```
%{                          // 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;
```

```
33 t_lista var_local;
34
35 // Definitions for explicit attributes
36
37 typedef struct s_attr {
38         int value ;
39         char *code ;
40 } t_attr ;
41
42 #define YYSTYPE t_attr
43
44 %}
45
46 // Definitions for explicit attributes
47
48 %token NUMBER
49 %token IDENTIF       // Identificador=variable
50 %token STRING
51 %token MAIN          // identifica el comienzo del proc. main
52 %token WHILE         // identifica el bucle main
53 %token PRINT        // identifica la impresion
54 %token SETQ
55 %token DEFUN
56 %token PRIN1
57 %token SETF
58 %token DO
59 %token LOOP
60 %token IF
61 %token PROGN
62 %token NOT
63 %token RETURN
64 %token FROM
65
```

```
66
67 %right '='                        // minima preferencia
68 %left OR                          //
69 %left AND                         //
70 %left EQUAL NOTEQ                //
71 %left '<' LEQ '>' GEQ             //
72 %left '+' '-'                     //
73 %left '*' '/' MOD                    //
74 %left UNARY_SIGN                  // maxima preferencia
75
76 %%                                // Seccion 3 Gramatica - Semantico
77
78 axioma:      '(' bloque ')' codigo   { printf ("%s\n%s", $2.code,$4.code); }
79             ;
80
81 codigo:      '(' bloque ')' codigo   { sprintf(temp, "%s\n%s", $2.code, $4.code); $$.code = gen_code(
   temp); }
82             | /* lambda */          { $$.code = ""; }
83             ;
84
85 bloque:      sentencia               { $$ = $1 ; }
86             | declaracion           { sprintf(temp, "%s", $1.code); $$.code = gen_code(temp); }
87             | DEFUN
88                 IDENTIF             { strcpy(nombre_funcion, $2.code); }
89                 '(' func_arg ')'
90                 codigo              {
91                                       char asign_args[2048];
92                                       strcpy(asign_args, "");
93                                       int i;
94                                       for (i = 0; i < argumentos.i ; ++i) {
95                                          sprintf(asign_args, "%sarg_%s_%s !\n", asign_args,
   nombre_funcion, argumentos.lista[i]);
96                                       } // asignacion de argumentos
```

```
97
98                                      char variables_locales[2048];
99                                      strcpy(variables_locales, "");
100                                     for (i = 0; i < var_local.i ; ++i) {
101                                         sprintf(variables_locales, "%svariable %s\n",
    variables_locales, var_local.lista[i]);
102                                     } // declaracion de var_locales
103
104                                     char asign_local[2048];
105                                     strcpy(asign_local, "");
106                                     for (i = 0; i < var_local.i ; ++i) {
107                                         sprintf(asign_local, "%s%i %s !\n", asign_local, var_local.
    values[i], var_local.lista[i]);
108                                     } // asignacion de var locales
109
110                                     sprintf(temp, "%s%s: %s\n%s%s%s;", $5.code, variables_locales,
    $2.code, asign_args, asign_local, $7.code);
111                                     $$.code = gen_code(temp); strcpy(nombre_funcion, "");
112                                     remove_all(&argumentos);
113                                     remove_all(&var_local);
114                                    }
115
116             | LOOP
117                 WHILE
118                 expresion
119                 DO
120                 codigo              {
121                                      sprintf(temp, "BEGIN %s WHILE %s REPEAT", $3.code, $5.code);
122                                      $$.code = gen_code(temp);
123                                     }
124             | IF
125                 expresion
126                 '(' PROGN codigo')'
```

```
127                 else                    {
128                                          sprintf(temp, "%s IF %s %s THEN", $2.code, $5.code, $7.code);
129                                          $$.code = gen_code(temp);
130                                         }
131
132             ;
133
134 func_arg:     /* lambda */              { $$.code = ""; }
135             | IDENTIF func_arg          {
136                                          sprintf(temp, "variable arg_%s_%s\n%s", nombre_funcion, $1.code,
    $2.code);
137                                          $$.code = gen_code(temp);
138                                          insert(&argumentos, $1.code, 0); // insertar en argumentos
139                                         }
140             ;
141
142 else:      /* lambda */                 { $$.code = ""; }
143         | '(' PROGN codigo')'           { sprintf(temp, "ELSE %s", $3.code); $$.code = gen_code(temp); }
144         ;
145
146 sentencia:     SETF IDENTIF expresion  {
147                                           char aux[2048] = "";
148                                           if (strcmp(nombre_funcion, "")) {
