```
/*Lucas Gallego Bravo Grupo: 93 */
/*100429005@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) ;

/*Funciones creadas para el uso de array_funciones*/
int searchArray(char*[],char*);

char temp [2048] ;
char drop[6];               /*Este char toma el valor DROP en caso de que un argumento se use en la funcion*/
char param_name[25];        /*Este char se usa para guardar el nombre del parametro de entrada de la funcion*/
char *array_funciones[50];  /*Array para los nombres de las funciones*/
int index_local = 0;        /*Index para el array*/

// Definitions for explicit attributes

typedef struct s_attr {
        int value ;
        char *code ;
```

```
} t_attr ;

#define YYSTYPE t_attr

%}

// Definitions for explicit attributes

%token DEFUN
%token SETQ
%token SETF
%token STRING
%token MAIN
%token WHILE
%token LOOP
%token IF
%token PROGN
%token RETURN
%token FROM
%token PRIN1
%token PRINT
%token IDENTIF
%token NUMBER
%token AND
%token OR
%token NOT
%token MOD
%token DIFF
%token SMALLER
%token BIGGER
%token DO

// Definicion de prioridades de los operadores
%right OR
```

```
%left AND
%left DIFF '='
%left '<' SMALLER '>' BIGGER
%left '+' '-'
%left '*' '/' MOD
%left UNARY_SIGN NOT

%%                          // Seccion 3 Gramatica - Semantico

axioma:     '('decl_var decl_fun      { strcpy(temp,"");
                                        strcpy(temp,"\n");
                                        strcat(temp, $2.code) ;
                                        strcat(temp, $3.code) ;
                                        printf("%s",temp);
                                        strcpy(temp,"");}

            '('MAIN')'          {printf("\n\nmain\nbye");}
          ;


decl_var:   /*lambda*/                  { strcpy(temp,"");
                                      $$.code = gen_code (temp) ; }

          | SETQ IDENTIF expresion ')' '(' decl_var    { strcpy(temp,"");
                                                  if (strcmp($3.code,"0")==0){
                                                      /*Si solo se declara la variable, entramos aqui*/
                                                      sprintf(temp, "variable %s\n%s" ,$2.code,$6.code);
                                                  }
                                                  else{
                                                      /*Si la variable se declara con algun valor, entramos aqui (int a = 2;)*/
                                                      sprintf(temp,"variable %s\n%s %s !\n%s" ,$2.code,$3.code,$2.code,$6.code);
                                                  }
                                                  $$.code = gen_code (temp) ;}
          ;
```

```
decl_fun:   DEFUN funciones                 {strcpy(temp,"");
                                    strcat(temp,$2.code);
                                    $$.code = gen_code (temp) ;}
        ;


funciones:  /*lambda*/                                  {strcpy(temp,"");
                                                $$.code = gen_code (temp) ;}

        | IDENTIF '(' argumentos ')'('decl_var cuerpo ')' '(' DEFUN funciones   { strcpy(temp,"");
                                                                strcpy(param_name,"");
                                                                array_funciones[index_local] =  $1.code;
                                                                index_local += 1;
                                                                strcmp($3.code, "( -- )") == 0 ? strcpy(drop,"") : strcpy(drop,"DROP");
                                                                sprintf(temp,"%s: %s %s\n%s%s\n;\n%s",
                                                                $6.code,$1.code,$3.code,$7.code, drop,$11.code);
                                                                $$.code = gen_code (temp) ;}

        | MAIN '(' argumentos ')'(' decl_var cuerpo ')'     {strcpy(temp,"");
                                                sprintf(temp,"%s: main %s\n%s;",$6.code,$3.code,$7.code);
                                                $$.code = gen_code (temp) ;}
        ;


argumentos:  /*lambda*/                                     {strcpy(temp,"( -- )");
                                                strcpy(param_name,"");
                                                $$.code = gen_code (temp) ;}

        | IDENTIF                       {strcpy(temp,"");
                                    sprintf(temp,"( %s -- )",$1.code);
                                    strcpy(param_name,$1.code);
                                    $$.code = gen_code (temp) ;}
```

```
        | IDENTIF   argumentos2             {strcpy(temp,"");
                                  sprintf(temp,"( %s %s -- )",$1.code,$2.code);
                                  $$.code = gen_code (temp) ;}
        ;


argumentos2:    IDENTIF                     {strcpy(temp,"");
                                  sprintf(temp,"%s",$1.code);
                                  $$.code = gen_code (temp) ;}

          | IDENTIF   argumentos2          {strcpy(temp,"");
                                  sprintf(temp,"%s %s",$1.code,$2.code);
                                  $$.code = gen_code (temp) ;}
        ;


cuerpo:       cuerpo2 ')'             {strcpy(temp,"");
                                 sprintf(temp,"%s",$1.code);
                                 $$.code = gen_code (temp) ;}

        |  cuerpo2 ')' '(' cuerpo      {strcpy(temp,"");
                                   sprintf(temp,"%s%s",$1.code,$4.code);
                                   $$.code = gen_code (temp) ;}
        ;


cuerpo2:    PRINT STRING               {strcpy(temp,"");
                                sprintf(temp,".\" %s\"CR\n",$2.code);
                                $$.code = gen_code (temp) ;}

        | SETF IDENTIF expresion   {strcpy(temp,"");
                                sprintf(temp,"%s %s !\n",$3.code,$2.code);
                                $$.code = gen_code (temp) ;}
```

```
