```
/*
MIGUEL JIMENO CASAS - HECTOR HERRAIZ DIEZ - GRUPO 211
 100495932@alumnos.uc3m.es - 100499734@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] ;
char *current_function = NULL;

// Tabla de variables locales
typedef struct local_var {
   char *name;
   struct local_var *next;
} t_local_var;

t_local_var *local_vars = NULL;

void add_local_var(char *name) {
   t_local_var *new_var = (t_local_var *)mi_malloc(sizeof(t_local_var));
   new_var->name = gen_code(name);
   new_var->next = local_vars;
   local_vars = new_var;
}
```

```
int is_local_var(char *name) {
    t_local_var *current = local_vars;
    while (current != NULL) {
        if (strcmp(current->name, name) == 0) {
            return 1;
        }
        current = current->next;
    }
    return 0;
}

void clear_local_vars() {
    local_vars = NULL;
}

// 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 bucle main
%token IF           // identifica la sentencia if
%token ELSE         // identifica la sentencia else
%token FOR
%token PUTS         // identifica la sentencia puts
%token PRINTF       // identifica la sentencia printf
%token RETURN
%token AND OR
%token IGUAL DISTINTO MENORIGUAL MAYORIGUAL
```

```
%right '='                  // asignación
%left OR                    // or lógico
%left AND                   // and lógico
%right NOT                  // not lógico
%left IGUAL DISTINTO        // comparación de igualdad
%left '<' '>' MENORIGUAL MAYORIGUAL  // comparación numérica
%left '+' '-'               // suma y resta
%left '*' '/'               // multiplicación y división
%left '%'                   // mod
%left UNARY_SIGN            // signo unario


%%                  // Seccion 3 Gramatica - Semantico

programa:
        declaracion_variables_globales funciones { printf("%s\n%s\n", $1.code, $2.code); }
        ;
```

```
declaracion_variables_globales:
        /* vacío */ { $$.code = gen_code(""); }
        | declaracion_variables_globales declaracion_variable_global ';' {
                                        sprintf(temp, "%s\n%s", $1.code, $2.code);
                                        $$.code = gen_code(temp);
                                      }
        ;

declaracion_variable_global:
        INTEGER lista_declaracion_global { $$.code = $2.code; }
        ;

lista_declaracion_global:
        IDENTIF { sprintf(temp, "(setq %s 0)", $1.code);
                        $$.code = gen_code(temp);
                        }
        | IDENTIF '=' NUMBER { sprintf(temp, "(setq %s %d)",
                        $1.code, $3.value);
```

```
                    $$.code = gen_code(temp);
                    }
        | IDENTIF '[' NUMBER ']' {  sprintf(temp, "(setq %s (make-array %d))", $1.code, $3.value);
                    $$.code = gen_code(temp);
                    }
        | lista_declaracion_global ',' IDENTIF { sprintf(temp, "%s\n(setq %s 0)", $1.code, $3.code);
                        $$.code = gen_code(temp);
                        }
        | lista_declaracion_global ',' IDENTIF '=' NUMBER { sprintf(temp, "%s\n(setq %s %d)", $1.code, $3.code, $5.value);
                            $$.code = gen_code(temp);
                            }
        | lista_declaracion_global ',' IDENTIF '[' NUMBER ']' {  sprintf(temp, "(setq %s (make-array %d))", $1.code, $3.value);
                                $$.code = gen_code(temp);
                                }
        ;


funciones:
        funcion_main            { $$ = $1; }
        | funcion funciones      {
```

```
                    sprintf(temp, "%s\n%s", $1.code, $2.code);
                    $$.code = gen_code(temp);
                    }
        ;

funcion_main:
        inicializacion_main '{' instrucciones '}' {
                      sprintf(temp, "(%s %s\n)", $1.code, $3.code);
                      $$.code = gen_code(temp);
                      clear_local_vars();
                    }
        ;

inicializacion_main:
          MAIN '(' ')' {
                  current_function = strdup("main");
                  sprintf(temp, "defun main ()\n");
                  $$.code = gen_code(temp);
          }
```

```
funcion:
        inicializacion_funcion '{' instrucciones '}' {
                                        // Verificar si la última instrucción es un return
                                        char *last_instr = strrchr($3.code, '\n');
                                        if (last_instr != NULL) last_instr++;
                                        else last_instr = $3.code;

                                        if (strstr(last_instr, "return") != NULL) {
                                          // Si termina con return, usamos el valor directamente
                                          sprintf(temp, "(%s %s)", $1.code, $3.code);
                                        } else {
                                          // Si no termina con return, agregamos nil al final
                                          sprintf(temp, "(%s %s\n)", $1.code, $3.code);
                                            }
                                        $$.code = gen_code(temp);
                                        free(current_function);
                                        }
```

```
        ;

inicializacion_funcion:
          IDENTIF '(' parametros ')' {
                    current_function = $1.code;
                    sprintf(temp, "defun %s (%s)\n", $1.code, $3.code);
                    $$.code = gen_code(temp);
          }

parametros:
        /* vacío */ { $$.code = gen_code(""); }
        | lista_parametros { $$.code = $1.code; }
        ;

lista_parametros:
        parametro { $$.code = $1.code; }
        | lista_parametros ',' parametro { sprintf(temp, "%s %s", $1.code, $3.code);
                    $$.code = gen_code(temp);
                    }
```

```
        ;

parametro:
        INTEGER IDENTIF {
                sprintf(temp, "%s", $2.code);
                $$.code = gen_code(temp);
                }
        ;



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

instruccion:
```

```
        sentencia_simple ';'   { $$ = $1; }
        | sentencia_bloque      { $$ = $1; }
        | declaracion_variable ';' { $$ = $1; }
        ;


sentencia_simple:
        IDENTIF '=' expresion {
                if(is_local_var($1.code)) {
                   sprintf(temp, "(setf %s_%s %s)", current_function, $1.code, $3.code);
                } else {
                   sprintf(temp, "(setf %s %s)", $1.code, $3.code);
                }
                $$.code = gen_code(temp);
                }
        | '@' expresion                 { sprintf(temp, "(print %s)", $2.code);
                        $$.code = gen_code(temp);
                        }
        | PUTS '(' STRING ')'           { sprintf(temp, "(print \"%s\")", $3.code);
```

```
                    $$.code = gen_code(temp);
                    }
| PRINTF '(' STRING ',' lista_elementos ')' {
                        $$.code = $5.code;
                        }
|RETURN expresion {
        if (current_function != NULL) {
          if (is_local_var($2.code)) {
            sprintf(temp, "(return-from %s %s_%s)", current_function, current_function, $2.code);
          } else {
            sprintf(temp, "(return-from %s %s)", current_function, $2.code);
          }
        }
        $$.code = gen_code(temp);
        }
| IDENTIF '(' argumentos ')' {  // Llamada a función como sentencia
            sprintf(temp, "(%s %s)", $1.code, $3.code);
            $$.code = gen_code(temp);
            }
```

```
        | IDENTIF '[' expresion ']' '=' expresion {
                                if(is_local_var($1.code)) {
                                   sprintf(temp, "(setf (aref %s_%s %s) %s)", current_function, $1.code, $3.code, $6.code);
                                } else {
                                   sprintf(temp, "(setf (aref %s %s) %s)", $1.code, $3.code, $6.code);
                                }
                                $$.code = gen_code(temp);
                                }
        ;

sentencia_bloque:
        if_sin_else

        | if_con_else

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

