Initiation à Flex & Introduction à l'analyse lexicale

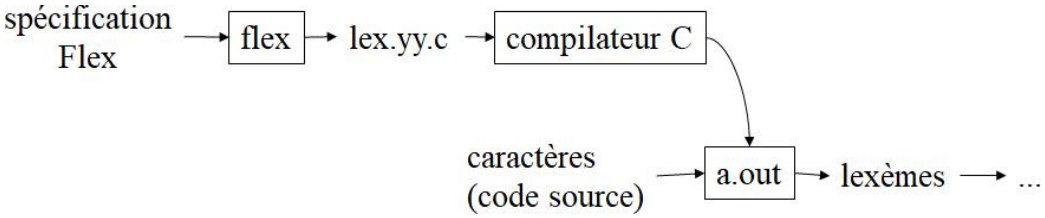
I. Objectif

L'objectif de ce TP est de s'initier à l'outil Flex afin de programmer un analyseur lexical et un analyseur syntaxique qui seront produits automatiquement à partir d'un fichier de spécification. Il s'agit entre autres de réutiliser une spécification Flex donnée, apprendre à la modifier et surtout savoir sur quelle partie de cette spécification doit-on agir pour la modifier dans le bon sens.

II. Utilisation de Flex : Rappel

FLex divise un flux de caractères en jetons. Il prend comme entrée une spécification qui associe des expressions régulières avec des actions. À partir de cette spécification, FLex construit une fonction implémentant un automate fini déterministe (AFD) qui reconnaît les expressions régulières dans le temps linéaire. Au moment de l'exécution, lorsqu'une expression régulière est appariée, son action associée est exécutée. Il permet de générer le code source en C de l'analyseur lexical à partir du fichier de spécification qui lui est fourni en paramètre. Le fichier généré s'appelle par défaut *« lex.yy.c »*. La commande pour générer le code C de l'analyseur est : *flex nom_fichier.l*

Il faut compiler ensuite le code source pour obtenir un analyseur lexical exécutable. Pour ce faire, on utilise la commande : gcc lex.yy.c -o nomfichier.exe



Remarque : Si le compilateur gcc produit un message d' erreur relatif à l' absence de la fonction yywrap(), trois solutions sont offertes :

(i) ajouter une option **%option noyywrap** en en-tête du fichier .l,

(ii) ou compiler le fichier **lex.yy.c** avec l' option de liaison **–lfl**,

(iii) ou introduire ces instructions avant le main comme illustré ci-dessous:

```
%%
int  yywrap()
{
return 1;
}
main(){
...
}
```

Le format d' un fichier de spécifications est le suivant :

```
%{
    déclarations de variable et inclusion de bibliothèques en C
%}
définitions :
 <identificateur><expression régulière>
...
%%
règles :
<expression régulière> {<commandes en C>}
%%
<programme principal et fonctions en C>
```

III. Travail demandé

Exercice 1 : Identifier le rôle d' une spécification Flex donnée

a. Que fait la spécification Flex suivante ?

```
----------file.l
%%
[a-zA-Z][a-zA-Z0-9]*    printf("[ %s ]", yytext );
%%
```

Pour tester cette spécification, utiliser l'exemple de texte source suivant :

```
----------data.txt
for( i=0; i < 10; i++ ) {
         tab[i] = 0;
         Id1= 12.3;
         Id2= .0;
       }
```

b. Construire et tester cet analyseur lexical :

```
> flex  file.l
> gcc  lex.yy.c -o  analyseur.exe
> analyseur < data.txt
```

c. Expliquer le résultat obtenu.

Exercice 2

On veut reconnaître les verbes des 1ers groupes (finissant par er) et 2èmes groupes (finissant par ir) dans un fichier texte. Le code affiche les chaînes reconnues et leurs longueurs.

