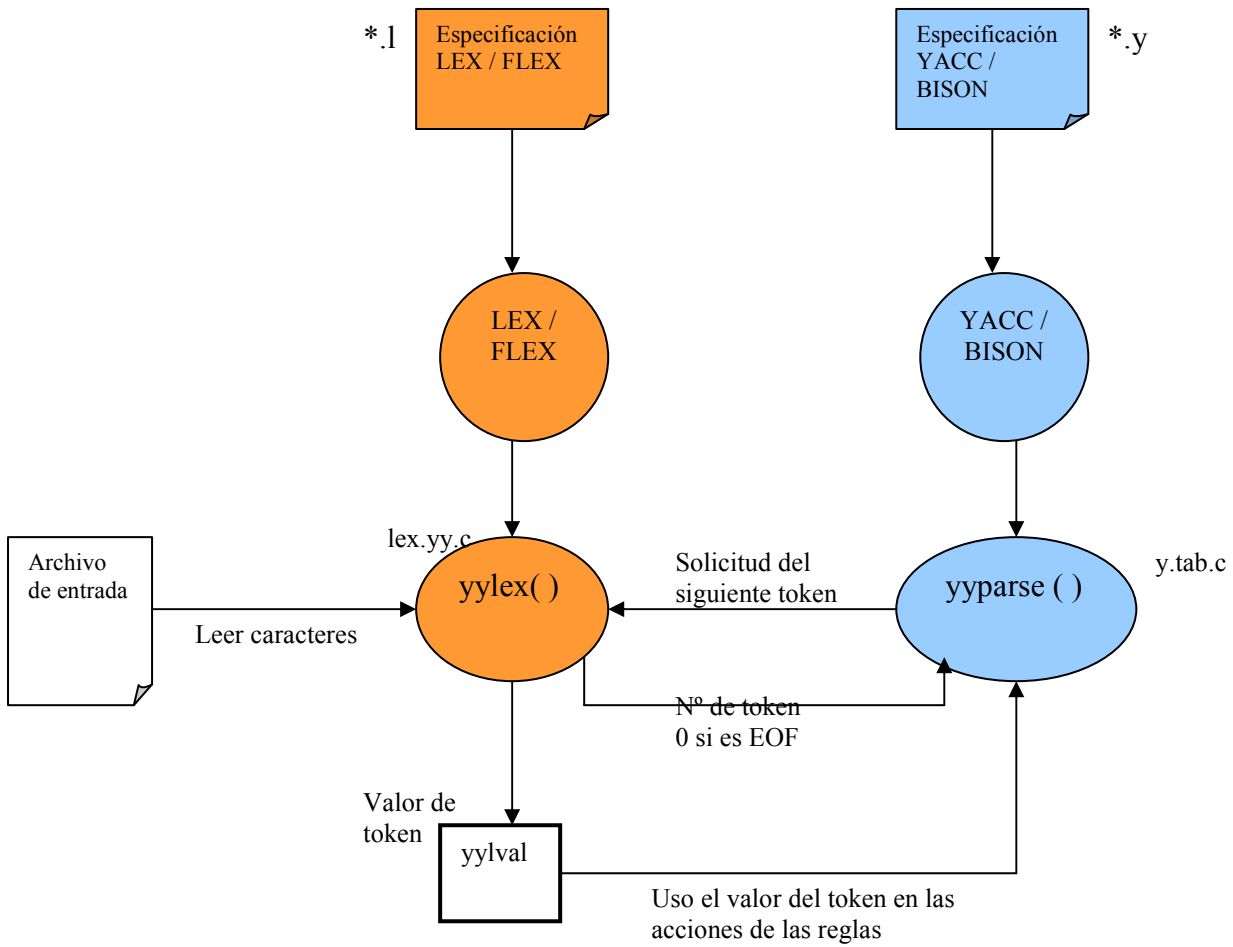
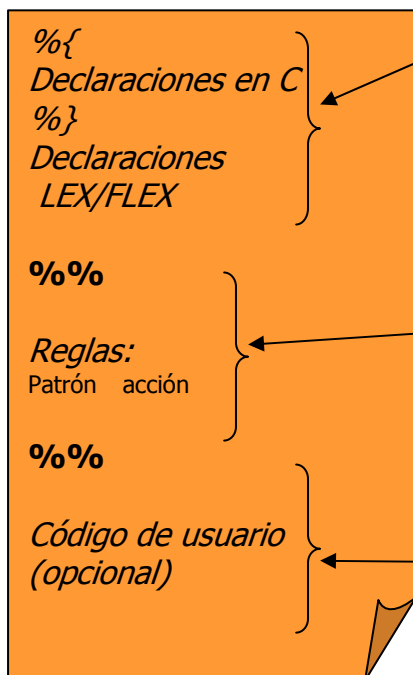


