

Práctica: Procesadores del Lenguaje

Autores:

Liang Ji Zhu

Ignacio Leal Sánchez



Fecha de entrega:

Mayo 2025

Listing 1: Código de trad.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
23 // Abstract Syntax Tree (AST) Node Structure
24
25 typedef struct ASTnode t_node ;
26
27 struct ASTnode {
28     char *op ;
29     int type ;                         // leaf, unary or binary nodes
30     t_node *left ;
31     t_node *right ;
32 } ;
33
34
35 // Definitions for explicit attributes
36
37 typedef struct s_attr {
38     int value ;                       // - Numeric value of a NUMBER
39     char *code ;                      // - to pass IDENTIFIER names, and other translations
40     t_node *node ;                    // - for possible future use of AST
41 } t_attr ;
42

```

```

43 #define YYSTYPE t_attr
44
45 %}
46
47 // Definitions for explicit attributes
48
49 %token NUMBER
50 %token IDENTIF // Identificador=variable
51 %token INTEGER // identifica el tipo entero
52 %token STRING
53 %token RETURN // identifica el return
54 %token MAIN // identifica el comienzo del proc. main
55 %token WHILE // identifica el bucle main
56 %token FOR // identifica el bucle for
57 %token IF // identifica el if
58 %token ELSE // identifica el else
59 %token PUTS // identifica la funci n puts()
60 %token PRINTF // identifica la funcion printf()
61
62 %right '==' /* asignaci n */
63 %left "||" /* l gico OR */
64 %left "&&" /* l gico AND */
65 %nonassoc "==" "!=" /* igualdad */
66 %nonassoc '<' '>' "<=" ">=" /* relacionales */
67 %left '+', '-' /* suma/resta */
68 %left '*', '/', '%', /* multiplic./m dulo */
69 %right UNARY_SIGN "!" /* unarios: +un, -un, ! */
70
71 %% // Seccion 3 Gramatica - Semantico
72
73 axioma: var_global funcion { printf ("%s%s\n", $1.code, $2.code); }
74 r_axioma { ; }
75 ;
76 r_axioma: { ; }
77 | axioma { ; }
78 ;
79
80
81
82 var_global: declaracion ';' var_global { sprintf (temp, "%s\n%s", $1.code, $3.code);
83 $$$code = gen_code (temp); }
84 | { $$$code = ""; }
85 ;
86

```

```

87 declaracion:      INTEGER IDENTIF valor_global r_declaracion
88                  { sprintf (temp, "(setq%s%s)%s", $2.code, $3.code, $4.code);
89                    $$$.code = gen_code (temp); }
90                  | INTEGER IDENTIF '[' NUMBER ']' r_declaracion
91                  { sprintf (temp, "(setq%s(make-array%d))\n%s", $2.code, $4.value, $6.code);
92                    $$$.code = gen_code (temp); }
93                  ;
94
95 valor_global:      { sprintf (temp, "%d", 0 );
96                    $$$.code = gen_code (temp);}
97                  | '=' NUMBER
98                    { sprintf (temp, "%d", $2.value);
99                      $$$.code = gen_code (temp); }
100
101 r_declaracion:     ', ' IDENTIF valor_global r_declaracion
102                  { sprintf (temp, "\n(setq%s%s)%s", $2.code, $3.code, $4.code);
103                    $$$.code = gen_code (temp); }
104                  | ', ' IDENTIF '[' NUMBER ']' r_declaracion
105                  { sprintf (temp, "\n(setq%s(make-array%d))%s", $1.code, $3.value, $5.code);
106                    $$$.code = gen_code (temp); }
107                  |
108                    { $$$.code = ""; }
109                  ;
110
111
112 funcion:           IDENTIF { strcpy(function_name, $1.code); operaciones = 1; } '(' argumento ')' '{' var_local cuerpo '}' funcion
113                  { sprintf (temp, "(defun%s%s)\n\t%s%s\n)\n\t\n%s", $1.code, $4.code, $7.code, $8.code, $10.code);
114                    $$$.code = gen_code (temp); }
115                  | funcion_principal
116                    { $$ = $1; }
117                  ;
118
119 funcion_principal: MAIN { strcpy(function_name, $1.code); operaciones = 1; } '(' argumento ')' '{' var_local cuerpo '}'
120                  { sprintf (temp, "(defunmain%s)\n\t%s%s\n)", $4.code, $7.code, $8.code);
121                    $$$.code = gen_code (temp); }
122                  ;
123
124 argumento:         INTEGER valor resto_argumento
125                  { sprintf (temp, "%s%s", $2.code, $3.code);
126                    $$$.code = gen_code (temp); }
127                  | valor resto_argumento
128                    { sprintf (temp, "%s%s", $1.code, $2.code);
129                      $$$.code = gen_code (temp); }
130                  |
131                    { $$$.code = ""; }
132                  ;
133
134 valor:             STRING
135                    { $$ = $1; }

```

