memoriser @ fin
    GEN("=?,"^Res^",0,nil"); //engendrer un quad
       incomplet avec nil
    STATLIST:
    SKIP('endloop');
    GEN("goto,nil,nil,"^str(DEBUT)); //str: itoa (
       integer to ascii)
    BACKPATCH([TEST], NEXTQUAD); //maj du quad
       incomplet
  end //whilestat
```

4.4.4 Traduction de déclarations

Définitions :

Listing 4.11– declaration

```
<declaration> ::= ident : <element>
```

Listing 4.12– element

```
<element> ::= Integer | array[Integer] of <element>
```

Exemples :

Listing 4.13– exemples de declarations

```
i: Integer;
j: Integer;
r: Integer;
T: array[10] of Integer;
tab: array[10] of array[20] of array[3] of [integer];
```

Ici, il n'y a pas de génération de quadruplets. Mais il faut *ranger les infos* issues de la déclaration dans la Table Des Symboles (TDS). Les informations à stocker sont :

- symbole (chaine de caractère correspondant au nom de variable)
- type (ici : Integer, Array, ...)
- taille de l'information (pour Integer : 1, Pour un tableau : dimensions, ...)
- ...

Exemple de table de symboles :

symbole	type	taille
i	int	1
t	vector	10
none	int	1
tab	vector	$10 * 20 * 3 * 1$
none	vector	$20 * 3 * 1$
none	vector	$3 * 1$
none	int	1

Opérations sur la TDS

Listing 4.14– Operations sur la TDS

```
ENTER(S: String, P: out Integer);
SEARCH(S: String, P: out Integer);
CANCEL(S: String);
```

Accès à la TDS :

Listing 4.15– Operations sur la TDS

```
  TDS[P].nomAttribut
```

Par convention, le premier indice de la TDS est 1. De plus, *on ne veut pas* plusieurs déclarations de même nom, même si type différent. Ca implique qu'il faut vérifier à chaque declaration si le symbole est déjà existant dans la TDS. Pour ça, on utilise $SEARCH$. Si $SEARCH$ renvoye 0, alors l'entrée n'existe pas. Sinon, il y a $ERREUR$

Listing 4.16– Procedures DECLARATION et ELEMENT

```
procedure DECLARATION is
P: Integer;
taille: Integer;
begin
  if NEXTS != 'ident' then ERREUR();
  else
    SEARCH(NEXTS.String, P);
    if (P != 0) then ERREUR();
    else //pas encore dans TDS
      ENTER(NEXTS.String, P);
      SKIP(':');
      ELEMENT(P,taille);
    endif
  endif
```

