

# 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) . . . . .                               | 3         |
| 2.2      | Follow(N) . . . . .                              | 3         |
| 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 . . . . .                       | 3         |
| 3.3.1    | Scan $G_0$ . . . . .                             | 3         |
| 3.3.2    | Scan GPL . . . . .                               | 5         |
| 3.3.3    | Action $G_0$ . . . . .                           | 5         |
| 3.4      | Construction de la grammaire $G_0$ . . . . .     | 5         |
| 3.4.1    | Notation B.N.F. . . . .                          | 5         |
| 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 . . . . .                   | 6         |
| 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 . . . . .                            | 11        |
| 3.9.4    | Arbre GPL . . . . .                              | 11        |
| <b>4</b> | <b>Tables S.R.</b>                               | <b>11</b> |
| 4.1      | Algorithme Table Analyse L.R. . . . .            | 11        |
| 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>14</b> |
| 5.1      | Mnémoniques associés à un accumulateur . . . . .       | 14        |
| 5.1.1    | Exemples . . . . .                                     | 14        |
| 5.2      | Automatisation du processus . . . . .                  | 14        |
| 5.2.1    | Exemple . . . . .                                      | 14        |
| 5.3      | Génération de code avec plusieurs registres . . . . .  | 15        |
| 5.3.1    | Opérations . . . . .                                   | 15        |
| 5.3.2    | Combien a-t-on besoin de registres ? . . . . .         | 15        |
| 5.3.3    | Règles générales . . . . .                             | 16        |
| 5.4      | P-code . . . . .                                       | 17        |
| 5.4.1    | Opérations . . . . .                                   | 17        |
| 5.4.2    | Exemples . . . . .                                     | 18        |
| 5.4.3    | Procédures <i>Exec</i> et <i>Interpréter</i> . . . . . | 18        |
| <b>6</b> | <b>Annexes</b>   | <b>20</b> |
| 6.1      | Exercices de TD . . . . .                              | 20        |
| 6.1.1    | Détermination d'un automate . . . . .                  | 20        |