```

131         | expresion                                { $$ = $1; }
132         ;
133
134 resto_argumento:  ',,' argumento                    { sprintf (temp, "%s", $2.code);
135                                                         $$code = gen_code (temp); }
136         |
137         ;
138
139
140 var_local:        declaracion_local ';,' var_local    { sprintf (temp, "%s\n\t%s", $1.code, $3.code);
141                                                         $$code = gen_code (temp); }
142         |
143         ;
144
145 declaracion_local: INTEGER IDENTIF valor_local r_decl_local
146                 { sprintf (temp, "(setq%s_%s)s", funcion_name, $2.code, $3.code, $4.code);
147                   $$code = gen_code (temp); }
148         | INTEGER IDENTIF '[,' NUMBER ']', r_decl_local
149                 { sprintf (temp, "(setq%s_(make-array%d))\n%s", $2.code, $4.value, $6.code);
150                   $$code = gen_code (temp); }
151         ;
152
153 valor_local:      /* lambda */                        { sprintf (temp, "%d", 0);
154                                                         $$code = gen_code (temp);}
155         | '=,' NUMBER                                { sprintf (temp, "%d", $2.value);
156                                                         $$code = gen_code (temp); }
157         ;
158 r_decl_local:     ',,' IDENTIF valor_local r_decl_local
159                 { sprintf (temp, "\n\t(setq%s_%s)", funcion_name, $2.code, $3.code);
160                   $$code = gen_code (temp); }
161         | ',,' IDENTIF '[,' NUMBER ']', r_decl_local
162                 { sprintf (temp, "(setq%s_(make-array%d))\n%s", $2.code, $4.value, $6.code);
163                   $$code = gen_code (temp); }
164         |
165         ;
166
167
168
169 cuerpo:          sentencia ';,' cuerpo                { sprintf (temp, "%s\n\t%s", $1.code, $3.code);
170                                                         $$code = gen_code (temp); }
171         | sentencia ';,'
172         | estructura cuerpo
173         | estructura
174         ;

```

```

175 | RETURN expresion ';'          { $$ = $2; }
176 ;
177
178
179
180 estructura: WHILE '(' expresion ')' '{' cuerpo_estructura '}'
181     { sprintf (temp, "(loop_while_s_do\n\t%s)", $3.code, $6.code);
182       $$code = gen_code (temp); }
183 | IF '(' expresion ')' '{' cuerpo_estructura '}'
184     { sprintf (temp, "(if_s\n\t%s)", $3.code, $6.code); operaciones = 1;
185       $$code = gen_code (temp); }
186 | IF '(' expresion ')' '{' cuerpo_estructura '}' ELSE '{' cuerpo_estructura '}'
187     { sprintf (temp, "(if_s\n\t%s\n\t%s)", $3.code, $6.code, $10.code); operaciones = 1;
188       $$code = gen_code (temp); }
189 | FOR '(' declaracion_for ';' expresion ';' asignacion ')' '{' cuerpo_estructura '}'
190     { sprintf (temp, "%s\n\t(loop_while_s_do\n\t%s\n\t%s)", $3.code, $5.code, $10.code, $7.code);
191       $$code = gen_code (temp); }
192 ;
193
194
195 declaracion_for: INTEGER IDENTIF valor_for r_declaracion_for
196     { sprintf (temp, "(setq_s_s_s)s", funcion_name, $2.code, $3.code, $4.code);
197       $$code = gen_code (temp); }
198 | IDENTIF valor_for r_declaracion_for
199     { sprintf (temp, "(setq_s_s_s)s", funcion_name, $1.code, $2.code, $3.code);
200       $$code = gen_code (temp); }
201 ;
202 valor_for: { sprintf (temp, "%d", 0);
203             $$code = gen_code (temp); }
204 | '=' NUMBER { sprintf (temp, "%d", $2.value);
205               $$code = gen_code (temp); }
206 ;
207 r_declaracion_for: ',' IDENTIF valor_for r_declaracion_for
208     { sprintf (temp, "\n(setq_s_s_s)s", funcion_name, $2.code, $3.code, $4.code);
209       $$code = gen_code (temp); }
210 | { $$code = ""; }
211 ;
212
213
214
215 cuerpo_estructura: sentencia ';' { if (operaciones == 2) {
216                                     $$ = $1; }
217 else {
218     sprintf (temp, "(progn\t%s)", $1.code);

```

