GUÍA BÁSICA DE FLEX Y BISON

PRÁCTICAS - PROCESAMIENTO DE LENGUAJES

GRADO EN INGENIERÍA INFORMÁTICA

Curso 2016/2017

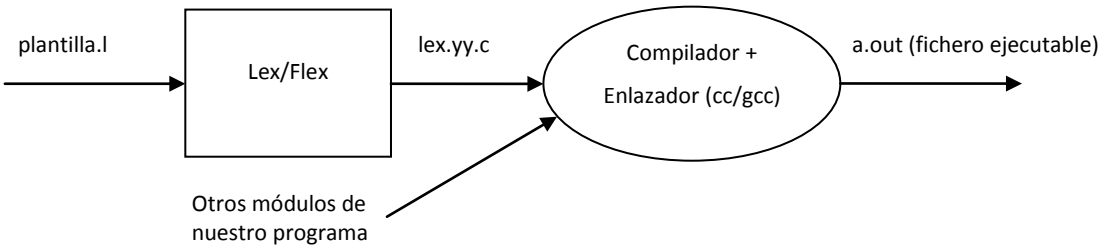
LEX/FLEX: GENERADOR DE ANALIZADORES LÉXICOS

1. Introducción

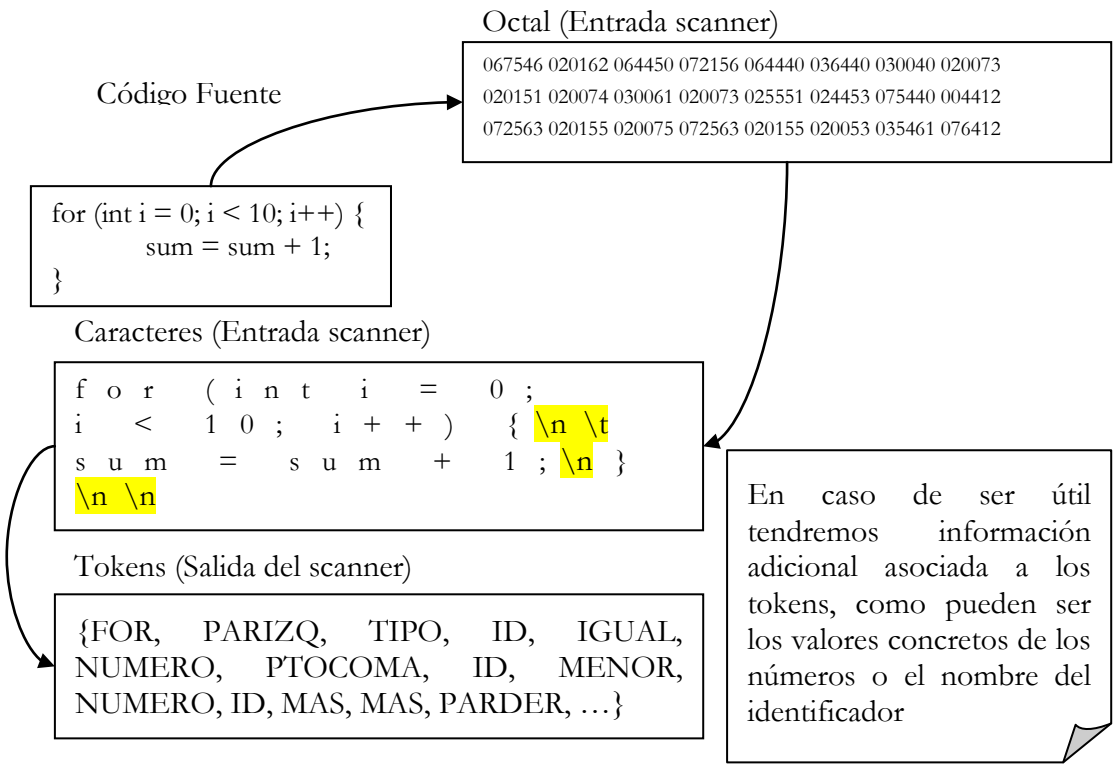
Lex es una de las herramientas que permiten generar analizadores léxicos (*scanners* o *lexers*) a partir de una especificación basada en expresiones regulares. Un escáner o analizador léxico es la primera fase de un compilador, cuya principal tarea es leer los caracteres de la entrada del programa fuente, agruparlos y producir como salida una secuencia de *tokens* (carácter o secuencia de caracteres del archivo fuente que se han reconocido y etiquetado: operadores, identificadores, números, palabras reservadas, etc.)

