TP2 - Analyse syntaxique suite Mini Projet

C. Mettre en œuvre un Analyseur syntaxique

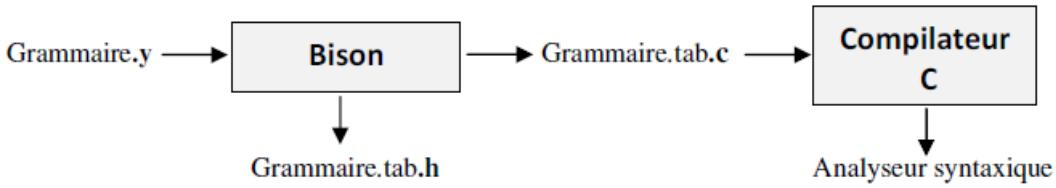
1. Mettre en œuvre un Analyseur syntaxique descendant. Pour ce faire, développez en langage C
   a. un algorithme pour éliminer la récursivité ;
   b. un algorithme pour Calculer les premiers de chaque non terminal ;
   c. un algorithme pour Calculer les suivants de chaque non terminal ;
   d. un algorithme pour Construire la Table LL1 ;
   e. un algorithme pour Exploiter la table LL1 ;

2. Mettre en œuvre un Analyseur syntaxique ascendant. Pour ce faire, développez en langage C
   a. un algorithme pour construire l'automate ;
   b. un algorithme pour Calculer les premiers de chaque non terminal ;
   c. un algorithme pour Calculer les suivants de chaque non terminal ;
   d. un algorithme pour Construire la Table SLR ;
   e. un algorithme pour Exploiter la table SLR ;

D.
Rappel (Bison)
Partie I

Le rôle de l'analyseur syntaxique est de vérifier la syntaxe du programme source. Il reçoit une suite d'unités lexicales fournie par l'analyseur lexical et doit vérifier que cette suite peut être engendrée par la grammaire du langage.

Bison (version GNU de YACC) est un générateur d'analyseurs syntaxiques. Il accepte en entrée la description d'un langage sous forme d'une grammaire et produit un programme écrit en C qui, une fois compilé, reconnaît les mots (programmes) appartenant au langage engendré par la grammaire en entrée.



Un fichier de spécifications Bison se compose de quatre parties :

```
%{
```

Les déclarations en c
%}
Déclaration des ***Unités lexicales*** *utilisées*
Déclaration de priorités et de types
%%
Règles de production avec éventuellement des actions sémantiques
%%
Bloc principal et fonctions auxiliaires en C

Règles de production :

Non-terminal **:** $prod_1$
| $prod_2$
| $prod_3$
….
| $prod_n$
;

Les symboles terminaux sont :

- Des unités lexicales (qu'on doit impérativement déclarer dans la partie 2)
  Syntaxe : % *token nom-unité-lexicale*
  Exemple : % *token* NB
  % *token* ID
- Des caractères entre quotes : '+', 'a'…
- Des chaînes de caractères entre guillemets : "while"

Les symboles non-terminaux représentent toute suite de lettres majuscules et/ou minuscules.

Remarque: La partie *Bloc principal et fonctions auxiliaires* doit contenir une fonction **yylex()** effectuant l'analyse lexicale du programme source. On peut soit écrire directement cette fonction soit utiliser la fonction produite par Flex.

Partie II – Installation de l'environnement et tests

1. Commencer par l'installation l'outil Bison.
2. Ouvrir un nouveau fichier texte et taper le code ci-dessus. Le fichier doit être enregistré avec l'extension **.y** (par exemple *grammaire.y*).
   2. Placer le fichier obtenu dans '*C:\Program Files\GnuWin32\bin* '
   3. A partir de l'invite de commande, lancer la commande :
      C:\Program Files\GnuWin32\bin> ***bison grammaire.y***
   4. En cas de réussite, le fichier *grammaire.tab.c* est généré dans le même répertoire.
   5. Compiler le fichier *grammaire.tab.c* pour générer l'exécutable.
   6. Quels sont les mots acceptés ?

```
#include <stdio.h>
int yylex(void);
int yyerror (char*);
%}
%%
mot : S '$' {printf("mot correct"); getchar();}
;
S : 'a'S'a'
| 'b'S'b'
| 'c'
;
%%
int yylex()
{
char c=getchar();
if (c=='a'||c=='b'||c=='c'||c=='$') return(c);
else printf("erreur lexicale");
}
int yyerror(char *s){
printf("%s \n",s);
return 0;
}
int main(){
yyparse();
printf("\n");
return 0;
}
```

Partie III

Ecrire un analyseur syntaxique pour chacun des langages suivants :

- $L = \{a^n b^n, n>0\}$
- $L = \{a^n b^n, n>=0\}$
- $L=\{a^n b^m c^n \setminus m, n>=1 \}$
- $L=\{a^m b^n c^p \setminus m+n=p\}$

Coordination entre Flex et Bison

On peut utiliser la fonction *yylex()* générée par Flex pour effectuer l'analyse lexicale.

Remarque: Un attribut est associé à chaque symbole de la grammaire (terminal ou non). L'attribut d'une unité lexicale est la valeur contenue dans la variable globale prédéfinie ***yylval***. Cette variable est l'outil de base de communication entre Flex et Bison au cours de l'analyse. Il faut donc penser à affecter correctement ***yylval*** lors de l'analyse lexicale. La définition de ***yylval*** est partagée dans le fichier ".h" issu de "bison -d".

