Práctica: Procesadores del Lenguaje

Autores:
Liang Ji Zhu
Ignacio Leal Sánchez

Fecha de entrega:
Mayo 2025

Listing 1: Código de `back.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 LOOP
%token WHILE         // identifica el bucle main
%token DO
%token SETQ
%token SETF
%token DEFUN
%token MAIN          // identifica el comienzo del proc. main
%token PRINT
%token PRINC
%token MOD
%token OR
%token AND
%token NOT
%token IF
%token PROGN

%right '='                              /* asignaci  n */
%left OR                               /* l   gico OR */
%left AND                               /* l   gico AND */
%nonassoc "/="                         /* igualdad */
%nonassoc '<' '>' "<=" ">="             /* relacionales */
%left '+' '-'                           /* suma/resta */
%left '*' '/' MOD                       /* multiplic./m   dulo */
%right UNARY_SIGN NOT                    /* unarios: +un, -un, ! */

%%                          // Seccion 3 Gramatica - Semantico

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

/* ================== Varibles globales ================== */
var_global:         declaracion                                             { $$ = $1; }
                    | var_global declaracion
                        { sprintf (temp, "%s\n%s", $1.code, $2.code);
```

```
                        $$.code = gen_code (temp); }
                    ;
declaracion:        '(' SETQ IDENTIF logical_or ')'
                        { sprintf (temp, "variable␣%s\n%s␣%s␣!", $3.code, $4.code, $3.code);
                        $$.code = gen_code (temp); }
                    ;
/* ================== ================== ================== */

/* ================== Funcion main y gen  rico ================== */
def_funcs:          def_funcs def_func
                        { sprintf (temp, "%s\n%s", $1.code, $2.code);
                        $$.code = gen_code (temp); }
                    | def_func                                                  { $$ = $1; }
                    | def_funcs llamada_main
                        { sprintf(temp, "%s\n%s", $1.code, $2.code);
                        $$.code = gen_code(temp); }
                    | llamada_main                                              { $$ = $1; }
                    ;
llamada_main:       '(' MAIN ')'
                        { sprintf(temp, "main");
                        $$.code = gen_code(temp); }
def_func:           '(' DEFUN MAIN '(' ')' cuerpo ')'
                        { sprintf (temp, ":␣main␣%s␣;", $6.code);
                        $$.code = gen_code (temp); }
                    | '(' DEFUN IDENTIF '(' ')' cuerpo ')'
                        { sprintf (temp, ":␣%s␣%s␣;", $3.code, $6.code);
                        $$.code = gen_code (temp); }
                    ;


cuerpo:             lista_sentencia                                             { $$ = $1; }
                    ;
lista_sentencia:    sentencia                                                   { $$ = $1; }
                    | lista_sentencia sentencia
                        { sprintf (temp, "%s\n%s", $1.code, $2.code);
                        $$.code = gen_code (temp); }
                    ;

/* ================== Impresion: print y princ ================== */
/* ================== Estructuras de Control: loop while, if then, if else then ================== */
sentencia:          '(' PRINT STRING ')'
                        { sprintf (temp, ".\"␣%s\"", $3.code);
                        $$.code = gen_code (temp); }
                    | '(' PRINC logical_or ')'
```

```
                        { sprintf (temp, "%s␣.", $3.code);
                        $$.code = gen_code (temp); }
                    | '(' PRINC STRING ')'
                        { sprintf (temp, "%s␣.", $3.code);
                        $$.code = gen_code (temp); }
                    | '(' SETF IDENTIF logical_or ')'
                        { sprintf (temp, "%s␣%s␣!", $4.code, $3.code);
                        $$.code = gen_code (temp); }
                    | '(' SETQ IDENTIF logical_or ')'
                        { sprintf (temp, "%s␣%s␣!", $4.code, $3.code);
                        $$.code = gen_code (temp); }
                    | '(' LOOP WHILE logical_or DO lista_sentencia ')'
                        { sprintf (temp, "begin\n\t%s\n\t%s\nrepeat", $4.code, $6.code);
                        $$.code = gen_code (temp); }
                    | '(' IF logical_or sentencia ')'
                        { sprintf (temp, "%s␣if␣\n\t%s␣\nthen", $3.code, $4.code);
                        $$.code = gen_code (temp); }
                    | '(' IF logical_or sentencia sentencia ')'
                        { sprintf (temp, "%s␣if␣\n\t%s␣\nelse␣\n\t%s␣\nthen", $3.code, $4.code, $5.code);
                        $$.code = gen_code (temp); }
                    | '(' PROGN lista_sentencia ')'                                 { $$ = $3; }
                    ;

/* ================== Operadores, precedencia y asociatividad ================== */
logical_or:         logical_and                                                     { $$ = $1; }
                    | '(' OR logical_or logical_and ')'
                        { sprintf (temp, "%s␣%s␣or", $3.code, $4.code);
                        $$.code = gen_code (temp); }
                    ;
logical_and:        igualdad                                                        { $$ = $1; }
                    | '(' AND logical_and igualdad ')'
                        { sprintf (temp, "%s␣%s␣and", $3.code, $4.code);
                        $$.code = gen_code (temp); }
                    ;
igualdad:           relacional                                                      { $$ = $1; }
                    | '(' '=' igualdad relacional ')'
                        {sprintf (temp, "%s␣%s␣=", $3.code, $4.code);
                        $$.code = gen_code (temp); }
                    | '(' '/' '=' igualdad relacional ')'
                        { sprintf (temp, "%s␣%s␣=␣0=", $4.code, $5.code);
                        $$.code = gen_code (temp); }
                    ;
relacional:         aditivo                                                         { $$ = $1; }
                    | '(' '<' relacional aditivo ')'
```

