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
```

```
43
44  %}
45
46  // Definitions for explicit attributes
47
48  %token NUMBER
49  %token IDENTIF       // Identificador=variable
50  %token INTEGER       // identifica el tipo entero
51  %token STRING
52  %token LOOP
53  %token WHILE         // identifica el bucle main
54  %token DO
55  %token SETQ
56  %token SETF
57  %token DEFUN
58  %token MAIN          // identifica el comienzo del proc. main
59  %token PRINT
60  %token PRINC
61  %token MOD
62  %token OR
63  %token AND
64  %token NOT
65  %token IF
66  %token PROGN
67  %token NE
68  %token LE
69  %token GE
70
71  %right '='                                  /* asignaci n */
72  %left OR                                    /* l gico OR */
73  %left AND                                   /* l gico AND */
74  %nonassoc NE                                /* igualdad */
75  %nonassoc '<' '>' LE GE                     /* relacionales */
76  %left '+' '-'                               /* suma/resta */
77  %left '*' '/' MOD                           /* multiplic./m dulo */
78  %right UNARY_SIGN NOT                       /* unarios: +un, -un, ! */
79
80  %%                          // Seccion 3 Gramatica - Semantico
81
82  axioma:             var_global def_funcs                                    { printf ("\n%s\n%s\n", $1.code, $2.code); }
83                      | def_funcs                                             { printf ("%s\n", $1.code); }
84                      ;
85
86  /* ================== 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);
                        $$.code = gen_code (temp); }
                    | '(' LOOP WHILE logical_or DO lista_sentencia ')'
                        { sprintf (temp, "begin\n\t%s\nwhile\n\t%s\nrepeat", $4.code, $6.code);
                        $$.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); }
                    | '(' NE igualdad relacional ')'
                        { sprintf (temp, "%s␣%s␣=␣0=", $3.code, $4.code);
                        $$.code = gen_code (temp); }
```

```
175                     ;
176 relacional:          operacion                                                   { $$ = $1; }
177                     | '(' '<' relacional operacion ')'
178                         { sprintf (temp, "%s␣%s␣<", $3.code, $4.code);
179                         $$.code = gen_code (temp); }
180                     | '(' '>' relacional operacion ')'
181                         { sprintf (temp, "%s␣%s␣>", $3.code, $4.code);
182                         $$.code = gen_code (temp); }
183                     | '(' LE relacional operacion ')'
184                         { sprintf (temp, "%s␣%s␣<=", $3.code, $4.code);
185                         $$.code = gen_code (temp); }
186                     | '(' GE relacional operacion ')'
187                         { sprintf (temp, "%s␣%s␣>=", $3.code, $4.code);
188                         $$.code = gen_code (temp); }
189                     ;
190 operacion:          unario                                                      { $$ = $1; }
191                     | '(' '+' operacion operacion ')'
192                         { sprintf (temp, "%s␣%s␣+", $3.code, $4.code);
193                         $$.code = gen_code (temp); }
194                     | '(' '-' operacion operacion ')'
195                         { sprintf (temp, "%s␣%s␣-", $3.code, $4.code);
196                         $$.code = gen_code (temp); }
197                     | '(' '*' operacion operacion ')'
198                         { sprintf (temp, "%s␣%s␣*", $3.code, $4.code);
199                         $$.code = gen_code (temp); }
200                     | '(' '/' operacion operacion ')'
201                         { sprintf (temp, "%s␣%s␣/", $3.code, $4.code);
202                         $$.code = gen_code (temp); }
203                     | '(' MOD operacion operacion ')'
204                         { sprintf (temp, "%s␣%s␣mod", $3.code, $4.code);
205                         $$.code = gen_code (temp); }
206                     ;
207 unario:             operando                                                    { $$ = $1; }
208                     | '(' NOT unario ')'
209                         { sprintf (temp, "%s␣0=", $3.code);
210                         $$.code = gen_code (temp); }
211                     | '+' operando %prec UNARY_SIGN                             { $$ = $1; }
212                     | '(' '-' operando %prec UNARY_SIGN ')'
213                         {sprintf (temp, "%s␣negate", $3.code);
214                         $$.code = gen_code (temp); }
215                     ;
216 operando:           IDENTIF                                                     { sprintf (temp, "%s␣@", $1.code);
217                                                                                 $$.code = gen_code (temp); }
218                     | NUMBER                                                    { sprintf (temp, "%d", $1.value);
```

```
                    | '(' logical_or ')'                                    $$.code = gen_code (temp); }
                    ;                                                     { $$ = $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 ;
```

```
}


/*****************************************************************************/
/********************** 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,
    "setq",         SETQ,               // a = 1;   -> setq a 1     -> variable a\n a 1 !
    "setf",         SETF,
    "defun",        DEFUN,              // main();  -> (defun main) -> : main <code> ;
    "print",        PRINT,              // (print "Hola Mundo") -> ." <string>"
    "princ",        PRINC,              // (princ 22) -> <string> .
    "loop",         LOOP,
    "while",        WHILE,
    "do",           DO,
    "if",           IF,
    "progn",        PROGN,
    "mod",          MOD,
    "or",           OR,
    "and",          AND,
    "not",          NOT,
    "/=",           NE,
    "<=",           LE,
    ">=",           GE,
    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 ;
```

```
307     while (sim [i].name != NULL) {
308             if (strcmp (sim [i].name, symbol_name) == 0) {
309                                                 // strcmp(a, b) devuelve == 0 si a==b
310             return &(sim [i]) ;
311         }
312         i++ ;
313     }
314
315     return NULL ;
316 }
317
318
319 /*****************************************************************************/
320 /****************** Seccion del Analizador Lexicografico *********************/
321 /*****************************************************************************/
322
323 char *gen_code (char *name)     // copia el argumento a un
324 {                                       // string en memoria dinamica
325     char *p ;
326     int l ;
327
328     l = strlen (name)+1 ;
329     p = (char *) my_malloc (l) ;
330     strcpy (p, name) ;
331
332     return p ;
333 }
334
335
336 int yylex ()
337 {
338 // NO MODIFICAR ESTA FUNCION SIN PERMISO
339     int i ;
340     unsigned char c ;
341     unsigned char cc ;
342     char ops_expandibles [] = "!<=|>%&/+-*" ;
343     char temp_str [256] ;
344     t_keyword *symbol ;
345
346     do {
347         c = getchar () ;
348
349         if (c == '#') { // Ignora las lineas que empiezan por #  (#define, #include)
350             do {                //      OJO que puede funcionar mal si una linea contiene #
```

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

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

```
    }

    return c ;
}


int main ()
{
    yyparse () ;
}
```