CONCEPTION et OPTIMISATION
des DESCENTES RECURSIVES

HUÀ-Thanh-Huy

Résumé

On introduit une nouvelle famille d'automates à pile, appelés automates simples, adaptés à la conception systématique et aux manipulations formelles des analyseurs de descente récursive déterministes. Ces automates simples -analogues au "separable transition diagram"- sont construits systématiquement à partir des règles $LL(1+\epsilon)$ -des $LL(1)$ pouvant être ambiguës sur ϵ - mises sous une forme normalisée des "regular right part grammars" appelée FBNBF. Ils font l'objet de manipulations formelles pouvant être utilisées comme optimisation d'analyseurs. On étudie aussi leur programmation itérative.

Abstract

A new class of pushdown automata, called simple automata, for systematic design and formal manipulation of deterministic recursive descent is investigated. These parsing automata -similar to separable transition diagrams- systematically obtained from $LL(1+\epsilon)$ rules - $LL(1)$ rules which may be ambiguous about ϵ - which are in a normal form of regular right part grammars, called FBNBF. Manipulations on these automata lead to optimization of the parsing. An iterative programming of these recursive automata is studied.

Laboratoire "Informatique Théorique et Programmation" (LITP-CNRS LA 248)

et Institut de Programmation, 55-65 (428),
Place Jussieu, 75320 Paris Cédex 05 (FRANCE)

0. INTRODUCTION

La méthode d'analyse syntactique par descente récursive, utilisée dès 1961 par Lucas, [Bauer 74], popularisée par [Conway 63], est aujourd'hui de plus en plus utilisée. Elle a suscité une abondante littérature tant pour sa mise en forme, [Tixier 67] et [Lomet 75], que pour sa mise en oeuvre [Gries 68 et 71], [Bordier 71], [Griffiths 74], [Friede 78],...

Indépendamment, L. Nolin a proposé de concevoir les analyseurs descendants sous forme d'automates, dits "automates simples" [Nolin 66]. La construction de ces automates requiert d'une part une normalisation de la notation de Backus-Naur, d'autre part que la grammaire normalisée vérifie une condition (grammaire factorisée) qui est en fait équivalente à la condition $LL(1)$, [Hua 80]. Cette approche conduit à une représentation de l'analyseur par des tables qui sont à la fois table de transition et table d'analyse (jouant le même rôle que les "goto table" et "parsing table" de l'analyse $LR(k)$, mais il s'agit ici d'une analyse descendante), à la différence de [Gries 68] qui suggérerait de gérer les procédures récursives de l'analyseur par l'intermédiaire d'une table de transition ; soulignons au passage que les "tables d'analyse" traditionnelles de l'analyse $LL(1)$ peuvent servir à concevoir un analyseur récursif mais non à le réaliser.

Cette technique de représentation tabulaire présente de très nombreux avantages, [Pleban 79], et un seul défaut : la place occupée. Outre les techniques de représentation adéquate pour réduire cette place, d'autres travaux s'orientent vers les techniques formelles d'optimisation de ces tables. Ces techniques d'optimisation ont fait et font encore l'objet de recherches poussées pour l'analyse ascendante, essentiellement $LR(1)$, cf. le tome II de [Aho Ullman 73].

Le présent travail reprend les idées de L. Nolin. Il donne une description formelle des automates simples et de diverses manipulations permettant de les optimiser : regroupement de colonnes représentant les symboles terminaux, d'où un gain de place, ou suppression de certaines lignes, d'où un gain un temps de calcul. Ces analyseurs se construisent pour la famille des grammaires que nous appelons $LL(1 + \epsilon)$, à savoir les grammaires $LL(1)$ auxquelles on autorise plus d'une dérivation donnant le mot vide ϵ , mises sous la forme de Nolin de la BNF, en abrégé FBNBF.

En outre, nous proposons de réaliser ces analyseurs récursifs de manière itérative (engérant une pile, bien entendu). On peut alors utiliser une modification des automates récursifs qui ne semble pas résulter des méthodes classiques, [Arsac 77]. En revenant aux grammaires, on retrouve ainsi des "regular right part grammars" sous une forme normalisée, une FBNBF élargie notée FBNBF. Il est facile de voir que toute grammaire "à membre droit régulier" peut être mise sous FBNBF, ce qui permet de construire directement un analyseur descendant itératif optimisé ou, le cas échéant, de diagnostiquer les incompatibilités.

Les propriétés énoncées ici sont sans preuves, celles-ci se trouvant dans notre thèse en préparation [Hua 80]. A moins d'une redéfinition explicite, les terminologie et notations utilisées sont classiques (cf. par exemple [Aho-Ullman 72]).

1. RAPPELS ET DEFINITIONS

1.1. Grammaire algébrique et BNF

Une grammaire algébrique (ou acontextuelle, context-free,...) G est le doublet $\langle V, P \rangle$ où

- le vocabulaire $V = \{v_i \mid 1 \leq i \leq +\infty\}$ est partitionné en
  - le vocabulaire (des) auxiliaire(s) $V_a = \{A_i \mid 0 \leq i \leq +\infty\}$
  - le vocabulaire (des) terminal(aux) $V_t = \{t_i \mid 0 \leq i \leq +\infty\}$
- l'ensemble $P \subset V \times V^*$ de relations en nombre non nul et fini appelées productions et notées $A_i \rightarrow m_i$ qui sont des A_i -productions.

Partitionnons P en règles $\langle A_i \rangle$ de toutes les A_i -productions de P : $\langle A_i \rangle = \{A_i \rightarrow m_{i1}, \dots, A_i \rightarrow m_{in}\}$. Cette règle $\langle A_i \rangle$ est dite (sous) BNF si elle est notée

$$A_i \rightarrow m_{i1} \mid \dots \mid m_{in}$$

$G = \langle V, R \rangle$ et R sont dites (sous) BNF si toutes les règles de R sont (sous) BNF.

Soit les règles $\langle A_i \rangle$ BNF

- $\langle A_i \rangle$ est une règle de type s (simple, séquence,...) ou est une s-règle si $\langle A_i \rangle$ est de la forme $A_i \rightarrow v_1 \dots v_p$ où $v_i \in V$ et $p > 1$
- $\langle A_i \rangle$ est une c-règle (composée, choix) si $\langle A_i \rangle$ est de la forme $A_i \rightarrow m_1 | \dots | m_n$ où $m_i \in V^*$ (resp. $m_i \in V$) et $n > 1$ (resp. $n = 1$).

