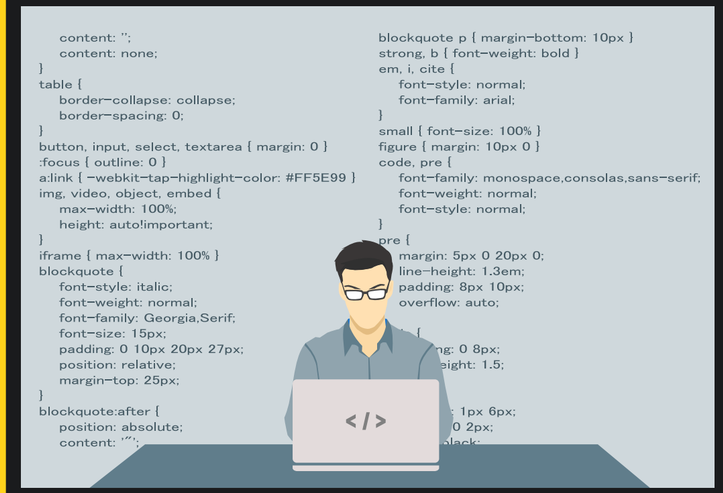
# TP COMPILATION

Flex et Bison

**Par: Lynda SAYOUD**

# 1. QU'EST CE QU'UN COMPILATEUR ?




Langage haut niveau (python, c, c#,...)

Traduction

Langage bas niveau
Langage machine,
Assembleur

- Un **compilateur** : Processus qui consiste à transformer un programme source écrit dans un langage de programmation évolué ( C, java..) en un code objet directement executable par l'ordinateur.

-> En effet, nous avons l'habitude de « **compiler** » le langage naturel que nous utilisons dans notre vie quotidienne, car lorsque notre cerveau lit ou entend une phrase, il :

1. Identifie les mots qui composent la phrase (**analyse lexicale**).
2. Vérifie si la structure de la phrase respecte les règles grammaticales (**analyse syntaxique**).
3. S'assure que la phrase a un sens (**analyse sémantique**).

Langage = Français.

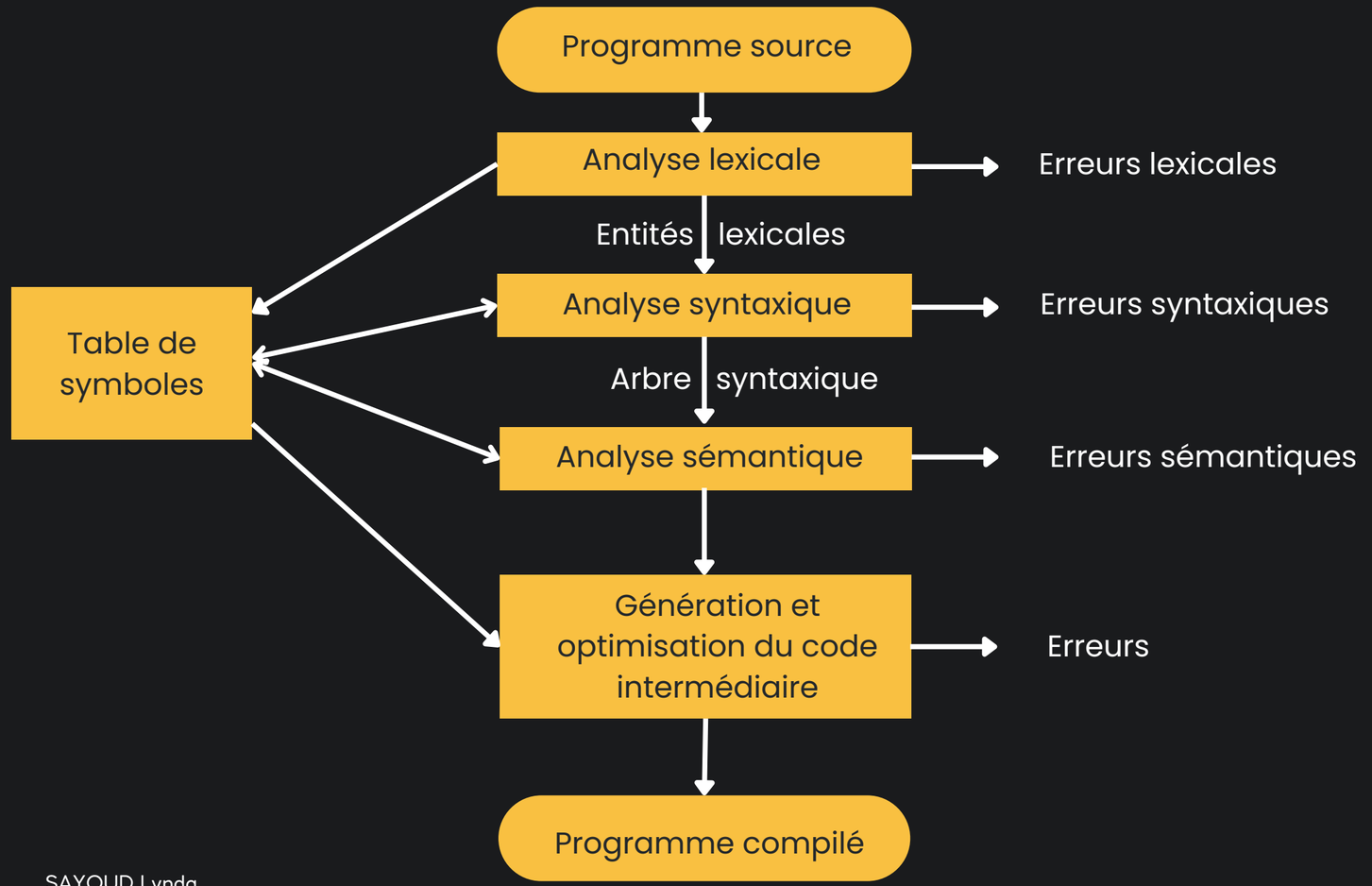
Exemple: Le programme chante le fichier.