## 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 compilo

## 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

## 3.3 Processus divers

### 3.3.1 Scan $G_0$

Scanne les

- éléments terminaux
- éléments non-terminaux

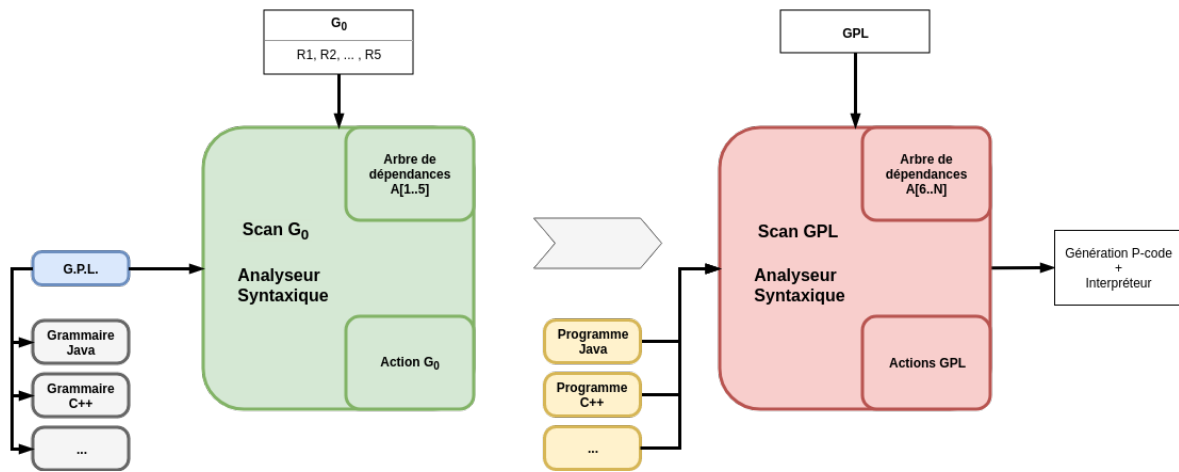


Figure 1: Projet Compilo

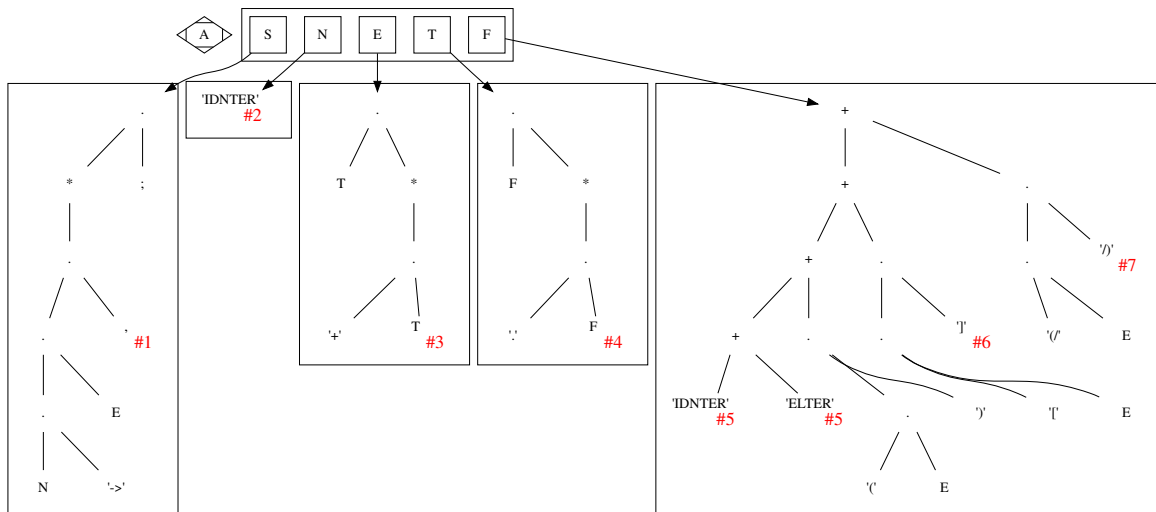


Figure 2: Arbres de dépendances  $G_0$

### 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 fois),  $n \geq 0$
- $(/X/) \iff X$  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 GO-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