```
end //DECLARATION

procedure ELEMENT is
begin
  switch NEXTS
  'Integer':
    TDS[P].type := int;
    TDS[P].taille := 1;
    taille := 1;
    SCAN;
  'array':
    SKIP('array');
    SKIP('[');
    TDS[P].type := vector;
    if NEXTS != 'number' then ERREUR();
    else
      nbreElem := NEXTS.Valeur;
      SKIP(']');
      SKIP('of');
      ELEMENT(P+1,taille);
      taille := nbreElem * taille;
      TDS[P].taille := taille;
    endif
  others:
    ERREUR();
end //ELEMENT
```

L'analyseur lexical si il trouve un :

identificateur US ident et un attribut string qui contient la chaine de caractères constituant l'ident

entier attribut valeur qui contient la valeur decimale de l'entier reconnu.

4.4.5 Conclusion sur les actions sémantiques couplées avec analyse syntaxique $LL(1)$

- avantages
  - simple
  - souple(extensible)
  - pas de gestion de pile
- inconvénients
  - mélange action sémantiques et traitement de la syntaxe : ERREUR traite d'erreurs syntaxiques *et* sémantiques.
  - Difficulté d'optimisation

A présent, on va voir une approche ascendante grammaire $LR(1)$. Celle ci implique une gestion de pile plus importante, mais moins de traitement de syntaxe.

4.5 analyse ascendante

Jusqu'a là nous avions une grammaire $LL(1)$ qui faisait une analyse *descendante* avec des procédures de descente récursive.

Maintenant, grammaire $LR(1) \Rightarrow$ analyseur *ascendant* et déterminer où il faut mettre les actions sémantiques.

4.5.1 Definitions

On veut analyser un mot des terminaux vers l'axiome. (on "remonte" vers l'axiome). Dans G, si on a $A \to \beta \in P, A \in N, \beta \in (N \cup X)^*$

On avait :
A se dérive en β :

$$\begin{array}{c} A \\ | \\ \beta \end{array}$$

là on a :
β se reduit en A :

$$\begin{array}{c} \beta \\ \nabla \\ A \end{array}$$

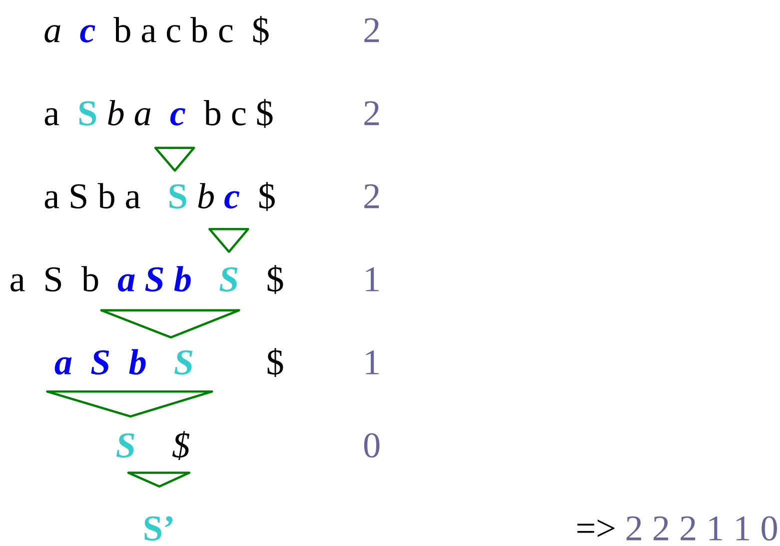
Faire de l'analyse ascendante d'un mot $\omega \in L(G)$ c'est partir des terminaux de ω, reconnaitre des manches pour les réduire en non terminaux et ça jusqu'a à l'axiome S'

On lit le mot de *gauche*(LL : Left) à droite et on dérive le non terminal le plus à droite(LR : Right).

4.5.2 Exemple

$$G =< \{a, b, c\}, \{S, S'\}, P, S' >$$
P :
$$S \rightarrow^0 S\$$$
$$S \rightarrow^1 aSbS$$
$$S \rightarrow^2 c$$

FIGURE 4.2– Analyse de $\omega = acbacbc\$$



$$S' \Rightarrow S\$ \Rightarrow aSbS\$ \Rightarrow aSbaSbS\$$$
$$\Rightarrow aSbaSbc\$$$
$$\Rightarrow aSbacbc\$$$
$$\Rightarrow acbacbc\$$$

L’ordre "chronologique" de réduction correpond à l’orde des dérivation du non terminal le plus à droite.

4.5.3 automatiser ce type d'analyse

Pour faire "automatiquement" ce type d'analyse
- On a les règles de P
- On a le mot sur le ruban
- On a la pile de l'automate

Il faut que l'analyse soit *déterministe*.

Analyseur k.prédictif regarder k symboles sur le ruban pour décider. Nous on se contente de k=0 ou k=1.

Exemple

$$
\begin{aligned}
&P : \\
&\quad S \to S\$ \\
&\quad S \to aSbS \\
&\quad S \to c
\end{aligned}
$$

pile	sommet	ruban	action
		acbacbc$	Decaler 'a'
	a	cbacbc$	Decaler 'c'
a	c	bacbc$	Reduire 2
	∇		
a	S	bacbc$	Decaler 'b'
aS	b	acbc$	Decaler 'a'
aSb	a	cbc$	Decaler 'c'
aSba	c	bc$	Reduire 2
aSba	S	bc$	Decaler 'b'
aSbaS	b	c$	Decaler 'c'
aSbaSb	c	$	Reduire 2
aSbaSb	S	$	Reduire 1
aSb	S	$	Reduire 1
S	S	$	Decaler '$'
S	$		ACCEPT

Actions :

SHIFT décalage du premier symbole lu du ruban au sommet de pile

REDUCE reconnaitre un manche β en sommet de pile et le réduire en A si $A \to \beta$

ACCEPT accepter le mot lu si on trouve S' au sommet de pile

Conflits

SHIFT/REDUCE On ne sait pas décider si il faut réduire ou décaler

REDUCE/REDUCE On ne sait pas décider quelle réduction on doit appliquer.

Pour décider de façon déterministe, on doit anticiper et regarder k symboles sur le ruban. Pour construire un analyseur ascendant $LR(k)$, il faut calculer des *classes d'items* (algorithme) et pour les grammaires $LR(1)$, il existe des outils qui engendrent *automatiquement* l'analyseur $LR(1)$: YACC, OCAMLYACC, BISON, ...

4.5.4 Exemple d'analyseur $LR(1)$

$L = ab^{2n+1}c, n \geq 0$

G_1 :

$$\begin{aligned} S' &\to^0 S\$ \\ S &\to^1 aAc \\ A &\to^2 Abb \\ A &\to^3 b \end{aligned}$$

G_2 :

$$\begin{aligned} S' &\to^0 S\$ \\ S &\to^1 aAc \\ A &\to^2 bAb \\ A &\to^3 b \end{aligned}$$

G_3 :

$$\begin{aligned} S' &\to^0 S\$ \\ S &\to^1 aAc \\ A &\to^2 bbA \\ A &\to^3 b \end{aligned}$$

On prends $\omega = abbbbbc\$$

Pour G_1

pile	ruban	action
	abbbbbc$	Decaler 'a'
a	bbbbbc$	Decaler 'b'
ab	bbbbc$	Reduire 3
aA	bbbbc$	
aAb	bbbc$	Decaler 'b'
aAbb	bbc$	Decaler 'b'
aA	bbc$	Reduire 2
aAb	bc$	Decaler 'b'
aAbb	c$	Decaler 'b'
aA	c$	Reduire 2
aAc	$	Decaler 'c'
S	$	Reduire 1
S		Decaler '$'
S$		Reduire 0
S'		ACCEPT

Listing 4.17– Algo de reconnaissance

