```
/* Grupo 37: Pablo Hidalgo Delgado, César López Navarro*/
/* 100451225@alumnos.uc3m.es 100451326@alumnos.uc3m.es*/
%{                          // SECCION 1 Declaraciones de C-Yacc

#include <stdio.h>
#include <ctype.h>            // declaraciones para tolower
#include <string.h>           // declaraciones para code
#include <stdlib.h>           // declaraciones para exit ()

#define FF fflush(stdout);    // para forzar la impresion inmediata

int yylex () ;
void yyerror () ;
char *my_malloc (int) ;
char *gen_code (char *) ;
char *int_to_string (int) ;
char *char_to_string (char) ;

char temp [2048] ;

char nombre_funcion [1024] ;

// Definitions for explicit attributes

typedef struct s_attr {
        int value ;
        char *code ;
} t_attr ;

#define YYSTYPE t_attr

%}

// Definitions for explicit attributes

%token NUMBER
```

```
%token IDENTIF       // Identificador=variable
%token INTEGER       // identifica el tipo entero
%token STRING
%token MAIN          // identifica el comienzo del proc. main
%token WHILE         // identifica el bucle while
%token FOR           // identifica el bucle for
%token IF            // identifica la condicion if
%token ELSE          // identifica la condicion else
%token RETURN        // identifica la funcion return
%token PUTS          // identifica la funcion puts (print a string)
%token PRINTF       // identifica la funcion printf (print varias expresiones)
%token AND          // identifica el operador logico and '&&'
%token OR           // identifica el operador logico or '||'
%token NEQ          // identifica el operador logico not equal '!='
%token EQ           // identifica el operador logico equal '=='
%token LE           // identifica el operador logico less or equal '<='
%token GE           // identifica el operador logico greater or equal '>='


// Definitions for implicit attributes.
// USE THESE ONLY AT YOUR OWN RISK
/*

%union {                      // El tipo de la pila tiene caracter dual
    int value ;             // - valor numerico de un NUMERO
    char *code ;          // - para pasar los nombres de IDENTIFES
}

%token <value> NUMBER       // Todos los token tienen un tipo para la pila
%token <code> IDENTIF       // Identificador=variable
%token <code> INTEGER       // identifica la definicion de un entero
%token <code> STRING
%token <code> MAIN          // identifica el comienzo del proc. main
%token <code> WHILE         // identifica el bucle main
%type <...> Axiom ...
```

```
*/

%right '='                    // es la ultima operacion que se debe realizar
%left OR                      // orden de precedencia 0
%left AND                     // orden de precedencia 1
%left NEQ EQ                  // orden de precedencia 2
%left '>' GE '<' LE           // orden de precedencia 3
%left '+' '-'                 // orden de precedencia 4
%left '*' '/' '%'             // orden de precedencia 5
%left UNARY_SIGN              // mayor orden de precedencia

%%                            // Seccion 3 Gramatica - Semantico

/* ---------------- AXIOMA DE LA GRAMÁTICA ----------------  */
axioma:       sentencia_global ';'                                                    { printf ("%s\n",
$1.code) ; }
               r_expr                                                                { ; }

           | IDENTIF {sprintf (nombre_funcion, "%s", $1.code) ; } '(' mas_argumentos ')' '{' mas_sentencias '}' { printf
("(defun %s(%s)\n%s)\n", $1.code, $4.code, $7.code) ; }
               r_func                                                                   { ; }

            | MAIN {sprintf (nombre_funcion, "%s", $1.code) ; } '(' mas_argumentos ')' '{' mas_sentencias '}'   { printf
("(defun %s(%s)\n%s)\n", $1.code, $4.code, $7.code) ;  }
                                                                                  { ; }
            ;

/* ---------------- RECURSIVIDAD DE SENTENCIAS GLOBALES ----------------  */
r_expr:    axioma                                  { ; }
            ;

/* ---------------- DEFINICIÓN DE FUNCIONES ----------------  */
r_func:       IDENTIF {sprintf (nombre_funcion, "%s", $1.code) ; } '(' mas_argumentos ')' '{' mas_sentencias '}' { printf
("(defun %s(%s)\n%s)\n", $1.code, $4.code, $7.code) ; }
               r_func                                                                { ; }
```

```
          |  MAIN {sprintf (nombre_funcion, "%s", $1.code) ; } '(' mas_argumentos ')' '{' mas_sentencias '}'          { printf
("(defun %s(%s)\n%s)\n", $1.code, $4.code, $7.code) ;  }
          ;

/* ---------------- SENTENCIAS GLOBALES ----------------  */
