Rédigé avec LaTeX
Version du 22 octobre 2014

Traduction de langages

TL

M1 Informatique – Développement Logiciel
Semestre 7

Cours donné par Christine Maurel
Rédigé par Antoine de Roquemaurel

2014

Table des matières

1 Analyse Lexicale – Système d'équations de langages

1.1 Analyse lexicale

$$C = \{0, 1, \cdots, 8, 9\}$$

1.1.1

$$\begin{aligned} L &= O^*2\$(0+1)^+ + 0^*3\$(0+1+2) + \cdots + 0^*10\$(0+1+\cdots+9)^+ + (0+\cdots+9)^+ \quad (1.1) \\ &= C^+\$c^+ + c^+ = cc^*\$cc^* + cc^* \quad (1.2) \end{aligned}$$

R

— Solution (1.1) est à déterminiser

— Solution (1.2) il faudra ajouter des contraintes

1.1.2

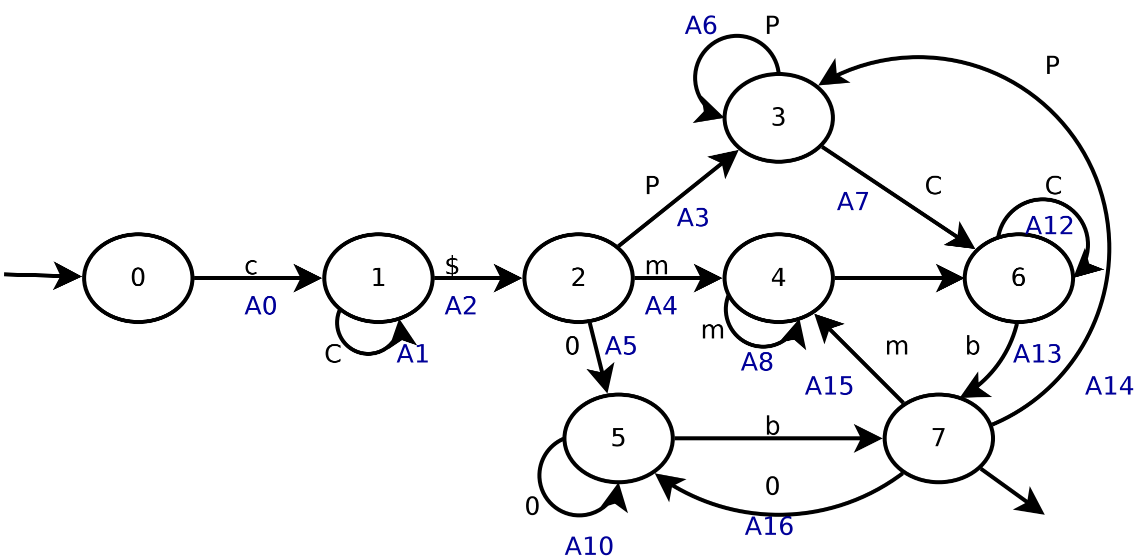

Figure 1.1 – Automate fini

```
  carrcour ;  val(carrcour)
variables:
  base; -- valeur décimale de la base avant '$'
  ecart; -- Valeur décimale d'un écart
  cpt; -- Compter les '+' ou '-' ou '0' successifs
  signe; -- Indique si on a des '+' ou '-' ou '0'
  --  +1 si on a '+'
  --  -1 si on a '-'
  --  0 si on a '0'

A1:  {base := base * 10 + val(valcarcour);}
A2:  {∅}
A3:  {cpt := 1; signe := 1;\}
A4:  {cpt := 1 ; signe := -1;\}
A5:  {cpt := 1; signe := 0; \}
A6:  {cpt := cpt + 1;\}
A7:  {ecart := val(carcours)\}
A8 = A10 = A6
A9 = A7
A12: \{ecart := ecart * 10 + val(carcours)}
A13: {
    tantque cpt != 0 faire
      printf(base + (signe * ecart));
      cpt := cpt - 1;
    fin tanque;
   }
 A11 = A13
 A14 = A3
 A15 = A4
 A16 = A5
```

1.1.3 Langage ALGOL60

R Une écriture en grammaire est également une écriture en langages, ainsi :

$$P = \begin{cases} A & \rightarrow & \alpha_1 \\ A & \rightarrow & \alpha_2 \\ & \cdots & \\ A & \rightarrow & \alpha_n \end{cases} \Leftrightarrow L(A) = \alpha_1 + \alpha_2 + \cdots + \alpha_n$$

