

```

1  /* Cesar Lopez Mantecón, Paula Subías Serrano, grupo 12 */
2  /* 100472092@alumnos.uc3m.es 100472119@alumnos.uc3m.es */
3  %{
4      // SECCION 1 Declaraciones de C-Yacc
5
6  #include <stdio.h>
7  #include <ctype.h>          // declaraciones para tolower
8  #include <string.h>         // declaraciones para cadenas
9  #include <stdlib.h>         // declaraciones para exit ()
10
11 #define FF fflush(stdout);   // para forzar la impresion inmediata
12
13 typedef struct s_lista { // lista de variabbbles locales
14     char lista[1024][1024];
15     int i;
16 } t_lista;
17
18 int yylex () ;
19 int yyerror () ;
20 char *mi_malloc (int) ;
21 char *gen_code (char *) ;
22 char *int_to_string (int) ;
23 char *char_to_string (char) ;
24 char *nombre_funcion;
25 int search_local(t_lista l, char *var);
26 int insert(t_lista *l, char *var);
27 int remove_all(t_lista *l);
28
29 char temp [2048] ;
30
31 t_lista var_list;
32 // Definitions for explicit attributes
33
34 typedef struct s_attr {
35     int value ;
36     char *code ;
37 } t_attr ;
38
39 #define YYSTYPE t_attr
40
41 %}
42
43 // Definitions for explicit attributes
44
45 %token NUMBER
46 %token IDENTIF          // Identificador=variable
47 %token INTEGER          // identifica el tipo entero
48 %token STRING
49 %token MAIN             // identifica el comienzo del proc. main
50 %token WHILE            // identifica el bucle main
51 %token PRINTF           // identifica la impresion
52 %token PUTS             // identifica puts
53 %token IF
54 %token ELSE
55 %token FOR
56 %token RETURN
57
58
59 %right '=='             // minima preferencia
60 %left OR                //
61 %left AND               //
62 %left EQUAL NOTEQ       //
63 %left '<' LEQ '>' GEQ    //
64 %left '+' '-'           //
65 %left '*' '/' '%'       //
66 %left UNARY_SIGN '!'    // maxima preferencia
67
68 %%
69 // Seccion 3 Gramatica - Semantico
70
71 axioma:
72     var_globales declaracion_func MAIN { nombre_funcion = gen_code(""); }
73     '{' declaracion_variables codigo '}' { printf ("(defun main () \n%s%s)\n", $9.code, $10.code); }
74     ;
75
76 var_globales : declaracion_variables { printf ("%s", $1.code); }

```

```

77 |         ;
78 |
79 | declaracion_func: { $$$.code = ""; }
80 | | IDENTIF '(' args ')', { nombre_funcion = gen_code($1.code); }
81 | | '{', {
82 |     declaracion_variables codigo
83 |     '}', {
84 |         printf("(defun %s (%s)\n%s%s)\n", $1.code, $3.code, $7.code, $8.code);
85 |         remove_all(&var_list);
86 |     }
87 |     declaracion_func
88 |     ;
89 |
90 | args: { $$$.code = ""; }
91 | | INTEGER IDENTIF rest_args { sprintf(temp, "%s %s", $2.code, $3.code); $$$.code = gen_code (temp); }
92 | ;
93 |
94 | rest_args: { $$$.code = ""; }
95 | | ', ' INTEGER IDENTIF rest_args { sprintf(temp, " %s %s", $3.code, $4.code); $$$.code = gen_code(temp); }
96 | ;
97 |
98 | declaracion_variables: { $$$.code = ""; }
99 | | INTEGER IDENTIF
100 |     '= ' NUMBER rest_declar ';',
101 |     declaracion_variables
102 |     {
103 |         char aux[1026] = "";
104 |         if (strcmp(nombre_funcion, "")) {
105 |             sprintf(aux, "%s_", nombre_funcion); insert(&var_list, $2.code);
106 |         }
107 |         sprintf (temp, "(setq %s%s %d) %s \n%s", aux, $2.code, $4.value, $5.code, $7.code);
108 |         $$$.code = gen_code (temp);
109 |     }
110 | | INTEGER IDENTIF rest_declar ';',
111 |     declaracion_variables
112 |     {
113 |         char aux[1026] = "";
114 |         if (strcmp(nombre_funcion, "")) {
115 |             sprintf(aux, "%s_", nombre_funcion); insert(&var_list, $2.code);
116 |         }
117 |         sprintf(temp, "(setq %s%s 0) %s \n%s", aux, $2.code, $3.code, $5.code);
118 |         $$$.code = gen_code(temp);
119 |     }
120 | | INTEGER IDENTIF '[' NUMBER ']',
121 |     rest_declar_vector ';',
122 |     declaracion_variables
123 |     {
124 |         char aux[1026] = "";
125 |         if (strcmp(nombre_funcion, "")) {
126 |             sprintf(aux, "%s_", nombre_funcion); insert(&var_list, $2.code);
127 |         }
128 |         sprintf(temp, "(setq %s%s (make-array %d)) %s \n%s",aux, $2.code, $4.value, $6.code, $8.code);
