

# Compilation M1

Félix Jamet, Mica Ménard

Avril 2018

## Contents

<b>1</b>	<b>Types des grammaires</b>	<b>2</b>
<b>2</b>	<b>Grammaires LL(k)</b>	<b>2</b>
2.1	First(N) . . . . .	2
2.2	Follow(N) . . . . .	2
2.3	Grammaire LL(1) . . . . .	3
<b>3</b>	<b>Projet compilo</b>	<b>3</b>
3.1	Définitions . . . . .	3
3.2	Schémas . . . . .	3
3.3	Processus divers . . . . .	4
3.3.1	Scan $G_0$ . . . . .	4
3.3.2	Scan GPL . . . . .	4
3.3.3	Action $G_0$ . . . . .	4
3.4	Construction de la grammaire $G_0$ . . . . .	4
3.4.1	Notation B.N.F. . . . .	4
3.4.2	Règle 1 . . . . .	5
3.4.3	Règle 2 . . . . .	5
3.4.4	Règle 3 . . . . .	5
3.4.5	Règle 4 . . . . .	5
3.4.6	Règle 5 . . . . .	5
3.5	Structure de données . . . . .	5
3.6	Construction des 5 Arbres . . . . .	6
3.6.1	Fonctions Gen* . . . . .	6
3.6.2	Arbres . . . . .	7
3.7	Scan $G_0$ . . . . .	8
3.8	Action $G_0$ . . . . .	9
3.9	Exemple . . . . .	10
3.9.1	Pile . . . . .	10
3.9.2	Dictionnaires . . . . .	10
3.9.3	Compilation . . . . .	10
3.9.4	Arbre GPL . . . . .	11
<b>4</b>	<b>Tables S.R.</b>	<b>12</b>
4.1	Algorithme Table Analyse L.R. . . . .	12
4.2	Astuces de construction de la table SR . . . . .	12
4.3	Génération automatique de la table SR . . . . .	12
4.3.1	Opérateurs $\doteq$ , $>$ , et $<$ . . . . .	12

4.4	Exemple de génération de table S.R. . . . .	13
4.4.1	GPL . . . . .	13
4.4.2	Fenêtre . . . . .	13
4.4.3	Questions . . . . .	13
4.4.4	Dérivation . . . . .	13
4.4.5	Arbre et poignées . . . . .	13
<b>5</b>	<b>Génération de code</b>	<b>13</b>
5.1	Mnémoniques associés à un accumulateur . . . . .	13
5.1.1	Exemples . . . . .	13
5.1.2	Opérations . . . . .	15

# 1 Types des grammaires

0 grammaires de type C

1 grammaires contextuelles (CS)  $\gamma \rightarrow \beta$  avec  $\|\gamma\| \leq \|\beta\|$

2 grammaires non contextuelles (CF)  $A \rightarrow B$  avec  $A \in V_N, B \in V^+$

3 grammaires régulières

$$\begin{cases} A \rightarrow aB \\ A \rightarrow a \end{cases} \quad \text{ou} \quad \begin{cases} A \rightarrow Ba \\ A \rightarrow a \end{cases}$$

Le langage  $L$  généré par une grammaire  $G$  est tel que:

$$L(G) = \{x \in V_T^* / S \xRightarrow{*} x\}$$

$S$  étant ici le symbole de départ de la grammaire  $G$  (*start symbol*).

l'intersection de deux langages de type  $x$  n'est pas forcément de type  $x$ .

## 2 Grammaires LL(k)

$k$  est une mesure de l'ambiguïté. Représente le nombre de caractères qu'il est nécessaire de regarder pour déterminer quelle règle utiliser. Bien entendu, les règles LL(1) sont préférables.

### 2.1 First(N)

- Si  $N \rightarrow A \dots$  alors  $First(N) = First(A)$
- Si  $N \rightarrow c \dots$  alors  $First(N) = \{c\}$
- Si  $N \rightarrow A.B \dots$  et si  $A \xRightarrow{*} \epsilon$  alors  $First(N) = First(B)$

Avec " $\xRightarrow{*}$ " signifiant "se derive en".

Il ne s'agit pas d'appliquer une règle a chaque fois, mais plutot d'appliquer toutes les règles possibles.

### 2.2 Follow(N)

- Si  $A \rightarrow \dots Nc \dots$  alors  $Follow(N) = \{c\}$
- Si  $A \rightarrow \dots NB \dots$  alors  $Follow(N) = First(B)$

- Si  $A \rightarrow N \dots$  alors  $Follow(N) = Follow(A)$

Concernant la dernière règle, hippolyte a noté: - Si  $A \rightarrow \dots N$  alors  $Follow(N) = Follow(A)$

À déterminer.

