Feuille de travaux pratiques n° 2
Utilisation de flex

Lex (ou `flex` sur Linux) est un générateur d'analyseurs lexicaux. Il prend un fichier en argument contenant une description d'une grammaire rationnelle et retourne un fichier contenant un programme (la fonction C `yylex`) pouvant découper un flux de caractères en une suite de *tokens* suivant cette grammaire. Pour chaque token reconnu, une action est exécutée. Cela peut-être une instruction ou un bloc C ou bien une action prédéfinie.
La "compilation" avec Lex : *fichier*`.l` → lex → `lex.yy.c`
`lex.yy.c` contient la fonction `yylex` : *flux d'entrée* → yylex → *tokens*/*actions*

Lex est particuliérement bien adapté pour son utilisation conjointe avec le générateur d'analyseur syntaxique *Yacc*. Dans ce cas, le flux d'entrée est d'abors décomposé en un flux d'éléments lexicaux (tokens) par l'analyseur lexicale construit par Lex puis ce flux passe par l'analyseur syntaxique engendré par Yacc. De nombreux programmes utilisent des programmes générés par Lex et/ou Yacc comme des compilateurs C (cc, gcc), des interpréteurs de commandes (sh, csh), des calculateurs de bureau (bc)...

Les versions d'origine de ses générateurs ont été écrites dans les années 70 dans les laboratoires Bell de AT&T par Stephen C. Johnson (yacc), Mike Lesk et Eric Schmidt (lex). Dans ce TP, nous utiliserons **flex**, une version domaine publique de Lex et **bison**, l'implantation de Yacc par la Free Software Foundation (le groupe GNU).

Premier programme

Lex est un générateur de programme pouvant générer du code C ou C++. Ici, nous utiliserons le C. Pour utiliser Lex, nous devons saisir un petit fichier (avec votre éditeur de texte préféré) contenant la description de la grammaire et des actions associées aux règles. Voici un premier exemple que vous devrez sauvegarder sous le nom `ex1.l` (on utilise l'extenxion `.l` pour désigner les fichiers sources de Lex) :

```
%%
if     printf("si");
else   printf("sinon");
%%
main() {
  yylex();
}
```

Ce fichier doit alors être traité par le programme `flex` pour qu'il le transforme en un programme C. On peut utiliser la commande (tapée dans un terminal :

```
flex ex1.l
```

Si tout va bien (c'est-à-dire si `flex` n'a pas détecté d'erreur), vous devez trouver le fichier `lex.yy.c` dans le répertoire où vous avez lancé `flex`. On peut alors compiler ce programme en utilisant un compilateur C (sans oublier de donner l'option `-ll` pour ajouter certaines fonctions utilisées par l'analyseur) :

```
gcc lex.yy.c -ll -o ex1
```

Le résultat (l'exécutable) porte le nom `ex1` correspondant à l'option `-o ex1`. Cet exécutable lit les caractères sur son entrée standard et écrit le résulat des actions sur sa sortie standard. La commande suivante lance ce programme avec pour entrée standard le fichier `fichier.txt` et pour sortie standard l'écran :

```
./ex1 <fichier.txt
```

Normalement, vous devez voir, sur votre écran le contenu du fichier `fichier.txt` dans lequel les occurences de “if” ont été remplacées par “si” et celles de “else” par “sinon”. En fait, le fichier Lex est transformé en un programme C contenant la fonction `main` définie dans `ex1.l`. Cette fonction principale appelle la fonction `yylex` qui a été générée par Lex à partir de la spécification :

```
if      printf("si");
else    printf("sinon");
```

Cette definition précise que si l'analyseur reconnaît le token “if”, il doit exécuter l'action `printf("si");` puis continuer l'analyse au caractère suivant. S'il reconnaît le token “else”, il doit exécuter `printf("sinon");` et continuer l'analyse. Sinon, il affiche le prochain caractère et continue l'analyse avec le caractère suivant. (si aucune règle ne peut s'appliquer, l'analyseur envoie le premier caractère sur la sortie standard et continue la reconnaissance avec le caractère suivant). La fonction s'arrête lorsqu'on arrive à la fin du fichier d'entrée.

Définition des fichiers sources de Lex

Un fichier source Lex est composé de trois parties délimitées par les caractères `%%` :

Définition
`%%`
Règles
`%%`
Fonctions et routines

Toutes les lignes précédant le premier délimiteur `%%` sont considérées comme une définition par Lex ou bien, pour les lignes comprises entre `%{` et `%}` ou celles débutant par un espace ou une tabulation, comme du code C copié tel quel dans le fichier résultat. Une définition sert à introduire des abréviations pour des motifs pouvant apparaître dans les règles. Ainsi, on peut spécifier l'ensemble des chiffres par la définition `D [0-9]` et celle d'exposant pour un nombre par la définition `E [DEde][-+]?{D}+`. Ces définitions peuvent alors être utilisées dans une règle comme une sorte de macro en utilisant *{nom}* avec *nom* le nom de la définition (ci-dessus `{D}` et `{E}` respectivement pour un chiffre et un exposant).