1.1.3.1 Définition de la grammaire

$$\begin{array}{rcl} X & = & \{0, \cdots, 9, +, -, ., e\} \\ N & = & \{I, J, K, L, M, N, O\} \\ S & = & O \\ P & = & I + \lambda = cP + \lambda \end{array}$$

1.1.3.2 Système d'équation de langage égaux à G

$$\left\{\begin{array}{ccc} I & = & c + cI \\ J & = & I + sI \\ K & = & .I \\ L & = & \underbrace{I + K + IK}_{Problème de régularité ?} \\ M & = & eJ \\ N & = & Ł + M + LM \\ O & = & N + sN \end{array}\right.$$

I Entier signé

c $c \in \{0 \cdots 9\}$

J Entier

s $s \in \{+, -\}$

K Partie fractionnaire

L Nombre décimal

M Partie exponentiel

N Nombre non signé

O Nombre

> **R** Un automate fini ne peut reconnaître que les langages réguliers, qui sont engendrés par des grammaires linéaires à droite.
> L'union de langages régulier engendre un langage régulier, de même le produit de deux langages réguliers donnent un langage régulier.

Dans notre cas, L ne pose donc aucun problème de régularité étant donné que I et K sont des langages réguliers ainsi, l'union et le produit de langages réguliers engendrant des langages réguliers, L sera régulier.

Le problème pourrait se poser pour L mais aussi pour N et M : la réponse étant la même.

O est régulier, on peut donc trouver un automate finis le reconnaissant.

$$
\begin{aligned}
I &= c(\lambda + I)cP \\
P &= I + \lambda = cP + \lambda \\
J &= I + sI = cP + sI \\
K &= .I \\
L &= I + K + IK = \underline{c}P + .I + \underline{c}PK = \underbrace{c(P + PK)}_{Q} + .I \\
&= cQ + .I \\
Q &= P + PK = cP + \lambda + (cP + \lambda)K = cP + \lambda + cPK + K = c(\underbrace{P + PK}_{Q}) + .I + \lambda \\
&= cQ + .I + \lambda \\
M &= eJ \\
N &= L + M + LM = cQ + .I + eJ + (cQ + .I)M = cQ + .I + eJ + cQM + .IM + .IM \\
&= c(\underbrace{Q + QM}_{R}) + .(\underbrace{I + IM}_{S}) + eJ \\
R &= Q + QM = cQ + .I + \lambda + cQM + .IM + eJM = c(Q + QM) + .(I + IM) + eJ + \lambda \\
S &= I + IM = cP + cPM = c(P + PM) \\
T &= P + PM = cP + \lambda + cPM + \overset{M}{\underbrace{eJ}} = c(P + PM) = e + \lambda \\
O &= N + sN = cR + .S + eJJ + sN
\end{aligned}
$$

D'où le système d'équation de langage suivant :

$$
\left\{
\begin{array}{ccc}
I & = & cP \\
P & = & cP + \lambda \\
J & = & cP + sI \\
K & = & .I \\
L & = & cQ + .I \\
Q & = & cQ + .I + \lambda \\
M & = & eJ \\
N & = & cR + .S + eJ \\
R & = & cR + .s + eJ + \lambda \\
S & = & cT \\
T & = & cT + eJ + \lambda \\
O & = & cR + .s + eJ + sN
\end{array}
\right.
$$

L'axiome O est un état initial et {T,R,Q,P} sont des états finaux.

