كلـيـــــــــة العلوم
السملالية - مراكش
FACULTE DES SCIENCES
SEMLALIA-MARRAKECH
MARRAKECH
جامعة القاضي عياض
UNIVERSITÉ CADI AYYAD

Département Informatique

LF - SMI – S6

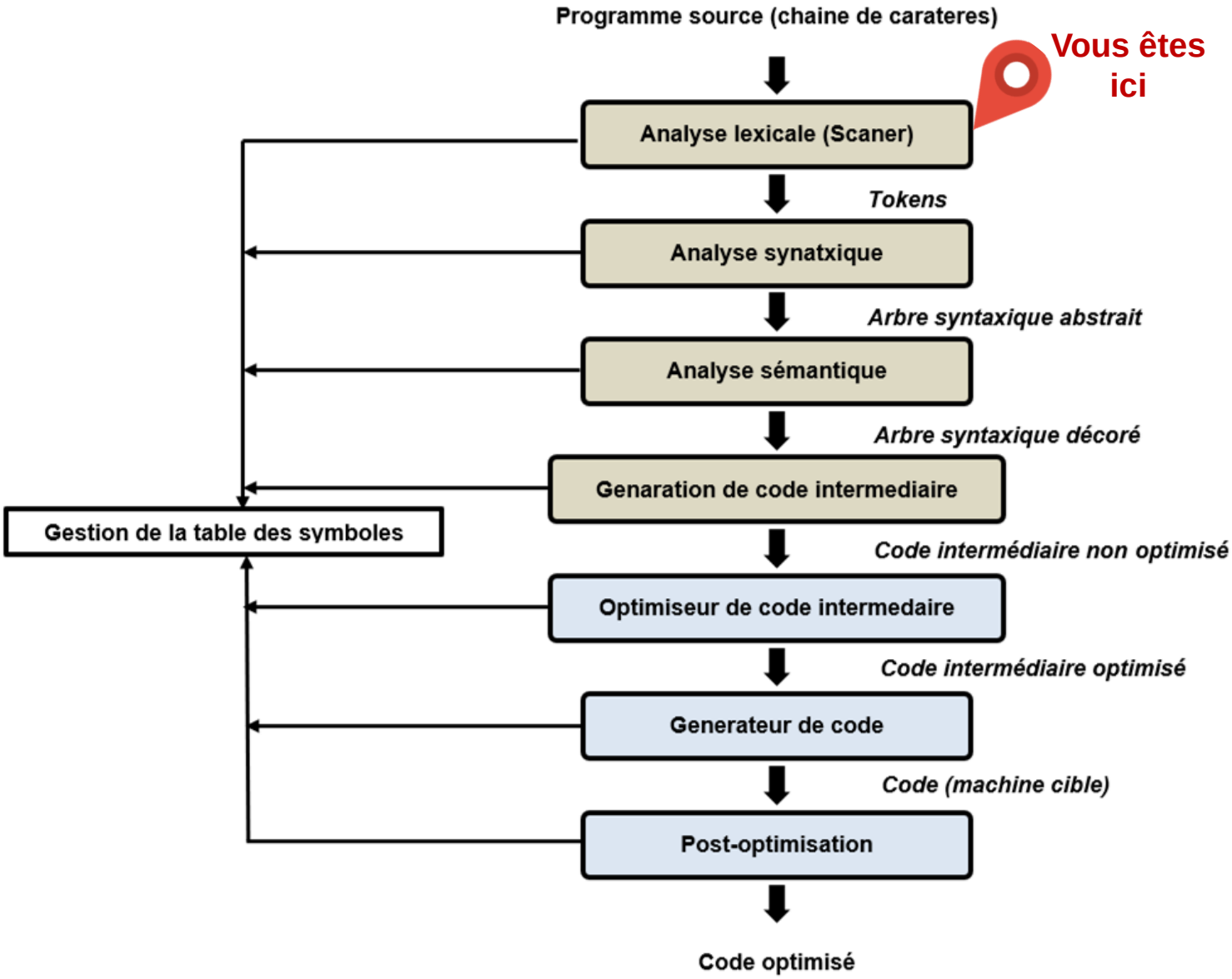
Compilation

Pr. Aimad QAZDAR

aimad.qazdar@uca.ac.ma

14 avril 2021

Rappel : Phases d'un compilateur



Chapitre 2 :

Analyse lexicale

Plan

- Introduction
- Unité lexicale
- Expressions régulières
- Diagramme de transition
- Automates finis
- Table de transition
- Générateurs automatiques d’analyseurs lexicaux

Introduction

- L'analyse lexicale représente la première phase de la compilation.
- Le code source, qui se présente comme un flot de caractères
- L'analyse lexicale reconnait les « **mots** » avec lesquels les phrases sont formées
- Ces mot nous les appelons des **unités lexicales (tokens)**,
- Les unités lexicales seront livré à la phase suivante : l'analyse syntaxique.