129 |         $$$.code = gen_code(temp);
130 |     }
131 | ;
132 |
133 | rest_declar_vector: { $$$.code = "" ; }
134 | | ', '
135 |     IDENTIF '[' NUMBER ']',
136 |     rest_declar_vector
137 |     {
138 |         char aux[1026] = "";
139 |         if (strcmp(nombre_funcion, "")) {
140 |             sprintf(aux, "%s_", nombre_funcion); insert(&var_list, $2.code);
141 |         }
142 |         sprintf(temp, "(setq %s%s (make-array %d)) %s",aux, $2.code, $4.value, $6.code);
143 |         $$$.code = gen_code(temp);
144 |     }
145 | ;
146 |
147 | rest_declar: { $$$.code = "" ; }
148 | | ', '
149 |     IDENTIF '= ' NUMBER
150 |     rest_declar
151 |     {
152 |         char aux[1026] = "";
153 |         if (strcmp(nombre_funcion, "")) {
154 |             sprintf(aux, "%s_", nombre_funcion); insert(&var_list, $2.code);
155 |         }
156 |         sprintf (temp, "(setq %s%s %d) %s", aux,$2.code, $4.value, $5.code);

```

```

152|                                     } $.code = gen_code (temp);
153|                                     }
154|         | ',' IDENTIF rest_declar {
155|                                     char aux[1026] = "";
156|                                     if (strcmp(nombre_funcion, "")) {
157|                                         sprintf(aux, "%s_", nombre_funcion); insert(&var_list, $2.code);
158|                                     }
159|                                     sprintf (temp, "(setq %s%s 0) %s", aux,$2.code, $3.code) ;
160|                                     $$$.code = gen_code (temp) ;
161|                                     }
162|         ;
163|
164|
165|codigo:      sentencia ';' r_expr { sprintf (temp, "%s\n%s", $1.code, $3.code); $$$.code = gen_code (temp); }
166|      | WHILE '(' expression ')'
167|        '{' codigo '}'
168|        r_expr { sprintf(temp, "(loop while %s do\n%s)\n%s", $3.code, $6.code, $8.code); $$$.code = gen_code(temp); }
169|      | IF '(' expression ')'
170|        '{' codigo '}'
171|        est_else
172|        r_expr { sprintf(temp, "(if %s\n(progn %s)\n%s)\n%s", $3.code, $6.code, $8.code, $9.code); $$$.code = gen_code(
173|temp); }
174|      | FOR
175|        '('
176|        inicializar ';'
177|        expression ';'
178|        IDENTIF '=' incr_decr
179|        ')'
180|        '{' codigo '}' r_expr {
181|                                     char aux[1026] = "";
182|                                     if (search_local(var_list, $7.code)) {
183|                                         sprintf(aux, "%s_", nombre_funcion);
184|                                     }
185|                                     sprintf(temp, "%s(loop while %s do\n%s(setf %s%s%s)) \n%s", $3.code, $5.code, $12.code, aux, $7.code,
186|                                     $$$.code = gen_code(temp);
187|                                     }
188|
189|inicializar: IDENTIF '=' expression {
190|                                     char aux[1026] = "";
191|                                     if (search_local(var_list, $1.code)) {
192|                                         sprintf(aux, "%s_", nombre_funcion);
193|                                     }
194|                                     sprintf (temp, "(setf %s%s %s)", aux,$1.code, $3.code) ;
195|                                     $$$.code = gen_code (temp) ;
196|                                     }
197|
198|
199|incr_decr:      expression '+' expression { sprintf (temp, "(+ %s %s)", $1.code, $3.code); $$$.code = gen_code (temp); }
200|      | expression '-' expression { sprintf (temp, "(- %s %s)", $1.code, $3.code); $$$.code = gen_code (temp); }
201|
202|est_else:      { $$$.code = ""; }
203|      | ELSE '{' codigo '}' { sprintf(temp, "(progn %s)\n", $3.code); $$$.code = gen_code(temp); }
204|
205|
206|r_expr:      { $$$.code = ""; }
207|      | codigo { $$ = $1; }
208|
209|
210|sentencia:      IDENTIF '=' expression {
211|                                     char aux[1026] = "";
212|                                     if (search_local(var_list, $1.code)) {
213|                                         sprintf(aux, "%s_", nombre_funcion);