Pour abréger, nous noterons ces règles

$$A_i \rightarrow \underset{i \in [1, p]}{.} v_i \quad (\text{resp. } A_i \rightarrow \underset{j \in [1, n]}{.} m_j)$$

Définition. Une grammaire algébrique d'axiome A_0 est dite épurée si aucun de ses auxiliaires A_i soit tel que $L(A_i) = \emptyset$ ou $L(A_i) = \{\epsilon\}$ ou inaccessible depuis A_0 .

Propriété. Toute règle BNF d'une grammaire algébrique épurée est soit une c-règle soit une s-règle

1.2. FNBNF (forme de Nolin de la BNF).

Les règles d'une grammaire algébrique $G = \langle V, R \rangle$ sous FNBNF sont de l'une des deux formes :

- $A_i \rightarrow \underset{i \in [1, e]}{.} v_i$ où $v_i \in V \cup \epsilon$ et $e \geq 1$

$$\bullet A_i \rightarrow \underset{i \in [1, e]}{.} v_i' \text{ où } v_i' \in V \text{ et } e > 1$$

Propriétés. • toute grammaire algébrique peut être mise sous FNBNF

- toute FNBNF est une BNF

1.3. Grammaires LL(1+ε).

Une grammaire algébrique d'axiome A_0 épurée et sous BNF est dite LL(1+ε) si toutes ses règles $\langle A_i \rangle$ sont LL(1+ε).

$\langle A_i \rangle$ et A_i sont dits LL(1+ε) si $\langle A_i \rangle$ vérifie les deux conditions

- $(\text{prem}(A_i) \cap \text{suiv}(A_i, A_0) = \emptyset) \vee (\bigcap \text{epsi}(A_i))$
- pour toute c-règle $A_i \rightarrow \underset{i \in [1, p]}{.} m_i$ alors $\bigcup_{i \neq j} (\text{prem}(m_i) \cap \text{prem}(m_j)) = \emptyset$

où avec la notation classique $\xrightarrow{*}$ pour 'dérivation au sens large' dans une grammaire

$G = \langle (V_a \cup V_t), R \rangle$ donnée, on dispose de

$$\text{prem}(m) := \{t \in V_t \mid m' \in (V_a \cup V_t)^* : m \xrightarrow{*} tm'\}, m \in (V_a \cup V_t)^*$$

$$\text{suiv}(v, v') := \{t \in V_t \mid \underset{m}{.} m' \in (V_a \cup V_t)^* : v' \xrightarrow{*} m t m'\}, \underset{v}{.} v' \in V_a \cup V_t$$

$$\text{epsi}(m) := \text{si } m \xrightarrow{*} \epsilon \text{ alors vrai sinon faux fsi}$$

Propriétés : a) Si $\langle A_i \rangle$ LL(1+ε) est ambiguë alors $\langle A_i \rangle$ ne peut être qu'une c-règle $A_i \rightarrow \underset{i' \in [1, p]}{.} m_{i'}$ où $\exists i' \neq j'$ tel que $\text{epsi}(m_{i'}) \wedge \text{epsi}(m_{j'}) = \text{vrai}$

b) Si cette dernière condition n'est pas vérifiée par une c-règle $\langle A_i \rangle$ LL(1+ε) alors $\langle A_i \rangle$ est LL(1) au sens de Lewis-Stearns, [Aho-Ullman 72]

c) Pour toute règle LL(1+ε), il existe une règle LL(1) équivalente

1.4. Descentes récursives

Cette très classique et intuitive technique d'analyse utilise un jeu de procédures égal au nombre de règles de la grammaire. Le corps de chaque procédure est calqué sur la "lecture de gauche à droite" des différents membres droits de règle, la rencontre d'un terminal provoquant sa lecture effective sur le mot d'entrée, celui d'un auxiliaire l'appel de la procédure correspondante. On a en général besoin d'un caractère de "lookahead" (grammaire LL(1)) pour choisir entre plusieurs possibilités. L'appel récursif de ces procédures entre-elles induit ainsi la stratégie d'analyse descendante. Cf. [Gries 71], [Grieffiths 74], ...

2. AUTOMATES SIMPLES

Ces automates sont une représentation matricielle des procédures de descente récursive de grammaire LL(1+ε) mise sous FBNBF et augmentée d'un méta-axiome $M \rightarrow A \#$ où A est l'axiome de la grammaire. A moins d'une mention contraire, nous ne travaillerons qu'avec ces grammaires.

Exemple Soit $\mathcal{G} = \langle \{A, a, b, c\}, \{A \rightarrow aAbA \mid c \mid \epsilon, M_0 \rightarrow A\# \} \rangle$ qui est LL(1+ε) et même LL(1). Les règles FBNBF de cette grammaire du langage des parenthèses sont

$$A \rightarrow B \mid C \mid c$$

$$B \rightarrow aAbA$$

$$M_0 \rightarrow A \# \text{ et leur automate}$$

simple associé :

	a	b	c	#
A	1	B	ε	→
	2	σ	σ	σ

	c	b	a	#
1			→	
2	A	ε	A	ε
3		→		
4	A	ε	A	ε
5	σ	σ	σ	σ

	b	#	a	c
1	ε	ε	A	A
2		α		
3	σ	σ	σ	σ

2.1. Conception des automates simples

2.1.1. Forme d'un automate simple. Un automate simple, a.s. en abrégé, associé à une règle $\langle A \rangle$ et noté $\mathcal{A}$ (majuscule arrondie correspondante) est représenté par une matrice de transition où

- chaque colonne de $\mathcal{A}$, correspond au terminal t de look-ahead de la pile d'analyse et à l'alphabet d'entrée d'un automate
- chaque ligne e de $\mathcal{A}$ ou état e de $\mathcal{A}$, noté $\mathcal{A}[e, \cdot]$ comporte des actions $\mathcal{A}[e, t]$ fonction de l'état e et du terminal t . Il y a $|\mathcal{A}|$ états pour $\mathcal{A}$.

2.1.2. Les actions d'un a.s. sont de deux types :

a. les *actions spécifiques*, analogues au classique alphabet de sortie, sont : 'ε' (instruction vide Algol 60), '→' (avancer d'un terminal sur le texte d'entrée), 'σ' (sortie de l'automate), 'α' (arrêt général) et '□' (action non-définie, pratiquement utilisée comme erreur).

b. les *actions calculées*, constituant "l'alphabet de pile" portent le nom des auxiliaires.