sentencia_global:    IDENTIF '=' expresion                          { sprintf (temp, "(setq %s %s)", $1.code, $3.code) ;
                                                                     $$.code = gen_code (temp) ; }
                | IDENTIF mas_variables '='
                            expresion mas_expresiones                    { sprintf (temp, "(setf (values %s %s) (values %s
%s))", $1.code, $2.code, $4.code, $5.code) ;
                                                                   $$.code = gen_code (temp) ; }
                 | vector '=' expresion                                  { sprintf (temp, "(setf %s %s)", $1.code, $3.code) ;
                                                                   $$.code = gen_code (temp) ; }
                 | declaracion                                     { sprintf (temp, "%s", $1.code) ;
                                                                   $$.code = gen_code (temp) ; }
                | PUTS '(' STRING ')'                              { sprintf (temp, "(print \"%s\")", $3.code) ;
                                                                   $$.code = gen_code (temp) ; }
                | PRINTF '(' STRING ',' resto_parametros_printf ')'  { sprintf (temp, "%s", $5.code) ;
                                                                   $$.code = gen_code (temp) ; }
           ;

/* ---------------- ARGUMENTOS DE LAS FUNCIONES ----------------  */
mas_argumentos: argumento ',' mas_argumentos                  {sprintf (temp, "%s %s", $1.code, $3.code) ;
                                                            $$.code = gen_code (temp) ; }
              | argumento                                  { sprintf (temp, "%s", $1.code) ;
                                                            $$.code = gen_code (temp) ; }

argumento: INTEGER IDENTIF                                  { sprintf (temp, "%s", $2.code) ;
                                                            $$.code = gen_code (temp) ; }
          | /*lambda*/                                            { $$.code = "" ; }
          ;


/* ---------------- SENTENCIAS DE LAS FUNCIONES ---------------- */
mas_sentencias: sentencia_local  mas_sentencias                          { sprintf (temp, "%s%s", $1.code, $2.code) ;
                                                            $$.code = gen_code (temp) ; }
```

```
              | /*lambda*/                                  { $$.code = "" ; }
              ;

sentencia_local:    IDENTIF '=' expresion ';'                      { sprintf (temp, "(setq %s %s)\n", $1.code, $3.code) ;
                                                                     $$.code = gen_code (temp) ; }

           | IDENTIF mas_variables '='
             expresion mas_expresiones ';'                               { sprintf (temp, "(setf (values %s %s) (values %s
%s))\n", $1.code, $2.code, $4.code, $5.code) ;
                                                                  $$.code = gen_code (temp) ; }
            | vector '=' expresion ';'                             { sprintf (temp, "(setf %s %s)\n", $1.code, $3.code) ;
                                                                  $$.code = gen_code (temp) ; }
            | declaracion ';'                                       { sprintf (temp, "%s\n", $1.code) ;
                                                                  $$.code = gen_code (temp) ; }
           | PUTS '(' STRING ')' ';'                                { sprintf (temp, "(print \"%s\")\n", $3.code) ;
                                                                  $$.code = gen_code (temp) ; }
           | PRINTF '(' STRING ',' resto_parametros_printf ')' ';'    { sprintf (temp, "%s", $5.code) ;
                                                                  $$.code = gen_code (temp) ; }
            | llamada_funcion ';'                                          { sprintf (temp, "%s\n", $1.code) ;
                                                                    $$.code = gen_code (temp) ; }
            | return ';'                                            { sprintf (temp, "%s\n", $1.code) ;
                                                                    $$.code = gen_code (temp) ; }
           | WHILE '(' condicion ')'
              '{' mas_sentencias '}'                                      { sprintf (temp, "(loop while %s do %s)\n", $3.code,
$6.code) ;
                                                                      $$.code = gen_code (temp) ; }