Lex es el analizar léxico estándar en los sistemas UNIX/Linux (se incluye en el estándar POSIX a partir de la 7ª edición) y fue desarrollado originalmente en los años 70 por Eric Schmidt y Mike Lesk en los laboratorios Bell de AT&T. Aunque tradicionalmente se trata de software propietario, existen versiones libres basadas en el código original, de las cuáles tal vez la más conocida sea Flex que es un desarrollo realizado por GNU bajo licencia GPL y cuenta con algunas características extra como un mejor soporte para reducciones o expresiones muy largas o complejas.

La principal característica de Lex/Flex es que es una herramienta que se basa en la búsqueda de concordancias en un fichero de entrada con un conjunto de expresiones regulares definidas, y en la ejecución de acciones descritas en código C asociadas a dichas expresiones. Internamente actúa como un autómata que localiza las expresiones regulares descritas y una vez reconocida la cadena representada por dicha expresión regular, ejecuta el código asociado a la regla correspondiente. Externamente puede ser considerado como una caja negra con la siguiente estructura:



Por ejemplo, para un bucle escrito en código C:



En las siguientes secciones se recoge el proceso de diseño de la plantilla de expresiones regulares y la compilación de los programas generados mediante estas herramientas.

2. Expresiones regulares

Una expresión regular, a menudo denominada también patrón, es una expresión que permite describir un conjunto de cadenas de símbolos del alfabeto sin que sea necesario enumerar sus elementos por separado. Se construye utilizando caracteres del alfabeto sobre el que se define el lenguaje y los siguientes operadores:

- Unión: se representa por una barra vertical que separa las alternativas.

  a | b, indica que en la expresión regular puede ir el símbolo del alfabeto 'a' o el símbolo 'b'.

- Concatenación: es la operación por la cual dos símbolos del alfabeto se unen para formar una cadena.

  'a' concatenado con 'b' nos daría la cadena 'ab'.

- Operador * (Cerradura de Kleene): indica que una determinada cadena puede aparecer cero o más veces.

  {a}* = {ε, a, aa, aaa, aaaa, aaaaa, aaaaaa,...}

- Operador +: sirve para indicar que una determinada cadena puede aparecer una o más veces. A diferencia del operador cerradura (*), la cadena ha de aparecer al menos una vez.

  {a}+ = {a, aa, aaa, aaaa, aaaaa, aaaaaa,...}

- Símbolo de interrogación (?): indica que la cadena puede aparecer o no, pero si aparece sólo puede hacerlo una vez.

  {ab?} = {a, ab}

- Corchetes ([]): su función es agrupar rangos de caracteres. Son útiles cuando se desea representar un conjunto de caracteres.

  [a-z] representa el conjunto de todas las letras minúsculas

  [A-Z] el conjunto de todas las mayúsculas

  [a-zA-Z] todas las letras del alfabeto

  [0-9] los números del cero al nueve.

- Punto (.): representa cualquier símbolo del alfabeto, pero uno sólo.

- Símbolo dólar ($): representa el final de la línea.

  a$ representa el símbolo 'a' cuando éste se encuentra al final de la línea.

- Acento circunflejo (^): este carácter tiene una doble funcionalidad, según aparezca dentro o fuera de los corchetes:

  - ^ como carácter individual, representa el principio de una línea.

    ^[a-z] encontrará párrafos que empiecen por minúscula.

