Compilation

2- Méthodes d'Analyse Lexicale

Obtenir un analyseur lexical

- On a vu un automate qui accepte ou rejette une chaîne de caractères.
- La tâche d'un analyseur lexical n'est pas juste d'accepter ou rejeter des chaînes de caractères.
- Il doit trouver la plus longue sous-chaîne de l'entrée correspondant à une expression régulière et retourner l'unité correspondante.
- Mais on peut adapter un automate pour garder une trace de la plus longue sous-chaîne acceptée

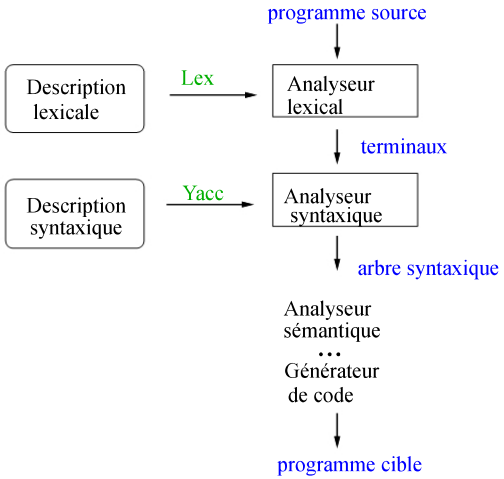
Obtenir un analyseur lexical

- Méthode 1
  - Donner l'automate d'états finis qui reconnaît les unités lexicales (voir transparent suivant)
  - Implémenter *getunite()* en traduisant cet automate
    1) Partir de l'état initial
    2) A l'état final, retourner l'unité lexicale reconnue
    3) Si l'on veut retourner toutes les unités lexicales par un seul appel à *getunite()* alors revenir à 1)
- Méthode 2
  - Utiliser Lex

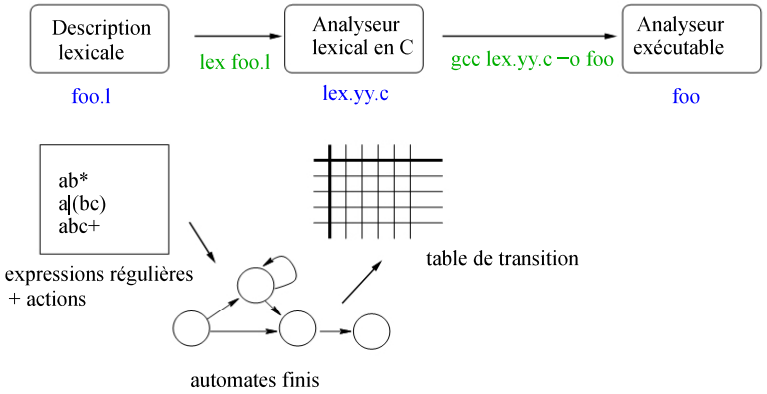
Méthode 2

- Utiliser un générateur d'analyseurs lexicaux
  - Il prend en entrée un ensemble d'expressions régulières
  - Il génère le code source de l'analyseur lexical
  - L'analyseur généré prend en entrée le programme source à analyser

Lex dans le processus de compilation



Fonctionnement de Lex



Organisation de la description lexicale Lex

%{

Déclarations pour le compilateur

%}

Définitions régulières

%%

Règles

%%

Fonctions supplémentaires

Organisation de la description lexicale Lex

Includes / Déclarations / définitions pour le programme C

```
%{
    int i;
%}
```

Définitions régulières

```
IDENT       [a-zA-Z][a-zA-Z0-9]*

%%
```

Expressions régulières et actions associées

```
    {IDENT}    printf(...);

%%
```

