

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 %token EQ
62 %token NE
63 %token LE
64 %token GE
65 %token OR
66 %token AND
67
68 %right '=<', /* asignaci n */
69 %left OR /* l gico OR "||" */
70 %left AND /* l gico AND "&&" */
71 %nonassoc EQ NE /* ==, != */
72 %nonassoc '<' '>' LE GE /* <, >, <=, >= */
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 funcion { printf ("%s%s\n", $1.code, $2.code); }
80 r_axioma { ; }
81 ;
82 r_axioma: { ; }
83 | axioma { ; }
84 ;
85
86

```

```

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

```

```

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

```

```

173
174 cuerpo:          sentencia ';' cuerpo          { sprintf (temp, "%s\n\t%s", $1.code, $3.code);
175                                                         $$$.code = gen_code (temp); }
176          | sentencia ';'          { $$ = $1; }
177          | estructura cuerpo      { sprintf (temp, "%s\n\t%s", $1.code, $2.code);
178                                                         $$$.code = gen_code (temp); }
179          | estructura          { $$ = $1; }
180          | RETURN expresion ';'   { $$ = $2; }
181          ;
182
183
184
185 estructura:      WHILE '(' expresion ')' '{' cuerpo_estructura '}'
186                  { sprintf (temp, "(loop_while%s\ndo\n\t%s)", $3.code, $6.code);
187                  $$$.code = gen_code (temp); }
188          | IF '(' expresion ')' '{' cuerpo_estructura '}'
189                  { sprintf (temp, "(if%s\n\t%s)", $3.code, $6.code); operaciones = 1;
190                  $$$.code = gen_code (temp); }
191          | IF '(' expresion ')' '{' cuerpo_estructura '}' ELSE '{' cuerpo_estructura '}'
192                  { sprintf (temp, "(if%s\n\t%s\n\t%s)", $3.code, $6.code, $10.code); operaciones = 1;
193                  $$$.code = gen_code (temp); }
194          | FOR '(' declaracion_for ';' expresion ';' asignacion ')' '{' cuerpo_estructura '}'
195                  { sprintf (temp, "%s\n\t(loop_while%s\ndo\n\t%s\n\t%s)", $3.code, $5.code, $10.code, $7.code);
196                  $$$.code = gen_code (temp); }
197          ;
198
199
200 declaracion_for:  INTEGER IDENTIF valor_for r_declaracion_for
201                  { sprintf (temp, "(setq%s_%s%s)%s", funcion_name, $2.code, $3.code, $4.code);
202                  $$$.code = gen_code (temp); }
203          | IDENTIF valor_for r_declaracion_for
204                  { sprintf (temp, "(setq%s_%s%s)%s", funcion_name, $1.code, $2.code, $3.code);
205                  $$$.code = gen_code (temp); }
206          ;
207 valor_for:
208
209          | '=' NUMBER          { sprintf (temp, "%d", 0);
210                                                         $$$.code = gen_code (temp); }
211          ;
212 r_declaracion_for:  ',', IDENTIF valor_for r_declaracion_for
213                  { sprintf (temp, "\n(setq%s_%s%s)%s", funcion_name, $2.code, $3.code, $4.code);
214                  $$$.code = gen_code (temp); }
215          |
216                  { $$$.code = ""; }
217          ;

```

```

217
218
219
220 cuerpo_estructura:  sentencia ';'
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

```

261         { sprintf (temp, "(setf_␣s_␣s)", $1.code, $3.code);
262         $$ .code = gen_code (temp); }
263
264     ;
265
266 expresion:      logical_or      { $$ = $1; }
267     ;
268
269 llamada:        IDENTIF '(' argumento ')' { sprintf (temp, "(s_␣s)", $1.code, $3.code);
270     ;                                           $$ .code = gen_code (temp); }
271
272
273 /* ===== Operadores, precedencia y asociatividad ===== */
274 logical_or:      logical_and      { $$ = $1; }
275     | logical_or OR logical_and { sprintf (temp, "(or_␣s_␣s)", $1.code, $3.code);
276     ;                                           $$ .code = gen_code (temp); }
277
278 logical_and:     igualdad          { $$ = $1; }
279     | logical_and AND igualdad { sprintf (temp, "(and_␣s_␣s)", $1.code, $3.code);
280     ;                                           $$ .code = gen_code (temp); }
281
282 igualdad:        relacional        { $$ = $1; }
283     | igualdad EQ relacional { sprintf (temp, "(=_␣s_␣s)", $1.code, $3.code);
284     ;                                           $$ .code = gen_code (temp); }
285     | igualdad NE relacional { sprintf (temp, "(/=␣s_␣s)", $1.code, $3.code);
286     ;                                           $$ .code = gen_code (temp); }
287
288 relacional:      aditivo           { $$ = $1; }
289     | relacional '<' aditivo { sprintf (temp, "(<_␣s_␣s)", $1.code, $3.code);
290     ;                                           $$ .code = gen_code (temp); }
291     | relacional '>' aditivo { sprintf (temp, "(>_␣s_␣s)", $1.code, $3.code);
292     ;                                           $$ .code = gen_code (temp); }
293     | relacional LE aditivo { sprintf (temp, "(<=_␣s_␣s)", $1.code, $3.code);
294     ;                                           $$ .code = gen_code (temp); }
295     | relacional GE aditivo { sprintf (temp, "(>=_␣s_␣s)", $1.code, $3.code);
296     ;                                           $$ .code = gen_code (temp); }
297
298 aditivo:         multiplicativo    { $$ = $1; }
299     | aditivo '+' multiplicativo { sprintf (temp, "(+_␣s_␣s)", $1.code, $3.code);
300     ;                                           $$ .code = gen_code (temp); }
301     | aditivo '-' multiplicativo { sprintf (temp, "(-_␣s_␣s)", $1.code, $3.code);
302     ;                                           $$ .code = gen_code (temp); }
303
304 multiplicativo:  ;
305     unario      { $$ = $1; }

