

# Práctica: Procesadores del Lenguaje

**Autores:**

Liang Ji Zhu

Ignacio Leal Sánchez



**Fecha de entrega:**

Mayo 2025

Listing 1: Código de back.y

```

1  /* 113 Liang Ji Zhu Ignacio Leal S nchez */
2  /* 100495723@alumnos.uc3m.es 100495680@alumnos.uc3m.es */
3  %{                                     // SECCION 1 Declaraciones de C-Yacc
4
5  #include <stdio.h>
6  #include <ctype.h>                     // declaraciones para tolower
7  #include <string.h>                   // declaraciones para cadenas
8  #include <stdlib.h>                   // declaraciones para exit ()
9
10 #define FF fflush(stdout);           // para forzar la impresion inmediata
11
12 int yylex () ;
13 int yyerror () ;
14 char *mi_malloc (int) ;
15 char *gen_code (char *) ;
16 char *int_to_string (int) ;
17 char *char_to_string (char) ;
18
19 char temp [2048] ;
20 char funcion_name[100];
21 int operaciones;
22 // Abstract Syntax Tree (AST) Node Structure
23
24 typedef struct ASTnode t_node ;
25
26 struct ASTnode {
27     char *op ;
28     int type ;                         // leaf, unary or binary nodes
29     t_node *left ;
30     t_node *right ;
31 } ;
32
33
34 // Definitions for explicit attributes
35
36 typedef struct s_attr {
37     int value ;                       // - Numeric value of a NUMBER
38     char *code ;                     // - to pass IDENTIFIER names, and other translations
39     t_node *node ;                   // - for possible future use of AST
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 INTEGER // identifica el tipo entero
51 %token STRING
52 %token LOOP
53 %token WHILE // identifica el bucle main
54 %token DO
55 %token SETQ
56 %token SETF
57 %token DEFUN
58 %token MAIN // identifica el comienzo del proc. main
59 %token PRINT
60 %token PRINC
61 %token MOD
62 %token OR
63 %token AND
64 %token NOT
65 %token IF
66 %token PROGN
67
68 %right '==' /* asignaci n */
69 %left "||" /* l gico OR */
70 %left "&&" /* l gico AND */
71 %nonassoc "==" "!=" /* igualdad */
72 %nonassoc '<' '>' "<=" ">=" /* relacionales */
73 %left '+', '-' /* suma/resta */
74 %left '*', '/', '%' /* multiplic./m dulo */
75 %right UNARY_SIGN "!" /* unarios: +un, -un, ! */
76
77 %% // Seccion 3 Gramatica - Semantico
78
79 axioma: var_global def_funcs { printf ("\n%s\n%s\n", $1.code, $2.code); }
80 | def_funcs { printf ("%s\n", $1.code); }
81 ;
82
83 /* ===== Variables globales ===== */
84 var_global: declaracion { $$ = $1; }
85 | var_global declaracion
86 { sprintf (temp, "%s\n%s", $1.code, $2.code);

```

```

87         $$code = gen_code (temp); }
88     ;
89 declaracion:    '( SETQ IDENTIF logical_or )'
90                 { sprintf (temp, "variable_ s\n s_ s!", $3.code, $4.code, $3.code);
91                   $$code = gen_code (temp); }
92     ;
93 /* ===== */
94
95 /* ===== Funcion main y gen rico ===== */
96 def_funcs:      def_funcs def_func
97                 { sprintf (temp, " s\n s", $1.code, $2.code);
98                   $$code = gen_code (temp); }
99     | def_func                                     { $$ = $1; }
100    ;
101 def_func:        '( DEFUN MAIN '( )' cuerpo )'
102                 { sprintf (temp, ":_main_ s_", $6.code);
103                   $$code = gen_code (temp); }
104     | '( DEFUN IDENTIF '( )' cuerpo )'
105         { sprintf (temp, ":_ s_ s_", $3.code, $6.code);
106           $$code = gen_code (temp); }
107    ;
108
109
110 cuerpo:          lista_sentencia                                     { $$ = $1; }
111    ;
112 lista_sentencia: sentencia                                     { $$ = $1; }
113     | lista_sentencia sentencia
114         { sprintf (temp, " s\n s", $1.code, $2.code);
115           $$code = gen_code (temp); }
116    ;
117
118 /* ===== Impresion: print y princ ===== */
119 /* ===== Estructuras de Control: loop while, if then, if else then ===== */
120 sentencia:        '( PRINT STRING )'
121                 { sprintf (temp, ". \"_ s \"", $3.code);
122                   $$code = gen_code (temp); }
123     | '( PRINC logical_or )'
124         { sprintf (temp, " s_. ", $3.code);
125           $$code = gen_code (temp); }
126     | '( PRINC STRING )'
127         { sprintf (temp, " s_. ", $3.code);
128           $$code = gen_code (temp); }
129     | '( SETF IDENTIF logical_or )'
130         { sprintf (temp, " s_ s_!", $4.code, $3.code);