Exemple:

```
%%
[0-9]+ {yylval=atoi(yytext); return NB ;}
%%
```

- Par défaut ***yylval*** est de type entier. On peut changer son type par la déclaration dans la partie 2 d'une union. Exemple :

```
% union {
int entier ;
double reel ;
char * chaine ;
}yylval ;
```

Dans ce cas, on peut stoker dans ***yylval*** des entiers, des réels ou bien des chaines de caractères. Il faut typer les unités lexicales et les symboles non-terminaux dont on utilise l'attribut, soit l'exemple:

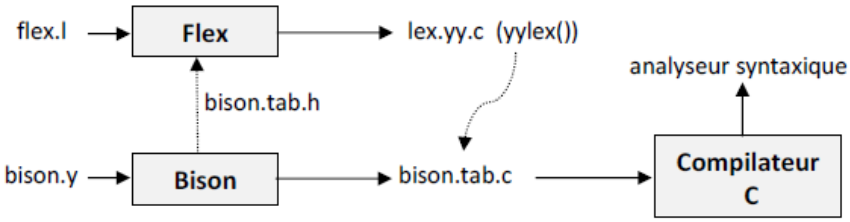
```
% token <entier> NB
% token <chaine> ID
% type <entier> S
% type <chaine> Expr
```

- Pour les symboles non-terminaux, $$ désigne la valeur de l'attribut associé au non-terminal de la partie gauche. $$_i$ désigne la valeur associée à l'ième symbole non-terminal de la partie droite, par exemple :

```
Expr : Expr '+' Expr {temp=$1 + $3 ;} '*' Expr {$$=temp + $6 ;} ;
```

- Les attributs peuvent être utilisés dans les actions sémantiques.

La génération d'analyseur syntaxique à l'aide de Flex et Bison est illustrée par la figure ci-dessous :



Partie IV

On veut écrire à l'aide de Flex et Bison un analyseur syntaxique qui permet de reconnaître les instructions de la forme suivante :

somme 1, 2, 3. produit 2,5. $

Il s'agit d'une liste d'entiers séparés par des virgules, se terminant par un point et précédées soit par le mot somme soit par le mot produit.

1. Ecrire tout d'abord le fichier de spécifications Bison suivant :

```
%{
#include<stdio.h>
int yylex(void);
int yyerror(char *s);
%}

%token FIN;
%token SOM;
%token PROD;
%token NB;
```

```
%%
liste:FIN {printf("correct");}
      |SOM listesom'.'liste
      |PROD listeprod'.'liste;
listesom: NB|listesom','NB;
listeprod: NB|listeprod','NB;
%%

#include "lex.yy.c"
int yyerror(char *s)
{
   printf ("%s", s);
   return (0);
}
int main()
{
  yyparse();
  getchar();
}
```

2. Compiler le fichier en utilisant l'option –d, par exemple bison -d nom.y
3. En cas de réussite, les fichiers *nom.tab.h* et *nom.tab.c* sont générés dans le même répertoire.
4. Inclure le fichier *nom.tab.h* dans le fichier de spécifications Flex (ce fichier contient la description d'unités lexicales utilisées).
5. Ouvrir un nouveau fichier texte et taper le code de spécifications Flex suivant :

```
%{
#include<stdio.h>
#include<math.h>
#include"nom.tab.h"
%}
%%
[0-9]+ {yylval=atoi(yytext); return NB;}
produit {return PROD;}
somme {return SOM;}
[,|.] {return yytext[0];}
[$] {return FIN;}
[ \t\n] {}
. {printf("Erreur");}
%%
int yywrap()
{return 1;}
```

6. Compiler ce fichier avec Flex pour générer le fichier lex.yy.c.
7. Inclure le fichier lex.yy.c dans le fichier de spécifications Bison.
8. Compiler maintenant le fichier nom.tab.c pour obtenir l'exécutable.
9. Tester le fichier nom.tab.exe obtenu pour vérifier qu'il fonctionne correctement.
10. Modifier l'exercice précédent pour que l'exécutable affiche la somme (respectivement le produit) des entiers formant chaque suite.

    Exemple : Si l'entrée est : *somme 2,5. Produit 3,6.$*
    Le résultat sera : *Somme = 7*
    Produit=18

11. Modifier cet exemple pour construire un analyseur syntaxique permettant de faire la soustraction et la division.

Annexe :

1. **Variables :**

   YYLVAL : variable prédéfinie qui contient la valeur de l'unité lexicale reconnue.

   YYACCEPT: instruction qui permet de stopper l'analyseur syntaxique. Dans ce cas, yyparse retourne la valeur 0 indiquant le succès.

   YYABORT: instruction qui permet également de stopper l'analyseur. yyparse retourne alors 1, ce qui peut être utilisé pour signaler l'échec de l'analyseur.

   % START non-terminal : c'est une action pour dire que le non-terminal est l'axiome.

2. **Fonctions :**

   int yyparse () : fonction principale qui lance l'analyseur syntaxique.

   int yyerror (char *s) : fonction appelée chaque fois que l'analyseur est confronté à une erreur.

Référence gnu : *http://www.gnu.org/software/bison/manual/*