Introduction

- Les analyseurs lexicaux assurent aussi certaines fonctionnes telles que :
  - La suppression des caractères de décoration (blancs, indentations, fins de ligne, etc.) et celle des commentaires (généralement considères comme ayant la même valeur qu'un blanc),
  - L'interface avec les fonctions de lecture de caractères, à travers lesquelles le code source est acquis,
  - La gestion des fichiers et l'affichage des erreurs, etc

Unité Lexicale

- Les principales sortes d'unités lexicales sont :
  - **Les caractères spéciaux simples** : +, =, etc.
  - **Les caractères spéciaux doubles** : <=, ++, etc.
  - **Les mots-clés** : if, while, etc.
  - **Les constantes littérales** : 159, -4, etc.
  - **Les identificateurs** : i, note_etudiant, etc.

Unité Lexicale

- On doit distinguer 4 notions : **l'*unite lexicale***, ***lexème***, ***attribut***, ***modele***
- **L'*unite lexicale***, représentée généralement par un code conventionnel.
  - Exemples

=	**EGAL**
<=	**INFEGAL**
+	**PLUS**
++	**PLUSPLUS**
if	**SI**
while	**TANTQUE**
159	**NOMBRE**
-4	**NOMBRE**
i	**IDENTIF**
note-etudiant	**IDENTIF**

Unité Lexicale

- Le ***lexème*** :
  - La chaine de caractères correspondante.
  - Pour les exemples précédents, les lexèmes correspondants sont : "+", "=", "<=", "++", "if", "while", "159", "-4", "i" et "note_etudiant "
- Un ***attribut*** :
  - Il dépend de l'unité lexicale en question,
  - Il permet de compléter l'unité lexicale .
  - Seules les "i" et "note_etudiant" de l'exemple précèdent ont un attribut ;
  - Pour un nombre, il s'agit de sa valeur (159, *-4*) ;
  - Pour un identificateur, il s'agit d'un renvoi à une table dans laquelle sont placés tous les identificateurs rencontrés.

Unité Lexicale

- Le ***modèle*** :
  - Spécifier l'unité lexicale.
  - Il existe des **moyens formels pour définir les modèles** ;
  - Descriptions informelles comme :
    - Pour les caractères spéciaux simples et doubles et les mots réserves, **le lexème et le modèle coïncident**,
    - Le modèle d'un nombre est « **une suite de chiffres, éventuellement précédée d'un signe** »,
    - Le modèle d'un identificateur est « **une suite de lettres, de chiffres et du caractère '_' , commençant par une lettre** ».

Expressions régulières

- Les **expressions régulières** sont une importante notation pour **spécifier formellement des modèles**.
- Nouveaux concepts :
  - **Alphabet**,
  - **Mot**,
  - **Langage**

Alphabet

- Un **alphabet** est un ensemble de symboles.
- Exemples :
  - {0,1},
  - {A,C,G, T},
  - L'ensemble de toutes les lettres,
  - L'ensemble des chiffres,
  - Le code ASCII,
  - etc.

Les caractères blancs (les espaces, les tabulations et les marques de fin de ligne) ne font généralement pas partie des alphabets

Mot

- Un **mot** (on dit aussi **chaine**) sur un alphabet Σ est une séquence finie de symboles de Σ.
- Exemples :

Alphabet	Chaine
{0,1}	00011011
{A,C,G, T}	ACCAGTTGAAGTGGACCTTT
L'ensemble de toutes les lettres	Bonjour
L'ensemble des chiffres	2001

La chaine vide ε, ne comportant aucun caractère

Mot

- Si x et y sont deux chaines, la concaténation de x et y, notée xy, est la chaine obtenue en écrivant y immédiatement après x.
- Par exemple, la concaténation des chaines « Bon » et « jour » est la « chaine Bonjour ».
- Si x est une chaine, on définit $x^0 = \varepsilon$ et, pour $n > 0$, $x^n = x^{n-1}x = xx^{n-1}$. On a donc $x^1 = x$, $x^2 = xx$, $x^3 = xxx$, etc.

Langage

- Un **langage** sur un alphabet Σ est un ensemble de chaines construites sur Σ.
- Exemples :
  - Ø : le langage vide
  - {ε} : le langage réduit a l'unique chaine vide.
  - L'ensemble des nombres en notation binaire,
  - L 'ensemble des chaines ADN,
  - L'ensemble des mots de la langue française,
  - etc.

