Analyse Lexicale

1. Généralités
2. Expressions Rationnelles
3. Automates finis
4. Analyseurs Lexicaux

Rôle de l'analyseur lexical

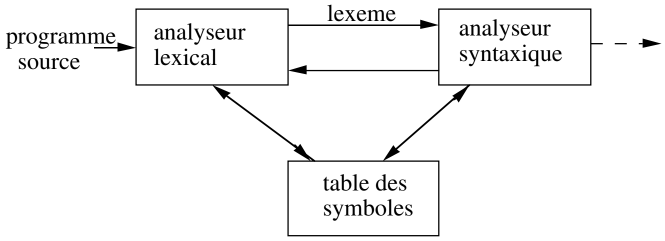


Figure 1: Interactions avec les autres modules

Avantages

1. Séparer l'analyse lexicale de l'analyse syntaxique réduit la complexité globale ainsi que celle de la conception de chacun des modules.
2. On augmente ainsi l'efficacité du compilateur
3. On augmente la portabilité.

Méthodes de construction

1. Utiliser un générateur d'analyseur lexical
2. Ecrire l'analyseur dans un langage évolué.
3. Ecrire l'analyseur en assembleur en gérant explicitement l'acquisition de données.

Lecture de la source

Un des schémas utilise un tampon géré en deux moitiés de taille N permettant une lecture en une seule commande.

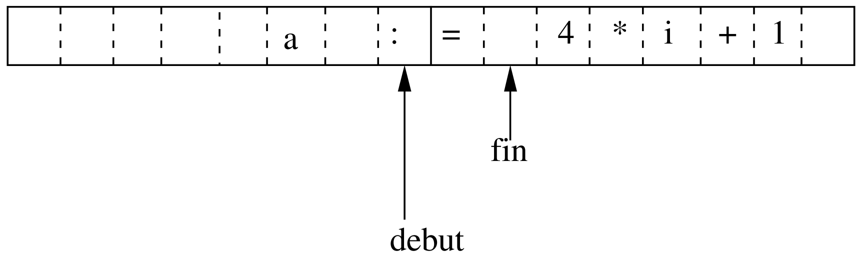


Figure 2: Tampon de lecture

Le tampon permet de prendre en compte l'*anticipation* : on doit lire en avant pour décider la valeur du lexème.

A l'extrême, on a besoin d'une anticipation non bornée comme dans

`DECLARE ( ARG 1, ARG 2, ... , ARG`n)

en PL/1: il faut attendre la fin de l'expression pour savoir si `DECLARE` est un mot cle ou un nom de tableau.

Gestion du pointeur *debut*

```
si debut a la fin la premiere moitie  alors
 debut
   recharger la deuxieme moitie
    debut := debut + 1
 fin
sinon si debut a la fin de la deuxieme moitie  alors
 debut
   recharger la premiere moitie
   placer  debut en tete de la premiere moitie
 fin
 sinon debut := debut +1
```

Mots et Langages Formels

On définit en général un *mot* sur un *alphabet* A comme une suite finie de symboles pris dans A.

L'alphabet A peut être l'ensemble des caractères de base et les mots les lexèmes ou:

	alphabet	mots
code ASCII	0,1	caractères
analyse lexicale	caractères	lexemes
analyse syntaxique	lexemes	programmes

On utilise le *mot vide* noté ϵ.

Un *langage formel* est un ensemble de mots.

Expressions Rationnelles

On se donne un alphabet.

On forme des expressions en partant des lettres (éléments de l'alphabet) et de ϵ en utilisant:

1. l'union notée | (ou +)
2. La concaténation (pas de symbole)
3. L'itération *

On utilise souvent des symboles supplémentaires:

1. + pour l'itération non vide
2. . pour l'alphabet
3. ? pour l'option
4. etc.

Exemples

Identificateurs en Pascal:

$$\begin{aligned}\textbf{lettre} &\rightarrow A \,|B\,|\ldots|Z\,|a\,|b\,|\ldots|z\\ \textbf{chiffre} &\rightarrow 0\,|\,1\,|\,2\,|\ldots|\,9\\ \textbf{id} &\rightarrow \textbf{lettre (lettre } |\textbf{ chiffre)*}\end{aligned}$$

