# Analyse lexicale

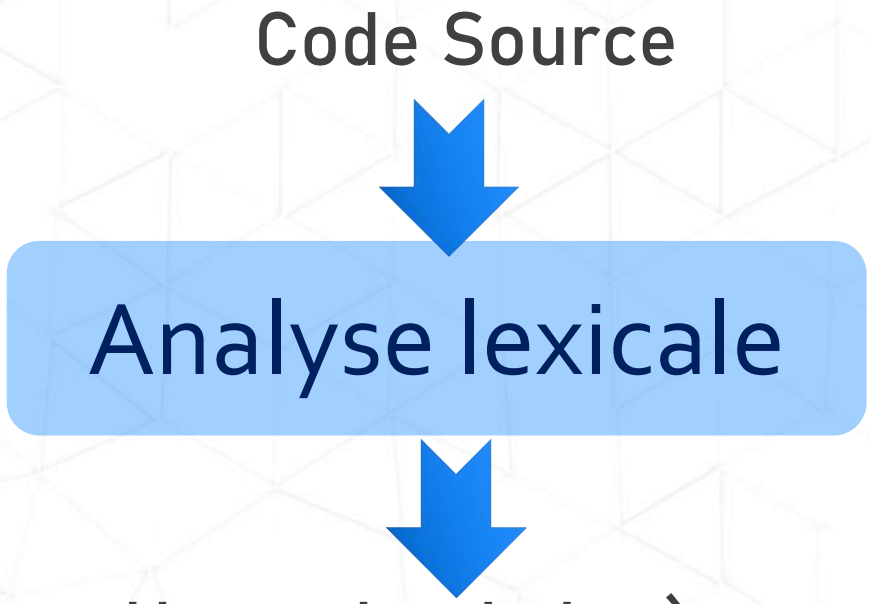
**Dr. Aida Lahouij**
Maitre assistante de l'enseignement supérieur et de la recherchce scientifique
Aida.lahouij@gmail.com

# CONTENTS

**Introduction:**

Qu'est-ce que l'analyse lexicale ?

- Première phase de la compilation.
- Transformation du code source en une suite de lexèmes (unités significatives).
- Objectif : Générer des tokens (symboles) pour l'analyse syntaxique.



**Introduction:**

**Exemple :**

- Code : `x = 42 + y;`

→ Tokens :
[IDENT, x],
[ASSIGN, =],
[NUMBER, 42],
[OP, +],
[IDENT, y],
[SEMICOLON, ;]

## Introduction:

## Composants clés

**Token :** Symbole catégorisé (ex: IF, NUMBER, IDENT).

**Lexème :** Séquence de caractères correspondant à un token.

**Motifs (patterns) :** Règles regex pour identifier les lexèmes.

# Introduction:

## Remarque

L'analyseur lexical ne traite en général pas les combinaisons d'unités lexicales, cette tâche étant laissée à l'analyseur syntaxique.

**Exemple:**

- un analyseur lexical typique peut reconnaître et traiter les parenthèses mais est incapable de les compter et donc de vérifier si chaque parenthèse fermante « ) » correspond à une parenthèse ouvrante « ( » précédente.

# Introduction:

## Remarque

On distingue deux types d'analyse lexicale:

- **l'analyse lexicale automatique**
  - par un générateur d'analyseurs lexicaux : Lex, Flex., ANTLR, etc.
- **l'analyse lexicale manuelle** (codée à la main)
  - « à la main » : il faut construire l'automate fini non déterministe à partir d'une expression rationnelle E, puis l'exécuter pour déterminer si une chaîne d'entrée appartient au langage reconnu par E ;
  - par **une table** décrivant l'automate et un programme exploitant cette table ;