## 2.3 Grammaire LL(1)

- si  $A \rightarrow \alpha_1 / \alpha_2 / \dots / \alpha_n$  alors

$$First(\alpha_i) \cap First(\alpha_j) = \emptyset, \forall i \neq j$$

- si  $A \Rightarrow \epsilon$  on doit avoir

$$First(A) \cap Follow(A) = \emptyset$$

Si une règle ne possède qu'une dérivation, la règle 1 ne s'applique pas. Si une règle ne possède pas de suivant, la règle 2 ne s'applique pas.

## 3 Projet compilé

### 3.1 Définitions

**GPL** Grammaire Petit Langage

**Scanner** analyseur lexical, découpe du texte en unités syntaxiquement correctes (tokens)

**Parseur** analyse syntaxique, s'assure que les tokens sont syntaxiquement corrects

### 3.2 Schémas

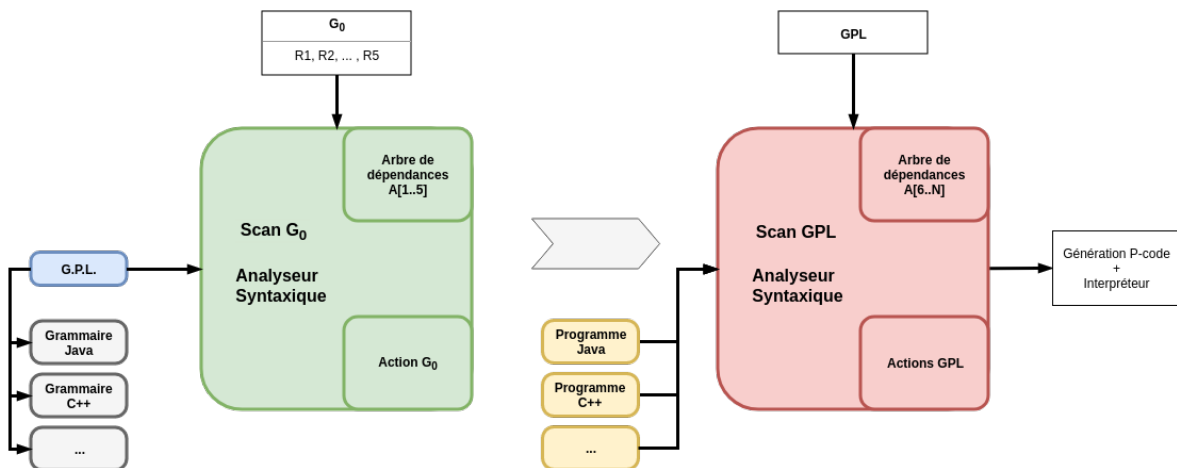


Figure 1: Projet Compilé

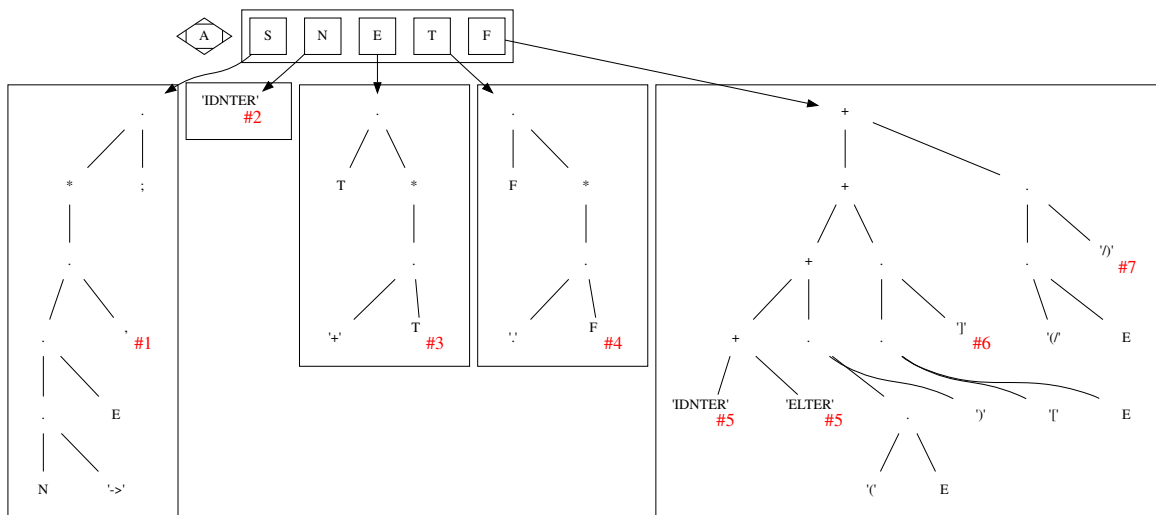


Figure 2: Arbres de dépendances  $G_0$

### 3.3 Processus divers

#### 3.3.1 Scan $G_0$

Scanne les

- éléments terminaux
- éléments non-terminaux

#### 3.3.2 Scan GPL

Scanne les

- identificateurs
- nombres entiers
- symboles (>, #, [, etc.)

#### 3.3.3 Action $G_0$

Construit l'arbre GPL

### 3.4 Construction de la grammaire $G_0$

#### 3.4.1 Notation B.N.F.

- $::= \iff \rightarrow$
- $[X] \iff X.X.X...X(n \text{ fois}), n \geq 0$
- $(/X/) \iff X \text{ ou Vide}$
- $/ \iff +$
- $concat \iff .$

- 'X' correspond à un élément terminal

### 3.4.2 Règle 1

$$S \rightarrow [N.' \rightarrow '.E.', ']'.';',$$

Une grammaire est forcément composée de plusieurs règles, séparées par des ',' et terminée par un ';'.

### 3.4.3 Règle 2

$$N \rightarrow 'IDNTER',$$

'IDNTER' signifie identificateur non terminal.

### 3.4.4 Règle 3

$$E \rightarrow T.[ ' + '.T],$$

E est une expression qui peut être un terme ou un autre.

### 3.4.5 Règle 4

$$T \rightarrow F.[ ' .F],$$

Un terme T peut être composé d'un seul facteur F ou de facteurs concaténés.

