Práctica: Procesadores del Lenguaje

Autores:
Liang Ji Zhu
Ignacio Leal Sánchez

Fecha de entrega:
Mayo 2025

Listing 1: Código de `trad.y`

```
/* 113 Liang Ji Zhu Ignacio Leal S nchez */
/* 100495723@alumnos.uc3m.es 100495680@alumnos.uc3m.es */
%{                          // 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

int yylex () ;
int yyerror () ;
char *mi_malloc (int) ;
char *gen_code (char *) ;
char *int_to_string (int) ;
char *char_to_string (char) ;

char temp [2048] ;
char funcion_name[100];
int operaciones;

// Abstract Syntax Tree (AST) Node Structure

typedef struct ASTnode t_node ;

struct ASTnode {
    char *op ;
    int type ;          // leaf, unary or binary nodes
    t_node *left ;
    t_node *right ;
} ;


// Definitions for explicit attributes

typedef struct s_attr {
    int value ;    // - Numeric value of a NUMBER
    char *code ;   // - to pass IDENTIFIER names, and other translations
    t_node *node ; // - for possible future use of AST
} t_attr ;

```

```
#define YYSTYPE t_attr

%}

// Definitions for explicit attributes

%token NUMBER
%token IDENTIF       // Identificador=variable
%token INTEGER       // identifica el tipo entero
%token STRING
%token RETURN        // identifica el return
%token MAIN          // identifica el comienzo del proc. main
%token WHILE         // identifica el bucle main
%token FOR           // identifica el bucle for
%token IF            // identifica el if
%token ELSE          // identifica el else
%token PUTS          // identifica la funci n puts()
%token PRINTF        // identifica la funcion printf()
%token EQ
%token NE
%token LE
%token GE
%token OR
%token AND

%right '='                              /* asignaci n */
%left OR                                /* l gico OR "||" */
%left AND                               /* l gico AND "&&" */
%nonassoc EQ NE                         /* ==, != */
%nonassoc '<' '>' LE GE                   /* <, >, <=, >= */
%left '+' '-'                           /* suma/resta */
%left '*' '/' '%'                       /* multiplic./m dulo */
%right UNARY_SIGN "!"                   /* unarios: +un, -un, ! */

%%                            // Seccion 3 Gramatica - Semantico

axioma:             var_global funcion                                  { printf ("%s%s\n", $1.code, $2.code); }
                    r_axioma                                            { ; }
                    ;
r_axioma:                                                               { ; }
                    |   axioma                                          { ; }
                    ;
```

```
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                                                                           $$.code = gen_code (temp);}
103                     | '=' NUMBER                                          { sprintf (temp, "%d", $2.value);
104                                                                           $$.code = gen_code (temp); }
105                     ;
106 r_declaracion:        ',' IDENTIF valor_global r_declaracion              { sprintf (temp, "\n(setq␣%s␣%s)%s", $2.code, $3.code, $4.
        code);
107                                                                           $$.code = gen_code (temp); }
108                     | ',' IDENTIF '[' NUMBER ']' r_declaracion           { sprintf (temp, "\n(setq␣%s␣(make-array␣%d))%s", $1.code,
                          $3.value, $5.code);
109                                                                           $$.code = gen_code (temp); }
110                     |                                                     { $$.code = ""; }
111                     ;
112
113
114
115
116 funcion:            IDENTIF { strcpy(funcion_name, $1.code); operaciones = 1; } '(' argumento ')' '{' var_local cuerpo '}' funcion
117                         { sprintf (temp, "(defun␣%s␣(%s)\n\t%s%s\n)\n\t\n%s", $1.code, $4.code, $7.code, $8.code, $10.code);
118                         $$.code = gen_code (temp); }
119                     | funcion_principal
120                         { $$ = $1; }
121                     ;
122
123 funcion_principal:  MAIN { strcpy(funcion_name, $1.code); operaciones = 1; }      '(' argumento ')' '{' var_local cuerpo '}'
124                         { sprintf (temp, "(defun␣main␣(%s)\n\t%s%s\n)", $4.code, $7.code, $8.code);
125                         $$.code = gen_code (temp); }
126                     ;
127
128 argumento:          INTEGER valor resto_argumento                         { sprintf (temp, "%s␣%s", $2.code, $3.code);
```

```
                                                                $$.code = gen_code (temp); }
                    | valor resto_argumento                     { sprintf (temp, "%s %s", $1.code, $2.code);
                                                                $$.code = gen_code (temp); }
                    |                                           { $$.code = ""; }
                    ;

valor:              STRING                                      { $$ = $1; }
                    | expresion                                 { $$ = $1; }
                    ;

resto_argumento:    ',' argumento                               { sprintf (temp, " %s", $2.code);
                                                                $$.code = gen_code (temp); }
                    |                                           { $$.code = ""; }
                    ;


var_local:          declaracion_local ';' var_local             { sprintf (temp, "%s\n\t%s", $1.code, $3.code);
                                                                $$.code = gen_code (temp); }
