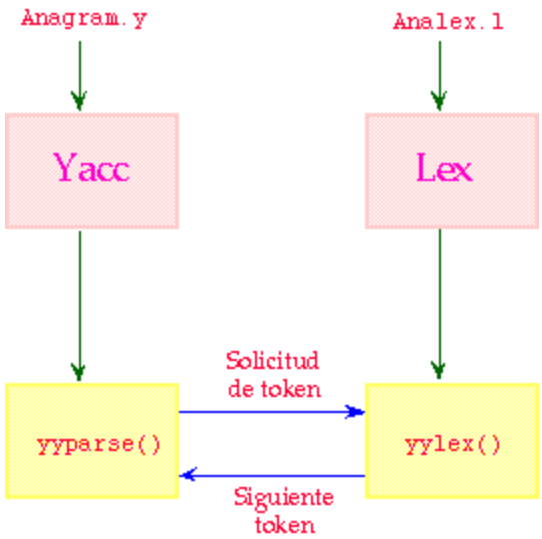
YACC

Yacc (Yet Another Compiler Compiler) es un programa que permite construir analizadores gramaticales en C a partir de una gramática libre al contexto. Junto con Lex permite construir rápidamente las primeras etapas de un traductor. Tanto Lex como Yacc son herramientas que fueron desarrolladas junto con el sistema UNIX y es posible encontrar implantaciones de ellas en otras plataformas como PC y Macintosh, al la vez que existen versiones de dominio público bajo los nombres de Flex y Bison.

Un analizador gramatical construido en Yacc genera una función que realizará el análisis gramatical con el nombre de yyparse(); esta función solicita un token al analizador de léxico por medio de la función yylex(). La comunicación que se da entre ambas funciones se muestra en el siguiente diagrama.



Un programa en Yacc tiene tres secciones. Estas secciones están delimitadas por %% como en Lex. La estructura del programa en Yacc es:

Definición del ambiente del analizador gramatical
%%
Gramática libre al contexto
%%
Procedimientos auxiliares

La sección de definición del ambiente del analizador gramatical posee una sección donde se hacen definiciones sobre las variables que el analizador empleará. Esta sección esta delimitada por %{ y %} ; en ella se hacen definiciones en C y se toma literalmente (Yacc no la revisa). En esta primera sección encontramos una serie de directivas de Yacc que permiten definir:

Los símbolos terminales que la gramática empleará.	%token
El axioma o símbolo inicial de la gramática.	%start

Los atributos que los símbolos de la gramática pueden poseer.	%union
	{
	$tipo_1$ $atributo_1$;
	$tipo_2$ $atributo_2$;
	…
	$tipo_n$ $atributo_n$;
	}

El atributo que cada símbolo posee.	%type <atributo>

La segunda sección está constituida por la gramática libre al contexto que guiará al analizador gramatical. La notación empleada es:

noterminal : $regla_1$

| $regla_2$
| $regla_3$
...
| $regla_n$
;

donde $regla_1$ a $regla_n$ representan las N producciones que tiene el noterminal en la gramática.

La tercer sección de un programa en Yacc tiene los procedimientos auxiliares que hacen operativo al analizador de léxico. En lo más simple, posee las funciones main y yyerror. Esta última sirve para especificar la reacción del analizador gramatical al encontrar un error. main simplemente invoca al analizador gramatical.

Un caso práctico

Supongamos que se quiere construir un analizador gramatical para revisar expresiones aritméticas constituidas por identificadores, números enteros y los operadores de suma, resta, multiplicación y división. El programa en Yacc para determinar si una expresión está bien o mal escrita es el siguiente:

```
%{
%}
%token ID NUM PA PC OPSR OPMD
%start NIVEL2
%%
NIVEL2 : NIVEL2 OPSR SIG_NIVEL

            | SIG_NIVEL
            ;

SIG_NIVEL : SIG_NIVEL OPMD OPERANDO

                | OPERANDO
                ;

OPERANDO : ID

        | NUM
        | PA NIVEL2 PC
        ;

%%

main()
{
yyparse();
}

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

Al procesar este archivo (llamémoslo anagram.y) en Yacc a través de la siguiente orden:

```
yacc -d anagram.y
```

se generará el programa anagram.c que contendrá la función yyparse() y el archivo yytab.h (a consecuencia de la opción -d) que contiene la designación de los valores a los tokens empleados en la gramática (especificados por la directiva %token). Yacc asigna los valores a los tokens a partir del 257; así que

```
%token ID NUM PA PC OPSR OPMD
```

genera en yytab.h las siguientes definiciones:

```
#define ID 257
#define NUM 258
#define PA 259
#define PC 260
#define OPSR 261
#define OPMD 262
```

La función yyerror(char * s) espera una secuencia de caracteres que representa el mensaje de error que el analizador gramatical encuentra; el mecanismo más simple de manejo es desplegar ese error (que es lo que hace en este caso la función).

El programa en Lex que realizará el análisis de léxico (analex.l) para devolver el token para cada unidad de léxico será:

```
%{
#include "yytab.h"
%}

L [A-Z]
D [0-9]
B [ \t\n]