149                                               if (search_local(var_local, $2.code)) {;}
150                                               if (search_local(argumentos, $2.code)) {sprintf(aux, "
    arg_%s_", nombre_funcion);}
151                                           }
152                                           sprintf (temp, "%s %s%s !", $3.code, aux,$2.code) ;
153                                           $$.code = gen_code (temp) ;
154                                         }
155             | PRINT STRING              { sprintf(temp, ".\" %s\"", $2.code); $$.code = gen_code(temp);
    }
156             | IDENTIF                   { sprintf(temp, "%s", $1.code); $$.code = gen_code(temp); }
```

```
157             | PRIN1 prin1_arg           { sprintf(temp, "%s", $2.code); $$.code = gen_code(temp); }
158             | RETURN '-' FROM
159                 IDENTIF expresion       { sprintf(temp, "%s\n exit", $5.code); $$.code = gen_code(temp)
    ; }
160             | funcion                   { $$ = $1; }
161             ;
162
163
164 prin1_arg:    expresion                 { sprintf(temp, "%s .", $1.code); $$.code = gen_code(temp); }
165             | STRING                    { sprintf(temp, ".\" %s\"", $1.code); $$.code = gen_code(temp);
     }
166             ;
167
168 expresion:    NUMBER                    { sprintf (temp, "%d", $1.value) ;$$.code = gen_code(temp); }
169             | IDENTIF                   {
170                                           char aux[2048] = "";
171                                           if (strcmp(nombre_funcion, "")) {
172                                               if (search_local(var_local, $1.code)) {;}
173                                               else if (search_local(argumentos, $1.code)) {sprintf(aux,
    "arg_%s_", nombre_funcion);}}
174                                               sprintf (temp, "%s%s @", aux,$1.code) ;
175                                           $$.code = gen_code(temp);
176                                         }
177             | '(' operacion ')'         { sprintf(temp, "%s", $2.code); $$.code = gen_code(temp); }
178             | '(' NOT expresion ')'     { sprintf(temp, "%s 0=", $3.code); $$.code = gen_code(temp); }
179             | '(' '+' expresion ')'     { sprintf(temp, "%s", $3.code);$$.code = gen_code(temp); }
180             | '(' '-' expresion ')'     { sprintf(temp, "0 %s -", $3.code); $$.code = gen_code(temp); }
181             ;
182
183 funcion:      IDENTIF args              { sprintf(temp, "%s %s", $2.code, $1.code); $$.code = gen_code(
    temp); }
184             ;
185
```

```
186 args:   /* lambda */                    { $$.code = ""; }
187         | expresion args                { sprintf(temp, "%s %s", $1.code, $2.code); $$.code = gen_code(
    temp); }
188
189 operacion:    '+' expresion expresion { sprintf(temp, "%s %s +", $2.code, $3.code); $$.code =
    gen_code(temp); }
190             | '-' expresion expresion { sprintf(temp, "%s %s -", $2.code, $3.code); $$.code =
    gen_code(temp); }
191             | '*' expresion expresion { sprintf(temp, "%s %s *", $2.code, $3.code); $$.code =
    gen_code(temp); }
192             | '/' expresion expresion { sprintf(temp, "%s %s /", $2.code, $3.code); $$.code =
    gen_code(temp); }
193             | MOD expresion expresion { sprintf(temp, "%s %s mod", $2.code, $3.code); $$.code =
    gen_code(temp); }
194             | '<' expresion expresion { sprintf(temp, "%s %s <", $2.code, $3.code); $$.code =
    gen_code(temp); }
195             | '>' expresion expresion { sprintf(temp, "%s %s >", $2.code, $3.code); $$.code =
    gen_code(temp); }
196             | '=' expresion expresion { sprintf(temp, "%s %s =", $2.code, $3.code); $$.code =
    gen_code(temp); }
197             | AND expresion expresion { sprintf(temp, "%s %s and", $2.code, $3.code); $$.code =
    gen_code(temp); }
198             | OR expresion expresion  { sprintf(temp, "%s %s or", $2.code, $3.code); $$.code =
    gen_code(temp); }
199             | GEQ expresion expresion { sprintf(temp, "%s %s >=", $2.code, $3.code); $$.code =
    gen_code(temp); }
200             | LEQ expresion expresion { sprintf(temp, "%s %s <=", $2.code, $3.code); $$.code =
    gen_code(temp); }
201             | NOTEQ expresion expresion { sprintf(temp, "%s %s = 0=", $2.code, $3.code); $$.code =
    gen_code(temp); }
202             | funcion                 { $$ = $1 ; }
203             ;
204
```

```
205 declaracion:    SETQ IDENTIF NUMBER   {
206                                         if (strcmp(nombre_funcion, "")) {
207                                             insert(&var_local, $2.code, $3.value);
208                                             $$.code = gen_code ("");
209                                         } else {
210                                             sprintf (temp, "variable %s\n%d %s !", $2.code, $3.value,
       $2.code) ;
211                                             $$.code = gen_code (temp) ;
212                                         }
213                                       }
214                 ;
215
216 /*vector: SETQ IDENTIF '(' MAKE-ARRAY NUMBER')' {sprintf (temp, "variable %s\n%d %s !", $2.code, $3.