Cette phrase est :

- lexicalement **correcte**,
  car les mots qui la composent appartiennent à la langue française
- Syntaxiquement **correcte**,
  car la structure de la phrase respecte les règles de la grammaire. Elle se compose d'un sujet ("Le programme"), d'un verbe ("chante") et d'un C.O.D. ("le fichier").

- Sémantiquement **Incorrecte**,
  car la phrase n'a pas un sens dans le contexte logique ou pratique.

# 2. ETAPES DE COMPILATION




**Exemple:**

Soit le programme en C suivant:

```
#include <stdio.h>
int main()
{ int x = 5;
float y =;
x + y;
return "texte";
}
```

- Lexicalement correcte.
- Syntaxiquement incorrecte.

- Sémantiquement incorrecte

# 3. EXPRÉSSIONS RÉGULIÈRES

- Les expressions régulières sont une écriture formelle qui sert à décrire des automates.
- Elles sont utilisées dans les **analyseurs lexicaux** pour identifier les lexèmes, c'est-à-dire des séquences finies de symboles du langage, afin de définir des éléments comme les mots-clés, les identificateurs ou les nombres.
- **Dans une expression régulière, tous les caractères, y compris les espaces, sont pris en compte.**
- Les caractères dans le tableau suivant sont **réservés** et doivent être précédés d'un antislash ( **** ) ou placés entre guillemets (**" "**) si l'on souhaite leur valeur littérale.

| caractères | significations | exemples |
|---|---|---|
| + | Répéter 1 ou plusieurs fois | x+ □ x...x |
| * | Répéter 0 ou plusieurs fois | t* □ vide ou t...t |
| ? | 0 ou 1 fois | a? □vide ou a |
| \| | union | ab\|bc□ ab ou bc |
| ( ) | factorisation | (a\|b)c □ac ou bc |
| " | valeur littérale des caractères | "+?"+ □ +?...+? |
| \ | valeur littérale de caractère qui le précède même entre "" | \++□+…+<br>"\"+ □ erreur car la première " ne se ferme pas. |
| . | N'importe quel caractère sauf la fin de ligne (\n). | .\|\n □n'importe quel caractère |
| [...] | ensemble de caractères. | [aeiou]□ la voyale :a ou e ou i ou o ou u. |
| - | Utilisé dans [] signifie un intervalle d'ensemble | [0-9] □ 0\|1\|2\|3\|4\|5\|6\|7\|8\|9 |
| ^ | Utilisé dans [] signifie le complément d'ensemble | [^0-9] □tout caractère sauf chiffre |

Attention :

[...] = ensemble de caractères, pas d'expressions régulières, ce qui veut dire:

[(0|1)+] □ un des caractères (, ), 0, 1, |, +, pas une suite de 0 ou 1.

