

```

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

```

```

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

```

```

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

```

```

226|         } else {
227|             sprintf(temp, "%s %s", $2.code, $1.code);
228|             $$$.code = gen_code(temp);
229|         }
230|         $$$.code = gen_code(temp);
231|     }
232|
233|
234|args:      /* lambda */
235|          | expression args
236|          ;
237|
238|operacion:  '+,' expression expression { sprintf(temp, "%s %s +", $2.code, $3.code); $$$.code = gen_code(temp); }
239|          | '-,' expression expression { sprintf(temp, "%s %s -", $2.code, $3.code); $$$.code = gen_code(temp); }
240|          | '*,' 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|          | MOD expression expression { sprintf(temp, "%s %s mod", $2.code, $3.code); $$$.code = gen_code(temp); }
243|          | '<,' expression expression { sprintf(temp, "%s %s <", $2.code, $3.code); $$$.code = gen_code(temp); }
244|          | '>,' expression expression { sprintf(temp, "%s %s >", $2.code, $3.code); $$$.code = gen_code(temp); }
245|          | '=,' expression expression { sprintf(temp, "%s %s =", $2.code, $3.code); $$$.code = gen_code(temp); }
246|          | AND expression expression { sprintf(temp, "%s %s and", $2.code, $3.code); $$$.code = gen_code(temp); }
247|          | OR expression expression { sprintf(temp, "%s %s or", $2.code, $3.code); $$$.code = gen_code(temp); }
248|          | GEQ expression expression { sprintf(temp, "%s %s >=", $2.code, $3.code); $$$.code = gen_code(temp); }
249|          | LEQ expression expression { sprintf(temp, "%s %s <=", $2.code, $3.code); $$$.code = gen_code(temp); }
250|          | NOTEQ expression expression { sprintf(temp, "%s %s = 0=", $2.code, $3.code); $$$.code = gen_code(temp); }
251|          | funcion
252|          ;
253|
254|declaracion:  SETQ IDENTIF NUMBER {
255|              if (strcmp(nombre_funcion, "")) {
256|                  insert(&var_local, $2.code, $3.value);
257|                  $$$.code = gen_code("");
258|              } else {
259|                  printf ( "variable %s\n%d %s !\n", $2.code, $3.value, $2.code) ;
260|                  $$$.code = gen_code (temp) ;
261|              }
262|          }
263|          | SETQ IDENTIF
264|            '(',
265|            MAKE '-,' ARRAY
266|            NUMBER
267|            ')',
268|          {
269|              if (strcmp(nombre_funcion, "")) {
270|                  insert(&vec_local, $2.code, $7.value);
271|                  $$$.code = gen_code("");
272|              } else {
273|                  printf ( "variable %s %d cells allot\n", $2.code, $7.value);
274|                  $$$.code = gen_code (temp);
275|              }
276|          }
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|}

```

```

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

```

```

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

```

```

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

```

Listing 1: back.y