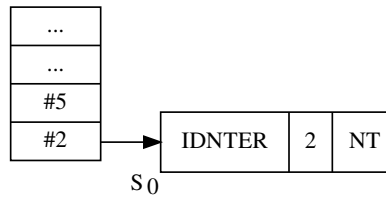


Figure 3: Pile

#### 3.9.2 Dictionnaires

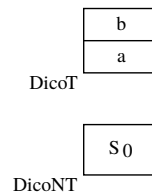


Figure 4: Dictionnaires

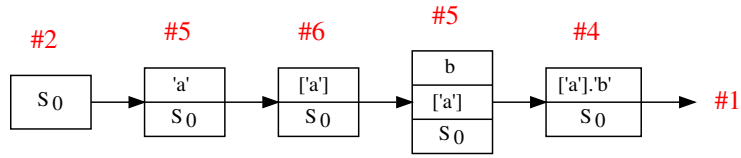


Figure 5: Compilation

### 3.9.3 Compilation

### 3.9.4 Arbre GPL

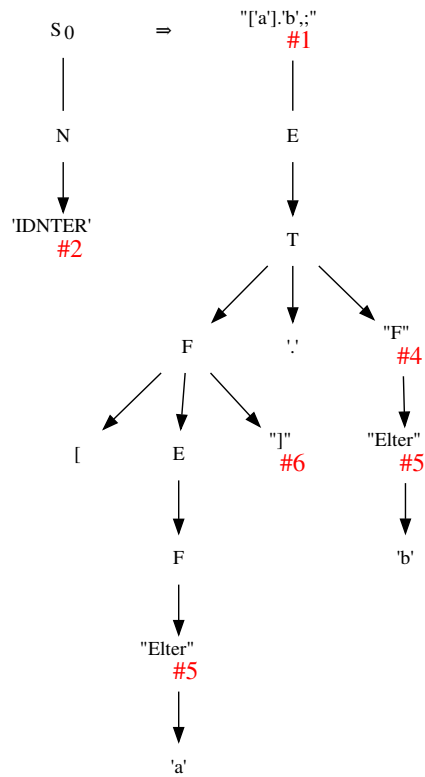


Figure 6: Arbre GPL

## 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 ( $A \rightarrow a$ )

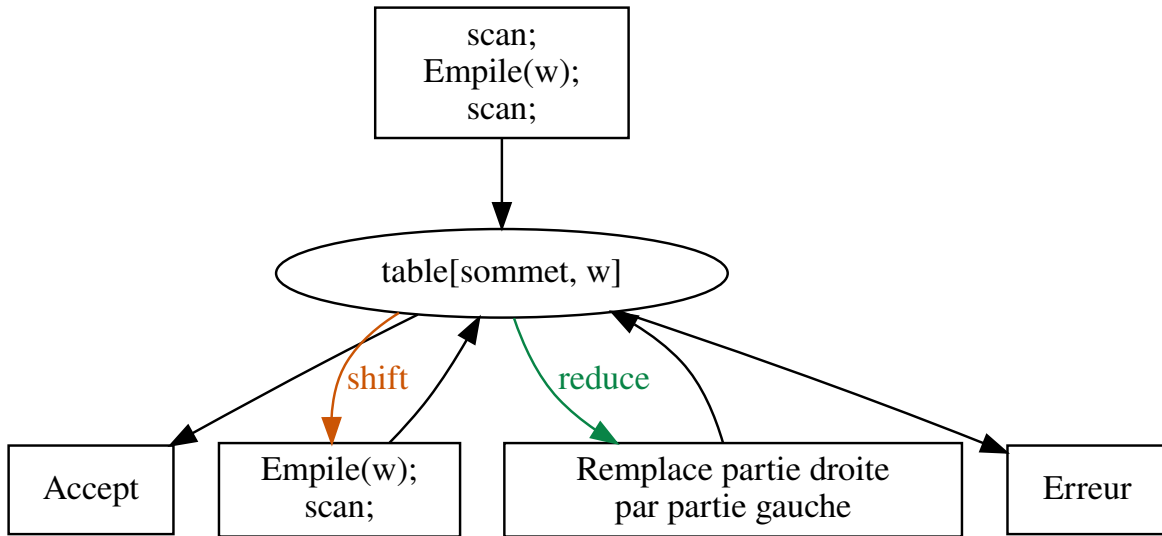


Figure 7: Algorithme Table Analyse L.R.

## 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

$$A \rightarrow \dots X.Q \dots \in \mathcal{P}$$

$$\text{et } Q \xRightarrow{*} Y$$

- $X \succ Y$  si

$$A \doteq Y$$

$$\text{et } A \xRightarrow{*} X$$

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

- (ligne  $\doteq$  colonne) et (ligne  $\prec$  colonne) se traduisent en (ligne Shift colonne)
- (ligne  $\succ$  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

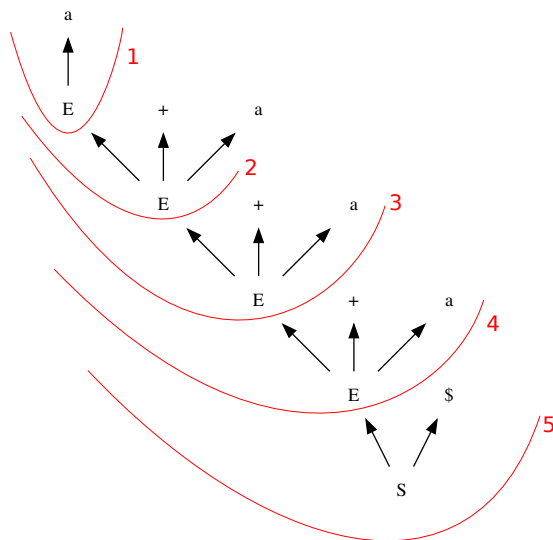


Figure 8: Arbre et poignées

## 5 Génération de code

### 5.1 Mnémoniques associés à un accumulateur

Load A  
STO A  
ADD A  
SUB A

#### 5.1.1 Exemples

$$y := a + b * c$$

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

### 5.2 Automatisation du processus

Utilisation de la notation post-fixée.

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

#### 5.2.1 Exemple

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

Notation post-fixée :  $ab + cd + /$

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

## 5.3 Génération de code avec plusieurs registres

### 5.3.1 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

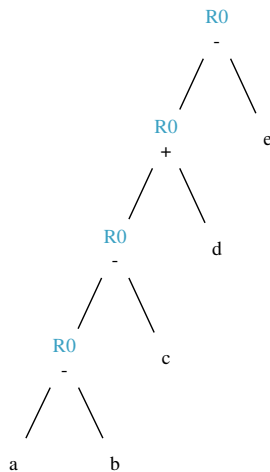
$$\Leftrightarrow R2 \leftarrow R2 + R1$$

**DIV** b, R1

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

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

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



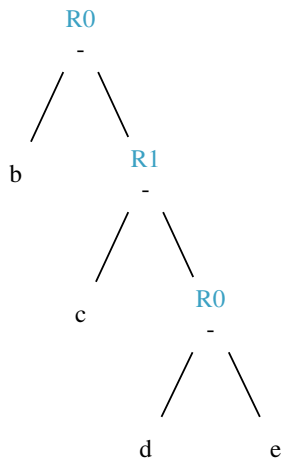
```

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

```

On a donc besoin d'un seul registre.

