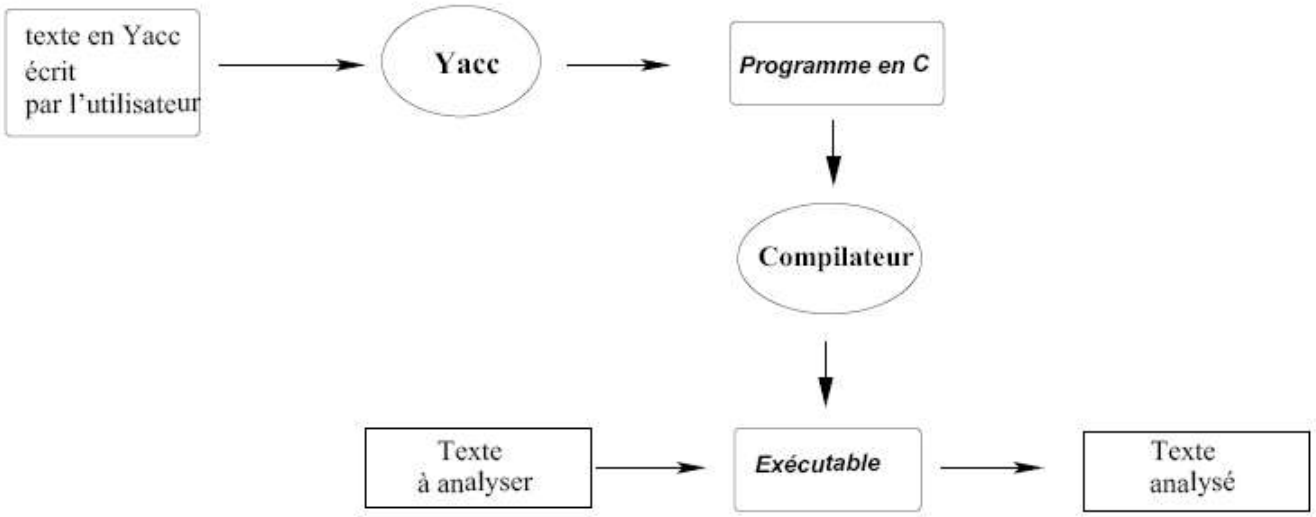
L'outil Yacc

Yacc signifie *Yet Another Compiler Compiler*, c'est à dire encore un compilateur de compilateur. Yacc est un outil logiciel qui permet de générer des analyseurs syntaxiques. Pour réaliser un tel analyseur il faut décrire d'abord l'outil à construire dans un fichier **monanalyseur.y**. Ce fichier permet d'obtenir un analyseur syntaxique en langage C qui a pour nom **y.tab.c**. Il convient d'ajouter un fichier lex qui va engendrer l'analyseur lexicale **lex.yy.c** qu'il faudra inclure dans le programme principal. Une fois obtenu cet analyseur il faut le compiler et l'on a alors un exécutable qui effectue l'analyse syntaxique d'un texte suivant les instructions données dans **monanalyseur.y**. Ce fonctionnement peut être schématisé par la figure suivante :



Les instructions nécessaires à la réalisation sont alors les suivantes
lex monanalyseur.lex
yacc monanalyseur.y
gcc -o monanalyseur y.tab.c

La structure d'une spécification Yacc

Un programme en Yacc se divise en trois parties séparées par des **%%**. La première contient des options, et des déclarations et initialisations de variables utiles pour le programme en C généré. La seconde contient la grammaire du langage et les actions à effectuer lorsque au cours de l'analyse on a réduit par une règle, enfin la troisième partie contient l'enveloppe du programme C généré.

%{
déclaration (en C) de variables, constantes, inclusions de fichiers,
%}
déclarations des unités lexicales utilisées
déclarations de priorités et de types
%%
règles de production et actions sémantiques
%%
routines C et bloc principal

Partie Déclaration

La partie déclarations contient les déclarations "C" (entre %{ et %}) ainsi que la déclaration des noms de tokens :
%token name1 name2 ...
Les noms qui n'ont pas été déclarés en tant que "nom de token" sont considérés comme des non-terminaux. Lorsque les tokens correspondent à des opérateurs pour lesquels on souhaite spécifier une propriété d'associativité, on utilise *left* et *right* à la place de token.

Exemple : on écrira :

```
%left '+' '-'   /* addition et soustraction associatives a gauche */
%right '^'      /* exponentiation associative a droite */
```

Grammaire et actions

Les règles de production sont des suites d'instructions de la forme

```
non-terminal :  prod1
              | prod2
                 ...
              | prodn
              ;
```

Les actions sémantiques sont des instructions en C insérées dans les règles de production. Elles sont exécutées chaque fois qu'il y a réduction par la production associée.
Exemple :

```
G : S B 'X' {printf("mot reconnu");}
;
S : A {printf("réduction par A");} T {printf("réduction par T");} 'a'
;
```

Le programme en C

La troisième partie doit contenir le programme principal main() qui doit en général faire un appel à la fonction **yyparse()** qui est créée par Yacc, et il doit aussi contenir un : **#include ``lex.yy.c''.**

Communication avec l'analyseur lexical : yylval

L'analyseur syntaxique et l'analyseur lexical peuvent communiquer entre eux par l'intermédiaire de la variable *int yylval.*
Dans une action lexicale (donc dans le fichier lex par exemple), l'instruction *return(ul)* permet de renvoyer à l'analyseur syntaxique l'unité lexicale *ul*. La valeur de cette unité lexicale peut être rangée dans *yylval*.
L'analyseur syntaxique prendra automatiquement le contenu de *yylval* comme valeur de l'attribut associé à cette unité lexicale.

