```
%{                          // 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

typedef struct s_lista { // tabla de s mbolos
    char lista[1024][1024];
    int values[1024];
    int i;
} t_lista;


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

// mi tabla de simbolos
int search_local(t_lista l, char *var);
int insert(t_lista *l, char *var, int n);
int remove_all(t_lista *l);

char temp [2048] ;
char nombre_funcion[1024];

t_lista argumentos;
t_lista var_local;
t_lista vec_local;

// Definitions for explicit attributes

typedef struct s_attr {
        int value ;
        char *code ;
} t_attr ;

#define YYSTYPE t_attr

%}

// Definitions for explicit attributes

%token NUMBER
%token IDENTIF       // Identificador=variable
%token STRING
%token MAIN          // identifica el comienzo del proc. main
%token WHILE         // identifica el bucle main
%token PRINT        // identifica la impresion
%token SETQ
%token DEFUN
%token PRIN1
%token SETF
%token DO
%token LOOP
%token IF
%token PROGN
%token NOT
%token RETURN
%token FROM
%token MAKE
%token ARRAY
%token AREF


%right '='                    // minima preferencia
%left OR                      //
%left AND                     //
%left EQUAL NOTEQ             //
%left '<' LEQ '>' GEQ         //
```

```
%left '+' '-'                   //
%left '*' '/' MOD                 //
%left UNARY_SIGN                // maxima preferencia

%%                              // Seccion 3 Gramatica - Semantico

axioma:     '(' bloque ')' codigo   { printf ("\n"); }
            ;

codigo:     '(' bloque ')' codigo   { sprintf(temp, "%s\n%s", $2.code, $4.code); $$.code = gen_code(temp); }
            | /* lambda */          { $$.code = ""; }
            ;

bloque:     sentencia               { $$ = $1 ; }
            | declaracion           { sprintf(temp, "%s", $1.code); $$.code = gen_code(temp); }
            | DEFUN
                IDENTIF             { strcpy(nombre_funcion, $2.code); }
                '(' func_arg ')'
                codigo              {
                                      char asign_args[2048];
                                      strcpy(asign_args, "");
                                      int i;
                                      for (i = 0; i < argumentos.i ; ++i) {
                                         sprintf(asign_args, "%sarg_%s_%s !\n", asign_args,nombre_funcion, argumentos.lista[i]);
                                      } // asignacion de argumentos

                                      char variables_locales[2048];
                                      strcpy(variables_locales, "");
                                      for (i = 0; i < var_local.i ; ++i) {
                                          sprintf(variables_locales, "%svariable %s\n",variables_locales, var_local.lista[i]);
                                      } // declaracion de var_locales

                                      char vec_locales[2048];
                                      strcpy(vec_locales, "");
                                      for (i = 0; i < vec_local.i ; ++i) {
                                          sprintf(vec_locales, "%svariable %s %d cells allot\n",vec_locales, vec_local.lista[i], var_local.values[i]);
                                      } // declaracion de vec_locales

                                      char asign_local[2048];
                                      strcpy(asign_local, "");
                                      for (i = 0; i < var_local.i ; ++i) {
                                          sprintf(asign_local, "%s%i %s !\n", asign_local, var_local.values[i], var_local.lista[i]);
                                      } // asignacion de var locales

                                      printf("%s%s%s: %s\n%s%s%s;\n", $5.code, variables_locales,vec_locales, $2.code, asign_args, asign_local, $7.code);
                                      $$.code = gen_code("");
                                      strcpy(nombre_funcion, "");
                                      remove_all(&argumentos);
                                      remove_all(&var_local);
                                     }

            | LOOP
                WHILE
                expresion
                DO
                codigo              {
                                     sprintf(temp, "BEGIN\n %s WHILE\n %s REPEAT\n", $3.code, $5.code);
                                     $$.code = gen_code(temp);
                                    }
            | IF
                expresion
                '(' PROGN codigo')'
                else                {
                                     sprintf(temp, "%s IF\n %s %s THEN\n", $2.code, $5.code, $7.code);
                                     $$.code = gen_code(temp);
                                    }

            ;

func_arg:    /* lambda */           { $$.code = ""; }