2.1.3. Fonctionnement d'un a.s. Le fonctionnement d'un a.s. est analogue à celui d'une procédure Algol 60 sans paramètre, travaillant sur une donnée globale qu'est le texte d'entrée et dont le corps est la séquence des états de $\mathcal{A}$, notés de 1 à $|\mathcal{A}|$ pour la commodité, qui seront *activités* séquentiellement depuis 1 jusqu'à $|\mathcal{A}|$

par pas de 1. L'activation d'un état revient à exécuter une action $A[e,t]$ unique par rapport à l'entrée t :

- . $A[e,t]$ est une action spécifique auquel cas
  - . si c'est ' ϵ ' ou ' $\rightarrow$ ' on exécute et on passe à l'état $e+1$
  - . si c'est ' σ ' on l'exécute en réalisant une sortie de A pour revenir à l'état suivant l'état qui a appelé A (voir ci-dessous).
  - . si c'est ' α ' on arrête tout
  - . si c'est ' $\square$ ' l'exécution est non définie

. $A[e,t]$ est une action calculée, par exemple ' B ', auquel cas on exécute l'appel de la procédure B par activation de l'état 1 de B . Lorsque B est activé, i.e. exécution de ' σ ' dans B , alors on revient à l'état $e+1$ de A .

2.1.4. Calcul d'un a.s. Les seuls liens des a.s. sont des appels de procédure au sens Algol 60, c'est-à-dire :

- les a.s. n'ont qu'une seule entrée (l'état 1) et une seule sortie (sortie normale par ' σ ' ou ' α ', ou anormale par ' $\square$ '). Les a.s. sont donc des automates "single-exit" au sens de [Tixier 67] et non les "multiple exit" de [Conway 63] formalisés par [Lomet 73].
- la donnée qui est le texte d'entrée à analyser, est une variable globale pour tous les automates.

Nous appellerons *calcul de A sur un mot m* , noté $A(m)$, la trace (ou signature) chronologique de tous les états activés des différents automates à partir de l'activation de l'état 1, $A[1,.]$, de A , et $|A(m)|$ le nombre total d'états activés à partir de $A[1,.]$ jusqu'à une sortie (normale ou non).

2.2. Construction des automates simples.

Pour l'écriture des algorithmes de construction, nous employons un simili Algol 68 où ' $'$ ' dénote la collatéralité, pour chaque une boucle de répétition collatérale, et cas le choix collatéral (cf. [Gerbier 77], [Keller 79]).

D'autre part, les manipulations des terminaux d'un a.s. amènent à associer les colonnes aux classes de terminaux t , cf. 3.2. ; aussi, nous introduisons la notion de *index* (t), en abrégé *index* (t), comme numérotation des colonnes. L'ensemble des *index* (t), $t \in V_a \cup \#$, de l'a.s. A forme *index* (A). Pour le moment, nous utilisons la codification 1-1 entre 1 et $|V_t|+1$ pour *index* (t).

Soit une grammaire G augmentée $LL(1+\epsilon)$ et mise sous FBNBF. Les algorithmes 2.2.1 et 2.2.2 définissent complètement l'a.s. A associé à toute règle $\langle A \rangle$ de G .

2.2.1. Automate simple associé à une c-règle

A , de type c ou c -a.s., associé à $A \rightarrow \bigcup_{i \in [1,n]} v_i$ où $n \geq 1$ et $v_i \in \{V_a \cup V_t \cup \epsilon\}$

comporte alors 2 états et est défini par l'algorithme suivant :

```

début état 1 : (pour chaque  $v_i$  telque  $i \in [1,n]$ 
                faire const. as  $\{A, 1, v_i\}$  fait ;
                pour chaque action de  $A[1,.]$  restante non encore définie
                faire la laisser non définie en la notant ' $\square$ ' fait),
état 2 : pour chaque  $i$  telque  $i \in [1, |V_t|+1]$  faire  $A[2,i] := '\sigma'$  fait
co un état plein ' $\sigma$ ' est appelé 'état fin' co
fin

```

où la fonction $\text{const.as}(\mathcal{A}, \text{état}, v)$ qui construit $\mathcal{A}[\text{état}, \cdot]$ par rapport à v est :

```

cas aux :  $v \in V_a$  co 'v' sera alors utilisé comme nom associé à  $\langle v \rangle$  co  $\Rightarrow$ 
    (pourchaque t telque  $t \in \text{prem}(v)$ 
      faire  $\mathcal{A}[\text{état}, \text{index}(t)] := 'v'$  fait,
      si  $\text{epsi}(v)$  alors pourchaque t telque  $t \in \text{suiv}(v, A_a)$ 
        faire  $\mathcal{A}[\text{état}, \text{index}(t)] := 'e'$  fait
      fsi),
term :  $v \in V_t \Rightarrow \mathcal{A}[\text{état}, \text{index}(v)] := '\rightarrow'$ ,
eps :  $v = 'e' \Rightarrow$  pourchaque t telque  $t \in \text{suiv}(A, A_a)$ 
  faire  $\mathcal{A}[\text{état}, \text{index}(t)] := 'e'$  fait
fcas

```

Remarques

a. La construction de l'action 'e' n'exige pas que la grammaire soit LL(1) strict. Elle détermine la politique de reconnaissance qui consiste, le cas échéant, à décider une ϵ -dérivation sans détailler celle-ci par toutes ses ϵ -productions.

b. Par contre, cette construction de 'e' est plus restrictive que les algorithmes de descente récursive classique qui sont d'ailleurs utilisés pour les analyseurs LL(1).

2.2.2. Automate simple associé à une s-règle.

a. Soit $A \rightarrow \cdot v_i$ où $p > 1$ et $v_i \in \{V_a, V_t\}$. Le s-a.s. $\mathcal{A}$ correspondant comporte alors $p+1$ états définis par l'algorithme suivant

```

début fin : pourchaque i tel que  $i \in [1, |V_t|+1]$  faire  $\mathcal{A}[p+1, i] := 'o'$  fait,
  autres : pourchaque k tel que  $k \in [1, p]$ 
    faire  $\text{const.as}(\mathcal{A}, k, v_k)$  ;
    pourchaque action de  $\mathcal{A}[k, \cdot]$  restante non définie
      faire la laisser non définie en la notant '□' fait
  fait
fin

```

b. Soit le méta-axiome $M \rightarrow A_0 \#$, où A_0 est l'axiome de la grammaire. $\mathcal{M}_0$ associé comporte alors 3 états qui sont

```

début état 1 :  $\text{const.as}(\mathcal{M}_0, 1, A_0)$ ,
  état 2 : début  $\mathcal{M}_0[2, \text{index}(\#)] := 'a'$  co arrêt total co,
    pourchaque i telque  $i \in [1, |V_t|+1] \wedge i \neq \text{index}(\#)$ 
      faire  $\mathcal{M}_0[2, i] := '□'$  fait
    fin,
  état 3 : pourchaque i telque  $i \in [1, |V_t|+1]$  faire  $\mathcal{M}_0[3, i] := 'o'$  fait
fin

```

En fait pour cet a.s. $\mathcal{M}_0$, on pourrait considérer ces actions non définies '□' comme des erreurs.