Fonctions et programme principal

```
int main () {

}
```

Pseudo algorithme de la fonction yylex

```
int yylex() {
   while ( non fin de fichier ) {
        trouver le plus long préfixe correspondant à une des expressions régulières
        si le lexème correspond à exp-reg-1 alors
                action-1
        sinon si le lexème correspond à exp-reg-2 alors
                action-2
        sinon
                ...
        sinon /* action par défaut */
                afficher le premier caractère sur stdout
        éliminer les caractères utilisés de l'entrée
   }
   return 0;
}
```


Organisation de la description lexicale Lex

Définitions régulières :

- De la forme

  <identificateur> *<expression régulière>*
- Les identificateurs ainsi définis peuvent être utilisés dans les règles et dans les définitions subséquentes ; il faut alors les encadrer par des **accolades**

Syntaxe des expressions régulières

Caractères simples

x le caractère x
. (point) tout caractère sauf newline
\n newline.
Autres car. spéciaux: \t, \r

Classes de caractères

[xyz] l'un des car. x, y, z (équivalent: x|y|z)
[A-Z] les car. A...Z
[^A-Z] tout car. sauf A...Z

Opérateurs

rs concaténation
r|s alternative
r*, r+ r{n} répétition (0 fois ou plus, 1 fois ou plus, n fois)
… et beaucoup plus

Variables et fonctions prédéfinies de Lex

Variables prédéfinies :

yyin fichier de lecture (par défaut : stdin)
yyout fichier d'ecriture (par défaut : stdout)
yytext dernière chaîne de caractère reconnue
yyleng longueur de yytext

Fonctions prédéfinies :

yylex() Appel de Lex, actif jusqu'au premier return ou fin de fichier
yywrap() Pour traiter plusieurs fichiers. Retourne 1 si un seul fichier à analyser (comportement par défaut);

Exemple 1

```
chiffre     [0-9]
entier      {chiffre}+
exposant    [eE][+-]?{entier}
reel        {entier}("."{entier})?{exposant}?
```

Exercice 1

Donner l'expression régulière Lex des **identificateurs**

Exemple 2

Que fait le programme Lex suivant :

```
%%
[ \t]+$         ;
[ \t]           printf(" ");
```


Exemple 3

```
%option noyywrap
%{
        int num_lignes = 0, num_cars = 0;
%}
%%
 \n      ++num_lignes; ++num_cars;
  .      ++num_cars;
%%
 main()
          {
          yylex();
           printf( "nombre de lignes = %d, nombre de
                   cars = %d\n", num_lignes,
          }        num_cars );
```

Exemple 4

```
%option noyywrap
%{
   #include "unites.h"
%}
   lettre [A-Za-z]
   chiffre [0-9]
   lettreouchiffre {lettre}|{chiffre}
%%
   if { return SI; }
   then { ECHO, return ALORS; }
   else { return SINON; }
   . . .
  {lettre}{lettreouchiffre}* { return IDENT; }
  ("+"|"-")?{chiffre}+("."{chiffre}+)? { printf("%s", yytext);
                                         return  NOMBRE; }
```

Exemple 4 (suite)

```
#include <stdio.h>
#include "unites.h"
int main(void) {
int unite;
do {
    unite = yylex();
    printf(" (unite: %d", unite);  // unité
          printf(" '%s'", yytext);  // lexème
    printf(")\n");
} while (unite != 0);
   return 0;
}
```


Exercice 2

- Donner la description lex qui reconnaît les unités suivantes :
  - WORDs, comme 'zone' et 'type' ...
  - FILENAMEs, comme '/etc/bind/db.root' ...
  - QUOTEs, "
  - LBRACEs, {
  - RBRACEs, }
  - SEMICOLONs, ;
- Quel est le résultat de l'application de l'analyseur généré au source suivant :

```
logging {
            category lame-servers { null; };
            category cname { null; };
};

zone "." {
            type hint;
            file "/etc/bind/db.root";
};
```


Exercice 3

```
<html>
<h1>Lists</h1>
<h2>Unordered Lists</h2>

<ul>
<li> first element </li>
<li> second element </li>
</ul>

<p>
In between a paragraph with text in bold and <i>italics</i> and in
<b><i>bold italics</i></b>.
</html>
```


Exercice 4

Un document LATEX est inclus entre

\documentclass{article}

\begin{document}

...