```
Decaler 'a'
Decaler 'b' et Reduire b en A // (3)
while on lit 'b' do
  Decaler 'b'
  Decaler 'b'
  Reduire "Abb" en A // (2)
endloop
Decaler 'c'
Reduire "aAc" en S // (1)
Decaler $
Reduire S$ en S' // (0)
```

Pour G_3

pile	ruban	action
	abbbbbc$	D
a	bbbbbc$	D
ab	bbbbc$	D
abb	bbbc$	D
abbb	bbc$	D
abbbb	bc$	D
abbbbb	c$	D
abbbbA	c$	Réduire b en A
abbA	c$	Réduire bbA en A
aA	c$	Réduire bbA en A
aAc	$	Decaler 'c'
...	...	...

Ici, on a du prendre une décision entre "Reduire b en A" ou "Decaler b". On prends k = 1. On peut décider en regardant un symbole sur le ruban : on décale 'b' tant qu'on voit 'b' sur le ruban et on commence à réduire quand on voit 'c' sur le ruban. Pour G_3, l'algo est déterministe si on regarde un symbole sur le ruban : $LR(1)$

Pour G_2

Il faut décaler 'b' jusqu'au b *du milieu* avant de commencer à réduire. On *ne sait pas* décider même en regardant k symboles si il faut réduire le sommet de pile en A ou bien décaler le prochain b. G_2 n'est PAS $LR(k), \forall k$

4.6 Piles syntaxiques et sémantiques

On a une grammaire $LR(1)$ et son analyseur *ascendant* (construit automatiquement avec YACC par ex.) On veut coupler les actions sémantiques (génération de quadruplets). Le problème c'est de savoir où mettre ces actions. $\Rightarrow$ mettre les actions sémantiques(du code à exécuter) au moment de la *réduction d'un manche.*

Pile syntaxique :

$$\alpha\beta_1\beta_2...\beta_n \text{ et on a } A \to \beta_1, \beta_2, ..., \beta_n \in P$$
$$\nabla$$
$$\alpha A$$

Où trouver les informations nécéssaires ?. Il faut piocher ça dans les *piles sémantiques.* On prend *une pile semantique* par *info à conserver* (attribut à conserver)

Les piles sémantiques sont gérées en même temps que la pile syntaxique et les attributs sont associés à *chaque non terminal*

pile syntaxique	$\alpha_\beta \beta_1 \beta_2 ... \beta_n$
pile semantique res	$\beta.res$
pile semantique adr	...

Note : $\beta_1 \ldots \beta_n$ se réduit en un non terminal β à qui on associe un attribut sémantique dans le pile res.

4.6.1 Exemple avec pile sémantique

Listing 4.18– whilestat