            | IDENTIF func_arg      {
                                     sprintf(temp, "variable arg_%s_%s\n%s", nombre_funcion, $1.code, $2.code);
                                     $$.code = gen_code(temp);
                                     insert(&argumentos, $1.code, 0); // insertar en argumentos
                                    }
            ;
```

```
else:    /* lambda */               { $$.code = ""; }
        | '(' PROGN codigo')'       { sprintf(temp, "ELSE %s", $3.code); $$.code = gen_code(temp); }
        ;

sentencia:    SETF IDENTIF expresion  {
                                        char aux[2048] = "";
                                        if (strcmp(nombre_funcion, "")) {
                                            if (search_local(var_local, $2.code)) {;}
                                            else if (search_local(argumentos, $2.code))
                                                {sprintf(aux, "arg_%s_", nombre_funcion);}
                                        }
                                        sprintf (temp, "%s %s%s !\n", $3.code, aux,$2.code) ;
                                        $$.code = gen_code (temp) ;
                                      }
            |  SETF vector expresion  {
                                        char aux[2048] = "";
                                        if (strcmp(nombre_funcion, "")) {
                                            if (search_local(vec_local, $2.code)) {;}
                                            else if (search_local(argumentos, $2.code))
                                                {sprintf(aux, "arg_%s_", nombre_funcion);}
                                        }
                                        sprintf (temp, "%s %s%s !\n", $3.code, aux,$2.code) ;
                                        $$.code = gen_code (temp) ;
                                      }
            | PRINT STRING            { sprintf(temp, ".\" %s\"", $2.code); $$.code = gen_code(temp); }
            | PRIN1 prin1_arg         { sprintf(temp, "%s", $2.code); $$.code = gen_code(temp); }
            | RETURN '-' FROM
              IDENTIF expresion       { sprintf(temp, "%s\n exit", $5.code); $$.code = gen_code(temp); }
            | funcion                 {
                                        // si estoy en el scope de una funcion
                                        if (strcmp(nombre_funcion, "")) {
                                            $$ = $1;
                                        } else {
                                            // caso @ (funcion)
                                            printf("%s\n", $1.code);
                                        }
                                      }
            ;


prin1_arg:    expresion               { sprintf(temp, "%s .", $1.code); $$.code = gen_code(temp); }
            | STRING                  { sprintf(temp, ".\" %s\"", $1.code); $$.code = gen_code(temp); }
            ;

expresion:    NUMBER                  { sprintf (temp, "%d", $1.value) ;$$.code = gen_code(temp); }
            | IDENTIF                 {
                                        char aux[2048] = "";
                                        if (strcmp(nombre_funcion, "")) {
                                            if (search_local(var_local, $1.code)) {;}
                                            else if (search_local(argumentos, $1.code)) {
                                                sprintf(aux, "arg_%s_", nombre_funcion);
                                            }
                                        }
                                        sprintf (temp, "%s%s @", aux,$1.code) ;
                                        $$.code = gen_code(temp);
                                      }
            |  vector                       {sprintf(temp, "%s @", $1.code); $$.code = gen_code(temp);}
            | '(' operacion ')'             { sprintf(temp, "%s", $2.code); $$.code = gen_code(temp); }
            | '(' NOT expresion ')'         { sprintf(temp, "%s 0=", $3.code); $$.code = gen_code(temp); }
            | '(' '+' expresion ')'         { sprintf(temp, "%s", $3.code);$$.code = gen_code(temp); }
            | '(' '-' expresion ')'         { sprintf(temp, "0 %s -", $3.code); $$.code = gen_code(temp); }
            ;

vector: '(' AREF IDENTIF expresion ')'    {
                                              char aux[2048] = "";

                                              sprintf (temp, "%s%s %s cells +", aux, $3.code, $4.code);
                                              $$.code = gen_code(temp);
                                          }

funcion:    IDENTIF args                  {
                                           if (strcmp(nombre_funcion, $1.code) == 0) {
                                              sprintf(temp, "%s %s", $2.code, "RECURSE");
```

