

```

1 %{                               // SECCION 1 Declaraciones de C-Yacc
2
3 #include <stdio.h>
4 #include <ctype.h>               // declaraciones para tolower
5 #include <string.h>             // declaraciones para cadenas
6 #include <stdlib.h>             // declaraciones para exit ()
7
8 #define FF fflush(stdout);      // para forzar la impresion inmediata
9
10 typedef struct s_lista { // tabla de smbolos
11     char lista[1024][1024];
12     int values[1024];
13     int i;
14 } t_lista;
15
16
17 int yylex ();
18 int yyerror ();
19 char *mi_malloc (int) ;
20 char *gen_code (char *) ;
21 char *int_to_string (int) ;
22 char *char_to_string (char) ;
23
24 // mi tabla de simbolos
25 int search_local(t_lista l, char *var);
26 int insert(t_lista *l, char *var, int n);
27 int remove_all(t_lista *l);
28
29 char temp [2048] ;
30 char nombre_funcion[1024];
31
32 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 ("\n"); }
79 ;
80
81 codigo:      '(', bloque ')', codigo    { sprintf(temp, "%s\n%s", $2.code, $4.code); $$code = gen_code(temp); }
82 | /* lambda */
83 |
84 ;
85 bloque:      sentencia
86 | declaracion
87 | DEFUN
88 | IDENTIF
89 | '(', func_arg ')',
90 | codigo
91 {
92     char asign_args[2048];
93     strcpy(asign_args, "");
94     int i;
95     for (i = 0; i < argumentos.i ; ++i) {
96         sprintf(asign_args, "%sarg_%s_%s !\n", asign_args, nombre_funcion, argumentos.lista[i]);
97     } // asignacion de argumentos
98
99     char variables_locales[2048];
100    strcpy(variables_locales, "");
101    for (i = 0; i < var_local.i ; ++i) {
102        sprintf(variables_locales, "%svariable %s\n", variables_locales, var_local.lista[i]);
103    } // declaracion de var_locales
104
105    char asign_local[2048];
106    strcpy(asign_local, "");
107    for (i = 0; i < var_local.i ; ++i) {
108        sprintf(asign_local, "%s%i %s !\n", asign_local, var_local.values[i], var_local.lista[i]);
109    } // asignacion de var locales
110
111    printf("%s%s: %s\n%s%s;\n", $5.code, variables_locales, $2.code, asign_args, asign_local, $7.code);
112    $$code = gen_code("");
113    strcpy(nombre_funcion, "");
114    remove_all(&argumentos);
115    remove_all(&var_local);
116 }
117
118 | LOOP
119 | WHILE
120 | expresion
121 | DO
122 | codigo
123 {
124     sprintf(temp, "BEGIN\n %s WHILE\n %s REPEAT\n", $3.code, $5.code);
125     $$code = gen_code(temp);
126 }
127
128 | IF
129 | expresion
130 | '(', PROGN codigo)',
131 | else
132 {
133     sprintf(temp, "%s IF\n %s %s THEN\n", $2.code, $5.code, $7.code);
134     $$code = gen_code(temp);
135 }
136
137 ;
138
139 func_arg:    /* lambda */
140 | IDENTIF func_arg
141 {
142     $$code = ""; }
143
144 | IDENTIF func_arg
145 {
146     sprintf(temp, "variable arg_%s_%s\n%s", nombre_funcion, $1.code, $2.code);
147     $$code = gen_code(temp);
148     insert(&argumentos, $1.code, 0); // insertar en argumentos
149 }
150
151 ;
152
153 else:        /* lambda */
154 | '(', PROGN codigo)',
155 {
156     $$code = ""; }
157
158 | '(', PROGN codigo)',
159 {
160     sprintf(temp, "ELSE %s", $3.code);
161     $$code = gen_code(temp); }
162
163 ;
164
165 sentencia:   SETF IDENTIF expresion {
166     char aux[2048] = "";
167     if (strcmp(nombre_funcion, "")) {
168         if (search_local(var_local, $2.code)) {}
169         else if (search_local(argumentos, $2.code))

```

```

152|                                     {sprintf(aux, "arg-%s-", nombre_funcion);}
153|                                     }
154|                                     sprintf (temp, "%s %s%s !\n", $3.code, aux,$2.code) ;
155|                                     $$$.code = gen_code (temp) ;
156|                                     }
157|                                     | PRINT STRING
158|                                     | PRIN1 prin1_arg
159|                                     | RETURN '-', FROM
160|                                     | IDENTIF expression
161|                                     | funcion
162|                                     {
163|                                     // si estoy en el scope de una funcion
164|                                     if (strcmp(nombre_funcion, "")) {
165|                                     $$$ = $1;
166|                                     } else {
167|                                     // caso @ (funcion)
168|                                     printf("%s\n", $1.code);
169|                                     }
170|                                     }
