TP n° 1
Introduction à Lex et Yacc

1 Introduction

Lex et Yacc sont des outils de génération d'analyseurs lexicaux et syntaxiques. Leurs implémentations GNU se nomment respectivement Flex [1] et Bison [2]. À partir d'une grammaire qui leur est propre, ces outils vont générer des analyseurs en langage C. L'objectif de ce TP est d'utiliser l'outil Lex/Yacc pour générer l'implémentation en langage C.

Attention, cette introduction n'a pas pour ambition d'être exhaustive sur les possibilités offertes par ces outils, mais simplement vous fournir les briques de bases pour démarrer.

1.1 Lex

Lex sert à générer des analyseurs lexicaux. Un analyseur lexical va déclencher des actions quand des motifs sont reconnus dans le flux entrant. Le but de l'analyse lexicale est de lire des suites de caractères et de les reconnaître comme des mots clés, des variables, des nombres, etc. Les motifs sont représentés par des expressions rationnelles (souvent mal traduites de l'anglais en expressions régulières - regexp).

Un fichier lex (.lex) est composé de trois parties séparées par '%%' :

1 Les définitions
2 %%
3 Les règles
4 %%
5 Le code utilisateur

Les parties *définitions* et le *code utilisateur* vont contenir du code C. Généralement la partie des définitions peut, comme son nom l'indique, contenir des définitions qui peuvent être de deux types :

- soit des déclarations ou des #define associés code C,
- soit des définitions d'expression régulières pour lex.

Lex différencie les deux types de définitions par la mise en forme qu'on donne aux lignes. Par défaut, les définitions pour lex sont des définitions qui commencent au premier caractère de chaque ligne. Les définitions pour le code C, par exemple des #include, des déclarations de variables, types, ou constante ('#define') doivent quant à elles être mises en forme différemment, et on a pour cela le choix entre deux méthodes :

- soit faire commencer la ligne par des tabulations
- soit encapsuler la ligne entre '%{' et '%}'

La partie centrale, *règles* est la plus importante et constitue le coeur de l'analyseur. On peut définir une action associée à chacune des expressions régulières qu'on recherche, une action qui sera effectuée

quand le programme aura détecté que le texte en entrée correspond au motif décrit par l'expression régulière.

Les expressions régulières suivent un format précis, décrit par exemple ici [3]. En cas d'oubli, on peut de toute façon retrouver la syntaxe dans la documentation de la commande *grep*, en tapant « *man grep* » [4].

A l'aide des règles, on peut reconnaître toutes sortes d'expressions régulières. Par exemple, ici on veut reconnaître les entiers.

```
ENTIER   [0-9]+
MOT      [a-zA-Z]+
%%
{ENTIER}   printf("(entier)");
{MOT}      printf("(mot)");
```

Le *code utilisateur* contient la fonction *yylex* qui lance explicitement l'analyse.

1.2 Yacc

YACC (Yet Another Compiler Compiler) génère des analyseurs syntaxiques. Un analyseur syntaxique va déclancher des actions quand des structures grammaticales sont reconnues.

Le format d'un fichier *yacc* est identique au format d'un fichier *lex*.

Une petite description de chaque partie :

- *définitions* : tout comme pour *lex*, le code éventuel entre un *'%{'* et un *'%}'* sera copié au début du fichier C produit ; il peut également y avoir diverses sortes de définitions. Par exemple, il peut contenir la déclaration des mots reconnus par la grammaire et qui sont principalement envoyés par l'analyseur lexical. Ces déclarations sont précédées de %token et peuvent être écrites sur une seule ligne ou sur plusieurs :

```
1 %token   Liste de mots-clés
2 %token   Liste de mots-clés
```

- *règles* : comme pour *lex*, un certain nombre de combinaisons motif/action ; mais les motifs sont ici ceux d'une grammaire hors-contexte, et non de simples expressions régulières comme c'était le cas dans *lex*.
- *code utilisateur* : des fonctions diverses et variées ; l'ingrédient principal est l'appel à *yyparse*, la fonction d'analyse grammaticale (au lieu de *yylex* comme c'était le cas dans *lex*) .