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



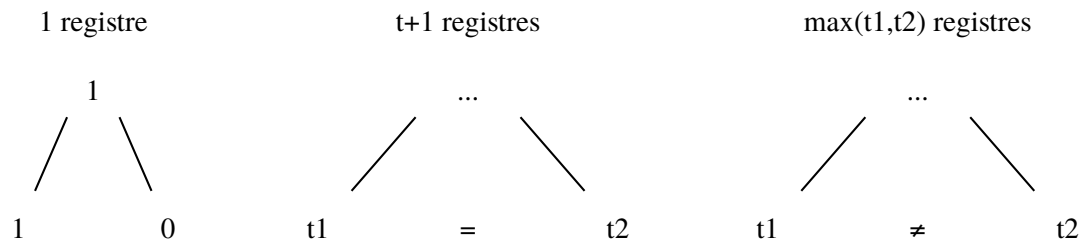
```

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

```

On a besoin de deux registres.

### 5.3.3 Règles générales





## 5.4 P-code

### 5.4.1 Opérations

Trois instances de chargement

```
LDA@ #address
LDV@ #value in add
LDC #constant
```

Quatre instances de saut

```
JMP #while
JIF #jump if false
JSR #jump if subroutine
RSR #return from subroutine
```

Opérateurs

```
ADD # +
Moins # -
DIV # /
Mult # *
NEG # (- unaire)
INC
DEC
```

Opérateurs relationnels

```
SUP # >
SUPE # >=
INF # <
INFE # <=
EG # =
DIFF # !=
```

Instances Els(?)

```
RD
RDLN # avec retour chariot
WRT
WRTLN
```

Opérateurs logiques

```
AND
OR
NOT
```

Affectation

```
AFF
```

Instances d'arrêt

```
STOP
INDA #Tableau -> adresse
INDV #Tableau -> valeur
```

### 5.4.2 Exemples

Programme Sommej #somme des entiers de i à n

```
var I, S, N :int;
début
  Read(N);
  S := 0;
  I := 1;
  while I <= N do
    début
      S := S + I;
      I := I + 1;
    fin
  writeln(S);
fin
```

| Instruction     | N° | Opération | V/A |
|-----------------|----|-----------|-----|
| Read(N)         | 1  | LDA       | 3   |
|                 | 3  | RD        |     |
|                 | 4  | AFF       |     |
| S := 0          | 5  | LDA       | 2   |
|                 | 7  | LDC       | 0   |
|                 | 9  | AFF       |     |
| I := 1          | 10 | LDA       | 1   |
|                 | 12 | LDC       | 1   |
|                 | 14 | AFF       |     |
| while I <= N do | 15 | LDA       | 1   |
|                 | 17 | LDA       | 3   |
|                 | 19 | INFE      |     |
|                 | 20 | JIF       | 39  |
| S := S + I      | 22 | LDA       | 2   |
|                 | 24 | LDV       | 2   |
|                 | 26 | LDV       | 1   |
|                 | 28 | ADD       |     |
|                 | 29 | AFF       |     |
| I := I + 1      | 30 | LDA       | 1   |
|                 | 32 | LDV       | 1   |
|                 | 34 | INC       |     |
|                 | 35 | ADD       |     |
|                 | 36 | AFF       |     |
| fin             | 37 | JMP       | 15  |
| writeln(S)      | 39 | LDV       | 2   |
|                 | 41 | WRTLN     |     |
| fin             | 42 | STOP      |     |

### 5.4.3 Procédures Exec et Interpréter

Procédure Exec

```
while P-code[c0] != STOP do Interpréter(P-code[c0])
```

```

Procédure Interpréter(x:int)
début
  case x of
    LDA:
      spx := spx + 1;
      Pilex[spx] := P-code[c0 + 1];
      c0 := c0 + 2;
      #le reste est spéculatif...
    LDV:
      spx := spx + 1;
      Pilex[spx] := Pilex[P-code[c0 + 1]];
      c0 := c0 + 2;
    JMP:
      c0 := c0 + 1;
    RD:
      spx := spx + 1;
      Pilex[spx] := Pilex[P-code[c0 - 1]];
    WRTLN:
      Pilex[spx]; # ou Pilex[P-code[c0 - 1]];
    AFF:
      Pilex[spx - 2] := Pilex[spx - 1]
    ...
  fin

