Licence Fondamentale Sciences de l'informatique	Classes : LGLSI-2
TP n°3 Compilation	
Objectifs	**Analyse lexicale et syntaxique avec Flex et Bison**

TP 3 : Flex et Bison

Exercice 1: (Un parseur sans création d'arbre syntaxique abstrait)
Attention, avec flex/bison, la séparation lexeur/parseur est différente du cours: on fait le maximum dans le parseur, et le lexeur n'est en apparence utilisé que pour les tokens non triviaux ; cela n'empêche pas que l'on passe toujours par le lexeur.

1. Créez trois fichiers :
   - **main.c** qui est le programme exécutable,
   - **lexeur.l** qui génère le lexeur,
   - parseur.y qui génère le parseur.
2. Dans main.c, on se contente d'essayer de parser, et d'afficher si l'annalyse syntaxique à réussie ou échouée :

```c
/* file main.c :: limited to yyparse() call and printed result */
/* compilation: gcc -o main main.c parseur.tab.c lex.yy.c */
/* result: main = syntaxic analysis */
/* usage: ./main < input.txt */
#include <stdio.h> /* printf */
#include <stdlib.h> /* exit */
int main(void)
{
if (yyparse()==0) { /* yyparse calls yylex */
    printf("\nParsing:: syntax OK\n"); /* reached if parsing folllows the grammar*/
 }
exit(EXIT_SUCCESS);
}
```

3. Dans parseur.y, on définit le parseur en utilisant des char comme TOKEN

```
/* file parseur.y */
/* compilation: bison -d parseur.y */
/* result: parseur.tab.c = C code for syntaxic analyser */
/* result: parseur.tab.h = def. of lexical units aka lexems */
%{
int yylex(void); /* -Wall : avoid implicit call */
int yyerror(const char*); /* same for bison */
%}
%token NOMBRE
%start resultat /* axiom */
%%
```

```
resultat: expression
;
expression:
expression '+' expression
| expression '-' expression

| expression '*' expression
| expression '/' expression
| '(' expression ')'
| '-' expression
| NOMBRE  /* default semantic value */
;
%%
#include <stdio.h> /* printf */
int yyerror(const char *msg){ printf("Parsing:: syntax error\n"); return 1;}
int yywrap(void){ return 1; } /* stop reading flux yyin */
```

Dans l'ordre :

- on définit le prototype du lexeur et de la fonction d'erreur de bison1
- on y décrit un unique token non trivial (càd qui n'est pas un simple char) : NOMBRE,
- on y décrit une grammaire avec expression, terme et facteur comme non terminaux et avec '+', '-', '*', '/', '(', ')', '-' ainsi que le token NOMBRE comme terminaux.
- on inclue des libs pour pouvoir écrire le programme renvoyé en cas d'erreur,
- on y décrit la fonction yyerror qui est appelée en cas d'erreur et la fonction yywarp afin d'arrêter le parseur lorsque yywrap retourne true (non-zero).

4. Dans lexeur.l, on définit l'unique token non trivial :

```
/* file lexeur.l */
/* compilation: flex lexeur.l */
/* result: lex.yy.c = lexical analyser in C */
%{
#include <stdio.h> /* printf */
#include "parseur.tab.h" /* token constants def. in parseur.y via #define */
%}
%%
[0-9]+ { printf("lex::NOMBRE %s\n",yytext); return NOMBRE; }
[ \t\n] { ; } /* ignore space, tab, and line return */
. { printf("lex::char %s\n",yytext); return yytext[0]; }
%%
```

[1] Il s'agit d'éviter certains warnings à la compilation :
https://stackoverflow.com/questions/20106574/simple-yacc-grammars-give-an-error

Licence Fondamentale Sciences de l'informatique	Classes : LGLSI-2
TP n°3 Compilation	
Objectifs	**Analyse lexicale et syntaxique avec Flex et Bison**

Dans l'ordre :

- On inclut parseur.tab.h qui est généré à partir de parseur.y et qui définit le token NOMBRE,
- selon l'expression régulière, on retourne le token NOMBRE,
- la ligne [\t]\n ; permet d'ignorer les séparateurs,
- la ligne . return yytext[0]; dit que si on lit autre chose, on renvoit au parseur un token trivial avec ce caractère

Compilez tout ça à l'aide des trois commandes suivantes dans le terminal :

```
$ bison -d parseur.y
$ flex lexeur.l
$ gcc -o main main.c parseur.tab.c lex.yy.c
```

Cela génère deux fichiers intermédiaires : votre parseur parseur.tab.c, votre lexeur lex.yy.c, puis votre exécutable main

5. Vous pouvez lancer main dans un terminal en lui passant un code source à analyser (data):
   $ ./main < data
   Si ce contenu est correct vous aurez un message l'indiquant sinon vous aurez un message d'erreur.
6. Modifier le contenu du fichier data et tester le résultat obtenu, exemple de fichier data :

```
2+5*8+7-22*78-3
```

```
2+a
```

7. Générer la version1 de votre analyseur qui permet d'écrire plusieurs expressions qui finissent par un " ;". Ajouter une expression régulière pour cela, et créez un nouveau token PT_VIRG sans oublier de modifier la grammaire du parseur.
8. Modifier, ensuite, le fichier flex afin de prendre en compte les nombres flottant non signés (exemple 12.478 12. .45) pour nos nombres.
9. Version finale : modifier le fichier Flex et le fichier bison afin de séparer l'analyse lexicale de l'analyse syntaxique : tous les terminaux du langage doivent être reconnus par flex et envoyés à bison. Rajouter, également, dans le parseur l'opérateur de division / .

