

COMPILATION

ANALYSE LEXICALE (2ÈME PARTIE)

EMSI - 4^{ÈME} IIR
2023/2024

Prof. M. D. RAHMANI, F.Z. TIJANE BADRI & R. FILALI

Le langage FLEX (1)

Un programme écrit en langage *LEX* construit d'une **manière automatique** un **analyseur lexical**.

Il prend en entrée un ensemble d'*expressions régulières* et de *définitions régulières* et produit en sortie un *code cible en langage C*.

Ce code en langage C doit être compilé par le compilateur du langage C pour produire un exécutable qui est un analyseur lexical correspondant au langage défini par les expressions régulières d'entrée.

Plusieurs langages dérivés de **LEX** existent;

- ✓ **Flex** produit du **C++**
- ✓ **JFlex** produit du **Java**
- ✓ **Lecl** produit du **Caml**

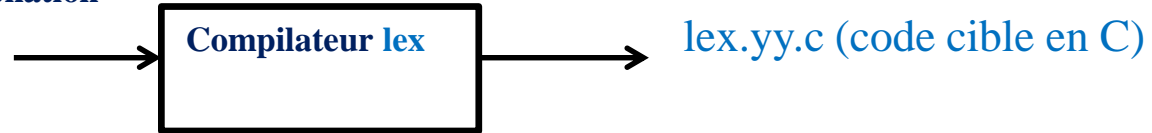
Le langage FLEX (2)

3

La **compilation** d'un code source en LEX se fait en **deux étapes**.

1/ 1ère étape de Compilation

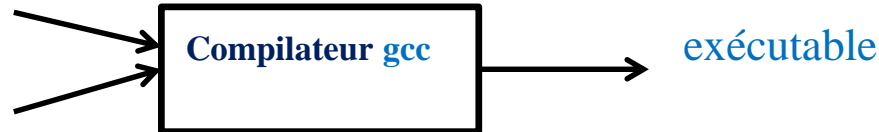
*expressions
régulières*



2/ 2ème étape de Compilation

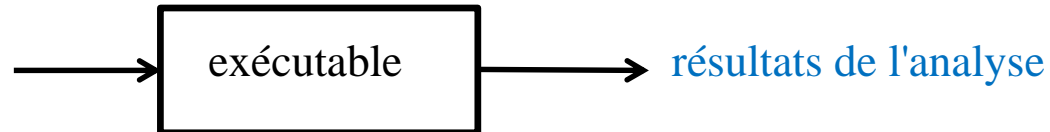
lex.yy.c

autres modules



Etape d'exécution

*source à
analyser*



1- Structure d'un programme FLEX (1)

Un programme source en langage **LEX** est constitué de **3 sections** délimitées par **%%**:

//1^{ère} partie: déclarations

%{

déclarations pour le compilateur C

%}

définitions régulières

%%

// 2^{ème} partie: règles de traduction

expressions régulières + actions à réaliser

%%

// 3^{ème} partie: fonction principale main et fonctions en C

Fonctions annexes en langage C

1- Structure d'un programme FLEX (2)

□ La 1^{ère} partie est constituée de:

- déclarations en langage C des bibliothèques, variables, structures,...
- déclarations de définitions régulières utilisables par les règles de traduction.

□ La 2^{ème} partie a la forme:

m_1 {action₁}

m_2 {action₂}

...

m_i {action_i}

...

m_n {action_n}

avec m_i une expression régulière ou une
définition régulière de la 1^{ère} partie

et

$action_i$ est l'action à réaliser par l'analyseur
lexical si un lexème est accepté par m_i .

1- Structure d'un programme FLEX (3)

- **La 3^{ème} partie:** est une **suite de fonctions en C** qui aident à l'analyse par les règles de traduction de la 2^{ème} partie.
Cette partie contient une fonction **main** du langage C.

