```
// 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

int yylex () ;
int yyerror () ;
char *mi_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
    t_node *node ; // - for possible future use of AST
} 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 Abucle main
%token SETQ
%token SETF
%token DEFUN
%token PRINT
%token PRINC
%token LOOP
%token DO
%token IF
%token ELSE
```



```
%token PROGN
%token AND OR NOT
%token DIST LE GE MOD


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

%left OR
%left AND       // Operadores lÃ³gicos
%left '=' DIST  // ==, != (deben estar despuÃ©s de <, >, etc.)
%left '<' '>' GE LE  // Comparaciones
%left '+' '-'      // Suma y resta
%left '*' '/' MOD // MultiplicaciÃ³n, divisiÃ³n y mÃ³dulo (alta prioridad)
%right NOT         // NegaciÃ³n (mayor prioridad)
%right UNARY_SIGN // Signo unario (-x, +x)

%%                            // Seccion 3 Gramatica - Semantico

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

axioma: creacionVar { ; } r_axioma { ; }
     | main  { ; } r_axioma { ; }
     | transcribir { ; } r_axioma { ; }
;

r_axioma: {;} //Lambda
     | axioma { $$ = $1; } //axioma
;

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

// ----------------------------TRANSCRIBIR------------------------------------
-

transcribir: '(' MAIN ')' {
        printf("main\n") ;
    }
;

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

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

main: '(' DEFUN MAIN '(' ')' cuerpo ')' {
    sprintf(temp, ": main\n%s\n;\n", $6.code) ;
    $$.code = gen_code(temp) ;
    printf("%s", $$.code) ;
}

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

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

cuerpo: sentencia cuerpo {
        sprintf(temp, "%s\n%s", $1.code, $2.code);  // Varias sentencias
        $$.code = gen_code(temp);}
    | sentencia { $$ = $1 ; }
```

```
;

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

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

sentencia: imprimir { $$ = $1; }
     | while { $$ = $1; }
     | if { $$ = $1; }
     | variables { $$ = $1; }
;

// ------------------------------------------------------------------------------
// ---------------------------------VARIABLES-----------------------------------

variables: '(' SETF IDENTIF expresion ')' {
        sprintf(temp, "%s %s !", $4.code, $3.code) ;
        $$.code = gen_code(temp) ;
    }
;

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

// --------------------CREACION VARIABLES GLOABLES---------------------------

creacionVar:  '(' SETQ IDENTIF expresion ')' {
        sprintf(temp, "variable %s\n%s %s !\n", $3.code, $4.code, $3.code) ;
        $$.code = gen_code(temp) ;
        printf("%s", $$.code) ;
    }
;

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

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

while: '(' LOOP WHILE expresion DO cuerpo ')' {
        sprintf(temp, "BEGIN\n\t%s\nWHILE\n\t%s\nREPEAT\n", $4.code, $6.code);
        $$.code = gen_code(temp);
    }
;

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

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


if: '(' IF expresion '(' PROGN cuerpo ')' ')' {
    sprintf(temp, "%s IF\n\t%s\nTHEN\n", $3.code, $6.code);  // Caso sin else
    $$.code = gen_code(temp);}

    | '(' IF expresion '(' PROGN cuerpo ')' '(' PROGN cuerpo ')' ')' {
        sprintf(temp, "%s IF\n\t%s\nELSE\n\t%s\nTHEN\n", $3.code, $6.code, $10.code);
 // Caso con else
        $$.code = gen_code(temp);
    }
;

// ------------------------------------------------------------------------------
```



```
// ---------------------------------PRINTS--------------------------------------

imprimir: '(' PRINT STRING ')' {
        sprintf(temp, ".\" %s\"", $3.code);
        $$.code = gen_code(temp);}
    | '(' PRINC expresion ')' {
        sprintf(temp, "%s .", $3.code);
        $$.code = gen_code(temp);
    }
    |'(' PRINC STRING ')' {
        sprintf(temp, ".\" %s\"", $3.code);
        $$.code = gen_code(temp);}
;

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

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


expresion: termino { $$ = $1; }

    |'+' expresion expresion {
        sprintf(temp, "%s %s +", $2.code, $3.code);
        $$.code = gen_code(temp);
    }
    | '-' expresion expresion {
        sprintf(temp, "%s %s -", $2.code, $3.code);
        $$.code = gen_code(temp);
    }
    | '*' expresion expresion {
        sprintf(temp, "%s %s *", $2.code, $3.code);
        $$.code = gen_code(temp);
    }
    | '/' expresion expresion {
        sprintf(temp, "%s %s /", $2.code, $3.code);
        $$.code = gen_code(temp);
    }
    | MOD expresion expresion {
        sprintf(temp, "%s %s mod", $2.code, $3.code);
        $$.code = gen_code(temp);
    }
    | AND expresion expresion {
        sprintf(temp, "%s %s and", $2.code, $3.code);
        $$.code = gen_code(temp);
    }
    | OR expresion expresion {
        sprintf(temp, "%s %s or", $2.code, $3.code);
        $$.code = gen_code(temp);
    }
    | NOT expresion {
        sprintf(temp, "%s 0=", $2.code);
        $$.code = gen_code(temp);
    }
    | '<' expresion expresion {
        sprintf(temp, "%s %s <", $2.code, $3.code);
        $$.code = gen_code(temp);
    }
    | LE expresion expresion {
        sprintf(temp, "%s %s <=", $2.code, $3.code);
        $$.code = gen_code(temp);
    }
    | '>' expresion expresion {
```

```
        sprintf(temp, "%s %s >", $2.code, $3.code);
        $$.code = gen_code(temp);
    }
    | GE expresion expresion {
        sprintf(temp, "%s %s >=", $2.code, $3.code);
        $$.code = gen_code(temp);
    }
    | '=' expresion expresion {
        sprintf(temp, "%s %s =", $2.code, $3.code);
        $$.code = gen_code(temp);
    }
    | DIST expresion expresion {
        sprintf(temp, "%s %s = 0=", $2.code, $3.code);
        $$.code = gen_code(temp);
    }
;

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

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

termino: operando { $$ = $1; }
    | '(' '-' operando ')' %prec UNARY_SIGN {
        sprintf(temp, "%s NEGATE", $3.code);
        $$.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 ;

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,
    "setq",         SETQ,
    "setf",         SETF,
    "defun",        DEFUN,
    "print",        PRINT,
    "princ",        PRINC,
    "while",        WHILE,
    "loop",         LOOP,
    "do",           DO,
    "if",           IF,
    "else",         ELSE,
    "progn",        PROGN,
    "/=",           DIST,
    ">=",           GE,
    "<=",           LE,
    "and",          AND,            // Operador lÃ³gico AND
    "or",           OR,             // Operador lÃ³gico OR
    "not",          NOT,
    "mod",          MOD,
    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, #incl
ude)
            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 comenta
rio
            cc = getchar () ;
            if (cc != '/') {   // Si el siguiente char es /  es un comentario, p
```



```
ero...
                ungetc (cc, stdin) ;
            } else {
                c = getchar () ;        // ...
                if (c == '@') { // Si es la secuencia //@  ==> transcribimos la
linea
                    do {                // Se trata de codigo inline (Codigo emb
ebido 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 == 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 ant
es vrariabre
//              printf ("\nDEV: IDENTIF %s\n", yylval.code) ;    // PARA DEPURA
R
```

```
            return (IDENTIF) ;
        } else {
//                printf ("\nDEV: OTRO %s\n", yylval.code) ;       // PARA DEPURA
R
            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 () ;
}
```