%%
{L}+ return ID;
{D}+ return NUM;
[\+\-] return OPSR;
[\*\/] return OPMD;
[\(] return PA;
[\)] return PC;
{B}
.
%%
```

En el programa en Lex incluye al archivo yytab.h para que al reconocer a cada patrón, devuelva el token como está definido en el programa anagram.y.

Los dos programas generados, anagram.c y analex.c, se unen para formar un solo ejecutable que realizará el análisis gramatical. Como ya se mencionó, yyparse() ya tiene invocaciones a yylex().

Supongamos ahora que lo que quiere obtener no es solo saber si la expresión aritmética está bien o mal escrita, sino evaluarla si es correcta gramaticalmente. Se espera tener un valor numérico entero para cada operando en la expresión y un carácter para saber que operador aplicar (ya que se tienen en las categorías OPSR cuando se es suma o resta y OPMD cuando es multiplicación o división). En Yacc esto se especifica con %union de la siguiente manera:

```
%union {

    int valor;
    char operador;
    }
```

Ahora hay que designar que tipo de atributo se aplicará a cada símbolo en la gramática; un operando puede ser ID, NUM, OPERANDO, SIG_NIVEL y NIVEL2, por lo que éstos tendrán el atributo valor; mientras que los operadores OPSR y OPMD, tendrán el atributo operador. En Yacc está asociación se da con la directiva %type <atributo>

```
%type <valor> ID NUM NIVEL2 SIG_NIVEL OPERANDO
%type <operador> OPSR OPMD
```

Como los paréntesis no tienen ningún uso al evaluar una expresión, ni PA ni PC tienen asociado atributo alguno.

En Yacc puede asociarse a cada producción una acción codificada en C; si en esa acción se quiere referir a los atributos que un símbolo de la gramática posee, se hace referencia a él por medio de $$ si es el noterminal que está definiendo la producción y $1, $2 ... $n para los símbolos gramaticales que se encuentran en el lado derecho de la producción en el orden 1, 2 ... n. Por ejemplo, si queremos que al encontrar dos elementos que deben multiplicarse o dividirse al cumplirse la regla:

```
SIG_NIVEL : SIG_NIVEL OPMD OPERANDO
```

podemos referir la acción como

```
$$=opera($2,$1,$3);
```

suponiendo que la función opera espera en primer lugar un carácter que representa la operación a realizar y los dos siguientes parámetros sean los valores numéricos de los operandos; el resultado de la operación se asigna a $$ porque éste será el nuevo valor de SIG_NIVEL después de cumplirse la regla.

La acción por omisión que tiene definida Yacc es $$=$1; por lo que el programa completo para evaluar una expresión aritmética en Yacc es:

```
%{

%}

%token ID NUM PA PC OPSR OPMD
%union {

    int valor;
    char operador;
    }

%type <valor> ID NUM NIVEL2 SIG_NIVEL OPERANDO
%type <operador> OPSR OPMD
%start NIVEL2
%%
NIVEL2 : NIVEL2 OPSR SIG_NIVEL {

            $$=opera($2,$1,$3);
            printf("%d %c %d = %d\n",$1,$2,$3,$$);

                }
```

```
                | SIG_NIVEL
                ;

SIG_NIVEL : SIG_NIVEL OPMD OPERANDO {

                $$=opera($2,$1,$3);
                printf("%d %c %d = %d\n", $1, $2, $3, $$ );

                                }

                | OPERANDO
                ;

OPERANDO : ID

                | NUM
                | PA NIVEL2 PC {$$=$2;}
                ;

%%

main()
{
yyparse();
}

opera( char a, int b, int c )
{
switch (a)
{

     case '+': return (b+c);
     case '-': return (b-c);
     case '*': return (b*c);
     case '/': return (b/c);

}
}

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

Este programa supone que ID y NUM son capaces de obtener un valor al momento que se reconocen por el analizador de léxico. Esto obliga a modificar el programa en Lex para que esto ocurra. El analizador gramatical en Yacc crea un espacio para poder asociar un valor a cada símbolo por medio de la variable yylval, que es del tipo definido por %union. Así que al asociar un valor desde Lex a un terminal, es necesario hacer referencia a la entidad que representa al atributo. Por ejemplo, si NUM debe asociar un valor entero como atributo, la orden

```
yylval.valor=atoi(yytext);
```

hace esta asignación al convertir el lexema de NUM en un entero y asociarlo a yylval.valor. El programa completo en Lex quedará como sigue:

```
%{
#include "yytab.h"
%}

L [A-Z]
D [0-9]
B [ \t\n]

%%
{L}+ {yylval.valor=*(yytext); return ID;}
{D}+ {yylval.valor=atoi(yytext); return NUM;}
[\+\-] {yylval.operador=*(yytext); return OPSR;}
[\*\/] {yylval.operador=*(yytext); return OPMD;}
[\(] return PA;
[\)] return PC;
{B}
.
%%
```

Como no se tiene una tabla de símbolos para guarda un valor a cada identificador, se asume que su valor es el valor ascii del primer carácter de su lexema. Tanto para PA como para PC no hay asignación de yylval ya que éstos no tienen definido atributo por %type.