```
/*
Grupo 1: Mario Resino Solis, Rafael Zhu Zhou
100428975@alumnos.uc3m.es 100429078@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

char *mi_malloc (int) ;
char *genera_cadena (char *) ;
int yylex () ;
int yyerror () ;

char temp [2048] ;
char nombres_funciones[128];
```

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

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



%}

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

```
%token <valor> NUMERO         // Todos los token tienen un tipo para la pila
%token <cadena> IDENTIF       // Identificador=variable
%token <cadena> INTEGER       // identifica la definicion de un entero
%token <cadena> STRING
%token <cadena> MAIN          // identifica el comienzo del proc. main
%token <cadena> WHILE         // identifica el bucle main
%token <cadena> IF
%token <cadena> ELSE
%token <cadena> FOR
%token <cadena> RETURN

%token <cadena> PUTS
%token <cadena> PRINTF
%token <cadena> AND
%token <cadena> OR
%token <cadena> NOTEQ
%token <cadena> EQ
%token <cadena> LESSEQ
%token <cadena> GREATEREQ

%type <cadena> axioma decl_variables decl_funciones
```

```
%type <cadena> declaracion_global rest_decl_global

%type <cadena> funciones func_main cuerpo_funciones argumentos rest_argumentos rest_arg_opc sentencia_func

%type <cadena> llamada_func parametros rest_parametros

%type <cadena> cuerpo sentencia

%type <cadena> bucles_for inic_for

%type <cadena> condicionales cuerpo_cond

%type <cadena> bucles

%type <cadena> impresion rest_impresion

%type <cadena> asignacion

%type <cadena> declaracion_local rest_decl_local

%type <cadena> expresion

%type <cadena> termino operando


%right '='                    // es la ultima operacion que se debe realizar

%left OR

%left AND

%left NOTEQ EQ

%left '<' '>' LESSEQ GREATEREQ

%left '+' '-'                 // menor orden de precedencia

%left '*' '/' '%'                // orden de precedencia intermedio

%left SIGNO_UNARIO            // mayor orden de precedencia
```

```
%%
                          // Seccion 3 Gramatica - Semantico
axioma:                        decl_variables decl_funciones                {;}
                          ;

// -------------------------- ESTRUCTURA CODIGO -------------------------

decl_variables:                declaracion_global ';' decl_variables        {;}
                          |                                                 {;}
                          ;

decl_funciones:                funciones func_main                          {sprintf(temp, "%s%s", $1,$2);
                                                                            $$ = genera_cadena(temp);}
                          |                                                 {$$="";}
                          ;
// -------------------------- DECLARACION GLOBAL -------------------------

declaracion_global:            INTEGER IDENTIF rest_decl_global             {sprintf(temp, "(setq %s 0)%s", $2,$3);
                                                                            printf("%s",temp);}
                          |    INTEGER IDENTIF '=' NUMERO rest_decl_global   {sprintf(temp, "(setq %s %d)%s", $2,$4,$5);
```

```
                                                                    printf("%s",temp);}
                          |     INTEGER IDENTIF '[' NUMERO ']'
                                      rest_decl_global              {sprintf(temp,
                                                                    "(setq %s (make-array %d))%s", $2,$4,$6);
                                                                    printf("%s",temp);}
                          ;

rest_decl_global:               ',' IDENTIF rest_decl_global        {sprintf(temp, "(setq %s 0)%s", $2,$3);
                                                                    $$ = genera_cadena(temp);}
                          |     ',' IDENTIF '=' NUMERO rest_decl_global  {sprintf(temp, "(setq %s %d)%s", $2,$4,$5);
                                                                    $$ = genera_cadena(temp);}
                          |     ',' IDENTIF '[' NUMERO ']' rest_decl_global  {sprintf(temp, "(setq %s (make-array %d))
                                                                    %s", $2,$4,$6);
                                                                    $$ = genera_cadena(temp);}
                          |                                         {$$="";}
                          ;
// -------------------------- FUNCIONES --------------------------

func_main:                      MAIN                                {sprintf(nombres_funciones,"%s",$1);}
                                      '(')''{' cuerpo_funciones '}'  {sprintf(temp, "(defun main () %s)", $6);
                                                                    printf("%s",temp);}
```

```
                          ;

funciones:                IDENTIF                          {sprintf(nombres_funciones,"%s",$1);}
                                '('argumentos')''{'
                                cuerpo_funciones '}'       {sprintf(temp, "(defun %s (%s) %s)",$1,$4,$7);
                                                           printf("%s",temp);}
                          funciones                        {;}
                        |                                  {$$="";}
                        ;

cuerpo_funciones:         sentencia_func cuerpo_funciones  {sprintf(temp, "%s %s", $1,$2);
                                                           $$ = genera_cadena(temp);}
                        |  RETURN expresion ';'            {$$=$2;}
                        |                                  {$$="";}
                        ;

argumentos:               rest_argumentos                  {$$=$1;}
                        |                                  {$$="";}
                        ;

rest_argumentos:          INTEGER IDENTIF                  {$$=genera_cadena($2);}
```

```
                        |     INTEGER IDENTIF ',' rest_argumentos          {sprintf(temp,"%s %s",$2,$4);
                                                                           $$=genera_cadena(temp);}
                        |     rest_arg_opc                                {sprintf(temp,"&optional %s",$1);
                                                                           $$ = genera_cadena(temp);}
                        ;

rest_arg_opc:                 INTEGER IDENTIF '=' NUMERO                 {sprintf(temp,"(%s %d)",$2,$4);
                                                                           $$=genera_cadena(temp);}
                        |     INTEGER IDENTIF '=' NUMERO ',' rest_arg_opc {sprintf(temp,"(%s %d)%s",$2,$4,$6);
                                                                           $$=genera_cadena(temp);}
                        ;

sentencia_func:               asignacion ';'                             {$$=$1;}
                        |     impresion ';'                              {$$=$1;}