           | FOR '(' inicializacion ';' condicion ';' incremento ')'
              '{' mas_sentencias '}'                                 { sprintf(temp, "%s(loop while %s do %s %s)\n", $3.code,
$5.code, $10.code, $7.code) ;
                                                                   $$.code = gen_code (temp) ; }
           | IF '(' condicion ')'
              '{' cuerpo_if '}'                             { sprintf (temp, "(if %s %s)\n", $3.code, $6.code) ;
                                                                    $$.code = gen_code (temp) ; }
           | IF '(' condicion ')'
              '{' cuerpo_if '}' ELSE
                '{' cuerpo_if '}'                           { sprintf (temp, "(if %s %s %s)\n", $3.code, $6.code, $10.code) ;
                                                                  $$.code = gen_code (temp) ; }
```

```
            ;

cuerpo_if: sentencia_local                                  { sprintf (temp, "%s", $1.code) ;
                                                            $$.code = gen_code (temp) ; }
         | sentencia_local sentencia_local mas_sentencias    { sprintf (temp, "(progn %s %s %s)", $1.code, $2.code, $3.code) ;
                                                            $$.code = gen_code (temp) ; }

/* ---------------- INICIALIZACIÓN DEL BUCLE FOR ---------------- */
inicializacion: INTEGER IDENTIF                              { sprintf (temp, "(setq %s 0)", $2.code) ;
                                                            $$.code = gen_code (temp) ; }
             | INTEGER IDENTIF '=' NUMBER                    { sprintf (temp, "(setq %s %d)", $2.code, $4.value) ;
                                                            $$.code = gen_code (temp) ; }
             | IDENTIF '=' NUMBER                           { sprintf (temp, "(setq %s %d)", $1.code, $3.value) ;
                                                            $$.code = gen_code (temp) ; }
           ;

/* ---------------- INCREMENTO EN EL BUCLE FOR ---------------- */
incremento:  IDENTIF '=' IDENTIF '+' NUMBER                  { sprintf (temp, "(setq %s (+ %s %d))", $1.code, $3.code,
$5.value) ;
                                                            $$.code = gen_code (temp) ; }

          | IDENTIF '=' IDENTIF '-' NUMBER                   { sprintf (temp, "(setq %s (- %s %d))", $1.code, $3.code, $5.value)
;
                                                            $$.code = gen_code (temp) ; }

           ;
/* ---------------- CONDICIONES ---------------- */
condicion: expresion_logica                                     { sprintf (temp, "%s", $1.code) ;
                                                                $$.code = gen_code (temp) ; }
         | expresion_aritmetica                                 { sprintf (temp, "%s", $1.code) ;
                                                                 $$.code = gen_code (temp) ; }

/* ---------------- EXPRESIONES LOGICAS EN LA CONDICION ---------------- */
expresion_logica:   termino                                       { $$ = $1 ; }
                  | vector                                      { $$ = $1 ; }
                  | llamada_funcion                              { $$ = $1 ; }
```

```
                  | condicion AND condicion                    { sprintf (temp, "(And %s %s)", $1.code, $3.code) ;
                                                             $$.code = gen_code (temp) ; }
                  | condicion OR condicion                          { sprintf (temp, "(or %s %s)", $1.code, $3.code) ;
                                                             $$.code = gen_code (temp) ; }
                  | condicion NEQ condicion                  { sprintf (temp, "(/= %s %s)", $1.code, $3.code) ;
                                                             $$.code = gen_code (temp) ; }
                  | condicion EQ condicion                   { sprintf (temp, "(= %s %s)", $1.code, $3.code) ;
                                                             $$.code = gen_code (temp) ; }
                  | condicion '<' condicion                         { sprintf (temp, "(< %s %s)", $1.code, $3.code) ;
                                                             $$.code = gen_code (temp) ; }
                  | condicion LE condicion                   { sprintf (temp, "(<= %s %s)", $1.code, $3.code) ;
                                                             $$.code = gen_code (temp) ; }
                  | condicion '>' condicion                   { sprintf (temp, "(> %s %s)", $1.code, $3.code) ;