  - ^ si aparece dentro de unos corchetes representa la negación.

    [^a-z] representa cualquier carácter que no sea una letra minúscula.

- Llaves ({}): sirven para indicar multiplicidades. Existen varios formatos:
  - R{n} indica que la expresión regular R aparece exactamente n veces.
  - R{n,m} indica que R puede aparecer un mínimo de n y un máximo de m veces.
  - R{n,} indica que R puede aparecer n o más veces.

Cualquier símbolo excepto el espacio en blanco, tabulador (\t), cambio de línea (\n) y los caracteres especiales se escriben tal cual en las expresiones regulares (patrones) de Lex. Los caracteres especiales son: " \ [^ - ? . * + | ( ) $ / { } % < >

Ejemplos de expresiones regulares

```
Expr. Regular |  Significado
--------------+-------------------
            x  |  El carácter x
            .  |  Cualquier carácter excepto \n
        [xyz]  |  O bien x, o bien y, o bien z
        [^bz]  |  Cualquier carácter EXCEPTO b y z
        [a-z]  |  Cualquier carácter entre a y z
       [^a-z]  |  Cualquier carácter EXCEPTO los comprendidos entre a y z
           R*  |  Ninguna R o más; R puede ser cualquier expresión regular
           R+  |  Una R o más
           R?  |  Una o cero R (es decir, R opcional)
       R{2,5}  |  De 2 a 5 R
        R{2,}  |  Dos R o más
         R{2}  |  Exactamente dos R
"[xyz\"clue"   |  La secuencia "[xyz"clue"
     {ALIAS}   |  Expansión de ALIAS, tal y como se ha definido anteriormente
          \0   |  Carácter ASCII 0
        \123   |  Carácter ASCII con código 123 en octal
        \x2A   |  Carácter ASCII con código 2A en hexadecimal
          RS   |  R seguido de S
         R|S   |  R o S
         R/S   |  R, solamente si va seguido de S
          ^R   |  R, solamente al principio de una línea
          R$   |  R, solamente al final de una línea
       <S>R$   |  R, solamente en la condición de arranque S
   <S1,S2>R$   |  R, solamente en cualquiera de las condiciones S1 o S2
     <<EOF>>   |  Fin de fichero
```

3. Estructura de un programa Lex/Flex

Como ya se ha comentado Lex/Flex genera un analizador léxico a partir de una descripción basada en expresiones regulares. Un programa para esta herramienta se divide en tres secciones, separadas por líneas que contienen solamente dos símbolos %, tal y como sigue:

```
{Sección de declaraciones}
%%
{Sección de reglas}
%%
{Sección de código}
```

La **sección de declaraciones** o definiciones contiene declaraciones para simplificar la especificación del escáner y hacerla más comprensible; también puede incluir condiciones de arranque que permiten el cambio de contexto y que se explicarán posteriormente. Estas declaraciones son del tipo ***nombre definición***, donde *nombre* es un identificador que el usuario le da a la declaración y *definición* es una expresión regular. Se considera que una definición comienza en el primer carácter que no sea un espacio en blanco siguiendo al nombre y continuando hasta el final de la línea. El objetivo de la sección de definiciones es simplificar expresiones regulares más complejas y hacer más comprensible el programa.

```
nombre1 definición1
nombre2 definición2…

%%
{Sección de reglas}
%%
{Sección de código}
```

Un ejemplo concreto de una declaración de este tipo podría ser **DIGITO [0-9]**, donde **[0-9]** es una expresión regular que define un rango de caracteres que van del carácter cero al carácter nueve. A esa expresión regular se la ha denominado **DIGITO**, posteriormente en la sección de reglas cuando se haga referencia a **{DIGITO}** (nombre de la definición entre llaves), se estará refiriendo a la expresión regular [0-9].