                        |     declaracion_local ';'                      {$$=$1;}
                        |     bucles                                     {$$=$1;}
                        |     condicionales                              {$$=$1;}
                        |     bucles_for                                 {$$=$1;}
                        |     llamada_func ';'                           {$$=$1;}
                        |     RETURN expresion ';' sentencia_func        {sprintf(temp, "(return-from %s %s) %s",
                                                                         nombres_funciones, $2,$4);
```

```
                                                                    $$ = genera_cadena(temp);}
                        ;

// ------------------------- LLAMADAS FUNC -------------------------

llamada_func:                   IDENTIF '(' parametros ')'          {sprintf(temp,"(%s %s)",$1,$3);
                                                                    $$=genera_cadena(temp);}
                        ;

parametros:                     rest_parametros                     {$$=$1;}
                        |                                           {$$="";}
                        ;

rest_parametros:                expresion                           {$$=$1;}
                        |       expresion ',' rest_parametros       {sprintf(temp,"%s %s",$1,$3);
                                                                    $$=genera_cadena(temp);}
                        ;

// ------------------------- SENTENCIAS/CODIGO -------------------------

cuerpo:                         sentencia cuerpo                    {sprintf(temp, "%s %s", $1,$2);
                                                                    $$ = genera_cadena(temp);}
```

```
                        |                                                   {$$="";}
                        ;

sentencia:                  asignacion ';'                                  {$$=$1;}
                        |   impresion ';'                                   {$$=$1;}
                        |   declaracion_local ';'                           {$$=$1;}
                        |   bucles                                          {$$=$1;}
                        |   condicionales                                   {$$=$1;}
                        |   bucles_for                                      {$$=$1;}
                        |   llamada_func ';'                                {$$=$1;}
                        |   RETURN expresion ';'                            {sprintf(temp, "(return-from %s %s)",
                                                                            nombres_funciones, $2);
                                                                            $$ = genera_cadena(temp);}
                        ;

// ------------------------- BUCLES FOR  -------------------------

bucles_for:                 FOR '(' inic_for ';' expresion ';'
                                asignacion ')''{' cuerpo '}'                {sprintf(temp, "%s (loop while %s do
                                                                            %s%s)",$3,$5,$10,$7);
                                                                            $$ = genera_cadena(temp);}
                        ;
```

```
inic_for:                       asignacion
                          |     declaracion_local
                          ;


//  ------------------------- IF-THEN-ELSE  -------------------------


condicionales:                  IF '('expresion')''{' cuerpo_cond '}'        {sprintf(temp, "(if %s %s)",$3,$6);
                                                                             $$ = genera_cadena(temp);}
                          |     IF '('expresion')''{' cuerpo_cond '}' ELSE
                                       '{' cuerpo_cond '}'                   {sprintf(temp, "(if %s %s %s)",$3,$6,$10);
                                                                             $$ = genera_cadena(temp);}
                          ;


cuerpo_cond:                    sentencia                                    {$$=$1;}
                          |     sentencia sentencia cuerpo                   {sprintf(temp, "(progn %s %s %s)",$1,$2,$3);
                                                                             $$ = genera_cadena(temp);}
                          |                                                  {$$="";}
                          ;


//  ------------------------- BUCLES WHILE -------------------------
```

```
bucles:                      WHILE '('expresion')''{' cuerpo '}'          {sprintf(temp, "(loop while %s do %s)",$3,$6);
                                                                          $$ = genera_cadena(temp);}
                          ;

//  ------------------------- PRINTS/PUTS  -------------------------

impresion:                    PRINTF '(' STRING rest_impresion ')'        {$$=$4;}
                          |   PUTS '(' STRING ')'                         {sprintf(temp, "(print \"%s\")", $3);
                                                                          $$ = genera_cadena(temp);}
                          ;

rest_impresion:               ',' expresion rest_impresion                {sprintf(temp, "(print %s)%s", $2,$3);
                                                                          $$ = genera_cadena(temp);}
                          |                                               {$$="";}
                          ;

//  ------------------------- ASIGNACIONES  -------------------------

asignacion:                   IDENTIF '=' expresion                       {sprintf(temp, "(setq %s %s)", $1,$3);
                                                                          $$ = genera_cadena(temp);}
                          |   IDENTIF '[' expresion ']''=' expresion      {sprintf(temp, "(setf (aref %s %s) %s)",
                                                                          $1,$3, $6);
                                                                          $$ = genera_cadena(temp);}
```

```
                        |     IDENTIF '=' expresion '?'
                              operando ':' operando                {sprintf(temp, "(setq %s (if %s %s %s))",
                                                                   $1, $3, $5, $7);
                                                                   $$ = genera_cadena(temp);}

                        |     IDENTIF '[' expresion ']' '=' expresion
                              '?'    operando ':' operando         {sprintf(temp,
                                                                   "(setf (aref %s %s) (if %s %s %s))",
                                                                   $1, $3, $6, $8, $10);
                                                                   $$ = genera_cadena(temp);}

                        ;


//  ------------------------- DECLARACION LOCAL  -------------------------