Opérations sur les langages

- Les opérations sur les langages permettent définir les expressions régulières : **Union**, **Concaténation**, **Etoile Kleene**, **Fermeture positive**
- Union de L et M (**L U M**): { x | x ∈ L ou x ∈ M}
- Concaténation de L et M (**LM**) : { xy | x ∈ L et y ∈ M}
- Etoile Kleene (appelé aussi la fermeture de Kleene ou fermeture itérative) (**L***) : $\{x_1x_2\ldots x_n \mid x_i \in L, n \in N \text{ et } n >= 0\}$
- Fermeture positive de L **(L^+)** : $\{x_1x_2\ldots x_n \mid x_i \in L, n \in N \text{ et } n > 0\}$

Opérations sur les langages

- Soit les deux alphabets L et C définie respectivement comme suit L = {A,B,….Z, a, b,…., z} et C = {0, 1,…,9},
- En considérant qu'un caractère est la même chose qu'une chaine de longueur un (1),
- Nous pouvons voir L et C comme des langages, formés de chaines de longueur un.
- **Qu'est ce que représente les langages suivants et illustré chaque langage par un exemple :**
  1. L U C
  2. LC
  3. L^4
  4. L^*
  5. C^+
  6. L(L U C)

Opérations sur les langages

1. **L U C**
   - L'ensemble des lettres et des chiffres
2. **LC**
   - L'ensemble des chaines formées d'une lettre suivie d'un chiffre
3. **L^4**
   - L'ensemble des chaines de quatre lettres,
4. **L^***
   - L'ensemble des chaines faites d'un nombre quelconque de lettres ; ε en fait partie
5. **C^+**
   - L'ensemble des chaines de chiffres comportant au moins un chiffre,
6. **L(L U C)**
   - L'ensemble des chaines de lettres et chiffres commençant par une lettre.

Expression régulière.

- Une expression régulière r sur Σ est une formule qui définit un langage L(r) sur Σ, de la manière suivante :
  1. ε est une expression régulière qui définit le langage {ε}
  2. Si a ∈ Σ, alors a est une expression régulière qui définit le langage {a}
  3. Soient x et y deux expressions régulières, définissant les langages L(x) et L(y). Alors :
     - (x)|(y) est une expression régulière définissant le langage L(x) U L(y)
     - (x)(y) est une expression régulière définissant le langage L(x)L(y)
     - $(x)^*$ est une expression régulière définissant le langage $(L(x))^*$
     - (x) est une expression régulière définissant le langage L(x)

NB :

- **Les opérateurs *, concat et | sont associatifs à gauche**
- **priorite(*) > priorite(concat) > priorite(|)**

***a* == *b* == *c* signifie quoi???**

Définitions régulières

- Les expressions régulières se construisent à partir d'autres expressions régulières : **définitions régulières.**

  $d_1 \rightarrow r_1$

  $d_2 \rightarrow r_2$

  . . .

  $d_n \rightarrow r_n$

- Où chaque d_i est une chaine sur un alphabet disjoint de Σ , distincte de d_1, d_2,…, d_{i-1}, et chaque r_i une expression régulière sur $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$.

Définitions régulières

- Exemple de quelques définitions régulières qui définissent les identificateurs et nombres du langage Pascal :

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

  chiffre → 0 | 1 | ... | 9

  identificateur → lettre (lettre | chiffre)*

  chiffres → chiffre chiffre*

  fraction-opt → . chiffres | ε

  exposant-opt → (E (+ | - | ε) chiffres) | ε

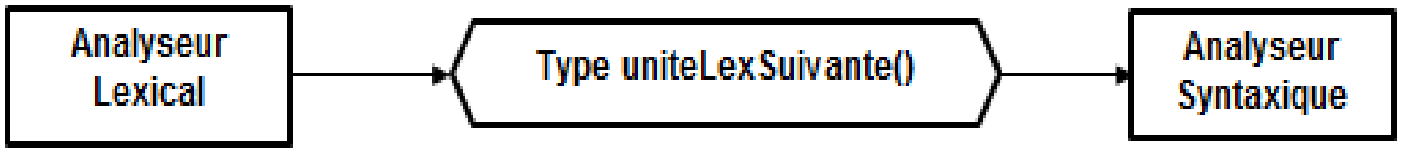
  nombre → chiffres fraction-opt exposant-opt

