Compilation et Analyse Lexicale
TD 6 - Correction

Licence 3 Informatique (2022-2023)

Jairo Cugliari, Guillaume Metzler
Institut de Communication (ICOM)
Université de Lyon, Université Lumière Lyon 2

jairo.cugliari@univ-lyon2.fr guillaume.metzler@univ-lyon2.fr

Exercice 1 Regardez la vidéo [1], en particulier le processus expliqué dans la partie '*Translating Source Code to Machine Code*' (dès la minute `3:38`). Vous aurez une vision d'ensemble du travail qu'on fera dans ce TP. Puis, installez dans votre séance de travail les outils `flex` et `bison` (que nous utiliserons jeudi prochain).

Pour l'installation des utilitaires, nous utilisons le gestionnaire de paquets `apt`

```
apt install flex
apt install bison
```

Exercice 2 (*Analyse lexicale*) Nous allons utiliser le générateur d'analyseur lexical `flex`. L'outil prend en entrée la définition des unités lexicales pour produire un programme `C` capable de reconnaitre ces unités et les transformer selon les règles données. Considérez cet exemple de fichier d'entrée

```
%%
[0-9]+   printf("?");
.        ECHO;
```

1. Après la ligne de début des règles (marquée par les caractères %%), la première colonne du fichier contient des expressions régulières. Que représentent elles ? La deuxième colonne contient les tokens de remplacement. Que feront ces règles ?

[1] Cette vidéo sert aussi à évaluer vos acquis du cours de Compilation. À la fin du cours, vous devriez être en mesure d'expliquer à quelqu'un qui n'a pas suivi le cours toutes les étapes mentionnées dans cette vidéo. Si ce n'est pas le cas, n'hésitez pas à retravailler le cours et à poser de questions à l'équipe enseignante

La première règle représente un motif formé par un où plusieurs chiffres. Ce motif sera remplacé par un seul caractère `?`. La deuxième règle utilise le motif générique (`.`), c'est-à-dire que le motif coïncide avec tous les caractères, pour le remplacer avec le motif d'entrée. Ces règles ont pour but de remplacer les nombres par le caractère `?` et de reproduire tout autre caractère à l'identique.

2. Créez un fichier de texte plat avec le contenu ci-dessus. Nommez le fichier `cacher_chiffres.l`.

   Faites bien attention à la syntaxe de flex. En cas de doute, n'hésitez pas à consulter la documentation en ligne.

3. Utilisez la commande `flex cacher_chiffres.l`. Elle produit un fichier dans votre répertoire de travail. Quel est le format de ce format ? Quel est son contenu ?

   Le résultat de cette étape est un fichier nommé `lex.yy.c`, contenant du code source `C`. Il s'agit du code pour créer un parseur en utilisant les règles définies dans le point 2.

4. Compilez `gcc -o cacher_chiffres lex.yy.c -ll` pour obtenir le parseur. Notez l'option -ll à la fin de la ligne, elle permet le linkage vers la librairie fournie par `flex`.

   Vous devriez voir maintenant un fichier exécutable parmi vos fichiers. Il s'agit du parseur que vous avez construit.

5. Exécutez votre parseur. Il traduira les lignes que vous donnerez selon les règles que nous avons définies.

   Le parseur vous permet maintenant d'introduite n'importe quelle chaine de caractères en entrée (le programme utilise l'entrée standard) et vous renvoie la version parsée (c'est-à-dire celle où les nombres sont cachés).

Exercice 3 (*Analyse lexicale*) Cette définition du parseur produit un résultat différent. La définition contient trois section. Une première où on définit de variables globales, la section du milieu que vous avez déjà travaillé dans l'exercice 2, puis une section finale avec une redéfinition de la fonction `main`.

```
%{
int numChars = 0, numWords = 0, numLines = 0;
%}
%%
\n {numLines++; numChars++;}
[^ \t\n]+ {numWords++; numChars += yyleng;}
. {numChars++;}
%%
int main() {
    yylex();
    printf("%d\t%d\t%d\n", numChars, numWords, numLines);
}
```

Obtenez le parseur associé, puis utilisez un fichier de texte plat de votre choix pour évaluer ce qui fait votre parseur.

Exercice 4 (*Analyse lexicale*)

```
%%
a*b      printf("1");
(a|b)*b  printf("2");
c*       printf("3");
```

Étant donné les jetons suivants et les expressions régulières qui leur sont associées, montrez quelle sortie est produite lorsque cet analyseur `flex` est exécuté sur les chaînes de caractères suivantes :

1. aaabccabbb
2. cbbbbac
3. cbabc

En créant le fichier `exo3.l` avec le contenu donnée plus haut, nous pouvons créer le parseur avec la chaîne qui suit

```
flex exo3.l
gcc -o exo3 lex.yy.c -ll
```

Puis, à l'exécution, nous insérons les trois chaines de caractères pour obtenir :

1. 132 (parsé comme aaab cc abbb)
2. 32a3 (parsé comme c bbbb a c, aucune règle ne coïncide avec le caractère a)
3. 323 (parsé comme c bab c)

Notez que la règle `a*b` dit 'un ou plus caractère a suivi d'un seul b' (et non n'importe quelle chaîne qui commence par a et fini par b). Ainsi, dans le premier cas, nous pourrions penser au motif alternatif (aaab cc ab bb), or `flex` privilégie les motifs les plus longs. Remarquez aussi que les expressions régulières suivent de règles qui dépendent du langage utilisé.