214|                                     }
215|                                     sprintf (temp, "(setf %s%s %s)", aux,$1.code, $3.code) ;
216|                                     $$$.code = gen_code (temp);
217|                                     }
218|
219|      | vector '=' expression {
220|                                     char aux[1026] = "";
221|                                     if (search_local(var_list, $1.code)) {
222|                                         sprintf(aux, "%s_", nombre_funcion);
223|                                     }
224|                                     sprintf (temp, "(setf %s%s %s)",aux, $1.code, $3.code) ;
225|                                     $$$.code = gen_code (temp);

```

```

225 |                                     }
226 | RETURN expression                    {sprintf(temp, "(return-from %s %s)", nombre_funcion,$2.code);
227 |                                     $$code = gen_code (temp); }
228 | PRINTF
229 | '('
230 |     STRING ',',
231 |     expression rest_print
232 |     ')'
233 |                                     {
234 |                                     sprintf (temp, "(prin1 %s) %s", $5.code, $6.code) ;
235 |                                     $$code = gen_code (temp) ;
236 |                                     }
237 | PRINTF
238 | '('
239 |     STRING ',',
240 |     STRING rest_print
241 |     ')'
242 |                                     {
243 |                                     sprintf (temp, "(prin1 \"%s\") %s", $5.code, $6.code) ;
244 |                                     $$code = gen_code (temp);
245 |                                     }
246 | PUTS
247 | '(' STRING ')'
248 |                                     {
249 |                                     sprintf (temp, "(print \"%s\")", $3.code) ;
250 |                                     $$code = gen_code (temp);
251 |                                     }
252 | IDENTIF
253 | '('
254 |     expression
255 |     rest_params
256 |     ')'
257 |                                     {
258 |                                     sprintf (temp, "(%s %s %s)", $1.code,$3.code, $4.code) ;
259 |                                     $$code = gen_code (temp);
260 |                                     }
261 | IDENTIF '(',')'
262 |                                     { sprintf (temp, "(%s)", $1.code); $$code = gen_code (temp); }
263 |
264 | rest_print:
265 | '(' expression rest_print
266 |                                     { $$code = ""; }
267 |                                     { sprintf(temp, "(prin1 %s) %s", $2.code, $3.code); $$code = gen_code(temp); }
268 | '(' STRING rest_print
269 |                                     { sprintf(temp, "(prin1 \"%s\") %s", $2.code, $3.code); $$code = gen_code(temp); }
270 |
271 | rest_params:
272 | '(' expression rest_params
273 |                                     { $$code = ""; }
274 |                                     {
275 |                                     sprintf(temp, "%s %s", $2.code, $3.code);
276 |                                     $$code = gen_code(temp);
277 |                                     }
278 |
279 | ;
280 |
281 | expression:
282 | termino
283 |                                     { $$ = $1 ; }
284 | expression '+' expression
285 |                                     { sprintf (temp, "(+ %s %s)", $1.code, $3.code); $$code = gen_code (temp); }
286 | expression '-' expression
287 |                                     { sprintf (temp, "(- %s %s)", $1.code, $3.code); $$code = gen_code (temp); }
288 | expression '*' expression
289 |                                     { sprintf (temp, "(* %s %s)", $1.code, $3.code); $$code = gen_code (temp); }
290 | expression '/' expression
291 |                                     { sprintf (temp, "(/ %s %s)", $1.code, $3.code); $$code = gen_code (temp); }
292 | expression '%' expression
293 |                                     { sprintf (temp, "(mod %s %s)", $1.code, $3.code); $$code = gen_code (temp); }
294 | expression AND expression
295 |                                     { sprintf (temp, "(and %s %s)", $1.code, $3.code); $$code = gen_code (temp); }
296 | expression OR expression