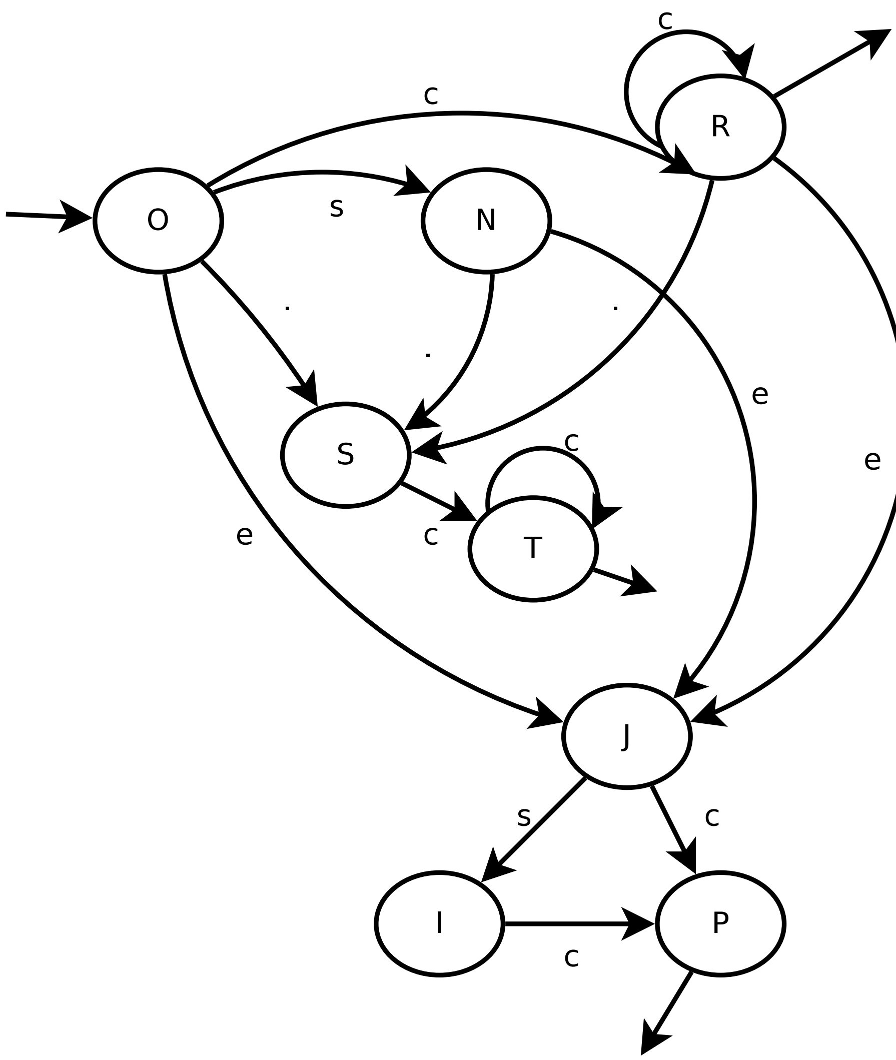

Figure 1.2 – Automate fini

1.2 Systèmes d'équations algébriques de langages

> **R** 3 théorèmes :
> **Arden** $X = r_1X + r_2 \Rightarrow X = r_1^*r_2$
> **Arden Bis** $X = Xr_1 + r_2 \Rightarrow X = r_2r_1^*$
> **AnBn** $X = Yr_1 + r_{\Rightarrow}X = r_2r_1^*$

1.2.1 G1

$$\begin{aligned} S &= S\underbrace{a}_{r_1} + \underbrace{b}_{r_2} \Rightarrow S \\ &\overset{Ardenbis}{=} ba* = ba^n n \geq 0 \end{aligned}$$

1.2.2 G2

$$\left\{\begin{array}{lcl} S & = & aSb + T \stackrel{AnBn}{\Longrightarrow} S = a^nTb^n, n \geq 0 = a^nb^+b^nn \geq 0 \\ T & = & Tb + b \stackrel{Ardenbis}{\Longrightarrow} \end{array}\right.$$

2 Gramaires LL(1) – Descente récursive

2.1

2.1.1 Firsts

$$
\begin{aligned}
S' &\rightarrow S\$ && (2.1)\\
S &\rightarrow bRS && (2.2)\\
S &\rightarrow RcSa && (2.3)\\
S &\rightarrow \lambda && (2.4)\\
R &\rightarrow acR && (2.5)\\
R &\rightarrow b && (2.6)\\
& && (2.7)
\end{aligned}
$$

$$
\begin{aligned}
first_1(R) &= \{a, b\}\\
first_1(S &= \{a, b, \lambda\}\\
first_2(R) &= \{ac, b\}\\
first_2(S) &= \{\underbrace{\lambda}_{2.3}, \underbrace{bb}_{2.2+R}\}
\end{aligned}
$$

R Pour le $first_2(R)$, ac est le préfixe de longueur 2 des mots prouits par R, et b est le mot de longueur inférieur ou égale à 2 produit par R.

Nous avons chercher les $first$ intuitivement, cependant nous nous sommes servis de la formule suivante : $first_k(\alpha_1\alpha_2\cdots\alpha_n) = first_k(first_k(\alpha_a)first(\alpha_2)\cdots first_k(\alpha_n)$

2.1.2 Follows

R $follow_k(y) = \cup first_k(S follow_k(x)))$

k = 2 Pour s :

1. $2lookahead(S \to bRS) first_2(bRS follow_2(S)) = b first_1(RS follow_2(S)) = \{ba, bb\} = E_1)$
2. $2lookahead(S \to RcSa) = first_2(Sa = follow_2(S)) = \{ac, bc\}$
3. $2lookahead(S \to RcSa) = first_2(RcSa = follow_2(S)) = first_2(first_2(R)c \cdots) = \{ac, bc\}$