```



```

305         | multiplicativo '*' unario
306
307         | multiplicativo '/' unario
308
309         | multiplicativo '%' unario
310
311     ;
312 unario:
313     | '!' unario
314
315     | '+' operando %prec UNARY_SIGN
316     | '-' operando %prec UNARY_SIGN
317
318     ;
319
320 operando:
321     IDENTIF
322     | IDENTIF '(' argumento ')'
323
324     | NUMBER
325
326     | '(' logical_or ')'
327     | vector
328     ;
329
330 vector:
331     IDENTIF '[' logical_or ']'
332
333     ;
334
335 %%
336                                     // SECCION 4      Codigo en C
337
338 int n_line = 1 ;
339
340 int yyerror (mensaje)
341 char *mensaje ;
342 {
343     fprintf (stderr, "%s en la linea %d\n", mensaje, n_line) ;
344     printf ( "\n" ) ;      // bye
345 }
346
347 char *int_to_string (int n)
348 {
349     sprintf (temp, "%d", n) ;
350     return gen_code (temp) ;
351 }

```

```

{ sprintf (temp, "(*%s%s)", $1.code, $3.code);
  $$ .code = gen_code (temp); }
{ sprintf (temp, "(/%s%s)", $1.code, $3.code);
  $$ .code = gen_code (temp); }
{ sprintf (temp, "(mod%s%s)", $1.code, $3.code);
  $$ .code = gen_code (temp); }

{ $$ = $1; }
{ sprintf (temp, "(not%s)", $2.code);
  $$ .code = gen_code (temp); }
{ $$ = $2; }
{ sprintf (temp, "(-%s)", $2.code);
  $$ .code = gen_code (temp); }

{ sprintf (temp, "%s_%s", funcion_name, $1.code);
  $$ .code = gen_code (temp); }
{ sprintf (temp, "(%s%s)", $1.code, $3.code);
  $$ .code = gen_code (temp); }
{ sprintf (temp, "%d", $1.value);
  $$ .code = gen_code (temp); }
{ $$ = $2; }
{ $$ = $1; }

{ sprintf (temp, "(aref%s%s)", $1.code, $3.code);
  $$ .code = gen_code (temp); }

```

```

349
350 char *char_to_string (char c)
351 {
352     sprintf (temp, "%c", c) ;
353     return gen_code (temp) ;
354 }
355
356 char *my_malloc (int nbytes)          // reserva n bytes de memoria dinamica
357 {
358     char *p ;
359     static long int nb = 0;           // sirven para contabilizar la memoria
360     static int nv = 0 ;               // solicitada en total
361
362     p = malloc (nbytes) ;
363     if (p == NULL) {
364         fprintf (stderr, "No queda memoria para %d bytes mas\n", nbytes) ;
365         fprintf (stderr, "Reservados %ld bytes en %d llamadas\n", nb, nv) ;
366         exit (0) ;
367     }
368     nb += (long) nbytes ;
369     nv++ ;
370
371     return p ;
372 }
373
374
375 /*****
376 /***** Seccion de Palabras Reservadas *****/
377 /*****/
378
379 typedef struct s_keyword { // para las palabras reservadas de C
380     char *name ;
381     int token ;
382 } t_keyword ;
383
384 t_keyword keywords [] = { // define las palabras reservadas y los
385     "main",          MAIN,          // y los token asociados
386     "int",           INTEGER,
387     "puts",          PUTS,
388     "printf",        PRINTF,
389     "while",         WHILE,
390     "==",            EQ,
391     "!=",            NE,
392     "<=",            LE,