INTERACTIVIDAD ENTRE LEX/FLEX Y YACC/BISON



Estructura de LEX/FLEX

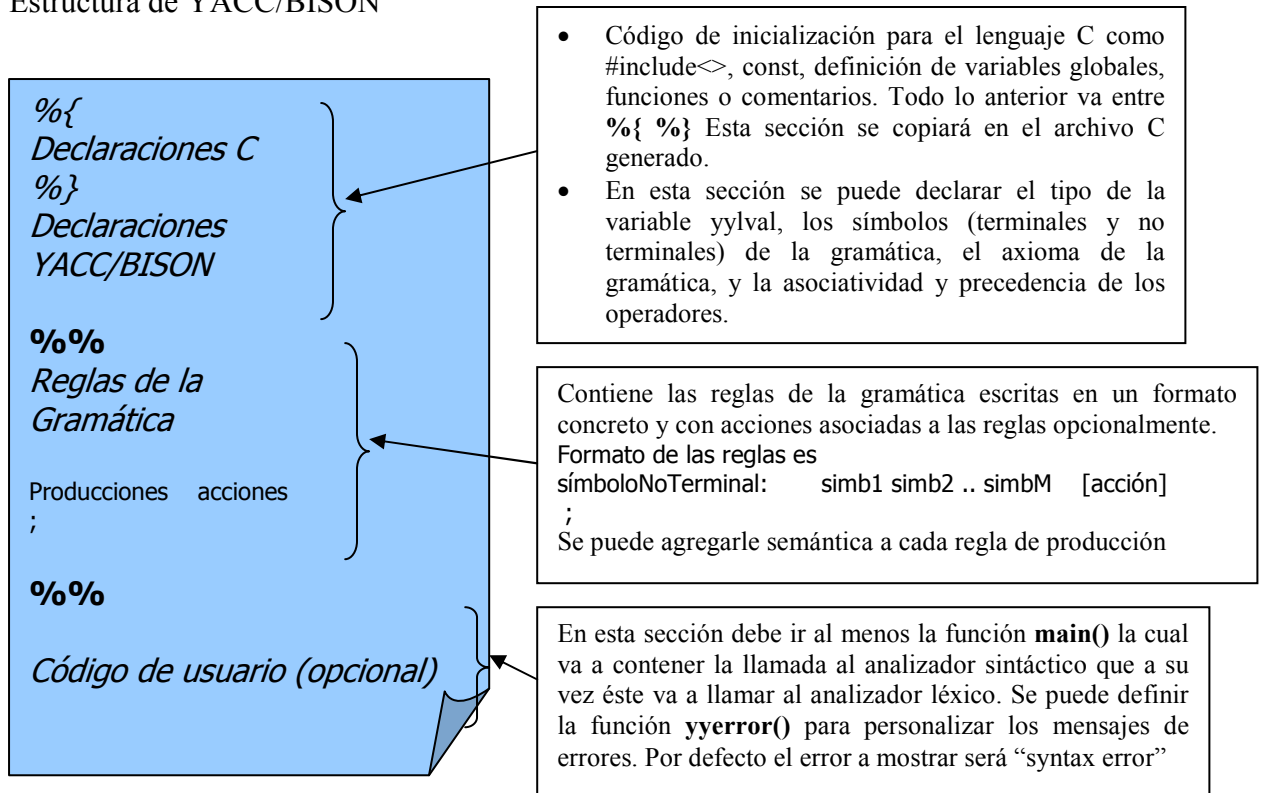


- Código de inicialización para el lenguaje C como `#include<>`, `const`, definición de variables globales, funciones o comentarios. Todo lo anterior va entre `%{ %}` Esta sección se copiará en el archivo C generado.
- Definiciones de especificaciones para LEX como macros, opciones del usuario, definición de patrones (se bautizan los patrones) y directivas.

Se divide en dos partes separados por un espacio, a la izquierda se ubican los patrones (exp. regulares) y a la derecha las acciones, código en lenguaje C, que debe realizar cuando la entrada concuerde con dicho patrón.

Código de usuario es opcional. Puede estar en un archivo separado con la condición de que en el archivo deba incluirse `lex.yy.c` en la cabecera del mismo para poder invocar a la función `yylex()`.
`#include "lex.yy.c"`

Estructura de YACC/BISON



Funciones y variables LEX/FLEX

yylex(): función que inicializa el análisis léxico.

yyomore(): Modifica la variable `yytext`. Esta concatena la última cadena que ha concordado con un patrón determinado con la cadena entrante que concuerda con patrón determinado. Por ejemplo:

```
...  
%%  
hacer-      yymore();  
deshacer    echo;  
%%  
...
```

En el ejemplo, se ingresa la cadena "hacer-" esta se guardará en la variable `yytext`, y la si la siguiente cadena es "deshacer" se concatenará a lo que tenía la variable `yytext`, es decir que va a producir una salida "hacer-deshacer"