En esta sección también se pueden definir macros (#define…), importar archivos de cabecera (#include …) y especificar código C auxiliar (procedimientos, declaración de tipos, etc.); este código C debe ir entre los símbolos **%{** y **%}**, como se ve en el ejemplo:

```
%{
#include <stdio.h>
#define HOLA "hola"

int foo() {
     printf("Hola mundo\n");
}

//typedefs, declaración de variables, etc.

%}

nombre1 definición1
nombre2 definición2…
%%
{Sección de reglas}
%%
{Sección de código}
```

Además de todo lo anterior, pueden definirse también dentro de esta sección condiciones de arranque (*start conditions*), que definen entornos dentro de los cuáles se podrán reconocer sub-expresiones dependiendo del contexto determinado, de forma que se permite generar una especie de "mini-analizadores" dentro del analizador global.

```
%{
#include <stdio.h>
%}

%start V1C

%%
                    ...
cant  {BEGIN V1C; /* Comienza el entorno V1C */}
                    ...
<V1C>o  {printf("1ra. Persona presente indicativo");}
<V1C>as {printf("2da. Persona presente indicativo");}
<V1C>a  {printf("3ra. Persona presente indicativo");}
                    ...
%%
{Sección de código}
```

Sección de reglas (reglas de traducción): es la sección más importante del fichero fuente, pues asocia patrones a sentencias de código C. Contiene una serie de reglas de la forma ***patrón acción***, donde el *patrón* debe especificarse sin sangría y la *acción* debe comenzar en la misma línea e ir normalmente encerrada entre llaves. Los **patrones** se especifican utilizando un conjunto extendido de **expresiones regulares**, como las que se han descrito; y la acción será, habitualmente, un **conjunto de instrucciones en C.** La presencia de la sección de reglas es opcional, es más, si se omite, se podría omitir también el segundo '%%' en el fichero de entrada.

Si la entrada concuerda con más de una expresión regular, se elegirá aquella en la que coincida un mayor número de caracteres; cuando el número de caracteres concordantes sea el mismo, se seleccionará la expresión regular que aparezca en primer lugar en el fichero de Lex/Flex, por lo que las expresiones regulares más específicas se deberían situar antes que las más generales. Por ejemplo, la expresión "." (equivalente a cualquier carácter excepto \n) deberá especificarse al final de la sección de reglas, ya que de otro modo siempre se ejecutaría la acción asociada a dicha expresión, sin llegar nunca a llevarse acabo ninguna otra acción.

```
%{
int suma = 0;
%}

DIGITO [0-9]
%%
{DIGITO}+ {suma += atoi(yytext);}
%%
int main() {
      yylex();
      printf("La suma es %d\n", suma);
      return 0;
}
```

La siguiente entrada en el programa anterior, **1 2adfaf1 2 s – 2 -2**, produciría como salida:

```
 adfaf s –  -
La suma es 10
```

Si no se especifica ninguna acción, se aplica la acción por defecto, es decir, se copia la secuencia leída en la salida. Por ejemplo, lo siguiente es la especificación de un analizador

que comprime varios espacios en blanco y tabuladores a un solo espacio en blanco, y desecha los espacios que se encuentren al final de una línea:

```
%%
[ \t]+   putchar( ' ');
[ \t]+$ /* ignora este token */
```

Si la acción contiene una llave '{', entonces abarcará hasta que se encuentre el correspondiente '}', y pudiendo entonces especificarse en varias líneas. Lex/Flex es capaz de reconocer las cadenas y comentarios de C, ya que no se confundirá con las llaves que encuentre dentro de éstos. Por último, una acción que consista solamente en una barra vertical (|) significaría que se debería ejecutar la misma acción que se especifique para la siguiente regla.

Las acciones también pueden incluir código C arbitrario, incluyendo sentencias *return* para devolver un valor desde cualquier función llamada **yylex().** Cada vez que se llama a la función yylex(), ésta continúa procesando *tokens* desde donde terminó la última vez, hasta que o bien llegue al final del fichero o bien se ejecute un *return*. También se pueden llamar a funciones definidas por el usuario en la parte de código de usuario del archivo fuente de Lex/Flex.