2.2.3. Propriété de la construction des a.s.

Pour toute règle $\langle A \rangle$ LL(1+ ϵ), aucune action de l'a.s. $\mathcal{A}$ associé, à part peut-être 'e', ne peut être définie plus d'une fois par les algorithmes 2.2.1. et 2.2.2 ci-dessus.

Cependant, à la manière de [Aho-Johnson-Ullman 75], on peut résoudre "à la main" certain conflit pour que les a.s. puissent analyser des langages comme les instructions conditionnelles Algol 60 dont la partie sinon est optionnelle et qui ne sont LL(k), $\forall k > 0$ (une F.a.s. est donnée en appendice).

2.3. Propriétés de l'ensemble des a.s. associés à une grammaire

Soit une grammaire G LL(1+ ϵ) augmentée de $M_0 \rightarrow A_0 \#$ et sous FBNBF

a) La construction de l'ensemble des a.s. associés à l'ensemble des règles de G par les algorithmes 2.2.1 et 2.2.2 est consistante en ce sens que $\forall m \in V_t^*$, $m \# \in L(M_0)$ (resp. $m \# \notin L(M_0)$) ssi le calcul $M_0(m\#)$ se termine (resp. se bloque) en un temps fini avec un arrêt normal α sur $\#$ (resp. avec une action ' $\square$ ' anormale).

b) $M_0(m\#)$ se bloque, il se bloque après avoir accepté le plus long facteur gauche.
Cependant l'état activé qui a provoqué ce blocage ne peut avoir une signification précise

3. OPTIMISATION D'A.S. DANS LES FAMILLES D'A.S.

3.1. Famille d'automates simples.

Une famille unifiable $\mathcal{F}$ d'a.s., abrégé en F.a.s. $\langle \mathcal{F} \rangle$, est l'union d'a.s. $\mathcal{Q}$ ayant même index($\mathcal{Q}$). $\langle \mathcal{F} \rangle$ se représente alors par un vecteur des a.s. $\mathcal{Q}$.

Exemples

1. Une famille $\langle \mathcal{F} \rangle$ possible des 3 a.s. ci-dessus, est le vecteur ci-contre

$\mathcal{F}$	#	a	b	c
A	ϵ	B	ϵ	$\rightarrow$
	σ	σ	σ	σ
B	$\rightarrow$			
	ϵ	A	ϵ	A
	ϵ	A	ϵ	A
	σ	σ	σ	σ
M_0	ϵ	A	ϵ	A
	α			
	σ	σ	σ	σ

2. Soit la grammaire des notations polonaises

post-fixées suivante : $P \rightarrow PP\emptyset | i$
 $\emptyset \rightarrow + | - | * | /$

dont une grammaire équivalente LL(1+ ϵ)
 sous FBNBF est en grammaire augmentée

$M \rightarrow P\#$
 $P \rightarrow iQ$
 $Q \rightarrow R | \epsilon$
 $R \rightarrow P\emptyset Q$
 $\emptyset \rightarrow + | - | * | /$

On peut lui faire correspondre la F.a.s.
 $\langle M_0 \rangle$ ci-contre

M_0	i	+	-	*	/	#
M	P					
	σ	σ	σ	σ	σ	σ
P	Q	ϵ	ϵ	ϵ	ϵ	ϵ
	σ	σ	σ	σ	σ	σ
Q	R	ϵ	ϵ	ϵ	ϵ	ϵ
	σ	σ	σ	σ	σ	σ
R	P					
	Q	ϵ	ϵ	ϵ	ϵ	ϵ
	σ	σ	σ	σ	σ	σ
$\emptyset$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$
	σ	σ	σ	σ	σ	σ

Dans tout ce qui suit, pour la clarté de l'exposé, nous ne considérerons qu'à une grammaire LL(1+ ϵ) complète est associée une seule F.a.s. (dans le cas des grammaires "grandeur nature" Algol 60 par exemple, une F.a.s. unique n'est sûrement pas une solution acceptable).

3.2. Réduction de l'index

Soit $\langle M_0 \rangle$ ci-dessus ; intuitivement $\{+, -, *, /\}$ ont le même "rôle" en ce sens que si on les avait codés avec le même index donnant une nouvelle $\langle M_0' \rangle$, alors, $\forall m \in V_t^*$ $M_0(m\#)$ et $M_0'(m\#)$ sont "identiques" à certaines équivalences près.

Dans $\langle \mathcal{F} \rangle$, le rôle d'un terminal t , noté $\text{rôle}(t)$, en abrégé $\text{rôle}(t)$ est le vecteur coupe verticale selon t de $\langle \mathcal{F} \rangle$.
Le i ème élément de $\text{rôle}(t)$ noté $\text{rôle}_i(t)$ est donc une certaine action $A[j, \text{index}(t)]$

3.2.1. Soit $\langle \mathcal{F} \rangle$ une F.a.s. des a.s. construits à partir d'une FNBNF. Deux rôles, $\text{rôle}(t_1)$ et $\text{rôle}(t_2)$, sont semblables par $\tilde{\langle \mathcal{F} \rangle}$ ou encore $t_1 \tilde{\langle \mathcal{F} \rangle} t_2$ ssi

- a) $\text{rôle}(t_1)$ (resp. $\text{rôle}(t_2)$) contient au plus une action ' $\rightarrow$ '
b) $\forall i \in [1, \sum_{A \in \langle \mathcal{F} \rangle} |A|]$ $\text{rôle}_i(t_1) \sim \text{rôle}_i(t_2)$

3.2.2. Deux actions $A[i, j]$ et $A[i, k]$ sont équivalentes dans le contexte $A[i, .]$ selon les index k et j , $A[i, j] \sim A[i, k]$, ssi elles sont soit identiques par $\stackrel{a}{=}$, soit des noms ' X ' et ' Y ' dont X et Y vérifient

- a) $|X| = |Y| = p$
b) $\forall s \in [1, p]$ $X[s, j] \stackrel{a}{=} Y[s, k]$ et $X[s, k] \stackrel{a}{=} Y[s, j]$
c) $\forall s \in [1, p] \forall x \in [1, \text{index}(\langle \mathcal{A} \rangle)] : x \neq j \text{ et } x \neq k \quad X[s, x] \stackrel{a}{=} Y[s, x]$

3.2.3. Deux actions a_1 et a_2 sont identiques par $\stackrel{a}{=}$, $a_1 \stackrel{a}{=} a_2$, ssi a_1 et a_2 sont

- a) soit des noms d'a.s. identiques par $\stackrel{a}{=}$
b) soit la même action spécifique, soit le même nom d'a.s.
c) si l'une des actions, a_1 par exemple, est spécifique et l'autre $a_2 = 'Y'$ est calculée, alors le cas. A associé à a_1 dans $\langle \mathcal{F} \rangle$ (construction intuitive et immédiate) vérifie $A \stackrel{a.s.}{=} Y$

3.2.4. Deux a.s. sont identiques $X \stackrel{a.s.}{=} Y$ ssi $|X| = |Y| = p$ et $\forall i \in [1, p], \forall j \in [1, \text{index}(\langle \mathcal{F} \rangle)]$, $X[i, j]$ et $Y[i, j]$ vérifient simultanément les conditions b et c de 3.2.3.

