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()

%right '='                                  /* asignaci n */
%left "||"                                  /* l gico OR */
%left "&&"                                  /* l gico AND */
%nonassoc "==" "!="                         /* igualdad */
%nonassoc '<' '>' "<=" ">="                /* relacionales */
%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                                          { ; }
                    ;



var_global:         declaracion ';'  var_global                         { sprintf (temp, "%s\n%s", $1.code, $3.code);
                                                                        $$.code = gen_code (temp); }
                    |                                                   { $$.code = ""; }
                    ;

```

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

valor_global:                                                   { sprintf (temp, "%d", 0 );
                                                                $$.code = gen_code (temp);}
                    | '=' NUMBER                                { sprintf (temp, "%d", $2.value);
                                                                $$.code = gen_code (temp); }
                    ;
r_declaracion:      ',' IDENTIF valor_global r_declaracion
                        { sprintf (temp, "\n(setq␣%s␣%s)%s", $2.code, $3.code, $4.code);
                        $$.code = gen_code (temp); }
                    | ',' IDENTIF '[' NUMBER ']' r_declaracion
                        { sprintf (temp, "\n(setq␣%s␣(make-array␣%d))%s", $1.code, $3.value, $5.code);
                        $$.code = gen_code (temp); }
                    |                                           { $$.code = ""; }
                    ;



funcion:            IDENTIF { strcpy(funcion_name, $1.code); operaciones = 1; } '(' argumento ')' '{' var_local cuerpo '}' funcion
                        { sprintf (temp, "(defun␣%s␣(%s)\n\t%s%s\n)\n\t\n%s", $1.code, $4.code, $7.code, $8.code, $10.code);
                        $$.code = gen_code (temp); }
                    | funcion_principal                         { $$ = $1; }
                    ;

funcion_principal:  MAIN { strcpy(funcion_name, $1.code); operaciones = 1; }      '(' argumento ')' '{' var_local cuerpo '}'
                        { sprintf (temp, "(defun␣main␣(%s)\n\t%s%s\n)", $4.code, $7.code, $8.code);
                        $$.code = gen_code (temp); }
                    ;

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; }
```

```
131                     | expresion                                                   { $$ = $1; }
132                     ;
133
134 resto_argumento:     ',' argumento                                               { sprintf (temp, "␣%s", $2.code);
135                                                                                   $$.code = gen_code (temp); }
136                     |                                                           { $$.code = ""; }
137                     ;
138
139
140 var_local:          declaracion_local ';' var_local                             { sprintf (temp, "%s\n\t%s", $1.code, $3.code);
141                                                                                   $$.code = gen_code (temp); }
142                     |                                                           { $$.code = ""; }
143                     ;
144
145 declaracion_local:  INTEGER  IDENTIF valor_local r_decl_local
146                         { sprintf (temp, "(setq␣%s_%s␣%s)%s", funcion_name, $2.code, $3.code, $4.code);
147                         $$.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                                                                                   $$.code = gen_code (temp); }
157                     ;
158 r_decl_local:       ',' IDENTIF valor_local r_decl_local
159                         { sprintf (temp, "\n\t(setq␣%s_%s␣%s)", funcion_name, $2.code, $3.code);
160                         $$.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                         $$.code = gen_code (temp); }
164                     |                                                           { $$.code = ""; }
165                     ;
166
167
168
169 cuerpo:             sentencia ';' cuerpo                                        { sprintf (temp, "%s\n\t%s", $1.code, $3.code);
170                                                                                   $$.code = gen_code (temp); }
171                     | sentencia ';'                                             { $$ = $1; }
172                     | estructura cuerpo                                         { sprintf (temp, "%s\n\t%s", $1.code, $2.code);
173                                                                                   $$.code = gen_code (temp); }
174                     | estructura                                                { $$ = $1; }
