Correction TD 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.*

✂ ----------

2- Une fonction `main` par défaut est ajouté, ne fait qu'appeler `yylex()`. Lorsqu'un caractère ne correspond à aucune expression régulière, il est affiché (par défaut). Les expressions pour le retour chariot et '.' capturent tous les caractères qui ne sont pas dans la premiere expression reguliere.

```
%{
#include <stdio.h>
%}
%%
[+-]?[0-9][0-9]*\.[0-9]*(e[+-]?[0-9][0-9]*)? { printf("%s\n",yytext); }
\n      { }
.       { }

%%
```

3- Pas de difficulté spéciale.

```
%{
#include <stdio.h>
int section=0;
int subsection=0;
%}
S "\\section{"
T "\\subsection{"
%%
{S}[^}]* { printf("%d- %s\n",++section,yytext+9); subsection=0; }
{T}[^}]* { printf("%d.%d- %s\n",section,++subsection,yytext+12); }
\n      { }
.       { }

%%
```

---------- ✂

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.*

✂ ----------

1. Ce qu'on ne voit pas : yacc génère un code (`lex.yy.c`) qui appelle en boucle `yylex` pour lui fournir les lexèmes.

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
%%
[a-zA-Z_][a-zA-Z_0-9]*  { return ID; }
[+-]?[0-9][0-9]*        { return N; }
\n                      { }
[+\-*=;()]              { return yytext[0]; }
.                       { }
%%
```

2. Il suffit de lancer l'exécutable généré puis de rentrer sur l'entrée standard des mots acceptés par la grammaire. On quitte avec Ctrl-D.
3. Il faut ajouter dans le fichier yacc la déclaration :

```
%union{
 int n;
 char *s;
}
```

Puis dans le fichier lex, les actions deviennent :
- Pour N : `{ yylval.n=atoi(yytext); return N;}`
- Pour ID : `{ yylval.s=strdup(yytext); return ID;}` . Il faut recopier `yytext` car sa valeur est changée à chaque expression régulière. Il est nécessaire d'ajouter un include, `#include <string.h>` pour `strdup`.

Notons qu'on ne fait rien pour l'instant de ces valeurs.

4. Pour le calcul, il faut stocker dans une table la valeur des variables. Ici, on fait une table de hachage rudimentaire. Notons les déclarations `%token` et `%type` qui définissent le champ de l'union pour typer l'attribut ou la valeur du lexème par défaut. Si on omet ces déclarations, il faut ajouter pour chaque accès d'un $i le nom du champ concerné (`.n` ou `.s`). Enfin, le `free` est le pendant du `strdup` dans le fichier lex, qui alloue de l'espace memoire avec malloc.

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

  int hash_table[101]; // initialisé par défaut à 0
  int hash(char *c) {
    int n=0;
    while(*c!='\0') n=n+8* *c++;
    return n%101;
  }
%}

%token <s> ID
```

```
%token <n> N
%type <n> E T F
%union{
  int n;
  char *s;
 }
%%

S
: I S
| I
;
I
: ID '=' E ';'  { hash_table[hash($1)]=$3; printf("%s = %d\n",$1,$3); }
;
E
: T '+' E     { $$=$1 + $3; }
| T '-' E     { $$=$1 - $3; }
| T           { $$=$1; }
;
T
: F '*' T     { $$=$1 * $3; }
| F           { $$=$1; }
;
F
: N           { $$ = $1 ; }
| ID          { $$ = hash_table[hash($1)]; free($1); }
| '(' E ')'   { $$ = $2 ; }
;
%%
int main (int argc, char *argv[]) {
    yyparse ();
    return 0;
}
```