TP 1 – FLex

Introduction

Un squelette de projet flex est disponible sur Moodle à l'URL suivante : `https://moodle.univ-angers.fr/mod/resource/view.php?id=368307`. Vous pouvez importer le projet directement dans Qt Creator ou alors le compiler et l'exécuter manuellement depuis un terminal en suivant la procédure suivante.

- Aller dans le dossier du projet.
- Créer un sous-dossier `build` :

```
mkdir build
```

- Dans le dossier `build`, exécuter la commande suivante pour configuer le projet. Cet étape est à effectuer à chaque ajout de nouveau fichier.

```
cmake ..
```

- Compiler le projet avec la commande :

```
make
```

- L'exécutable scanner peut être lancé ensuite par la commande :

```
./flex_base
```

- Lors de l'appel à l'exécutable :
  - soit vous entrez du texte à traiter puis testez le lexeur en appuyant sur la touche Entrée,
  - soit vous nourrissez votre lexeur à l'aide d'un fichier texte en utilisant la commande :

```
./flex_base < mon_fichier.txt
```

  - Pour quitter le programme, il faut entrer au clavier la commande fin de fichier, c'est-à-dire : Ctrl + D en début de ligne (attention à ne pas taper Ctrl + C qui interromprait le programme sans le laisser terminer son travail)

La documentation de flex est disponible en utilisant la commande `info flex` ou en allant sur la page suivante : `https://westes.github.io/flex/manual/`

Exercice 1

Écrire un lexeur Flex qui supprime les espaces redondants : la sortie du programme sera le texte d'entrée où chaque séquence d'espaces est remplacée par un seul espace.

Modifier ce lexeur pour que tous les espaces apparaissant en début de ligne soient supprimés.

Exercice 2

Écrire un lexeur flex qui compte nombre de mots, de nombres et de signes de ponctuation dans un texte. Attention, dans 12 345,45 euros, il n'y a qu'un seul nombre et aucun signe de ponctuation !

Exercice 3

Écrire un lexeur Flex capable de reconnaître les entités suivantes :

- les nombres (séquences de chiffres et/ou la virgule "," pour indiquer des réels)
- les mot-clefs begin et end ;
- les identificateurs (séquences de chiffres et de lettres commençant par une lettre, et qui ne sont pas des mot-clefs) ;
- les opérateurs : +, -, * et **.

Les espaces, tabulations et retours à la ligne sont des séparateurs. Toute autre séquence de caractères qui ne forme pas une entité est considérée comme une erreur. Le lexeur affichera à l'écran la liste des entités reconnues :

```
123
-> nombre : 123
abc
-> identificateur : "abc"
$123
-> erreur
```

Exercice 4

Une chaîne de caractères est délimitée par des guillemets `"` ; à l'intérieur de la chaîne, il est possible de taper d'autres guillemets `"` à condition de les précéder de `\` (pour ne pas les confondre avec le délimiteur de fin de chaîne). Par exemple :

```
"Galilée a dit \" la terre tourne\" ".
```

Ajouter au lexeur précédent la gestion des chaînes de caractères. Pour chaque chaîne de caractères, afficher sa taille.

Exercice 5

Nous considérons un langage où les commentaires sont encadrés entre `/*` et `*/` ou entre `{` et `}`. Ajouter au lexeur précédent la gestion des commentaires de ce langage.
Quelle est la sortie générée par :

- `bla /* blo /* bli */ blu */ bligerc`
- `/** toto **/`

Exercice 6

Supposons un programme logique classique. Nous souhaitons reconnaître l'ensemble des éléments du programme. Écrire les définitions de :

- Variable (chaine de caractères commençant par une majuscule)
- Constante symbolique (chaine de caractères commençant par une minuscule)
- Constante numérique (nombre entier)
- Nombre (nombre réel)

Pour cela on définira les types simples :

- Lettre minuscule
- Lettre majuscule
- Lettre
- Chiffre

Rappel : les définitions sont déclarées avant les règles.