```

```

131     $$$.code = gen_code (temp); }
132 | '(' LOOP WHILE logical_or DO lista_sentencia ')'
133     { sprintf (temp, "begin\n\t%s\n\t%s\nrepeat", $4.code, $6.code);
134     $$$.code = gen_code (temp); }
135 | '(' IF logical_or sentencia ')'
136     { sprintf (temp, "%s_if_\n\t%s_\nthen", $3.code, $4.code);
137     $$$.code = gen_code (temp); }
138 | '(' IF logical_or sentencia ')'
139     { sprintf (temp, "%s_if_\n\t%s_\nelse_\n\t%s_\nthen", $3.code, $4.code, $5.code);
140     $$$.code = gen_code (temp); }
141 | '(' PROGNO lista_sentencia ')' { $$ = $3; }
142 ;
143
144 /* ===== Operadores, precedencia y asociatividad ===== */
145 logical_or: logical_and { $$ = $1; }
146 | '(' OR logical_or logical_and ')'
147     { sprintf (temp, "%s_%s_or", $3.code, $4.code);
148     $$$.code = gen_code (temp); }
149 ;
150 logical_and: igualdad { $$ = $1; }
151 | '(' AND logical_and igualdad ')'
152     { sprintf (temp, "%s_%s_and", $3.code, $4.code);
153     $$$.code = gen_code (temp); }
154 ;
155 igualdad: relacional { $$ = $1; }
156 | '(' '=' igualdad relacional ')'
157     { sprintf (temp, "%s_%s=", $3.code, $4.code);
158     $$$.code = gen_code (temp); }
159 | '(' '/' '=' igualdad relacional ')'
160     { sprintf (temp, "%s_%s=_0=", $4.code, $5.code);
161     $$$.code = gen_code (temp); }
162 ;
163 relacional: aditivo { $$ = $1; }
164 | '(' '<' relacional aditivo ')'
165     { sprintf (temp, "%s_%s<", $3.code, $4.code);
166     $$$.code = gen_code (temp); }
167 | '(' '>' relacional aditivo ')'
168     { sprintf (temp, "%s_%s>", $3.code, $4.code);
169     $$$.code = gen_code (temp); }
170 | '(' '<' '=' relacional aditivo ')'
171     { sprintf (temp, "%s_%s<=", $4.code, $5.code);
172     $$$.code = gen_code (temp); }
173 | '(' '>' '=' relacional aditivo ')'
174     { sprintf (temp, "%s_%s>=", $4.code, $5.code);

