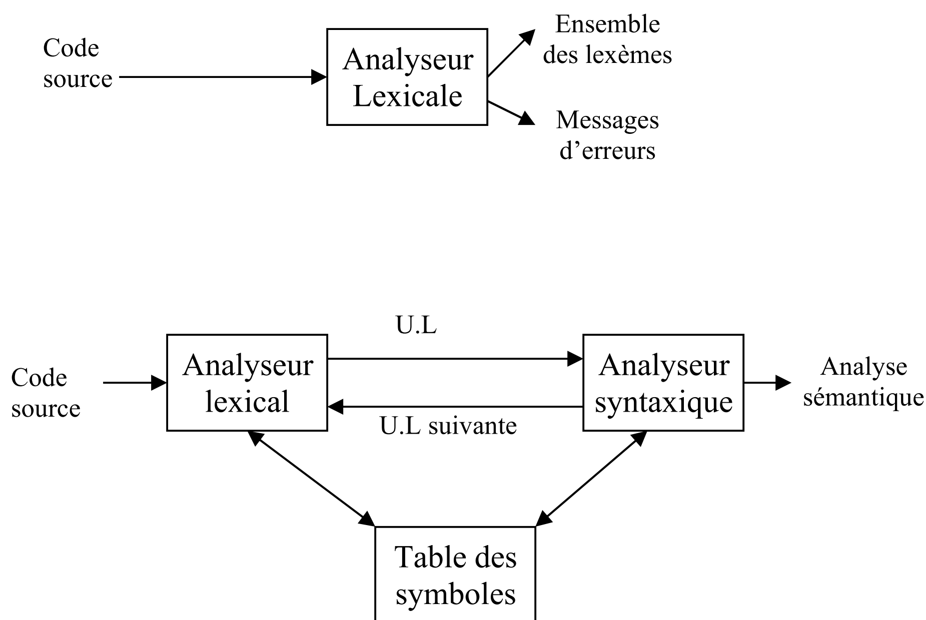
Chapitre 02 : Analyse lexicale

1. Introduction
2. Définitions
3. Expressions régulières
4. Automates
5. Erreurs Lexicales
6. Mis en œuvre d'un analyseur lexical
7. Outil Lex (générateur d’analyseur Lexical)

1. Introduction

L'analyseur lexical constitue la première étape d'un compilateur. Sa tâche principale est de lire les caractères d'entrée et de produire comme résultat une suite d'unités lexicales que l'analyseur syntaxique aura à traiter.



En plus, l'analyseur lexical réalise certaines tâches secondaires comme l'élimination de caractères superflus (commentaires, tabulations, fin de lignes, ...), et gère aussi les numéros de ligne dans le programme source pour pouvoir associer à chaque erreur rencontrée par la suite la ligne dans laquelle elle intervient.

2. Définitions

1. **Une unité lexicale** est une suite de caractères qui a une signification collective.
2. **Un lexème** toute suite de caractère du programme source qui concorde avec **le modèle** d'une unité lexicale.
3. **Un modèle (Règle lexicale**) est une règle associée à une unité lexicale qui décrit l'ensemble des chaînes du programme qui peuvent correspondre à cette unité lexicale.
4. **Attributs** : informations concernant le lexème (champs dans la table des symboles) ;

Exemples

- L'unité lexicale IDENT (identificateurs) en C a pour modèle : toute suite non vide de caractères composée de chiffres, lettres ou du symbole "_" et qui commencent par une lettre. Des exemples de lexèmes pour cette unité lexicale sont : truc, i, a1, ajouter_valeur ...
- L'unité lexicale NOMBRE (entier signé) a pour modèle : toute suite non vide de chiffres précédée éventuellement d'un seul caractère parmi(+,-) . Comme par exemple : -12, 83204, +0 ...
- L'unité lexicale REEL a pour modèle : tout lexème correspondant à l'unité lexicale NOMBRE suivi éventuellement d'un point et d'une suite (vide ou non) de chiffres, le tout suivi éventuellement du caractère E ou e et d'un lexème correspondant à l'unité lexicale NOMBRE. Cela peut également être un point suivi d'une suite de chiffres, et éventuellement du caractère E ou e et d'un lexème correspondant à l'unité lexicale NOMBRE. Exemples de lexèmes : 12.4, 0.5e3, 10., -4e-1, -.103e+2