Además, existen algunas directivas especiales que pueden incluirse dentro de una acción: ECHO que copia la entrada reconocida a la salida del escáner, BEGIN seguido del nombre de la condición de arranque que pone al escáner en el contexto correspondiente, etc.

En resumen, el comportamiento de un programa en Lex/Flex es el siguiente: para todo aquello que reconoce, ejecuta las reglas asociadas a los patrones, y lo que no conoce se limita a volcarlo por la salida carácter a carácter, de ahí la salida mostrada en el ejemplo anterior de suma de dígitos, ya que el programa sólo reconoce números enteros positivos.

Sección de código: contiene sentencias en código C y funciones que serán directamente trasladadas al archivo fuente generado. Entre estas sentencias se puede especificar código que podrá ser invocado por las reglas en las acciones (también podrían ir en la sección de declaraciones como ya se ha visto). En esta sección se suele declarar el método **main** que invoca al analizador léxico **yylex**.

No se debe olvidar, como concepto de C, que si la implementación de los procedimientos se realiza después de su invocación (en el caso de Lex/Flex, lo más probable es que se hayan invocado en la sección de reglas), deben haber sido declarados previamente (en la sección de definiciones).

```
{Sección de declaraciones}
%%
{Sección de reglas}
%%
int main() {
    yylex();
}
```

Como se ha ido reflejando en algunos ejemplos, Lex/Flex incorpora algunas variables, funciones, procedimientos y macros que permiten realizar de una forma más sencilla todo el procesamiento de la entrada.

Como variables, las más utilizadas son: **yytext** apunta al texto del *token* actual o última palabra reconocida en algún patrón (por ejemplo printf("%s", yytext) lo escribiría por pantalla), **yyleng** que contiene la longitud del *token* actual, **yyin** que referencia al fichero de entrada o **yylval** que contiene la estructura de datos de la pila con la que trabaja el analizador sintáctico YACC/Bison y que, por tanto, sirve para el intercambio de información entre ambas herramientas.

Como métodos (funciones y procedimientos) destacarían: **yylex**() que invoca al analizador léxico dando lugar al comienzo del procesamiento, **yymore**() que añade el *yytext* actual al siguiente reconocido, **yyless**(n) que devuelve los *n* últimos caracteres de la cadena *yytext* a la entrada, **yyerror**() que se invoca automáticamente cuando no se puede aplicar ninguna regla o **yywrap**() que se invoca automáticamente al encontrar el final del fichero de entrada.

4. Funcionamiento del analizador

Cuando el escáner generado está funcionando, analizará su entrada buscando cadenas que concuerden con cualquiera de sus patrones. Como ya se ha comentado, si encuentra más de un emparejamiento, toma el que empareje más texto (para reglas de contexto posterior, se incluye la longitud de la parte posterior, incluso si se devuelve a la entrada). Si encuentra dos o más emparejamientos de la misma longitud, se escoge la regla listada en primer lugar en el fichero de entrada de Lex/Flex.

Una vez que se determina el emparejamiento, el texto correspondiente al emparejamiento (denominado el *token*) está disponible en el puntero global de tipo char * **yytext**, y su longitud en la variable global de tipo entero **yyleng**. Entonces la acción correspondiente al patrón emparejado se ejecuta y la entrada restante se analiza para otro posible emparejamiento.

Si no se encuentra una coincidencia con las expresiones regulares, entonces se ejecuta la regla por defecto: el siguiente carácter en la entrada se considera reconocido y se copia a la salida estándar. Así, la entrada válida más simple de Lex/Flex es %% que generará un escáner que simplemente copia su entrada (un carácter a la vez) a la salida.