# 01

# Analyse lexicale automatique

# L'analyse lexicale automatique:

## Outils et technologies

**Lex** (outil UNIX pour générer des analyseurs lexicaux).

**Flex** (version open source de Lex).

**Expressions régulières** (définir les motifs des tokens).

**ANTLR** (générateur d'analyseurs syntaxiques et lexicaux).

# L'analyse lexicale automatique:

## Flex (Fast Lexical Analyzer Generator)

### Qu'est-ce que Flex ?

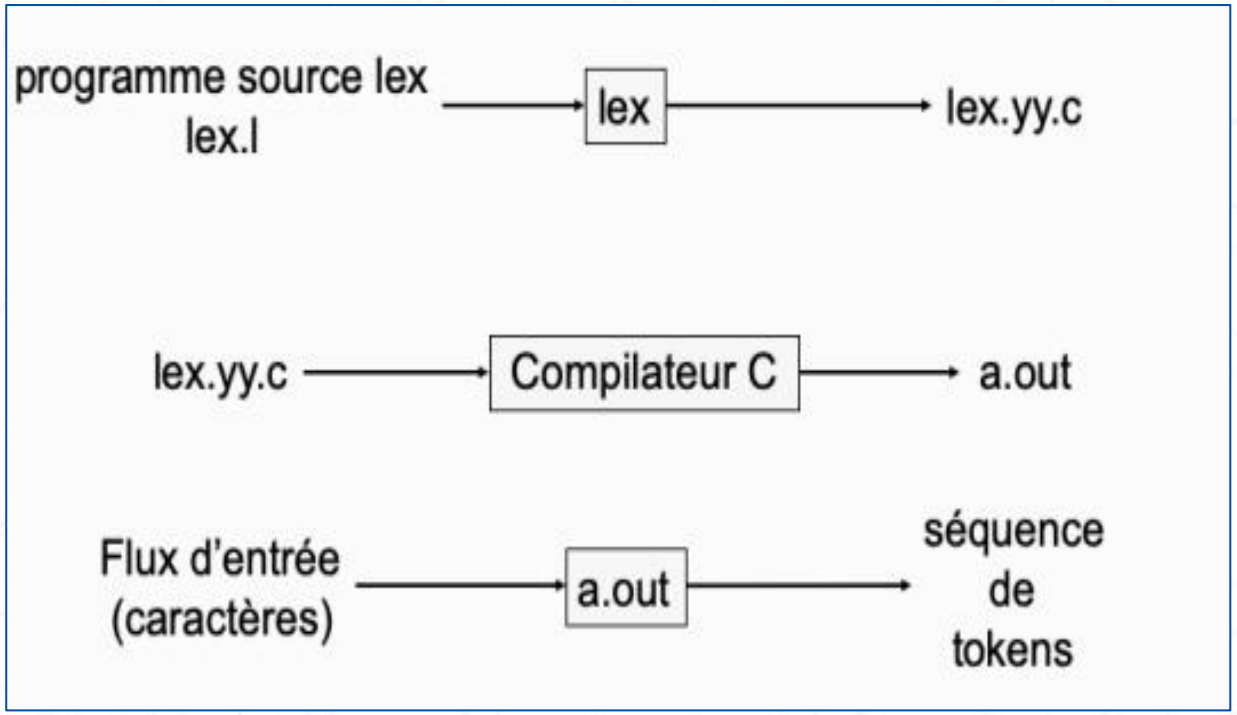
- Outil open source pour générer des analyseurs lexicaux (lexers).
- Transforme des règles regex en code C optimisé.

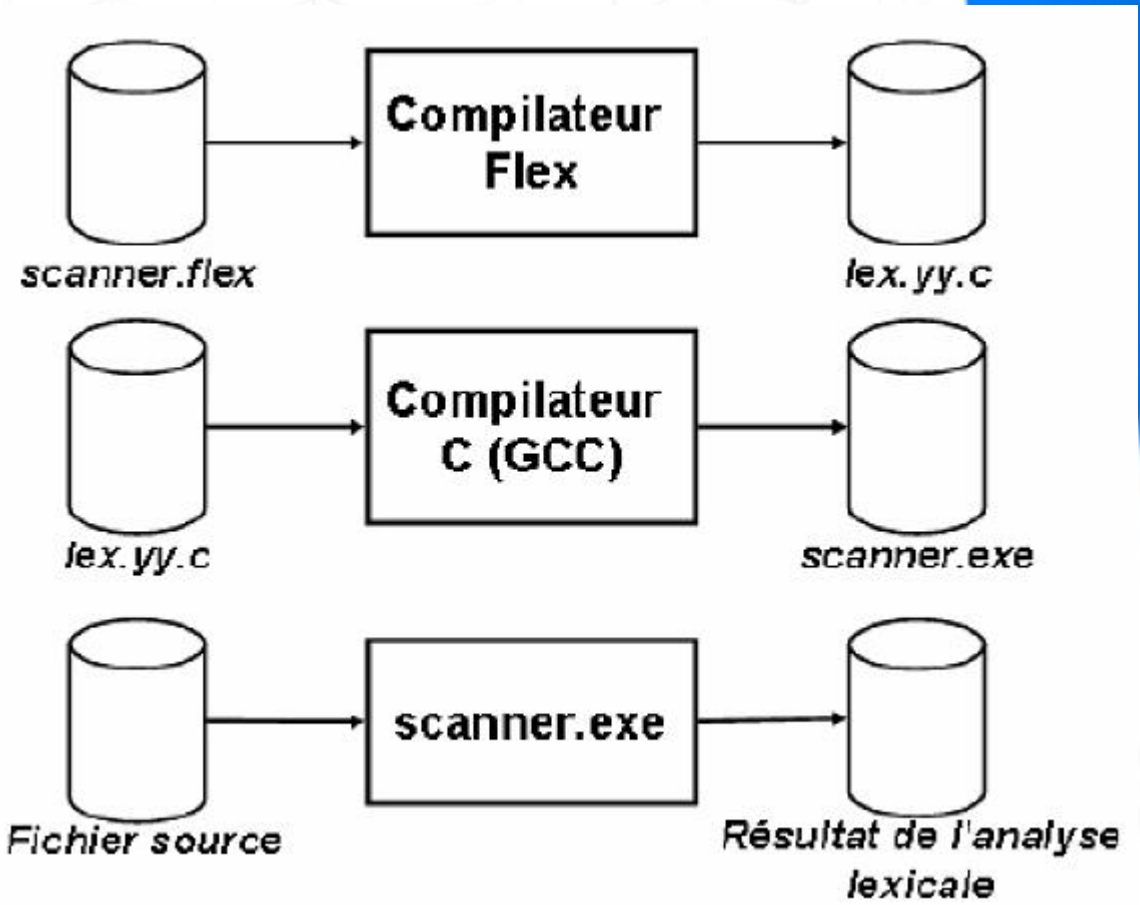
### Pourquoi Flex ?

