

```

1  /* Cesar Lopez Mantecon, Paula Subias 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 { // tabla de smbolos
14     char lista[1024][1024];
15     int values[1024];
16     int i;
17 } t_lista;
18
19 int yylex () ;
20 int yyerror () ;
21 char *mi_malloc (int) ;
22 char *gen_code (char *) ;
23 char *int_to_string (int) ;
24 char *char_to_string (char) ;
25
26 // mi tabla de simbolos
27 int search_local(t_lista l, char *var);
28 int insert(t_lista *l, char *var, int n);
29 int remove_all(t_lista *l);
30
31 char temp [2*2048] ;
32 char nombre_funcion[1024];
33
34 t_lista argumentos;
35 t_lista var_local;
36 t_lista vec_local;
37
38 // Definitions for explicit attributes
39
40 typedef struct s_attr {
41     int value ;
42     char *code ;
43 } t_attr ;
44
45 #define YYSTYPE t_attr
46
47 %}
48
49 // Definitions for explicit attributes
50
51 %token NUMBER
52 %token IDENTIF          // Identificador=variable
53 %token STRING
54 %token MAIN             // identifica el comienzo del proc. main
55 %token WHILE            // identifica el bucle main
56 %token PRINT            // identifica la impresion
57 %token SETQ
58 %token DEFUN
59 %token PRIN1
60 %token SETF
61 %token DO
62 %token LOOP
63 %token IF
64 %token PROGN
65 %token NOT
66 %token RETURN
67 %token FROM
68 %token MAKE
69 %token ARRAY
70 %token AREF
71
72
73 %right '=='             // minima preferencia
74 %left OR                //
75 %left AND               //

```

```

76| %left EQUAL NOTEQ          //
77| %left '<' LEQ '>' GEQ      //
78| %left '+' '-'             //
79| %left '*' '/' MOD         //
80| %left UNARY_SIGN          // maxima preferencia
81|
82| %%                          // Seccion 3 Gramatica - Semantico
83|
84| axioma:      '(' bloque ')' codigo { printf ("\n"); }
85| ;
86|
87| codigo:      '(' bloque ')' codigo { sprintf(temp, "%s\n%s", $2.code, $4.code); $$code = gen_code(temp); }
88| | /* lambda */ { $$code = ""; }
89| ;
90|
91| bloque:      sentencia { $$ = $1 ; }
92| | declaracion { sprintf(temp, "%s", $1.code); $$code = gen_code(temp); }
93| | DEFUN
94| | IDENTIF
95| | '(' func_arg ')'
96| | codigo
97| {
98|     char asign_args[4*2048];
99|     char asign_aux[2*2048];
100|     strcpy(asign_args, "");
101|     int i;
102|     for (i = 0; i < argumentos.i ; ++i) {
103|         sprintf(asign_aux, "arg_%s_%s !\n", nombre_funcion, argumentos.lista[i]);
104|         strcat(asign_args, asign_aux);
105|     } // asignacion de argumentos
106|
107|     char variables_locales[4*2048];
108|     strcpy(variables_locales, "");
109|     for (i = 0; i < var_local.i ; ++i) {
110|         sprintf(asign_aux, "variable %s\n", var_local.lista[i]);
111|         strcat(variables_locales, asign_aux);
112|     } // declaracion de var_locales
113|
114|     char vec_locales[4*2048];
115|     strcpy(vec_locales, "");
116|     for (i = 0; i < vec_local.i ; ++i) {
117|         sprintf(asign_aux, "variable %s %d cells allot\n", vec_local.lista[i], var_local.values[i]);
118|         strcat(vec_locales, asign_aux);
119|     } // declaracion de vec_locales
120|
121|     char asign_local[2048];
122|     strcpy(asign_local, "");
123|     for (i = 0; i < var_local.i ; ++i) {
124|         sprintf(asign_aux, "%i %s !\n", var_local.values[i], var_local.lista[i]);
125|         strcat(asign_local, asign_aux);
126|     } // asignacion de var locales
127|
128|     printf("%s%s%s: %s\n%s%s%s;\n", $5.code, variables_locales, vec_locales, $2.code, asign_args, asign_local, $7.code);
129|     $$code = gen_code("");
130|     strcpy(nombre_funcion, "");
131|     remove_all(&argumentos);
132|     remove_all(&var_local);
133|     remove_all(&vec_local);
134| }
135|
136| | LOOP
137| | WHILE
138| | expresion
139| | DO
140| | codigo
141| {
142|     sprintf(temp, "BEGIN\n %s WHILE\n %s REPEAT\n", $3.code, $5.code);
143|     $$code = gen_code(temp);
144| }
145|
146| | IF
147| | expresion
148| | '(' PROGN codigo ')'
149| | else
150| {
151|     sprintf(temp, "%s IF\n %s %s THEN\n", $2.code, $5.code, $7.code);
152|     $$code = gen_code(temp);
153| }