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

args:   /* lambda */                  { $$.code = ""; }
        | expresion args               { sprintf(temp, "%s %s", $1.code, $2.code); $$.code = gen_code(temp); }
        ;

operacion:    '+' 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); $$.code = gen_code(temp); }
            | '<' expresion expresion  { sprintf(temp, "%s %s <", $2.code, $3.code); $$.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); }
            | 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); }
            | GEQ expresion expresion  { sprintf(temp, "%s %s >=", $2.code, $3.code); $$.code = gen_code(temp); }
            | LEQ expresion expresion  { sprintf(temp, "%s %s <=", $2.code, $3.code); $$.code = gen_code(temp); }
            | NOTEQ expresion expresion { sprintf(temp, "%s %s = 0=", $2.code, $3.code); $$.code = gen_code(temp); }
            | funcion                  { $$ = $1 ; }
            ;

declaracion:    SETQ IDENTIF NUMBER   {
                                        if (strcmp(nombre_funcion, "")) {
                                            insert(&var_local, $2.code, $3.value);
                                            $$.code = gen_code ("");
                                        } else {
                                            printf ( "variable %s\n%d %s !\n", $2.code, $3.value, $2.code) ;
                                            $$.code = gen_code (temp) ;
                                        }
                                       }
                | SETQ IDENTIF
                    '('
                    MAKE'-'ARRAY
                    NUMBER
                    ')'                {
                                        if (strcmp(nombre_funcion, "")) {
                                            insert(&vec_local, $2.code, $7.value);
                                            $$.code = gen_code ("");
                                        } else {
                                            printf ( "variable %s %d cells allot\n", $2.code, $7.value);
                                            $$.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) ;
}
```

```
301|
302| char *my_malloc (int nbytes)       // reserva n bytes de memoria dinamica
303| {
304|     char *p ;
305|     static long int nb = 0;        // sirven para contabilizar la memoria
306|     static int nv = 0 ;            // solicitada en total
307|
308|     p = malloc (nbytes) ;
309|     if (p == NULL) {
310|         fprintf (stderr, "No queda memoria para %d bytes mas\n", nbytes) ;
311|         fprintf (stderr, "Reservados %ld bytes en %d llamadas\n", nb, nv) ;
312|         exit (0) ;
313|     }
314|     nb += (long) nbytes ;
315|     nv++ ;
316|
317|     return p ;
318| }
319|
320| /******************************************************************************/
321| /*********************  Seccion de tabla de s mbolos  *********************/
322| /******************************************************************************/
323|
324| /**
325| * Busca un nombre en la tabla
326| *
327| */
328| int search_local(t_lista l, char *var) {
329|     for (int i = 0; i < l.i; ++i) {
330|         if (strcmp(l.lista[i], var) == 0) { // se encuentra la variable
331|             return 1;
332|         }
333|
334|     }
335|     return 0;
336| }
337|
338| /**
339| * inserta un nombre en la lista
340| *
341| */
342| int insert(t_lista *l, char *var, int n) {
343|     strcpy(&(l->lista[l->i][0]), var);
344|     l->values[l->i] = n;
345|
346|     ++(l->i);
347|     return l->i;
348| }
349|
350| /**
351| * vacia la lista
352| */
353| int remove_all(t_lista *l) {
354|     l->i = 0;
355|     return 0;
356| }
357|
358|
359|
360| /******************************************************************************/
361| /********************* Seccion de Palabras Reservadas *********************/
362| /******************************************************************************/
363|
364| typedef struct s_keyword { // para las palabras reservadas de C
365|     char *name ;
366|     int token ;
367| } t_keyword ;
368|
369| t_keyword keywords [] = { // define las palabras reservadas y los
370|     // "main",        MAIN,           // y los token asociados
371|     "print",       PRINT,
372|     "mod",         MOD,
373|     "and",         AND,
374|     "or",          OR,
375|     "/=",          NOTEQ,
376|     "<=",          LEQ,
377|     ">=",          GEQ,
```

```
    "setq",         SETQ,
    "not",          NOT,
    "defun",        DEFUN,
    "prin1",        PRIN1,
    "setf",         SETF,
    "loop",         LOOP,
    "do",           DO,
    "while",        WHILE,
    "if",           IF,
    "progn",        PROGN,
    "return",       RETURN,
    "from",         FROM,
    "make",         MAKE,
    "array",        ARRAY,
    "aref",         AREF,
    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 ()
{
    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) ;
```

Listing 1: back.y