Structure d'un fichier Lex (.ll)

Les fichiers Lex sont divisés en trois sections, separées par des lignes commençant par deux signes %. Les sections sont les suivantes :

- La section définition qui permet de définir des macros et l'importation de fichiers headers écrits en C ou C++. Il est aussi possible d'écrire du code C/C++ dans cette partie qui sera copié tel quel dans le fichier source généré.
- La section règles associe des expressions régulières à des instructions en C ou C++. Lorsqu'un motif est reconnu le code C/C++ associé est exécuté. Cette section correspond à la définition de la fonction yylex, on peut donc accéder aux méthodes et aux membres de la classe Scanner.
- La section code C/C++, contient des instructions et fonctions en C/C++ qui seront copiées dans le fichier source généré. Ces instructions peuvent contenir du code qui sera appelé par les règles de la section précédente. Lorsque les programmes sont d'une taille importante il est préférable d'écrire le code C/C++ dans d'autres fichiers séparés et de les linker lors de la compilation. C'est le cas dans notre exemple avec le fichier scanner.hh qui contient la définition de la classe Scanner.

```
/*** Section definition ***/
%{
/* Code C qui sera ajoute avant les regles */
#include "scanner.hh"
#define YY_NO_UNISTD_H
%}

/* options Flex specifiant dans l'ordre :
   Le langage utilise
   Le nom de la classe utilisee pour le Lexeur
   Signifie a Flex de lire un seul fichier en entree
*/
%option c++
%option yyclass="Scanner"
%option noyywrap

/*** Fin de la section definition ***/
%%
/*** Section regles  ***/
[a-zA-Z]+    {
    std::cout << "un mot : " << YYText() << std::endl;
    /* YYText() renvoie une chaine de caracteres contenant le texte correspondant */
}

.            {

}
/*** Fin de la section regles ***/
%%
/*** Section code C ajoute apres les regles ***/
int main( int argc, char* argv[] )
{
    Scanner* lexer = new Scanner(std::cin, std::cout);
    while(lexer->yylex() != 0);

    // placer son code ici pour effectuer des actions apres le parsing du fichier
    std::cout << "Affiche quelque chose apres le parsing" << std::endl;

    return 0;
}
/*** Fin de la section code C ***/
```

Précisions sur l'écriture du fichier Lex (.ll)

Flex va toujours essayer de trouver le plus long mot correspondant à l'expression régulière d'une règle. Pour cela il va tester toutes les règles pour toutes les longueurs de mots possibles tant qu'il existe au moins une expression régulière qui correspond au mot de longueur testée.

Les règles sont testées dans leur ordre d'apparition mais lorsque deux expressions régulières peuvent correspondre pour un même "mot" l'expression correspondant à la longueur de mot la plus longue est prioritaire. Dans le cas où une expression régulière est plus restrictive qu'une seconde, la règle associée à celle-ci ne sera jamais appliquée. Exemple :

```
%%
[a-zA-Z] {
    std::cout << "lettre seule" << std::endl;
}

[a-zA-Z]+ {
    std::cout << "plusieurs lettres" << std::endl;
}

azerty {
    std::cout << "azerty" << std::endl;
}

. {

}
%%
```

Pour le programme précédent si le texte est le suivant :

```
a azerty azertyuiop
```

L'affichage du programme sera :

```
lettre seule
plusieurs lettres
plusieurs lettres
```

Dans notre cas, le programme commence au début de la phrase et lit (`a`), il choisi d'appliquer la première règle en premier car il n'existe pas d'autres règles pouvant s'appliquer à une longueur de mot plus grande à partir de cette position. Le programme affiche "une lettre seule". Le programme lit ensuite un espace qui ne peut correspondre qu'avec la règle par défaut. Le programme lit ensuite (`a`) mais ne s'arrête pas sur la première règle car il existe un mot plus long commençant par (`a`) et correspondant aux deux autres règles (`azerty`). Dans ce cas le programme ne va pas appliquer la troisième règle mais la seconde car elle apparaît avant dans l'ordre du programme. Le compilateur va d'ailleurs prévenir l'utilisateur lors de la compilation par le message suivant :

```
scanner.ll:XX: warning, la règle ne peut pas correspondre
```

Le programme affiche alors "plusieurs lettres". Enfin, dans le dernier cas, il n'existe qu'une seule règle pouvant être appliquée pour le mot le plus long étant (`azertyuiop`). Le programme affiche alors "plusieurs lettres".

Attention, dans une règle, l'**expression régulière** et l'**action à effectuer** (le plus souvent entre `{}`) doivent être **séparées d'au moins un espace**.

Liste des motifs autorisés dans les expressions régulières :

```
https://westes.github.io/flex/manual/Patterns.html#Patterns
```