```

```

151 |
152 |
153 |
154 | func_arg:      /* lambda */      { $$code = ""; }
155 | | IDENTIF func_arg      {
156 |     sprintf(temp, "variable arg_%s_%s\n%s", nombre_funcion, $1.code, $2.code);
157 |     $$code = gen_code(temp);
158 |     insert(&argumentos, $1.code, 0); // insertar en argumentos
159 | }
160 |
161 |
162 | else:         /* lambda */      { $$code = ""; }
163 | | '(', PROGN codigo')',      { sprintf(temp, "ELSE %s", $3.code); $$code = gen_code(temp); }
164 | ;
165 |
166 | sentencia:     SETF IDENTIF expresion {
167 |     char aux[2048] = "";
168 |     if (strcmp(nombre_funcion, "")) {
169 |         if (search_local(var_local, $2.code)) {};
170 |         else if (search_local(argumentos, $2.code))
171 |             {sprintf(aux, "arg_%s_", nombre_funcion);}
172 |     }
173 |     sprintf (temp, "%s %s%s !\n", $3.code, aux,$2.code) ;
174 |     $$code = gen_code (temp) ;
175 | }
176 | | SETF vector expresion {
177 |
178 |     sprintf (temp, "%s %s !\n", $3.code,$2.code) ;
179 |     $$code = gen_code (temp) ;
180 | }
181 | | PRINT STRING      { sprintf(temp, ".\n %s\n", $2.code); $$code = gen_code(temp); }
182 | | PRIN1 prin1_arg    { sprintf(temp, "%s", $2.code); $$code = gen_code(temp); }
183 | | RETURN '-', FROM
184 | | IDENTIF expresion { sprintf(temp, "%s\n exit", $5.code); $$code = gen_code(temp); }
185 | | funcion            {
186 |     // si estoy en el scope de una funcion
187 |     if (strcmp(nombre_funcion, "")) {
188 |         $$ = $1;
189 |     } else {
190 |         // caso @ (funcion)
191 |         printf("%s\n", $1.code);
192 |     }
193 | }
194 | ;
195 |
196 |
197 | prin1_arg:     expresion      { sprintf(temp, "%s .", $1.code); $$code = gen_code(temp); }
198 | | STRING      { sprintf(temp, ".\n %s\n", $1.code); $$code = gen_code(temp); }
199 | ;
200 |
201 | expresion:     NUMBER      { sprintf (temp, "%d", $1.value) ;$$code = gen_code(temp); }
202 | | IDENTIF      {
203 |     char aux[2048] = "";
204 |     if (strcmp(nombre_funcion, "")) {
205 |         if (search_local(var_local, $1.code)) {};
206 |         else if (search_local(argumentos, $1.code)) {
207 |             sprintf(aux, "arg_%s_", nombre_funcion);
208 |         }
209 |     }
210 |     sprintf (temp, "%s%s @", aux,$1.code) ;
211 |     $$code = gen_code(temp);
212 | }
213 | | vector      { sprintf(temp, "%s @", $1.code); $$code = gen_code(temp); }
214 | | '(', operacion ')',      { sprintf(temp, "%s", $2.code); $$code = gen_code(temp); }
215 | | '(', NOT expresion ')', { sprintf(temp, "%s 0=", $3.code); $$code = gen_code(temp); }
216 | | '(', '+', expresion ')', { sprintf(temp, "%s", $3.code); $$code = gen_code(temp); }
217 | | '(', '-', expresion ')', { sprintf(temp, "0 %s -", $3.code); $$code = gen_code(temp); }
218 | ;
219 |
220 | vector: '(', AREF IDENTIF expresion ')', {
221 |     sprintf (temp, "%s %s cells +", $3.code, $4.code);
222 |     $$code = gen_code(temp);
223 | }
224 |
225 | funcion:      IDENTIF args      {

