```
%{                          // 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;

// 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


%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 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\n%s%s%s;\n", $5.code, variables_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) ;
                                    $$.code = gen_code (temp) ;
                                }
            | PRINT STRING          { sprintf(temp, ".\" %s\"", $2.code); $$.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);
                                }
            | '(' 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); }
            ;

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) ;
                                        }
                                    }
                ;

/*vector: SETQ IDENTIF '(' MAKE-ARRAY NUMBER')' {sprintf (temp, "variable %s\n%d %s !", $2.code, $3.value, $2.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 tabla de s mbolos  *******************/
/***************************************************************************/

/**
* Busca un nombre en la tabla
*
*/
int search_local(t_lista l, char *var) {
    for (int i = 0; i < l.i; ++i) {
        if (strcmp(l.lista[i], var) == 0) { // se encuentra la variable
            return 1;
        }

    }
    return 0;
}

/**
* inserta un nombre en la lista
*
*/
int insert(t_lista *l, char *var, int n) {
    strcpy(&(l->lista[l->i][0]), var);
    l->values[l->i] = n;
```

```
303|
304|     ++(l->i);
305|     return l->i;
306| }
307|
308| /**
309| * vacia la lista
310| */
311| int remove_all(t_lista *l) {
312|     l->i = 0;
313|     return 0;
314| }
315|
316|
317|
318| /*****************************************************************************/
319| /********************** Seccion de Palabras Reservadas **********************/
320| /*****************************************************************************/
321|
322| typedef struct s_keyword { // para las palabras reservadas de C
323|     char *name ;
324|     int token ;
325| } t_keyword ;
326|
327| t_keyword keywords [] = { // define las palabras reservadas y los
328|     // "main",        MAIN,           // y los token asociados
329|     "print",        PRINT,
330|     "mod",          MOD,
331|     "and",          AND,
332|     "or",           OR,
333|     "/=",           NOTEQ,
334|     "<=",           LEQ,
335|     ">=",           GEQ,
336|     "setq",         SETQ,
337|     "not",          NOT,
338|     "defun",        DEFUN,
339|     "prin1",        PRIN1,
340|     "setf",         SETF,
341|     "loop",         LOOP,
342|     "do",           DO,
343|     "while",        WHILE,
344|     "if",           IF,
345|     "progn",        PROGN,
346|     "return",       RETURN,
347|     "from",         FROM,
348|     NULL,           0               // para marcar el fin de la tabla
349| } ;
350|
351| t_keyword *search_keyword (char *symbol_name)
352| {                                   // Busca n_s en la tabla de pal. res.
353|                                     // y devuelve puntero a registro (simbolo)
354|     int i ;
355|     t_keyword *sim ;
356|
357|     i = 0 ;
358|     sim = keywords ;
359|     while (sim [i].name != NULL) {
360|       if (strcmp (sim [i].name, symbol_name) == 0) {
361|                                     // strcmp(a, b) devuelve == 0 si a==b
362|           return &(sim [i]) ;
363|         }
364|         i++ ;
365|     }
366|
367|     return NULL ;
368| }
369|
370|
371| /*****************************************************************************/
372| /******************* Seccion del Analizador Lexicografico ********************/
373| /*****************************************************************************/
374|
375| char *gen_code (char *name)     // copia el argumento a un
376| {                                       // string en memoria dinamica
377|     char *p ;
378|     int l ;
379|
380|     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 () ;
```

```
456|         }
457|         temp_str [i] = '\0' ;
458|         ungetc (c, stdin) ;
459|
460|         yylval.code = gen_code (temp_str) ;
461|         symbol = search_keyword (yylval.code) ;
462|         if (symbol == NULL) {    // no es palabra reservada -> identificador antes vrariabre
463| //              printf ("\nDEV: IDENTIF %s\n", yylval.code) ;    // PARA DEPURAR
464|             return (IDENTIF) ;
465|         } else {
466| //              printf ("\nDEV: OTRO %s\n", yylval.code) ;       // PARA DEPURAR
467|             return (symbol->token) ;
468|         }
469|     }
470|
471|     if (strchr (ops_expandibles, c) != NULL) { // busca c en ops_expandibles
472|         cc = getchar () ;
473|         sprintf (temp_str, "%c%c", (char) c, (char) cc) ;
474|         symbol = search_keyword (temp_str) ;
475|         if (symbol == NULL) {
476|             ungetc (cc, stdin) ;
477|             yylval.code = NULL ;
478|             return (c) ;
479|         } else {
480|             yylval.code = gen_code (temp_str) ; // aunque no se use
481|             return (symbol->token) ;
482|         }
483|     }
484|
485| //    printf ("\nDEV: LITERAL %d #%c#\n", (int) c, c) ;      // PARA DEPURAR
486|     if (c == EOF || c == 255 || c == 26) {
487| //          printf ("tEOF ") ;                               // PARA DEPURAR
488|         return (0) ;
489|     }
```

Listing 1: back.y