### 3.4.6 Règle 5

$$F \rightarrow 'IDNTER' + 'ELTER' + '('.E.')' + '['.E.']* + '(/.E./)';$$

## 3.5 Structure de données

Syntaxe maison...

```
Type Atomtype = (Terminal, Non-Terminal);
Operation = (Conc, Union, Star, UN, Atom); ##Atom = {IDNTER, ELTER}
PTR = ↑Node

Node = Enregistrement
  case operation of
    Conc: (left, right : PTR);
    Union: (left, right : PTR);
    Star: (stare: PTR);
    UN: (UNE : PTR);
    ATOM: (COD, Act : int ; AType: Atomtype);
  EndEnregistrement

A: Array [1..5] of PTR;
```

## 3.6 Construction des 5 Arbres

### 3.6.1 Fonctions Gen\*

```
Fonction GenConc(P1, P2 : PTR) : PTR;  
  var P : PTR;  
debut  
  New(P, conc);  
  P↑.left := P1;  
  P↑.right := P2;  
  P↑.class := conc;  
  GenConc := P;  
fin
```

```
Fonction GenUnion(P1, P2 : PTR) : PTR;  
  var P : PTR;  
  début  
    New(P, union);  
    P↑.left := P1;  
    P↑.right := P2;  
    P↑.class := union;  
    GenUnion := P;  
  fin
```

```
Fonction GenStar(P1 : PTR) : PTR; ##0 ou n fois  
  var P:PTR;  
  début  
    New(P, star);  
    P↑.stare := P1;  
    P↑.class := star;  
    GenStar := P;  
  fin
```

```
Fonction GenUn(P1 : PTR) : PTR; ##0 ou une fois  
  var P:PTR;  
  début  
    New(P, un);  
    P↑.une := P1;  
    P↑.class := un;  
    GenUn := P;  
  fin
```

```
Fonction GenAtom(COD, Act : int, AType : Atomtype) : PTR  
  var P:PTR;  
  début  
    New(P, atom);  
    P↑.COD := COD;  
    P↑.Act := Act;  
    P↑.AType := AType;  
    GenAtom := P;  
  fin
```

### 3.6.2 Arbres

1. S

```
A[S] :=
  GenConc(
    GenStar(
      GenConc(
        GenConc(
          GenConc(GenAtom('N', 0, NonTerminal),
            GenAtom('->', 0, Terminal)
          ),
          GenAtom('E', 0, NonTerminal)
        ),
        GenAtom(',', 1, Terminal)
      ),
      GenAtom(';', 0, Terminal)
    );
```