```

```

175         $$$.code = gen_code (temp); }
176     ;
177 aditivo:      multiplicativo      { $$ = $1; }
178 | '(' '+' aditivo multiplicativo ')'
179 | { sprintf (temp, "%s%s+", $3.code, $4.code);
180   | $$$.code = gen_code (temp); }
181 | '(' '-' aditivo multiplicativo ')'
182 | { sprintf (temp, "%s%s-", $3.code, $4.code);
183   | $$$.code = gen_code (temp); }
184 ;
185 multiplicativo: unario      { $$ = $1; }
186 | '(' '*' multiplicativo unario ')'
187 | { sprintf (temp, "%s%s*", $3.code, $4.code);
188   | $$$.code = gen_code (temp); }
189 | '(' '/' multiplicativo unario ')'
190 | { sprintf (temp, "%s%s/", $3.code, $4.code);
191   | $$$.code = gen_code (temp); }
192 | '(' MOD multiplicativo unario ')'
193 | { sprintf (temp, "%s%smod", $3.code, $4.code);
194   | $$$.code = gen_code (temp); }
195 ;
196 unario:      operando      { $$ = $1; }
197 | '(' NOT unario ')'
198 | { sprintf (temp, "%s0=", $3.code);
199   | $$$.code = gen_code (temp); }
200 | '+' operando %prec UNARY_SIGN      { $$ = $1; }
201 | '(' '-' operando %prec UNARY_SIGN ')'
202 | { sprintf (temp, "%snegate", $3.code);
203   | $$$.code = gen_code (temp); }
204 ;
205 operando:    IDENTIF
206
207 | NUMBER
208
209 | '(' logical_or ')'
210 ;
211
212
213 %%                                // SECCION 4     Codigo en C
214
215 int n_line = 1 ;
216
217 int yyerror (mensaje)
218 char *mensaje ;

```

```

219 {
220     fprintf (stderr, "%s_en_la_linea%d\n", mensaje, n_line) ;
221     printf ( "\n" ) ;      // bye
222 }
223
224 char *int_to_string (int n)
225 {
226     sprintf (temp, "%d", n) ;
227     return gen_code (temp) ;
228 }
229
230 char *char_to_string (char c)
231 {
232     sprintf (temp, "%c", c) ;
233     return gen_code (temp) ;
234 }
235
236 char *my_malloc (int nbytes)      // reserva n bytes de memoria dinamica
237 {
238     char *p ;
239     static long int nb = 0;        // sirven para contabilizar la memoria
240     static int nv = 0 ;           // solicitada en total
241
242     p = malloc (nbytes) ;
243     if (p == NULL) {
244         fprintf (stderr, "No queda memoria para %d bytes mas\n", nbytes) ;
245         fprintf (stderr, "Reservados %ld bytes en %d llamadas\n", nb, nv) ;
246         exit (0) ;
247     }
248     nb += (long) nbytes ;
249     nv++ ;
250
251     return p ;
252 }
253
254
255 /*****
256 /***** Seccion de Palabras Reservadas *****/
257 /*****
258
259 typedef struct s_keyword { // para las palabras reservadas de C
260     char *name ;
261     int token ;
262 } t_keyword ;

```

```

263
264 t_keyword keywords [] = { // define las palabras reservadas y los
265     "main",      MAIN,      // y los token asociados
266     "int",       INTEGER,
267     "setq",      SETQ,      // a = 1;    -> setq a 1      -> variable a\n a 1 !
268     "setf",      SETF,
269     "defun",     DEFUN,     // main(); -> (defun main) -> : main <code> ;
270     "print",     PRINT,     // (print "Hola Mundo") -> ." <string>"
271     "princ",     PRINC,     // (princ 22) -> <string> .
272     "loop",      LOOP,
273     "while",     WHILE,
274     "do",        DO,
275     "if",        IF,
276     "progn",     PROGN,
277     "mod",       MOD,
278     "or",        OR,
279     "and",       AND,
280     "not",       NOT,
281     NULL,        0         // para marcar el fin de la tabla
282
283 } ;
284
285 t_keyword *search_keyword (char *symbol_name)
286 {
287     // Busca n_s en la tabla de pal. res.
288     // y devuelve puntero a registro (simbolo)
289
290     int i ;
291     t_keyword *sim ;
292
293     i = 0 ;
294     sim = keywords ;
295     while (sim [i].name != NULL) {
296         if (strcmp (sim [i].name, symbol_name) == 0) {
297             // strcmp(a, b) devuelve == 0 si a==b
298             return &(sim [i]) ;
299         }
300         i++ ;
301     }
302
303     return NULL ;
304 }
305
306 /*****
307 *****/
308 Seccion del Analizador Lexicografico *****/