1. Insérer le code suivant et l' enregistrer le fichier sous ex2.l

```
%%
[a-z]*er { printf(" : verbe du 1er groupe\n",yytext); }
    [a-z]*ir { printf(" : verbe du 2eme groupe\n",yytext);}
```

2. Compiler le fichier "ex2.l" avec la commande "flex". Le fichier "lex.yy.c" contient entre autre. La fonction "yylex( )". Permet de lire le fichier source caractère par caractère. Une fois elle reconnaît une chaîne de caractères, elle renvoie le code numérique de son unité lexicale et la stocke dans la variable prédéfinie "yytext". Celle-ci est une chaîne de caractères et sa longueur est déjà disponible à travers la variable prédéfinie "yyleng" (qui est un entier).
3. Compiler le fichier "lex.yy.c" en utilisant la commande "gcc" pour créer l'exécutable nommé "ex2.exe".
4. Créer un fichier texte "test.txt", puis y insérer le texte à analyser :

   En nuit, manger et marcher !!! En jour, manger et dormir !!!
5. Taper les commandes suivantes et faites les remarques adéquates :
   a. ex2 < test.txt
   b. ex2 < test.txt > result.txt
6. Modifier le code précédent, en ajoutant : .|\n ;
7. Exécuter de nouveau, et commenter le résultat en le comparant avec le code sans l'ajout de cette dernière instruction.
8. Insérer le code suivant et l'enregistrer le fichier sous ex2_2.l

```
%%

    [a-z]*er ECHO; printf(" : verbe du 1er groupe\n");
    [a-z]*ir ECHO; printf(" : verbe du 2eme groupe\n");
```

9. Commenter

Exercice 3

1) Donner sous FLex le code qui permet d'identifier et d'afficher le message (ceci est un opérateur !) s'il trouve un des quatre opérateurs suivants : + - / * et rien sinon.
2) Modifier ce code pour qu'il puisse identifier et afficher le message (ceci est un opérateur !) s'il trouve un des quatre opérateurs suivants : + - / * , afficher un message (erreur !) sinon, et à la fin afficher le nombre d'opérateurs trouvé.

3) Modifier ce code pour qu' il puisse identifier et afficher le message (ceci est un opérateur !) s' il trouve un des quatre opérateurs suivants : + - / * , afficher un message (erreur !) sinon, et à la fin afficher le nombre d' opérateurs et de message d' erreurs trouvés.
4) Partant du code identifiant et affichant le message (ceci est un opérateur !) s' il trouve un des quatre opérateurs suivants : + - / * , et un message (erreur !) sinon, on se propose de le compléter pour qu' il puisse en plus identifier et afficher :

- Le message : « suite de F » s' il trouve par exemple : exemple F, FF, FFF, …
- Le message : « deux ab ou plus » s' il trouve par exemple une suite de 2 ab ou plus par exemple abab, ababab, …

- Le message : « mot de longueur 3 » s' il trouve par exemple une suite de 3 caractères minuscules, majuscules ou chffres (143, avc, …).
- Le message : « Ceci est un réel » s' il trouve par exemple -12.234 ou 12.3 (chaîne composée éventuellement d' un - suivi d' au moins d' un chiffre ensuite une virgule et d' au moins d' un chiffre après la virugule.

Exercice 4 : Réutiliser une spécification Flex

a. Modifiez et complétez la spécification donnée dans l' exercice précédent (Ex1) pour que l'analyseur reconnaisse les unités lexicales suivantes :

- Les mots-clés d'un langage, par exemple : for, while, if, else…
- Les opérateurs arithmétiques: + * / -
- Les opérateurs de comparaison: < >
- Les séparateurs: ( ) { } ; , [ ]

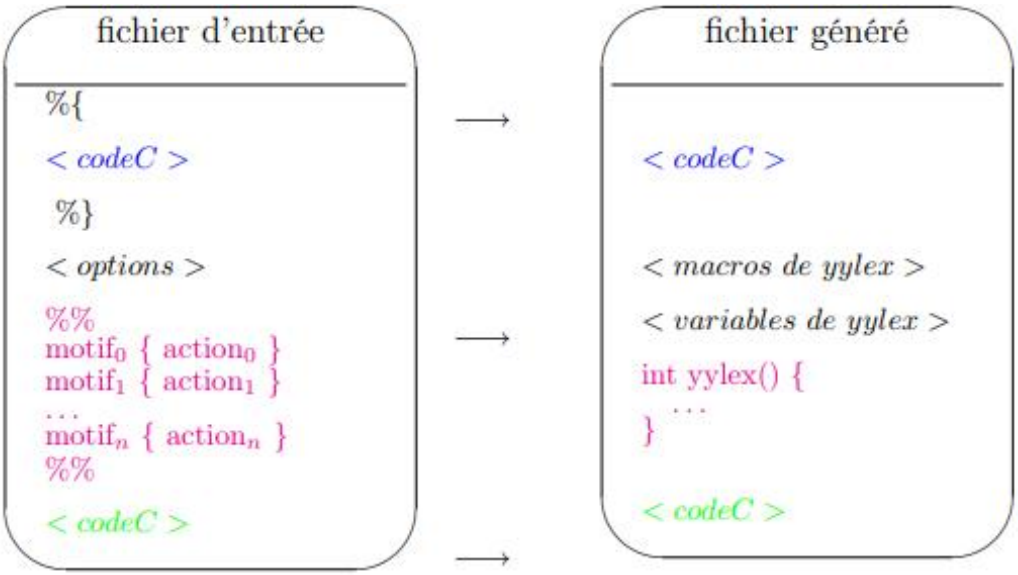
L'affichage distinguera les différentes catégories lexicales (mots-clés, opérateurs arithmétiques, de comparaison, séparateurs). Pour cela, définissez une fonction