```
<whilestat> ::= while <exprBool> loop <statlist>
   endloop
```

Schéma équivalent

$$
\begin{array}{l}
\quad DEBUT \\
\uparrow \\
\vdots \text{ quadruplets pour } <exp> \\
\downarrow \\
=?, <expr>.res, 0, ?, FIN \leftarrow \text{ne pas rentrer dans la boucle} \\
\uparrow \\
\vdots \text{ quadruplets pour } <statlist> \\
\downarrow \\
\quad goto, nil, nil, DEBUT \\
\quad FIN
\end{array}
$$

Problèmes liées à la solution

Verification de type verifier que $<expr>.res$ est de type Boolean. On peut utiliser $CHECKTYPE$ pour ça.

Référence en avant à FIN
- Engendrer un quadruplet *incomplet* c'est à dire qu'on mets nil dans le champs adresse.
- Mémoriser l'adresse de ce quadruplet incomplet dans une variable $TEST$
- Mettre à jour le champs adresse du quad incomplet $TEST$ qui connait l'adresse FIN. On utilise $BACKPATCH$.

Référence en arrière à $DEBUT$
- Mémoriser l'adresse de quadruplet $DEBUT$ avant de l'engendrer dans une variable $DEBUT$

Piles sémantiques

- pile des résultats de <expr> ⇒ pile *Res*
- pile des adresses(@) des quadruplets à conserver ⇒ pile *Adr*

Coupures ? si oui, ou ?

- on modifie la grammaire en ajoutant des règles factices. ($M \to \lambda$, $N \to \lambda$)

Solution

pile synt	while λ
pile res	
pile adr	

Listing 4.19– premiere etape

```
Reduire M -> lambda
Action semantique {
  M.Adr := NEXTQUAD;
}
```

pile synt	while M <expr> loop λ
pile res	<expr>.res
pile adr	M.adr

Listing 4.20– deuxieme etape

```
Reduire N -> lambda
Action semantique {
  N.Adr := NEXTQUAD;
  CHECKTYPE(<expr>.Res, bool);
  GEN("=?,"^<expr>.Res^",0,nil");
}
```

pile synt	while M <expr> loop N <statlist> endloop
pile res	<expr>.res
pile adr	M.adr , N.adr

Listing 4.21– troisieme etape

```
Reduire while ... endloop-> whilestat
Action semantique {
  GEN("goto,nil,nil,"^str(M.Adr));
  BACKPATCH([N.Adr], NEXTQUAD);
}
```