3. Erreurs Lexicales

Peu d'erreurs sont détectables au seul niveau Lexical :

Plusieurs stratégies sont possibles :

- mode panique : on ignore les caractères qui posent problème et on continue. Cette technique se contente de refiler le problème à l'analyseur syntaxique

- transformations du texte source : insérer un caractère, remplacer, échanger, etc. Elle se fait en calculant le nombre minimum de transformations à apporter au mot qui pose problème pour en obtenir un qui ne pose plus de problèmes. Cette technique de récupération d'erreur est très peu utilisée en pratique car elle est trop coûteuse à implanter.

4. Expressions régulières

Une expression régulière est une notation pour décrire un langage régulier.
Soit A un alphabet (un ensemble de lettres), une expression régulière est donc:

1. Les éléments de A, ε et $\varnothing$ sont des expressions régulières.
2. Si α et β sont des expressions régulières, alors $(\alpha \mid \beta)$, $(\alpha\beta)$ et α^* sont des expressions régulières. $(\alpha \mid \beta)$ représente l'union, $(\alpha\beta)$ la concaténation et α^* la répétition (un nombre quelconque de fois). ε est l'élément neutre par rapport à la concaténation et $\varnothing$ est l'ensemble vide de caractère, neutre par rapport à l'union.

Exemple :

(a|b)*abb,

Identificateur = alpha (alpha | numer)*
En fait, les expressions régulières sont beaucoup plus puissantes que ce dont on a besoin lorsqu'on fait de l'analyse lexicale.

Exemple

On décrit les lexèmes par des expressions régulières.
Les mots clés: "for", "if"
Les variables: ['a'-'z']+ ['0'-'9']*
Les entiers: ['0'-'9']+
Les symboles: '(', ')', '+', '*', '='
Le lexème vide: (' ' | '\n')

L'ordre de priorité

Les opérateurs ∗, concaténation et | sont **associatifs à gauche**, et vérifient

1. ∗
2. concaténation
3. |

Définitions régulières

La nomination des expressions régulières est dite une définition régulière. Ces noms seront utilisés pour construire d'autres expressions régulières. On écrit donc

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\ldots$$
$$d_n \rightarrow r_n$$

où chaque d_i est un nom distinct et chaque r_i est une expression régulière sur l'alphabet $\Sigma \cup \{d_1, d_2, d_{i-1}\}$

Exemple

Voici quelques définitions régulières, et notamment celles de identificateur et nombre, qui définissent les identificateurs et les nombres du langage Pascal :

lettre → A | B | . . . | Z | a | b | . . . | z
chiffre → 0 | 1 | . . . | 9
identificateur → lettre (lettre | chiffre)∗
chiffres → chiffre chiffre ∗
frac → . chiffres | ε
Exp → (E (+ | - | ε) chiffres) | ε
nombre → chiffres frac exp

Notations abrégées

Pour alléger certaines écritures, on complète la définition des expressions régulières en ajoutant les notations suivantes :

- Soit x une expression régulière, définissant le langage L(x) ; alors $(x)^+$ est une expression régulière, qui définit le langage $(L(x))^+$,
- Soit x une expression régulière, définissant le langage L(x) ; alors (x)? est une expression régulière, qui définit le langage $L(x) \cup \{ \varepsilon \}$,
- Si c_1 , c_2,. . c_k sont des caractères, l'expressions régulière $c_1 |c_2 |... |c_k$ peut se noter $[c_1\ c_2 \ldots c_k]$,
- L'expression $[c_1\text{–}c_2]$ désigne la séquence de tous les caractères c tels que $c_1 \le c \le c_2$.