Por ejemplo, si tenemos el siguiente código formando parte de la sección de reglas en un archivo fuente de Lex:

```
%%
aa {printf("1");}
aab {printf("2");}
uv {printf("3");}
xu {printf("4");}
```

Con el texto de entrada **Laabdgf xuv**, se obtendría como salida **L2dgf 4v**. El ejecutable copiará **L**, reconocerá **aab** (porque es más largo que **aa**) y realizará el printf("2"), copiará **dgf**, reconocerá **xu** (porque aunque tiene la misma longitud que **uv**, y **uv** está antes en el fuente, en el momento de reconocer la **x** Lex/Flex no reconoce el posible conflicto, puesto que sólo **xu** puede emparejarse con algo que empieza por **x**), y ejecutará el printf("4") para luego copiar la **v** que falta y que no es posible emparejar con ninguna de las expresiones regulares definidas.

5. Compilación y ejecución de un programa en Lex / Flex

Si tenemos un programa Lex/Flex en un archivo fuente con nombre *unejemplo.l*, los pasos para convertirlo en un ejecutable serían los siguientes:

```
flex unejemplo.l
/* lex se ejecutaría como: lex unejemplo.l */
gcc -o unejemplo lex.yy.c -lfl
/* si se utilizan llamadas a funciones C de otras librerías es
necesario enlazarlas, ejemplo: la librería matemática –lm */
./unejemplo < fichero_prueba.txt
```

- Lex lee los ficheros de entrada indicados, o bien la entrada estándar si no se le ha indicado ningún nombre de fichero, con la descripción de un escáner a generar. La descripción se encuentra en forma de parejas de expresiones regulares y código C. Lex genera como salida un fichero fuente en C, 'lex.yy.c', que define una función 'yylex()'.
- lex.yy.c se compila (es un programa C) y se enlaza con la librería correspondiente generándose el ejecutable (la librería lex se indica con -ll, con flex se puede indicar -ll o -lfl).
- Este ejecutable analizará la entrada léxicamente.

El fichero "lex.yy.c" contiene las tablas del autómata generado y la función "yylex", la cual simula el analizador especificado y sirve de interfaz con el código de usuario (yylex() deberá de ser llamada en algún punto del código del usuario). En cada llamada, yylex() irá tomando caracteres de la entrada hasta que empareje una de las expresiones regulares de la especificación; entonces se almacenará el texto que se ha emparejado con la expresión regular en la variable yytext y se ejecutarán las acciones asociadas al patrón. La función continúa hasta que se alcance el final del fichero (punto en el que devuelve el valor 0) o una de sus acciones ejecute una sentencia *return*.

Lex/Flex requiere un formato bastante estricto de su fichero de entrada. En particular los caracteres no visibles (espacios en blanco, tabuladores, saltos de línea) fuera de sitio causan errores difíciles de encontrar. Sobre todo es muy importante no dejar líneas en blanco de más ni empezar reglas con espacios en blanco o tabuladores.

6. Ejemplos

Ejemplo 1. Reemplazo de cadena "username" por el login del usuario

```
%%
username          printf("%s", getlogin() );
```

Ejemplo2. Identificador de números y palabras

```
digit     [0-9]
letter    [A-Za-z]
delim     [ \t\n]
ws        {delim}+
%%
{ws}      { /* ignore white space */ }
{digit}+  { printf("%s", yytext);  printf(" = number\n"); }
{letter}+ { printf("%s", yytext); printf(" = word\n"); }
%%
main()
{
       yylex();
}
```

Ejemplo 3. Un contador de palabras similar al wc de UNIX

```
%{
unsigned charCount = 0, wordCount = 0, lineCount = 0;
%}
word [^ \t\n]+
eol \n
%%
{word} {wordCount++; charCount += yyleng;}
{eol}  {charCount++; lineCount++; }
.      {charCount++;}
%%
main()
{
       yylex();
       printf("%d %d %d\n", lineCount, wordCount, charCount);
}
```

Ejemplo 4. Reconocedor de comentarios en lenguaje C.

```
%%
(.|\n)? {printf(""); }
"/*""/"*([^*/]|[^*]"/"|"*"[^/])*"*"*"*/" {printf("[comment]%s", yytext);}
%%
```

NOTA: Probar con el siguiente ejemplo:

/* comentarios en una linea */
/* comentarios sobre multiples
    lineas */
comentarios /*incluidos dentro de una */linea
comentarios /**/ raros
/*/Acepta comentarios que no lo son?
/**_/*/ Es correcto, lo acepta?