```

```
                    | RETURN expresion ';'                                  { $$ = $2; }
                    ;



estructura:         WHILE '(' expresion ')' '{' cuerpo_estructura '}'
                        { sprintf (temp, "(loop while %s do\n\t%s)", $3.code, $6.code);
                        $$.code = gen_code (temp); }
                    | IF '(' expresion ')' '{' cuerpo_estructura '}'
                        { sprintf (temp, "(if %s\n\t%s)", $3.code, $6.code); operaciones = 1;
                        $$.code = gen_code (temp); }
                    | IF '(' expresion ')' '{' cuerpo_estructura '}' ELSE '{' cuerpo_estructura '}'
                        { sprintf (temp, "(if %s\n\t%s\n\t%s)", $3.code, $6.code, $10.code); operaciones = 1;
                        $$.code = gen_code (temp); }
                    | FOR '(' declaracion_for ';' expresion ';' asignacion ')' '{' cuerpo_estructura '}'
                        { sprintf (temp, "%s\n\t(loop while %s do\n\t%s\n\t%s)", $3.code, $5.code, $10.code, $7.code);
                        $$.code = gen_code (temp); }
                    ;


declaracion_for:    INTEGER  IDENTIF valor_for r_declaracion_for
                        { sprintf (temp, "(setq %s_%s %s)%s", funcion_name, $2.code, $3.code, $4.code);
                        $$.code = gen_code (temp); }
                    | IDENTIF valor_for r_declaracion_for
                        { sprintf (temp, "(setq %s_%s %s)%s", funcion_name, $1.code, $2.code, $3.code);
                        $$.code = gen_code (temp); }
                    ;
valor_for:                                                                { sprintf (temp, "%d", 0);
                                                                          $$.code = gen_code (temp);}
                    | '=' NUMBER                                          { sprintf (temp, "%d", $2.value);
                                                                          $$.code = gen_code (temp); }
                    ;
r_declaracion_for:  ',' IDENTIF valor_for r_declaracion_for
                        { sprintf (temp, "\n(setq %s_%s %s)%s", funcion_name, $2.code, $3.code, $4.code);
                        $$.code = gen_code (temp); }
                    |                                                     { $$.code = ""; }
                    ;



cuerpo_estructura:  sentencia ';'                                         { if (operaciones == 2) {
                                                                                $$ = $1;}
                                                                          else {
                                                                                sprintf (temp, "(progn\t%s)", $1.code);
```

```
                                                                $$.code = gen_code(temp); } }
                    | estructura                                { $$ = $1; }
                    | sentencia ';' cuerpo_estructura           { sprintf (temp, "(progn\t%s\n\t%s)", $1.code, $3.code);
                                                                $$.code = gen_code (temp); }
                    | estructura cuerpo_estructura              { sprintf (temp, "(progn\t%s\n\t%s)", $1.code, $2.code);
                                                                $$.code = gen_code (temp); }
                    | RETURN expresion ';'
                        { sprintf (temp, "(return-from␣%s␣%s)", funcion_name, $2.code);
                        $$.code = gen_code (temp); }
                    ;
sentencia:          asignacion                                  { $$ = $1; }
                    | '@' expresion                             { sprintf (temp, "(print␣%s)", $2.code);
                                                                $$.code = gen_code (temp); }
                    | PUTS '(' STRING ')'                       { sprintf (temp, "(print␣\"%s\")", $3.code);
                                                                $$.code = gen_code (temp) ;}
                    | PRINTF printf                             { $$.code = $2.code; }
                    | llamada                                   { $$.code = $1.code; }
                    ;

printf:             '(' STRING r_printf ')'                     { $$.code = $3.code; }
                    ;

r_printf:           ',' expresion r_printf
                        { sprintf(temp, "(princ␣%s)\n\t%s", $2.code, $3.code); operaciones ++;
                        $$.code = gen_code(temp); }
                    | ',' STRING r_printf
                        { sprintf(temp, "(princ␣\"%s\")\n\t%s", $2.code, $3.code); operaciones ++;
                        $$.code = gen_code(temp); }
                    |                                           { $$.code = gen_code(""); }
                    ;



asignacion:         IDENTIF '=' expresion
                        { sprintf (temp, "(setf␣%s_%s␣%s)", funcion_name, $1.code, $3.code);