```
        | FOR '(' inicializacion ';' expresion ';' incremento ')' '{' instrucciones '}' {sprintf(temp, "%s\n(loop while %s do\n%s\n%s)",
                                                                $3.code, $5.code, $10.code, $7.code);
                                                                $$.code = gen_code(temp);
                                                              }
        ;



if_sin_else:
        IF '(' expresion ')' '{' instrucciones '}' {
                                        if (strchr($6.code, '\n') != NULL) {
                                          sprintf(temp, "(if %s\n(progn %s))", $3.code, $6.code);
                                        } else {
                                          sprintf(temp, "(if %s\n%s)", $3.code, $6.code);
                                        }
                                        $$.code = gen_code(temp);
                                        }
        ;


if_con_else:
```

```
        IF '(' expresion ')' '{' instrucciones '}' ELSE '{' instrucciones '}' {
                                                    if (strchr($6.code, '\n') != NULL && strchr($10.code, '\n') != NULL) {
                                                    sprintf(temp, "(if %s\n(progn\n%s)\n(progn\n%s))", $3.code, $6.code, $10.code);
                                                    } else if (strchr($6.code, '\n') != NULL) {
                                                    sprintf(temp, "(if %s\n(progn\n%s)\n%s)", $3.code, $6.code, $10.code);
                                                    } else if (strchr($10.code, '\n') != NULL) {
                                                    sprintf(temp, "(if %s\n%s\n(progn\n%s))", $3.code, $6.code, $10.code);
                                                    } else {
                                                    sprintf(temp, "(if %s\n%s\n%s\n)", $3.code, $6.code, $10.code);
                                                    }
                                                    $$.code = gen_code(temp);
                                                    }
        ;

inicializacion:

        INTEGER IDENTIF '=' operando {sprintf(temp, "(setq %s_%s %s)", current_function, $1.code, $3.code);
                $$.code = gen_code(temp);
                }
```

```
        | IDENTIF '=' operando {sprintf(temp, "(setf %s_%s %s)",current_function, $1.code,  $3.code);
                        $$.code = gen_code(temp);
                        }
        ;


incremento:

        IDENTIF '=' expresion {sprintf(temp, "(setf %s_%s %s)",  current_function, $1.code, $3.code);
                         $$.code = gen_code(temp);
                         }
        ;

declaracion_variable:
        INTEGER lista_declaracion   { $$.code = $2.code; }
        ;

lista_declaracion:
        IDENTIF {add_local_var($1.code);
```

```
        sprintf(temp, "(setq %s_%s 0)", current_function, $1.code);
        $$.code = gen_code(temp);
        }
| IDENTIF '=' NUMBER {add_local_var($1.code);
                    sprintf(temp, "(setq %s_%s %d)", current_function, $1.code, $3.value);
                    $$.code = gen_code(temp);
                    }
| IDENTIF '[' NUMBER ']' {add_local_var($1.code);
                      sprintf(temp, "(setq %s_%s (make-array %d))",current_function, $1.code, $3.value);
                      $$.code = gen_code(temp);
                      }
| lista_declaracion ',' IDENTIF {add_local_var($3.code);
                        sprintf(temp, "%s\n(setq %s_%s 0)", $1.code, current_function, $3.code);
                        $$.code = gen_code(temp);
                        }
| lista_declaracion ',' IDENTIF '=' NUMBER {add_local_var($3.code);
                                sprintf(temp, "%s\n(setq %s_%s %d)", $1.code, current_function, $3.code, $5.value);
                                $$.code = gen_code(temp);
                                }
```

```
        | lista_declaracion ',' IDENTIF '[' NUMBER ']' {add_local_var($3.code);
                                        sprintf(temp, "(setq %s_%s (make-array %d))", current_function,$1.code, $3.value);
                                        $$.code = gen_code(temp);
                                        }
        ;


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 '>' 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 IGUAL expresion     { sprintf(temp, "(= %s %s)", $1.code, $3.code);
                    $$.code = gen_code(temp);
                    }
| expresion DISTINTO expresion   { sprintf(temp, "(/= %s %s)", $1.code, $3.code);
                    $$.code = gen_code(temp);
                    }
| expresion MENORIGUAL expresion { sprintf(temp, "(<= %s %s)", $1.code, $3.code);
                    $$.code = gen_code(temp);
                    }
| expresion MAYORIGUAL expresion { sprintf(temp, "(>= %s %s)", $1.code, $3.code);
                    $$.code = gen_code(temp);
                    }
```

```
        | expresion AND expresion    { sprintf(temp, "(and %s %s)", $1.code, $3.code);
                                     $$.code = gen_code(temp);
                                     }
        | expresion OR expresion     { sprintf(temp, "(or %s %s)", $1.code, $3.code);
                                     $$.code = gen_code(temp);
                                     }
        | '!' expresion %prec UNARY_SIGN { sprintf(temp, "(not %s)", $2.code);
                                        $$.code = gen_code(temp);
                                        }
        | expresion '%' expresion   { sprintf(temp, "(mod %s %s)", $1.code, $3.code);
                                    $$.code = gen_code(temp);
                                     }
        | IDENTIF '(' argumentos ')'{ sprintf(temp, "(%s %s)", $1.code, $3.code);
                                     $$.code = gen_code(temp);
                                     }
        ;

termino:
        operando             { $$ = $1; }
```

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


operando:
        IDENTIF {
                if(current_function != NULL && is_local_var($1.code)) {
                  // Variable local o parámetro
                  sprintf(temp, "%s_%s", current_function, $1.code);
                } else {
                  // Variable global
                  sprintf(temp, "%s", $1.code);
                }
                $$.code = gen_code(temp);
              }
        | NUMBER    { sprintf(temp, "%d", $1.value);
                      $$.code = gen_code(temp);
```

```
            }
        | '(' expresion ')'         { $$ = $2; }
        | IDENTIF '[' expresion ']' {
                        if(current_function != NULL && is_local_var($1.code)) {
                          // Variable local o parámetro
                          sprintf(temp, "(aref %s_%s %s)", current_function, $1.code, $3.code);
                        } else {
                          // Variable global
                          sprintf(temp, "(aref %s %s)", $1.code, $3.code);
                        }
                        $$.code = gen_code(temp);
                      }
        ;

argumentos:
        /* vacío */ { $$.code = gen_code(""); }
        | lista_argumentos { $$.code = $1.code; }
        ;
```

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

lista_elementos:
        elemento { $$.code = $1.code; }
        | lista_elementos ',' elemento { sprintf(temp, "%s\n%s", $1.code, $3.code);
                $$.code = gen_code(temp);
                }
        ;

elemento:       expresion { sprintf(temp, "(princ %s)", $1.code);
                $$.code = gen_code(temp);
                }
        | STRING { sprintf(temp, "(princ \"%s\")", $1.code);
                $$.code = gen_code(temp);
```

```
                        }
                ;



%%                      // 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 *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_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,
    "if",         IF,
    "else",       ELSE,
    "while",      WHILE,
    "puts",       PUTS,
    "printf",     PRINTF,
    "==",         IGUAL,
    "!=",         DISTINTO,
    "<=",         MENORIGUAL,
    ">=",         MAYORIGUAL,
    "&&",         AND,
    "||",         OR,
    "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 *) mi_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, #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 == 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();
    return 0;
}
```