2.2

Grammaire d'axiome θ puis augmentée :

$$\begin{cases} S' & \to & \theta\$^R \\ \theta & \to & cR \mid .S \mid eJ \mid sN \\ R & \to & cR \mid .S \mid J \mid \lambda \\ S & \to & cT \\ J & \to & cP \mid sI \\ N & \to & cR \mid .S \mid eJ \\ T & \to & cT \mid eJ \mid \lambda \\ P & \to & cP \mid \lambda \\ I & \to & cP \end{cases}$$

2.2.1 LL(1) ?

$$\begin{aligned} 1lookahead(\theta \to cR) &= \{c\} = \{0, 1, \cdots, 9\} \\ 1lookahead(\theta \to .S) &= \{.\} \\ 1lookahead(\theta \to eJ) &= \{e\} \\ 1lookahead(\theta \to sN) &= \{s\} = \{+, -\} \end{aligned}$$

2.2.2 Vérifions pour R, T et P

2.2.2.1 R ?

$$\begin{aligned} 1lookahead(R \to \lambda) &= first_1(\lambda.follow_1(R)) = \{\$\} \\ 1lookahead(R \to cR) &= \{c\} \\ 1lookahead(R \to .S) &= \{.\} \\ 1lookahead(R \to eJ) &= \{e\} \end{aligned}$$

OK, pas de conflit

2.2.2.2 P ?

$$follow_2(P) = \{\$\}$$

Ok, pas de conflit.

2.2.2.3 T ?

$$\begin{aligned} 1lookahead(T \to \lambda) &= first_1(\lambda follow_1(T)) = \{\$\} \\ &\Rightarrow follow_1(T) = follow_1(S) = follow_1(\theta) \cup follow_1(R) \cup follow_1(N) = \{\$\} \end{aligned}$$

Ok, pas de conflit

> **R** Si on a une grammaire linéaire à droite, on est sûr qu'elle est $LL(1)$.
>
> Quelque soit la règle de l'ensemble des règles de production P, $\forall A : A \to \lambda, A \to xB$

2.2.3 Procédures de descente récursive

A chaque $A \in N$ on associe une procédure de nom A qui contient « l'image de β ».

$$Si\ A \to \beta\ avec \left\{ \begin{array}{cccc} \beta = & x \in X & \Longrightarrow & SKIP('x') \\ \beta = & \lambda & \Longrightarrow & NULL \\ \beta = & B \in N & \Longrightarrow & B \\ \beta_1 = & \beta_1\beta_2\cdots\beta_n & \Longrightarrow & image(\beta_1); image(\beta_2); \cdots; image(\beta_n); \end{array} \right.$$

$$\left. \begin{array}{lcl} Si\ A & \to & \beta_1 \\ Si\ A & \to & \beta_1 \\ Si\ A & \to & \beta_1 \end{array} \right\} \Longleftrightarrow \begin{array}{lccc} & 1lookahead(A \to \beta_1) & : & image\beta_1 \\ | & 1lookahead(A \to \beta_2) & : & image\beta_2 \\ & & \vdots & \\ | & 1lookahead(A \to \beta_n) & : & image\beta_n \\ | & Others & : & ERREUR; \end{array}$$

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

```
procedure θ is
begin
  switch NEXTS
     number : SKIP(number.val); R;
    |. : SKIP('.') ; S ;
    | e : SKIP('e') ; J;
    | + : SKIP('+') ; N;
    | - : SKIP('-') ; N;
    | others : ERREUR;
end θ
```

```
procedure R is
begin
  switch NEXTS
     number : SKIP(number.val); R;
    |. : SKIP('.'); S;
    |e : SKIP('e') ; J ;
    |$ : NULL;
    |others : ERREUR;
```

```
procedure S is
begin
  SKIP(number.val);
  T;
end S
```

2.3

$$
\begin{array}{rcl}
< program > & \rightarrow & program < suiteDct > begin < switchInst > end \\
< suiteInst > & \rightarrow & < inst > | < suiteInst >; < inst > \\
< inst > & \rightarrow & if < exp > then < suiteInst > else < suiteInst > endif \\
 & & |while < exp > loop < suiteInst > endloop \\
 & & |repeat < suiteInst > until < exp > endloop \\
< suiteDcl > & \rightarrow & \cdots \\
< exp > & \rightarrow & \cdots
\end{array}
$$

R

— Non terminal : $< \cdots >$
— Terminal : $X = \{program, begin, end, if, \cdots\}$