Nombres sans signe en Pascal:

$$\begin{aligned}\textbf{chiffre} &\rightarrow 0\,|\,1\,|\,2\,|\ldots|\,9\\ \textbf{chiffres} &\rightarrow \textbf{chiffre chiffre*}\\ \textbf{frac} &\rightarrow \textbf{.chiffres} \,|\, \epsilon\\ \textbf{exp} &\rightarrow (E(+\,|\,-\,|\epsilon)\textbf{chiffres} \,|\, \epsilon\\ \textbf{num} &\rightarrow \textbf{chiffres frac exp}\end{aligned}$$

Avec les abréviations:

$$\begin{aligned}\textbf{chiffre} &\rightarrow 0\,|\,1\,|\,2\,|\ldots|\,9\\ \textbf{chiffres} &\rightarrow \textbf{chiffre}^{+}\\ \textbf{frac} &\rightarrow (\textbf{.chiffres})?\\ \textbf{exp} &\rightarrow (E(+\,|\,-)?\textbf{chiffres})?\\ \textbf{num} &\rightarrow \text{chiffres frac exp}\end{aligned}$$

Reconnaissance des lexèmes

Si on définit les expressions rationnelles:

$$
\begin{aligned}
\textbf{delim} &\rightarrow \textbf{blanc} \mid \textbf{tab} \mid \textbf{newline} \\
\textbf{ws} &\rightarrow \textbf{delim}^{+}
\end{aligned}
$$

on obtient le tableau de spécifications:

EXPRESSION	LEXEME	ATTRIBUT
ws	-	-
if	**if**	-
then	**then**	-
else	**else**	-
id	**id**	pointeur
num	**num**	pointeur
$<$	**relop**	LT
$<=$	**relop**	LE
$=$	**relop**	EQ
$<>$	**relop**	NE
$>$	**relop**	GT
$>=$	**relop**	GE

Automates finis

Pour analyser les lexèmes, on construit des diagrammes avec:

1. des sommets appelés *états*
2. des transitions étiquetées
3. des actions dans les états de reconnaissance.