Par exemple, en reprenant le dernier exemple donné à la section précédente sur *lex*, nous avons deux mots-clés (NUM et TRUC) à déclarer :

```
1 %token <int>NUM
2 %token MOT
```

Le <int> collé avant NUM indique que NUM est de type entier. Dans cette section, il faut également déclarer par quelle règle la grammaire commence à l'aide de %start et quel est sont type. Par exemple si notre première règle s'appelle *main* et que la structure renvoyée par l'analyseur est de type Arbre, on aura la déclaration suivante :

```
1 %start main
2 %type <Arbre>main
```

La partie *règles* contient la grammaire qui est décrite sous une forme proche de BNF[1]. Les règles syntaxiques sont écrites sous la forme suivante :

```
1  Non-terminal :
        2    Symbole ... Symbole {Action sémantique}
        3 | ...
        4 |  Symbole ... Symbole {Action sémantique}
        5 ;
```

Comme son nom l'indique <Non-terminal> représente un symbole non- terminal de la grammaire qui constituera ici le nom des règles. Symbole désigne un symbole terminal ou non-terminal. Un symbole termial sera ici un caractère ou un mot-clé déclaré précédemment.

Les actions sémantiques sont des parties de code C. Elles utilisent les variables $1, $2, etc. Pour représenter les valeurs prises par les symboles. $1 est par exemple la valeur du premier symbole.

Exécution de flex et bison

Pour obtenir le code généré, nous appelons nos analyseurs lexicaux et syntaxiques en tapant, sous UNIX les commandes suivantes :

Flex	**Couple (Flex/Bison)**
Shell# flex ex1.l Shell# gcc lex.yy.c -lfl -o nom de sortie Shell# ./nom de sortie	Shell# flex ex1.l Shell# bison –d ex1.y Shell# mv ex1.tab.h ex1.h Shell# gcc lex.yy.c ex1.tab.c -o nom de sortie Shell# ./nom de sortie
Bison	
Shell# bison -v ex1.y Shell# gcc ex1.tab.c	

Où «ex1.l » un fichier *lex* et «ex1.y » un fichier *yacc*. Le fichier «ex1.y » doit contenir les définitions des fonctions :

- int yylex() qui renvoie le prochain caractère à traiter.
- void yyerror(char* s) appelée en cas d'erreur de syntaxe.

2 Exemples (Rappel)

Les exemples suivants vous sont donnés à titre d'échauffement à la programmation en C et à l'utilisation de flex et bison.

Travail à effectuer

– Récupérer sur madoc l'archive «exemples.zip» contenant les fichiers d'exemples. ;
– Exécuter les programmes ;
– Comprendre le fonctionnement des exemples ;
– Créer d'autres fichiers d'entrée ;

[1]Backus–Naur Form, grammaire décrite à l'aide de règles de dérivations de telle façon que les symboles n'apparaissant pas dans la partie gauche de la règle sont les symboles dits terminaux.

– Modifier les programmes pour ajouter des fonctionnalités et étudier leur comportement.

3 Exercices

Exercice 1 (flex)

Ecrire 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 (flex) :

Ecrire un lexeur flex qui compte le 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 (bison) :

Ecrire un programme bison qui lit une ligne de texte et affiche OK sur l'écran si la ligne peut être générée par la grammaire :

$$\begin{cases} list \rightarrow (in) \mid a \\ in \rightarrow list \mid in, list \end{cases}$$

Il faudra définir une fonction *yylex* qui fournit à bison le prochain caractère (on pourra utiliser la fonction *getchar*).

Exercice 4 (bison) :

Ecrire un parseur qui accepte les mots générés par la grammaire

$$\begin{cases} list \rightarrow DEBUT\ in\ FIN \mid ID \\ in \rightarrow list \mid in\ SEPARE\ list \end{cases}$$ Où :

-- le terminal DEBUT peut être le caractère (ou le mot-clef begin ;
-- le terminal FIN peut être le caractère) ou le mot-clef end ;
-- le terminal SEPARE peut être, . ; ou : ;
-- le terminal ID est un identificateur quelconque ;
-- les espaces, tabulations et retours à la ligne sont ignorés.

Cette fois, la fonction yylex, complexe, sera générée automatiquement par flex.

Références

[1] Site officiel de Flex : http://flex.sourceforge.net/
[2] Site officiel de Bison : http://www.gnu.org/software/bison/
[3] http://www.infeig.unige.ch/support/cpil/lect/lex/node2.html
[4] http://www.linux-kheops.com/doc/man/manfr/man-html-0.9/man1/grep.1.html