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) U { ε },
  - Si c_1, c_2… c_k sont des caractères, l'expression régulière $c_1|c_2|$ … $|c_k$ peut se noter $[c_1c_2 \ldots c_k]$,
  - A l'intérieur d'une paire de crochets comme ci-dessus, l'expression c_1-c_2 désigne la séquence de tous les caractères c tels que $c_1 \leq c \leq c_2$.
- lettre → A | B | ... | Z | a | b | ... | z
  - **lettre → [A-Za-z]**
- chiffre → 0 | 1 | ... | 9
  - **chiffre → [0-9]**

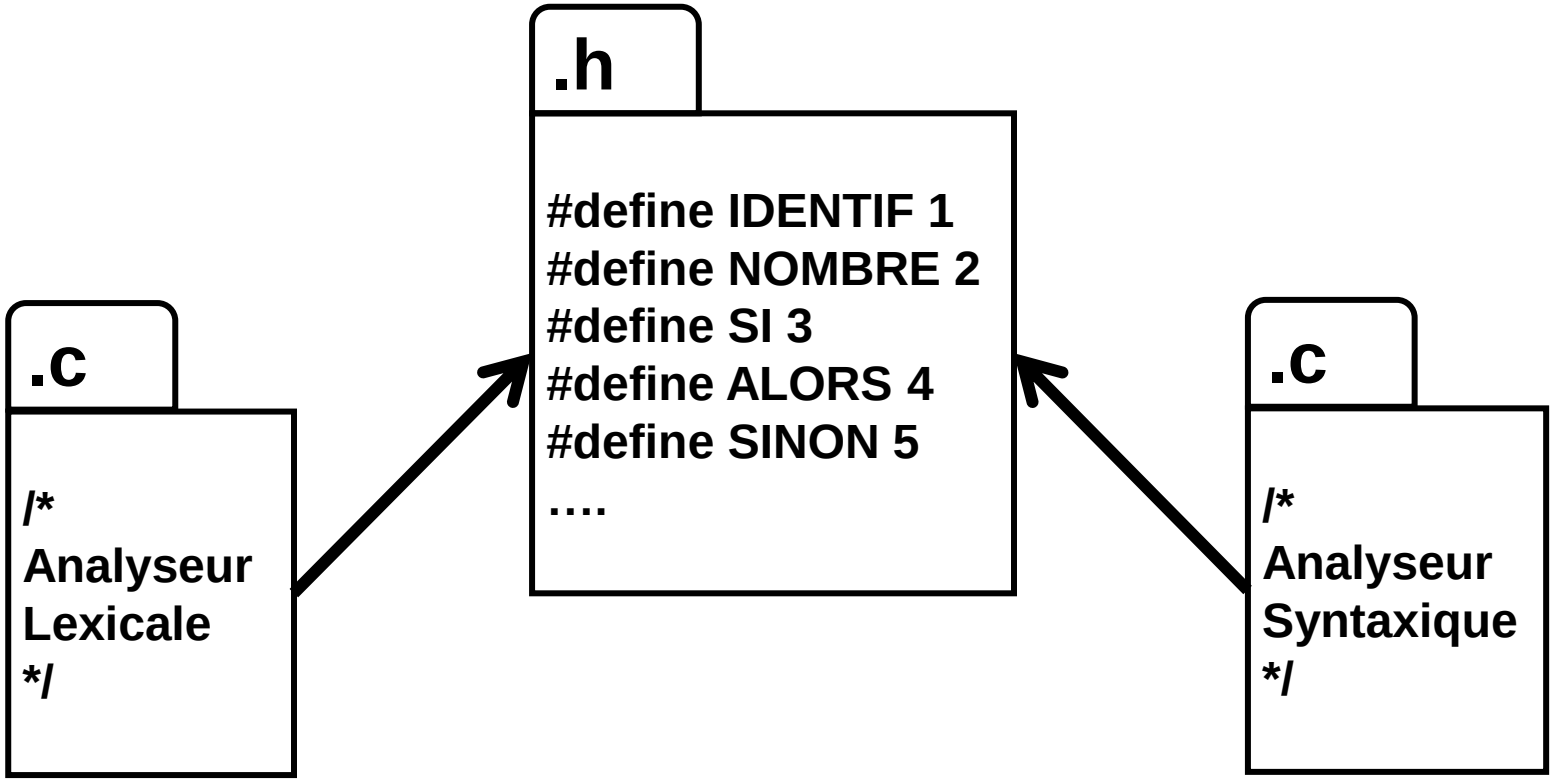
Identification des unités lexicales

- Nous avons vu comment définir les unités lexicales ;
- Nous allons voir comment écrire un programme qui identifie les unités lexicales dans le code source : « L' **analyseur lexical** ».



