I53- Compilation et théorie des langages
Introduction à `Flex` et `Bison`

Licence 3 - 2023/2024

1 Flex

Pour compiler un programme flex on utilise la commande `flex prog.lex` pour produire un fichier `C lex.yy.c` lui-même compilé avec l'option `-lfl`: `gcc lex.yy.c -lfl`. Utiliser un *makefile* pour compiler le programme et l'exécuter par la commande:
`./mot_le_plus_long < lex.yy.c`.

1. Modifier le programme pour que celui-ci retourne la ligne et la colonne où se trouve le mot le plus long.
2. Modifier le programme pour que celui-ci retourne la somme des entiers présents dans le fichiers.
3. Utiliser la variable interne gérant le flux d'entrée de `flex`

   ```
   FILE *yyin
   ```

   pour pouvoir passer le fichier à analyser en paramètre de la commande.

2 Bison

Le programme `bison` est un générateur automatique d'analyseur syntaxique. Pour construire un analyseur syntaxique avec la commande bison on édite un fichier de suffixe .y, disons fichier mini.y incluant la définition d'un analyseur lexical, obligatoirement identifié par yylex(), la description des lexèmes (tokens) et des règles. Les actions sémantiques écrites en langage C entre accolades calculent l'attribut du père, représenté par $$, en fonction des attributs des fils de gauche à droite représentés par $1, $2, ...

Un programme `bison` se décompose en 4 parties: prologue, définitions, règles et épilogue. Les zones du code correspondantes sont séparées dans le code source par les balises `%{, %}, %%, %%`.

```
%{
Prologue : declarations pour le compilateur C
%}
Definitions : definition des lexemes
%%
Grammaire : production et regles syntaxiques
```

```
%%
Epilogue : corps de la fonction main()
```

Le main du programme est réduit à sa plus simple expression : un appel de l'analyseur syntaxique yyparse() qui utilise implicitement la variable yylval et l'analyseur lexical yylex(), une fonction qui renvoie la valeur du lexème (token) courant dont l'attribut est transmis par la variable yylval. Les erreurs de syntaxes provoque l'appel de la fonction yyerror(). Ci-dessous, un exemple d'analyseur lexical rudimentaire.

```
int yylex( ) {
   int car ;
   car = getchar() ;
   if ( car == EOF ) return 0 ;
       if ( isdigit(car) ) {
           yylval = car - '0';
           return NB;
         }
   switch ( car ) {
     case '+' : return PLUS;
     case '\n': return FIN;
   }

 }
```

1. Exécuter les commandes:

   ```
   ./calculatrice
   ```

   et rentrer une expression arithmétique contenant uniquement des +.

2. Améliorer l'analyseur lexical pour filtrer les espaces et tabulations.

3. Modifier la grammaire pour gérer les autres opérations : multiplication, division, soustraction.

4. Intégrer une fonction `int myexp( int x, int n)` pour gérer les exponentiations, l'opérateur sera représenté par deux étoiles.

5. Gérez les parenthèses.

6. Modifier `yylex()` pour manipuler des nombres de plusieurs chiffres.

3 Utilisation conjointe de `Flex` et `Bison`

Il est possible d'écrire de puissant analyseurs syntaxiques avec `Bison` tout en sous-traitant la construction de l'analyseur lexical à `Flex`. Pour cela on commence par écrire l'analyseur syntaxique puis l'analyseur lexical chargé de reconnaitre les unités lexicales (tokens) définies par `Bison`.

Les fichiers `parser.y` et `lexer.lex` fournissent un exemple élémentaire d'utilisation conjointe de `Flex` et `Bison`. Le programme `Flex` récupère la définition des unités lexicales grâce au fichier `.h` qui sera produit par `Bison` lors de la compilation du fichier `parser.y` avec l'option `-d`. La compilation de la calculatrice se fait alors avec la séquence de commandes suivantes :

```
$ bison -o parser.c -d parser.y
$ flex -o lexer.c lexer.lex
$ gcc -Wall -o eval parser.c lexer.c -lfl
```

1. Ajouter la gestion des nombres à plusieurs chiffres.
2. Ajouter la gestion des opérateurs -,*,/,%.
3. Modifier la grammaire pour pouvoir saisir plusieurs calculs les uns après les autres.

On souhaite maintenant ajouter la gestion des identificateurs d'au plus 31 caractères. Pour cela on ajoute un `token ID` au parser dont le type sera `<id>`:

```
%union{
  int nb;
  char id[32];
 }
...
%token <id> ID
```

1. Modifier l'analyseur lexical pour que celui-ci gère le token `ID`;
2. Implanter (dans une fichier `.c` séparé) une table de symboles élémentaire permettant de rechercher ou d'ajouter des identificateurs. On pourra se limiter à fixer par avance la taille maximale de la table.

   ```
   struct identificateur {
     char id[32];
     int val;
    };

   typedef struct identificateur table_symbole[16];
   ```

3. Ajouter la gestion des identificateurs avec notamment la gestion de l'affectation.