                                                             $$.code = gen_code (temp) ; }
                  | condicion GE condicion                          { sprintf (temp, "(>= %s %s)", $1.code, $3.code) ;
                                                             $$.code = gen_code (temp) ; }
                  | condicion '%' condicion                         { sprintf (temp, "(mod %s %s)", $1.code, $3.code) ;
                                                             $$.code = gen_code (temp) ; }
                  ;

/* ---------------- EXPRESIONES ARITMETICAS EN LA CONDICION ---------------- */
expresion_aritmetica:    condicion '+' condicion                    { sprintf (temp, "(/= 0 (+ %s %s))", $1.code,
$3.code) ;
                                                                              $$.code = gen_code (temp) ; }
                     | condicion '-' condicion                  { sprintf (temp, "(/= 0 (- %s %s))", $1.code,
$3.code) ;
                                                                              $$.code = gen_code (temp) ; }
                       | condicion '*' condicion                  { sprintf (temp, "(/= 0 (* %s %s))", $1.code,
$3.code) ;
                                                                              $$.code = gen_code (temp) ; }
                     | condicion '/' condicion                  { sprintf (temp, "(/= 0 (/ %s %s))", $1.code,
$3.code) ;
                                                                              $$.code = gen_code (temp) ; }


/* ---------------- LLAMADAS A FUNCIONES ---------------- */
```

```
llamada_funcion: IDENTIF '(' mas_parametros ')'                          { sprintf (temp, "(%s %s)", $1.code, $3.code) ;
                                                                         $$.code = gen_code (temp) ; }
              ;

/* ---------------- PARAMETROS PASADOS A LAS LLAMADAS A FUNCIONES ---------------- */
mas_parametros: parametro ',' mas_parametros                               { sprintf (temp, "%s %s", $1.code, $3.code) ;
                                                                           $$.code = gen_code (temp) ; }
              | parametro                                                 { sprintf (temp, "%s", $1.code) ;
                                                                           $$.code = gen_code (temp) ; }
parametro: expresion                                                  { sprintf (temp, "%s", $1.code) ;
                                                                       $$.code = gen_code (temp) ; }
          | /*lambda*/                                                      { $$.code = "" ; }
          ;

/* ---------------- RETURN ---------------- */
return: RETURN expresion                                               { sprintf (temp, "(return-from %s %s)", nombre_funcion, $2.code) ;
                                                                       $$.code = gen_code (temp) ; }
       | RETURN                                                       { sprintf (temp, "(return-from %s)", nombre_funcion) ;
                                                                       $$.code = gen_code (temp) ; }
       ;

/* ---------------- ASIGNACION DE VALORES A VARIAS VARIABLES A LA VEZ ---------------- */
mas_variables: ',' IDENTIF mas_variables                                 { sprintf (temp, "%s %s", $2.code, $3.code) ;
                                                                       $$.code = gen_code (temp) ; }
            | ',' IDENTIF                                             { sprintf (temp, "%s", $2.code) ;
                                                                       $$.code = gen_code (temp) ; }
            ;

mas_expresiones: ',' expresion mas_expresiones                            { sprintf (temp, "%s %s", $2.code, $3.code) ;
                                                                       $$.code = gen_code (temp) ; }
              | ',' expresion                                                { sprintf (temp, "%s", $2.code) ;
                                                                       $$.code = gen_code (temp) ; }
              ;

/* ---------------- PARAMETROS DE LA FUNCION PRINTF ---------------- */
resto_parametros_printf:  expresion                                    { sprintf (temp, "(print %s)", $1.code) ;
```

```
                                                        $$.code = gen_code (temp) ; }
          | expresion ',' resto_parametros_printf          { sprintf (temp, "(print %s) %s", $1.code, $3.code) ;