Type uniteLexSuivante() : Renvoie à chaque appel l'unité lexicale suivante identifiée dans le code source

Identification des unités lexicales



Diagrammes de transition

- Les diagrammes de transition sont une étape préparatoire pour la réalisation d'un analyseur lexical.
- Au fur et à mesure qu'il reconnaît une unité lexicale, l'analyseur lexical passe par divers états.
- Ces états sont numérotés et représentés dans le diagramme par des cercles.
- De chaque état *e* sont issues une ou plusieurs flèches étiquetées par des caractères c.
- Nous distinguons 2 types de diagrammes : ***déterministe*** et ***non déterministe***

Diagrammes de transition

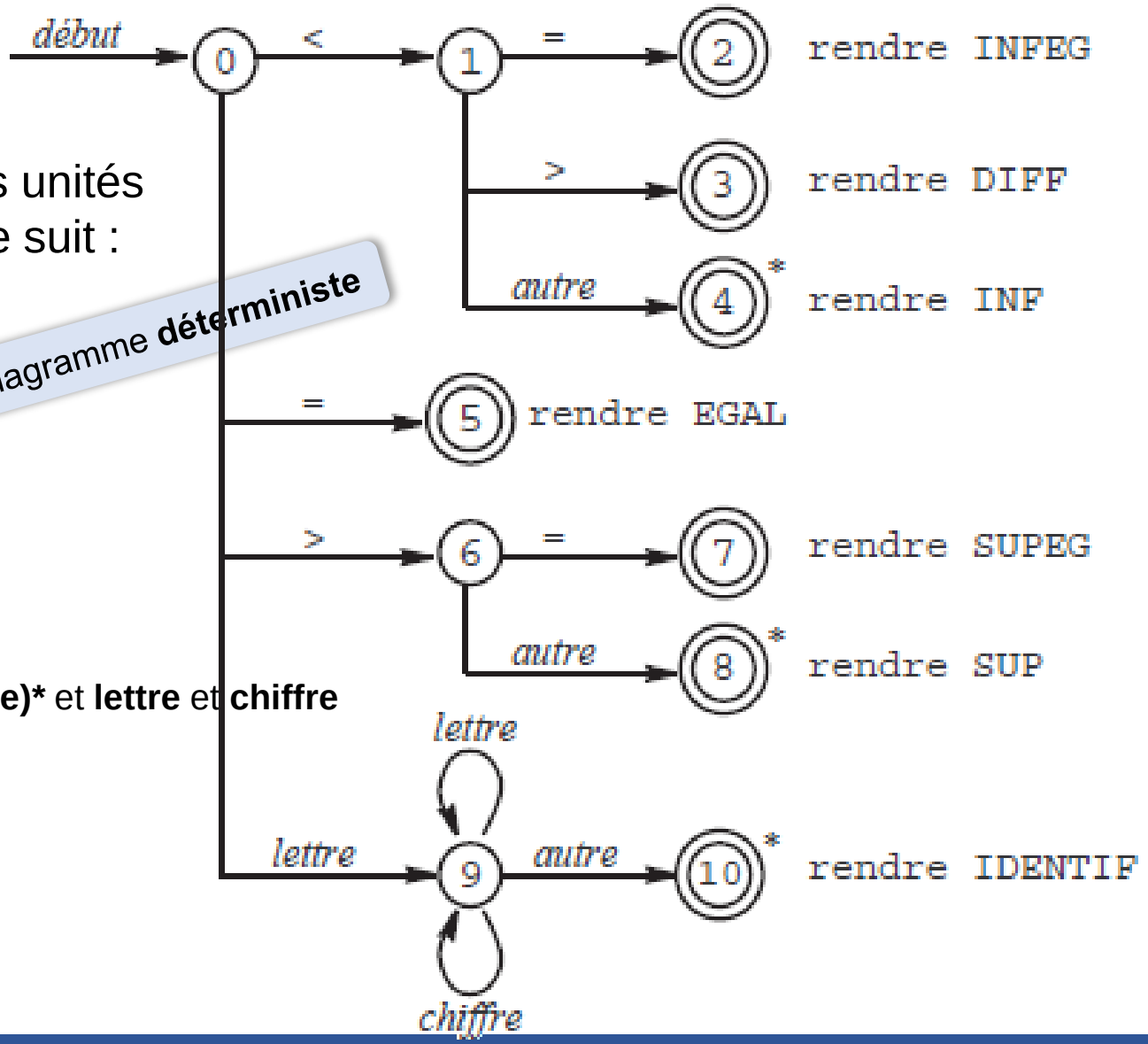
- Un diagramme de transition est dit **non déterministe** lorsqu'il existe, issues d'un même état, **plusieurs flèches** étiquetées par le même caractère, ou bien lorsqu'il existe des flèches étiquetées par la chaîne vide ε.
- Dans le cas contraire le diagramme est dit **déterministe**.