- Rapide, léger, compatible avec Bison (analyse syntaxique).
- Idéal pour les compilateurs, interprètes, ou outils de traitement de te

# L'analyse lexicale automatique:

## Flex



01

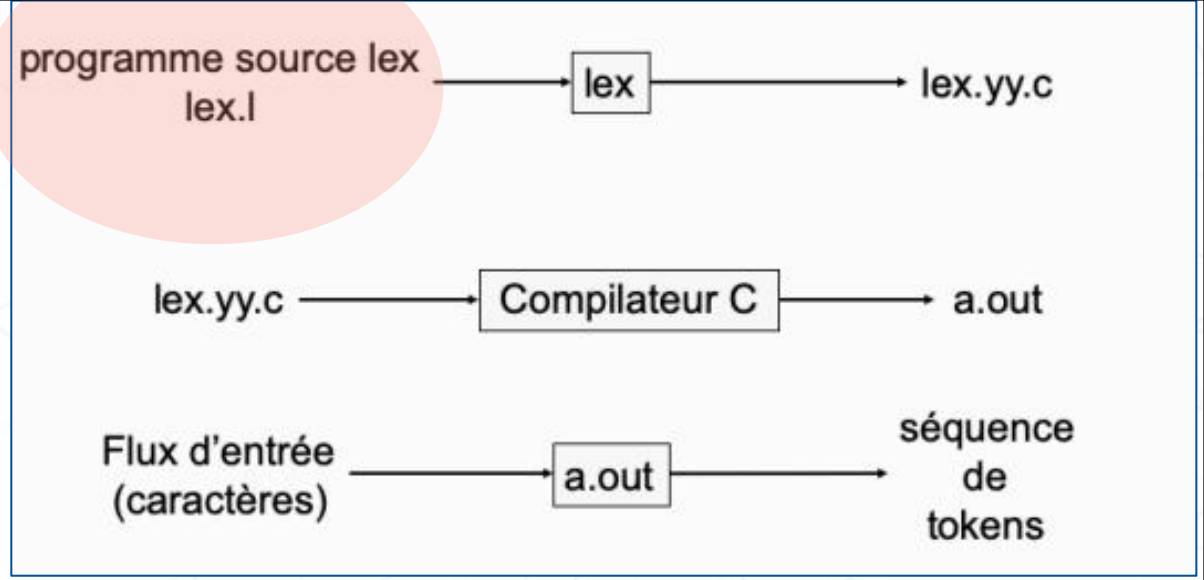
# Analyse lexicale automatique

## Programme source lex (Fichier .l ou .lex)

# L'analyse lexicale automatique:
## Flex

### Programme source lex (Fichier .l ou .lex)

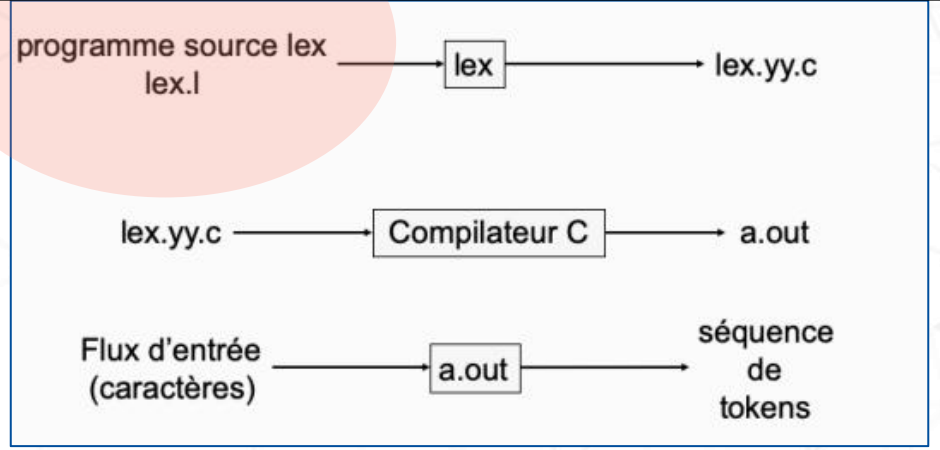
- C'est le fichier source contenant les règles lexicales écrites en syntaxe Flex/Lex.
- Définit les motifs (regex) et les actions associées (code C).




```
%{
#include <stdio.h>
%}
%%
[0-9]+   { printf("Nombre: %s\n", yytext); }
[a-zA-Z]+ { printf("Mot: %s\n", yytext); }
%%
int main()
{ yylex();
return 0; }
```

