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);
                        $$.code = gen_code (temp); }
                    | '(' LOOP WHILE logical_or DO lista_sentencia ')'
                        { sprintf (temp, "begin\n\t%s\n\t%s\nrepeat", $4.code, $6.code);
                        $$.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:           aditivo                                                    { $$ = $1; }
177 |                       | '(' '<' relacional aditivo ')'
178 |                           { sprintf (temp, "%s␣%s␣<", $3.code, $4.code);
179 |                           $$.code = gen_code (temp); }
180 |                       | '(' '>' relacional aditivo ')'
181 |                           { sprintf (temp, "%s␣%s␣>", $3.code, $4.code);
182 |                           $$.code = gen_code (temp); }
183 |                       | '(' LE relacional aditivo ')'
184 |                           { sprintf (temp, "%s␣%s␣<=", $3.code, $4.code);
185 |                           $$.code = gen_code (temp); }
186 |                       | '(' GE relacional aditivo ')'
187 |                           { sprintf (temp, "%s␣%s␣>=", $3.code, $4.code);
188 |                           $$.code = gen_code (temp); }
189 |                       ;
190 | aditivo:              multiplicativo                                             { $$ = $1; }
191 |                       | '(' '+' aditivo multiplicativo ')'
192 |                           { sprintf (temp, "%s␣%s␣+", $3.code, $4.code);
193 |                           $$.code = gen_code (temp); }
194 |                       | '(' '-' aditivo multiplicativo ')'
195 |                           { sprintf (temp, "%s␣%s␣-", $3.code, $4.code);
196 |                           $$.code = gen_code (temp); }
197 |                       ;
198 | multiplicativo:       unario                                                     { $$ = $1; }
199 |                       | '(' '*' multiplicativo unario ')'
200 |                           { sprintf (temp, "%s␣%s␣*", $3.code, $4.code);
201 |                           $$.code = gen_code (temp); }
202 |                       | '(' '/' multiplicativo unario ')'
203 |                           { sprintf (temp, "%s␣%s␣/", $3.code, $4.code);
204 |                           $$.code = gen_code (temp); }
205 |                       | '(' MOD multiplicativo unario ')'
206 |                           { sprintf (temp, "%s␣%s␣mod", $3.code, $4.code);
207 |                           $$.code = gen_code (temp); }
208 |                       ;
209 | unario:               operando                                                   { $$ = $1; }
210 |                       | '(' NOT unario ')'
211 |                           { sprintf (temp, "%s␣0=", $3.code);
212 |                           $$.code = gen_code (temp); }
213 |                       | '+' operando %prec UNARY_SIGN                          { $$ = $1; }
214 |                       | '(' '-' operando %prec UNARY_SIGN ')'
215 |                           {sprintf (temp, "%s␣negate", $3.code);
216 |                           $$.code = gen_code (temp); }
217 |                       ;
218 | operando:             IDENTIF                                                    { sprintf (temp, "%s", $1.code);
```

```
                    | NUMBER                                                  $$.code = gen_code (temp); }
                                                                              { sprintf (temp, "%d", $1.value);
                    | '(' logical_or ')'                                      $$.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++ ;
```

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

```
    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 () ;

```

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

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

```
//          printf ("tEOF ") ;                                  // PARA DEPURAR
        return (0) ;
    }

    return c ;
}


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