```
// Grupo 403, Mario Ramos Salson y Miguel Yubero Espinosa
// 100495849@alumnos.uc3m.es 100495984@alumnos.uc3m.es

%{  // SECCION 1 Declaraciones de C-Yacc

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

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

// Declaraciones de funciones y variables globales
void limpiar_tabla_local();
void insertar_variable_local(char *nombre);
int es_variable_local(char *nombre);
extern int num_locales;
char current_function[256] = "";  // Para guardar el nombre de la funciÃ³n actua
l

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

char temp[2048];

// Abstract Syntax Tree (AST) Node Structure
typedef struct ASTnode t_node;

struct ASTnode {
    char *op;
    int type;     // leaf, unary or binary nodes
    t_node *left;
    t_node *right;
};

// Definitions for explicit attributes
typedef struct s_attr {
    int value;    // - Numeric value of a NUMBER
    char *code;   // - to pass IDENTIFIER names, and other translations
    char *op;
    t_node *node; // - for possible future use of AST
} t_attr;

#define YYSTYPE t_attr
%}

// Definitions for explicit attributes

%token NUMBER
%token IDENTIF
%token INTEGER
%token STRING
%token MAIN
%token WHILE
%token PUTS
%token PRINTF
%token IF
%token ELSE
```



```
%token FOR
%token RETURN

// En la secciÃ³n de declaraciones, ajusta el orden de precedencia:

%right '='
%left OR
%left AND
%left EQUAL DIST
%left '<' '>' GREATEREQ LESSEQ
%left '+' '-'
%left '*' '/' '%'
%right '!'
%right UNARY_SIGN

// Seccion 3 Gramatica - Semantico
%%

// -------------------------AXIOMA-----------------------------------------

axioma: printGlobales printFunciones main { ; }

// ------------------------------------------------------------------------
-

// -------------------------DECLARACION MAIN-------------------------------
-

main: MAIN '(' ')' { strcpy(current_function, "main"); }
    '{' cuerpo '}' {
        limpiar_tabla_local();
        sprintf(temp, "\n(defun main ()\n\n%s\n)\n\n", $6.code);
        $$.code = gen_code(temp);
        strcpy(current_function, "");  // Borramos funcion actual
        printf("%s", $$.code);
    }
;

// ------------------------------------------------------------------------
-

// --------------------DECLARACION VARIABLES GLOBALES----------------------
-

printGlobales: { $$.code = NULL; }
    | globales {
        printf("%s", $$.code);
    }
;

globales: creacionVarGlobales {
        sprintf(temp, "%s", $1.code);
        $$.code = gen_code(temp);
    }

    | creacionVarGlobales globales {
        sprintf(temp, "%s\n%s", $1.code, $2.code);
        $$.code = gen_code(temp);
    }
;

creacionVarGlobales: INTEGER IDENTIF asignacionVar concatenacionVarGlobales ';' {
```

```
        if ($3.code == NULL && $4.code == NULL) {
            sprintf(temp, "(setq %s 0)\n", $2.code);  // Valor por defecto 0 si no h
ay asignaciÃ³n
        } else if ($4.code == NULL) {
            sprintf(temp, "(setq %s %s)\n", $2.code, $3.code);
        } else if ($3.code == NULL) {
            sprintf(temp, "(setq %s 0) %s\n", $2.code, $4.code);
        }else {
            sprintf(temp, "(setq %s %s) %s\n", $2.code, $3.code, $4.code);
        }
        $$.code = gen_code(temp);
    }
    | INTEGER IDENTIF '[' NUMBER ']' concatenacionVarGlobales ';' {
        if ($6.code == NULL) {
            sprintf(temp, "(setq %s (make-array %d))\n", $2.code, $4.value);
        } else {
            sprintf(temp, "(setq %s (make-array %d)) %s\n", $2.code, $4.value, $6.code)
;
        }
        $$.code = gen_code(temp);
    }
;

asignacionVar: { $$.code = NULL; }  // Lambda
    | '=' expresion { sprintf(temp, "%s", $2.code);
                  $$.code = gen_code(temp);
    }
;

concatenacionVarGlobales: { $$.code = NULL; }  // Lambda
    | ',' IDENTIF concatenacionVarGlobales {
        if ($3.code == NULL) {
            sprintf(temp, "\n(setq %s 0)\n", $2.code);
        } else {
            sprintf(temp, "\n(setq %s 0) %s\n", $2.code, $3.code);
        }
        $$.code = gen_code(temp);
    }

    | ',' IDENTIF '[' NUMBER ']' concatenacionVarGlobales {
        if ($6.code == NULL) {
            sprintf(temp, "\n(setq %s (make-array %d))\n", $2.code, $4.value);
        } else {
