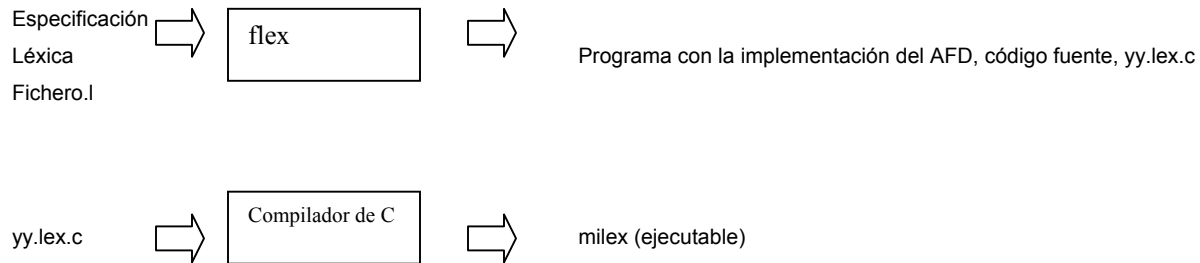


FLEX: A FAST LEXICAL ANALYZER GENERATOR



Para compilar

```
flex fichero.l
```

```
cc yylex.c -o milex -lfl
```

lfl es una librería propia de Flex.

Esquema de un fichero de especificación léxica

Consta de 3 secciones que vienen separadas por los símbolos %%

Declaraciones y Definiciones

%%

Patrones y Acciones

%%

Código Auxiliar

Sección de Declaraciones

Se incluyen las declaraciones de: variables, constantes, includes ... (en general variables y código C que necesitemos para ejecutar las acciones). Se coloca entre los símbolos

```
%{ ... }.
```

En esta sección también se definen abreviaciones de expresiones regulares que se utilizarán más tarde para la definición de los patrones. Por ejemplo:

```
%{  
#include <stdio.h>  
  
...  
int nlines=0;  
%}
```

```
DIGITO [0-9]
```

```
LETRA [a-zA-Z]
```

Sección de Patrones y Acciones

```
Patrón 1 {Acción 1}
```

```
Patrón 2 {Acción 2}
```

```
...
```

```
Patrón n {Acción n}
```

Los patrones son las expresiones regulares de los diferentes componentes léxicos a reconocer. Las acciones son el código a ejecutar cuando se reconoce un determinado componente léxico. Si las acciones a ejecutar ocupa más de una línea, debe obligatoriamente ponerse entre llaves, si no, las llaves no son necesarias.

Sección de Código Auxiliar

Código C de las funciones que se utilizan en las acciones.

Se puede colocar también la función `main`.

Operadores que se pueden utilizar al definir las expresiones regulares:

[]	rango de valores
*	Cero o más repeticiones
+	Al menos una vez
?	Opcionalidad
	Alternativa
.	Cualquier otra cosa diferente a las expresiones regulares anteriores
^	Negación

Para utilizar estos caracteres como tales y no como metasímbolos de definición expresiones regulares, colocar el carácter `\` delante de ellos o ponerlos entre `“ ”`.

Colocar `{ }` para hacer referencia a las abreviaturas que se definen en la sección de declaraciones.

```
{DIGITO}+"."{DIGITO}+ {printf("Encontrado NUMERO REAL");}
```

Variables internas de Flex

<code>char * yytext</code>	<i>Lexema del patrón reconocido</i>
<code>int yyleng</code>	<i>Longitud del lexema reconocido</i>
<code>FILE * yyin</code>	<i>Fichero de entrada (por defecto stdin)</i>
<code>FILE * yyout</code>	<i>Fichero de salida (por defecto stdout)</i>
<code>int yylex()</code>	<i>Llamada al analizador léxico</i>
<code>int yyparse()</code>	<i>Llamada al analizador sintáctico</i>

Ambigüedades de las reglas

Si una entrada encaja con más de un patrón se sigue un criterio:

Se elige la opción que encaje con un mayor número de caracteres. A igual número de caracteres se elige la regla que se encuentra en primer lugar (de aquí que pongáis la definición de las palabras clave antes que los identificadores).

Por ejemplo para la entrada `ende`, la reconoceríamos como un identificador

```
"end" {return TKN_END;}  
{LETRA}+ {return TKN_ID;}
```

Interacción con Bison

El analizador sintáctico pide al analizador léxico que le proporcione un nuevo *token* cada vez que durante el proceso de análisis sintáctico encuentra un símbolo terminal en la gramática. El analizador léxico lee de la entrada e intenta reconocer algún patrón, entonces ejecuta la acción y devuelve al analizador sintáctico el *token* reconocido. Dentro de las acciones debemos poner `return TKN;` y devuelve al analizador sintáctico el control del programa.