                    |                                           { $$.code = ""; }
                    ;

declaracion_local:  INTEGER  IDENTIF valor_local r_decl_local
                        { sprintf (temp, "(setq %s %s)%s", $2.code, $3.code, $4.code);
                        $$.code = gen_code (temp); }
                    | INTEGER  IDENTIF '[' NUMBER ']' r_decl_local
                        { sprintf (temp, "(setq %s (make-array %d))\n%s", $2.code, $4.value, $6.code);
                        $$.code = gen_code (temp); }
                    ;

valor_local:        /* lambda */                                { sprintf (temp, "%d", 0);
                                                                $$.code = gen_code (temp);}
                    | '=' NUMBER                                { sprintf (temp, "%d", $2.value);
                                                                $$.code = gen_code (temp); }
                    ;
r_decl_local:       ',' IDENTIF valor_local r_decl_local
                        { sprintf (temp, "\n\t(setq %s %s)", $2.code, $3.code);
                        $$.code = gen_code (temp); }
                    | ',' IDENTIF '[' NUMBER ']' r_decl_local
                        { sprintf (temp, "(setq %s (make-array %d))\n%s", $2.code, $4.value, $6.code);
                        $$.code = gen_code (temp); }
                    |                                           { $$.code = ""; }
                    ;
```

```
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 ';'                                { $$ = $2; }
181                     ;
182
183
184
185 estructura:         WHILE '(' expresion ')' '{' cuerpo_estructura '}'
186                         { sprintf (temp, "(loop␣while␣%s␣do\n\t%s)", $3.code, $6.code);
187                         $$.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␣do\n\t%s\n\t%s)", $3.code, $5.code, $10.code, $7.code);
196                         $$.code = gen_code (temp); }
197                     ;
198
199
200 declaracion_for:    INTEGER  IDENTIF valor_for r_declaracion_for
201                         { sprintf (temp, "(setq␣%s␣%s)%s", $2.code, $3.code, $4.code);
202                         $$.code = gen_code (temp); }
203                     | IDENTIF valor_for r_declaracion_for
204                         { sprintf (temp, "(setq␣%s␣%s)%s", $1.code, $2.code, $3.code);
205                         $$.code = gen_code (temp); }
206                     ;
207 valor_for:                                                                { sprintf (temp, "%d", 0);
208                                                                            $$.code = gen_code (temp);}
209                     | '=' NUMBER                                          { sprintf (temp, "%d", $2.value);
210                                                                            $$.code = gen_code (temp); }
211                     ;
212 r_declaracion_for:  ',' IDENTIF valor_for r_declaracion_for
213                         { sprintf (temp, "\n(setq␣%s␣%s)%s", $2.code, $3.code, $4.code);
214                         $$.code = gen_code (temp); }
215                     |                                                     { $$.code = ""; }
216                     ;
```

```
217
218
219
220 cuerpo_estructura:  sentencia ';'                                   { if (operaciones == 2) {
221                                                                         $$ = $1;}
222                                                                     else {
223                                                                         sprintf (temp, "(progn\t%s)", $1.code);
224                                                                         $$.code = gen_code(temp); } }
225                     | estructura                                    { $$ = $1; }
226                     | sentencia ';' cuerpo_estructura               { sprintf (temp, "(progn\t%s\n\t%s)", $1.code, $3.code);
227                                                                       $$.code = gen_code (temp); }
228                     | estructura cuerpo_estructura                  { sprintf (temp, "(progn\t%s\n\t%s)", $1.code, $2.code);
229                                                                       $$.code = gen_code (temp); }
230                     | RETURN expresion ';'
231                         { sprintf (temp, "(return-from␣%s␣%s)", funcion_name, $2.code);
232                         $$.code = gen_code (temp); }
233                     ;
234 sentencia:          asignacion                                      { $$ = $1; }
235                     | '@' expresion                                 { sprintf (temp, "(print␣%s)", $2.code);
236                                                                       $$.code = gen_code (temp); }
237                     | PUTS '(' STRING ')'                           { sprintf (temp, "(print␣\"%s\")", $3.code);
238                                                                       $$.code = gen_code (temp) ;}
239                     | PRINTF printf                                 { $$.code = $2.code; }
240                     | llamada                                       { $$.code = $1.code; }
241                     ;
242
243 printf:             '(' STRING r_printf ')'                         { $$.code = $3.code; }
244                     ;
245
246 r_printf:           ',' expresion r_printf
247                         { sprintf(temp, "(princ␣%s)\n\t%s", $2.code, $3.code); operaciones ++;
248                         $$.code = gen_code(temp); }
249                     | ',' STRING r_printf
250                         { sprintf(temp, "(princ␣\"%s\")\n\t%s", $2.code, $3.code); operaciones ++;