```

219                                     $$$.code = gen_code(temp); } }
220 | estructura                                { $$ = $1; }
221 | sentencia ';' cuerpo_estructura          { sprintf (temp, "(progn\t%s\n\t%s)", $1.code, $3.code);
222                                     $$$.code = gen_code (temp); }
223 | estructura cuerpo_estructura             { sprintf (temp, "(progn\t%s\n\t%s)", $1.code, $2.code);
224                                     $$$.code = gen_code (temp); }
225 | RETURN expresion ';'
226     { sprintf (temp, "(return-from%s%s)", funcion_name, $2.code);
227     $$$.code = gen_code (temp); }
228 ;
229 sentencia: asignacion                    { $$ = $1; }
230 | '@' expresion                          { sprintf (temp, "(print%s)", $2.code);
231                                     $$$.code = gen_code (temp); }
232 | PUTS '(' STRING ')',                   { sprintf (temp, "(print%s)", $3.code);
233                                     $$$.code = gen_code (temp); }
234 | PRINTF printf                          { $$$.code = $2.code; }
235 | llamada                                { $$$.code = $1.code; }
236 ;
237
238 printf: '(' STRING r_printf ')',         { $$$.code = $3.code; }
239 ;
240
241 r_printf: ',,' expresion r_printf
242     { sprintf(temp, "(princ%s)\n\t%s", $2.code, $3.code); operaciones ++;
243     $$$.code = gen_code(temp); }
244 | ',,' STRING r_printf
245     { sprintf(temp, "(princ%s)\n\t%s", $2.code, $3.code); operaciones ++;
246     $$$.code = gen_code(temp); }
247 |
248                                     { $$$.code = gen_code(""); }
249 ;
250
251
252 asignacion: IDENTIF '=' expresion
253     { sprintf (temp, "(setf%s%s)", funcion_name, $1.code, $3.code);
254     $$$.code = gen_code (temp); }
255 | vector '=' expresion                  { sprintf (temp, "(setf%s)", $1.code, $3.code);
256                                     $$$.code = gen_code (temp); }
257 ;
258
259
260 expresion: logical_or                    { $$ = $1; }
261 ;
262

```

