Chapitre 2 : Analyse lexicale

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 peut également réaliser certaines taches secondaires, une de ces tâches est l'élimination dans le programme source des commentaires et des espaces qui apparaissent sous formes de caractères blanc, tabulation ou fin de ligne. Et gère aussi les numéros de ligne dans le programme source pour pouvoir associer à chaque erreur rencontrée par le numéro de ligne correspondant.

1. Unité lexicale

Définition 1: Une unité lexicale est une suite de caractères qui a une signification collective

Exemple : Les chaines <,>,= sont des opérateurs relationnels, l'unité lexicale est OPREL par exemple

Définition 2: Un modèle est une règle associe à unité lexicale qui décrit l'ensemble des chaines du programme qui peuvent correspondre à cette unité lexicale

Définition 3 : on appelle lexème toute suite de caractère du programme source qui concorde avec le modèle d'une unité lexicale.

Exemple : L'unité lexicale IDENT (identificateur) en C a pour modèle toute suite non vide de caractère composé de chiffre, lettre, ou des symboles et qui commence par une lettre

Exemple de lexème pour cette unité lexicale sont a ; b ; montant, tot1….

L'unité lexicale NOMBRE (entier signé) a pour modèle : toute suite non vide de chiffres précédée éventuellement d'un seul caractère parmi {+,-}. Lexèmes possibles : -12, 5697

Pour décrire un modèle d'une unité lexicale on utilisera **les expressions régulières.**

2. Expression régulière

Rappel

Une expression régulière est une formule permettant de désigner un ensemble de chaines de caractères construites à partir d'un alphabet ∑. On appelle cet ensemble de chaîne de caractère un langage.

- Une expression régulière est une notation pour décrire un langage régulier.
- On appelle **alphabet** un ensemble fini non vide ∑ de symboles.
- On appelle **mot** toute séquence finie d'éléments de ∑.
- On note ε le mot vide.

- On note ∑* ensemble infini contenant tous les mots possibles sur ∑.
- On note ∑+ ensemble des mots non vides que l'on peut former sur ∑, c'est-à-dire ∑+=∑*-{ε}
- On note| m| la longueur du mot m c'est-à-dire le nombre de symboles de ∑ composant le mot.

Exemple :

1- Soit l'alphabet ∑= {a,b,c}

aaba, bbbacbb, c, ε, ca sont des mots de ∑*.

2 - Soit C={a,b}

- l'expression régulière a |b dénote l'ensemble {a,b}
- L'expression régulière (a|b) (a|b) dénote {aa,ab,ba,bb} l'ensemble de toutes les chaines de a et b de longueur 2.
- L'expression régulière a* dénote l'ensemble de toutes les chaines formées d'un nombre quelconque de a : c. à. d {ε,a,aa,aaa, ……………..}
- On peut décrire un identificateur comme : Identificateur = lettre (lettre | chiffre|sep)* tel que : lettre= [a-zA-Z] Chiffre=[0-9] Sep=_.

Définition 4: On appelle **langage** sur un alphabet ∑ tout sous ensemble de ∑*.

Exemple : Soit l'alphabet ∑={a,b,c}

Soit L_1 l'ensemble des mots de ∑* ayant autant de a que de b et c. L1 est le langage infini { ε, c ,ccc,….ab,abccc,…}.

Définition 5: un langage régulier L sur un alphabet ∑ est défini récursivement de la manière suivante :

- {ε} est un langage régulier sur ∑
- Si a est une lettre de ∑, {a}est un langage régulier sur ∑.
- Si R est un langage régulier sur ∑, alors R^n et R^* sont des langages réguliers sur ∑.
- Si R1 et R2 sont des langages réguliers sur ∑ , alors R1U R2 et R1R2 sont des langages réguliers.

Les langages réguliers se décrivent très facilement par une expression régulière (ER).

Définition 6 : 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 :

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

. . .

$d_n \rightarrow r_n$

Où chaque d_i est un nom distinct et chaque r_i est une ER.

Exemple :