BISON: GENERADOR DE ANALIZADORES SINTÁCTICOS

La entrada es un fichero con la extensión `.y` que contiene la definición de los terminales, no-terminales y la gramática. La salida es un programa en C con la implementación de un analizador sintáctico LR. Al igual que Flex consta de tres secciones:

Declaraciones y Definiciones

```
%%
```

Reglas de la gramática

```
%%
```

Código Auxiliar

Sección de Declaraciones

Esta sección incluye:

- La inclusión de ficheros `include` y la definición de variables globales. Habrá que incluir necesariamente la función `yylex` declarándola como externa y el prototipo de la función `yycerror`. Es decir, incluir las siguientes líneas:

```
extern int yylex(void);
```

```
void yyerror(char *);
```

- La declaración de los símbolos de la gramática. Los terminales con `%token` y los no-terminales con `%type`.
- Se indica el símbolo de inicio de la gramática con `%start`
- Se indican las precedencias y asociatividad de los operadores (`%left`, `%right`, `%nonassoc`)
- La definición de la estructura union. `%union`

Los símbolos de la gramática pueden almacenar valores, tanto los terminales y los no-terminales. Con `%union` se definen los diferentes tipos de datos que pueden almacenar los símbolos de la gramática.

Al definir los símbolos pondremos entre `<tipo_de_dato>`, el tipo de dato que queremos almacenar y que debe corresponder con uno de los campos de la estructura `%union`. Si no se especifica nada, por defecto el tipo de datos que se pueden almacenar en las variables `$i` son números enteros. Por ejemplo:

```
%union {  
float real;  
char * nombre;  
....
```

```
}
```

```
%token <real> TKN_NUM
```

También podríamos definirnos una estructura con diferentes campos, en el caso de que quisiésemos almacenar más de un dato por *token* y luego definir en `%union` un campo que sea de tipo dicha estructura.

Para evitar situaciones de conflicto desplaza/reduce, se debe asignar una precedencia y asociatividad a los distintos operadores y para ello usaremos los operadores `%left`, `%right`. Tendrán menos precedencia los declarados primero e igual precedencia si están colocados en la misma línea. Por ejemplo:

```
%left TKN_MAS TKN_MENOS
%left TKN_MULT TKN_DIV
```

Los operadores `TKN_MULT` y `TKN_DIV` tienen mayor prioridad que el `TKN_MAS` y `TKN_MENOS` ya que están en una línea posterior. Los operadores `TKN_MAS` y `TKN_MENOS` tienen igual prioridad. Todos son asociativos a izquierdas.

Nota: Si queremos cambiar la precedencia y asociatividad de un operador para el caso de una producción particular de la gramática, esto se puede hacer utilizando el operador `%prec nombre_token` al final de la regla.

Por ejemplo, supongamos que tenemos el `TKN_MENOS` pero que, dependiendo de la producción de la gramática en la que estemos puede tratarse del operador resta o del menos unario (cambio de signo). En la parte léxica, el *token* reconocido será el mismo `TKN_MENOS`, pero a nivel sintáctico tienen significado diferente: uno tiene asociatividad por la izquierda (la resta) y el otro por la derecha (menos unario). Además el menos unario tiene mayor prioridad que el menos binario. Esto nos lleva a cambiar dicha precedencia en la producción en cuestión. Se deben definir un nuevo tipo de token, `TKN_UNARIO`.

Ejemplo:

```
...
%left TKN_RESTA
%right TKN_UNARIO
...
%%
expresion : TKN_NUM TKN_RESTA TKN_NUM      |
           TKN_RESTA DIGITO %prec TKN_UNARIO ;
%%
```

Sección de Reglas con el formato

no-terminal : secuencia de símbolos en la parte derecha ;

Notad el símbolo `;` para indicar final de producción. Por ejemplo:

```
E : E TKN_MAS E ;
```

Sección de Código Auxiliar

Código C de las funciones que se utilizan en las acciones. Se puede colocar también la función `main`. Nota: Si no hay código auxiliar, el `%%` anterior no se escribe necesariamente.

