Travaux Dirigés Compilation: TP1

Informatique 2ème année. ENSEIRB 2012/2013

——

Une fiche de présentation de yacc/bison est disponible par le document refcard.pdf.
Une documentation plus exhaustive est disponible par le document lexnyacc.pdf, concernant `lex` et `yacc`.

Utilisation de lex/flex

L'outil `lex` est un outil qui permet de générer des analyseurs lexicaux. On lui donne un fichier d'entrée, dont la syntaxe est décrite après, et il construit à partir d'expressions régulières, un automate les reconnaissant. Pour chaque expression régulière, on peut déclencher une action, qui sera décrite par du code C. La fonction générée par `lex` qui fait l'analyse lexicale s'appelle `yylex()`. Quand l'analyseur lexical est associé à un analyseur syntaxique, l'action déclenchée à chaque expression régulière consiste à retourner le lexème reconnu.

Un fichier d'entrée pour `lex` suit le squelette suivant. On utilise le suffixe `.l` pour ce type de fichiers :

```
%{
#include <stdio.h>
... code C optionnel ...
%}
Nommage d'expression regulieres
%%
Liste des expressions regulières / actions
%%
... code C optionnel ...
```

On utilisera la notation suivante pour écrire les expressions régulières :

Patterns de base :	Correspond à
x	le caractère x
.	n'importe quel caractère, sauf retour chariot
$[xyz...]$	n'importe quel caractère parmi $x, y, z, \ldots$
$[x-z]$	n'importe quel caractère entre les caractères x et z, dans l'ordre ASCII
Opérateurs de répétition :	
$R?$	Un R ou rien
$R*$	Zéro ou plus occurrences de R
$R+$	Une ou plus occurences de R
Composition :	
R_1R_2	R_1 suivi par R_2
$R_1\|R_2$	soit un R_1, soit un R_2
Groupement :	
(R)	Juste R.

Pour les caractères ayant dans cette syntaxe une valeur spéciale (comme $[,], (,), +, *, ., |, ?$), on fait précéder ce caractère par un backslash pour désigner le caractère.

La liste des expressions régulières à reconnaitre, avec leur action, consiste en une liste de lignes de la forme :

```
expression-reguliere    { code C a executer quand elle est reconnue }
```

Dans le code C associé à une action régulière, la variable prédéfinie `char *yytext` contient la chaine de caractères correspondant à l'expression reconnue.

```
%{
#include <stdio.h>
int num_lines = 0, num_chars = 0;
%}
%%
\n      {  ++num_lines; ++num_chars; }
.       { ++num_chars; }
```

```
%%

int main() {
  yylex();
  printf( "# of lines = %d, # of chars = %d\n",
          num_lines, num_chars );
}
```

L'ordre des expressions est important : `lex` cherche à reconnaitre l'expression régulière correspondant au mot le plus long, puis en cas d'égalité, prend la première expression apparaissant dans le texte.

On peut donner des noms à des expressions régulières de la façon suivante, avec une définition par ligne. Par exemple,

```
DIGIT [0-9]
```

définit un chiffre. Ce nom peut être utilisé dans les autres expressions régulières en entourant le nom d'accolades : `{DIGIT}+` reconnait les nombres.

▸**Exercice 1.** *Prise en main de lex*

1. *Recopier l'exemple précédent, qui compte le nom de lignes et de caractères d'un fichier. Compiler le fichier avec*

   ```
   lex monfichier.l
   ```

   puis compiler avec gcc le fichier `lex.yy.c` généré en un binaire `count`, avec l'option `-ll` (pour lex) ou `-lfl` (pour flex). Pour l'utiliser, faire `count < monfichier`.
2. *Ecrire un fichier de description lex qui affiche chaque nombre à virgule flottante d'un fichier. Pour cela, on utilisera la variable prédéfinie `yytext`, déclarée comme tableau de `char` et qui contient à chaque action la chaine de caractères reconnue.*
3. *Ecrire un fichier de description lex qui affiche chaque titre de section, et de subsection dans un fichier latex.*

Utilisation de yacc/bison

Un fichier yacc `.y` décrit une grammaire algébrique. La syntaxe est la suivante :

```
%{
 ... code C initial ...
%}
 ... declaration des lexemes et des types ...
%%
... regles de grammaire ...
%%
... Code C optionnel
```

La syntaxe de chacune des parties est exposée sur la fiche de présentation.

On considère la grammaire décrite en yacc suivante :

```
%{
#include <stdio.h>
#include "y.tab.h"
%}

%token ID N

%%

S
: I S
| I
;
I
```

```
: ID '=' E ';'
;
E
: T '+' E
| T '-' E
| T
;
T
: F '*' T
| F
;
F
: N
| ID
| '(' E ')'
;
%%
int main (int argc, char *argv[]) {
    yyparse ();
    return 0;
}
```

Cette grammaire utilise des lexèmes pour `ID` et `N`, ainsi que certains caractères comme `+,=,-,*,(,)`.

▸**Exercice 2.**

1. *Ecrire un analyseur lexical qui reconnait des expressions régulières correspondant à ces lexèmes. Pour retourner à yacc un lexeme, lex retourne le type du lexème (avec un* `return`*) défini comme une constante par les* `%token`*. Par exemple, une action dans l'analyseur lexical se terminant par* `return N;` *indique à l'analyseur syntaxique que le lexème de type* `N` *est reconnu. Pour les lexèmes de ponctuation, on peut utiliser la valeur du caractère comme type de lexème : '+' est un lexème de type '+' par exemple.*
2. *Générer avec lex le fichier* `lex.yy.c`*, générer l'analyseur syntaxique avec yacc (ou bison) :*

   ```
   yacc monfichier.y
   ```

   puis compiler les fichiers C générés. Vérifier que le programme produit lit bien les expressions bien formées et détecte les erreurs.
3. *On s'interesse maintenant à la valeur d'un lexème. Cela concerne* `N` *et* `ID`*. Pour le premier, c'est un entier, pour le second, une chaine de caractères. La valeur d'un lexème est transmise par l'analyseur lexical à l'analyseur syntaxique par une variable globale,* `yylval`*. Le type de cette variable est par défaut un* `int`*. On peut le modifier en ajoutant dans le fichier yacc une déclaration* `%union`*, qui définit le type de* `yylval` *comme étant une union. Modifier l'analyseur lexical pour qu'il renvoie bien la valeur des lexèmes.*
4. *On va maintenant définir des actions sémantiques pour la grammaire. Après chaque règle de grammaire, on peut écrire un bloc de code C. Dans ce bloc, les variables spéciales* $\$1, \$2, \ldots$ *font références aux valeurs de l'attribut pour chaque terme à droite de la règle.* $\$\$$ *concerne la valeur de l'attribut de la variable à gauche de la règle. Le type de cet attribut est le même que celui déclaré par* `%union`*. Définir les actions sémantiques permettant de stocker, pour chaque identificateur, sa valeur calculée. On affichera toutes les valeurs des identificateurs à la fin de l'exécution. Par défaut, les valeurs des identificateurs sont égales à* 0*.*