Mais les caractères ^, \, - restent des caractères spéciaux.

[^\]] □tout caractère sauf ]

# Exercice:

Écrivez les expressions régulières qui définissent les entités suivantes :

1. Une suite de caractères alphanumériques qui commence par un caractère alphabétique.

2. Les constantes numérique signées par le (-/+).

3. N'importe quel caractère (la fin de ligne (\n) est inclue).

4. Tous les caractère à part les espace, les tabulations et les fins de lignes.

# Solution:

1. Une suite de caractères alphanumériques qui commence par un caractère alphabétique.

[a-zA-Z]([a-zA-Z]|[0-9])*

2. Les constantes numérique signées par le (-/+).

[+-]?([1-9][0-9]*|0) <=> ([+-][1-9][0-9]*)|0

3. N'importe quel caractère (la fin de ligne (\n) est inclue).

.|\n

4. Tous les caractère à part les espace, les tabulations et les fins de lignes.

[^ \n\t]

# 4. ANALYSE LEXICALE

L'**analyseur lexical** constitue la **première étape** d'un compilateur.

Ses principales tâches sont :

- Lire les caractères d'entrée et produire comme résultat une suite d'entités lexicales que l'analyseur syntaxique devra traiter.
- Éliminer les caractères superflus, tels que les commentaires, les tabulations et les fins de ligne.
- Gérer les numéros de ligne dans le programme source afin de pouvoir associer chaque erreur rencontrée à la ligne correspondante.

**Exemple:**

Code source avant la compilation :

```
int x;
 x=2;
```

Après l'analyse lexicale: Mc_int idf pvg
idf aff cst pvg

# ANALYSE LEXICALE

L'analyseur lexical est basé sur l'algorithme simple suivant :

```
Lire (ChaineEntrée) ;
Switch (ChaineEntrée)
Case (ExpReg1) : coder(« Entité1 ») ;
Case (ExpReg2) : coder (« Entité2 ») ;
....
Case (ExpRegN) : coder (« EntitéN ») ;
Default : écrire (« Erreur lexicale ») ;
```

L'implémentation d'un tel algorithme nécessitera l'écriture de centaines/milliers de lignes de code. Heureusement des outils logiciels, tel que **F(lex)**, existent pour nous faciliter la tâche.

# FLEX

Flex est un **générateur d'analyseur lexical** qui traduit notre spécification écrite dans un langage simple (Flex) en code C.


**But:**

- Reconnaître les entités lexicales du langage
- Exécuter des actions à chaque entité rencontrée (pour chaque entité lexicale identifiée par l'analyseur lexical, une action spécifique est effectuée).

# FORMAT D'UN FICHIER FLEX

Un fichier Flex est composé de trois parties séparées par '%%'.

**%{**

Définitions en langage C

**%}**

Les définitions des expressions régulières

} Partie 1

**%%**

ExpressionRégulière { Action C}

} Partie 2

**%%**

Redéfinitions des fonctions prédéfinies

} Partie 3

# PARTIE 1:

- Elle sert à donner les ER des différentes entités lexicales du langage sous la forme suivante :

**<identificateur_de_l'entité>** <expression_régulière>

-> **identificateur_de_l'entité :** doit commencer par une lettre et ne comporte que
des caractère alphanumériques, des _ ou bien des - .

-> **expression_régulière:** doit être valide

**Exemple:**

chiffre [0-9]

# PARTIE 2 : LES RÈGLES DE TRADUCTION

- Elle présente l'ensemble des actions associées à chaque entité lexicale sous la forme suivante:

  <pattern> <action>

- Un **pattern** : une expression régulière décrivant une entité lexicale
- Une **action** : c'est un code C qui sera exécuté à chaque fois qu'une entité lexicale apparaît.