                                                        $$.code = gen_code (temp) ; }

          ;

/* ---------------- VECTOR ---------------- */
vector:  IDENTIF '[' dentro_vector ']'                          { sprintf (temp, "(aref %s %s)", $1.code, $3.code) ;
                                                        $$.code = gen_code (temp) ; }

          ;

/* ---------------- POSIBLES VALORES DENTRO DE LOS CORCHETES DE UN VECTOR ---------------- */
dentro_vector: termino_dentro_vector                            { $$ = $1 ; }

            | dentro_vector '+' dentro_vector                    { sprintf (temp, "(+ %s %s)", $1.code, $3.code) ;
                                                         $$.code = gen_code (temp) ; }
            | dentro_vector '-' dentro_vector                    { sprintf (temp, "(- %s %s)", $1.code, $3.code) ;
                                                         $$.code = gen_code (temp) ; }
            ;

termino_dentro_vector: operando_dentro_vector                                   { $$ = $1 ; }

                | '+' operando_dentro_vector %prec UNARY_SIGN          { sprintf (temp, "(+ %s)", $2.code) ;
                                                                      $$.code = gen_code (temp) ; }
                | '-' operando_dentro_vector %prec UNARY_SIGN          { sprintf (temp, "(- %s)", $2.code) ;
                                                                      $$.code = gen_code (temp) ; }
                ;

operando_dentro_vector:   IDENTIF                                 { sprintf (temp, "%s", $1.code) ;
                                                                 $$.code = gen_code (temp) ; }
                      | NUMBER                                  { sprintf (temp, "%d", $1.value) ;
                                                                 $$.code = gen_code (temp) ; }
                ;

/* ---------------- DECLARACION DE VARIABLES ---------------- */
declaracion:  INTEGER IDENTIF  resto_declaracion             { sprintf (temp, "(setq %s 0) %s", $2.code, $3.code) ;
                                                        $$.code = gen_code (temp) ; }
```

```
          | INTEGER IDENTIF '=' expresion resto_declaracion         { sprintf (temp, "(setq %s %s) %s", $2.code, $4.code,
$5.code) ;
                                                                    $$.code = gen_code (temp) ; }
          | INTEGER IDENTIF '[' NUMBER ']'                          { sprintf (temp, "(setq %s (make-array %d))", $2.code, $4.value) ;
                                                                    $$.code = gen_code (temp) ; }
           ;


resto_declaracion:  ',' IDENTIF resto_declaracion            { sprintf (temp, "(setq %s 0) %s", $2.code, $3.code) ;
                                                             $$.code = gen_code (temp) ; }
                 | ',' IDENTIF '=' expresion resto_declaracion   { sprintf (temp, "(setq %s %d) %s", $2.code, $4.value,
$5.code) ;
                                                              $$.code = gen_code (temp) ; }
                | /*lambda*/                                      { $$.code = "" ; }
                ;

/* ---------------- EXPRESIONES ---------------- */
expresion:    termino                                                 { $$ = $1 ; }

          | llamada_funcion                                 { $$ = $1 ; }

          | vector                                         { $$ = $1 ; }

           | expresion '+' expresion                                  { sprintf (temp, "(+ %s %s)", $1.code, $3.code) ;
                                                                    $$.code = gen_code (temp) ; }
           | expresion '-' expresion                                  { sprintf (temp, "(- %s %s)", $1.code, $3.code) ;
                                                                    $$.code = gen_code (temp) ; }
           | expresion '*' expresion                                  { sprintf (temp, "(* %s %s)", $1.code, $3.code) ;
                                                                    $$.code = gen_code (temp) ; }
           | expresion '/' expresion                                  { sprintf (temp, "(/ %s %s)", $1.code, $3.code) ;
                                                                    $$.code = gen_code (temp) ; }
          | expresion '%' expresion                                    { sprintf (temp, "(mod %s %s)", $1.code, $3.code) ;
                                                                  $$.code = gen_code (temp) ; }