       value, $2.code) ;
217                                                                                 $$.code = gen_code (temp)
       ; }*/
218
219 %%                              // SECCION 4    Codigo en C
220
221 int n_line = 1 ;
222
223 int yyerror (mensaje)
224 char *mensaje ;
225 {
226     fprintf (stderr, "%s en la linea %d\n", mensaje, n_line) ;
227     printf ( "\n") ;  // bye
228 }
229
230 char *int_to_string (int n)
231 {
232     sprintf (temp, "%d", n) ;
233     return gen_code (temp) ;
234 }
```

```
235
236 char *char_to_string (char c)
237 {
238     sprintf (temp, "%c", c) ;
239     return gen_code (temp) ;
240 }
241
242 char *my_malloc (int nbytes)        // reserva n bytes de memoria dinamica
243 {
244     char *p ;
245     static long int nb = 0;         // sirven para contabilizar la memoria
246     static int nv = 0 ;             // solicitada en total
247
248     p = malloc (nbytes) ;
249     if (p == NULL) {
250         fprintf (stderr, "No queda memoria para %d bytes mas\n", nbytes) ;
251         fprintf (stderr, "Reservados %ld bytes en %d llamadas\n", nb, nv) ;
252         exit (0) ;
253     }
254     nb += (long) nbytes ;
255     nv++ ;
256
257     return p ;
258 }
259
260 /*****************************************************************************/
261 /*********************  Seccion de tabla de s mbolos  ********************/
262 /*****************************************************************************/
263
264 /**
265 * Busca un nombre en la tabla
266 *
267 */
```

```
268 int search_local(t_lista l, char *var) {
269     for (int i = 0; i < l.i; ++i) {
270         if (strcmp(l.lista[i], var) == 0) { // se encuentra la variable
271             return 1;
272         }
273 
274     }
275     return 0;
276 }
277 
278 /**
279 * inserta un nombre en la lista
280 *
281 */
282 int insert(t_lista *l, char *var, int n) {
283     strcpy(&(l->lista[l->i][0]), var);
284     l->values[l->i] = n;
285 
286     ++(l->i);
287     return l->i;
288 }
289 
290 /**
291 * vacia la lista
292 */
293 int remove_all(t_lista *l) {
294     l->i = 0;
295     return 0;
296 }
297 
298 
299 
300 /*****************************************************************************/
```

```
301 /********************** Seccion de Palabras Reservadas **********************/
302 /****************************************************************************/
303
304 typedef struct s_keyword { // para las palabras reservadas de C
305     char *name ;
306     int token ;
307 } t_keyword ;
308
309 t_keyword keywords [] = { // define las palabras reservadas y los
310     // "main",        MAIN,           // y los token asociados
311     "print",        PRINT,
312     "mod",          MOD,
313     "and",          AND,
314     "or",           OR,
315     "/=",           NOTEQ,
316     "<=",           LEQ,
317     ">=",           GEQ,
318     "setq",         SETQ,
319     "not",          NOT,
320     "defun",        DEFUN,
321     "prin1",        PRIN1,
322     "setf",         SETF,
323     "loop",         LOOP,
324     "do",           DO,
325     "while",        WHILE,
326     "if",           IF,
327     "progn",        PROGN,
328     "return",       RETURN,
329     "from",         FROM,
330     NULL,           0               // para marcar el fin de la tabla
331 } ;
332
333 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 ;
```

```
367 }
368
369
370 int yylex ()
371 {
372     int i ;
373     unsigned char c ;
374     unsigned char cc ;
375     char ops_expandibles [] = "!<=>|%/&+-*" ;
376     char temp_str [256] ;
377     t_keyword *symbol ;
378
379     do {
380         c = getchar () ;
381         if (c == '#') { // Ignora las lineas que empiezan por #  (#define, #include)
382             do {    //  OJO que puede funcionar mal si una linea contiene #
383                 c = getchar () ;
384             } while (c != '\n') ;
385         }
386
387         if (c == '/') { // Si la linea contiene un / puede ser inicio de comentario
388             cc = getchar () ;
389             if (cc != '/') {   // Si el siguiente char es /  es un comentario, pero...