Remarques:

- Le code doit commencer à la **1^{ère} colonne**.
- La 1^{ère} partie et la dernière sont facultatives (dans le cas de linux).
- Le fichier LEX doit avoir l'extension **.l** ou **.lex**

Exemple du code, sous unix, en langage LEX: **espace.l**

```
%%
[ \t]+      { /* supprime les espaces et tabulations */ }
bslama      { exit(0) ; }
```

1- Structure d'un programme FLEX (4)

1er programme en Flex: `espace.1`

```
%{
#include <stdio.h>
}%
%%
[ \t]+    /* supprime les espaces et tabulations */ }
bslama    {exit(0) ;}
%%
int main(){
    printf("donnez une chaine: ");
    yylex();

    return 0;
}
int yywrap() { // pour délimiter la fin de la chaine
    return 1;
}
```

Options de compilation en ligne de commande:

Avec LEX:

- 1/ `lex espace.l`
produit `lex.yy.c`
- 2/ `cc lex.yy.c -ll`
pour *library lex*

Sous Windows avec FLEX:

- 1/ `flex espace.l`
produit `lex.yy.c`
- 2/ `gcc lex.yy.c`
produit l'exécutable: **A.exe**

Reconnaissance des unités lexicales (1)

8

Soit le fragment de grammaire des instructions conditionnelles:

```
inst      —————>  si (exp) alors inst
                        | si (exp) alors inst sinon inst
                        | autre_inst
exp        —————>  terme operel terme
                        | terme
terme      —————>  id
                        | nb
```

Les terminaux de cette grammaire sont:

si, alors, sinon, (,), operel, id et **nb**.

Pour les reconnaître, nous allons d'abord donner les définitions régulières associées.

Reconnaissance des unités lexicales (2)

- Définitions régulières des terminaux de la grammaire:

A noter qu'il faut reconnaître les blancs aussi pour les ignorer.

<code>delim</code>	→	<code>espace tabulation fin_de_ligne</code>
<code>blanc</code>	→	<code>(delim)+</code>
<code>IF</code>	→	<code>si</code>
<code>THEN</code>	→	<code>alors</code>
<code>ELSE</code>	→	<code>sinon</code>
<code>operel</code>	→	<code>< <= == <> >= ></code>
<code>id</code>	→	<code>[A-Za-z][A-Za-z0-9]*</code>
<code>nb</code>	→	<code>(+ -)?[0-9]+(.[0-9]+)?((e E)(+ -)?[0-9]+)?</code>

Remarque: *Les commentaires et les blancs sont traités comme des modèles qui ne retournent aucune unité lexicale.*

2- Expressions régulières du langage FLEX

Expression	Signification	Exemple
c	tout caractère 'c' qui n'est pas un opérateur	a
\c	caractère littéral 'c'	*
"s"	chaîne littérale s	"**"
.	tout caractère sauf fin de ligne	a.b
^r	r en début de ligne	^abc
r\$	r en fin de ligne	abc\$
[s]	tout caractère appartenant à s	[abc]
[^s]	tout caractère n'appartenant pas à s	[^abc]
[a-z]	tout caractère (lettre minuscule) entre 'a' et 'z'	d
[^a-z]	tout caractère qui n'est pas une lettre minuscule	5
r*	zéro ou plusieurs r	a*
r+	un ou plusieurs r	a+
r?	zéro ou un r	a?
r{m,n}	entre m et n occurrences de r	a{1,5}
r{3,}	trois r ou plus	b{3,}
r{2}	exactement deux r	c{2}
rs	r puis s	ab
r s	r ou s	a b
(r)	r	(a b)
r/s	r quand suivi de s	abc / 123

3- Ecriture d'un analyseur lexical avec FLEX (1)

Le programme *analex.l*

1^{ère} partie:

/ déclarations en C */*

```
%{
#include<stdio.h>
%}
delim  [ \t]
bl      {delim}+
lettre [a-zA-Z]
chiffre [0-9]
id  {lettre}({lettre}|{chiffre})*
nb   (\+|\-)?{chiffre}+(\.{chiffre}+)?((e|E)(\+|\-)?{chiffre}+)?
%%
```

Remarque: - Les caractères `+`, `-` et `.` sont précédés de `\` pour les distinguer des opérateurs correspondants.

- Les accolades précisent qu'il s'agit du nom d'une expression régulière.

3- Ecriture d'un analyseur lexical avec FLEX (2)

2ème partie:

```
{bl}      { /* supprimer de la sortie */}
sinon      {printf("\n Mot cle: ELSE\n");}
si         {printf("\n Mot cle: IF\n");}
alors      {printf("\n Mot cle: THEN\n");}
{id}       {printf("\n Identificateur:%s\n",yytext);}
{nb}       {printf("\n Nombre:%s\n",yytext);}
"<="      {printf("\n Operateur relationnel: PPE\n");}
"<>"      {printf("\n Operateur relationnel: DIF\n");}
"<"       {printf("\n Operateur relationnel: PPQ\n");}
">="      {printf("\n Operateur relationnel: PGE\n");}
">"       {printf("\n Operateur relationnel: PGQ\n");}
"=="      {printf("\n Operateur relationnel: EGA\n");}
" ("       {printf("\n PO\n");}
") "       {printf("\n PF\n");}
\n         {return 0;}
.          {printf("\n%s: Caractère non reconnu\n",yytext);}
%%
```