                        $$.code = gen_code (temp); }
                    | vector '=' expresion                      { sprintf (temp, "(setf␣%s␣%s)", $1.code, $3.code);
                                                                $$.code = gen_code (temp); }

                    ;

expresion:          logical_or                                  { $$ = $1; }
                    ;
```

```
llamada:            IDENTIF '(' argumento ')'                    { sprintf (temp, "(%s %s)", $1.code, $3.code);
                                                                 $$.code = gen_code (temp); }
                    ;

/* ================== Operadores, precedencia y asociatividad ================== */
logical_or:         logical_and                                  { $$ = $1; }
                    | logical_or '|''|' logical_and              { sprintf (temp, "(or %s %s)", $1.code, $4.code);
                                                                 $$.code = gen_code (temp); }
                    ;
logical_and:        igualdad                                     { $$ = $1; }
                    | logical_and '&''&' igualdad                { sprintf (temp, "(and %s %s)", $1.code, $4.code);
                                                                 $$.code = gen_code (temp); }
                    ;
igualdad:           relacional                                   { $$ = $1; }
                    | igualdad '=''=' relacional                 { sprintf (temp, "(= %s %s)", $1.code, $4.code);
                                                                 $$.code = gen_code (temp); }
                    | igualdad '!''=' relacional                 { sprintf (temp, "(/= %s %s)", $1.code, $4.code);
                                                                 $$.code = gen_code (temp); }
                    ;
relacional:         aditivo                                      { $$ = $1; }
                    | relacional '<' aditivo                     { sprintf (temp, "(< %s %s)", $1.code, $3.code);
                                                                 $$.code = gen_code (temp); }
                    | relacional '>' aditivo                     { sprintf (temp, "(> %s %s)", $1.code, $3.code);
                                                                 $$.code = gen_code (temp); }
                    | relacional '<''=' aditivo                  { sprintf (temp, "(<= %s %s)", $1.code, $4.code);
                                                                 $$.code = gen_code (temp); }
                    | relacional '>''=' aditivo                  { sprintf (temp, "(>= %s %s)", $1.code, $4.code);
                                                                 $$.code = gen_code (temp); }
                    ;
aditivo:            multiplicativo                               { $$ = $1; }
                    | aditivo '+' multiplicativo                 { sprintf (temp, "(+ %s %s)", $1.code, $3.code);
                                                                 $$.code = gen_code (temp); }
                    | aditivo '-' multiplicativo                 { sprintf (temp, "(- %s %s)", $1.code, $3.code);
                                                                 $$.code = gen_code (temp); }
                    ;
multiplicativo:     unario                                       { $$ = $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_%s", funcion_name ,$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) ;
}

char *char_to_string (char c)
{
    sprintf (temp, "%c", c) ;
    return gen_code (temp) ;
}

char *my_malloc (int nbytes)       // reserva n bytes de memoria dinamica
```

```
{
    char *p ;
    static long int nb = 0;         // sirven para contabilizar la memoria
    static int nv = 0 ;             // solicitada en total

    p = malloc (nbytes) ;
    if (p == NULL) {
        fprintf (stderr, "No queda memoria para %d bytes mas\n", nbytes) ;
        fprintf (stderr, "Reservados %ld bytes en %d llamadas\n", nb, nv) ;
        exit (0) ;
    }
    nb += (long) nbytes ;
    nv++ ;

    return p ;
}


/******************************************************************************/
/********************** Seccion de Palabras Reservadas ************************/
/******************************************************************************/

typedef struct s_keyword { // para las palabras reservadas de C
    char *name ;
    int token ;
} t_keyword ;

t_keyword keywords [] = { // define las palabras reservadas y los
    "main",         MAIN,           // y los token asociados
    "int",          INTEGER,
    "puts",         PUTS,
    "printf",        PRINTF,
    "while",       WHILE,
    "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 {
```

```
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 == ' ' || 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 vrariabre
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 }
```