La seconde partie d'un source Lex, délimitée avant et après par `%%` comporte une liste de règles lexicales dont chacune est composée de deux parties :

- un **motif** (*pattern* en anglais) représentant une expression régulière. Une suite de caractères donnée correspond ou non à un motif. Par exemple, le motif `[A-Za-Z]+` correspond à toutes les suites d'au moins un caractère et composées seulement de lettres. Les motifs doivent débuter une ligne (pas d'espace ni de tabulation).
- une **action** associée au motif et qui est exécutée lorsque l'analyseur reconnaît le motif. Cett action peut être une instruction C simple comme `printf("si");`, un bloc d'instruction C délimité par `{` et `}` (sans `;` à la fin), une action prédéfinies comme `ECHO;` ou bien être vide en donnant un simple point-virgule `;`. Si l'action ne comporte pas d'instruction `return`, d'appel récursif à l'analyseur `yylex();` et si le buffer d'entrée n'est pas modifié, l'analyseur continue avec la lecture du prochain token. On peut associer la même action à plusieurs motifs en utilisant le caractère `|` à la place de l'action.

Finalement, la dernière partie d'un source Lex, après le second `%%`, est recopié tel quel à la fin du programme `lex.yy.c`. Il sert à définir des fonctions utiles à l'analyseur. Dans le premier programme, nous y avions défini la fonction `main` pour obtenir un programme C complet. Voici un exemple de source Lex :

```
%{
 int nombres;
 int mots;
%}
D [0-9]
E [DEde][-+]?{D}+
%%
{D}+\.?{D}*{E}?     |
\.{D}+{E}?          nombres++;
lex                 ECHO;
[A-Za-z]+           { mots++; }
.|[\n]              ;
%%
```

```
main() {
  nombres = 0;
  mots = 0;
  yylex();
  printf("\nNombres : %d, Mots : %d\n", nombres, mots);
}
```

Ce source Lex déclare deux variables globales `nombres` et `mots`. Ensuite, on trouve les deux définitions pour `{D}` et `{E}`. Après, nous avons les différentes règles. Les deux premiers motifs `{D}+\.{D}*{E}?` et `\.{D}+{E}?` reconnaissent des nombres (avec exposant optionnel). Ils déclenchent la même action qui consiste à incrémenter la variable `nombres` de un. Le troisième motif `lex` utilise l'action prédéfinies `ECHO` qui affiche le token sur l'écran. Le motif suivant reconnaît un mot. On incrémente `mots` de un. Enfin, le denier motif est constitué de n'importe quel caractère. Dans ce cas, aucune action n'est réalisée. Cela permet de ne pas utiliser l'action par défaut qui est d'afficher le premier caractère lorsqu'aucun motif ne convient.

Exercice 2.1

Saisir ce second source Lex puis le tester.

Motifs

Les motifs sont des expressions régulières comme pour la commande `grep` (voir le TP 1). Quelques spécificités sont toutefois à noter :

- Une suite de caractères correspond à la suite de caractères dans le flux d'entrée. On peut placer ces caractères (ou une partie seulement) entre guillemets (en particulier, cela est utile si l'on inclut un blanc, une tabulation ou un opérateur Lex `\ [ ] ^ - ? . * + | [ ] $ / { } % < >`. On peut aussi, dans ce dernier cas utiliser ce caractère précédé de `\`. Le caractère tabulation se code avec `\t` et le passage à la ligne `\n`. Ainsi, le motif `xyz"++"\n` correspond aux chaînes constituées des caractères `x y z + +` et finissant par le caractère "passage à la ligne" (cette chaîne doit donc se trouver en fin de ligne).
- Une classe de caractères est indiquée avec des crochets. Ainsi `[+a-z0-9]` correspond au caractère `+`, à toute lettre minuscule ou à tout chiffre. Dans `a-z` et `0-9`, le symbole `-` a permis de spécifier tous les caractères compris entre deux caractères . Si l'on désire ce caractère `-`, il faut le placer en tête de la liste comme dans `[-+0-9]`. On peut placer un blanc entre crochet sans le faire précéder de `\`. On obtient la liste opposée des caractères spécifiés en ajoutant `^` après le `[`. Ainsi, `[^abc]` correpond à tous les caractères sauf `a`, `b` et `c`, même les caractères de contrôle comme les passages à la ligne.
- `.` correspond à tous les caractères sauf le passage à la ligne. Il est donc équivalent à l'expression `[^\n]`.
- `?` : zéro ou une occurrence de l'expression qui le précède.
- `*` spécifie la répétition d'un nombre quelconque de fois d'une expression.
- `+` spécifie une répétition *non vide* d'une expression.
- `|` permet d'avoir une alternance entre plusieurs motifs.
- `(` et `)` permettent de grouper des expressions comme dans `(ab|cd+)?(ef)*` qui correspond à des chaînes comme `abefef`, `efefef`, `cdef` ou `cddd` mais pas à `abc`, `abcd` ou `abcdef`.
- `^` au début du motif indique que le motif ne peut être reconnu qu'en début de ligne.
- `$` à la fin du motif spécifie une reconnaissance en fin de ligne (juste avant un caractère "passage à la ligne").
- `/` permet de spécifier un contexte à droite. Ainsi `ab/cd` reconnaît les caractères `ab` seulement s'ils sont suivis des caractères `cd`. En fait, le motif `ab$` est équivalent à `ab/\n`.
- `{`*nom*`}` utilise la définition de *nom*.
- `{`n`}` (`{`n`,`m`}`) correspond à un motif répété n fois (de n à m fois). Ainsi `a{1,5}` correpond à une chaîne composée de 1 à 5 `a`.