On peut vérifier qu'on a bien quatre relations d'équivalence. Nous réaliserons alors une optimisation en regroupant les éléments d'une même classe : optimisation de colonnes pour les terminaux semblables, optimisation d'a.s. pour les a.s. identiques. Si on utilise $V_t / \tilde{\langle \mathcal{F} \rangle}$ comme index ($\langle \mathcal{F} \rangle$), nous aurons alors un index canonique qui est pratiquement minimal : en effet, une relation d'équivalence plus fine que celle de 3.2.1 (en étendant sa condition a) est facilement définie en termes d'équivalence de grammaires mais trop lourde à définir ici avec les a.s. Il en est de même si on veut généraliser en rendant les définitions 3.2.2, 3.2.3 et 3.2.4 mutuellement récursives : on le peut car on peut montrer que dans le cadre des F.a.s. associées aux grammaires $LL(1+\epsilon)$, ces définitions récursives "s'arrêtent" en temps fini. Cependant, on peut ramener ces cas généraux (et très rares) aux conditions ci-dessus par une des manipulations d'a.s. que nous verrons ci-après.

Exemple :
 $\langle \mathcal{M} \rangle$ est obtenue après réduction
de l'index sur $\langle \mathcal{M}_0 \rangle$ vue en 3.1
ci-dessus.

$\mathcal{M}_1$	i	+	/	#
M	P			
				α
	σ	σ	σ	
P	$\rightarrow$			
	Q	E	E	
	σ	σ	σ	
Q	R	E	E	
	σ	σ	σ	
	P			
R		$\emptyset$		
	Q	E	E	
	σ	σ	σ	
$\emptyset$		$\rightarrow$		
	σ	σ	σ	

3.3. Substitution d'a.s.

L'idée est qu'à la place d'un appel de procédure, on va y insérer sa "valeur", la "valeur" d'un a.s. étant l'a.s. sans son état fin.

3.3.1. Insertion d'un c-a.s. dans un c-a.s.

Soit un c-a.s. A tel que $A[1,i] = 'B'$ dont B est un c-a.s. Alors l'insertion du c-a.s. B dans A se réalise par appel de insertion (cas B , $A[1,.]$) où insertion (cas B , $A[e,.]$) est définie par :

pour chaque $B[1,j]$ tel que $j \in \text{index}(< \mathcal{F} >)$
faire $A[e,j] := \text{subst}(B[1,j], A[e,j])$ fait

et où la valeur de la fonction $\text{subst}(B[1,j], A[e,j])$ est complètement définie par le tableau suivant

inséré $\mathcal{I}[i,j]$	insérant $A[e,j]$	valeur de $\text{subst}(\mathcal{I}[i,j], A[e,j])$
$X(\text{resp. } \rightarrow)$	I	$X(\text{resp. } \rightarrow)$
ε	$\square(\text{resp. } \varepsilon)$	$\square(\text{resp. } \varepsilon)$
$\square$	$Y(\text{resp. } \rightarrow, \varepsilon, \square)$	$Y(\text{resp. } \rightarrow, \varepsilon, \square) \quad \forall Y \neq I$

Propriété : l'insertion réussie (i.e. la valeur de subst est toujours définie par le tableau ci-dessus) d'un c-a.s. B dans un c-a.s. A donne un c-a.s. A .

3.3.2. Insertion d'un s-a.s. dans un s-a.s.

Soit un s-a.s. A tel que $A[e,i] = 'B'$ dont B est un s-a.s.

L'insertion de B dans $A[e,.]$ est réalisé par l'algorithme suivant

début pour chaque $B[1,i]$ tel que $i \in \text{index}(< \mathcal{F} >)$
faire $A[e,i] := \text{subst}(B[1,i], A[e,i])$ fait;
 $A := A[1:e-1,.] \ @ \ A[e,.] \ @ \ B[2:|B|-1,.] \ @ \ A[e+1,|A|]$
co $\ @$ réalise la concaténation des blocs d'états en respectant
 $\text{index}(< \mathcal{F} >)$ co

fin

Propriété : l'insertion réussie d'un s-a.s. B dans un s-a.s. A donne un s-a.s. A .

3.3.3. Insertion d'un c-a.s. dans un s-a.s.

Soit un s-a.s. A tel que $A[e,i] = 'B'$ dont B est un c-a.s.

L'insertion de B à l'état e de A est le résultat de la fonction $\text{insertion}(\text{cas } B, A[e,.])$ où insertion est définie en 3.3.1.

Définitions : 1) un c-a.s. (resp. s-a.s.) est un a.s. dont tous les états, sauf l'état fin (cf. 2.2.1.), sont dits de type c-a.s. (resp. s-a.s.)

2) un $\hat{s}$ -a.s. est un a.s. dont tous les états, au nombre au moins égal à 2 non compris l'état fin, sont soit c-a.s. soit s-a.s.; si le nombre d'états c-a.s. d'un $\hat{s}$ -a.s. est nul, ce $\hat{s}$ -a.s. est un s-a.s.

Propriété : l'insertion réussie d'un c-a.s. B dans un s-a.s. A donne A de type $\hat{s}$ -a.s.

Exemples

$\langle M_2 \rangle$ est obtenue à partir de $\langle M_1 \rangle$ ci-dessus en y faisant l'insertion systématique des c-a.s. A et C qui deviennent ainsi "inaccessibles" depuis M_2 .

$\langle M_3 \rangle$ est obtenue à partir de $\langle M_2 \rangle$ en y insérant partout le s-a.s. P_2 qui devient aussi inaccessible.

On obtiendra la même F.a.s. $\langle M_3 \rangle$ en faisant d'abord l'insertion de $\mathcal{F}$ dans $\langle M_1 \rangle$ puis dans cette nouvelle F.a.s. l'insertion des c-a.s. A et C .