```
                        { sprintf (temp, "%s␣%s␣<", $3.code, $4.code);
                        $$.code = gen_code (temp); }
                    | '(' '>' relacional aditivo ')'
                        { sprintf (temp, "%s␣%s␣>", $3.code, $4.code);
                        $$.code = gen_code (temp); }
                    | '(' '<' '=' relacional aditivo ')'
                        { sprintf (temp, "%s␣%s␣<=", $4.code, $5.code);
                        $$.code = gen_code (temp); }
                    | '(' '>' '=' relacional aditivo ')'
                        { sprintf (temp, "%s␣%s␣>=", $4.code, $5.code);
                        $$.code = gen_code (temp); }
                    ;
aditivo:            multiplicativo                                          { $$ = $1; }
                    | '(' '+' aditivo multiplicativo ')'
                        { sprintf (temp, "%s␣%s␣+", $3.code, $4.code);
                        $$.code = gen_code (temp); }
                    | '(' '-' aditivo multiplicativo ')'
                        { sprintf (temp, "%s␣%s␣-", $3.code, $4.code);
                        $$.code = gen_code (temp); }
                    ;
multiplicativo:     unario                                                  { $$ = $1; }
                    | '(' '*' multiplicativo unario ')'
                        { sprintf (temp, "%s␣%s␣*", $3.code, $4.code);
                        $$.code = gen_code (temp); }
                    | '(' '/' multiplicativo unario ')'
                        { sprintf (temp, "%s␣%s␣/", $3.code, $4.code);
                        $$.code = gen_code (temp); }
                    | '(' MOD multiplicativo unario ')'
                        { sprintf (temp, "%s␣%s␣mod", $3.code, $4.code);
                        $$.code = gen_code (temp); }
                    ;
unario:             operando                                                { $$ = $1; }
                    | '(' NOT unario ')'
                        { sprintf (temp, "%s␣0=", $3.code);
                        $$.code = gen_code (temp); }
                    | '+' operando %prec UNARY_SIGN                         { $$ = $1; }
                    | '(' '-' operando %prec UNARY_SIGN ')'
                        {sprintf (temp, "%s␣negate", $3.code);
                        $$.code = gen_code (temp); }
                    ;
operando:           IDENTIF                                                 { sprintf (temp, "%s", $1.code);
                                                                            $$.code = gen_code (temp); }
                    | NUMBER                                                { sprintf (temp, "%d", $1.value);
                                                                            $$.code = gen_code (temp); }
```

```
                    | '(' logical_or ')'                                    { $$ = $2; }
                    ;


%%                          // 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 ;
}
```

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

```
307         }
308         i++ ;
309     }
310
311     return NULL ;
312 }
313
314
315 /*****************************************************************************/
316 /****************** Seccion del Analizador Lexicografico *********************/
317 /*****************************************************************************/
318
319 char *gen_code (char *name)     // copia el argumento a un
320 {                                       // string en memoria dinamica
321     char *p ;
322     int l ;
323
324     l = strlen (name)+1 ;
325     p = (char *) my_malloc (l) ;
326     strcpy (p, name) ;
327
328     return p ;
329 }
330
331
332 int yylex ()
333 {
334 // NO MODIFICAR ESTA FUNCION SIN PERMISO
335     int i ;
336     unsigned char c ;
337     unsigned char cc ;
338     char ops_expandibles [] = "!<=|>%&/+-*" ;
339     char temp_str [256] ;
340     t_keyword *symbol ;
341
342     do {
343         c = getchar () ;
344
345         if (c == '#') { // Ignora las lineas que empiezan por #  (#define, #include)
346             do {                //      OJO que puede funcionar mal si una linea contiene #
347                 c = getchar () ;
348             } while (c != '\n') ;
349         }
350
```

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

```
395
396     if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')) {
397         i = 0 ;
398         while (((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z') ||
399             (c >= '0' && c <= '9') || c == '_') && i < 255) {
400             temp_str [i++] = tolower (c) ;
401             c = getchar () ;
402         }
403         temp_str [i] = '\0' ;
404         ungetc (c, stdin) ;
405 
406         yylval.code = gen_code (temp_str) ;
407         symbol = search_keyword (yylval.code) ;
408         if (symbol == NULL) {    // no es palabra reservada -> identificador antes vrariabre
409 //              printf ("\nDEV: IDENTIF %s\n", yylval.code) ;    // PARA DEPURAR
410             return (IDENTIF) ;
411         } else {
412 //              printf ("\nDEV: OTRO %s\n", yylval.code) ;       // PARA DEPURAR
413             return (symbol->token) ;
414         }
415     }
416 
417     if (strchr (ops_expandibles, c) != NULL) { // busca c en ops_expandibles
418         cc = getchar () ;
419         sprintf (temp_str, "%c%c", (char) c, (char) cc) ;
420         symbol = search_keyword (temp_str) ;
421         if (symbol == NULL) {
422             ungetc (cc, stdin) ;
423             yylval.code = NULL ;
424             return (c) ;
425         } else {
426             yylval.code = gen_code (temp_str) ; // aunque no se use
427             return (symbol->token) ;
428         }
429     }
430 
431 //    printf ("\nDEV: LITERAL %d #%c#\n", (int) c, c) ;      // PARA DEPURAR
432     if (c == EOF || c == 255 || c == 26) {
433 //          printf ("tEOF ") ;                               // PARA DEPURAR
434         return (0) ;
435     }
436 
437     return c ;
438 }
```

```
439
440
441 int main ()
442 {
443     yyparse () ;
444 }
```