Exemple

Les définitions de lettre et chiffre données ci-dessus peuvent donc se réécrire :
lettre → [A–Za–z]
chiffre → [0–9]

5. Automates

Un automate à états finis (AEF) est défini par

- un ensemble fini E d'états
- un état e_0 distingué comme étant l'état initial
- un ensemble fini T d'états distingués comme états finaux (ou états terminaux)
- un alphabet Σ des symboles d'entrée
- une fonction de transition $\delta : A \times E \rightarrow E$ qui à tout couple formé d'un état et d'un symbole de fait correspondre un ensemble (éventuellement vide) d'états : $\delta\ (e_i,a)=\{ei1,\ldots,e_{in}\}$

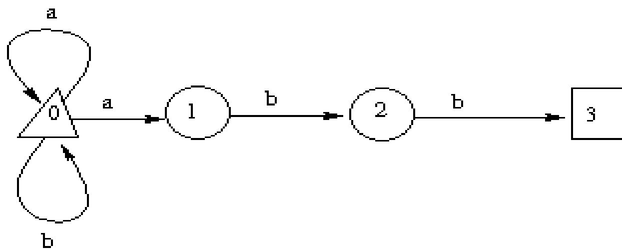
Les automates sont souvent donnés sous la forme d'un graphe: les états sont les nœuds du graphe et les arcs correspondent à la fonction de transition.

Exemple

$\Sigma=\{a,b\}$, $E=\{0,1,2,3\}$, $e_0=0$, $T=\{3\}$

$\delta(0,a)=\{0,1\}$, $\delta(0,b)=\{0\}$, $\delta(1,b)=\{2\}$, $\delta(2,b)=\{3\}$,

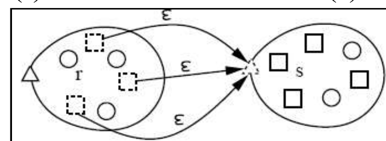
Représentation graphique :



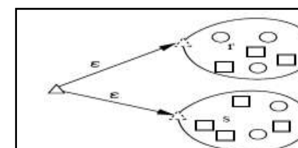
Construction d'un AFN à partir d'une E.R.

Pour une expression régulière s, on note A(s) un automate reconnaissant cette expression.

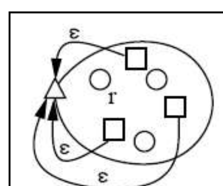
1. automate acceptant la chaîne vide
2. automate acceptant la lettre a
3. automate acceptant (r)(s)
   1. mettre une ε -transition de chaque état terminal de A(r) vers l'état initial de A(s)
   2. les états terminaux de A(r) ne sont plus terminaux
   3. le nouvel état initial est celui de A(r)
   4. (l'ancien état initial de A(s) n'est plus état initial)



4. automate reconnaissant r|s
   1. créer un nouvel état initial q
   2. mettre une ε-transition de q vers les états initiaux de A(r) et A(s)
   3. (les états initiaux de A(r) et A(s) ne sont plus états initiaux)