declaracion_local:            INTEGER IDENTIF rest_decl_local      {sprintf(temp, "(setq %s 0)%s", $2,$3);
                                                                   $$ = genera_cadena(temp);}

                        |     INTEGER IDENTIF '=' expresion
                              rest_decl_local                      {sprintf(temp, "(setq %s %s)%s", $2,$4,$5);
                                                                   $$ = genera_cadena(temp);}

                        |     INTEGER IDENTIF '[' NUMERO ']'
                              rest_decl_local                      {sprintf(temp, "(setq %s (make-array %d))%s"
                                                                   ,$2,$4,$6);
                                                                   $$ = genera_cadena(temp);}

                        ;


rest_decl_local:              ',' IDENTIF '=' expresion rest_decl_local    {sprintf(temp, "(setq %s %s)%s", $2,$4,$5);
                                                                   $$ = genera_cadena(temp);}
```

```
                    |     ',' IDENTIF rest_decl_local                       {sprintf(temp, "(setq %s 0)%s", $2,$3);
                                                                            $$ = genera_cadena(temp);}

                    |     ',' IDENTIF '[' NUMERO ']' rest_decl_local       {sprintf(temp, "(setq %s (make-array %s))%s",
                                                                            $2,$4,$6);
                                                                            $$ = genera_cadena(temp);}

                    |                                                     {$$="";}

                    ;


//  -------------------------- EXPR, OP.LOGICOS Y COMPARADORES  --------------------------


expresion:          termino                                               {$$=$1;}

                    |     expresion '+' expresion                         {sprintf(temp, "(+ %s %s)", $1,$3);
                                                                            $$ = genera_cadena(temp);}

                    |     expresion '-' expresion                         {sprintf(temp, "(- %s %s)", $1,$3);
                                                                            $$ = genera_cadena(temp);}

                    |     expresion '*' expresion                         {sprintf(temp, "(* %s %s)", $1,$3);
                                                                            $$ = genera_cadena(temp);}

                    |     expresion '/' expresion                         {sprintf(temp, "(/ %s %s)", $1,$3);
                                                                            $$ = genera_cadena(temp);}

                    |     expresion AND expresion                         {sprintf(temp, "(And %s %s)", $1,$3);
                                                                            $$ = genera_cadena(temp);}

                    |     expresion OR expresion                          {sprintf(temp, "(or %s %s)", $1,$3);
                                                                            $$ = genera_cadena(temp);}
```

```
                    |     expresion NOTEQ expresion                    {sprintf(temp, "(/= %s %s)", $1,$3);
                                                                       $$ = genera_cadena(temp);}

                    |     expresion EQ expresion                      {sprintf(temp, "(= %s %s)", $1,$3);
                                                                       $$ = genera_cadena(temp);}

                    |     expresion '<' expresion                     {sprintf(temp, "(< %s %s)", $1,$3);
                                                                       $$ = genera_cadena(temp);}

                    |     expresion '>' expresion                     {sprintf(temp, "(> %s %s)", $1,$3);
                                                                       $$ = genera_cadena(temp);}

                    |     expresion LESSEQ expresion                  {sprintf(temp, "(<= %s %s)", $1,$3);
                                                                       $$ = genera_cadena(temp);}

                    |     expresion GREATEREQ expresion               {sprintf(temp, "(>= %s %s)", $1,$3);
                                                                       $$ = genera_cadena(temp);}

                    |     expresion '%' expresion                     {sprintf(temp, "(mod %s %s)", $1,$3);
                                                                       $$ = genera_cadena(temp);}

                          ;


//  ------------------------- TERMINOS  -------------------------


termino:               operando                                      {$$=$1;}

                    |  '+' operando %prec SIGNO_UNARIO               {sprintf(temp, "(+%s)", $2);
                                                                       $$ = genera_cadena(temp);}

                    |  '-' operando %prec SIGNO_UNARIO               {sprintf(temp, "(-%s)", $2);
                                                                       $$ = genera_cadena(temp);}

                    ;
```

```
operando:               IDENTIF                                          {$$ = genera_cadena($1);}
                      |     IDENTIF'['expresion']'                      { sprintf(temp, "(aref %s %s)",$1, $3);
                                                                       $$ = genera_cadena(temp);}
                      |   NUMERO                                         {$$ = int_to_string($1); }
                      |   '(' expresion ')'                              {$$ = $2;}
                      |     llamada_func                                 {$$ = $1;}
            ;


%%
                           // SECCION 4    Codigo en C
int n_linea = 1 ;

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

char *mi_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_pal_reservadas { // para las palabras reservadas de C
    char *nombre ;
    int token ;
} t_reservada ;

t_reservada pal_reservadas [] = { // define las palabras reservadas y los
    "main",        MAIN,           // y los token asociados
    "int",         INTEGER,
    "while",    WHILE,
    "puts",            PUTS,
    "printf",   PRINTF,
    "&&",              AND,
    "||",              OR,
    "!=",              NOTEQ,
    "==",              EQ,
    "<=",              LESSEQ,
    ">=",              GREATEREQ,
    "if",              IF,
    "else",            ELSE,
    "for",             FOR,
```

```
    "return",   RETURN,

    NULL,          0               // para marcar el fin de la tabla
} ;

t_reservada *busca_pal_reservada (char *nombre_simbolo)
{                                   // Busca n_s en la tabla de pal. res.
                                    // y devuelve puntero a registro (simbolo)
    int i ;
    t_reservada *sim ;

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