# L'analyse lexicale automatique:
## Flex

## Programme source lex (Fichier .l ou .lex)




Le contenu d'un fichier Flex comprend 3 sections séparées par une ligne contenant le symbole « %% » :

```
%{
 /*   Déclarations *  /
%}
/*   Définitions *  /
 %%
 /*   Règles et actions
*  /
 %%
 /*   Code utilisateur *  /
```

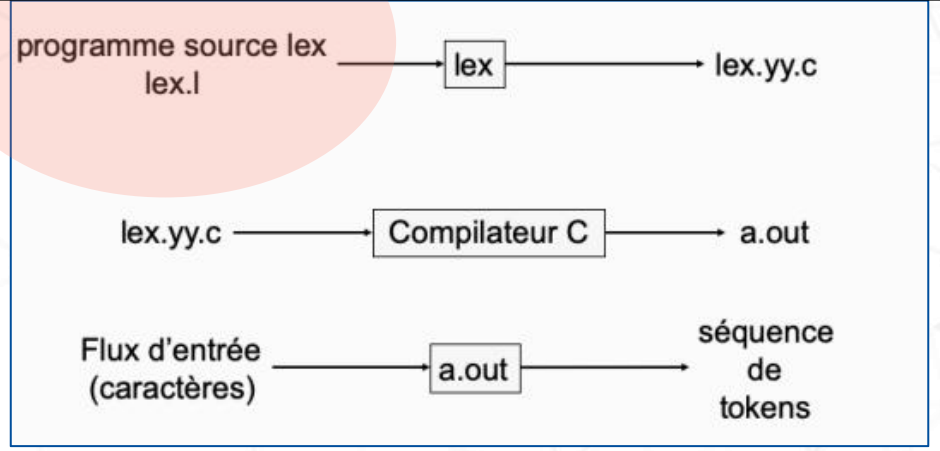
**1.Déclarations** (%{ … %}) : Code C (en-têtes, variables).
**2.Règles Lexicales** (%% … %%) : Regex + Actions.
**3.Code Utilisateur :** Fonctions supplémentaires (ex: main()).

**Exemple Minimal :**

```lex
%{
#include <stdio.h>
%}
%%
.   { printf("Caractère: %s\n", yytext); }
%%
int main() { yylex(); return 0; }
```

# L'analyse lexicale automatique:
## Flex

## Programme source lex (Fichier .l ou .lex)




/* Déclarations * /

- Contient la déclaration des variables et des fonctions globales, des inclusions de fichiers.

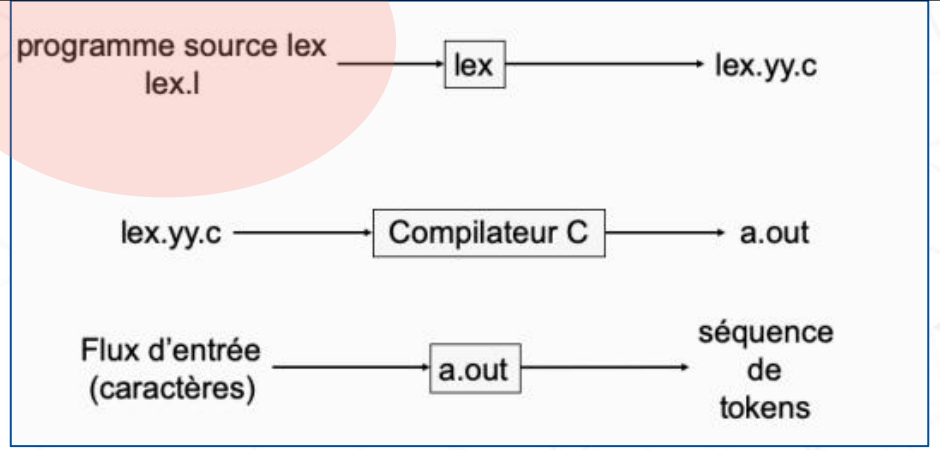
- Les inclusions de fichiers et les déclarations des variables et des fonctions globales sont mises entre les symboles « %{ » et « %} », chacun sur une seule ligne qui ne doit pas être indentée).

**Exemple :**

```
%{
#include<stdio.h>
#define N 100
int x = 12;
float z = 1.5; // une variable globale
int fct(int, float); // une fonction globale
%}
```

# L'analyse lexicale automatique:
## Flex

## Programme source lex (Fichier .l ou .lex)




/* Définitions */

- Une définition permet de nommer une expression régulière.
- Une définition est un couple formé d'un identificateur et d'une expression régulière, séparée par au moins un caractère espace (ou tabulation).
- Une expression régulière peut contenir une référence à un identificateur déjà défini.
- Dans ce cas, l'identificateur doit être écrit entre les symboles "{" et "}".
- Une définition doit être écrite sur une seule ligne.