Ejemplo 5. Calculadora

```
%{
/*
**      lex program for simple dc
**      input of the form:  opd opr opd
**      output of the form: opd opr opd = ans
*/
#define NUM 1
#define PLUS 2
#define MINUS 3
#define MUL 4
#define DIV 5
#define MOD 6
#define DELIM 7
%}

ws     [ \t]+

%%
{ws}   ;
-?[0-9]+        {
        printf("%s",yytext);
        return (NUM);
        }

"+"     {
        ECHO;
        return (PLUS);
        }
"-"     {
        ECHO;
        return (MINUS);
        }
"*"     {
        ECHO;
        return (MUL);
        }
"/"     {
        ECHO;
        return (DIV);
        }
"%"     {
        ECHO;
        return (MOD);
        }
. |
"\ n"   {
        ECHO;
        return (DELIM);
        }
%%
/* main.c */
# include <stdio.h>

main()
{
int opd1, opd2, opr;

       /* Look for first operand. */
        if (yylex() != NUM)
                {
                printf("missing operand\n");
                exit(1);
                }
        opd1 = atoi(yytext);
```

```
        /* Check for operator. */
          opr = yylex();
          if (opr == DELIM)
                  {
                  printf("missing operator\n");
                  exit(1);
                  }
        else if (opr == NUM)
               {
               if ((opd2=atoi(yytext)) >= 0)
                       {
                        printf("missing operator\n");
                        exit(1);
                       }
               else
                       {
                       opr = MINUS;
                       opd2 = -opd2;
                       }
               }
          else if (yylex() != NUM)
                  {
                  printf("missing operand\n");
                  exit(1);
                  }
        else /* Must have found operand 2 */
               opd2 = atoi(yytext);
          switch (opr)
          {
          case PLUS:      {
                          printf(" = %d\n",opd1 + opd2);
                          break;
                          }
          case MINUS:     {
                          printf(" = %d\n",opd1 - opd2);
                          break;
                          }
          case MUL:       {
                          printf(" = %d\n",opd1 * opd2);
                          break;
                          }
          case DIV:       {
                      if (opd2 == 0)
                             {
                             printf("\nERROR: attempt to divide by zero!\n");
                             exit(1);
                             }
                      else
                             {
                             printf(" = %d\n",opd1 / opd2);
                             break;
                             }
                          }
          case MOD:       {
                      if (opd2 == 0)
                             {
                             printf("\nERROR: attempt to divide by zero!\n");
                             exit(1);
                             }
                      else
                             {
                                    printf(" = %d\n",opd1 % opd2);
                             break;
                             }
                          }
          }
}
```

7. Bibliografía

- Introduction to compiling techniques a first course using ANSI C, LEX and YACC. Bennet, Jeremy Peter. McGraw-Hill, [1996].
- Lex and Yacc / John R. Levine, Tony Mason, Doug Brown Levine, John R. Sebastopol. O'Reilly, [1992] XXII, 364 p.
- Flex and Bison. Text Processing Tools / John R. Levine. O'Reilly, [2009], 304 p.
- Tutoriales y manuales de Lex/Flex en la web.
  - Web oficial flex:
    http://flex.sourceforge.net/
    http://flex.sourceforge.net/manual/
  - Otros recursos:
    http://dinosaur.compilertools.net/
    http://gnuu.org/2009/09/18/writing-your-own-toy-compiler/
    http://www.mactech.com/articles/mactech/Vol.16/16.07/UsingFlexandBison
    http://www.medina-web.com/programas/documents/tutoriales/lex_yacc/