lettre → A | B | . . . | Z | a | b | . . . | z

chiffre → 0 | 1 | . . . | 9

id → lettre (lettre | chiffre)*

chiffres → chiffre chiffre *

frac → . chiffres | ε

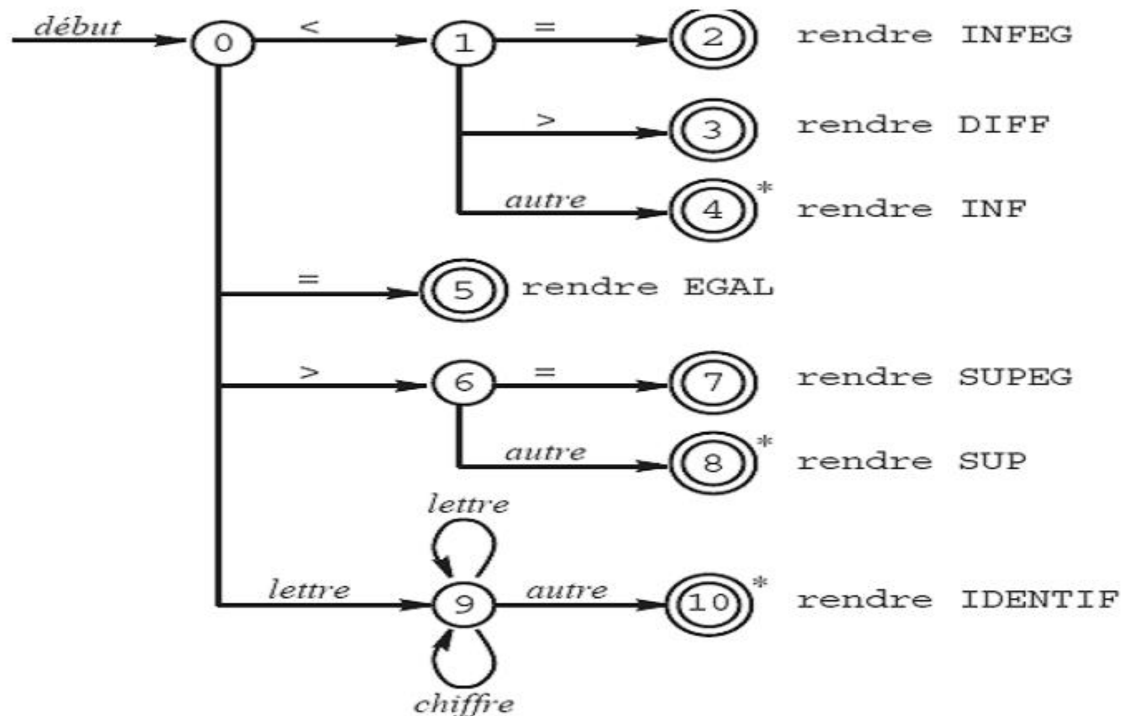
3. Reconnaissance des unités lexicales

Un reconnaisseur pour un langage est un programme qui prend en entrée une chaine x et répond oui si x est un mot du langage et non autrement. On compile une expression régulière en un reconnaisseur en construisant un diagramme de transition généralisé appelé **automate fini.**

Exemple :

Pour illustrer cette section nous allons nous donner comme exemple le problème de la reconnaissance des unités lexicales INFEG, DIFF, INF, EGAL, SUPEG, SUP, IDENTIF, respectivement définies par les expressions régulières <=, <>, <, =, >=, > et lettre(lettre/chiffre)*.

Par exemple, la figure suivante montre les diagrammes traduisant la reconnaissance des unités lexicales INFEG, DIFF, INF, EGAL, SUPEG, SUP et IDENTIF.

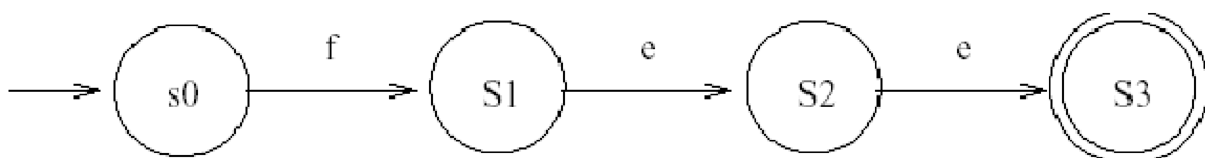


Rappel

Présentation de l'automate :

Un automate est souvent représenté par un graphe orienté dont les sommets sont les états et les arêtes étiquetées correspondent aux transitions, on distingue les états d'acceptations que l'on entourera d'un cercle et l'état initial.

Exemple



Sur ce dessin, l'etat initial est S0, S3 est l'etat final signifiant qu'on a reconnut le mot « fee ». Si on est en train de lire un autre mot, une des transitions ne pourra pas s'effectuer, on rejettera le mot.

Remarque : Le langage défini par un AFN (Automate Fini Non déterministe) est l'ensemble des chaines d'entrée qu'il accepte.

Un AFN accepte une chaine d'entrée x si et seulement s'il existe un certain chemin dans le graphe de transition entre l'état initial et l'état final.

Les automates fini peut être déterministes ou non :

- **Les automates finis non déterministe (AFND):**

Un automate fini non déterministe (AFND) est un quintuplet A=(Q , ∑, δ, q0, F), où

-Q est un ensemble fini d'états

- ∑ est un alphabet

- δ : Q x ∑ u { ε } →Q est la fonction de transition

-q0 est l'état initial

- F est l'ensemble des états terminaux ou finaux.