```

```

226|                                     if (strcmp(nombre_funcion, $1.code) == 0) {
227|                                         sprintf(temp, "%s %s", $2.code, "RECURSE");
228|                                     } else {
229|                                         sprintf(temp, "%s %s", $2.code, $1.code);
230|                                         $$$.code = gen_code(temp);
231|                                     }
232|                                     $$$$.code = gen_code(temp);
233|
234|                                     ;
235|
236| args:      /* lambda */
237|           | expression args
238|           ;
239|
240| operacion:  '+', expression expression { sprintf(temp, "%s %s +", $2.code, $3.code); $$$$.code = gen_code(temp); }
241|           '- ', expression expression { sprintf(temp, "%s %s -", $2.code, $3.code); $$$$.code = gen_code(temp); }
242|           '*', expression expression { sprintf(temp, "%s %s *", $2.code, $3.code); $$$$.code = gen_code(temp); }
243|           '/', expression expression { sprintf(temp, "%s %s /", $2.code, $3.code); $$$$.code = gen_code(temp); }
244|           MOD expression expression { sprintf(temp, "%s %s mod", $2.code, $3.code); $$$$.code = gen_code(temp); }
245|           '<', expression expression { sprintf(temp, "%s %s <", $2.code, $3.code); $$$$.code = gen_code(temp); }
246|           '>', expression expression { sprintf(temp, "%s %s >", $2.code, $3.code); $$$$.code = gen_code(temp); }
247|           '=', expression expression { sprintf(temp, "%s %s =", $2.code, $3.code); $$$$.code = gen_code(temp); }
248|           AND expression expression { sprintf(temp, "%s %s and", $2.code, $3.code); $$$$.code = gen_code(temp); }
249|           OR expression expression { sprintf(temp, "%s %s or", $2.code, $3.code); $$$$.code = gen_code(temp); }
250|           GEQ expression expression { sprintf(temp, "%s %s >=", $2.code, $3.code); $$$$.code = gen_code(temp); }
251|           LEQ expression expression { sprintf(temp, "%s %s <=", $2.code, $3.code); $$$$.code = gen_code(temp); }
252|           NOTEQ expression expression { sprintf(temp, "%s %s = 0=", $2.code, $3.code); $$$$.code = gen_code(temp); }
253|           | funcion { $$$ = $1 ; }
254|           ;
255|
256| declaracion:  SETQ IDENTIF NUMBER {
257|                                     if (strcmp(nombre_funcion, "")) {
258|                                         insert(&var_local, $2.code, $3.value);
259|                                         $$$$.code = gen_code("");
260|                                     } else {
261|                                         printf ( "variable %s\n%d %s !\n", $2.code, $3.value, $2.code) ;
262|                                         $$$$.code = gen_code (temp) ;
263|                                     }
264|                                     }
265|           | SETQ IDENTIF
266|             '(',
267|             MAKE '-' ARRAY
268|             NUMBER
269|             ')',
270|           {
271|               if (strcmp(nombre_funcion, "")) {
272|                   insert(&vec_local, $2.code, $7.value);
273|                   $$$$.code = gen_code("");
274|               } else {
275|                   printf ( "variable %s %d cells allot\n", $2.code, $7.value);
276|                   $$$$.code = gen_code (temp);
277|               }
278|           }
279|           ;
280|
281| %%                                     // SECCION 4     Codigo en C
282|
283| int n_line = 1 ;
284|
285| int yyerror (mensaje)
286| char *mensaje ;
287| {
288|     fprintf (stderr, "%s en la linea %d\n", mensaje, n_line) ;
289|     printf ( "\n" ) ; // bye
290| }
291|
292| char *int_to_string (int n)
293| {
294|     sprintf (temp, "%d", n) ;
295|     return gen_code (temp) ;
296| }
297|
298| char *char_to_string (char c)
299| {
300|     sprintf (temp, "%c", c) ;

```

```

301 |     return gen_code (temp) ;
302 | }
303 |
304 | char *my_malloc (int nbytes)          // reserva n bytes de memoria dinamica
305 | {
306 |     char *p ;
307 |     static long int nb = 0 ;          // sirven para contabilizar la memoria
308 |     static int nv = 0 ;              // solicitada en total
309 |
310 |     p = malloc (nbytes) ;
311 |     if (p == NULL) {
312 |         fprintf (stderr, "No queda memoria para %d bytes mas\n", nbytes) ;
313 |         fprintf (stderr, "Reservados %ld bytes en %d llamadas\n", nb, nv) ;
314 |         exit (0) ;
315 |     }
316 |     nb += (long) nbytes ;
317 |     nv++ ;
318 |     return p ;
319 | }
320 |
321 |
322 | /*****
323 | ***** Seccion de tabla de s mbolos *****/
324 | /*****
325 | *****/
326 | /**
327 | * Busca un nombre en la tabla
328 | *
329 | */
330 | int search_local(t_lista l, char *var) {
331 |     for (int i = 0; i < l.i; ++i) {
332 |         if (strcmp(l.lista[i], var) == 0) { // se encuentra la variable
333 |             return i;
334 |         }
335 |     }
336 |     return 0;
337 | }
338 |
339 |
340 | /**
341 | * inserta un nombre en la lista
342 | *
343 | */
344 | int insert(t_lista *l, char *var, int n) {
345 |     strcpy(&(l->lista[l->i][0]), var);
346 |     l->values[l->i] = n;
347 |
348 |     ++(l->i);
349 |     return l->i;
350 | }
351 |
352 | /**
353 | * vacia la lista
354 | */
355 | int remove_all(t_lista *l) {
356 |     l->i = 0;
357 |     return 0;
358 | }
359 |
360 |
361 |
362 | /*****
363 | ***** Seccion de Palabras Reservadas *****/
364 | /*****
365 | *****/
366 | typedef struct s_keyword { // para las palabras reservadas de C
367 |     char *name ;
368 |     int token ;
369 | } t_keyword ;
370 |
371 | t_keyword keywords [] = { // define las palabras reservadas y los
372 |     // "main",          MAIN,          // y los token asociados
373 |     "print",           PRINT,
374 |     "mod",             MOD,
375 |     "and",             AND,
376 |     "or",              OR,
377 |     "/=",              NOTEQ,