M_2	i	$\vdash^*$	#
M	P_2		
			α
	σ	σ	σ
P_2	$\rightarrow$		
	R_2	ε	ε
	σ	σ	σ
R_2	P_2		
		$\rightarrow$	
	R_2	ε	ε
	σ	σ	σ

M_3	i	$\vdash^*$	#
M3	$\rightarrow$		
	R_3	ε	ε
			α
	σ	σ	σ
R_3	$\rightarrow$		
	R_3	ε	ε
		$\rightarrow$	
	R_3	ε	ε
	σ	σ	σ

3.3.4. Propriétés des insertions

- a) L'insertion de l'a.s. B dans A étant en fait une insertion de B dans un état de A , le tableau suivant détermine le type de A après cette insertion.

type de B \ type état de A	s-a.s.	c-a.s.	
c-a.s.	$\hat{s}$ -a.s.	c-a.s.	
s-a.s.	s-a.s.	s-a.s.	} seulement au cas où la c-règle $\langle A \rangle$ est $A \rightarrow B$ avec $B \in V_a$
$\hat{s}$ -a.s.	$\hat{s}$ -a.s.	$\hat{s}$ -a.s.	

- b) Toutes ces insertions peuvent faire gagner de la place et font sûrement gagner du temps de calcul (au sens longueur du calcul, définie en 2.1.4).

3.4. Éclatement d'a.s.

C'est exactement l'opération duale des différentes insertions pour créer un nouvel a.s. à partir des parties communes de plusieurs a.s., ce qui va amener un éventuel gain de place mais un allongement du temps de calcul. Les algorithmes de construction sont analogues à ceux des insertions, nous ne les détaillerons pas ici.

3.5. Propriétés des manipulations d'a.s. dans les F-a.s.

Soit deux F.a.s. $\langle M \rangle$ et $\langle M' \rangle$ différentes, car non isomorphes, associées aux deux méta-axiomes M et M' de deux grammaires $LL(1+\epsilon)$.

a) $\langle M \rangle$ et $\langle M' \rangle$ différentes sont dites équivalentes si, $\forall m \# \in V_{\#}$, les calculs $M(m\#)$ et $M'(m\#)$ soit se terminent par α sous $\#$ soit se bloquent sur le même plus long facteur gauche.

b) Soit $\langle M \rangle$ associée à une grammaire augmentée $LL(1+\epsilon)$ sous FBNBF alors :
Un nombre fini d'opérations de réductions d'index (3.2), de substitution (3.3) ou d'éclatement (3.4) appliquée à $\langle M \rangle$ donne une suite de $\langle M' \rangle$ équivalentes entre-elles et à $\langle M \rangle$.

3.6. FBNBF élargie : la $\hat{F}BNBF$

Pour traduire ces nouvelles F.a.s., étendons la définition de la FBNBF pour une grammaire $G = \langle V, R \rangle$.

Toute règle de G sous $\hat{F}BNBF$ est soit de type c (au sens FBNBF) soit de type $\hat{s}$ qui est de la forme $A \rightarrow \cdot \hat{v}_i$ où $\hat{v}_i$ sont des éléments de $2^{V \cup \epsilon} \setminus \emptyset$. Ces éléments $\{v_1, \dots, v_p\}$

où $v_i \in V \cup \epsilon$, seront notés $\{v_1 | \dots | v_p\}$ si $p > 1$ ou simplement v_1 si $p = 1$ et $v_1 \neq \epsilon$

Exemple : $\langle M_3 \rangle$ ci-dessus admet la $\hat{F}BNBF$ suivante :

$$\begin{aligned} M_3 &\rightarrow i \{R_3 | \epsilon\} \# \\ R_3 &\rightarrow i \{R_3 | \epsilon\} \{+ | - | * | /\} \{R_3 | \epsilon\} \end{aligned}$$

Propriétés. • Pour toute grammaire BNF ou FBNBF, il existe une grammaire $\hat{F}BNBF$ équivalente et inversement.

- L'obtention d'une F.a.s. directement à partir d'une grammaire $\hat{F}BNBF$, et vice-versa, est possible.

4. UNE PROGRAMMATION NON RECURSIVE DES F.A.S.

Le fonctionnement des F.a.s. suggère la très classique solution de gestion de la pile d'exécution, celle des étiquettes de retour, car nous ne pensons pas que les algorithmes connus de "dérécursivation", [Arsac 77], puissent être appliqués dans ce cas. Cette programmation des F.a.s. donnent alors une famille d'automates à pile d'états F.a.p.e.

4.1. Construction d'une F.a.p.e. à partir d'une F.a.s.

Nous utilisons les étiquettes numériques, si usuelles, pour la codification des états déterminée par l'algorithme suivant :

début coder en 1-1 tous les états de $\mathcal{A}$ avec les étiquettes de $[1 : \sum |A|]$
 $\mathcal{A} \in \langle f \rangle$

co pour la commodité, choisissons une codification séquentielle du 1er état à l'état fin de chaque a.s. co ;

associer à chaque étiquette e de $\mathcal{A}[i,.]$ l'étiquette succ(e, $\mathcal{A}[i,.]$) successeur de e définie par

succ(e, $\mathcal{A}[i,.]$) := si $\mathcal{A}[i,.]$ est état fin alors e sinon e + 1 fsi
 co l'état fin est son propre successeur co ;

substituer tout appel 'A' dans les actions par l'étiquette e de $\mathcal{A}[1,.]$

fin

En se gérant une pile d'exécution, la sémantique des actions est la suivante :

'ε' ou '→' : l'exécuter puis passer à l'état suivant s, i.e.

début dépiler ; empiler s ; allera étiquette du sommet de pile fin

'σ' : revenir à l'état successeur de l'état d'appel, i.e.

début dépiler ; remplacer sommet s par son successeur ;

allera étiquette du sommet de pile fin

'σ' (resp. '□') arrêt total (resp. sortie anormale)

étiquette e : début empiler e ; allera étiquette du sommet de pile fin

Exemple :

Les F.a.p.e $\langle \mathcal{M}_4 \rangle$ associée à $\langle \mathcal{M}_5 \rangle$, et $\langle \mathcal{M}_5 \rangle$ réduite (cf. 4.2) à partir de $\langle \mathcal{M}_4 \rangle$ sont

$\mathcal{M}_4$

succ(e)	e	i	+	-	#
2	1	→			
3	2	5	ε	ε	
4	3				α
4	4	σ	σ	σ	
6	5	→			
7	6	5	ε	ε	
8	7	→			
9	8	5	ε	ε	
9	9	σ	σ	σ	

$\mathcal{M}_5$

succ(e)	e	i	+	-	#
2	1	→			
3	2	5	ε	ε	
4	3				α
4	4	σ	σ	σ	
6	5	→			
7	6	5	ε	ε	
8	7	→			
4	8	5	ε	ε	

4.2. Manipulations des états d'une F.a.p.e.

Pour les F.a.p.e., on peut reprendre toutes les manipulations d'a.s. vues en 3. Bornons-nous ici à suggérer une équivalence simple entre états d'une F.a.p.e. qui permet un gain de place (mais non temps de calcul) en détectant les mêmes facteurs droits.