Diagrammes de transition

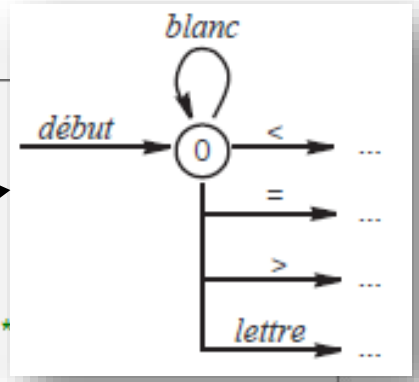
Exemple :
La reconnaissance des unités lexicales définit comme suit :

EGAL	**=**
INFEGAL	**<=**
DIFF	**<>**
INF	**<**
SUPEG	**>=**
SUP	**>**
IDENTIF	**lettre(lettre|chiffre)*** et **lettre** et **chiffre**

C'est un diagramme **déterministe**



Diagrammes de transition




```
int uniteLexiSuivante(void) {
    char c;
    c = lireCar(); /* etat = 0 */
    while (estBlanc(c))
        c = lireCar();
    if (c == '<') {
        c = lireCar(); /* etat = 1 */
        if (c == '=')
            return INFEG; /* etat = 2 */
        else if (c == '>')
            return DIFF; /* etat = 3 */
        else {
            delireCar(c); /* etat = 4 */
            return INF;
        }
    }
    else if (c == '=')
        return EGAL; /* etat = 5 */
    else if (c == '>') {
        c = lireCar(); /* etat = 6 */
        if (c == '=')
            return SUPEG; /* etat = 7 */
        else {
            delireCar(c); /* etat = 8 */
            return SUP;
        }
    }
    else if (estLettre(c)) {
        lonLex = 0; /* etat = 9 */
        lexeme[lonLex++] = c;
        c = lireCar();
        while (estLettre(c) || estChiffre(c)) {
            lexeme[lonLex++] = c;
            c = lireCar();
        }
        delireCar(c); /* etat = 10 */
        return IDENTIF;
    }
    else {
        delireCar(c);
        return NEANT; /* ou bien donner une erreur */
    }
}
```

Diagrammes de transition

```c
int estBlanc(char c) {
    return c == ' ' || c == '\t' || c == '\n';
}

int estLettre(char c) {
    return 'A' <= c && c <= 'Z' || 'a' <= c && c <= 'z';
}

int estChiffre(char c) {
    return '0' <= c && c <= '9';
}
```

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

char lireCar(void) {
    return getc(stdin);
}
```

```c
struct {
    char *lexeme;
    int uniteLexicale;
} motRes[] = {
    { "si", SI },
    { "alors", ALORS },
    { "sinon", SINON },
    ...
};

int nbMotRes = sizeof motRes / sizeof motRes[0];
```

Diagrammes de transition

- Les mots réservés appartiennent au langage défini par l'expression régulière lettre(lettre|chiffre)*, tout comme les identificateurs.
- Leur reconnaissance peut donc se traiter de deux manières :
  - Soit on incorpore les mots réservés aux diagrammes de transition, ce qui permet d'obtenir un analyseur très efficace, mais au prix d'un travail de programmation plus important, car les diagrammes de transition deviennent très volumineux;
  - Soit on laisse l'analyseur traiter de la même manière les mots réservés et les identificateurs puis, quand la reconnaissance d'un « identificateur-ou-mot-réservé » est terminée, on recherche le lexème dans une table pour déterminer s'il s'agit d'un identificateur ou d'un mot réservé.

Diagrammes de transition

```
struct {
    char *lexeme;
    int uniteLexicale;
} motRes[] = {
    { "si", SI },
    { "alors", ALORS },
    { "sinon", SINON },
    ...
};

int nbMotRes = sizeof motRes / sizeof motRes[0];
```

La fonction **int uniteLexiSuivante(void)** sera modifié comme suit :