2. N

*##Ajouts de ma part, je ne suis pas sûr des résultats :*

```
A[N] := GenAtom('IDNTER', 2, Terminal);
```

3. E

```
A[E] := GenConc(
  GenAtom('T', 0, NonTerminal),
  GenStar(
    GenConc(
      GenAtom('+', 0, Terminal),
      GenAtom('T', 3, Terminal)
    )
  )
);
```

4. T

```
A[T] := GenConc(
  GenAtom('F', 0, NonTerminal),
  GenStar(
    GenConc(
      GenAtom('.', 0, Terminal),
      GenAtom('F', 4, Terminal)
    )
  )
);
```

5. F

```
A[F] := GenUnion(
  GenUnion(
    GenUnion(
      GenUnion(
        GenAtom('IDNTER', 5, Terminal),
```

```

        GenAtom('ELTER', 5, Terminal)
    ),
    GenConc(
        GenConc(
            GenAtom('(', 0, Terminal),
            GenAtom('E', 0, NonTerminal)
        ),
        GenAtom(')', 0, Terminal)
    )
),
GenConc(
    GenConc(
        GenAtom('[', 0, Terminal),
        GenAtom('E', 0, NonTerminal)
    ),
    GenAtom(']', 6, Terminal)
)
),
GenConc(
    GenConc(
        GenAtom('/', 0, Terminal),
        GenAtom('E', 0, NonTerminal)
    ),
    GenAtom('/', 7, Terminal)
)
)

```

### 3.7 Scan $G_0$

```

fonction Analyse(P : PTR) : booléen
début
    case P↑.class of
        Conc: if Analyse(P↑.left) then Analyse := true
              else Analyse := Analyse(P↑.right);
        Union: if Analyse(P↑.left) then Analyse := true
              else Analyse := Analyse(P↑.right);
        Star: Analyse := true;
              while Analyse(P↑.stare) do;
        Un: Analyse := true;
           if Analyse(P↑.une) then;
        Atom: case P↑.Atype of
            Terminal: if P↑.cod = code then #cod = code ASCII
                      début
                        Analyse := true;
                        if P↑.act != 0 then GO-action(P↑.act)
                        scanG0;
                      fin
                        else Analyse := false;
            Non-Terminal: if Analyse(A[P↑.cod]) then
                          début

```



```

        if P↑.act != 0 then G0-action(P↑.act);
        Analyse := true;
    fin
else Analyse := false;

fin

Main() #vérifie si une grammaire est correcte
{
    scan;
    if Analyse(A[s]) then write('OK');
}

```

### 3.8 Action $G_0$

De quoi a-t-on besoin ?

- Deux dictionnaires : *DicoT*, *DicoNT*
- Tableau *pile*[*I*] : Tableau de pointeurs

Remarque : les nombres du case correspondent aux actions associées aux numéros inscrits dans les arbres.

```

Procédure Action G0(Act : int);
var T1, T2 : PTR;
début
    case Act of
    1: Dépiler(T1);
        Dépiler(T2);
        A[T2↑.cod + 5] := T1; ##Arbres GPL commencent à 6
    2: Empiler(GenAtom(Recherche(DicoT), Action, CAType)) ##donne la
        ##partie gauche d'une règle
        ##Recherche() stocke le token si non stocké dans dico
    3: Dépiler(T1);
        Dépiler(T2);
        Empiler(GenUnion(T2,T1))
    4: Dépiler(T1);
        Dépiler(T2);
        Empiler(GenConc(T2,T1))
    5: if CAType = Terminal then
        Empiler(GenAtom(Recherche(DicoT), Action, Terminal))
        else
        Empiler(GenAtom(Recherche(DicoNT), Action, Terminal))
    6: Dépiler(T1);
        Empiler(GenStar(T1));
    7: Dépiler(T1);
        Empiler(GenUn(T1));

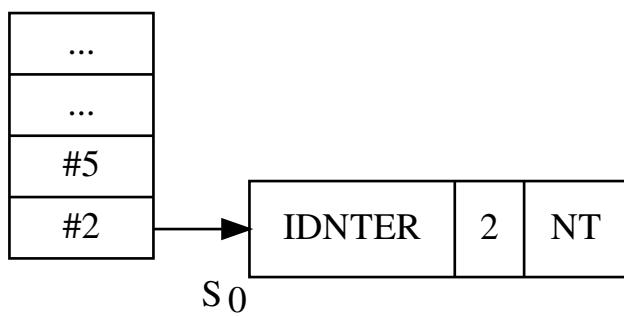
Pile : Array[1..50] : PTR;
DicoT, DicoNT: Dico;
Dico : Array[1..50] : String[10];