171|                                     ;
172|
173| prin1_arg:      expression
174|                 | STRING
175|                 ;
176|
177| expresion:      NUMBER
178|                 | IDENTIF
179|                 {
180|                 char aux[2048] = "";
181|                 if (strcmp(nombre_funcion, "")) {
182|                 if (search_local(var_local, $1.code)) {};
183|                 else if (search_local(argumentos, $1.code)) {sprintf(aux, "arg-%s-", nombre_funcion);}
184|                 sprintf (temp, "%s%s @", aux,$1.code) ;
185|                 $$$$.code = gen_code(temp);
186|                 }
187|                 | '(' operacion ')'
188|                 | '(' NOT expresion ')'
189|                 | '(' '+' expresion ')'
190|                 | '(' '-' expresion ')'
191|                 ;
192| funcion:        IDENTIF args
193|                 {
194|                 if (strcmp(nombre_funcion, $1.code) == 0) {
195|                 sprintf(temp, "%s %s", $2.code, "RECURSE");
196|                 } else {
197|                 sprintf(temp, "%s %s", $2.code, $1.code); $$$$.code = gen_code(temp);
198|                 }
199|                 $$$$.code = gen_code(temp);
200|                 fprintf(stderr, "%s - %s\n", nombre_funcion, $1.code);
201|                 }
202|                 ;
203| args:           /* lambda */
204|                 | expresion args
205|                 ;
206|
207| operacion:      '+' expresion expresion
208|                 | '-' expresion expresion
209|                 | '*' expresion expresion
210|                 | '/' expresion expresion
211|                 | MOD expresion expresion
212|                 | '<' expresion expresion
213|                 | '>' expresion expresion
214|                 | '=' expresion expresion
215|                 | AND expresion expresion
216|                 | OR expresion expresion
217|                 | GEQ expresion expresion
218|                 | LEQ expresion expresion
219|                 | NOTEQ expresion expresion
220|                 | funcion
221|                 ;
222|
223| declaracion:    SETQ IDENTIF NUMBER {
224|                 if (strcmp(nombre_funcion, "")) {
225|                 insert(&var_local, $2.code, $3.value);
226|                 $$$$.code = gen_code ("");

```

```

227|         } else {
228|             printf ( "variable %s\n%d %s !\n", $2.code, $3.value, $2.code) ;
229|             $$code = gen_code (temp) ;
230|         }
231|     }
232|
233|     ;
234| /*vector: SETQ IDENTIF '(' MAKE-ARRAY NUMBER')' {sprintf (temp, "variable %s\n%d %s !", $2.code, $3.value, $2.code) ;
235|                                     $$code = gen_code (temp) ; }*/
236|
237|%%                                     // SECCION 4     Codigo en C
238|
239|int n_line = 1 ;
240|
241|int yyerror (mensaje)
242|char *mensaje ;
243|{
244|    fprintf (stderr, "%s en la linea %d\n", mensaje, n_line) ;
245|    printf ( "\n" ) ; // bye
246|}
247|
248|char *int_to_string (int n)
249|{
250|    sprintf (temp, "%d", n) ;
251|    return gen_code (temp) ;
252|}
253|
254|char *char_to_string (char c)
255|{
256|    sprintf (temp, "%c", c) ;
257|    return gen_code (temp) ;
258|}
259|
260|char *my_malloc (int nbytes)          // reserva n bytes de memoria dinamica
261|{
262|    char *p ;
263|    static long int nb = 0 ;           // sirven para contabilizar la memoria
264|    static int nv = 0 ;                // solicitada en total
265|
266|    p = malloc (nbytes) ;
267|    if (p == NULL) {
268|        fprintf (stderr, "No queda memoria para %d bytes mas\n", nbytes) ;
269|        fprintf (stderr, "Reservados %ld bytes en %d llamadas\n", nb, nv) ;
270|        exit (0) ;
271|    }
272|    nb += (long) nbytes ;
273|    nv++ ;
274|
275|    return p ;
276|}
277|
278|/*****
279|***** Seccion de tabla de s mbolos *****/
280|/*****
281|
282|/**
283| * Busca un nombre en la tabla
284| *
285| */
286|int search_local(t_lista l, char *var) {
287|    for (int i = 0; i < l.i; ++i) {
288|        if (strcmp(l.lista[i], var) == 0) { // se encuentra la variable
289|            return 1;
290|        }
291|    }
292|    return 0;
293|}
294|
295|/**
296| * inserta un nombre en la lista
297| *
298| */
299|int insert(t_lista *l, char *var, int n) {
300|    strcpy(&(l->lista[l->i][0]), var);
301|    l->values[l->i] = n;
302|}

```

```

303|     ++(l->i);
304|     return l->i;
305| }
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| {
353|     // Busca n_s en la tabla de pal. res.