```
...
else if (estLettre(c)) {
    lonLex = 0; /* etat = 9 */
    lexeme[lonLex++] = c;
    c = lireCar();
    while (estLettre(c) || estChiffre(c)) {
        lexeme[lonLex++] = c;
        c = lireCar();
    }
    delireCar(c); /* etat = 10 */
    lexeme[lonLex] = '\0';
    for (i = 0; i < nbMotRes; i++)
        if (strcmp(lexeme, motRes[i].lexeme) == 0)
            return motRes[i].uniteLexicale;
    return IDENTIF;
}
...
```

Automates finis

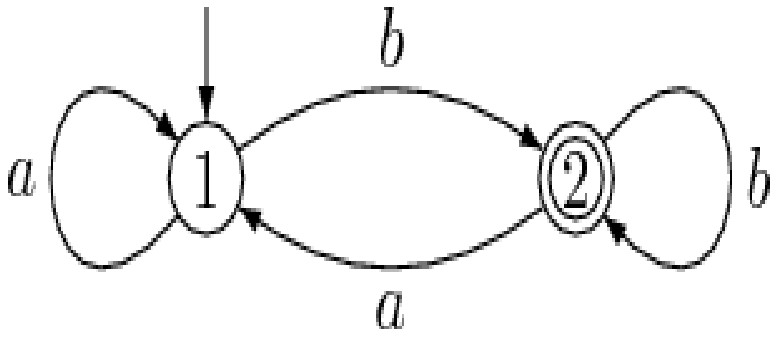
- Un **automate fini** est défini par la donnée de
  - un ensemble fini d'états E ;
  - un ensemble fini de symboles (ou alphabet) d'entrée Σ ;
  - une fonction de transition, transit : E x Σ → E ;
  - un état ε_0 distingué, appelé état initial ;
  - un ensemble d'états F, appelés états d'acceptation ou états finaux
- Un **automate** peut être représenté graphiquement par un graphe (***même structure que le diagramme de transition***) où les états sont figurés par des cercles (les états finaux par des cercles doubles) et la fonction de transition par des flèches étiquetées par des caractères :
  - si *transit*(e_1, *c*) = e_2 alors le graphe a une flèche étiquetée par le caractère *c*, issue de e_1 et aboutissant à e_2.

Automates finis

- Un automate fini accepte une chaîne d'entrée $s = c_1c_2 \ldots c_k$ si, et seulement si, il existe dans le graphe de transition un chemin joignant l'état initial e_0 à un certain état final e_k, composé de k flèches étiquetées par les caractères c_1, c_2… c_k.
- La transformation d'un automate fini en un analyseur lexical se passe en associant une unité lexicale à chaque état final et de faire en sorte que l'acceptation d'une chaîne produise comme résultat l'unité lexicale associée à l'état final en question.

Table de transition d’état

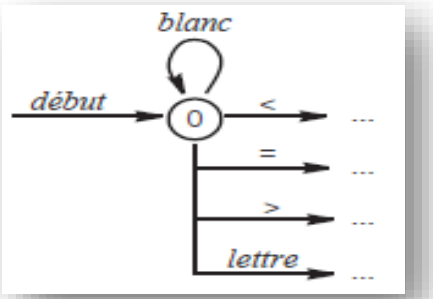
- Une table de transition d'état est un tableau montrant dans quel état d'un automate fini se déplacer, sur la base de l'état actuel et des autres.



	a	***b***
→ 1	**1**	**2**
*** 2**	**1**	**2**

Table de transition d'état

- Table de transition d'état selon les diagrammes précèdent :



	' '	'\t'	'\n'	'<'	'='	'>'	lettre	chiffre	autre
0	0	0	0	1	5	6	9	erreur	erreur
1	4*	4*	4*	4*	2	3	4*	4*	4*
6	8*	8*	8*	8*	7	8*	8*	8*	8*
9	10*	10*	10*	10*	10*	10*	9	9	10*

Table de transition d’état

- Implémentation de deux nouvelles fonctions **transit** et **final**
- final[e] = 0 si *e* n'est pas un état final (vu comme un booléen, final[e] est faux),
- final[e] = U + 1 si *e* est final, sans étoile et associé à l'unité lexicale *U* (en tant que booléen, final[e] est vrai, car les unités lexicales sont numérotées au moins à partir de zéro),
- final[e] = ¡(U + 1) si *e* est final, étoilé et associé à l'unité lexicale *U* (en tant que booléen, final[e] est encore vrai)

Table de transition d'état

```
#define NBR_ETATS ...
#define NBR_CARS 256

int transit[NBR_ETATS][NBR_CARS];
int final[NBR_ETATS + 1];

int uniteSuivante(void) {
    char caractere;
    int etat = etatInitial;
    while ( ! final[etat]) {
        caractere = lireCar();
        etat = transit[etat][caractere];
    }
    if (final[etat] < 0)
        delireCar(caractere);
    return abs(final[etat]) - 1;
}
```

Table de transition d'état

- Les tableaux transit et final devraient être initialisés pour correspondre aux diagrammes précédents :

