Université des Sciences et de la Technologie Houari Boumediene
Faculté d'Electronique et d'Informatique
Département d'Informatique

Manuel BISON

1. Bison

Bison est un parseur qui permet de transformer une grammaire LALR(1) en code C. Il suffit de compiler le code généré (nommé par défaut NomFichier.tab.c) afin de générer un code exécutable qui effectue l'analyse syntaxique. Pour ce faire, Bison prend en entrée un fichier avec l'extension « .y » contenant la grammaire et d'autres instructions nécessaires pour la génération de l'analyseur syntaxique (voir Figure 1).

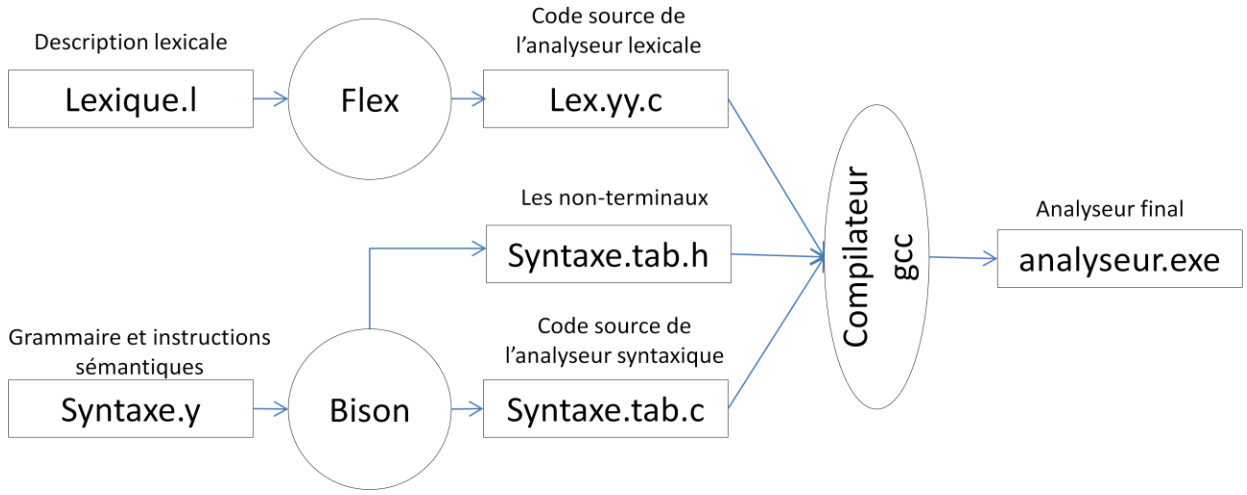


Figure 1 : fonctionnement Bison-Flex

2. Structure du fichier d'entrée de Bison « .y »

Le fichier « .y » d'entré est découpée en 4 zones séparées par les balises %{, %}, %%, %%. (Voir figure 2)

```
%{
 Déclaration C (pré-code)
%}
 Définitions et options
%%
 Règles de production
%%
 Code principal en C
```

Figure 2 : Structure du fichier d'entré de Bison

2.1. Déclaration C (pré-code)

Cette partie peut contenir les en-têtes, les macros et les autres déclarations C nécessaire pour le code de l'analyseur syntaxique.

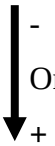
2.2. Définitions et options

Cette partie contient toutes les déclarations nécessaires pour la grammaire :

a. **Déclaration des symboles terminaux** : ceci est effectué en utilisant le mot clé **%token**.
   Exemple : **%token** MAIN IDF Accolade PointVirgule
   On peut préciser le type d'un terminal par **%token<type>.** *Exemple* :%token<int> entier

b. **Définition des priorités et d'associativité**: l'associativité est définie par les mots clé : **%left, %right** et **%nonassoc.** Quand à la priorité, elle est définie selon l'ordre de déclaration des unités lexicales.
   Exemple **:**

   %left A B /*associativité de gauche à droite*/
   %right C D /* associativité de droite à gauche*/
   %nonassoc E F /* pas d'associativité*/

c. **Autres déclarations :**
   - **%start** : permet de définir l'axiome de la grammaire. En l'absence de cette déclaration, Bison considère le premier non-terminal de la grammaire en tant que son axiome.
   - **%type** : définir un type à un symbole non-terminal.
   - **%union** : permet de spécifier tous les types possibles pour les valeurs sémantiques.