            sprintf(temp, "\n(setq %s (make-array %d)) %s\n", $2.code, $4.value, $6.code
);
        }
        $$.code = gen_code(temp);
    }

    | ',' IDENTIF '=' expresion concatenacionVarGlobales {
        if ($5.code == NULL) {
            sprintf(temp, "\n(setq %s %s)\n", $2.code, $4.code);
        } else {
            sprintf(temp, "\n(setq %s %s) %s\n", $2.code, $4.code, $5.code);
        }
        $$.code = gen_code(temp);
    }
;

// ------------------------------------------------------------------------

// ----------------------------------FUNCIONES-------------------------------
```



```
printFunciones: { $$.code = NULL; }
    | funciones {
        printf("%s", $$.code);
    }
;

funciones: creacionFunciones {
        limpiar_tabla_local();
        sprintf(temp, "%s", $1.code);
        $$.code = gen_code(temp);
        strcpy(current_function, "");
    }

    | creacionFunciones funciones {
        limpiar_tabla_local();
        sprintf(temp, "%s\n%s", $1.code, $2.code);
        $$.code = gen_code(temp);
        strcpy(current_function, "");
    }
;

creacionFunciones: IDENTIF {

        strcpy(current_function, $1.code);
    } '(' parametros ')' '{' cuerpo '}' {
        if ($4.code == NULL) {
            sprintf(temp, "(defun %s ()\n\n%s\n\n)\n", $1.code, $7.code);
        } else {
            sprintf(temp, "(defun %s (%s)\n\n%s\n\n)\n", $1.code, $4.code, $7.code);
        }
        $$.code = gen_code(temp);
    }
;

parametros: { $$.code = NULL; }  // Lambda
    | INTEGER IDENTIF r_parametros {
        insertar_variable_local($2.code);
        char param_name[512];
        snprintf(param_name, sizeof(param_name), "%s_%s", current_function, $2.c
ode);

        if ($3.code == NULL) {
            $$.code = gen_code(param_name);
        } else {
            snprintf(temp, sizeof(temp), "%s %s", param_name, $3.code);
            $$.code = gen_code(temp);
        }
    }
;

r_parametros: { $$.code = NULL; }  // Lambda
    | ',' parametros { $$ = $2; }
;

// ------------------------------------------------------------------------
-

// ----------------------------------CUERPO----------------------------------
-

cuerpo: expresion ';' r_cuerpo {
```

```
        if ($3.code == NULL) {
            sprintf(temp, "%s", $1.code);
        } else {
            sprintf(temp, "%s%s", $1.code, $3.code);
        }
        $$.code = gen_code(temp);
    }  // Expresiones

    | sentencia r_cuerpo {
        if ($2.code == NULL) {
            sprintf(temp, "%s", $1.code);
        } else {
            sprintf(temp, "%s%s", $1.code, $2.code);
        }
        $$.code = gen_code(temp);
    }  // Sentencias

    | bucles r_cuerpo {
        if ($2.code == NULL) {
            sprintf(temp, "%s", $1.code);
        } else {
            sprintf(temp, "%s%s", $1.code, $2.code);
        }
        $$.code = gen_code(temp);
    }  // Bucles

    | if r_cuerpo {
        if ($2.code == NULL) {
            sprintf(temp, "%s", $1.code);
        } else {
            sprintf(temp, "%s%s", $1.code, $2.code);
        }
        $$.code = gen_code(temp);
    }  // If

    | locales ';' r_cuerpo {
        if ($3.code == NULL) {
            sprintf(temp, "%s", $1.code);
        } else {
            sprintf(temp, "%s%s", $1.code, $3.code);
        }
        $$.code = gen_code(temp);
    }  // Declaracion de variables
;

r_cuerpo: { $$.code = NULL; }  // Lambda
    | cuerpo { $$ = $1; }
;

// ------------------------------------------------------------------------------
-

// ---------------------------------SENTENCIA------------------------------------
-

sentencia: PRINTF '(' STRING printElem ')' ';' {
        sprintf(temp, "%s", $4.code);
        $$.code = gen_code(temp);
    }

    | PUTS '(' STRING ')' ';' {
        sprintf(temp, "\t(print \"%s\")\n", $3.code);
```



```
        $$.code = gen_code(temp);
    }

    | IDENTIF '=' expresion ';' {
        if (es_variable_local($1.code)) {
            sprintf(temp, "\t(setf %s_%s %s)\n", current_function, $1.code, $3.code);
  // Variable local
        } else {
            sprintf(temp, "\t(setf %s %s)\n", $1.code, $3.code);  // Variable global
        }
        $$.code = gen_code(temp);
    }