Deux états e_1 et e_2 d'une F.a.p.e sont équivalents ssi

. toutes leurs actions de même index i sont équivalentes

. leur état successeur sont des états équivalents

avec comme axiomes : . les actions spécifiques identiques sont équivalentes

. les actions calculées sont équivalentes si elles sont identiques ou si leur étiquette sont des étiquettes d'états équivalents

D'autre part, en gérant soi-même la pile d'exécution, on s'est donné une très grande souplesse pour les techniques sophistiquées de reprises sur erreur en ne laissant plus non-définies les actions '□'.

4.3. Les états et a.s. itératifs.

La convention de décider qu'un état fin est son propre successeur n'est non seulement pas aberrante, pour pouvoir normaliser la représentation d'une F.a.p.e, mais encore justifiée et utile. En effet, dans ces F.a.p.e., on peut très bien envisager, car adéquat par rapport à leur fonctionnement, des états, autres que les états fins,

qui sont leur propre successeur : nous avons ainsi les états dits récurrents programmés de façon purement itérative. En fait nous avons là de la fausse récursivité car exprimable avec le produit '*' de Kleene par application du lemme d'Arden, [Tixier 67]. Comme d'autre part, la récursivité gauche des auxiliaires, qui permet d'exprimer parfaitement la sémantique de la séquence, est très utilisée dans la définition des langages classiques (i.e. autre que APL par exemple) [Nolin 66], repris par [Hua 80], admet dans la FBNBF un troisième type de règle, le type r pour itératif, de la forme $A \rightarrow m B^*$ où $m \in V^*$ et $B \in V$ dont le r-a.s. associé $\hat{A}$ est de construction intuitive : pour le dernier état de $\hat{A}$ (qui n'est plus l'état fin), on place les actions correspondantes à B ('→' si $B \in V$ et 'B' si $B \in V$) aux index(t) tel que $t \in \text{prem}(B)$, et les actions 'σ' partout ailleurs. Cet état de type r, ne peut être que le dernier état d'un a.s. : se permettre de la placer ailleurs dénaturerait le concept de l'a.s. Aussi nous pouvons parfaitement étendre l'étude des c-a.s. et s-a.s. aux r-a.s. en particulier les manipulations 3.2 à 3.4 ainsi que la construction des F.a.p.e. associées.

En revenant aux formes de grammaire, la FBNBF associée admet en plus la r-règle $A \rightarrow m^*$ où $m \in V^*$ et $\hat{A}$ est définie comme en 3.6. Dans ce cas, la FBNBF est alors une forme normalisée des "regular right part grammars" très utilisées maintenant dans l'analyse syntactique ascendante ([Madsen-Kristensen 76], [Lalonde 79],...) ou descendante ([Lewi et al. 78],...).

Exemples

1. Soit la grammaire des notations polonaises vue en 3.1. En appliquant le lemme d'Arden, les règles FBNBF de cette grammaire sont $P \rightarrow i Q^*$ et $Q \rightarrow P\{+|-|*|/\}$ qui admettent la F.a.s. $\langle M_6 \rangle$ et la F.a.p.e. $\langle M_7 \rangle$ suivante

M_6	i	+	-	*	/	#
	P					
M_6						α
	σ	σ	σ			
P	→					
	Q	σ	σ			
	P					
Q		→				
	σ	σ	σ			

	succ(e)	e	i	+	-	*	/	#
	2	1	4					
	3	2						α
M_7	3	3	σ	σ	σ			
	5	4	→					
	5	5	6	σ	σ			
	7	6	4					
	3	7				→		

2. Sur le morceau de grammaire (où '-' et '+' n'ayant qu'un rôle purement syntaxique),

$$E \rightarrow E+T \mid E-T \mid -T \mid +T \mid T$$

l'application du lemme d'Arden et la mise sous FBNBF donne

$$E \rightarrow \{T_1 \mid T_2 \mid T\} \{T_1 \mid T_2\}^* \quad T_1 \rightarrow -T \quad T_2 \rightarrow +T$$

En supposant que $\text{prem}(T) = \{(, i\}$ on a la F.a.s. $\langle \mathcal{E} \rangle$ ci-dessous où l'index 'autre' correspond à $\{*, /, \#, \dots\}$. En étendant légèrement la définition 3.2.4. de = aux a.s. non définis dans la même F.a.s., on peut déduire l'équivalence de '+' et '-' (resp. de 'i' et '(') ainsi que celle de $\mathcal{E}_1$ et $\mathcal{E}_2$ dans $\langle \mathcal{E} \rangle$. $\langle \mathcal{E}_1 \rangle$ est alors équivalente à $\langle \mathcal{E} \rangle$ et dont $\langle \mathcal{E}_2 \rangle$ en est la F.a.p.e. correspondante

$\mathcal{E}$	+	-	i	(	autre
E	T ₂	T ₁	T	T	
	T ₂	T ₁	σ	σ	σ
T ₁		→			
	σ	σ	σ	σ	σ
T ₂	→				
	σ	σ	σ	σ	σ

$\mathcal{E}_1$	+	i	autre
E	T ₁	T	
	T ₁	σ	σ
T ₁	→		
	σ	σ	σ

	succ(e)	e	+	i	autre
	2	1	3	T	
	2	2	3	σ	σ
$\mathcal{E}_2$	4	3	→		
	5	4		T	
	5	5	σ	σ	σ

(au niveau de la F.a.p.e., on a donc le petit problème l'a.s. non défini dans une même famille)

5. CONCLUSIONS

Le besoin d'un exposé écrit fait qu'on a dû formaliser la construction et la manipulation des a.s. En fait, lorsqu'on a pu établir qu'on peut manipuler les règles de BNF avec les trois opérations régulières, ces construction et manipulation deviennent "évidentes".

Comme la possibilité d'inclusion des actions sémantiques est naturelle et intuitive dans ces analyseurs descendants, ceux-ci sont des outils de choix pour la programmation des traducteurs (compilateurs, interpréteurs) de langages. L'intérêt supplémentaire des F.a.s. et F.a.p.e. provient du fait que leur construction est systématique et leurs manipulations sont formelles.

6. REMERCIEMENTS

Nous tenons à exprimer notre reconnaissance à L. Nolin de nous avoir apporté un constant soutien. Nos vifs remerciements aussi à tous ceux, nombreux, qui ont aidé à la mise en forme et encouragé à la publication de ce travail: les remarques très constructives de J.F. Perrot nous sont particulièrement déterminantes.