2.3.1 Procédure de descente récursive

2.3.1.1 LL(1) ?

Pour $< program >$: ok
Pour $< inst >$: ok

$$
\begin{array}{rcl}
1lookahead() & = & \{if\} \\
1lookahead() & = & \{while\} \\
1lookahead() & = & \{repeat\}
\end{array}
$$

Pour $< suiteInst >$:

$$\begin{aligned} 1look(< suiteInst > &\rightarrow < inst >) = \{if, while, repeat\} \\ 1look(< suiteInst > &\rightarrow < suiteInst >; < inst >) = \{if, while, repeat\} \end{aligned}$$

La grammaire est non LL(1) car elle est récursive à gauche !, à cause de $< suiteInst > \rightarrow < suiteInst >; < inst >$

On peut la transformer en une grammaire linéaire à droite grâce à Arden.

$$A \rightarrow A\beta|\alpha \overset{ArgenBis}{\Longrightarrow} = \alpha\beta^*$$

Éliminer la récursivité à gauche :

$$\begin{aligned} L(A) &= \alpha\beta^* = \alpha L(A') \\ L(A') &= \beta^*\lambda \Longrightarrow \beta L(A') + \lambda \end{aligned}$$

Donc $< suiteInst > \rightarrow < inst > \mid < suiteInst >; < inst >$ devient :

$$\begin{aligned} \overbrace{< suiteInst >}^{A} &\rightarrow \overbrace{< inst >}^{\alpha}\overbrace{< suiteInstPrime >}^{A'} \\ < suiteInstPrime > &\rightarrow ; < inst >< suiteInstPrime > |\lambda \end{aligned}$$

On vérifie de nouveau qu'elle soit bien LL(1) ;

Pour <suiteInst> pas de problème, car une seule règle.

Pour $< suiteInstPrime >$:

$$\begin{aligned} 1lookahead(< suiteInstPrime > \rightarrow ; < inst > \leftrightarrow) &= \{;\} \\ 1lookahead(< suiteInstPrime > \rightarrow \lambda) &= first_1(\lambda.follow_1(< suiteInstPrime >) \\ &= \{end, endif, else, endloop, until\} = E_2. \end{aligned}$$

Donc G' est LL(1).

```
procedure SUITEINST is
begin
  INST;
  SUITEINSTPRIME;
end SUITEINST
```

```
procedure SUITEINSTPRIME is
begin
  switch NEXTS
  ';' : SKIP(';') ; INST ; SUITEINSTPRIME ;
  |end|endif|else|endloop|until : NULL;
  |others : ERREUR;
end SUITEINSTPRIME
```

k = 2 Pour s :

1. $2lookahead(S \rightarrow bRS)first_2(bRSfollow_2(S)) = bfirst_1(RSfollow_2(S)) = \{ba, bb\} = E_1$
2. $2lookahead(S \rightarrow RcSa) = first_2(Sa = follow_2(S)) = \{ac, bc\}$
3. $2lookahead(S \rightarrow RcSa) = first_2(RcSa = follow_2(S)) = first_2(first_2(R)c\cdots) = \{ac, bc\}$

3 Génération de code en quadruplets

Mettre en œuvre un processus de traduction en quadruplets des structures de contrôle <repeatstat>, <forstat> et <casestat> par la méthode de la descente récursive.

```
<statlist > := <inst> <suite>
<suite> := ; <statlist >  λ
<inst> := \ldots<ifstat><whilestat><repeatstat><forstat><casestat>\ldots
```

Méthodologie

1. Donner le schéma équivalent en quadruplets
2. Inventaire des « problèmes » et solution apportées
3. Mise en œuvre des porcédures de descente récursive pour engendrer le schéma en quadruplets voulus en tenant compte des solutions

1. `<repeatstat> ...= repeat <statlist> until <exp> endrepeat`
2. `<forstat> ...= for ident := <exp1> step <exp2> until <exp3> do <statlist> endfor`
3. `<casestat> ...= case <exp> of <alternative> {; <alternative> }* otherwise <statlist> endcase`
4. `<alternative> ...= nombre : <statlist>`

3.1

```
procedure REPEATSTAT is
  // declaration
  DEBUT : Integer
  TEST : Integer
  RES : String

  // instruction
  begin
    SKIP('repeat');
    DEBUT := NEXTQUAD;
    EXP(Res);
    GEN("=?,"^Res^", 0, nil");
    SKIP('until');
    EXPR(RES);
    CHECKTYPE(Res, Boolean);
    STATLIST;
    SKIP('endloop');
```

```
    GEN("goto, nil, nil, " ^str(debut));
    BACKPATCH([TEST], NEXTQUAD);
end REPEATSTAT
```