           | expresion AND expresion                                  { sprintf (temp, "(And %s %s)", $1.code, $3.code) ;
                                                                  $$.code = gen_code (temp) ; }
          | expresion OR expresion                                    { sprintf (temp, "(or %s %s)", $1.code, $3.code) ;
```

```
            | expresion NEQ expresion                   $$.code = gen_code (temp) ; }
                                                      { sprintf (temp, "(/= %s %s)", $1.code, $3.code) ;
             | expresion EQ expresion                                 $$.code = gen_code (temp) ; }
                                                       { sprintf (temp, "(= %s %s)", $1.code, $3.code) ;
             | expresion '<' expresion                                $$.code = gen_code (temp) ; }
                                                                    { sprintf (temp, "(< %s %s)", $1.code, $3.code) ;
             | expresion LE expresion                                 $$.code = gen_code (temp) ; }
                                                       { sprintf (temp, "(<= %s %s)", $1.code, $3.code) ;
             | expresion '>' expresion                                $$.code = gen_code (temp) ; }
                                                                    { sprintf (temp, "(> %s %s)", $1.code, $3.code) ;
             | expresion GE expresion                 $$.code = gen_code (temp) ; }
                                                                    { sprintf (temp, "(>= %s %s)", $1.code, $3.code) ;
                                                      $$.code = gen_code (temp) ; }
             ;

/* ---------------- TERMINO ---------------- */
termino:        operando                                  { $$ = $1 ; }

             |   '+' operando %prec UNARY_SIGN            { sprintf (temp, "(+ %s)", $2.code) ;
                                                             $$.code = gen_code (temp) ; }
             |   '-' operando %prec UNARY_SIGN            { sprintf (temp, "(- %s)", $2.code) ;
                                                             $$.code = gen_code (temp) ; }
             ;

/* ---------------- OPERANDO ---------------- */
operando:       IDENTIF                                   { sprintf (temp, "%s", $1.code) ;
                                                             $$.code = gen_code (temp) ; }
             |   NUMBER                                   { sprintf (temp, "%d", $1.value) ;
                                                             $$.code = gen_code (temp) ; }
             |   '(' expresion ')'                        { $$ = $2 ; }
             ;


%%                          // SECCION 4    Codigo en C
```

```
int n_line = 1 ;

void yyerror (char *message)
{
    fprintf (stderr, "%s in line %d\n", message, n_line) ;
    printf ( "\n") ;
}

char *int_to_string (int n)
{
    sprintf (temp, "%d", n) ;
    return gen_code (temp) ;
}

char *char_to_string (char c)
{
    sprintf (temp, "%c", c) ;
    return gen_code (temp) ;
}

char *my_malloc (int nbytes)       // reserva n bytes de memoria dinamica
{
    char *p ;
    static long int nb = 0;        // sirven para contabilizar la memoria
    static int nv = 0 ;            // solicitada en total

    p = malloc (nbytes) ;
    if (p == NULL) {
        fprintf (stderr, "No memoria left for additional %d bytes\n", nbytes) ;
        fprintf (stderr, "%ld bytes reserved in %d calls\n", nb, nv) ;
        exit (0) ;
    }
    nb += (long) nbytes ;
    nv++ ;

    return p ;
```

```
}


/*****************************************************************************/
/********************** Seccion de Palabras Reservadas ***********************/
/*****************************************************************************/

typedef struct s_keyword { // para las palabras reservadas de C
    char *name ;
    int token ;
} t_keyword ;

t_keyword keywords [] = { // define las palabras reservadas y los
    "main",        MAIN,           // y los token asociados
    "int",         INTEGER,
    "puts",        PUTS,
    "printf",      PRINTF,
    "while",       WHILE,
    "for",         FOR,
    "if",          IF,
    "else",        ELSE,
    "return",      RETURN,
    "&&",          AND,
    "||",          OR,
    "!=",         NEQ,
    "==",         EQ,
    "<=",         LE,
    ">=",         GE,
    NULL,          0              // para marcar el fin de la tabla
} ;

t_keyword *search_keyword (char *symbol_name)
{                                  // Busca n_s en la tabla de pal. res.