5. automate reconnaissant r+

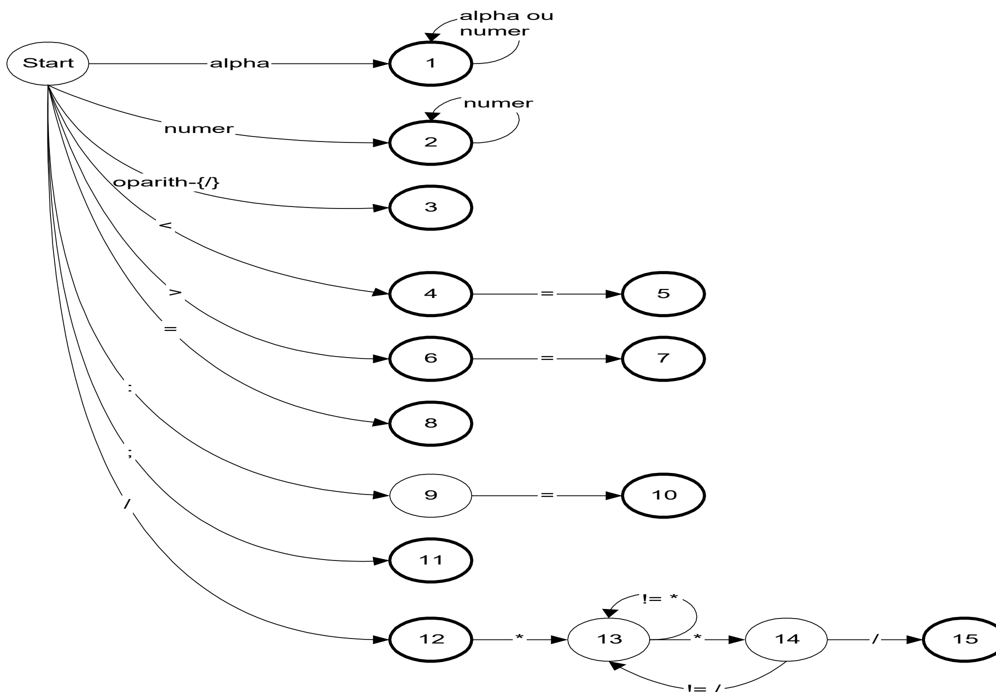


6. Mise en œuvre d'un analyseur lexical

A) A la main

Un automate peut très facilement être simulé par un algorithme. C'est encore plus facile si l'automate est déterministe. Alors, on peut écrire un programme reconnaissant tout mot de tout langage régulier. Ainsi, si l'on veut faire l'analyse lexicale d'un langage régulier, il suffit d'écrire un programme simulant l'automate qui lui est associé.

Exemples



```
buffer=&Buffer[0];
etat=0;
while(1) {
  c = lireChar();
  switch(etat){
   case: 0
          switch(type(c)){
                buffer*=c;buffer++;
                case ALPHA: etat=1;break;
                case NUMER: etat=2;break;
                case OPARITH-{/}: etat=3; break;
                case '<': etat=4; break;
                case '>': etat=6; break;
                case '=': etat=8; break;
                case ':': etat=9; break;
                case ';': etat=11; break;
                case '/': etat=12; break;
                default: erreur;}
                break;
    case 1:
                if (type(c) == ALPHA ou type(c) == NUMER) {
                buffer*=c;buffer++;
                etat=1;}
                else {buffer*='\0'; remettreCar();//ungetc(f,c);
                return(ajouter_token(Buffer,TYPE_IDENT));
     etat=0     } break;
                case 2:
                if (type(c) == NUMER){
                buffer*=c;buffer++;
                etat=2;}
                else { buffer*='\0';remettreCar();
                return(ajouter_token(Buffer,TYPE_NUMER));} break;
                case 3:
                return(ajouter_token(Buffer,TYPE_OPER)); break;
                {{à finir!!}} }
```

B) Automatiquement

Il existe des outils pour écrire des programmes simulant des automates à partir de simples définitions régulières. Par exemple : flex

L'outil Lex (générateur d'analyseur Lexical)

1. Introduction

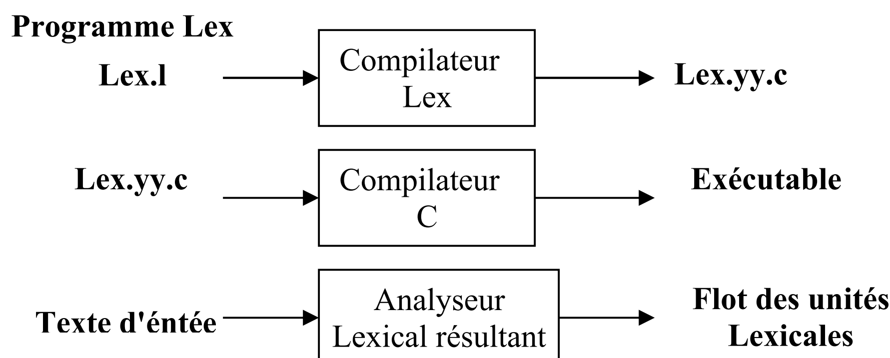
Lex est un utilitaire d'Unix. Son grand frère fLex est un produit GNU. Lex accepte en entrée des spécifications d'unités Lexicales sous forme de **définitions régulières** et produit un programme écrit dans le langage C qui, une fois compilé, reconnaît ces unités Lexicales. Ce programme est donc un analyseur Lexical).

```
> Lex fichier.l
```

donne le fichier Lex.yy.c qu'il faut compiler avec Lex :

```
> gcc Lex.yy.c –l l
```