Cómo almacenar y acceder a valores almacenados en los símbolos de la gramática

La variable interna de Bison, `yyval`, es del tipo *union*, que hayamos definido en Bison, y sirve para comunicarse entre FLEX y BISON. Los símbolos de la gramática pueden almacenar valores tanto los terminales y los no-terminales. Para ello, en la parte de Flex daremos valor a la variable `yyval` y éste se almacenará directamente en el símbolo de la gramática que se está reconociendo en ese momento.

Para acceder a los valores almacenados en los símbolos de la gramática, se utiliza la notación `$`.

`$$` para acceder al símbolo de la parte izquierda (el padre).

`$i` para acceder al símbolo en la posición `i` de la parte derecha (los hijos).

Nota: Tener en cuenta que en Bison, si se ha introducido código entre llaves en medio de las producciones, esto es contabilizado como un número de hijo más aunque no proceda el que devuelva un valor con `$i`).

Cómo añadir acciones semánticas

En la posición de la regla que nos interese, se coloca entre llaves el código C a ejecutar en ese momento. Sirve para implementar la construcción de la tabla de símbolos, comprobaciones de tipos, declaración de identificadores antes de uso, etc.

Cuidado!: las acciones cuentan como uno más al contabilizar las posiciones cuando queremos acceder con `$` a los valores almacenados en las variables. Por ejemplo, si queremos calcular el valor de la expresión como la suma de lo almacenado en los dos operandos.

```
E : E {printf("%d", $1);} TKN_MAS E {$$ = $1+$4;} | TKN_NUM {$$ = $1;} ;
```

Para compilar

```
bison -d sintactico.y
```

```
flex lexico.l
```

```
cc lex.yy.c sintactico.tab.c -o mianalizador -lfl
```

Al compilar el Bison con la opción `-d` se crea un fichero `sintactico.tab.h` con la definición de los *tokens*, que hay que incluir en la parte léxica y un fichero `sintactico.tab.c` con la implementación del analizador. Flex genera un fichero `.c` llamado siempre `lex.yy.c` que será el que se compile con el compilador de C junto con el `sintactico.tab.c`.

Podéis compilar con la opción `bison -d --verbose sintactico.y` y os creará un fichero con la extensión `.output` donde podréis ver si la implementación del autómeta desplaza/reduce, estado por estado y las transiciones. Además, se indicarán los estados donde se encuentran los conflictos (si existen) y si hay algún símbolo de la gramática que no es utilizado o no es accesible.

Aspectos a tener en cuenta al almacenar y al acceder a símbolos de la gramática

Los lexemas de los *tokens* reconocidos siempre se almacenan en la misma variable `yytext`, por lo que dicha variable sólo contiene el lexema del último *token* reconocido. En el caso de que nos interese almacenar la cadena de caracteres para su uso posterior, deberemos guardarla inmediatamente después de que haya sido reconocido el *token*. Existen dos maneras de recuperar dicho lexema:

1.- En cada producción capturar el valor de `yytext` justo después de haber reconocido el *token* mediante la inclusión de código en las producciones.

2.- Hacer que sea el analizador léxico el que, cada vez que reconozca el tipo de *token*, almacene en una variable el valor del lexema y que se almacenará en la variable `$` de forma interna entre Flex/Bison.

¿Cómo implementar la primera forma?:

- Declarar la variable `yytext` como externa en el fichero `.y`

```
%{
...
extern char * yytext;
%}

...
%union{
int entero;
}
...
%token <entero> TKN_NUM
...
%%
inicio : TKN_NUM {$1 = atoi(yytext);}
        TKN_RESTA
        TKN_NUM {$4 = atoi(yytext); printf("%d , %d", $1, $4); } ;
%%
...
```

¿Cómo implementar la segunda forma?:

- En el fichero léxico se usa la variable `yylval`, pudiendo después utilizar directamente las variables `$i` (que son las que contienen la información) en el sintáctico.
- Tener en cuenta que esa variable `yylval` será, en cada momento, del tipo que hayamos definido para el *token* correspondiente.

Ejemplo de fichero léxico:

```
DIGITO [0-9]
...
%%
{DIGITO}+ { yyval = atoi(yytex);
            return TKN_NUM; }
...
%%
```

Ejemplo de fichero sintáctico:

```
...
%union{
int entero;
}
...
%token <entero> TKN_NUM
...
%%
inicio : TKN_NUM TKN_RESTA TKN_NUM
      {
        $$ = $1-$3;
        printf("%d - %d = %d", $1, $3, $$);
      };
...
%%
...
```