3- Ecriture d'un analyseur lexical avec FLEX (3)

3ème partie:

```
int main() {  
    printf("donnez un texte a analyser: ");  
    yylex();  
    return 0;  
}  
  
int yywrap() { // pour delimitier la fin de la chaine  
    return 1;  
}
```

4- Compilation d'un programme FLEX

Remarques:

"." est un opérateur qui veut dire tous caractère sauf le retour à la ligne.

"^" est un opérateur pour le complémentaire d'une classe.

yytext est un pointeur sur la chaîne analysée.

yyin correspond à l'entrée

yylex() est la fonction principale du programme écrit en LEX.

Options de compilation:

Avec **LEX**: 1/ *lex analex.l* produit *lex.yy.c*
 2/ *cc lex.yy.c -ll* pour *library lex*

Avec **FLEX**: 1/ *flex analex.l* produit *lex.yy.c*
 2/ *gcc lex.yy.c*

5- Fonctions prédéfinies en FLEX

15

- ❑ **char yytext[]**: tableau de caractères qui contient la chaîne reconnue.
- ❑ **int yylex()**: fonction qui lance l'analyseur (et appelle **yywrap()**).
- ❑ **int yywrap()**: fonction toujours appelée en fin du flot d'entrée. Elle ne fait rien par défaut, mais l'utilisateur peut la redéfinir dans la section des fonctions supplémentaires. **yywrap()** retourne **0** si l'analyse doit se poursuivre (sur un autre fichier d'entrée) et **1** sinon.
- ❑ **int main()**: la fonction principale du langage C, elle doit appeler la fonction **yylex()**.

6-Outils

16

1. Site d'installation des Compilateurs C et C++ :

<https://jmeubank.github.io/tdm-gcc/>

2. Téléchargement des logiciels Flex et Bison:

<https://sourceforge.net/projects/winflexbison/>

3. Une vidéo qui peut aider pour les tests:

https://www.youtube.com/watch?v=f_2_l2Ub-SU

COMPILATION

TP4 : LE LANGUAGE FLEX

EMSI - 4^{ÈME} IIR

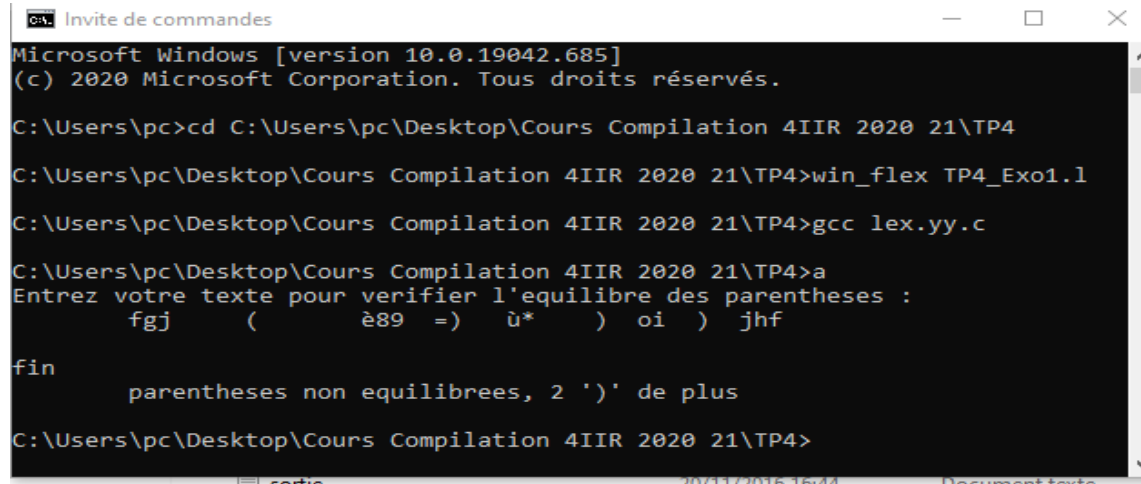
2023/2024

Profs. M. D. RAHMANI, F.Z. TIJANE BADRI & R. FILALI

Exercice 1:

18

Ecrire un programme en langage FLEX qui vérifie si une expression a autant de **parenthèses ouvrantes** que de **parenthèses fermantes**.



```
Invite de commandes
Microsoft Windows [version 10.0.19042.685]
(c) 2020 Microsoft Corporation. Tous droits réservés.

C:\Users\pc>cd C:\Users\pc\Desktop\Cours Compilation 4IIR 2020 21\TP4
C:\Users\pc\Desktop\Cours Compilation 4IIR 2020 21\TP4>win_flex TP4_Exo1.1
C:\Users\pc\Desktop\Cours Compilation 4IIR 2020 21\TP4>gcc lex.yy.c
C:\Users\pc\Desktop\Cours Compilation 4IIR 2020 21\TP4>a
Entrez votre texte pour verifier l'equilibre des parentheses :
fgj ( è89 =) ù* ) oi ) jhf

fin
parentheses non equilibrees, 2 ')' de plus
C:\Users\pc\Desktop\Cours Compilation 4IIR 2020 21\TP4>
```

Exercice 2

19

Ecrire un programme en langage *FLEX* qui traduit les abréviations contenues dans un texte donné.

On considérera les abréviations suivantes :

- **cad** : abréviation de **c'est à dire**,
- **ssi** : abréviation de **si et seulement si**,
- **afd** : **automate à états finis déterministe**.

Exécution:

```
Microsoft Windows [version 10.0.19042.685]
(c) 2020 Microsoft Corporation. Tous droits réservés.

C:\Users\pc>cd C:\Users\pc\Desktop\Cours Compilation 4IIR 2020 21\TP4

C:\Users\pc\Desktop\Cours Compilation 4IIR 2020 21\TP4>win_flex TP4_Exo2.1

C:\Users\pc\Desktop\Cours Compilation 4IIR 2020 21\TP4>gcc lex.yy.c

C:\Users\pc\Desktop\Cours Compilation 4IIR 2020 21\TP4>a
Entrez votre texte avec les abreviations :
    voici un afd qui fonctionne ssi il est bien represente !
voici un automate a etats finis deterministe qui fonctionne si et seulement si il est bien represente !
```

Exercice 3:

Ecrire un programme en langage *FLEX* qui compte le **nombre de mots** d'un texte saisi au clavier.

```
C:\Users\pc\Desktop\Cours Compilation 4IIR 2020 21\TP4>win_flex TP4_Exo3_1v1.1
C:\Users\pc\Desktop\Cours Compilation 4IIR 2020 21\TP4>gcc lex.yy.c
C:\Users\pc\Desktop\Cours Compilation 4IIR 2020 21\TP4>a
Donner un texte avec une suite de mots :
      glk                12)=ç: lmj $)=      hh
                        --Le texte contient 5 mots--
C:\Users\pc\Desktop\Cours Compilation 4IIR 2020 21\TP4>
```

Exercice 4

21

Ecrire un programme en langage *FLEX* qui:

- remplace toute suite d'espaces ou de tabulations par un seul espace,
- supprime les espaces ou tabulations de fin de ligne,
- supprime les lignes vides.
- supprime les lignes blanches.

Remarque :

- une ligne vide ne contient aucun caractère du début jusqu'à la fin de la ligne.
- une ligne blanche ne contient que *des espaces* et *des tabulations* du début à la fin de la ligne.

Exercice 5

22

Ecrire un programme en langage *FLEX* pour reconnaître et afficher le résultat:

- 1- Les opérateurs arithmétiques **+**, **-**, *****, **/**
- 2- Une suite de **F** au moins **2**
- 3- Deux "**ab**" ou plus et finissant par un '**c**'.
- 4- mots de *lettres* et *chiffres* de **longueur 5**
- 5- entiers **multiples de 10** sans les zéros inutiles aux début.
- 6- un code en Flex qui reconnaît une *chaîne de caractères* en langage C.

Exemple: "**voici, une chaîne valide en C !**"

Exercice 6

1- Ecrire un programme en langage *FLEX* qui accepte les **commentaires à la C++**.

2- Ecrire un programme en langage *FLEX* qui accepte les **commentaires à la C**.

commentaire à la C++: **// commentaire valide**\n

commentaire à la C: **/* commentaire valide */**

Exercice 7

Ecrire un programme d'un analyseur lexicale en langage **FLEX** qui reconnaît:

- ✓ les **mots clés**,
- ✓ les **identificateurs**,
- ✓ les **réels**,
- ✓ l'**opérateur d'affectation**,
- ✓ les **opérateurs de relation**,
- ✓ les **parenthèses**.