    | IDENTIF '[' expresion ']' '=' expresion ';' {
        if (es_variable_local($1.code)) {
            sprintf(temp, "\t(setf (aref %s_%s %s) %s)\n", current_function, $1.code, $3
.code, $6.code);  // Vector local
        } else {
            sprintf(temp, "\t(setf (aref %s %s) %s)\n", $1.code, $3.code, $6.code);  //
Vector global
        }
        $$.code = gen_code(temp);
    }

    | RETURN expresion ';' {
        sprintf(temp, "\t(return-from %s %s)", current_function, $2.code);
        $$.code = gen_code(temp);
    }
;

// ------------------------------------------------------------------------------

// ------------------------------------IF----------------------------------------

if: IF '(' expresion ')' '{' cuerpo '}' else {
        if ($8.code == NULL) {
            sprintf(temp, "(if %s\n\t(progn %s)\n)\n\n", $3.code, $6.code);
        } else {
            sprintf(temp, "(if %s\n\t(progn %s)\n%s\n)\n\n", $3.code, $6.code, $8.code);
        }
        $$.code = gen_code(temp);
    }

else: { $$.code = NULL; }  // Lambda
    | ELSE '{' cuerpo '}' {
        sprintf(temp, "\t(progn %s)\n", $3.code);
        $$.code = gen_code(temp);
    }
;

// ------------------------------------------------------------------------------

// ---------------------------------BUCLES---------------------------------------

bucles: WHILE '(' expresion ')' '{' cuerpo '}' {
        sprintf(temp, "(loop while %s do\n%s\n)\n\n", $3.code, $6.code);
        $$.code = gen_code(temp);
    }

    | FOR '(' creacionBucle ';' expresion ';' inicializadorDec ')' '{' cuerpo '}'
{
        sprintf(temp, "%s(loop while %s do\n%s\n%s\n)\n\n", $3.code, $5.code, $10.code,
```

```
$7.code);
        $$.code = gen_code(temp);
    }

creacionBucle: IDENTIF '=' NUMBER {
        if (es_variable_local($1.code)) {
            sprintf(temp, "\t(setq %s_%s %d)\n", current_function, $1.code, $3.value)
;
        } else {
            sprintf(temp, "\t(setq %s %d)\n", $1.code, $3.value);
        }
        $$.code = gen_code(temp);
    }  // Inicializa la variable del bucle

inicializadorDec: IDENTIF '=' IDENTIF opBucle {
        if (es_variable_local($1.code)) {
            if (es_variable_local($3.code)) {
                sprintf(temp, "\t(setf %s_%s (%s %s_%s %d))\n", current_function, $1.co
de, $4.op, current_function, $3.code, $4.value);
            } else {
                sprintf(temp, "\t(setf %s_%s (%s %s %d))\n", current_function, $1.code,
 $4.op, $3.code, $4.value);
            }
        } else {
            if (es_variable_local($3.code)) {
                sprintf(temp, "\t(setf %s (%s %s_%s %d))\n", $1.code, $4.op, current_fu
nction, $3.code, $4.value);
            } else {
                sprintf(temp, "\t(setf %s (%s %s %d))\n", $1.code, $4.op, $3.code, $4.v
alue);
            }
        }
        $$.code = gen_code(temp);
    }

opBucle:  '+' NUMBER { $$.op = "+"; $$.value = $2.value; }
        | '-' NUMBER { $$.op = "-"; $$.value = $2.value; }
        | '/' NUMBER { $$.op = "/"; $$.value = $2.value; }
        | '*' NUMBER { $$.op = "*"; $$.value = $2.value; }
;

// ------------------------------------------------------------------------------

// ---------------------------------printElem---------------------------------

printElem: ',' expresion r_elem {
        if ($3.code == NULL) {
            sprintf(temp, "\t(princ %s)\n", $2.code);
        } else {
            sprintf(temp, "\t(princ %s) %s\n", $2.code, $3.code);
        }
        $$.code = gen_code(temp);
    }

    | ',' STRING r_elem {
        if ($3.code == NULL) {
            sprintf(temp, "\t(princ \"%s\")\n", $2.code);
        } else {
            sprintf(temp, "\t(princ \"%s\") %s\n", $2.code, $3.code);
        }
        $$.code = gen_code(temp);
    }
```



```
;

r_elem: { $$.code = NULL; }  // Lambda
    | printElem { $$ = $1; }
;

// ------------------------------------------------------------------------------

// ---------------------DECLARACION VARIABLES LOCALES---------------------------
-

locales: creacionVarLocales r_varLocales {
    if ($2.code == NULL) {
        $$ = $1;
    } else {
        sprintf(temp, "%s %s", $1.code, $2.code);
        $$.code = gen_code(temp);
    }
}

creacionVarLocales: INTEGER IDENTIF asignacionVar concatenacionVarLocales {