Exercice 2 (passage de paramètres et calcul des opérations arithmétique).
Il n'est pas demandé dans un compilateur de faire le calcul des opérations arithmétiques, c'est le rôle de l'interpréteur. Cependant, dans ce TP, nous allons le faire pour se familiariser avec l'outil Bison.
Utiliser la première version du fichier Flex (celle de la question4 de l'exercice1).

Licence Fondamentale Sciences de l'informatique	Classes : LGLSI-2

TP n°3 Compilation

Objectifs	Analyse lexicale et syntaxique avec Flex et Bison

1. Commencer par modifier le lexeur (fichier flex) afin qu'il transmette la valeur des nombres reconnus. Pour se faire, on modifie la ligne qui permet de reconnaître NOMBRE afin de passer l'entier reconnu:

```
[0-9]+ { printf("lex::NOMBRE %s\n",yytext); yylval=atoi(yytext); return NOMBRE ; }
```

2. Ensuite il faut exprimer les contenus créés à chaque règle :

```
resultat: expression { printf("Resultat= %d\n", $1); }
;
expression:
expression '+' expression { $$ = $1+$3; }

| expression '-' expression { $$ = $1-$3; }
| expression '*' expression { $$ = $1*$3; }
| expression '/' expression { $$ = $1/$3; }
| '(' expression ')' { $$ = $2; }
| '-' expression { $$ = -$2; }
| NOMBRE { $$ = $1; } /* default semantic value */
;
```

Dans les actions (entre les acolades), $1,$2,$3 désignent le contenu du premier, second et troisième token utilisé dans la règle. Les actions, ici, sont de type entier, c'est l'entier que l'on va mettre dans le token créé.

3. Tester votre nouvel analyseur le fichier suivant :

```
2*5+2
```

Que pensez-vous du résultat obtenu ?

Exercice 3: (règles de priorité et d'associativité).
Pour cet exercice, utiliser la première version du fichier Flex (celle de la question4 de l'exercice1).

1. Allez dans parseur.y et modifiez ainsi le fichier :

```
%token NOMBRE
%left '+' '-'
%left '*'
%nonassoc MOINSU
%%
resultat: expression { printf("Resultat= %d\n", $1); }
;
expression:
expression '+' expression { $$ = $1+$3; }
```

Licence Fondamentale Sciences de l'informatique	Classes : LGLSI-2

TP n°3 Compilation

Objectifs	Analyse lexicale et syntaxique avec Flex et Bison

```
| expression '-' expression { $$ = $1-$3; }
| expression '*' expression { $$ = $1*$3; }
| expression '/' expression { $$ = $1/$3; }
| '(' expression ')' { $$ = $2; }
| '-' expression %prec MOINSU { $$ = -$2; }
| NOMBRE { $$ = $1; } /* default semantic value */
;
```

Cette version fait exactement la même chose que la précédente, mais en plus concis et intuitif grâce aux lois de priorité et d'associativité :

- Les règles d'associativité sont indiquées par %left ou %right.
- Les règles de priorité sont implicites : '*' est prioritaire sur '+' et '-' car %left '+' '-' est définit avant %left '*'.
- Lorsque l'associativité n'a aucun sens (par exemple pour un opérateur unaire) mais on veut indiquer sa priorité, on utilise %nonassoc et on place les opérateurs au bon niveau.

2. Lorsqu'un token est utilisé dans plusieurs règles avec des priorités/associativités différentes (comme '-'), on utilise une balise pour indiquer la priorité d'une des règles, ici la seconde règle du moins est balisée MOINSU qui a une autre priorité que '-' binaire.
3. Rajouter dans le parseur l'opérateur de division / (attention à sa priorité !)

Exercice4

1. Modifier votre analyseur pour prendre en compte les nombres flottants.
   Indications :
   Par défaut les TOKENs de bison ont le type int, mais, si l'on devait utiliser d'autres types associés au TOKENs il faudrait ajouter %union { double dval; int ival; } ; au parseur (fichier bison) et modifier le lexeur (fichier flex) en conséquence yylval.dval=...
   Pour indiquer au parseur que les terminaux et les non-teminaux vont créer des tokens contenant des types autres que int il faut aussi utiliser %token <dval> un_terminal et %type <dval> un_non_terminal si nécessaire.
   Vous pouvez utiliser la fonction C atof() qui permet de convertir une chaine de caractères en un nombre flottant.

2. Générer la dernière version de votre analyseur qui intègre les fonctionnalités demandées dans les questions 6,7 et 8 de l'exercice1.