\end{document}

Le titre d'une section resp. sous-section est inclus entre \section{...} resp. \subsection{...}. Pour créer une liste numérotée, on utilise l'environnement

\begin{enumerate}

\item ....

\end{enumerate}

Pour une liste non numérotée, on utilise le mot-clé itemize.

Un nouveau paragraphe commence tout simplement avec une ligne vide.

testez votre programme en invoquant latex avec le fichier généré.

Expressions régulières prédéfinies de Lex

Symbole	Signification
x	Le caractere 'x'
.	N'importe quel caractere sauf \n
[xyz]	Soit x, soit y, soit z
[^bz]	Tous les caracteres, SAUF b et z
[a-z]	N'importe quel caractere entre a et z
[^a-z]	Tous les caracteres, SAUF ceux compris entre a et z
R*	Zero R ou plus, ou R est n'importe quelle expression reguliere
R+	Un R ou plus
R?	Zero ou un R (c'est-a-dire un R optionnel)
R{2,5}	Entre deux et cinq R
R{2,}	Deux R ou plus
R{2}	Exactement deux R
"[xyz\"foo"	La chaine '[xyz"foo'
{NOTION}	L'expansion de la notion NOTION definie plus haut
\X	Si X est un 'a', 'b', 'f', 'n', 'r', 't', ou 'v', represente l'interpretation ANSI-C de \X.
\0	Caractere ASCII 0
\123	Caractere ASCII dont le numero est 123 EN OCTAL
\x2A	Caractere ASCII en hexadecimal
RS	R suivi de S
R|S	R ou S
R/S	R, seulement s'il est suivi par S
^R	R, mais seulement en debut de ligne
R$	R, mais seulement en fin de ligne
<<EOF>>	Fin de fichier

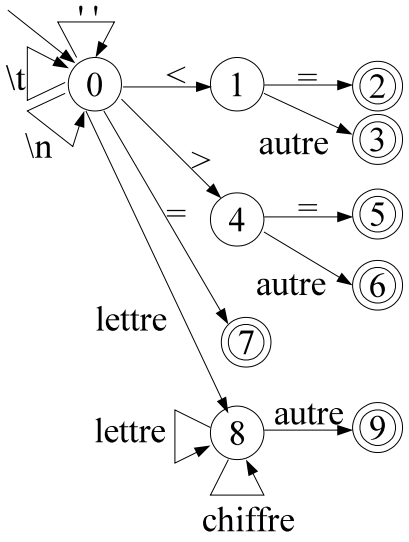
Méthode 1

Exemple :

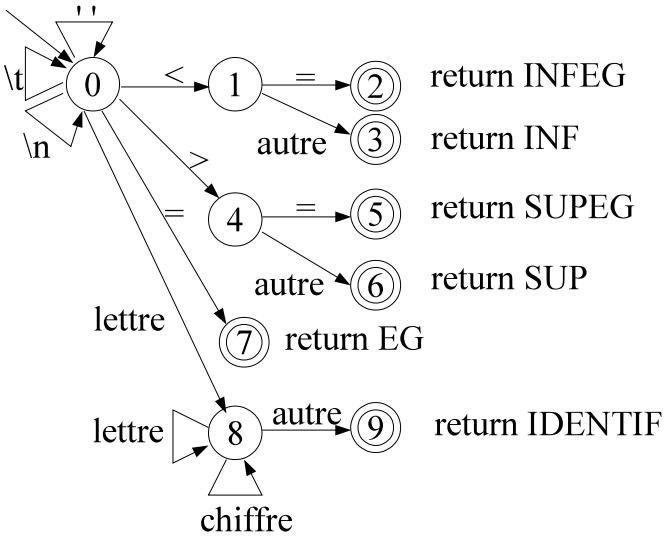
Donner l'automate qui reconnaît les chaînes suivantes