yywrap(): esta función maneja múltiples ficheros de entrada. Devuelve 0 si encuentra otro fichero para ser analizado o 1 si encuentra un fin de ficheros. Si se necesita trabajar con multiples archivos, se debe definir `yywrap`. Si no se necesita trabajar con multiples archivos de entrada se debe agregar la directiva **%option noyywrap** en la sección de *definiciones* del archivo `lex` o compilar el archivo `lex.yy.c` con la opción `-ll` (`-lfl` en Flex).

yytext: es una variable que apunta a la cadena de entrada a analizar. Por cada nueva cadena de entrada la variable `yytext` se reinicializará, es decir, borrará la cadena que

almacenaba anteriormente. El uso de `yymore()` evita que se pierda lo que `ytext` tenía anteriormente.

yylen: variable que almacena la dimensión de la cadena guardada en `ytext`.

yylval: variable nexa entre Lex y YACC. En esta variable `lex` colocará el token (o componente léxico) de un lexema que concuerda a una determinada expresión regular para que YACC pueda continuar realizando el análisis sintáctico.

yyin: Permite manipular los procesos de entrada, por ejemplo se almacena en `yyin` el archivo de entrada el cual `lex` manejará automáticamente.

yyout: Permite manipular los procesos de salida. Es una variable de tipo FILE donde se define un archivo de salida el cual se copiarán todas las salidas.

Funciones, variables, especificaciones y declaraciones de Bison

En la sección de declaraciones de bison se puede declarar el tipo de la variable **yylval**, los símbolos (terminales y no terminales) de la gramática, el axioma de la gramática, y la asociatividad y precedencia de los operadores.