Le fichier de spécifications Lex contient des expressions régulières suivies d'actions (règles de traduction). L'analyseur Lexical obtenu lit le texte d'entrée caractère par caractère jusqu'à ce qu'il trouve le plus long préfixe du texte d'entrée qui corresponde à l'une des expressions régulières. Dans le cas où plusieurs règles sont possibles, c'est la première règle rencontrée (de haut en bas) qui l'emporte. Il exécute alors l'action correspondante. Dans le cas où aucune règle ne peut être sélectionnée, l'action par défaut consiste à copier le caractère du fichier d'entrée en sortie.



2. Les expressions réguliers Lex

Une expression régulière Lex se compose de caractères normaux et de méta-caractères, qui ont une signification spéciale: $, \, ^, [,], {,}, <, >, +, -, *, /,|,?. Le tableau suivant donne les expressions régulières reconnues par Lex. Attention, Lex fait la différence entre les minuscules et les majuscules.

Expression	Signification	Exemple
c	tout caractère *c* qui n'est pas opérateur ou métacaractère	a
\c	caractère littéral c (lorsque c est un métacaractère)	\+ \.
"s "	chaîne de caractères	"bonjour "
.	n'importe quel caractère, sauf retour à la ligne (\n)	a.b
^	l'expression qui suit ce symbole débute une ligne	^abc
$	l'expression qui précède ce symbole termine une ligne	abc$

[s]	n’importe quel caractère de s	[abc]
[^s]	n’importe quel caractère qui n’est pas dans s	[^xyz]
r*	0 ou plusieurs occurrences de r	b*
r+	1 ou plusieurs occurrences de r	a+
r?	0 ou 1 occurrence de r	d?
r{m}	m occurrences de r	e{3}
r{m,n}	entre m et n occurrences de r	f{2,4}
r_1r_2	r_1 suivie de r_2	ab
$r_1\|r_2$	r_1 ou r_2	c|d
r_1/r_2	r1 si elle est suivie de r2	ab/cd
(r)	r	(a|b)?c
<x>r	r si Lex se trouve dans l’état x	<x>abc

3. Structure d'un fichier Lex

Un fichier de description pour Lex est formé de trois parties, selon le schéma suivant :

```
%{
        Déclaration en C des variables, des constants,…etc.
%}
    Déclaration des définitions régulières.
%%
    Expression régulières et actions correspondantes
%%
    Déclaration des procédures auxiliaires
```

Dans lequel aucune partie n'est obligatoire. Cependant, le séparateur "%%" est utilisé pour séparer entre les déclarations et expression régulières et actions correspondantes (le productions).

A. Partie des déclarations

La section Déclarations peut elle même se composer de :

a.Bloc littéral

Cette partie d'un fichier Lex peut contenir :

- Commence par **%{** et se termine par **%}**. Où **%{** et **%}** doivent être placés en début de ligne.
- Contient des déclarations et définitions en C.
- Est copié tel quel dans le fichier Lex.yy.c produit par la commande Lex
- Les définitions et déclarations qu’il contient sont globales au programme produit par Lex.

b.Définitions

Associations d’identificateurs à des expressions régulières. . Ces définitions sont de la forme :

notion ***expression régulière***

Exemple :

separ [\t\n]

```
espace  {separ}+
lettre   [A-Za-z]
chiffre  [0-9]
ident    {lettre}({lettre}|{chiffre})*
nbre     {chiffre}+(\.{chiffre}+)?(E[+\-]?{chiffre}+)?
```

Remarque

Lorsqu'on se réfère à une expression régulière en utilisant un identificateur, celui-ci est mis entre accolades.

c. Conditions de départ

Les conditions de départ permettent de définir plusieurs états de Lex

Exemple

%start etat1 etat2 ….
Où etat1, état2 … sont les états possibles de Lex

B. Partie des productions

Contient deux parties

Partie gauche :

- spécification des expressions régulières reconnues
- pour une chaîne de lettres et de chiffres, les guillemets peuvent être omis
- identificateurs d'expressions mis entre accolades

Partie droite :