```
    return NULL ;
}


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

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

    l = strlen (nombre)+1 ;
    p = (char *) mi_malloc (l) ;
    strcpy (p, nombre) ;

    return p ;
}
```

```
int yylex ()
{
    int i ;
    unsigned char c ;
    unsigned char cc ;
    char ops_expandibles [] = "!<=>|&%+-/*" ;
    char cadena [256] ;
    t_reservada *simbolo ;

    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_linea++ ;

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

```
    if (c == '\"') {
        i = 0 ;
        do {
            c = getchar () ;
            cadena [i++] = c ;
        } while (c != '\"' && i < 255) ;
        if (i == 256) {
            printf ("AVISO: string con mas de 255 caracteres en linea %d\n", n_linea) ;
        }                 // habria que leer hasta el siguiente " , pero, y si falta?
        cadena [--i] = '\0' ;
        yylval.cadena = genera_cadena (cadena) ;
        return (STRING) ;
    }

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

```
    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) {
            cadena [i++] = tolower (c) ;
            c = getchar () ;
        }
        cadena [i] = '\0' ;
        ungetc (c, stdin) ;

        yylval.cadena = genera_cadena (cadena) ;
        simbolo = busca_pal_reservada (yylval.cadena) ;
        if (simbolo == NULL) {    // no es palabra reservada -> identificador
//              printf ("\nDEV: IDENTIF %s\n", yylval.cadena) ;    // PARA DEPURAR
            return (IDENTIF) ;
        } else {
//              printf ("\nDEV: OTRO %s\n", yylval.cadena) ;       // PARA DEPURAR
            return (simbolo->token) ;
        }
    }
```

```
    if (strchr (ops_expandibles, c) != NULL) { // busca c en ops_expandibles
        cc = getchar () ;
        sprintf (cadena, "%c%c", (char) c, (char) cc) ;
        simbolo = busca_pal_reservada (cadena) ;
        if (simbolo == NULL) {
            ungetc (cc, stdin) ;
            yylval.cadena = NULL ;
            return (c) ;
        } else {
            yylval.cadena = genera_cadena (cadena) ; // aunque no se use
            return (simbolo->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 () ;
}
```