297 |                                     { sprintf (temp, "(or %s %s)", $1.code, $3.code); $$code = gen_code (temp); }
298 | expression NOTEQ expression
299 |                                     { sprintf (temp, "(/= %s %s)", $1.code, $3.code); $$code = gen_code (temp); }
300 | expression EQUAL expression
301 |                                     { sprintf (temp, "(= %s %s)", $1.code, $3.code); $$code = gen_code (temp); }
302 | expression GEQ expression
303 |                                     { sprintf (temp, "(>= %s %s)", $1.code, $3.code); $$code = gen_code (temp); }
304 | expression LEQ expression
305 |                                     { sprintf (temp, "(<= %s %s)", $1.code, $3.code); $$code = gen_code (temp); }
306 | expression '>' expression
307 |                                     { sprintf (temp, "(> %s %s)", $1.code, $3.code); $$code = gen_code (temp); }
308 | expression '<' expression
309 |                                     { sprintf (temp, "(< %s %s)", $1.code, $3.code); $$code = gen_code (temp); }
310 | IDENTIF
311 | '('expression rest_params')'
312 |                                     { sprintf (temp, "(%s %s %s)", $1.code,$3.code, $4.code); $$code = gen_code (temp); }
313 | IDENTIF '(',')'
314 |                                     { sprintf (temp, "(%s)", $1.code); $$code = gen_code (temp); }
315 |
316 | termino:
317 | operando
318 |                                     { $$ = $1 ; }
319 | '+' operando %prec UNARY_SIGN
320 |                                     { sprintf (temp, "(+ %s)", $2.code) ;
321 |                                     $$code = gen_code (temp) ; }
322 | '-' operando %prec UNARY_SIGN
323 |                                     { sprintf (temp, "(- %s)", $2.code) ;
324 |                                     $$code = gen_code (temp) ; }
325 | '!' operando %prec UNARY_SIGN
326 |                                     { sprintf(temp, "(not %s)", $2.code);
327 |                                     $$code = gen_code(temp);}
328 |
329 | ;

```

```

300|operando:      IDENTIF      {
301|                char aux[1026] = "";
302|                if (search_local(var_list, $1.code)) {
303|                    sprintf(aux, "%s_", nombre_funcion);
304|                }
305|                sprintf (temp, "%s%s", aux,$1.code) ;
306|                $$code = gen_code (temp);
307|            }
308|            |      NUMBER      {
309|                sprintf (temp, "%d", $1.value);
310|                $$code = gen_code (temp);
311|            }
312|            |      '(' expression ')',
313|            |      vector
314|            ;
315|            { $$ = $2 ; }
316|            { sprintf (temp, "%s", $1.code); $$code = gen_code (temp); }
317|
318|vector:      IDENTIF '[' expression ']' {
319|                char aux[1026] = "";
320|                if (search_local(var_list, $1.code)) {
321|                    sprintf(aux, "%s_", nombre_funcion);
322|                }
323|                sprintf (temp, "(aref %s%s %s)", aux, $1.code, $3.code) ;
324|                $$code = gen_code (temp) ;
325|            }
326|
327|%%
328|                // SECCION 4      Codigo en C
329|
330|int n_line = 1 ;
331|char *mensaje ;
332|{
333|    fprintf (stderr, "%s en la linea %d\n", mensaje, n_line) ;
334|    printf ( "\n" ); // bye
335|}
336|
337|char *int_to_string (int n)
338|{
339|    sprintf (temp, "%d", n) ;
340|    return gen_code (temp) ;
341|}
342|
343|char *char_to_string (char c)
344|{
345|    sprintf (temp, "%c", c) ;
346|    return gen_code (temp) ;
347|}
348|
349|char *my_malloc (int nbytes) // reserva n bytes de memoria dinamica
350|{
351|    char *p ;
352|    static long int nb = 0; // sirven para contabilizar la memoria
353|    static int nv = 0 ; // solicitada en total
354|
355|    p = malloc (nbytes) ;
356|    if (p == NULL) {
357|        fprintf (stderr, "No queda memoria para %d bytes mas\n", nbytes) ;
358|        fprintf (stderr, "Reservados %ld bytes en %d llamadas\n", nb, nv) ;
359|        exit (0) ;
360|    }
361|    nb += (long) nbytes ;
362|    nv++ ;
363|
364|    return p ;
365|}
366|/**
367| * Busca una variable en la lista de locales.