**Exemple :**

CHIFFRE [0 – 9]

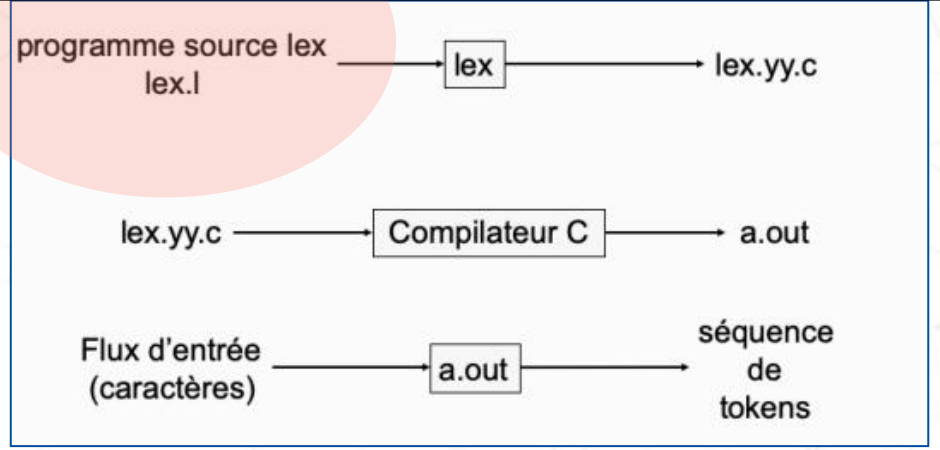
ENTIER (+|–)? {CHIFFRE} +

**Syntaxe** :

**Nom** Expression_Régulière

# L'analyse lexicale automatique:
## Flex

### Programme source lex (Fichier .l ou .lex)




/* Règles */

- Une règle Flex est la donnée :
  - d'une expression régulière.
  - et d'une action (celle qui sera exécutée lorsqu'une séquence d'entrée est reconnue).
- Les règles de traduction sont de la forme

```
p1   { action1 }
p2   { action2 }
...
pn   { actionn }
```

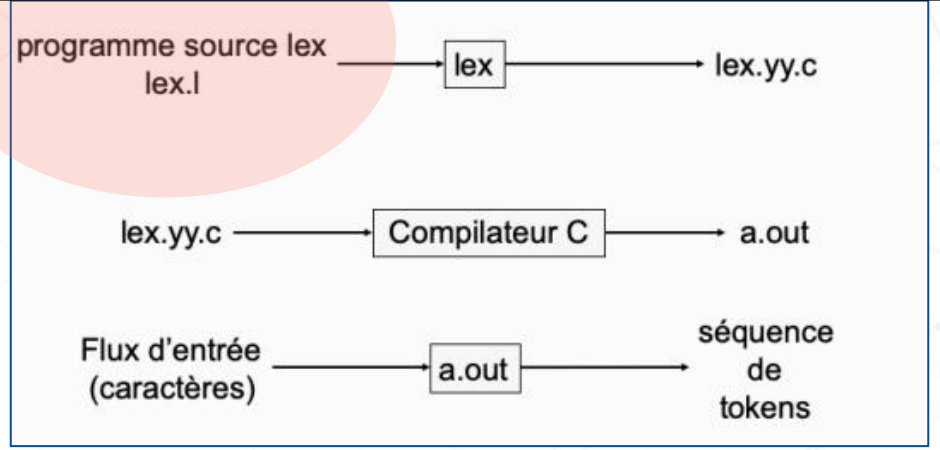
où chaque pi est une expression rationnelle et chaque action est une suite d'instructions en C.

**Exemple :**

"int" {printf ("Mot cle int"); }

A chaque fois que la chaîne de caractères "int" sera reconnue, le message "Mot cle int" sera affiché sur la sortie standard.

# L'analyse lexicale automatique:
## Flex




## Programme source lex (Fichier .l ou .lex)

| Motif (Pattern) | Correspondances possibles |
|---|---|
| `[0-9]` | Tous les chiffres entre `0` et `9`. |
| `[0+9]` | Soit `0`, `+` ou `9`. |
| `[0, 9]` | Soit `0`, `,`, `' '` (espace) ou `9`. |
| `[0 9]` | Soit `0`, `' '` (espace) ou `9`. |
| `[-09]` | Soit `-`, `0` ou `9`. |
| `[-0-9]` | Soit `-`, ou tous les chiffres entre `0` et `9`. |
| `[0-9]+` | Un ou plusieurs chiffres entre `0` et `9`. |
| `[^a]` | Tous les caractères sauf `a`. |
| `[^A-Z]` | Tous les caractères sauf les lettres majuscules ( `A-Z` ). |

