

```

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

```

```

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

```

```

152 |         | ',' IDENTIF rest_declar {
153 |
154 |         char aux[1026] = "";
155 |         if (strcmp(nombre_funcion, "")) {
156 |             sprintf(aux, "%s_", nombre_funcion); insert(&var_list, $2.code);
157 |         }
158 |         sprintf (temp, "(setq %s%s 0) %s", aux,$2.code, $3.code) ;
159 |         $$code = gen_code (temp) ;
160 |
161 |     ;
162 |
163 | codigo:  sentencia ';' r_expr { sprintf (temp, "%s\n%s", $1.code, $3.code); $$code = gen_code (temp); }
164 | | WHILE '(' expression ')'
165 | | '{' codigo '}'
166 | | r_expr { sprintf(temp, "(loop while %s do\n%s)\n%s", $3.code, $6.code, $8.code); $$code = gen_code(temp); }
167 | | IF '(' expression ')'
168 | | '{' codigo '}'
169 | | est_else
170 | | r_expr { sprintf(temp, "(if %s\n(progn %s)\n%s)\n%s", $3.code, $6.code, $8.code, $9.code); $$code = gen_code(
temp); }
171 | | FOR
172 | | '('
173 | | inicializar ';'
174 | | expression ';'
175 | | IDENTIF '=' incr_decr
176 | | ')'
177 | | '{' codigo '}' r_expr {
178 |
179 |         char aux[1026] = "";
180 |         if (search_local(var_list, $7.code)) {
181 |             sprintf(aux, "%s_", nombre_funcion);
182 |             sprintf(temp, "%s(loop while %s do\n%s(setf %s%s%s)) \n%s", $3.code, $5.code, $12.code, aux, $7.code,
$9.code, $14.code);
183 |             $$code = gen_code(temp);
184 |         }
185 |
186 |     ;
187 | inicializar: IDENTIF '=' expression {
188 |
189 |         char aux[1026] = "";
190 |         if (search_local(var_list, $1.code)) {
191 |             sprintf(aux, "%s_", nombre_funcion);
192 |         }
193 |         sprintf (temp, "(setf %s%s %s)", aux,$1.code, $3.code) ;
194 |         $$code = gen_code (temp) ;
195 |
196 |     }
197 | incr_decr: expression '+' expression { sprintf (temp, "(+ %s %s)", $1.code, $3.code); $$code = gen_code (temp); }
198 | | expression '-' expression { sprintf (temp, "(- %s %s)", $1.code, $3.code); $$code = gen_code (temp); }
199 |
200 | est_else: { $$code = ""; }
201 | | ELSE '{' codigo '}' { sprintf(temp, "(progn %s)\n", $3.code); $$code = gen_code(temp); }
202 | ;
203 |
204 | r_expr: { $$code = ""; }
205 | | codigo { $$ = $1; }
206 | ;
207 |
208 | sentencia: IDENTIF '=' expression {
209 |
210 |         char aux[1026] = "";
211 |         if (search_local(var_list, $1.code)) {
212 |             sprintf(aux, "%s_", nombre_funcion);
213 |         }
214 |         sprintf (temp, "(setf %s%s %s)", aux,$1.code, $3.code) ;
215 |         $$code = gen_code (temp);
216 |     }
217 | | vector '=' expression {
218 |
219 |         char aux[1026] = "";
220 |         if (search_local(var_list, $1.code)) {
221 |             sprintf(aux, "%s_", nombre_funcion);
222 |         }
223 |         sprintf (temp, "(setf %s%s %s)",aux, $1.code, $3.code) ;
224 |         $$code = gen_code (temp);
225 |     }
226 | | RETURN expression {sprintf(temp, "(return-from %s %s)", nombre_funcion,$2.code);