```

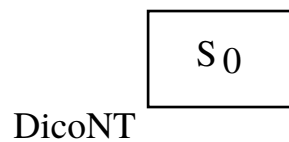
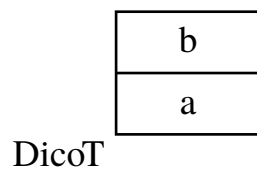
### 3.9 Exemple

GPL :  $S_0 \rightarrow ['a'].b'$ ; Regex :  $a^nb$

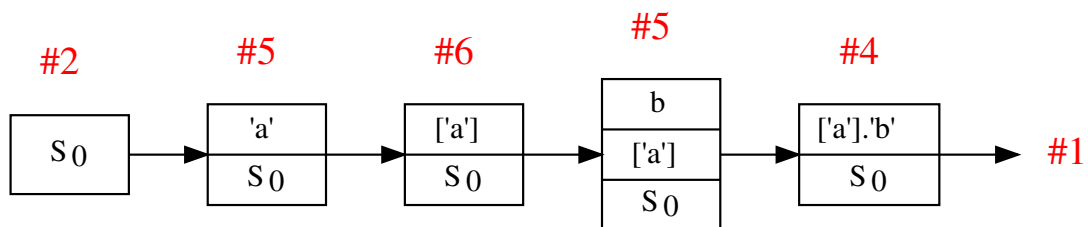
#### 3.9.1 Pile



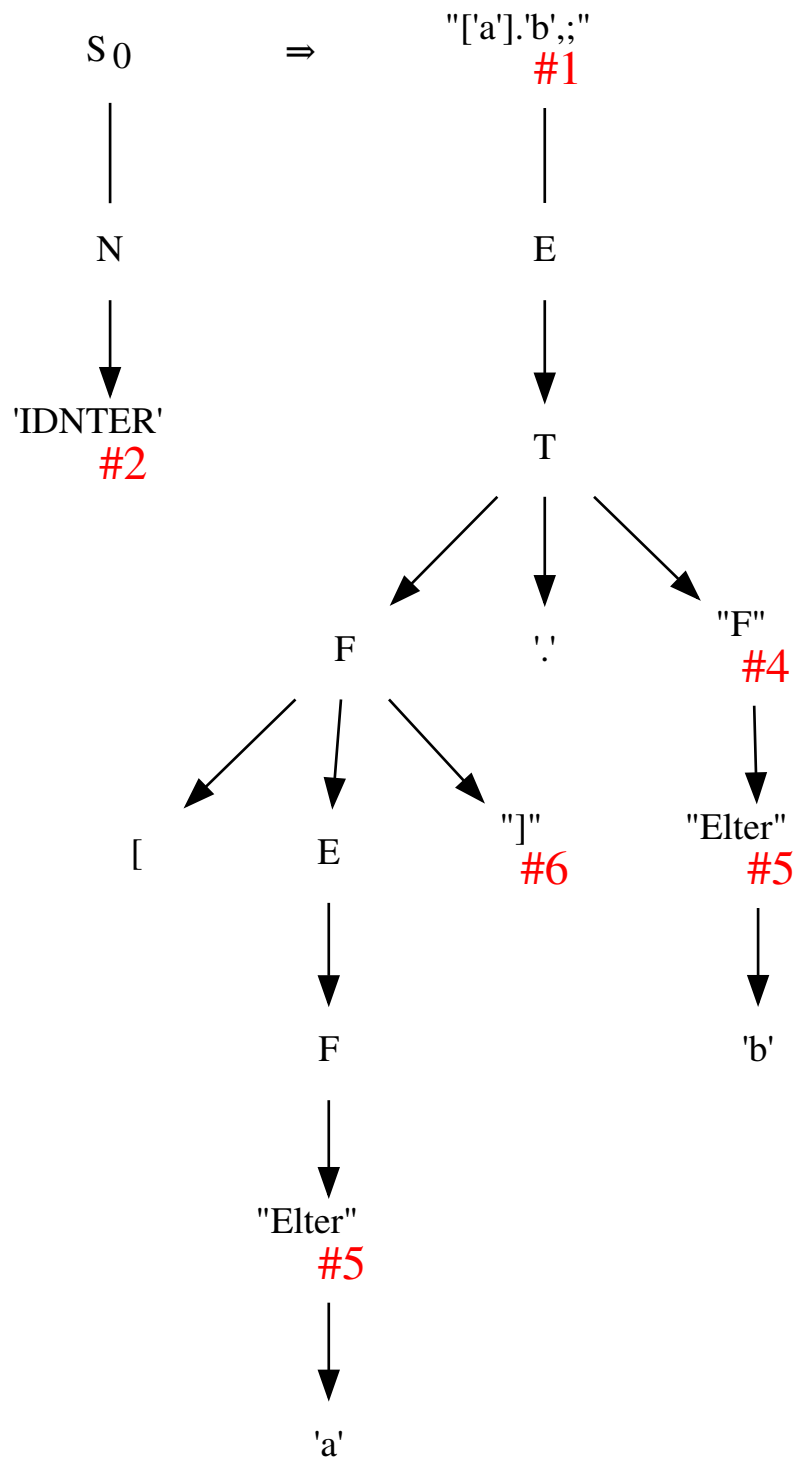
#### 3.9.2 Dictionnaires



#### 3.9.3 Compilation



### 3.9.4 Arbre GPL



## 4 Tables S.R.

### 4.1 Algorithme Table Analyse L.R.

**Shift** Empiler le caractère; scan;

**Reduce** Remplacer la partie droite au sommet de la pile par la partie gauche ( $A \rightarrow a$ )

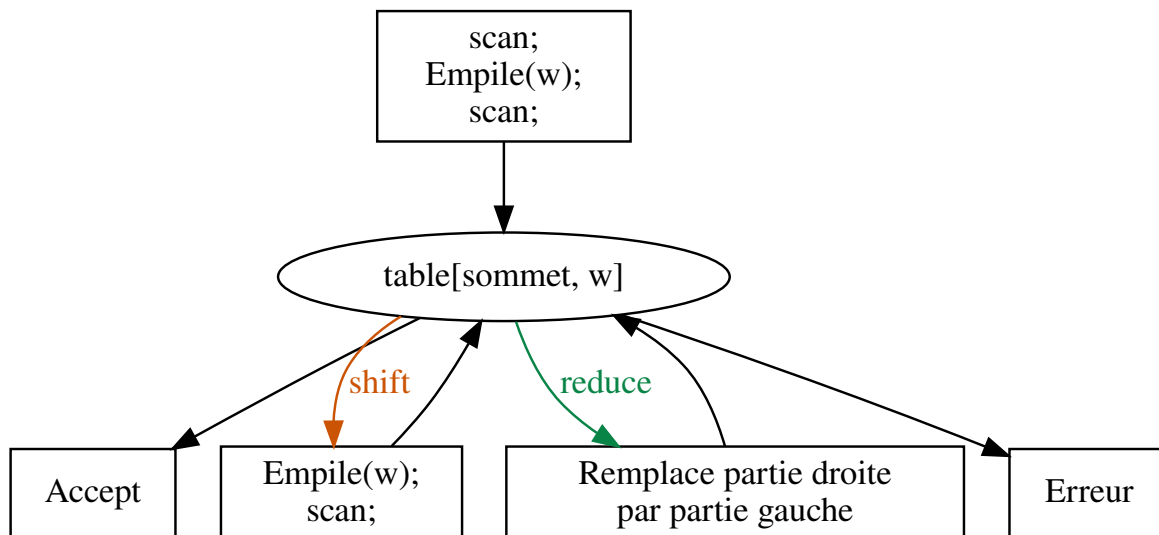


Figure 3: Algorithme Table Analyse L.R.

### 4.2 Astuces de construction de la table SR

- Commencer par remplir l'accept (en regardant la première règle) et les erreurs.
- On reduce quand une poignée apparaît dans la pile.
- Si un mot n'est pas en fin de règle, sa ligne ne comprendra pas de reduce.

### 4.3 Génération automatique de la table SR

#### 4.3.1 Opérateurs $\doteq$ , $\succ$ , et $\prec$

- $X \doteq Y$  si  

$$A \rightarrow \dots X.Y \dots \in \mathcal{P}$$
- $X \prec Y$  si  

$$A \rightarrow \dots X.Q \dots \in \mathcal{P}$$
  
 et  $Q \overset{*}{\Rightarrow} Y$
- $X \succ Y$  si  

$$A \doteq Y$$
  
 et  $A \overset{*}{\Rightarrow} X$

On peut remplir le tableau SR à partir des relations  $\doteq$ ,  $>$  et  $<$  :

- (ligne  $\doteq$  colonne) et (ligne  $<$  colonne) se traduisent en (ligne Shift colonne)
- (ligne  $>$  colonne) se traduit en (ligne Reduce colonne)