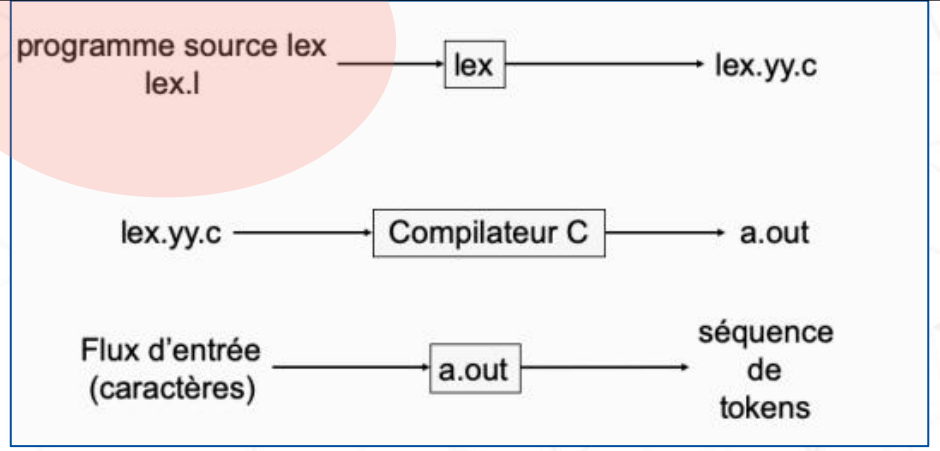
| | |
|---|---|
| `a{2,4}` | `aa`, `aaa` ou `aaaa`. |
| `a{2,}` | Deux occurrences ou plus de `a`. |
| `a{4}` | Exactement 4 occurrences de `a`, soit `aaaa`. |
| `.` | N'importe quel caractère sauf une nouvelle ligne ( `\n` ). |
| `a*` | Zéro ou plusieurs occurrences de `a`. |
| `a+` | Une ou plusieurs occurrences de `a`. |
| `[a-z]` | Toutes les lettres minuscules ( `a` à `z` ). |
| `[a-zA-Z]` | Toute lettre alphabétique ( `a-z` ou `A-Z` ). |
| `w(x\|y)z` | `wxz` ou `wyz` |

# L'analyse lexicale automatique:
## Flex

### Programme source lex (Fichier .l ou .lex)

/* Code Utilisateur* /

- La fonction main() pour exécuter l'analyseur lexical.
- Des fonctions auxiliaires (ex: affichage, calculs).
- Des initialisations personnalisées.

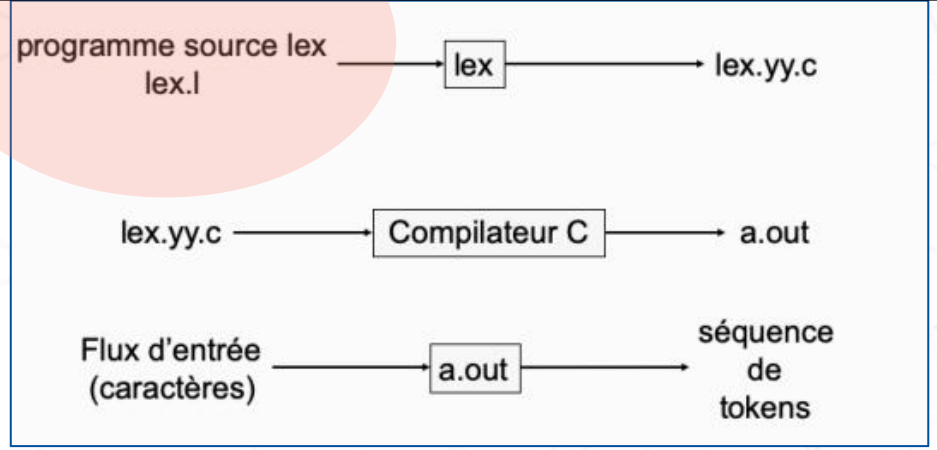



### Exemple de main() classique

```
int main() {
  yylex();  // Lance l'analyse lexicale
  return 0;
}
```

# L'analyse lexicale automatique:
## Flex

## Programme source lex (Fichier .l ou .lex)




/* Code Utilisateur* /

**Exemple 2:** Lire depuis un fichier

```
int main(int argc, char *argv[]) {
  if (argc > 1) {
    FILE *f = fopen(argv[1], "r");
    yyin = f;  // Redirige l'entrée vers le fichier
  }
  yylex();
  return 0;
}
```

**Sans fopen :** L'entrée provient de stdin, donc on peut entrer du texte manuellement ou rediriger un fichier avec <.

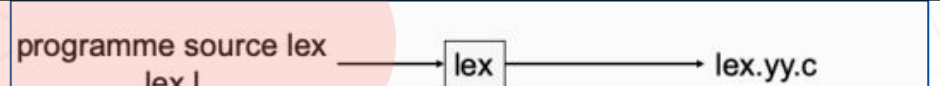
**Avec fopen :** On ouvre explicitement un fichier et yyin est défini comme un fichier spécifique