```

## 6 Annexes

### 6.1 Exercices de TD

#### 6.1.1 Déterminisation d'un automate

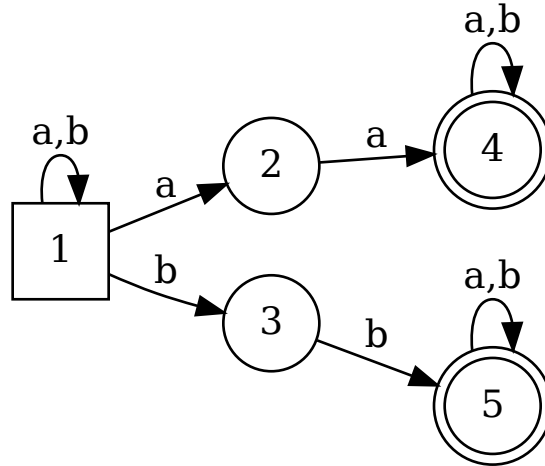


Figure 9: Automate non déterministe

#### 1. Donner la grammaire associée

$$S \rightarrow aS|bS|aA|aB$$

$$A \rightarrow aC|a$$

$$B \rightarrow bD|b$$

$$C \rightarrow aC|bC|a$$

$$D \rightarrow aD|bD|a|b$$

#### 2. La grammaire est-elle LL(1) ?

Non, car elle n'est pas déterministe.  $First(a) = \{a\}$   $First(aC) = \{a\}$   $First(a) \cup First(aC) = \{a\} \neq \emptyset$

#### 3. Déterminiser l'automate

|           | a             | b             |   |
|-----------|---------------|---------------|---|
| {1}       | {1,2} (A)     | {1,3} (B)     | S |
| {1,2}     | {1,2,4} (C)   | {1,3} (B)     | A |
| {1,3}     | {1,2} (A)     | {1,3,5} (D)   | B |
| {1,2,4}   | {1,2,4} (C)   | {1,3,4} (E)   | C |
| {1,3,5}   | {1,2,5} (F)   | {1,3,5} (D)   | D |
| {1,3,4}   | {1,2,4} (C)   | {1,3,4,5} (G) | E |
| {1,2,5}   | {1,2,4,5} (H) | {1,3,5} (D)   | F |
| {1,3,4,5} | {1,2,4,5} (H) | {1,3,4,5} (G) | G |
| {1,2,4,5} | {1,2,4,5} (H) | {1,3,4,5} (G) | H |

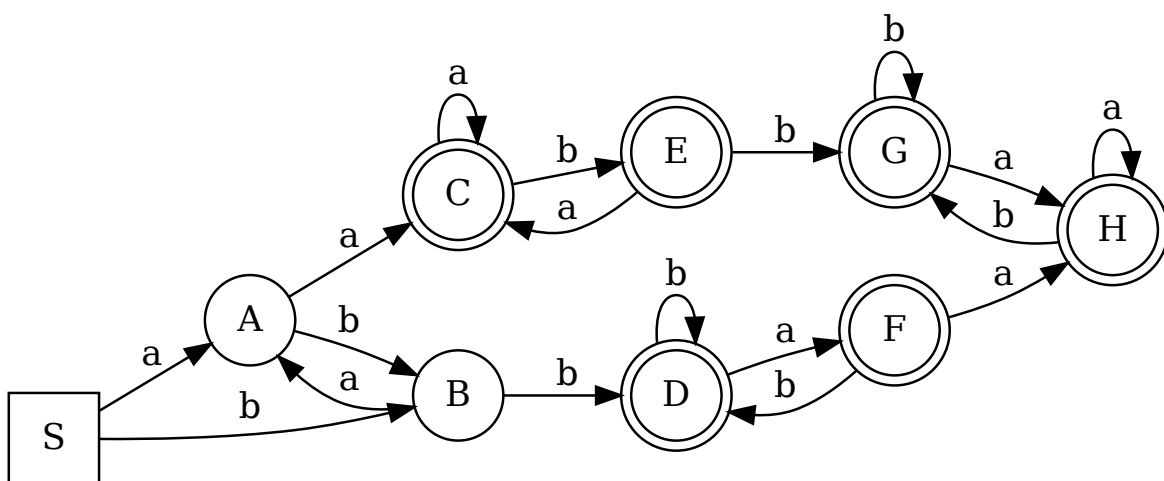


Figure 10: Automate déterministe