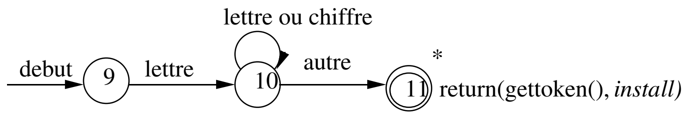


Figure 3: Identificateurs en Pascal

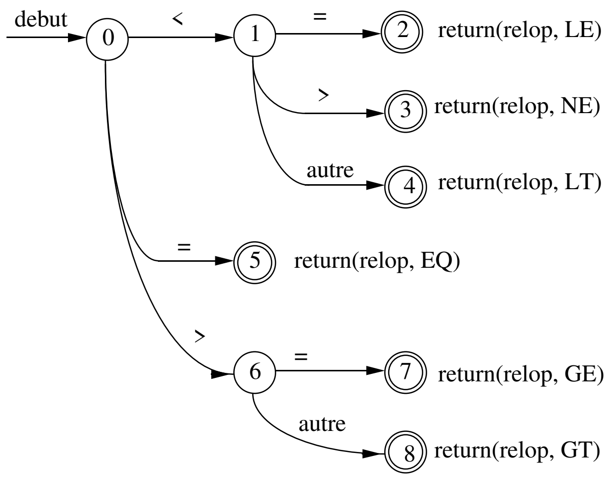


Figure 4: Opérateurs relationels

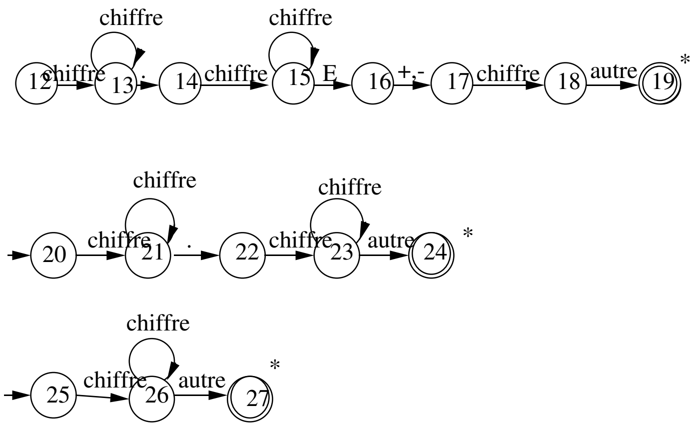


Figure 5: Nombres sans signe

Implémentation en C

Recherche de l'état de départ:

```
int state = 0, start = 0;
int lexical_value;
  /*pour la deuxieme partie du lexeme*/
int fail()
{
   forward = token_debut;
   switch (start) {
      case 0:   start = 9; break;
      case 9:   start= 12; break;
      case 12:  start= 20; break;
      case 20:  start= 25; break;
      case 25:  recover(); break;
      default:   /*erreur*/
```

```
    }
    return start;
}
```

```
token nexttoken()
{ while(1) {
    switch ( state) {
    case 0: c = nextchar();
    /*c est le caract\`ere de lecture*/
    if (c==blank || c==tab || c==newline) {
      state = 0;
      debut++;
      /*avance le debut du lexeme*/
    }
    else if (c == '<') state = 1;
    else if (c =='=') state = 5;
    else if (c == '<') state = 6;
    else state = fail();
    break;
    .../*cas 1-8 */

    case 9: c = nextchar();
      if (isletter(c)) state = 10;
      else state= fail();
      break;

    case 10: c = nextchar();
      if (isletter(c)) state = 10;
      else if (isdigit(c)) state = 10;
```

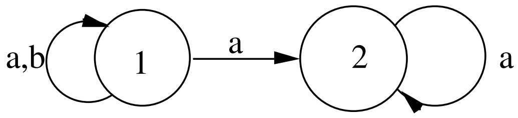
```
      else state = 11;
      break;
    case 11: retract(1); install_id();
      return(gettoken() );

    .../*cas 12-24 ici*/

    case 25: c = nextchar();
      if (isdigit(c)) state = 26;
      else state = fail();
      break;
    case 26: c = nextchar();
      if (isdigit(c)) state = 26;
      else state = 27;
      break;
    case 27: retract(1); install_num();
      return(NUM);
    }
  }
}
```

Automates déterministes

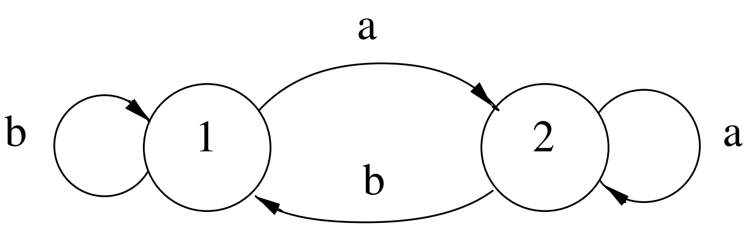
Un automate est déterministe si:

1. Il possède un seul état initial
2. De chaque état part au plus une flèche étiquetée par chaque symbole.

Par exemple, l'automate:



n'est pas déterministe alors que:



est déterministe.

Déterminisation

Tout automate fini est équivalent à un automate fini déterministe. C'est une propriété fondamentale des automates.

Algorithme: on part d'un automate non-déterministe $\mathcal{A} = (Q, I, T)$. On construit l'automate déterministe $\mathcal{D}$ ayant

1. comme états les parties de Q
2. comme état initial I
3. comme transitions: (U, a, V) si V est l'ensemble des états atteignables depuis U par une transition a
4. comme états terminaux les $U \subset Q$ tels que $U \cap T \neq \emptyset$.

Complexité: si $\mathcal{A}$ a n états, $\mathcal{D}$ peut avoir 2^n états.

Exemple

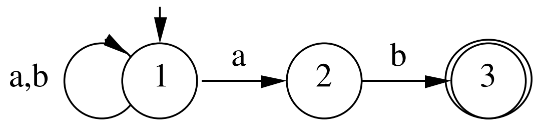


Figure 6: Un automate non-déterministe

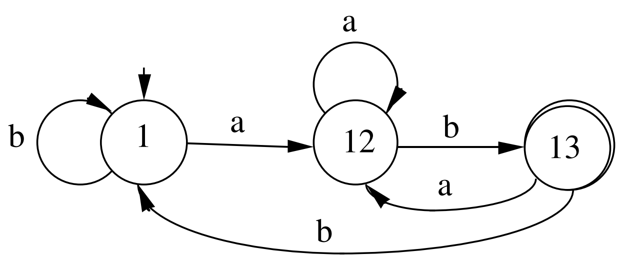


Figure 7: Le résultat de l'algorithme

Généralisation

On peut admettre des transitions d'étiquette ϵ (transitions *spontanées*).

Pour déterminiser, il faut cette fois

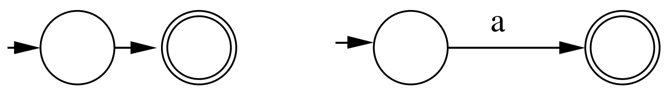
1. prendre pour l'étape 2 les états atteignables depuis I par un ϵ-chemin.
2. et pour 3 l'ensemble des états atteignables par un *chemin* d'étiquette a.