- actions exécutées lorsque unités Lexicales reconnues
- actions définies avec syntaxe C
- si scanner appelé par parser YACC, alors :
- les attributs de l'unité Lexicale reconnue doivent être déposés dans **yylval**
- l'unité Lexicale reconnue doit être retournée

Exemple

```
{ \t\n}{ /* pas d 'action */ }
``<=``{return(PPE);}
``<>``{return(DIF);}
``<``   {return(PPQ);}
``=``  {return(EGA);}
``>=``{return(PGE);}
``>``   {return(PGQ);}
``:=``  {return(AFF);}
 if       {return(si);}
 then   {return(alors);}
else     {return(sinon);}
{ident}{yylval = yytext; return(id);}
{nbre}{yylval = yytext; return(nb);}
```

Remarques

- En cas de conflit, Lex choisit toujours la règle qui produit le plus long lexème.

  prog action1
  program action2

La deuxième règle sera choisie.

- Si plusieurs règles donnent des lexèmes de mêmes longueurs, Lex choisit la première.

  prog action1
  [a-z]$^+$ action2 La première règle sera choisie.
- Si aucune règle ne correspond au flot d'entrée, Lex choisit sa règle par défaut implicite :
  .|\n {ECHO} recopie le flot d'entrée sur le flot de sortie

C. Section Procédures auxiliaires

Section optionnelle qui permet de :
- définir toutes les fonctions utilisées dans les actions associées aux expressions reconnues
- définir (si nécessaire) le programme principal (main()).

4. Exemples

Ce premier exemple compte le nombre de voyelles, consonnes et caractères de ponctuations d'un texte entré au clavier.

```
%{
 int nbVoyelles, nbConsonnes, nbPonct;
%}
consonne      [b-df-hj-np-tv-xz]
ponctuation   [,;:?!\.]
%%
[aeiouy]        nbVoyelles++;
{consonne}      nbConsonnes++;
{ponctuation}   nbPonct++;
.|\n            // ne rien faire
%%
main()
{nbVoyelles = nbConsonnes = nbPonct = 0;   yylex();
printf("Il y a %d voyelles, %d consonnes et %d ponctuations.\n",nbVoyelles, nbConsonnes,
nbPonct); }
```

5. Variables et fonctions prédéfinies

- **char yytext[ ]** : tableau de caractères qui contient la chaîne d'entrée qui a été acceptée.
- **int yyleng** : longueur de cette chaîne.
- **int yylex()** : fonction qui lance l'analyseur (et appelle `yywrap()`).
- **yylval:** retourne la valeur associée à l'unité lexicale reconnue**.**
- **ECHO** afficher l'unité lexicale reconnue (équivalente à **printf("%s",yytext);**)
- **FILE *yyout:** fichier de sortie.
- **FILE *yyin:** fichier d'entrée
- **int yywrap()**: fonction toujours appelée en fin de flot d'entrée. Elle ne fait rien par défaut, mais l'utilisateur peut la redéfinir dans la section des fonctions auxiliaires. **`yywrap()`** retourne 0 si l'analyse doit se poursuivre (sur un autre fichier d'entrée) et 1 sinon.
- **unput(char c)** : remet le caractère dans le flot d'entrée.
- **int yylineno** : numéro de la ligne courante.
- **yymore()**: fonction qui concatène la chaîne actuelle **yytext** avec celle qui a été reconnue avant

- **yyless()**: fonction admettant un entier comme argument, yyless(k>0) :
  - supprime les **(yyleng-k)** derniers caractères de **yytext**, dont la longueur devient alors k
  - recule le pointeur de lecture sur le fichier d'entrée de **(yyleng-k)** positions, les caractères supprimés de **yytext** seront donc considérés pour la reconnaissance des prochaines unités
- **yyterminate()** : fonction qui stoppe l'analyseur .

6. Exemple

L'exemple suivant insert le numéro de ligne à chaque ligne dans un fichier.

```
%{ int yylineno;
%}
%%
^(.*)\n printf("%4d\t%s", ++yylineno, yytext);
%%
int main(int argc, char *argv[])
{
yyin = fopen(argv[1], "r");
yylex();
fclose(yyin);
}
```

Etude de cas…