La variable yylval est de type YYSTYPE (déclaré dans la bibliothèque yacc/bison) qui est un *int* par défaut. On peut changer ce type par:

```
#define YYSTYPE autre_type_C
```

Ou encore par :
%union {champs d'une union C}
qui déclarera automatiquement YYSTYPE comme étant une telle union.
Par exemple

```
%union {
        int entier;
```

```
        double reel;
        char * chaine;
      }
```

permet de stocker dans yylval à la fois des int, des double et des char *.
L'analyseur lexical pourra par exemple contenir

```
{nombre}  {
        yylval.entier=atoi(yytext);
        return NOMBRE;
        }
```

Le type des lexèmes doit alors être précisé en utilisant les noms des champs de l'union

```
%token <entier> NOMBRE
%token <chaine> IDENT CHAINE COMMENT
```

On peut également typer des non-terminaux (pour pouvoir associer une valeur de type autre que int à un non-terminal) par

```
%type <entier> S
%type <chaine> expr
```

Variables, fonctions et actions prédéfinies

YYPARSE() : appel de l'analyseur syntaxique.
YYACCEPT : instruction permettant de stopper l'analyseur syntaxique.
YYABORT : instruction qui permet également de stopper l'analyseur. yyparse retourne alors 1, ce qui peut être utilisé pour signifier l'echec de l'analyseur.
main() : le main par défaut se contente d'appeler yyparse(). L'utilisateur peut écrire sa propre main dans la partie du bloc principal.
%start non-terminal : action pour signifier quel non-terminal est l'axiome. Par défaut, c'est le premier décrit dans les règles de production.

Exemple :

Un exemple simple est le suivant, c'est un analyseur qui lit une suite de a et de b représentant un mot de parenthèse (a ouvrante, b fermante) et qui donne les règles qui l'ont engendré suivant la grammaire:
S →aSbS, S → aSb, S →abS, S→ab
Le point virgule sert de marqueur de fin

```
%token PVIRG PARO PARF
%%
s0 :exp PVIRG {printf("fini\n"); exit(0);}
    ;
exp:    PARO exp PARF exp {printf(" regle1 \n");}
|PARO exp PARF {printf(" regle2 \n");}
|PARO PARF exp {printf(" regle3 \n");}
|PARO PARF {printf(" regle4 \n");}
;
%%
#include <ctype.h>
#include <stdio.h>
#include "lex.yy.c"
main(){
yyparse();
}
```

```
yyerror(char *s){
printf ("%s\n",s);}
```

Etude de cas:

1. Le source Lex du mini-interprète d'expressions

```
%{#include "global.h"
#include "calc.h"
#include <stdlib.h>
%}
blancs    [ \t]+
chiffre   [0-9]
entier    {chiffre}+
exposant  [eE][+-]?{entier}
reel    {entier}("."{entier})?{exposant}?
%%
{blancs}  { /* On ignore */ }
{reel}    {
     yylval=atof(yytext);
     return(NOMBRE);
   }
"+"   return(PLUS);
"-"   return(MOINS);
"*"   return(FOIS);
"/"   return(DIVISE);
"^"   return(PUISSANCE);
"("   return(PARENTHESE_GAUCHE);
")"   return(PARENTHESE_DROITE);
"\n"  return(FIN);
```

2. Le source Yacc du mini-interprète d'expressions

C'est le plus important, et le plus intéressant. Le voici :

```
%{
#include "global.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
%}
%token  NOMBRE
%token  PLUS  MOINS FOIS  DIVISE  PUISSANCE
%token  PARENTHESE_GAUCHE PARENTHESE_DROITE
%token  FIN
%left PLUS  MOINS
%left FOIS  DIVISE
%left NEG
%right  PUISSANCE
%start Input
%%
Input:
    /* Vide */
  | Input Ligne
  ;
```

```
Ligne:
   FIN
  | Expression FIN    { printf("Resultat : %f\n",$1); }
  ;
Expression:
   NOMBRE      { $$=$1; }
  | Expression PLUS Expression {$$=$1+$3; }
  | Expression MOINS Expression {$$=$1-$3; }
  | Expression FOIS Expression {$$=$1*$3; }
  | Expression DIVISE Expression {$$=$1/$3; }
  | MOINS Expression %prec NEG {$$=-$2; }
  | Expression PUISSANCE Expression {$$=pow($1,$3); }
  | PARENTHESE_GAUCHE Expression PARENTHESE_DROITE {$$=$2;}
   ;
%%
int yyerror(char *s) {
  printf("%s\n",s);
}
int main(void) {
  yyparse();
}
```

Remarque

Veuillez noter que le fichier **global.h** contiendra :

```
#define YYSTYPE double
extern YYSTYPE yylval;
```