## 4.4 Exemple de génération de table S.R.

### 4.4.1 GPL

$S \rightarrow E\$$

$E \rightarrow E + a$

$E \rightarrow a$

Type 2 car deux terminaux ('+' et 'a')

### 4.4.2 Fenêtre

$a + a + a + a\$$

### 4.4.3 Questions

1. Poignées ?
2. Configuration de la pile
3. Table S.R.

### 4.4.4 Dérivation

$a + a + a + a\$ \rightarrow E + a + a + a\$ \rightarrow E + a + a\$ \rightarrow E + a\$ \rightarrow E\$ \rightarrow S$

### 4.4.5 Arbre et poignées

## 5 Génération de code

### 5.1 Mnémoniques associés à un accumulateur

Load A  
STO A  
ADD A  
SUB A

#### 5.1.1 Exemples

$y := a + b * c$

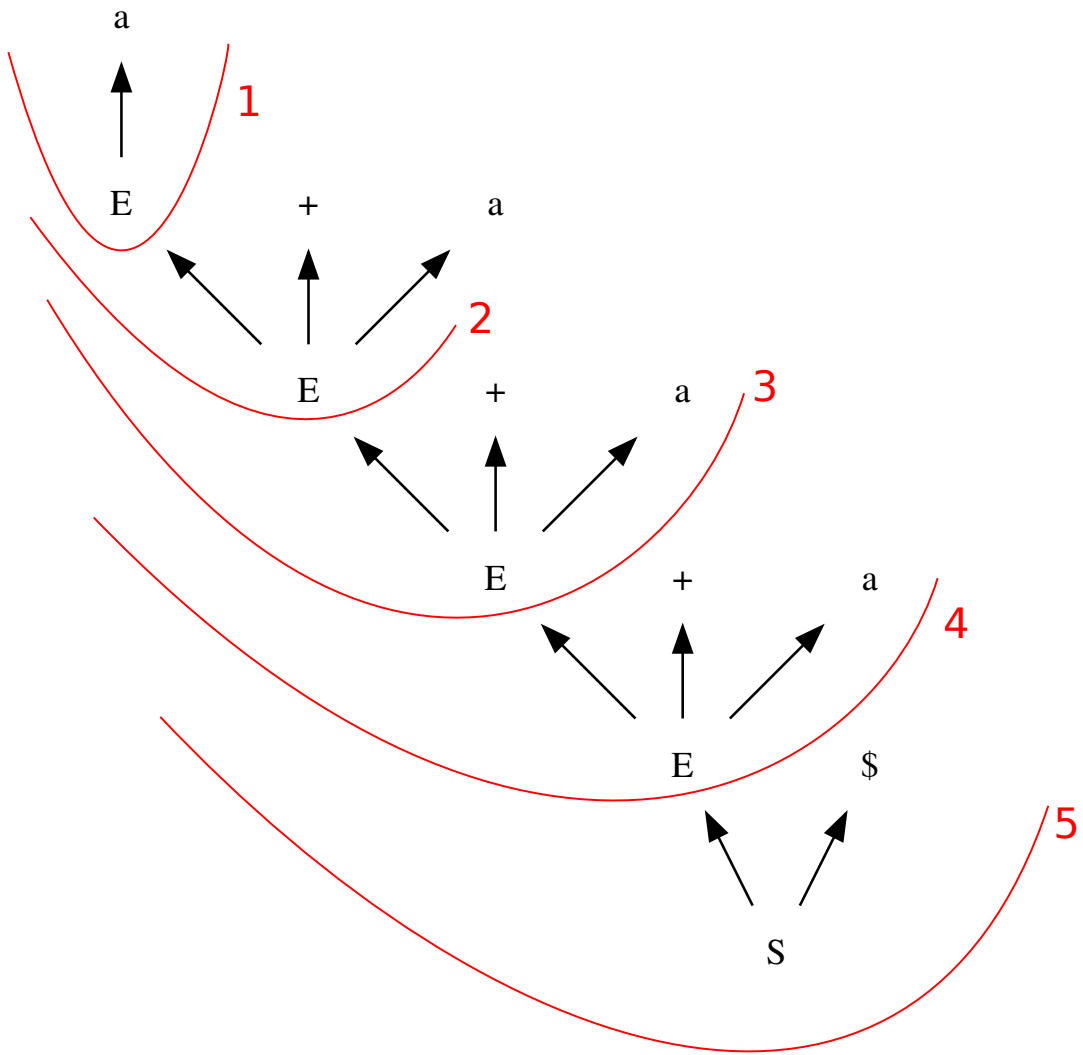


Figure 4: Arbre et poignées

```

Load b
MULT c
ADD a
STO y

```

$$a := (a + b) * c$$

```

Load a
ADD b
MULT c
STO a

```

$$a := c / (a + b)$$

```

Load a
ADD b
STO d
Load c
DIV d

```

## Automatisation du processus  
 Utilisation de la notation post-fixée.

$$c / (a + b) \rightarrow cab + /$$

### Exemple

$$(a + b) / (c + d)$$

Notation post-fixée :  $ab + cd + /$

```

Load c
ADD d