```

```

393     ">=",          GE,
394     "||",          OR,
395     "&&",          AND,
396     "if",          IF,
397     "else",        ELSE,
398     "for",          FOR,
399     "return",       RETURN,
400     NULL,          0           // para marcar el fin de la tabla
401 } ;
402
403 t_keyword *search_keyword (char *symbol_name)
404 {
405     // Busca n_s en la tabla de pal. res.
406     // y devuelve puntero a registro (simbolo)
407
408     int i ;
409     t_keyword *sim ;
410
411     i = 0 ;
412     sim = keywords ;
413     while (sim [i].name != NULL) {
414         if (strcmp (sim [i].name, symbol_name) == 0) {
415             // strcmp(a, b) devuelve == 0 si a==b
416             return &(sim [i]) ;
417         }
418         i++ ;
419     }
420
421     return NULL ;
422 }
423
424 /**** Seccion del Analizador Lexicografico *****/
425
426
427 char *gen_code (char *name)    // copia el argumento a un
428 {                               // string en memoria dinamica
429
430     char *p ;
431     int l ;
432
433     l = strlen (name)+1 ;
434     p = (char *) my_malloc (l) ;
435     strcpy (p, name) ;
436
437     return p ;

```

```

437 }
438
439
440 int yylex ()
441 {
442     // NO MODIFICAR ESTA FUNCION SIN PERMISO
443     int i ;
444     unsigned char c ;
445     unsigned char cc ;
446     char ops_expandibles [] = "!<=>%&/+~*" ;
447     char temp_str [256] ;
448     t_keyword *symbol ;
449
450     do {
451         c = getchar () ;
452
453         if (c == '#') { // Ignora las lineas que empiezan por # (#define, #include)
454             do { // OJO que puede funcionar mal si una linea contiene #
455                 c = getchar () ;
456             } while (c != '\n') ;
457         }
458
459         if (c == '/') { // Si la linea contiene un / puede ser inicio de comentario
460             cc = getchar () ;
461             if (cc != '/') { // Si el siguiente char es / es un comentario, pero...
462                 ungetc (cc, stdin) ;
463             } else {
464                 c = getchar () ; // ...
465                 if (c == '@') { // Si es la secuencia //@ ==> transcribimos la linea
466                     do { // Se trata de codigo inline (Codigo embebido en C)
467                         c = getchar () ;
468                         putchar (c) ;
469                     } while (c != '\n') ;
470                 } else { // ==> comentario, ignorar la linea
471                     while (c != '\n') {
472                         c = getchar () ;
473                     }
474                 }
475             }
476         } else if (c == '\\') c = getchar () ;
477
478         if (c == '\n')
479             n_line++ ;
480

```

```

481 } while (c == '\u' || c == '\n' || c == 10 || c == 13 || c == '\t') ;
482
483 if (c == '\\"') {
484     i = 0 ;
485     do {
486         c = getchar () ;
487         temp_str [i++] = c ;
488     } while (c != '\\"' && i < 255) ;
489     if (i == 256) {
490         printf ("AVISO: string con mas de 255 caracteres en linea %d\n", n_line) ;
491     } // habria que leer hasta el siguiente " , pero, y si falta?
492     temp_str [--i] = '\0' ;
493     yyval.code = gen_code (temp_str) ;
494     return (STRING) ;
495 }
496
497 if (c == '.' || (c >= '0' && c <= '9')) {
498     ungetc (c, stdin) ;
499     scanf ("%d", &yyval.value) ;
500 //     printf ("\nDEV: NUMBER %d\n", yyval.value) ; // PARA DEPURAR
501     return NUMBER ;
502 }
503
504 if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')) {
505     i = 0 ;
506     while (((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z') ||
507         (c >= '0' && c <= '9') || c == '_') && i < 255) {
508         temp_str [i++] = tolower (c) ;
509         c = getchar () ;
510     }
511     temp_str [i] = '\0' ;
512     ungetc (c, stdin) ;
513
514     yyval.code = gen_code (temp_str) ;
515     symbol = search_keyword (yyval.code) ;
516     if (symbol == NULL) { // no es palabra reservada -> identificador antes variable
517 //         printf ("\nDEV: IDENTIF %s\n", yyval.code) ; // PARA DEPURAR
518         return (IDENTIF) ;
519     } else {
520 //         printf ("\nDEV: OTRO %s\n", yyval.code) ; // PARA DEPURAR
521         return (symbol->token) ;
522     }
523 }
524

```

```

525     if (strchr (ops_expandibles, c) != NULL) { // busca c en ops_expandibles
526         cc = getchar () ;
527         sprintf (temp_str, "%c%c", (char) c, (char) cc) ;
528         symbol = search_keyword (temp_str) ;
529         if (symbol == NULL) {
530             ungetc (cc, stdin) ;
531             yylval.code = NULL ;
532             return (c) ;
533         } else {
534             yylval.code = gen_code (temp_str) ; // aunque no se use
535             return (symbol->token) ;
536         }
537     }
538
539     // printf ("\nDEV: LITERAL %d #%c#\n", (int) c, c) ; // PARA DEPURAR
540     if (c == EOF || c == 255 || c == 26) {
541         // printf ("tEOF ") ; // PARA DEPURAR
542         return (0) ;
543     }
544
545     return c ;
546 }
547
548
549 int main ()
550 {
551     yyparse () ;
552 }

```