```

263 llamada: IDENTIF '(', argumento ')', { sprintf (temp, "(%s%s)", $1.code, $3.code);
264                                     $$code = gen_code (temp); }
265                                     ;
266
267 /* ===== Operadores, precedencia y asociatividad ===== */
268 logical_or: logical_and { $$ = $1; }
269             | logical_or '|' logical_and { sprintf (temp, "(or%s%s)", $1.code, $4.code);
270                                             $$code = gen_code (temp); }
271             ;
272 logical_and: igualdad { $$ = $1; }
273             | logical_and '&' igualdad { sprintf (temp, "(and%s%s)", $1.code, $4.code);
274                                             $$code = gen_code (temp); }
275             ;
276 igualdad: relacional { $$ = $1; }
277          | igualdad '=' relacional { sprintf (temp, "(=%s%s)", $1.code, $4.code);
278                                       $$code = gen_code (temp); }
279          | igualdad '!' relacional { sprintf (temp, "(/=s%s)", $1.code, $4.code);
280                                       $$code = gen_code (temp); }
281          ;
282 relacional: aditivo { $$ = $1; }
283            | relacional '<' aditivo { sprintf (temp, "<=%s%s)", $1.code, $3.code);
284                                       $$code = gen_code (temp); }
285            | relacional '>' aditivo { sprintf (temp, ">=%s%s)", $1.code, $3.code);
286                                       $$code = gen_code (temp); }
287            | relacional '<=' aditivo { sprintf (temp, "(<=%s%s)", $1.code, $4.code);
288                                       $$code = gen_code (temp); }
289            | relacional '>=' aditivo { sprintf (temp, "(>=%s%s)", $1.code, $4.code);
290                                       $$code = gen_code (temp); }
291            ;
292 aditivo: multiplicativo { $$ = $1; }
293         | aditivo '+' multiplicativo { sprintf (temp, "(+%s%s)", $1.code, $3.code);
294                                         $$code = gen_code (temp); }
295         | aditivo '-' multiplicativo { sprintf (temp, "(-%s%s)", $1.code, $3.code);
296                                         $$code = gen_code (temp); }
297         ;
298 multiplicativo: unario { $$ = $1; }
299                | multiplicativo '*' unario { sprintf (temp, "(%s%s)", $1.code, $3.code);
300                                                $$code = gen_code (temp); }
301                | multiplicativo '/' unario { sprintf (temp, "(/%s%s)", $1.code, $3.code);
302                                                $$code = gen_code (temp); }
303                | multiplicativo '%' unario { sprintf (temp, "(mod%s%s)", $1.code, $3.code);
304                                                $$code = gen_code (temp); }
305                ;
306 unario: operando { $$ = $1; }

```


307	'!' unario	{ sprintf (temp, "(not%s)", \$2.code);
308		\$\$\$.code = gen_code (temp); }
309	'+' operando %prec UNARY_SIGN	{ \$\$ = \$2; }
310	'-' operando %prec UNARY_SIGN	{ sprintf (temp, "(-%s)", \$2.code);
311		\$\$\$.code = gen_code (temp); }
312	;	
313		
314	operando: IDENTIF	{ sprintf (temp, "%s_%s", funcion_name , \$1.code);
315		\$\$\$.code = gen_code (temp); }
316	IDENTIF '(' argumento ')'	{ sprintf (temp, "(%s%s)", \$1.code, \$3.code);
317		\$\$\$.code = gen_code (temp); }
318	NUMBER	{ sprintf (temp, "%d", \$1.value);
319		\$\$\$.code = gen_code (temp); }
320	'(' logical_or ')'	{ \$\$ = \$2; }
321	vector	{ \$\$ = \$1; }
322	;	
323		
324	vector: IDENTIF '[' logical_or ']'	{ sprintf (temp, "(aref%s%s)", \$1.code, \$3.code);
325		\$\$\$.code = gen_code (temp); }
326	;	
327	%% // SECCION 4 Codigo en C	
328		
329	int n_line = 1 ;	
330		
331	int yyerror (mensaje)	
332	char *mensaje ;	
333	{	
334	fprintf (stderr, "%s_en_la_linea_%d\n", mensaje, n_line) ;	
335	printf ("\n") ; // bye	
336	}	
337		
338	char *int_to_string (int n)	
339	{	
340	sprintf (temp, "%d", n) ;	
341	return gen_code (temp) ;	
342	}	
343		
344	char *char_to_string (char c)	
345	{	
346	sprintf (temp, "%c", c) ;	
347	return gen_code (temp) ;	
348	}	
349		
350	char *my_malloc (int nbytes) // reserva n bytes de memoria dinamica	

```

351 {
352     char *p ;
353     static long int nb = 0;          // sirven para contabilizar la memoria
354     static int nv = 0 ;              // solicitada en total
355
356     p = malloc (nbytes) ;
357     if (p == NULL) {
358         fprintf (stderr, "No queda memoria para %d bytes mas\n", nbytes) ;
359         fprintf (stderr, "Reservados %ld bytes en %d llamadas\n", nb, nv) ;
360         exit (0) ;
361     }
362     nb += (long) nbytes ;
363     nv++ ;
364
365     return p ;
366 }
367
368
369 /*****
370 /***** Seccion de Palabras Reservadas *****/
371 /*****/
372
373 typedef struct s_keyword { // para las palabras reservadas de C
374     char *name ;
375     int token ;
376 } t_keyword ;
377
378 t_keyword keywords [] = { // define las palabras reservadas y los
379     "main",          MAIN,          // y los token asociados
380     "int",           INTEGER,
381     "puts",          PUTS,
382     "printf",        PRINTF,
383     "while",         WHILE,
384     "if",            IF,
385     "else",          ELSE,
386     "for",           FOR,
387     "return",        RETURN,
388     NULL,            0              // para marcar el fin de la tabla
389 } ;
390
391 t_keyword *search_keyword (char *symbol_name)
392 {
393     // Busca n_s en la tabla de pal. res.
394     // y devuelve puntero a registro (simbolo)
395
396     int i ;

```

