```
/* C sar  L pez  Mantec n , Paula  Sub as  Serrano, grupo 12 */
/* 100472092@alumnos.uc3m.es 100472119@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

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 [2*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[4*2048];
                                      char asign_aux[2*2048];
                                      strcpy(asign_args, "");
                                      int i;
                                      for (i = 0; i < argumentos.i ; ++i) {
                                         sprintf(asign_aux, "arg_%s_%s !\n",nombre_funcion, argumentos.lista[i]);
                                         strcat(asign_args, asign_aux);

                                      } // asignacion de argumentos

                                      char variables_locales[4*2048];
                                      strcpy(variables_locales, "");
                                      for (i = 0; i < var_local.i ; ++i) {
                                          sprintf(asign_aux, "variable %s\n", var_local.lista[i]);
                                          strcat(variables_locales, asign_aux);
                                      } // declaracion de var_locales

                                      char vec_locales[4*2048];
                                      strcpy(vec_locales, "");
                                      for (i = 0; i < vec_local.i ; ++i) {
                                          sprintf(asign_aux, "variable %s %d cells allot\n", vec_local.lista[i], var_local.values[i]);
                                          strcat(vec_locales, asign_aux);
                                      } // declaracion de vec_locales

                                      char asign_local[2048];
                                      strcpy(asign_local, "");
                                      for (i = 0; i < var_local.i ; ++i) {
                                          sprintf(asign_aux, "%i %s !\n", var_local.values[i], var_local.lista[i]);
                                          strcat(asign_local, asign_aux);
                                      } // 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);
                                      remove_all(&vec_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);
                                    }
```

```
151|             ;
152|
153|
154| func_arg:     /* lambda */           { $$.code = ""; }
155|             | IDENTIF func_arg       {
156|                                        sprintf(temp, "variable arg_%s_%s\n%s", nombre_funcion, $1.code, $2.code);
157|                                        $$.code = gen_code(temp);
158|                                        insert(&argumentos, $1.code, 0); // insertar en argumentos
159|                                       }
160|             ;
161|
162| else:      /* lambda */              { $$.code = ""; }
163|          | '(' PROGN codigo')'       { sprintf(temp, "ELSE %s", $3.code); $$.code = gen_code(temp); }
164|          ;
165|
166| sentencia:    SETF IDENTIF expresion  {
167|                                         char aux[2048] = "";
168|                                         if (strcmp(nombre_funcion, "")) {
169|                                             if (search_local(var_local, $2.code)) {;}
170|                                             else if (search_local(argumentos, $2.code))
171|                                                 {sprintf(aux, "arg_%s_", nombre_funcion);}
172|                                         }
173|                                         sprintf (temp, "%s %s%s !\n", $3.code, aux,$2.code) ;
174|                                         $$.code = gen_code (temp) ;
175|                                       }
176|             |  SETF vector expresion  {
177|
178|                                         sprintf (temp, "%s %s !\n", $3.code,$2.code) ;
179|                                         $$.code = gen_code (temp) ;
180|                                       }
181|             | PRINT STRING            { sprintf(temp, ".\" %s\"", $2.code); $$.code = gen_code(temp); }
182|             | PRIN1 prin1_arg         { sprintf(temp, "%s", $2.code); $$.code = gen_code(temp); }
183|             | RETURN '-' FROM
184|                 IDENTIF expresion     { sprintf(temp, "%s\n exit", $5.code); $$.code = gen_code(temp); }
185|             | funcion                 {
186|                                         // si estoy en el scope de una funcion
187|                                         if (strcmp(nombre_funcion, "")) {
188|                                             $$ = $1;
189|                                         } else {
190|                                             // caso @ (funcion)
191|                                             printf("%s\n", $1.code);
192|                                         }
193|                                       }
194|             ;
195|
196|
197| prin1_arg:    expresion               { sprintf(temp, "%s .", $1.code); $$.code = gen_code(temp); }
198|             | STRING                  { sprintf(temp, ".\" %s\"", $1.code); $$.code = gen_code(temp); }
199|             ;
200|
201| expresion:    NUMBER                  { sprintf (temp, "%d", $1.value) ;$$.code = gen_code(temp); }
202|             | IDENTIF                 {
203|                                         char aux[2048] = "";
204|                                         if (strcmp(nombre_funcion, "")) {
205|                                             if (search_local(var_local, $1.code)) {;}
206|                                             else if (search_local(argumentos, $1.code)) {
207|                                                 sprintf(aux, "arg_%s_", nombre_funcion);
208|                                             }
209|                                         }
210|                                         sprintf (temp, "%s%s @", aux,$1.code) ;
211|                                         $$.code = gen_code(temp);
212|                                       }
213|             |  vector                      { sprintf(temp, "%s @", $1.code); $$.code = gen_code(temp); }
214|             | '(' operacion ')'            { sprintf(temp, "%s", $2.code); $$.code = gen_code(temp); }
215|             | '(' NOT expresion ')'        { sprintf(temp, "%s 0=", $3.code); $$.code = gen_code(temp); }
216|             | '(' '+' expresion ')'        { sprintf(temp, "%s", $3.code);$$.code = gen_code(temp); }
217|             | '(' '-' expresion ')'        { sprintf(temp, "0 %s -", $3.code); $$.code = gen_code(temp); }
218|             ;
219|
220| vector: '(' AREF IDENTIF expresion ')'    {
221|                                             sprintf (temp, "%s %s cells +", $3.code, $4.code);
222|                                             $$.code = gen_code(temp);
223|                                           }
224|
225| 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) ;
```

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

```
    "<=",           LEQ,
    ">=",           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...
```

```
455|                 ungetc (cc, stdin) ;
456|             } else {
457|                 c = getchar () ;  // ...
458|                 if (c == '@') { // Si es la secuencia //@  ==> transcribimos la linea
459|                     do {     // Se trata de codigo inline (Codigo embebido en C)
460|                         c = getchar () ;
461|                         putchar (c) ;
462|                     } while (c != '\n') ;
463|                 } else {    // ==> comentario, ignorar la linea
464|                     while (c != '\n') {
465|                         c = getchar () ;
466|                     }
467|                 }
468|             }
469|         } else if (c == '\\') c = getchar () ;
470|
471|         if (c == '\n')
472|             n_line++ ;
473|
474|     } while (c == ' ' || c == '\n' || c == 10 || c == 13 || c == '\t') ;
475|
476|     if (c == '\"') {
477|         i = 0 ;
478|         do {
479|             c = getchar () ;
480|             temp_str [i++] = c ;
481|         } while (c != '\"' && i < 255) ;
482|         if (i == 256) {
483|             printf ("AVISO: string con mas de 255 caracteres en linea %d\n", n_line) ;
484|         }     // habria que leer hasta el siguiente " , pero, y si falta?
485|         temp_str [--i] = '\0' ;
486|         yylval.code = gen_code (temp_str) ;
487|         return (STRING) ;
488|     }
489|
490|     if (c == '.' || (c >= '0' && c <= '9')) {
491|         ungetc (c, stdin) ;
```

Listing 1: back.y