REFERENCES

- Aho A.V. - Johnson S.C. - Ullman J.D. 'Deterministic parsing of ambiguous grammars', CACM, 1975, pp. 441-452.
- Aho A.V. - Ullman J.D. 'The theory of parsing, translation and compiling', vol. I (1972) & vol II (1973), Prentice Hall publ.
- Arsac J. 'La construction de programmes structurés', Dunod publ., 1977.
- Bauer F.L. 'Historical remarks on compiler construction, in LNCS n° 21. 1974, pp 603-621, Springer Verlag publ.
- Bordier J. 'Méthodes pour la mise au point de grammaires LL(1)', thèse de 3ème cycle, Université de Grenoble, 1971.
- Conway M.E. 'Design of a separable transition diagram compiler' CACM 1963, pp 396-408.
- Earley J. 'An efficient context-free parsing algorithm', CACM, 1970, pp. 94-102.
- Friede D. 'Transition diagrams and strict deterministic grammars', in LNCS n° 67, 1978, pp 113-123, Springer Verlag publ.
- Gerbier A. 'Mes premières constructions de programmes', LNCS n° 55, 1977 Springer Verlag publ.
- Griffiths M. 'LL(1) grammars and analysers', in LNCS n° 21, 1974. pp. 57-84, Springer Verlag publ.
- Gries D. 'The use of transition matrices in compiling', CACM 1968, pp. 26-34
- Gries D. 'Compiler construction for digital computers', 1971, J. Wiley & Sons publ.
- Hua-T-H. 'Pratique des grammaires factorisées', notes de l'Institut de Programmation, Paris, 1973.
- Hua-T-H. 'Contribution à l'étude des analyseurs déterministes descendants', Institut de Programmation, Paris, 1980, à paraître.
- Keller R.F. 'On selective and repetitive constructs', Sigplan Notices, Nov. 79, pp. 60-63.
- Lalonde W.R. 'Constructing LR parsers for regular right part grammars' Acta informatica, 1979, pp. 177-193.

- Lewi J. - de Vlamink K. - Huens J. - Huybretchs M. 'The ELL(1) parser generator and the error recovery mechanism', Acta Informatica, 1978, pp. 209-228.
- Lomet D.B. 'A formalization of transition diagram systems', JACM, 1963, pp. 235-257.
- Madsen O.L. - Kristensen B.B. 'LR parsing of extended context-free grammars', Acta Informatica, 1976, pp 61-74.
- Nolin L. 'Automates simples et langages factorisés', note de l'Institut Blaise Pascal, CNRS, Paris, 1966.
- Pleban U.F. 'The use of transition matrices in a recursive descent compiler', Proc. of Sigplan Symp. on Compiler Construction, Sigplan Notices, Aug. 1979, pp. 144-151.
- Tixier V. 'Recursive functions of regular expressions in language analysis', Tech. rep. n° 58, Computer Science Dept., Stanford University, 1967.

APPENDICE I

Analyse déterministe et descendante d'une grammaire ambiguë

Soit la grammaire ambiguë des instructions conditionnelles ci-dessous qui n'est pas LL(k). A la manière de [Aho-Johnson-Ullman 75], dans le conflit entre 'ε' ou 'T' nous privilégions 'T', autrement dit choisir la politique du plus long facteur gauche et ne décider de "sortir par ε" que lorsqu'on ne peut plus continuer. Cette politique est d'ailleurs conforme à la politique des pseudo-blocs qui consiste à associer la première "parenthèse fermante" (')', 'sinon',...) à la dernière "parenthèse ouvrante" ('(', 'alors,...) "non associée" et si besoin est, de "compléter" par autant de parenthèses fermantes que de parenthèses ouvrantes non associées.

$$\left. \begin{array}{l} M \rightarrow C \# \\ C \rightarrow cIS \\ I \rightarrow i|C \\ S \rightarrow \varepsilon|T \\ T \rightarrow sI \end{array} \right\} \rightarrow \hat{FNBNF}: \left\{ \begin{array}{l} M \rightarrow C \# \\ C \rightarrow c\{C|i\}\{T|\varepsilon\} \\ T \rightarrow s\{C|i\} \end{array} \right.$$

avec $V_t = \{c \text{ sialors, s sinon, i instruction non conditionnelle}\}$

$V_a = \{C \text{ instruction conditionnelle, I instruction, S partie sinon}\}$

	C	i	A	#
M	C			α
	σ	σ	σ	σ
C	C	→		
			T	ε
	σ	σ	σ	σ
			→	
T	C	→		
	σ	σ	σ	σ

succ(e)	e	C	i	A	#
2	1	4			
3	2				α
3	3	σ	σ	σ	σ
5	4	→			
6	5	4	→		
3	6			8	ε
9	8			→	
3	9	4	→		

APPENDICE II

Exemples de fonctionnement des F.a.s. et F.a.p.e.

Pour les F.a.s., nous simulons les appels de procédure par une pile des états à l'exécution où la partie de pile (sens $\rightarrow$) non modifiée n'est pas représentée et où A_i dénote le i ème état de Q . Une convention analogue est prise pour le calcul d'une F.a.p.e.

En reprenant les différentes $\mathcal{M}_{60}$ (cf. 3.1) et $\mathcal{M}_{65}$ (cf. 4.1) nous avons leur calcul suivant sur les mots $ii+i\#$ et $ii+i\#$

$\mathcal{M}_{60}(ii+i\#)$	$\mathcal{M}_{65}(ii+i\#)$
i $M_1 P_1$ ii $P_2 Q_1 R_1 P_1$ $ii+$ P_2 P_3 $R_2 \emptyset_1$ $ii+i$ $\emptyset_2$ $R_3 Q_1 R_1 P_1$ $ii+i*$ P_2 P_3 $R_2 \emptyset_1$ $ii+i*\#$ $\emptyset_2$ R_3 R_4 Q_2 R_4 Q_2 P_3 $M_2 \Rightarrow 'a'$ arrêt normal	i 1 ii 2,5 $ii+$ 6 7 $ii+i$ 8,5 $ii+i*$ 6 7 $ii+i*\#$ 8 4 4 3 $\Rightarrow 'a'$ arrêt normal
$\mathcal{M}_{60}(ii+i\#)$	$\mathcal{M}_{65}(ii+i\#)$
i $M_1 P_1$ ii $P_2 Q_1 R_1 P_1$ $ii+$ P_2 P_3 $R_2 \emptyset_1$ $ii+i$ $\emptyset_2$ R_3 R_4 Q_2 P_3 $M_2 \Rightarrow 'a'$ non défini	i 1 ii 2,5 $ii+$ 6 7 $ii+i$ 8 4 3 $\Rightarrow 'a'$ non défini