```



```

307 /*****
308
309 char *gen_code (char *name)      // copia el argumento a un
310 {                                // string en memoria dinamica
311     char *p ;
312     int l ;
313
314     l = strlen (name)+1 ;
315     p = (char *) my_malloc (l) ;
316     strcpy (p, name) ;
317
318     return p ;
319 }
320
321
322 int yylex ()
323 {
324     // NO MODIFICAR ESTA FUNCION SIN PERMISO
325     int i ;
326     unsigned char c ;
327     unsigned char cc ;
328     char ops_expandibles [] = "!<=>%&/+-*" ;
329     char temp_str [256] ;
330     t_keyword *symbol ;
331
332     do {
333         c = getchar () ;
334
335         if (c == '#') { // Ignora las lineas que empiezan por #  (#define, #include)
336             do {                // OJO que puede funcionar mal si una linea contiene #
337                 c = getchar () ;
338             } while (c != '\n') ;
339         }
340
341         if (c == '/') { // Si la linea contiene un / puede ser inicio de comentario
342             cc = getchar () ;
343             if (cc != '/') { // Si el siguiente char es / es un comentario, pero...
344                 ungetc (cc, stdin) ;
345             } else {
346                 c = getchar () ; // ...
347                 if (c == '@') { // Si es la secuencia //@ ==> transcribimos la linea
348                     do { // Se trata de codigo inline (Codigo embebido en C)
349                         c = getchar () ;
350                         putchar (c) ;

```

```

351         } while (c != '\n') ;
352     } else { // ==> comentario, ignorar la linea
353         while (c != '\n') {
354             c = getchar () ;
355         }
356     }
357 }
358 } else if (c == '\\') c = getchar () ;
359
360 if (c == '\n')
361     n_line++ ;
362
363 } while (c == '_' || c == '\n' || c == 10 || c == 13 || c == '\t') ;
364
365 if (c == '\"') {
366     i = 0 ;
367     do {
368         c = getchar () ;
369         temp_str [i++] = c ;
370     } while (c != '\"' && i < 255) ;
371     if (i == 256) {
372         printf ("AVISO: string con mas de 255 caracteres en linea %d\n", n_line) ;
373     } // habria que leer hasta el siguiente " , pero, y si falta?
374     temp_str [--i] = '\0' ;
375     yyval.code = gen_code (temp_str) ;
376     return (STRING) ;
377 }
378
379 if (c == '.' || (c >= '0' && c <= '9')) {
380     ungetc (c, stdin) ;
381     scanf ("%d", &yyval.value) ;
382     // printf ("\nDEV: NUMBER %d\n", yyval.value) ; // PARA DEPURAR
383     return NUMBER ;
384 }
385
386 if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')) {
387     i = 0 ;
388     while (((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z') ||
389         (c >= '0' && c <= '9') || c == '_') && i < 255) {
390         temp_str [i++] = tolower (c) ;
391         c = getchar () ;
392     }
393     temp_str [i] = '\0' ;
394     ungetc (c, stdin) ;

```

```

395     yylval.code = gen_code (temp_str) ;
396     symbol = search_keyword (yylval.code) ;
397     if (symbol == NULL) { // no es palabra reservada -> identificador antes variable
398         printf ("\nDEV: IDENTIF %s\n", yylval.code) ; // PARA DEPURAR
399     //     return (IDENTIF) ;
400     } else {
401         printf ("\nDEV: OTRO %s\n", yylval.code) ; // PARA DEPURAR
402         return (symbol->token) ;
403     }
404 }
405
406
407 if (strchr (ops_expandibles, c) != NULL) { // busca c en ops_expandibles
408     cc = getchar () ;
409     sprintf (temp_str, "%c%c", (char) c, (char) cc) ;
410     symbol = search_keyword (temp_str) ;
411     if (symbol == NULL) {
412         ungetc (cc, stdin) ;
413         yylval.code = NULL ;
414         return (c) ;
415     } else {
416         yylval.code = gen_code (temp_str) ; // aunque no se use
417         return (symbol->token) ;
418     }
419 }
420
421 //     printf ("\nDEV: LITERAL %d %#c#\n", (int) c, c) ; // PARA DEPURAR
422 if (c == EOF || c == 255 || c == 26) {
423     //     printf ("tEOF ") ; // PARA DEPURAR
424     return (0) ;
425 }
426
427 return c ;
428 }
429
430
431 int main ()
432 {
433     yyparse () ;
434 }

```