```
void initialiser(void) {
    int i, j;

    for (i = 0; i < NBR_ETATS; i++) final[i] = 0;
    final[ 2] = INFEG + 1;
    final[ 3] = DIFF + 1;
    final[ 4] = - (INF + 1);
    final[ 5] = EGAL + 1;
    final[ 7] = SUPEG + 1;
    final[ 8] = - (SUP + 1);
    final[10] = - (IDENTIF + 1);
    final[NBR_ETATS] = ERREUR + 1;

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

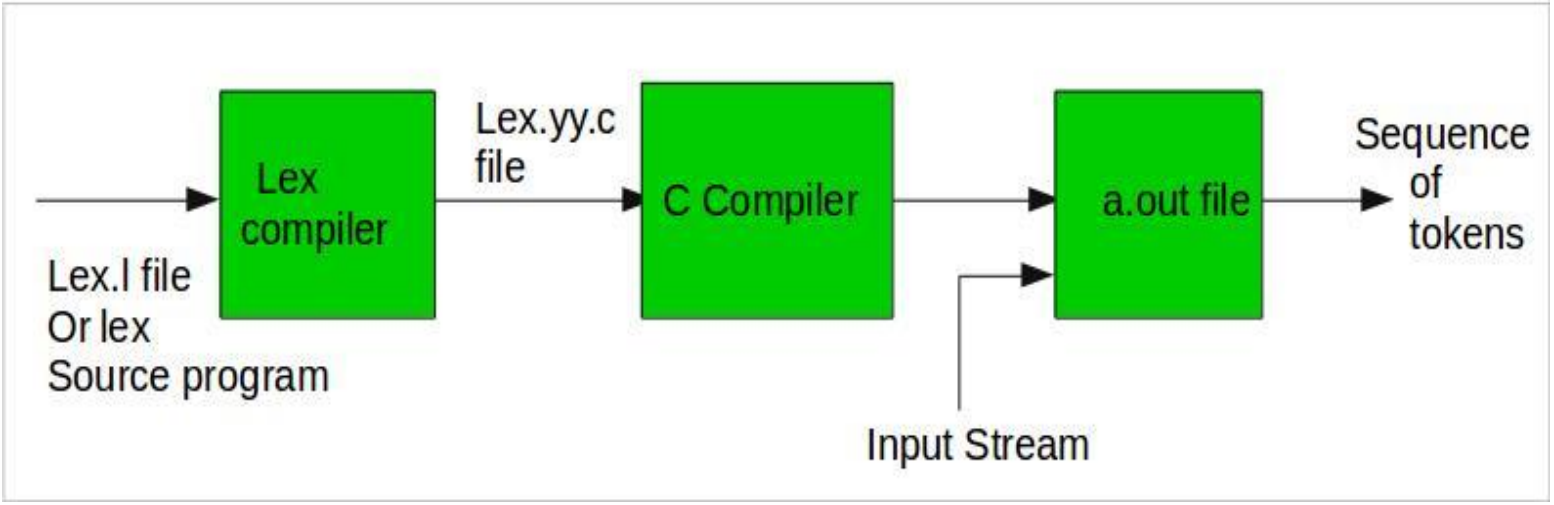
```
    transit[0][' '] = 0;
    transit[0]['\t'] = 0;
    transit[0]['\n'] = 0;

    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;
}
```

Générateurs d’analyseurs lexicaux

- Les analyseurs lexicaux basés sur des tables de transitions sont les plus efficaces.
- Malgré la construction de cette table est une opération longue et délicate.
- Le programme ***lex*** (**flex)** permet de faire cette construction automatiquement.

Flex

- FLEX (**fast lexical analyzer generator**) is a tool/computer program for generating lexical analyzers (scanners or lexers) written by Vern Paxson in C around 1987



Flex Program Structure

- In the input file, there are 3 sections:

```
%{
// Definitions
%}

%%
Rules
%%

User code section
```

- The definition section contains the declaration of variables, regular definitions, manifest constants.
- In the definition section, text is enclosed in **"%{ %}"** brackets.
- Anything written in this brackets is copied directly to the file **lex.yy.c**

- The rules section contains a series of rules in the form: *pattern action* and pattern must be unintended and action begin on the same line in {} brackets.
- The rule section is enclosed in **"%% %%"**.

- This section contain C statements and additional functions

How to run the program

- To run the program, it should be first saved with the extension **.l**
- Run the below commands on terminal in order to run the program file.

```
$ flex analex.l

$ gcc lex.yy.c -o myprog

$ myprog < code.txt
```