```

```

225 |                                     $$$.code = gen_code (temp) ;}
226 |
227 | | PRINTF
228 | |   '(',
229 | |   STRING ',,'
230 | |   expression rest_print
231 | |   ')',
232 | |   {
233 | |       sprintf (temp, "(prn1 %s) %s", $5.code, $6.code) ;
234 | |       $$$.code = gen_code (temp) ;
235 | |   }
236 | | PRINTF
237 | |   '(',
238 | |   STRING ',,'
239 | |   STRING rest_print
240 | |   ')',
241 | |   {
242 | |       sprintf (temp, "(prn1 \"%s\") %s", $5.code, $6.code) ;
243 | |       $$$.code = gen_code (temp);
244 | |   }
245 | | PUTS
246 | |   '(', STRING ')',
247 | |   {
248 | |       sprintf (temp, "(print \"%s\")", $3.code) ;
249 | |       $$$.code = gen_code (temp);
250 | |   }
251 | | IDENTIF
252 | |   '(',
253 | |   expression
254 | |   rest_params
255 | |   ')',
256 | |   {
257 | |       sprintf (temp, "(%s %s %s)", $1.code,$3.code, $4.code) ;
258 | |       $$$.code = gen_code (temp);
259 | |   }
260 | | IDENTIF '(',')',
261 | |   { sprintf (temp, "(%s)", $1.code); $$$.code = gen_code (temp); }
262 | ;
263 | rest_print:
264 | |   ',' expression rest_print
265 | |   ',' STRING rest_print
266 | |   ;
267 | { $$$.code = "" ; }
268 | { sprintf(temp, "(prn1 %s) %s", $2.code, $3.code); $$$.code = gen_code(temp); }
269 | { sprintf(temp, "(prn1 \"%s\") %s", $2.code, $3.code); $$$.code = gen_code(temp); }
270 | ;
271 | rest_params:
272 | |   ',' expression rest_params
273 | |   ;
274 | { $$$.code = ""; }
275 | { sprintf(temp, "%s %s", $2.code, $3.code);
276 |   $$$.code = gen_code(temp);
277 | }
278 | ;
279 | expression:
280 | | termino
281 | | | expression '+' expression
282 | | | expression '-' expression
283 | | | expression '*' expression
284 | | | expression '/' expression
285 | | | expression '%' expression
286 | | | expression AND expression
287 | | | expression OR expression
288 | | | expression NOTEQ expression
289 | | | expression EQUAL expression
290 | | | expression GEQ expression
291 | | | expression LEQ expression
292 | | | expression '>' expression
293 | | | expression '<' expression
294 | | | IDENTIF
295 | | | '(',')expression rest_params')'
296 | | | IDENTIF '(',')',
297 | | | ;
298 | | { $$ = $1 ; }
299 | | { sprintf (temp, "(+ %s %s)", $1.code, $3.code); $$$.code = gen_code (temp); }
300 | | { sprintf (temp, "(- %s %s)", $1.code, $3.code); $$$.code = gen_code (temp); }
301 | | { sprintf (temp, "(* %s %s)", $1.code, $3.code); $$$.code = gen_code (temp); }
302 | | { sprintf (temp, "(/ %s %s)", $1.code, $3.code); $$$.code = gen_code (temp); }
303 | | { sprintf (temp, "(mod %s %s)", $1.code, $3.code); $$$.code = gen_code (temp); }
304 | | { sprintf (temp, "(and %s %s)", $1.code, $3.code); $$$.code = gen_code (temp); }
305 | | { sprintf (temp, "(or %s %s)", $1.code, $3.code); $$$.code = gen_code (temp); }
306 | | { sprintf (temp, "(/= %s %s)", $1.code, $3.code); $$$.code = gen_code (temp); }
307 | | { sprintf (temp, "(= %s %s)", $1.code, $3.code); $$$.code = gen_code (temp); }
308 | | { sprintf (temp, "(>= %s %s)", $1.code, $3.code); $$$.code = gen_code (temp); }
309 | | { sprintf (temp, "(<= %s %s)", $1.code, $3.code); $$$.code = gen_code (temp); }
310 | | { sprintf (temp, "(> %s %s)", $1.code, $3.code); $$$.code = gen_code (temp); }
311 | | { sprintf (temp, "(< %s %s)", $1.code, $3.code); $$$.code = gen_code (temp); }
312 | | { sprintf (temp, "(%s %s %s)", $1.code,$3.code, $4.code); $$$.code = gen_code (temp); }
313 | | { sprintf (temp, "(%s)", $1.code); $$$.code = gen_code (temp); }
314 | ;
315 | termino:
316 | | operando
317 | | | '+' operando %prec UNARY_SIGN
318 | | | '-' operando %prec UNARY_SIGN
319 | | | '!' operando %prec UNARY_SIGN
320 | | | ;
321 | | { $$ = $1 ; }
322 | | { sprintf (temp, "(+ %s)", $2.code) ;
323 | |   $$$.code = gen_code (temp) ; }
324 | | { sprintf (temp, "(- %s)", $2.code) ;
325 | |   $$$.code = gen_code (temp) ; }
326 | | { sprintf(temp, "(not %s)", $2.code);
327 | |   $$$.code = gen_code(temp);}
328 | ;
329 | operando:
330 | | IDENTIF
331 | | | {
332 | | | | char aux[1026] = "";

```

```

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

```

```

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

```

```

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

```

Listing 1: trad.y