```

395     t_keyword *sim ;
396
397     i = 0 ;
398     sim = keywords ;
399     while (sim [i].name != NULL) {
400         if (strcmp (sim [i].name, symbol_name) == 0) {
401             // strcmp(a, b) devuelve == 0 si a==b
402             return &(sim [i]) ;
403         }
404         i++ ;
405     }
406
407     return NULL ;
408 }
409
410
411 /*****
412 /***** Seccion del Analizador Lexicografico *****/
413 /*****
414
415 char *gen_code (char *name)      // copia el argumento a un
416 {                                // string en memoria dinamica
417     char *p ;
418     int l ;
419
420     l = strlen (name)+1 ;
421     p = (char *) my_malloc (l) ;
422     strcpy (p, name) ;
423
424     return p ;
425 }
426
427
428 int yylex ()
429 {
430     // NO MODIFICAR ESTA FUNCION SIN PERMISO
431     int i ;
432     unsigned char c ;
433     unsigned char cc ;
434     char ops_expandibles [] = "!<=|>%&/+ -*" ;
435     char temp_str [256] ;
436     t_keyword *symbol ;
437
438     do {

```

```

439 c = getchar () ;
440
441 if (c == '#') { // Ignora las lineas que empiezan por # (#define, #include)
442     do { // OJO que puede funcionar mal si una linea contiene #
443         c = getchar () ;
444     } while (c != '\n') ;
445 }
446
447 if (c == '/') { // Si la linea contiene un / puede ser inicio de comentario
448     cc = getchar () ;
449     if (cc != '/') { // Si el siguiente char es / es un comentario, pero...
450         ungetc (cc, stdin) ;
451     } else {
452         c = getchar () ; // ...
453         if (c == '@') { // Si es la secuencia //@ ==> transcribimos la linea
454             do { // Se trata de codigo inline (Codigo embebido en C)
455                 c = getchar () ;
456                 putchar (c) ;
457             } while (c != '\n') ;
458         } else { // ==> comentario, ignorar la linea
459             while (c != '\n') {
460                 c = getchar () ;
461             }
462         }
463     }
464 } else if (c == '\\') c = getchar () ;
465
466 if (c == '\n')
467     n_line++ ;
468
469 } while (c == '\r' || c == '\n' || c == 10 || c == 13 || c == '\t') ;
470
471 if (c == '"') {
472     i = 0 ;
473     do {
474         c = getchar () ;
475         temp_str [i++] = c ;
476     } while (c != '"' && i < 255) ;
477     if (i == 256) {
478         printf ("AVISO: string con mas de 255 caracteres en linea%d\n", n_line) ;
479     } // habria que leer hasta el siguiente " , pero, y si falta?
480     temp_str [--i] = '\0' ;
481     yylval.code = gen_code (temp_str) ;
482     return (STRING) ;

```

```

483 }
484
485 if (c == '.' || (c >= '0' && c <= '9')) {
486     ungetc (c, stdin) ;
487     scanf ("%d", &yylval.value) ;
488 //     printf ("\nDEV: NUMBER %d\n", yylval.value) ;           // PARA DEPURAR
489     return NUMBER ;
490 }
491
492 if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')) {
493     i = 0 ;
494     while (((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z') ||
495         (c >= '0' && c <= '9') || c == '_') && i < 255) {
496         temp_str [i++] = tolower (c) ;
497         c = getchar () ;
498     }
499     temp_str [i] = '\0' ;
500     ungetc (c, stdin) ;
501
502     yylval.code = gen_code (temp_str) ;
503     symbol = search_keyword (yylval.code) ;
504     if (symbol == NULL) {      // no es palabra reservada -> identificador antes variable
505 //         printf ("\nDEV: IDENTIF %s\n", yylval.code) ;      // PARA DEPURAR
506         return (IDENTIF) ;
507     } else {
508 //         printf ("\nDEV: OTRO %s\n", yylval.code) ;          // PARA DEPURAR
509         return (symbol->token) ;
510     }
511 }
512
513 if (strchr (ops_expandibles, c) != NULL) { // busca c en ops_expandibles
514     cc = getchar () ;
515     sprintf (temp_str, "%c%c", (char) c, (char) cc) ;
516     symbol = search_keyword (temp_str) ;
517     if (symbol == NULL) {
518         ungetc (cc, stdin) ;
519         yylval.code = NULL ;
520         return (c) ;
521     } else {
522         yylval.code = gen_code (temp_str) ; // aunque no se use
523         return (symbol->token) ;
524     }
525 }
526

```

```

527 //      printf ("\nDEV: LITERAL %d %#c#\n", (int) c, c) ;           // PARA DEPURAR
528     if (c == EOF || c == 255 || c == 26) {
529 //          printf ("tEOF ") ;                                       // PARA DEPURAR
530         return (0) ;
531     }
532
533     return c ;
534 }
535
536
537 int main ()
538 {
539     yyparse () ;
540 }

```