# L'analyse lexicale automatique:
## Flex

## Programme source lex (Fichier .l ou .lex)

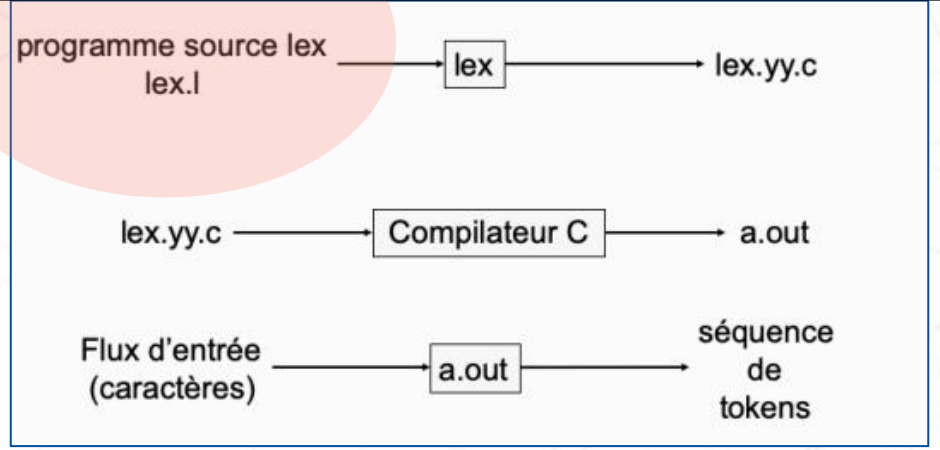
/* Code Utilisateur* /

**Exemple 3:** définir des fonctions supplémentaires pour traiter des actions spécifiques




```
%{
#include <stdio.h>
#include "lex.yy.c" // Inclusion du fichier généré par Flex

// Prototype d'une fonction utilisateur supplémentaire
void message();
%}

%%
[0-9]+      { printf("Nombre détecté: %s\n", yytext); }
[a-zA-Z]+   { printf("Mot détecté: %s\n", yytext); }
.           { printf("Caractère inconnu: %s\n", yytext); }
%%

// Fonction utilisateur supplémentaire
void message() {
    printf("Analyse lexicale terminée.\n");
}

// Fonction principale (main)
int main() {
    printf("Début de l'analyse lexicale :\n");
    yylex(); // Appelle l'analyseur lexical
    message(); // Appelle la fonction utilisateur
    return 0;
}
```

# L'analyse lexicale automatique:
## Flex




## Programme source lex (Fichier .l ou .lex)

/* Code Utilisateur* /

**Remarques:**

- Si vous appelez **yylex()** sans configuration supplémentaire, le scanner lira depuis :Le clavier (saisie utilisateur) ou Un fichier redirigé via la ligne de commande (ex: ./a.out < input.txt).
- Si vous ne définissez pas de fonction **main()** dans votre code Flex : Flex génère une main() par défaut qui lit depuis stdin. Vous devrez alors fournir l'entrée manuellement ou via une redirection.

| Source | Configuration Requise |
|---|---|
| Clavier (stdin) | Aucune : `yyin = stdin` par défaut. |
| Fichier | `yyin = fopen("fichier.txt", "r");` |
| Chaîne en mémoire | `yy_scan_string("ma_chaine");` |
| Redirection shell | `./a.out < input.txt` (automatique). |

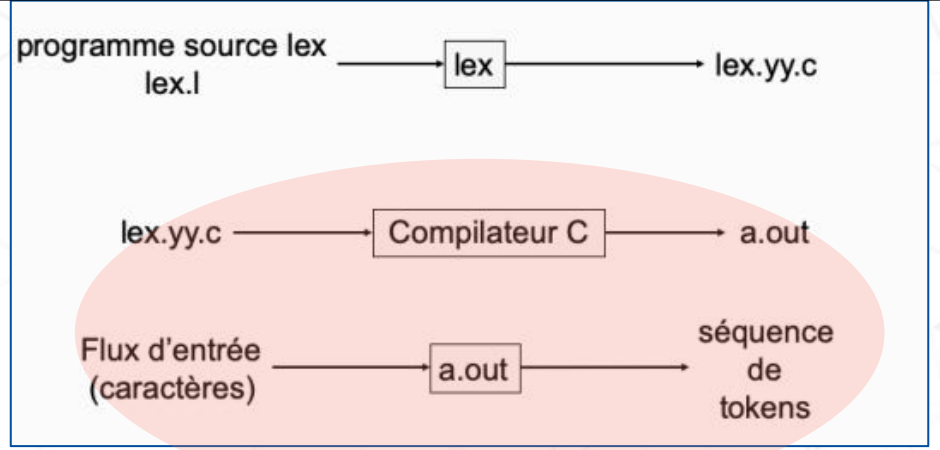
01

# Analyse lexicale automatique

## Compilation et exécution

# L'analyse lexicale automatique:
## Flex

### Compilation et exécution




1. **Générer le lexer :**

```bash
flex exemple.l  # Produit lex.yy.c
```

2. **Compiler avec le code utilisateur :**

```bash
gcc lex.yy.c -o mon_lexer
```