2.3. Règles de production

Ici, on décrit la grammaire LALR(1) du langage à compiler et les routines sémantiques à effectuées selon la syntaxe suivante :

<Non_terminal> : <séquence de symboles1> {routines sémantiques en langage C}
| <séquence de symboles2> {routines sémantiques en langage C}
| <séquence de symbolesN> {routines sémantiques en langage C}
;

- **Non_terminal** : c'est le symbole interne utilisé par l'analyseur syntaxique
- **<séquence de symboles1>** : les symboles sont représentés par des identificateurs terminaux (lexèmes provenant de l'analyse lexicale) et non-terminaux (symboles internes à l'analyseur syntaxique)

- **Routines sémantiques** : ce sont les instructions en C à exécuter à chaque réduction.

  Exemple :

  DEC **:** TYPE IDF PointVirgule {/*insérer IDF dans la table des symboles s'il n y existe pas*/}

  ;

2.4. Post-code C

C'est le code principal de l'analyseur syntaxique. Il contient le main ainsi que les définitions des fonctions.

3. Communication Flex-Bison

L'analyseur syntaxique et l'analyseur lexical communiquent par l'intermédiaire de la variable « yylval » qui stocke les valeurs sémantiques des entités lexicales.

La variable « yylval » est de type YYSTYPE (déclaré dans la bibliothèque Bison) qui est par défaut un « int ». Toutefois, nous pouvons changer ce type par un autre type ou même un ensemble de types de la manière suivante :

Changement de type	**Syntaxe**	**Exemple**
De « int » vers un autre type	#define YYSTYPE autre-type	#define YYSTYPE float
De « int » vers un ensemble de types	% union {type1 idf1 ; type2 idf2 ;}	%union {char *chaine ; int entier ;}

4. Premier pas avec flex-bison

Dans cette partie, nous allons implémenter un analyseur pour le langage traité précédemment dans la partie lexicale.

```
BEGIN
   INT NomVariable ;
   NomVariable  = EntierNat  ;
END
```

Ci-dessous les fichiers d'entré de Flex et Bison (« .l » et « .y ») nécessaires.

```
%{
#include<stdio.h>
#include "exp.tab.h" /* liaison avec bison* /
extern YYSTYPE yylval;
%}
IDF  [a-zA-Z]+
Entier 0|[1-9][0-9]*
%%
BEGIN { printf("Begin\n");
            return Begin ;}
END     { printf("End\n");
            return End ;}
INT { printf("Int\n");
        return Int ;}
";"   { printf(";\n");
        return ';' ;}
"="  { printf("=\n");
        return '=' ;}
"\n" {printf("saut de ligne\n");}
" " {printf("saut de ligne\n");}
{Entier}  {printf("entier\n");
              yylval.entier=atoi(yytext) ;
              return Entier ;}
{IDF} {printf("idf\n");
           yylval.chaine=strdup(yytext) ;
           return IDF ;}
.         {printf("erreur\n");}
%%
int yywrap()
{return 0 ;
}
```

```
%{
#include<stdio.h>
extern FILE* yyin ; /*fichier contenant le code à compiler*/
%}

%union
{char *chaine ;
 int entier ;
 }

%token BEGIN END INT  ';'  '='
%token <chaine> IDF
%token <entier> Entier

%%
S : BEGIN Dec Aff END {printf ("programme correct\n");}
;
Dec : INT IDF  ';'         {printf("Déclaration de %s\n",$2) ;
}
;
Aff : IDF '='  Entier ';'    {printf("Affectation de la valeur %d\n",$3) ;}
;
%%
int yyerror(char* msg)
{printf("%s",msg);
return 1;
}
int main()
{
yyin=fopen("code.txt","r");
yyparse();
return 0;
}
```

Etapes de compilation :

Générez le code C de l'analyseur lexical « lex.yy.c » avec **flex exp.l**
Générez le code C « exp.tab.c » et le header « exp.tab.h » de l'analyseur lexical avec **bison –d exp.y**
Compiler les sources **gcc lex.yy.c exp.tab.c -o exp**
Executer **./exp (pour linux) ou exp (pour windows)**