Un automate non-déterministe est un automate qui présente la particularité suivante :

- Plusieurs arcs reconnaissant le même caractère peuvent « sortir » du même état.
- Il peut y avoir des transitions, notées ε, qui ne correspondent à une aucune reconnaissance de caractères.
- Le langage reconnu par un automate (Q , ∑,δ,q0, F) est l'ensemble des mots w tels qu' il existe : q0--→wF.

➢ **Automate fini déterministe (AFD).**

L'automate est dit déterministe lorsque :

1. la fonction δ associe à chaque couple (état, lettre) un état unique

2. aucun état n'a de ε transition

Remarque : Le passage d'un AEFND vers un AEFD a pour but :

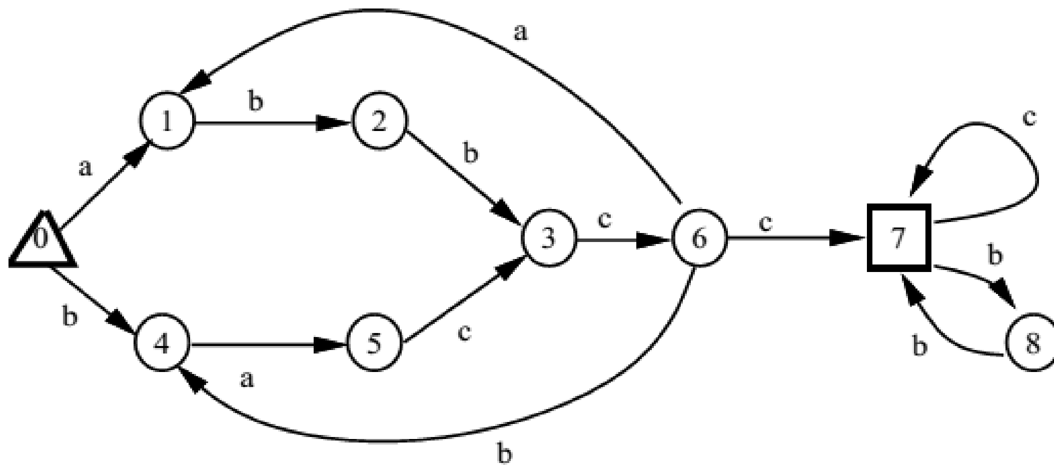
La complexité de la reconnaissance d'un mot (ex : analyse lexicale) par un AEFND est exponentielle. De ce fait cela influe directement sur le temps de reconnaissance d'un mot. C'est pour ça il est préférable de transformer l'AEFND à un AEFD.

4. Analyseur lexical

Le rôle d'un analyseur lexical est la reconnaissance des unités lexicales. Une unité lexicale peut être exprimée sous forme de définitions régulières.

Dans la théorie des langages on définit des automates qui sont permettant la reconnaissance de mots. Un langage régulier est reconnu par un automate fini.

Exemple : Le langage régulier décrit par l'ER $(abbc|bacc)^+$ c(c|bb)* est reconnu par l'automate :



qui s'écrit également

État	a	b	c
0	1	4	
1		2	
2		3	
3			6
4	5		
5			3
6	1	4	7
7		8	7
8		7	

Un langage régulier et donc une ER peut être reconnu par un automate. Donc pour écrire un analyseur lexical de notre programme source, il suffit d'écrire un programme simulant l'automate. Lorsqu'une unité lexicale est reconnue elle envoyée à l'analyseur syntaxique, qui la traite, puis repasse la main à l'analyseur lexical qui lit l'unité lexicale suivante dans le programme source. Et ainsi de suite, jusqu'à tomber sur une erreur ou jusqu'a ce que le programme source soit traité en entier.

Un exemple de générateur d'analyseur lexical : LEX

Un Générateur d'analyseur lexical vise à :

Prend en entrée la définition des unités lexicales.

Produit un automate fini déterministe permettant de reconnaitre les unités lexicales.

L'automate est produit sous la forme d'un programme C.

→Lex accepte en entrée des spécifications d'unités lexicales sous forme de définitions régulières et produit un programme écrit dans un langage de haut niveau (le langage C) qui une fois compilé, reconnait ces unités lexicales (ce programme est donc un analyseur Lexical)

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

```
%{

Partie 1 : déclarations pour le compilateur C

%}

Partie 2 : définitions régulières

%%

Partie 3 : règles

%%

Partie 4 : fonctions C supplémentaires
```

Partie 1 : les déclarations

Cette partie d'un fichier Lex peut contenir :

Un code écrit dans le langage cible, encadré par %{ et %}, qui se retrouvera au début du fichier synthétisé par Lex. C'est ici que l'on va spécifier les fichiers à inclure. Lex recopie tel quel tout ce qui est écrit entre ces deux signes, qui devront toujours être placés en début de ligne.