        insertar_variable_local($2.code);  // AÃ±adimos la variable a la tabla l
ocal

        if ($3.code == NULL && $4.code == NULL) {
            sprintf(temp, "\t(setq %s_%s 0)\n", current_function, $2.code);
        } else if ($3.code == NULL) {
            sprintf(temp, "\t(setq %s_%s 0) %s\n", current_function, $2.code, $4.code
);
        } else if ($4.code == NULL) {
            sprintf(temp, "\t(setq %s_%s %s)\n", current_function, $2.code, $3.code);
        } else {
            sprintf(temp, "\t(setq %s_%s %s) %s\n", current_function, $2.code, $3.cod
e, $4.code);
        }
        $$.code = gen_code(temp);
    }
    | INTEGER IDENTIF '[' NUMBER ']' concatenacionVarLocales {
        insertar_variable_local($2.code);  // AÃ±adir la variable a la tabla loc
al
        if ($6.code == NULL) {
            sprintf(temp, "\t(setq %s_%s (make-array %d))\n", current_function, $2.code,
 $4.value);
        } else {
            sprintf(temp, "\t(setq %s_%s (make-array %d)) %s\n", current_function, $2.cod
e, $4.value, $6.code);
        }
        $$.code = gen_code(temp);
    }
;

concatenacionVarLocales: { $$.code = NULL; }  // Lambda
    | ',' IDENTIF concatenacionVarLocales {
        insertar_variable_local($2.code);  // AÃ±adimos esta variable a la tabla
 local
        if ($3.code == NULL) {
            sprintf(temp, "\n\t(setq %s_%s 0)\n", current_function, $2.code);
        } else {
            sprintf(temp, "\nt(setq %s_%s 0) %s\n", current_function, $2.code, $3.code
);
        }
```

```
        $$.code = gen_code(temp);
    }

    | ',' IDENTIF '[' NUMBER ']' concatenacionVarLocales {
        insertar_variable_local($2.code);
        if ($6.code == NULL) {
            sprintf(temp, "\n\t(setq %s_%s (make-array %d))\n", current_function, $2.code
, $4.value);
        } else {
            sprintf(temp, "\n\t(setq %s_%s (make-array %d)) %s\n", current_function, $2.c
ode, $4.value, $6.code);
        }
        $$.code = gen_code(temp);
    }

    | ',' IDENTIF '=' expresion concatenacionVarLocales {
        insertar_variable_local($2.code);
        if ($5.code == NULL) {
            sprintf(temp, "\n\t(setq %s_%s %s)\n", current_function, $2.code, $4.code
);
        } else {
            sprintf(temp, "\n\t(setq %s_%s %s) %s\n", current_function, $2.code, $4.co
de, $5.code);
        }
        $$.code = gen_code(temp);
    }

;

r_varLocales: { $$.code = NULL; }  // Lambda
    | locales { $$ = $1; }
;

// ------------------------------------------------------------------------------

// ---------------------------------EXPRESION---------------------------------

expresion: termino { $$ = $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 OR expresion {
        sprintf(temp, "(or %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 GREATEREQ expresion {
        sprintf(temp, "(>= %s %s)", $1.code, $3.code);
        $$.code = gen_code(temp);
    }
    | expresion LESSEQ expresion {
        sprintf(temp, "(<= %s %s)", $1.code, $3.code);
        $$.code = gen_code(temp);
    }
    | expresion EQUAL expresion {
        sprintf(temp, "(= %s %s)", $1.code, $3.code);
        $$.code = gen_code(temp);
    }
    | expresion DIST expresion {
        sprintf(temp, "(/= %s %s)", $1.code, $3.code);
        $$.code = gen_code(temp);
    }
    | expresion '%' expresion {
        sprintf(temp, "(mod %s %s)", $1.code, $3.code);
        $$.code = gen_code(temp);
    }
    | expresion '<' expresion {
        sprintf(temp, "(< %s %s)", $1.code, $3.code);
        $$.code = gen_code(temp);
    }
    | expresion '>' expresion {
        sprintf(temp, "(> %s %s)", $1.code, $3.code);
        $$.code = gen_code(temp);
    }
    | '!' expresion {
    sprintf(temp, "(not %s)", $2.code);
    $$.code = gen_code(temp);
    }
;

// ------------------------------------------------------------------------------
-

// ---------------------------------TERMINO---------------------------------
-

termino: operando { $$ = $1; }
    | '+' operando %prec UNARY_SIGN { $$ = $2; }
    | '-' operando %prec UNARY_SIGN {
        sprintf(temp, "(- %s)", $2.code);
        $$.code = gen_code(temp);
    }
;

// ------------------------------------------------------------------------------