390                 ungetc (cc, stdin) ;
391             } else {
392                 c = getchar () ;  // ...
393                 if (c == '@') { // Si es la secuencia //@  ==> transcribimos la linea
394                     do {    // Se trata de codigo inline (Codigo embebido en C)
395                         c = getchar () ;
396                         putchar (c) ;
397                     } while (c != '\n') ;
398                 } else {    // ==> comentario, ignorar la linea
399                     while (c != '\n') {
```

```
400                         c = getchar () ;
401                     }
402                 }
403             }
404         } else if (c == '\\') c = getchar () ;
405 
406         if (c == '\n')
407             n_line++ ;
408 
409     } while (c == ' ' || c == '\n' || c == 10 || c == 13 || c == '\t') ;
410 
411     if (c == '\"') {
412         i = 0 ;
413         do {
414             c = getchar () ;
415             temp_str [i++] = c ;
416         } while (c != '\"' && i < 255) ;
417         if (i == 256) {
418             printf ("AVISO: string con mas de 255 caracteres en linea %d\n", n_line) ;
419         }     // habria que leer hasta el siguiente " , pero, y si falta?
420         temp_str [--i] = '\0' ;
421         yylval.code = gen_code (temp_str) ;
422         return (STRING) ;
423     }
424 
425     if (c == '.' || (c >= '0' && c <= '9')) {
426         ungetc (c, stdin) ;
427         scanf ("%d", &yylval.value) ;
428 //          printf ("\nDEV: NUMBER %d\n", yylval.value) ;         // PARA DEPURAR
429         return NUMBER ;
430     }
431 
432     if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')) {
```

```
433         i = 0 ;
434         while (((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z') ||
435             (c >= '0' && c <= '9') || c == '_') && i < 255) {
436             temp_str [i++] = tolower (c) ;
437             c = getchar () ;
438         }
439         temp_str [i] = '\0' ;
440         ungetc (c, stdin) ;
441 
442         yylval.code = gen_code (temp_str) ;
443         symbol = search_keyword (yylval.code) ;
444         if (symbol == NULL) {    // no es palabra reservada -> identificador antes vrariabre
445 //              printf ("\nDEV: IDENTIF %s\n", yylval.code) ;    // PARA DEPURAR
446             return (IDENTIF) ;
447         } else {
448 //              printf ("\nDEV: OTRO %s\n", yylval.code) ;       // PARA DEPURAR
449             return (symbol->token) ;
450         }
451     }
452 
453     if (strchr (ops_expandibles, c) != NULL) { // busca c en ops_expandibles
454         cc = getchar () ;
455         sprintf (temp_str, "%c%c", (char) c, (char) cc) ;
456         symbol = search_keyword (temp_str) ;
457         if (symbol == NULL) {
458             ungetc (cc, stdin) ;
459             yylval.code = NULL ;
460             return (c) ;
461         } else {
462             yylval.code = gen_code (temp_str) ; // aunque no se use
463             return (symbol->token) ;
464         }
465     }
```

```
466
467 //     printf ("\nDEV: LITERAL %d #%c#\n", (int) c, c) ;       // PARA DEPURAR
468     if (c == EOF || c == 255 || c == 26) {
469 //          printf ("tEOF ") ;                                 // PARA DEPURAR
470         return (0) ;
471     }
472
473     return c ;
474 }
475
476
477 int main ()
478 {
479     yyparse () ;
480 }
```

/home/cesar/Documents/Universidad/Tercero/ProcesadoresLenguaje/PL–practicaFinal/cuartaParte/back4.y