- Symboles de comparaison < , >
- Symboles de comparaison <= , >=
- Symbole d'affectation =
- Identificateurs

Méthode 1



Méthode 1



Méthode 1

```
int getunite(void) {
  char caractere;
  int etat = etatInitial;
  while ( ! final[etat]) {
       caractere = lireCar();
       etat = transit[etat][caractere];
  }
  return unite(etat);
}
```


Méthode 1

```
int getunite(void) {
  char caractere;
  int etat = etatInitial;
  while ( ! final[etat]) {
       caractere = lireCar();
       etat = transit[etat][caractere];
  }
  if (etat ==  ...) delireCar();
  return unite(etat);
}
```


Méthode 1

```
int getunite(void) {
  char caractere;
  int etat = etatInitial;
  while ( ! final[etat]) {
       caractere = lireCar();
       etat = transit[etat][caractere];
  }
  if (etat ==  ...) delireCar();
  return unite(etat);
}
```


Méthode 1

Cas de la reconnaissance d'un lexème avec lecture d'un caractère en trop => il faudra alors retourner en arrière avec delireCar avant la prochaine lecture :

```
void delireCar(char c) {
       ungetc(c, stdin);
}
char lireCar(void) {
       return getc(stdin);
}
```

Il est possible de donner une implémentation de *delireCar* et *lireCar* sans utiliser ungetc

Méthode 1

```
int carEnAvance = -1;
void delireCar(char c) {
       carEnAvance = c;
}
char lireCar(void) {
      char r;
      if (carEnAvance >= 0) {
          r = carEnAvance;
          carEnAvance = -1;
      } else
          r = getc(stdin);
     return r
}
```


Méthode 1

```
#define NBR_ETATS
#define NBR_CARS 256
int transit[NBR_ETATS][NBR_CARS];
int final[NBR_ETATS + 1];
void initialiser_final(void) {
     int i, j;
     for (i = 0; i < NBR_ETATS; i++)
               final[i] = 0;
     final[ 2] = INFEG ;
     final[ 3] = DIFF;
     final[ 4] = - INF; // valeur négative pour se
                        // rappeler de la lecture en trop
     final[ 5] = EGAL;
     final[ 7] = SUPEG;
     final[ 8] = - SUP;
     final[10] = - IDENTIF
     final[NBR_ETATS] = ERREUR ;
}
void initialiser_transit(void) {
  // Exercice : a compléter
}
```


Méthode 1

```
for (i = 0; i < NBR_ETATS; i++)
    for (j = 0; j < NBR_CARS; j++)
        transit[i][j] = NBR_ETATS;

transit[0][' '] = 0;
transit[0]['\t'] = 0;
transit[0]['\n'] = 0;
transit[0]['<'] = 1;
transit[0]['='] = 5;
transit[0]['>'] = 6;
for (j = 'A'; j <= 'Z'; j++) transit[0][j] = 9;
for (j = 'a'; j <= 'z'; j++) transit[0][j] = 9;
for (j = 0; j < NBR_CARS; j++) transit[1][j] = 4;
transit[1]['='] = 2;
transit[1]['>'] = 3;
for (j = 0; j < NBR_CARS; j++) transit[6][j] = 8;
transit[6]['='] = 7;
for (j = 0; j < NBR_CARS; j++) transit[9][j] = 10;
for (j = 'A'; j <= 'Z'; j++) transit[9][j] = 9;
for (j = 'a'; j <= 'z'; j++) transit[9][j] = 9;
for (j = '0'; j <= '9'; j++) transit[9][j] = 9;
```


Méthode 1

L'automate permet-il de reconnaître les mots réservés du langage ?

- Reste à modifier légèrement le programme afin de différencier entre les mots réservés et les identificateurs normaux