-

// ---------------------------------OPERANDO---------------------------------
-

operando: IDENTIF {
        if (es_variable_local($1.code)) {
            sprintf(temp, "%s_%s", current_function, $1.code);  // Variable loca
l con prefijo
            $$.code = gen_code(temp);
        } else {
            $$.code = $1.code;
```

```
        }

    }

    | NUMBER {
        sprintf(temp, "%d", $1.value);
        $$.code = gen_code(temp);
    }
    | '(' expresion ')' { $$ = $2; }

    | IDENTIF '(' argumentos ')' {
        if ($3.code == NULL) {
            sprintf(temp, "\t(%s)\n", $1.code);
            $$.code = gen_code(temp);
        } else {
            sprintf(temp, "\t(%s %s)\n", $1.code, $3.code);
            $$.code = gen_code(temp);
        }
    }

    | IDENTIF '[' expresion ']' {
        if (es_variable_local($1.code)) {
            sprintf(temp, "(aref %s_%s %s)", current_function, $1.code, $3.code);
        } else {
            sprintf(temp, "(aref %s %s)", $1.code, $3.code);
        }
        $$.code = gen_code(temp);
    }
;

// ------------------------------------------------------------------------
---

// ---------------------------------ARGUMENTOS-----------------------------------
---

argumentos: { $$.code = NULL; }  // Lambda
    | expresion r_argumentos {
        if ($2.code == NULL) {
            $$ = $1;
        } else {
            sprintf(temp, "%s %s", $1.code, $2.code);
            $$.code = gen_code(temp);
        }
    }
;

r_argumentos: { $$.code = NULL; }  // Lambda
    | ',' expresion r_argumentos {
        if ($3.code == NULL) {
            $$ = $2;
        } else {
            sprintf(temp, "%s %s", $2.code, $3.code);
            $$.code = gen_code(temp);
        }
    }
;

// ------------------------------------------------------------------------
-

// SECCION 4    Codigo en C
```



```
%%

// Tabla de variables locales
char *tabla_local[100];
int num_locales = 0;

void insertar_variable_local(char *nombre) {
    tabla_local[num_locales++] = strdup(nombre);
}

int es_variable_local(char *nombre) {
    for (int i = 0; i < num_locales; i++) {
        if (strcmp(nombre, tabla_local[i]) == 0) return 1;
    }
    return 0;
}

void limpiar_tabla_local() {
    for (int i = 0; i < num_locales; i++) {
        free(tabla_local[i]);
    }
    num_locales = 0;
}

int n_line = 1;

int yyerror(mensaje)
char *mensaje;
{
    fprintf(stderr, "%s en la linea %d\n", mensaje, n_line);
    printf("\n");  // bye
}

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 queda memoria para %d bytes mas\n", nbytes);
        fprintf(stderr, "Reservados %ld bytes en %d llamadas\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,
    "||", OR,
    "&&", AND,
    "<=", LESSEQ,
    ">=", GREATEREQ,
    "!=", DIST,
    "==", EQUAL,
    "if", IF,
    "else", ELSE,
    "for", FOR,
    "return", RETURN,
    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() {
    // NO MODIFICAR ESTA FUNCION SIN PERMISO
    int i;
    unsigned char c;
    unsigned char cc;
    char ops_expandibles[] = "!<=|>%&/+-*";
    char temp_str[256];
    t_keyword *symbol;

    do {
        c = getchar();

        if (c == '#') {  // Ignora las lineas que empiezan por #  (#define, #inc
lude)
            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 coment
ario
            cc = getchar();
            if (cc != '/') {  // Si el siguiente char es /  es un comentario, pe
ro...
                ungetc(cc, stdin);
            } else {
                c = getchar();  // ...
                if (c == '@') {  // Si es la secuencia //@  ==> transcribimos l
a linea
                    do {         // Se trata de codigo inline (Codigo embebido e
n 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 == 10 || c == 13 || c == '\t');

    if (c == '\"') {
        i = 0;
        do {
            c = getchar();
            temp_str[i++] = c;
        } while (c != '\"' && i < 255);
        if (i == 256) {
            printf("AVISO: string con mas de 255 caracteres en linea %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(ops_expandibles, c) != NULL) {  // busca c en ops_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();
}
```