3. **Exécuter :**

```bash
./mon_lexer < input.txt
```

01

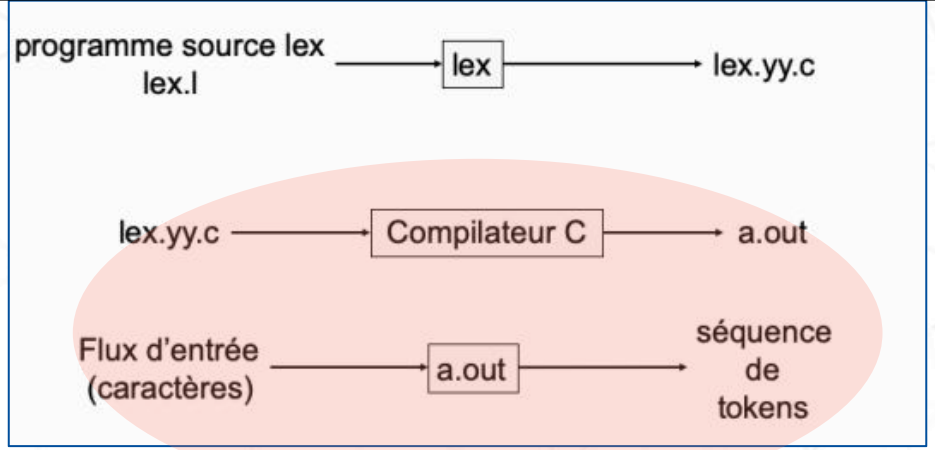
# Analyse lexicale automatique

## Exemple

# L'analyse lexicale automatique:
## Flex

## Exemple Complet

```
%{
#include <stdio.h>
int count = 0;
%}
%option noyywrap
%%
[a-zA-Z]+   { count++; }  // Règle pour les mots
.|\n        ;             // Ignorer les autres caractères
%%

int main() {
  yylex();
  printf("Nombre de mots : %d\n", count);
  return 0;
}
```




1. copier le code dans un fichier texte et enregistrer sous le nom ex1 et l'extension .l
2. générer l'analyseur en utilisant la commande:
   win_flex ex1.l
3. compiler le code de l'analyseur à l'aide de la commande:
   gcc lex.yy.c -o ex1Exe.exe
4. creer un fichier texte test
5. Lancer l'analyse du fichier à l'aide de la commande:
   TP1Exe < test.txt

01

# Analyse lexicale automatique

## Résolution de conflit

# L'analyse lexicale automatique:
## Flex

### Résolution des conflits

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

Exemples

"prog" action1
"program" action2

**La deuxième règle sera choisie en cas de conflit.**

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

Exemples

"prog" action1
[a – z]+ action2

**La première règle sera choisie en cas de conflit de lexèmes ayant mêmes longueurs**

Si aucune règle ne correspond au flot d'entrée, Flex choisit sa règle par défaut implicite :

.|\n {ECHO}

Cette règle par défaut, recopie le flot d'entrée sur le flot de sortie.

01

# Analyse lexicale automatique

## Annexe

# Annexe

| 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 |

# Annexe

**Les variables yyin et yyout** :

– Ce sont des variables globales de Flex de type « pointeur vers un fichier ».
– La variable yyin pointe vers le fichier d'entrée (fichier source) à analyser.
– Par défaut c'est stdin (yyin = stdin), c'est-à-dire que l'analyseur va lire les données à partir du clavier.
– On peut ouvrir un fichier source par l'instruction : « yyin = fopen(…); ».
– La variable yyout pointe vers le fichier de sortie (fichier cible).
– Par défaut c'est stdout (yyout = stdout), c'est-à-dire que l'analyseur va écrire les résultats sur l'écran.
– On peut ouvrir un fichier cible par l'instruction : « yyout = fopen(…); ».
- Une fois un lexème est reconnu, l'analyseur lexical le stocke sous forme d'une chaîne de caractères dans une variable globale appelée « yytext ».
- La longueur du lexème est également stockée sous forme d'un entier dans une variable globale appelée « yyleng » (yyleng = strlen(yytext)).
- Il est possible de contrôler la longueur maximale d'un lexème reconnu.

# Annexe

- `yylex()` : c'est la fonction C de l'automate déterministe, renvoie 0 lorsque le caractère `EOF` est atteint.
- `ECHO` : afficher la chaîne reconnue
- `yytext` : chaîne de caractères reconnue
- `yyleng()` : longueur du *token*
- `yylineno()` : numéro de ligne dans le fichier où le *token* a été reconnu
- `yyin`: fichier d'entrée (`stdin` par défaut)

# Merci

aida.lahouij@gmail.com