Directivas:

%union

Por defecto, la variable **yylval** (variable nexa) mediante la cual Lex/Flex le pasa a Yacc/Bison los valores semánticos de los tokens es de tipo "int". Pero habitualmente, los valores semánticos de los tokens son de distintos tipos. Por ejemplo, un token de tipo identificador (ID) tiene un valor semántico de tipo "char*", mientras que un token de tipo constante numérica (NUM) tiene un valor semántico de tipo "int". Con la declaración `%union` se define indirectamente una unión de C con un campo para cada tipo de valor semántico, por ejemplo:

```
%union
{
char* cadena;
int numero;
}
```

%token

Se utiliza para definir los símbolos no terminales (tokens) de la gramática. La forma más sencilla es:

```
%token NOMBRE_TOKEN
```

Una forma más completa es:

```
%token <campo_union>
NOMBRE_TOKEN
```

Ejemplo:

```
%token IF
%token THEN
...
%token <cadena> ID
%token <numero> NUM
```

Desde Lex/Flex, los tokens cualificados se devolverán haciendo, por ejemplo:

```
[A-Z]+ { yylval.identificador = yytext; return ID; }
[0-9]+ { yylval.numero = atoi(yytext); return NUM; }
```

Por convenio, los nombres de los tokens se ponen en mayúsculas. Se pueden agrupar varios token en una línea si tienen en mismo tipo. Los tokens de un único carácter no es necesario declararlos, porque están declarados implícitamente.

Desde Lex/Flex, este tipo de tokens se devolverán haciendo, por ejemplo:

```
"+" {return '+'; }
"(" {return '('; }
";" {return ';' ; }
```

El número 0 está reservado para el fin de fichero. Lex/Flex hará lo siguiente:

```
<<EOF>> {return 0; }
```

Cuando se compila el fichero de especificación .y se utiliza la opción -d, que fuerza a que Yacc/Bison genere el fichero y.tab.h/*tab.h que contiene las definiciones de los tokens. Si este fichero se incluye en la especificación de Lex/Flex, se consigue que Yacc/Bison y Lex/Flex compartan las definiciones de tokens.

%type

Se utiliza cuando se ha hecho la declaración %union para especificar múltiples tipos de valores. Permite declarar el tipo de los símbolos no terminales. Lo no terminales a los que no se les asigna ningún valor a través de \$\$ no es necesario declararlos (ver sección de reglas de la gramática)

La declaración es de la forma

```
%type <campo_union> nombre_no_terminal
```

Por convenio, los nombres de los no terminales se ponen en minúsculas. Se pueden agrupar varios no terminales en una línea si tienen en mismo tipo.

%start

Declara cual es el axioma de la gramática. Si se omite la declaración, se asume que el axioma de la gramática es el primer no terminal de la sección de reglas de la gramática.

La declaración es de la forma

```
%start axioma
```

%left y %right

Permiten declarar la asociatividad de los operadores de la gramática. La declaración %left especifica asociatividad por la izquierda. La declaración %right especifica asociatividad por la derecha.

La precedencia de los operadores queda establecida por el orden de aparición de las declaraciones de asociatividad, siendo de menor precedencia en operador que antes se declara.

Por ejemplo, las declaraciones:

```
%left '+' '-'
```

```
%left '*' '/'
```

Establecen que los cuatro operadores son asociativos por la izquierda, que '+' y '-' son de la misma precedencia, pero de menor que '*' y '/'.

yyerror:

Esta función invocada desde yyparse cuando ocurre un error sintáctico. Se debe especificar la función yyerror() en el código de usuario.

Por ejemplo, la forma mas sencilla de definirlo es:

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

El mensaje genérico que se mostrará es "syntax error"

Si se desea personalizar esta función y tener una buena política de errores, se deberán capturar los errores y ser enviados a la rutina. El analizador léxico puede ayudar a realizar este trabajo.

Por ejemplo en la construcción de un compilador para detectar caracteres ilegales en el código fuente, el analizador léxico podría aceptar estos caracteres e identificarlos como erróneos. En flex, hacer que estos caracteres ilegales concuerden con un patrón para luego mandar el mensaje de error es una buena forma de ser más precisos en los errores. Es decir, en la gramática se tendría que considerar y armar producciones de error.

En lex/flex:

```
[?"!")"(")+      {yyval.cilegal = yytext;
                    printf(stderr, "Error, carácter ilegal");
                    return CI;}
```