251                         $$.code = gen_code(temp); }
252                     |                                               { $$.code = gen_code(""); }
253                     ;
254
255
256
257 asignacion:         IDENTIF '=' expresion
258                         { sprintf (temp, "(setf␣%s␣%s)",  $1.code, $3.code);
259                         $$.code = gen_code (temp); }
260                     | vector '=' expresion
```

```
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:    unario                                      { $$ = $1; }
```

```
                    | multiplicativo '*' unario                 { sprintf (temp, "(* %s %s)", $1.code, $3.code);
                                                                $$.code = gen_code (temp); }
                    | multiplicativo '/' unario                 { sprintf (temp, "(/ %s %s)", $1.code, $3.code);
                                                                $$.code = gen_code (temp); }
                    | multiplicativo '%' unario                 { sprintf (temp, "(mod %s %s)", $1.code, $3.code);
                                                                $$.code = gen_code (temp); }
                    ;
unario:             operando                                    { $$ = $1; }
                    | '!' unario                                { sprintf (temp, "(not %s)", $2.code);
                                                                $$.code = gen_code (temp); }
                    | '+' operando %prec UNARY_SIGN             { $$ = $2; }
                    | '-' operando %prec UNARY_SIGN             { sprintf (temp, "(- %s)", $2.code);
                                                                $$.code = gen_code (temp); }
                    ;

operando:           IDENTIF                                     { sprintf (temp, "%s",$1.code);
                                                                $$.code = gen_code (temp); }
                    | IDENTIF '(' argumento ')'                 { sprintf (temp, "(%s %s)", $1.code, $3.code);
                                                                $$.code = gen_code (temp); }
                    | NUMBER                                    { sprintf (temp, "%d", $1.value);
                                                                $$.code = gen_code (temp); }
                    | '(' logical_or ')'                        { $$ = $2; }
                    |  vector                                   { $$ = $1; }
                    ;

vector:             IDENTIF '[' logical_or ']'                  { sprintf (temp, "(aref %s %s)", $1.code, $3.code);
                                                                $$.code = gen_code (temp); }
                    ;
%%                          // SECCION 4    Codigo en C

int n_line = 1 ;

int yyerror (mensaje)
char *mensaje ;
{
    fprintf (stderr, "%s en la linea %d\n", mensaje, n_line) ;
    printf ( "\n") ;    // bye
}

char *int_to_string (int n)
{
    sprintf (temp, "%d", n) ;
    return 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,
```

```
    ">=",           GE,
    "||",           OR,
    "&&",           AND,
    "if",           IF,
    "else",         ELSE,
    "for",          FOR,
    "return",       RETURN,
    NULL,           0               // para marcar el fin de la tabla
} ;

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 ;
```

```
}


int yylex ()
{
// NO MODIFICAR ESTA FUNCION SIN PERMISO
    int i ;
    unsigned char c ;
    unsigned char cc ;
    char ops_expandibles [] = "!<=|>%&/+-*" ;
    char temp_str [256] ;
    t_keyword *symbol ;

    do {
        c = getchar () ;

        if (c == '#') { // Ignora las lineas que empiezan por #  (#define, #include)
            do {                //      OJO que puede funcionar mal si una linea contiene #
                c = getchar () ;
            } while (c != '\n') ;
        }

        if (c == '/') { // Si la linea contiene un / puede ser inicio de comentario
            cc = getchar () ;
            if (cc != '/') {   // Si el siguiente char es /  es un comentario, pero...
                ungetc (cc, stdin) ;
            } else {
                c = getchar () ;        // ...
                if (c == '@') { // Si es la secuencia //@  ==> transcribimos la linea
                    do {                // Se trata de codigo inline (Codigo embebido en C)
                        c = getchar () ;
                        putchar (c) ;
                    } while (c != '\n') ;
                } else {                // ==> comentario, ignorar la linea
                    while (c != '\n') {
                        c = getchar () ;
                    }
                }
            }
        } else if (c == '\\') c = getchar () ;

        if (c == '\n')
            n_line++ ;

```

```
481     } while (c == ' ' || 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         yylval.code = gen_code (temp_str) ;
494         return (STRING) ;
495     }
496
497     if (c == '.' || (c >= '0' && c <= '9')) {
498         ungetc (c, stdin) ;
499         scanf ("%d", &yylval.value) ;
500 //          printf ("\nDEV: NUMBER %d\n", yylval.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         yylval.code = gen_code (temp_str) ;
515         symbol = search_keyword (yylval.code) ;
516         if (symbol == NULL) {    // no es palabra reservada -> identificador antes vrariabre
517 //              printf ("\nDEV: IDENTIF %s\n", yylval.code) ;    // PARA DEPURAR
518             return (IDENTIF) ;
519         } else {
520 //              printf ("\nDEV: OTRO %s\n", yylval.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 }
```