echo() dans la section «définitions» de la spécification flex qui affichera pour chaque fragment reconnu, sa catégorie et le contenu du fragment (i.e "[KeyWord:if][Sep:(]...")

b. Complétez l'analyseur précédent :

- Ajouter la reconnaissance de valeurs numériques (entières ou flottantes)
- Ajouter une catégorie Identificateur pour reconnaître les noms de variable
- Mettre en place une règle-balai permettant d'afficher de manière particulière les caractères non attendus (quels sont-ils au fait?)
- Gérer les caractères spéciaux 'espace', 'tabulation' ou 'retour à la ligne' lorsqu'ils sont rencontrés

Annexe



Attention: les commandes %... et les motifs doivent commencer en première colonne.

2.4 Motifs

Expressions régulières (base)

`x` match the character 'x'

`.` any character (byte) except newline

`\X` if X is an 'a', 'b', 'f', 'n', 'r', 't', or 'v', then the ANSI-C interpretation of \X. Otherwise, a literal 'X' (used to escape operators such as '*')

`[xyz]` a "character class"; in this case, the pattern matches either an 'x', a 'y', or a 'z'

`[abj-oZ]` a "character class" with a range in it; matches an 'a', a 'b', any letter from 'j' through 'o', or a 'Z'

`[↑A-Z]` a "negated character class", i.e., any character but those in the class. In this case, any character EXCEPT an

uppercase letter.

`[↑A-Z\n]`	any character EXCEPT an uppercase letter or a newline
`"[xyz]\"foo"`	the literal string: [xyz]"foo
`\0`	a NUL character (ASCII code 0)
`\123`	the character with octal value 123
`\x2a`	the character with hexadecimal value 2a
`«EOF»`	an end-of-file

Expressions régulières (composition)

`(r)`	match an r; parentheses are used to override precedence
`r*`	zero or more r's, where r is any regular expression
`r+`	one or more r's
`r?`	zero or one r's (that is, "an optional r")
`r{2,5}`	anywhere from two to five r's
`r{2,}`	two or more r's
`r{4}`	exactly 4 r's
`rs`	the regular expression r followed by the regular expression s; called "concatenation"
`r|s`	either an r or an s
`r/s`	an r but only if it is followed by an s.
`↑r`	an r, but only at the beginning of a line.
`r$`	an r, but only at the end of a line (i.e., just before a newline). Equivalent to "$r/\n$".

Expressions régulières (extension)

{name} the expansion of the "name" definition

<**s**>**r** an r, but only in start condition s

<**s1,s2,s3**>**r** same, but in any of start conditions s1, s2, or s3

<*>**r** an r in any start condition, even an exclusive one.

<**s1,s2**>**«EOF»** an end-of-file when in start condition s1 or s2

`Caractères à "échaper"` / etc

`Exemples`

- un mot
- un mot commençant par une majuscule
- un entier
- une séquence sans blanc ni tabulation.
- un identifiant C
- un commentaire shell

2.5 Actions

- pour les sections "codes C" et les actions tout le langage C ou C++.
- la variable **yytext** (char*) contient le motif reconnu.
- la variable **yyleng** (int) contient le nombre de caractères de yytext.
- action par défaut: écriture du motif
- action ";" permet d'ignorer le motif

Attention: yytext est une zone mémoire fixe et donc réécrasée dans chaque action.