La partie 2 : Des définitions régulières définissant des notions non terminales, telles que lettre, chiffre et nombre. Ces spécifications sont de la forme :

```
notion          expression_reguliere
```

Exemple :

```
%{

#include <stdio.h>
#include <stdlib.h>

%}
/* Expressions regulieres */
```

Chiffre [0,9]

Lettre [a-z A-Z]

entier {chiffre}+

ident {lettre}({lettre}|{chiffre})*

%%

Partie 3 : les règles

Cette partie sert à indiquer à Lex ce qu'il devra faire lorsqu'il rencontrera telle ou telle notion. Celle-ci peut contenir :

Des productions de la forme :

```
expression_reguliere  action
```

où expressionRégulière est écrit au début de la ligne ; action est un morceau de code source C, qui sera recopié tel quel, au bon endroit, dans la fonction yylex().

La fonction yylex()qui doit être appelée pour utiliser l'analyseur lexical :

- analyse séquentiellement un fichier d'entrée

- retourne 0 lorsqu'elle rencontre une fin de fichier

- effectue des opérations spécifiées par le programme Lex, lorsqu'une unité lexicale est reconnue

Enfin, la variable `yytext` désigne dans les actions les caractères acceptés par `expression_régulière`. Il s'agit d'un tableau de caractère de longueur `yyleng`.

Exemple :

```
%{

#include <stdio.h>
#include <stdlib.h>

%}
/* Expressions regulieres */
```

chiffre [0,9]

lettre [a-z A-Z]

entier {chiffre} +

ident {lettre}({lettre}|{chiffre})*

%%

«:= » {return AFF ;}

« <= » {return OPREL ;}

if|IF|If|iF {return MC_IF ;}

{entier} {return NB ;}

{ident} {return ID ;}

%%

Partie 4 : `fonctions C supplémentaires`

Tu peux mettre dans cette partie facultative tous le code que tu veux. Si tu ne mets rien, Lex considère que c'est juste :

```
main() {
  yylex();
}
```

Remarque : Les expressions réguliers Lex

<table>
<tr><th>Expression</th><th>Signification</th><th>Exemple</th></tr>
<tr><td>c</td><td>tout caractère *c* qui n'est pas opérateur ou métacaractère</td><td>a</td></tr>
<tr><td>\c</td><td>caractère littéral c (lorsque c est un métacaractère)</td><td>\+ \.</td></tr>
<tr><td>"s "</td><td>chaîne de caractères</td><td>"bonjour "</td></tr>
<tr><td>.</td><td>n'importe quel caractère, sauf retour à la ligne (\n)</td><td>a.b</td></tr>
<tr><td>^</td><td>l'expression qui suit ce symbole débute une ligne</td><td>^abc</td></tr>
<tr><td>$</td><td>l'expression qui précède ce symbole termine une ligne</td><td>abc$</td></tr>
<tr><td>[s]</td><td>n'importe quel caractère de s</td><td>[abc]</td></tr>
<tr><td>[^s]</td><td>n'importe quel caractère qui n'est pas dans s</td><td>[^xyz]</td></tr>
<tr><td>r*</td><td>0 ou plusieurs occurrences de r</td><td>b*</td></tr>
<tr><td>r+</td><td>1 ou plusieurs occurrences de r</td><td>a+</td></tr>
<tr><td>r?</td><td>0 ou 1 occurrence de r</td><td>d?</td></tr>
<tr><td>r{m}</td><td>m occurrences de r</td><td>e{3}</td></tr>
<tr><td>r{m,n}</td><td>entre m et n occurrences de r</td><td>f{2,4}</td></tr>
<tr><td>r_1r_2</td><td>r_1 suivie de r_2</td><td>ab</td></tr>
<tr><td>$r_1|r_2$</td><td>r_1 ou r_2</td><td>c|d</td></tr>
<tr><td>r_1/r_2</td><td>r1 si elle est suivie de r2</td><td>ab/cd</td></tr>
<tr><td>(r)</td><td>r</td><td>(a|b)?c</td></tr>
<tr><td><x>r</td><td>r si Lex se trouve dans l'état x</td><td><x>abc 20</td></tr>
</table>

Bibliographie

Mohamed BOUHDADI, « Compilation théorie des langages », Support du cours, Université Mohammed V - Agdal,Faculté des Sciences, Rabat.

SAOUDI Lalia, (2008), « Analyse lexicale», Support du cours.

A.Aho, R.Sethi, J.Ullman, (1991), « Compilateurs principes techniques et outils », InterEditions, 1991.