```

```

378 |     "<=",          LEQ,
379 |     ">=",          GEQ,
380 |     "setq",        SETQ,
381 |     "not",         NOT,
382 |     "defun",       DEFUN,
383 |     "prin1",       PRIN1,
384 |     "setf",        SETF,
385 |     "loop",        LOOP,
386 |     "do",          DO,
387 |     "while",       WHILE,
388 |     "if",          IF,
389 |     "progn",       PROGN,
390 |     "return",      RETURN,
391 |     "from",        FROM,
392 |     "make",        MAKE,
393 |     "array",       ARRAY,
394 |     "aref",        AREF,
395 |     NULL,          0                // para marcar el fin de la tabla
396 | } ;
397 |
398 | t_keyword *search_keyword (char *symbol_name)
399 | {
400 |     // Busca n_s en la tabla de pal. res.
401 |     // y devuelve puntero a registro (simbolo)
402 |     int i ;
403 |     t_keyword *sim ;
404 |     i = 0 ;
405 |     sim = keywords ;
406 |     while (sim [i].name != NULL) {
407 |         if (strcmp (sim [i].name, symbol_name) == 0) {
408 |             // strcmp(a, b) devuelve == 0 si a==b
409 |             return &(sim [i]) ;
410 |         }
411 |         i++ ;
412 |     }
413 |     return NULL ;
414 | }
415 |
416 |
417 |
418 | /*****
419 | /***** Seccion del Analizador Lexicografico *****/
420 | /*****
421 |
422 | char *gen_code (char *name)        // copia el argumento a un
423 | {                                  // string en memoria dinamica
424 |     char *p ;
425 |     int l ;
426 |
427 |     l = strlen (name)+1 ;
428 |     p = (char *) my_malloc (l) ;
429 |     strcpy (p, name) ;
430 |
431 |     return p ;
432 | }
433 |
434 |
435 | int yylex ()
436 | {
437 |     int i ;
438 |     unsigned char c ;
439 |     unsigned char cc ;
440 |     char ops_expandibles [] = "!<=>|%/&+-*" ;
441 |     char temp_str [256] ;
442 |     t_keyword *symbol ;
443 |
444 |     do {
445 |         c = getchar () ;
446 |         if (c == '#') { // Ignora las lineas que empiezan por # (#define, #include)
447 |             do { // OJO que puede funcionar mal si una linea contiene #
448 |                 c = getchar () ;
449 |             } while (c != '\n') ;
450 |         }
451 |
452 |         if (c == '/') { // Si la linea contiene un / puede ser inicio de comentario
453 |             cc = getchar () ;
454 |             if (cc != '/') { // Si el siguiente char es / es un comentario, pero...

```

```

455|         ungetc (cc, stdin) ;
456|     } else {
457|         c = getchar () ; // ...
458|         if (c == '\0') { // Si es la secuencia //@ ==> transcribimos la linea
459|             do { // Se trata de codigo inline (Codigo embebido en C)
460|                 c = getchar () ;
461|                 putchar (c) ;
462|             } while (c != '\n') ;
463|         } else { // ==> comentario, ignorar la linea
464|             while (c != '\n') {
465|                 c = getchar () ;
466|             }
467|         }
468|     }
469| } else if (c == '\\') c = getchar () ;
470|
471| if (c == '\n')
472|     n_line++ ;
473|
474| } while (c == ' ' || c == '\n' || c == 10 || c == 13 || c == '\t') ;
475|
476| if (c == '\"') {
477|     i = 0 ;
478|     do {
479|         c = getchar () ;
480|         temp_str [i++] = c ;
481|     } while (c != '\"' && i < 255) ;
482|     if (i == 256) {
483|         printf ("AVISO: string con mas de 255 caracteres en linea %d\n", n_line) ;
484|         // habria que leer hasta el siguiente " , pero, y si falta?
485|         temp_str [--i] = '\0' ;
486|         yylval.code = gen_code (temp_str) ;
487|         return (STRING) ;
488|     }
489| }
490|
491| if (c == '.' || (c >= '0' && c <= '9')) {
492|     ungetc (c, stdin) ;

```

Listing 1: back.y