La construction récursive

On peut construire récursivement un automate fini reconnaissant une expression rationnelle. L'automate construit est tel que:

1. Il a un seul état initial i
2. Il a un seul état final $t \neq i$
3. Aucune flèche ne rentre dans i
4. Aucune flèche ne sort de t

Pour ϵ et a, on prend:



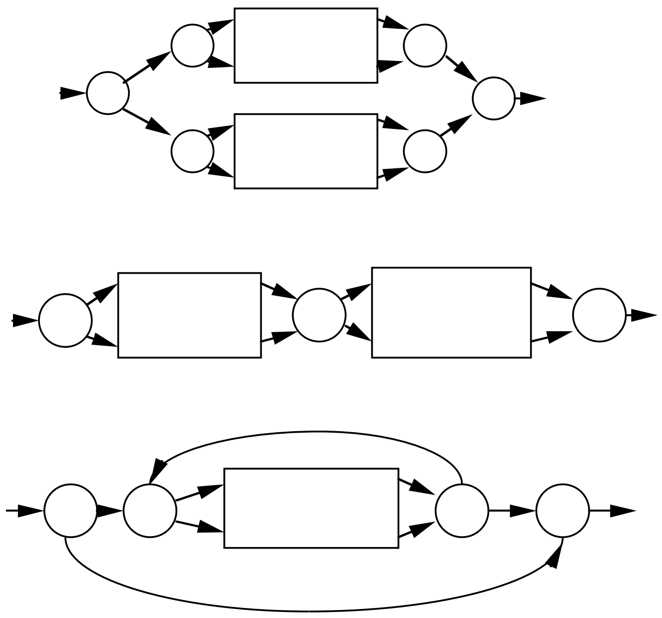


Figure 8: Union, produit et étoile

Utilisation de Lex

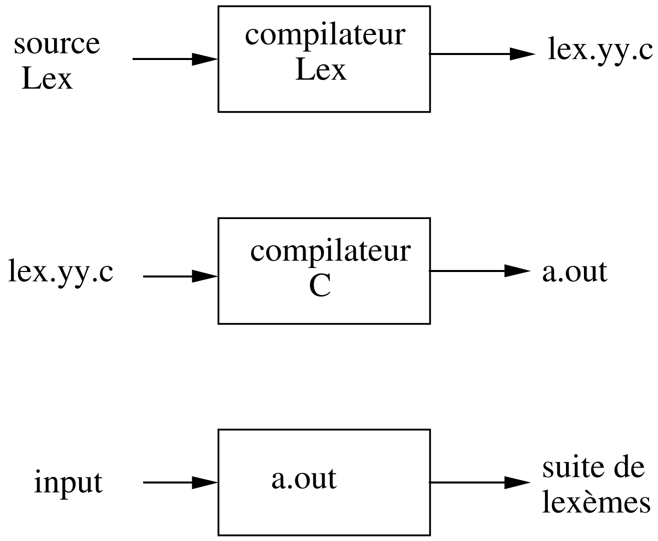


Figure 9: Mise en oeuvre

Spécifications Lex

Un programme Lex est fait de trois parties:

déclarations
%%
règles de traduction
%%
procédures auxiliaires

Les règles de traduction sont de la forme:

$$\begin{array}{ll} p_1 & \{action_1\} \\ p_2 & \{action_2\} \\ \ldots & \\ p_n & \{action_n\} \end{array}$$

où chaque p_n est une expression rationnelle et chaque action est une suite d'instructions C.

Exemple

```
%{
   /*definition de constantes
    LT, LE, EQ, ... */
%}

/* definitions */
delim     [ \t\n]
ws        {delim}*
letter    {A-Za-z}
digit     {0-9}
id        {letter}({letter}|{digit})*
number    {digit}+{\.{digit}+?(E{+\-}?{digit}+)*

%%

{ws}      {/*pas d'action*/}
if        {return(IF);}
then      {return(THEN);}
else      {return(ELSE);}
{id}      {yyval = install_id(); return(ID);}
{number}  {yyval = install_num(); return(NUMBER);}
"<"       {yyval = LT; return(RELOP);}
"<="      {yyval = LE; return(RELOP);}
```

```
%%

install_id() {
   /* procedure d'installation du lexeme
    dont le premier caractere est point\'e
    par yytext, dont la longueur est yylength
    dans la table des symboles. Retourne
    un pointeur sur la table*/
}

install_num() {
   /* idem pour les nombres*/
}
```