                                   // y devuelve puntero a registro (simbolo)
    int i ;
    t_keyword *sim ;
```

```
    i = 0 ;
    sim = keywords ;
    while (sim [i].name != NULL) {
           if (strcmp (sim [i].name, symbol_name) == 0) {
                                          // strcmp(a, b) devuelve == 0 si a==b
            return &(sim [i]) ;
        }
        i++ ;
    }

    return NULL ;
}


/*****************************************************************************/
/****************** Seccion del Analizador Lexicografico *********************/
/*****************************************************************************/

char *gen_code (char *name)     // copia el argumento a un
{                                      // string en memoria dinamica
    char *p ;
    int l ;

    l = strlen (name)+1 ;
    p = (char *) my_malloc (l) ;
    strcpy (p, name) ;

    return p ;
}


int yylex ()
{
    int i ;
    unsigned char c ;
```

```
unsigned char cc ;
char expandable_ops [] = "!<=>|%/&+-*" ;
char temp_str [256] ;
t_keyword *symbol ;

do {
    c = getchar () ;

    if (c == '#') {        // Ignora las lineas que empiezan por #  (#define, #include)
        do {              //     OJO que puede funcionar mal si una linea contiene #
            c = getchar () ;
        } while (c != '\n') ;
    }

    if (c == '/') {        // Si la linea contiene un / puede ser inicio de comentario
        cc = getchar () ;
        if (cc != '/') {   // Si el siguiente char es /  es un comentario, pero...
            ungetc (cc, stdin) ;
        } else {
            c = getchar () ;       // ...
            if (c == '@') {        // Si es la secuencia //@  ==> transcribimos la linea
                do {              // Se trata de codigo inline (Codigo embebido en C)
                    c = getchar () ;
                    putchar (c) ;
                } while (c != '\n') ;
            } else {              // ==> comentario, ignorar la linea
                while (c != '\n') {
                    c = getchar () ;
                }
            }
        }
    } else if (c == '\\') c = getchar () ;

    if (c == '\n')
        n_line++ ;
```

```
    } while (c == ' ' || c == '\n' || c == '\r' || c == 10 || c == 13 || c == '\t') ;

    if (c == '\"') {
        i = 0 ;
        do {
            c = getchar () ;
            temp_str [i++] = c ;
        } while (c != '\"' && i < 255) ;
        if (i == 256) {
            printf ("WARNING: string with more than 255 characters in line %d\n", n_line) ;
        }                   // habria que leer hasta el siguiente " , pero, y si falta?
        temp_str [--i] = '\0' ;
        yylval.code = gen_code (temp_str) ;
        return (STRING) ;
    }

    if (c == '.' || (c >= '0' && c <= '9')) {
        ungetc (c, stdin) ;
        scanf ("%d", &yylval.value) ;
//         printf ("\nDEV: NUMBER %d\n", yylval.value) ;        // PARA DEPURAR
        return NUMBER ;
    }

    if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')) {
        i = 0 ;
        while (((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z') ||
            (c >= '0' && c <= '9') || c == '_') && i < 255) {
            temp_str [i++] = tolower (c) ;
            c = getchar () ;
        }
        temp_str [i] = '\0' ;
        ungetc (c, stdin) ;

        yylval.code = gen_code (temp_str) ;
        symbol = search_keyword (yylval.code) ;
        if (symbol == NULL) {    // no es palabra reservada -> identificador antes vrariabre
```

```
//              printf ("\nDEV: IDENTIF %s\n", yylval.code) ;    // PARA DEPURAR
            return (IDENTIF) ;
        } else {
//              printf ("\nDEV: OTRO %s\n", yylval.code) ;       // PARA DEPURAR
            return (symbol->token) ;
        }
    }

    if (strchr (expandable_ops, c) != NULL) { // busca c en operadores expandibles
        cc = getchar () ;
        sprintf (temp_str, "%c%c", (char) c, (char) cc) ;
        symbol = search_keyword (temp_str) ;
        if (symbol == NULL) {
            ungetc (cc, stdin) ;
            yylval.code = NULL ;
            return (c) ;
        } else {
            yylval.code = gen_code (temp_str) ; // aunque no se use
            return (symbol->token) ;
        }
    }

//    printf ("\nDEV: LITERAL %d #%c#\n", (int) c, c) ;      // PARA DEPURAR
    if (c == EOF || c == 255 || c == 26) {
//        printf ("tEOF ") ;                                 // PARA DEPURAR
        return (0) ;
    }

    return c ;
}


int main ()
{
    yyparse () ;
}
```