**Exemple :**

- {cst} {printf ("L'entité reconnu est une constante") ; }

# Exemple 1.

## Analyseur lexicale pour : x= 5;

```
%{
   int nb_ligne=1;
%}
lettre [a-zA-Z]
chiffre [0-9]
IDF {lettre}({lettre}|{chiffre})*
cst {chiffre}+
%%
{IDF} printf ("IDF ");
{cst} printf("cst ");
"=" printf ("aff ");
";" printf("pvg \n");
[ \t]
\n {nb_ligne++; }
. printf("erreur lexicale a la ligne %d \n",nb_ligne) ;
%%

int main ()
{
 yylex () ;
 return 0;
}
```

# Commandes de compilation

- **Pour compiler le programme :**
  **flex MonTP.l**

- **cc lex.yy.c -o MonTP -lfl (Linux)**
  **gcc lex.yy.c -o MonTP -lfl (Windows)**
- **Pour exécuter le programme :**
  **./MonTP**
  **ou MonTP.exe<test.txt**
- **Pour arrêter le programme :**
  **Ctrl + c**

# Macro-actions de FLEX

| | | |
|---|---|---|
| Fonctions | yylex | Permet de lancer l'analyseur lexical |
| | yywrap | Elle est appelée par le Lexer quand il rencontre la fin du fichier. Elle doit soit obtenir un nouveau flux d'entrée et retourner simplement la valeur 0 soit renvoyer 1 signifiant que la totalité des flux a été consommée et que le lexer a fini sa tâche. |
| | yyterminate | Permet de provoquer la fin d'exécution du lexer |
| | ECHO | Affiche l'unité lexicale reconnue |
| Variables | yytext | Récupère le texte formant le lexème reconnu |
| | yyleng | Détermine la langueur du texte contennue dans yytext |
| | yylval | Est une variable globale utilisé par FLEX pour stocker la valeur correspondante au Token reconnu |
| | yylineno | Est le numéro de la ligne courante |
| | yyin | Fichier d'entrée |
| | yyout | Fichier de sortie |

# VARIABLES ET FONCTIONS PRÉDÉFINIES DE FLEX

char **yytext**[]: tableau de caractères qui contient la chaîne d'entrée en cours d'analyse.

```
IDF [a-zA-Z]([a-zA-Z]|[0-9])*
%%
{IDF} {printf ("idf: ") ; printf (" %s ",yytext) ;}
```

```
Hello
World
```

Prgrm Source

```
Idf : Hello
Idf : World
```

Prgrm Compilé

int **yyleng**: retourne la longueur de la chaîne d'entrée en cours d'analyse.

```
IDF [a-zA-Z]([a-zA-Z]|[0-9])*
%%
{IDF} {if (yyleng <=6) printf (" idf valide ") ;
else printf ("erreur lexicale : idf trop long") ;
}
```

```
Hello
HelloWorld
```

Prgrm Source

```
Idf valide
erreur lexicale: idf trop long
```

Prgrm Compilé

# VARIABLES ET FONCTIONS PRÉDÉFINIES DE FLEX

**yylex()**: c'est la fonction qui lance l'analyseur lexical. Si on change le main on doit pas oublier de l'ajouter dans le main.

```
%{
int nb_ligne=1;
%}
%%
\n nb_ligne++;
%%
int main()
{
yylex();
printf("nombre de ligne %d",nb_ligne);
}
```

```
TP
Compil
L3
ACAD
B
```

Prgrm Source

```
Nombre de ligne : 5
```

Prgrm Compilé

## Exemple 2:

Analyseur lexicale pour : x= 5; y= 5.5;

```
%{
  int nb_ligne=1;
%}
lettre [a-zA-Z]
chiffre [0-9]
IDF {lettre}({lettre}|{chiffre})*
cst {chiffre}+
reel {chiffre}+"."+{chiffre}+
%%
{IDF} {printf ("%s ",yytext);}
{reel} { printf ("%s ",yytext);}
{cst} {printf ("%s ", yytext);}
= printf ("aff ");
; printf ("pvg ");
[ \t]
\n {nb_ligne++; }
. printf("erreur lexicale à la ligne %d \n", nb_ligne) ;
%%
int main()
{
 yylex();
 return 0;
}
```

# Exemple 3:

```
%{
#include<stdio.h>
int nb_ligne=1;
%}
lettre [a-zA-Z]
chiffre [0-9]
IDF {lettre}({lettre}|{chiffre})*
cst {chiffre}+
%%
{IDF}   printf(" IDF reconnu %s \n",yytext);
{cst}   printf(" CST reconnu\n");
"="    printf(" = reconnu\n");
";"    printf(" ; reconnu\n");
[ \t]
\n nb_ligne++;
. printf("erreur lexicale a la ligne %d \n",nb_ligne) ;
%%
int main( )
{
 yyin = fopen( "test.txt", "r" );
 if (yyin==NULL) printf("ERROR \n");
   else yylex();
}
```