STO x
Load a
ADD b
DIV x

```

## Génération de code avec plusieurs registres

### 5.1.2 Opérations

$$Mov \left\{ \begin{matrix} R \\ A \end{matrix} \right\}, \left\{ \begin{matrix} R \\ A \end{matrix} \right\}$$

avec  $R$  : registres et  $A$  : addresses. La première partie est la source et la deuxième la destination.

Mov A, R : prendre le contenu de  $A$  et le mettre dans  $R$ .

$$Op \left\{ \begin{matrix} R \\ A \end{matrix} \right\}, R$$

Exemples :

ADD R1, R2

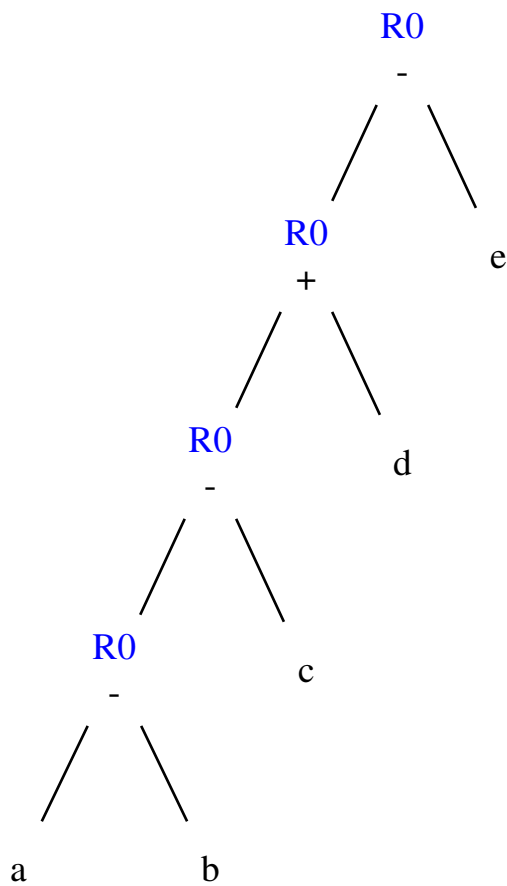
$$\Leftrightarrow R2 \leftarrow R2 + R1$$

DIV b, R1

$$\Leftrightarrow R1 \leftarrow R1/b$$

### Combien a-t-on besoin de registres ?

$$[((a - b) - c) + d] - e \rightarrow ab - c - d + e -$$





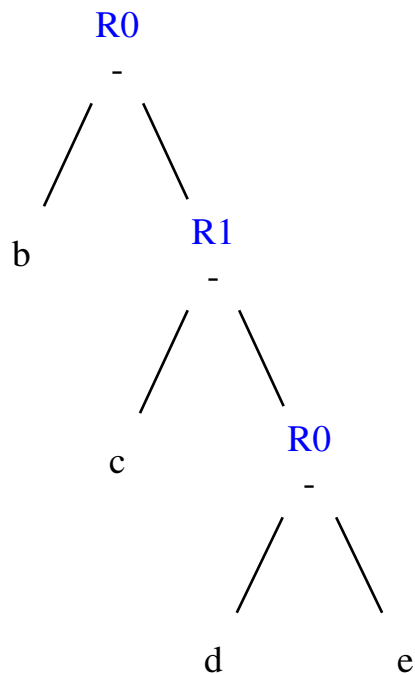
```

MOV a, R0
SUB b, R0
SUB c, R0
ADD d, R0
SUB e, R0

```

On a donc besoin d'un seul registre.

$$b - (c - (d - e)) \rightarrow bcde ---$$



```

MOV d, R0
SUB e, R0
MOV c, R1
SUB R0, R1
MOV b, R0
SUB R1, R0

```

On a besoin de deux registres.

### Règles générales