354|     // y devuelve puntero a registro (simbolo)
355|     int i ;
356|     t_keyword *sim ;
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|     return NULL ;
367| }
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 ;

```

```

381| p = (char *) my_malloc (1) ;
382| strcpy (p, name) ;
383|
384| return p ;
385| }
386|
387|
388| int yylex ()
389| {
390|     int i ;
391|     unsigned char c ;
392|     unsigned char cc ;
393|     char ops_expandibles [] = "!<=>|%/&+-*" ;
394|     char temp_str [256] ;
395|     t_keyword *symbol ;
396|
397|     do {
398|         c = getchar () ;
399|         if (c == '#') { // Ignora las lineas que empiezan por # (#define, #include)
400|             do { // OJO que puede funcionar mal si una linea contiene #
401|                 c = getchar () ;
402|             } while (c != '\n') ;
403|         }
404|
405|         if (c == '/') { // Si la linea contiene un / puede ser inicio de comentario
406|             cc = getchar () ;
407|             if (cc != '/') { // Si el siguiente char es / es un comentario, pero...
408|                 ungetc (cc, stdin) ;
409|             } else {
410|                 c = getchar () ; // ...
411|                 if (c == '@') { // Si es la secuencia //@ ==> transcribimos la linea
412|                     do { // Se trata de codigo inline (Codigo embebido en C)
413|                         c = getchar () ;
414|                         putchar (c) ;
415|                     } while (c != '\n') ;
416|                 } else { // ==> comentario, ignorar la linea
417|                     while (c != '\n') {
418|                         c = getchar () ;
419|                     }
420|                 }
421|             }
422|         } else if (c == '\\') c = getchar () ;
423|
424|         if (c == '\n')
425|             n_line++ ;
426|
427|     } while (c == ' ' || c == '\n' || c == 10 || c == 13 || c == '\t') ;
428|
429|     if (c == '\n') {
430|         i = 0 ;
431|         do {
432|             c = getchar () ;
433|             temp_str [i++] = c ;
434|         } while (c != '\n' && i < 255) ;
435|         if (i == 256) {
436|             printf ("AVISO: string con mas de 255 caracteres en linea %d\n", n_line) ;
437|             // habria que leer hasta el siguiente " , pero, y si falta?
438|             temp_str [--i] = '\0' ;
439|             yylval.code = gen_code (temp_str) ;
440|             return (STRING) ;
441|         }
442|
443|         if (c == '.' || (c >= '0' && c <= '9')) {
444|             ungetc (c, stdin) ;
445|             scanf ("%d", &yylval.value) ;
446|             // printf ("nDEV: NUMBER %d\n", yylval.value) ; // PARA DEPURAR
447|             return NUMBER ;
448|         }
449|
450|         if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')) {
451|             i = 0 ;
452|             while (((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z') ||
453|                 (c >= '0' && c <= '9') || c == '_' && i < 255) {
454|                 temp_str [i++] = tolower (c) ;
455|                 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 variable
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