| PRIN1 expresion         {strcpy(temp,"");
                        sprintf(temp,"%s .\n",$2.code);
                        $$.code = gen_code (temp) ;}

| PRIN1 STRING            {strcpy(temp,"");
                        sprintf(temp,".\" %s\"CR\n",$2.code);
                        $$.code = gen_code (temp) ;}

| LOOP WHILE '(' expresion ')' DO '(' cuerpo    {strcpy(temp,"");
                                    sprintf(temp,"BEGIN %s WHILE\n%s\nREPEAT\n",$4.code,$8.code);
                                    $$.code = gen_code (temp) ;}

| IF '(' expresion ')' '(' PROGN '(' cuerpo ')'  {strcpy(temp,"");
                                     sprintf(temp,"%s IF\n%s\nTHEN\n",$3.code,$8.code);
                                     $$.code = gen_code (temp) ;}

| IF '(' expresion ')' '(' PROGN '(' cuerpo ')' '(' PROGN '(' cuerpo ')'  {strcpy(temp,"");
                                                sprintf(temp,"%s IF\n%s\nELSE\n%s\nTHEN\n",$3.code,$8.code,$13.code);
                                                $$.code = gen_code (temp) ;}

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

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

| RETURN '-' FROM IDENTIF expresion         {strcpy(temp,"");
                                     if (strcmp(param_name,"")==0){
                                       strcpy(drop,"");
                                     }
                                     else{
```

```
                                        strcpy(drop,"DROP\n");
                                    }
                                    sprintf(temp,"%s\n%sEXIT\n",$5.code,drop);
                                    strcpy(drop,"");
                                    $$.code = gen_code (temp) ;}
        ;


param_entrada:   expresion                  {strcpy(temp,"");
                                    sprintf(temp,"%s",$1.code);
                                    $$.code = gen_code (temp) ;}

            | expresion   param_entrada2      {strcpy(temp,"");
                                    sprintf(temp,"%s %s",$1.code,$2.code);
                                    $$.code = gen_code (temp) ;}
            ;


param_entrada2:     expresion               {strcpy(temp,"");
                                    sprintf(temp,"%s",$1.code);
                                    $$.code = gen_code (temp) ;}

              | expresion   param_entrada2     {strcpy(temp,"");
                                    sprintf(temp,"%s %s",$1.code,$2.code);
                                    $$.code = gen_code (temp) ;}
              ;


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) ; }

|  OR expresion  expresion   { sprintf (temp, "%s %s or", $2.code, $3.code) ;
                         $$.code = gen_code (temp) ; }
|  AND expresion  expresion   { sprintf (temp, "%s %s and", $2.code, $3.code) ;
                         $$.code = gen_code (temp) ; }

|  DIFF expresion expresion   { sprintf (temp, "%s %s = 0=", $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) ; }

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

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

|  NOT expresion expresion   { sprintf (temp, "%s %s 0=", $2.code, $3.code) ;
                         $$.code = gen_code (temp) ; }
```

```
        |   MOD expresion expresion   { sprintf (temp, "%s %s mod", $2.code, $3.code) ;
                                        $$.code = gen_code (temp) ; }
        ;


termino:        operando                        { $$ = $1 ; }
        |   '(' '+' operando  %prec UNARY_SIGN ')'      { sprintf (temp, "%s", $2.code) ;
                                                $$.code = gen_code (temp) ; }
        |   '(' '-' operando %prec UNARY_SIGN ')'       { sprintf (temp, "0 %s -", $3.code) ;
                                                $$.code = gen_code (temp) ; }
        ;

operando:   IDENTIF                 { sprintf (temp, "%s @", $1.code);
                                    if (searchArray(array_funciones, $1.code) == 0){
                                        sprintf (temp, "%s",$1.code) ;
                                    }
                                    else{
                                        if ( strcmp($1.code, param_name) == 0 ){
                                            sprintf (temp, "DUP");
                                        }
                                        else{
                                            sprintf (temp, "%s @", $1.code);
                                        }
                                    }
                                    $$.code = gen_code (temp);}

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

        |   '(' expresion ')'       { $$ = $2 ; }

        | '('IDENTIF param_entrada ')'      { sprintf (temp, "%s %s",$2.code, $3.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 *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
    "setq",        SETQ,
    "setf",        SETF,
    "defun",       DEFUN,
    "while",       WHILE,
    "loop",        LOOP,
    "progn",       PROGN,
    "if",          IF,
    "return",      RETURN,
    "from",        FROM,
    "print",       PRINT,
    "prin1",       PRIN1,
    "do",          DO,
    "and",         AND,
```

```
    "or",          OR,
    "not",         NOT,
    "mod",         MOD,
    "/=",          DIFF,
    "<=",          SMALLER,
    ">=",          BIGGER,
    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 de Funciones creadas Array_Local ****************/
/*****************************************************************************/

int searchArray(char *array[], char *target) {
    for (int i = 0; i < 50; i++) {
```

```
        if (array[i] != NULL && strcmp(array[i], target) == 0) {
            return 0;
        }
    }
    return 1; // Return 1 if the target is not found
}

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

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

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

    return p ;
}


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