368| */
369|
370|int search_local(t_lista l, char *var) {
371|    for (int i = 0; i < l.i; ++i) {
372|        if (strcmp(l.lista[i], var) == 0) { // se encuentra la variable
373|            return i;
374|        }
375|    }

```

```

376 |     }
377 |     return 0;
378 | }
379 |
380 | /**
381 | * inserta una variable local en la lista
382 | */
383 |
384 | int insert(t_lista *l, char *var) {
385 |     strcpy(&(l->lista[l->i][0]), var);
386 |
387 |     ++(l->i);
388 |     return l->i;
389 | }
390 |
391 | /**
392 | * vac a la lista de variables locales
393 | */
394 |
395 | int remove_all(t_lista *l) {
396 |     l->i = 0;
397 |     return 0;
398 | }
399 |
400 | /*****
401 | /***** Seccion de Palabras Reservadas *****/
402 | /*****
403 |
404 | typedef struct s_keyword { // para las palabras reservadas de C
405 |     char *name ;
406 |     int token ;
407 | } t_keyword ;
408 |
409 | t_keyword keywords [] = { // define las palabras reservadas y los
410 |     "main",          MAIN,          // y los token asociados
411 |     "int",           INTEGER,
412 |     "puts",          PUTS,
413 |     "printf",        PRINTF,
414 |     "&&",             AND,
415 |     "||",            OR,
416 |     "!=",            NOTEQ,
417 |     "==",            EQUAL,
418 |     "<=",             LEQ,
419 |     ">=",             GEQ,
420 |     "while",         WHILE,
421 |     "if",             IF,
422 |     "else",          ELSE,
423 |     "for",            FOR,
424 |     "return",        RETURN,
425 |     NULL,             0          // para marcar el fin de la tabla
426 | } ;
427 |
428 | t_keyword *search_keyword (char *symbol_name)
429 | {
430 |     // Busca n_s en la tabla de pal. res.
431 |     // y devuelve puntero a registro (simbolo)
432 |
433 |     int i ;
434 |     t_keyword *sim ;
435 |
436 |     i = 0 ;
437 |     sim = keywords ;
438 |     while (sim [i].name != NULL) {
439 |         if (strcmp (sim [i].name, symbol_name) == 0) {
440 |             return &(sim [i]) ; // strcmp(a, b) devuelve == 0 si a==b
441 |         }
442 |         i++ ;
443 |     }
444 |     return NULL ;
445 | }
446 |
447 |
448 | /*****
449 | /***** Seccion del Analizador Lexicografico *****/
450 | /*****
451 |
452 | char *gen_code (char *name) // copia el argumento a un

```

```

453| {                                     // string en memoria dinamica
454|     char *p ;
455|     int l ;
456|
457|     l = strlen (name)+1 ;
458|     p = (char *) my_malloc (l) ;
459|     strcpy (p, name) ;
460|     return p ;
461| }
462|
463|
464|
465| int yylex ()
466| {
467|     int i ;
468|     unsigned char c ;
469|     unsigned char cc ;
470|     char ops_expandibles [] = "!<=>|%/&+-*" ;
471|     char temp_str [256] ;
472|     t_keyword *symbol ;
473|
474|     do {
475|         c = getchar () ;
476|
477|         if (c == '#') { // Ignora las lineas que empiezan por # (#define, #include)
478|             do { // OJO que puede funcionar mal si una linea contiene #
479|                 c = getchar () ;
480|             } while (c != '\n') ;
481|         }
482|
483|         if (c == '/') { // Si la linea contiene un / puede ser inicio de comentario
484|             cc = getchar () ;
485|             if (cc != '/') { // Si el siguiente char es / es un comentario, pero...
486|                 ungetc (cc, stdin) ;
487|             } else {
488|                 c = getchar () ; // ...
489|                 if (c == '@') { // Si es la secuencia //@ ==> transcribimos la linea
490|                     do { // Se trata de codigo inline (Codigo embebido en C)
491|                         c = getchar () ;

```

Listing 1: trad.y