Exercice 2.2

Ecrire et tester un source Lex permettant de reconnaître les motifs suivants. Ce programme devra afficher les motifs reconnus et seulement ceux-ci ainsi que le numéro du motif reconnu entre parenthèses.

1. Le mot `Yacc`.
2. Le mot `From` en début de ligne.
3. Une ligne composée d'un signe + répété entre 3 et 10 fois.

4. Un identificateur composé de majuscules, minuscules, du caractère _ et de chiffres mais ne commençant pas par un chiffre.
5. un nombre avec virgule flottante.

Reconnaissance

La reconnaissance par Lex s'arrête lorsqu'il arrive à la fin du flux. La fonction `yylex` retourne alors la valeur 0. On peut aussi sortir de cette fonction par une instruction `return` dans une action.

Lorsqu'on spécifie une suite de motif, il se peut que plusieurs motifs différents peuvent s'appliquer au même moment et cela sur des chaîne de longueur différente. Par exemple, avec les deux motifs `a[bc]+` et `ac*` et la chaîne `accbcade` comme flux d'entrée, les chaînes `ac`, `acc`, `accb` et `accbc` correspondent au premier motif tandis que `a`, `ac` et `acc` correspondent au second. **Lex choisit toujours le motif pour avoir la chaîne reconnue la plus longue**. Dans l'exemple, cela serait le pemier motif et le token serait `accbc`. **Si deux motifs permettent d'obtenir le token le plus long, alors c'est la première règle qui est choisie**. L'ordre des motifs dans le source Lex a donc une importance dans ce cas.

Actions

Nous avons vu qu'une action peut être une instruction ou un bloc C ou bien une action prédéfinie. Dans une action, on peut utiliser les variables suivantes :

— `yytext` de type `char*` et pointant sur le token reconnu (le premier caractère).

— `yyleng` de type `int` et retournant la longueur du token.

On peut replacer une part des caractères composant le token en utilisant la fonction `yyless(nb)` qui replace `nb` caractères dans le flux d'entrée. Par exemple, `yyless(yyleng-1)` ne conserve que le premier caractère du token. L'analyse continuera donc avec le second caractère.

La routine `yymore();` permet d'indiquer que l'on doit conserver le token reconnu dans le buffer et le concaténer avec le token a reconnaître.

On peut utiliser l'instruction `return` n`;` dans une action ce qui retourne la valeur de n à la fonction qui a appelé `yylex`. Un appel postérieur à cette fonction continuera l'analyse lexicale avec le token suivant.

Exercice 2.3

Ecrire un programme qui compte le nombre de caractères, de mots et de lignes d'un fichier (voir l'utilitaire `wc`).

Exercice 2.4

Ecrire un programme qui transforme les nombres entiers d'un fichier en les multipliant par 10. Il faudra éviter que les identificateurs du genre `Id33` ne soient modifiés.

Exercice 2.5

Le but de cet exercice va consister à écrire un petit calculateur en notation polonaise inversée. Cette notation utilise une pile d'arguments (de longueur fixe) et des opérations sur cette pile. Par exemple, l'expression "10 3 6 + * 2 /" comporte 7 opérations qui sont :

— On place 10 sur la pile.

— On ajoute 3 à la pile (elle est donc composée des deux nombres 3 puis 10).

— On place un troisième nombre 6 sur la pile.

— On exécute l'opération + qui consiste à dépiler les deux dernières valeurs de la pile (ici 6 et 3), à calculer la somme de ces deux nombres puis à empiler le résultat 9. Après cette opération, la pile est composée des deux nombres 9 puis 10.

— De même, on exécute l'opération * qui remplace les deux valeurs 9 et 10 de la pile par leur produit 90.

— On empile alors la valeur 2.

— On divise 90 par 2 pour donner 45. La pile ne contient plus que la valeur 45.

Ici, vous allez écrire un source Lex pour évaluer une expression. Les valeurs et opérateurs peuvent être séparés par des blancs, des tabulations ou des caractères passage à la ligne. Les nombres seront au format entier ou scientifique (à vous de trouvez les bons motifs). La pile du calculateur sera un tableau de 10 `double` initialisé à zéro. Les opérations d'empilage et de dépilage seront un décalage des valeurs du tableau (donc une des valeurs sera perdue). Vous pourrez mettre les opérations numériques usuelles (+ - * /) mais aussi des opérations de changement de signe (+/-), d'effacement des valeurs de la pile, une gestion de cases mémoire, des fonctions ln, exp, sin, cos... Le programme devra affiché la valeur se trouvant en haut de la pile.