

## 目次

1	iscm0.h	2
2	main.c	3
3	eval.c	8

# 1 iscm0.h

```
1  /* iscm0 -- a scheme like language.
2  * Copyright (c) UNNO Hideyuki <unno.hideyuki@nifty.com>.
3  * License: The MIT License (see COPYING).
4  */
5  #define ISCM_VERSION "0.1.1"
6
7  enum {INT_OBJ, STR_OBJ, ID_OBJ, PAIR_OBJ,
8        FUNC, BUILTIN_FUNC, MACRO /* ,LEGACY_MACRO */};
9
10 typedef struct S_OBJ {
11     int ty;
12     union {
13         char* str;
14         long ival;
15         struct S_OBJ* car;
16         void* clos;
17     } u;
18     struct S_OBJ* cdr;
19     long lineno;
20 } OBJ;
21
22 typedef struct S_EXPS {
23     OBJ* pexp;
24     struct S_EXPS* next;
25 } EXPS;
26
27 void* mycalloc(int);
28
29 void append_exp(OBJ *const);
30 OBJ* create_pair(OBJ *const);
31 OBJ* append_to_list(OBJ *const, OBJ *const);
32 OBJ* nil();
33 OBJ* set_cdr(OBJ *const, OBJ *const);
34 void setlineno(OBJ *const pobj, int lineno);
35 void* create_obj_int(long);
36 void* create_obj_str(const char *const);
37 void* create_obj_id(const char *const);
38
39 void print_obj(const OBJ*);
40 OBJ* eval(OBJ* exp, void* env);
```

## 2 main.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include "iscm0.h"
5
6 extern int lineno;
7 extern int yyparse();
8 static EXPS* exps = NULL;
9
10 int
11 main()
12 {
13     EXPS *p;
14
15     lineno = 1;
16     if(yyparse() == 0){
17         p = exps;
18         while (p){
19             eval(p->pexp, NULL);
20             p = p->next;
21         }
22     }
23     return 0;
24 }
25
26 void*
27 mycalloc(int size)
28 {
29     void* p;
30     if((p = calloc(1, size)) == NULL){
31         fprintf(stderr, "Fatal: malloc failed.\n");
32         exit(1);
33     }
34     return p;
35 }
36
37 static OBJ*
38 new_obj()
39 {
40     return mycalloc(sizeof(OBJ));
41 }
42
43 void
44 append_exp(OBJ *const pexp)
45 {
```

```

46  EXPS *p, *node;
47
48  node = mycalloc(sizeof(EXPS));
49  node->pexp = pexp;
50
51  if (exps){
52      p = exps;
53      while (p->next){
54          p = p->next;
55      }
56      p->next = node;
57  } else {
58      exps = node;
59  }
60 }
61
62 OBJ*
63 create_pair(OBJ *const p)
64 {
65     OBJ* ret;
66     ret = new_obj();
67     ret->ty = PAIR_OBJ;
68     ret->u.car = p;
69     ret->cdr = NULL;
70     return ret;
71 }
72
73 OBJ*
74 append_to_list(OBJ *const plist, OBJ *const pobj)
75 {
76     OBJ* p;
77     p = plist;
78     while(p->cdr){
79         p = p->cdr;
80     }
81     p->cdr = create_pair(pobj);
82     return plist;
83 }
84
85 OBJ*
86 nil()
87 {
88     return NULL;
89 }
90
91 OBJ*
92 set_cdr(OBJ *const plist, OBJ *const pobj)
93 {
94     OBJ *p;

```

```

95  p = plist;
96  while(p->cdr){
97      p = p->cdr;
98  }
99  p->cdr = pobj;
100 return plist;
101 }
102
103 void
104 setlineno(OBJ *const pobj, int lineno)
105 {
106     pobj->lineno = lineno;
107 }
108
109 void*
110 create_obj_int(long ival)
111 {
112     OBJ* p;
113     p = new_obj();
114     p->ty = INT_OBJ;
115     p->u.ival = ival;
116     return p;
117 }
118
119 void*
120 create_obj_str(const char *const str)
121 {
122     OBJ* p;
123     const char* ps;
124     char *pd;
125
126     p = new_obj();
127     p->ty = STR_OBJ;
128     p->u.str = mycalloc(strlen(str)); /* enough */
129
130     ps = &str[1]; /* skip the first " */
131     pd = p->u.str;
132
133     while(*ps != '\0' && *ps != ''){
134         if(*ps == '\\\ ' && *(ps + 1) == 'n'){
135             *pd++ = '\n';
136             ps += 2;
137         } else {
138             *pd++ = *ps++;
139         }
140     }
141
142     *pd = '\0';
143

```

```

144     return p;
145 }
146
147 void*
148 create_obj_id(const char *const str)
149 {
150     OBJ* p;
151
152     p = new_obj();
153     p->ty = ID_OBJ;
154     p->u.str = mycalloc(strlen(str) + 1);
155     strcpy(p->u.str, str);
156
157     return p;
158 }
159
160 static void
161 print_obj_sub(const OBJ* obj)
162 {
163     if(obj){
164         switch(obj->ty){
165             case INT_OBJ:
166                 printf("%ld", obj->u.ival);
167                 break;
168             case STR_OBJ:
169                 printf("\"%s\"", obj->u.str);
170                 break;
171             case ID_OBJ:
172                 printf("%s", obj->u.str);
173                 break;
174             case PAIR_OBJ:
175                 printf("(");
176                 print_obj_sub(obj->u.car);
177                 printf("_");
178                 print_obj_sub(obj->cdr);
179                 printf(")");
180                 break;
181             case FUNC:
182                 printf("#<closure>");
183                 break;
184             case BUILTIN_FUNC:
185                 printf("#<built-in_%ld>", obj->u.ival);
186         }
187     }else{
188         printf("()");
189     }
190 }
191
192 void

```

```
193 print_obj(const OBJ* obj)
194 {
195     print_obj_sub(obj);
196 }
```

### 3 eval.c

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include "iscm0.h"
5
6 static OBJ* funccall(OBJ* op, OBJ* params, void* env, long lno);
7
8 /**** environment ***/
9
10 typedef struct S_KVPAIR {
11     const char* key;
12     OBJ* val;
13     struct S_KVPAIR* next;
14 } KEY_VALUE_PAIR;
15
16 typedef struct S_SCM_ENV {
17     KEY_VALUE_PAIR* kvs;
18     struct S_SCM_ENV* parent;
19 } SCM_ENV;
20
21 static SCM_ENV root_env = {NULL, NULL};
22
23 static int
24 lookup(const SCM_ENV* env, const char *const key, OBJ** pp)
25 {
26     KEY_VALUE_PAIR *kvs;
27
28     env = env ? env : &root_env;
29
30     kvs = env->kvs;
31     while (kvs){
32         if (kvs->key && key && strcmp(kvs->key, key) == 0){
33             *pp = kvs->val;
34             return 1;
35         }
36         kvs = kvs->next;
37     }
38
39     return env->parent && lookup(env->parent, key, pp);
40 }
41
42 static void
43 define_var(SCM_ENV* env, const char *const key, OBJ* exp)
44 {
45     KEY_VALUE_PAIR *kv;
```



```

46
47     env = env ? env : &root_env;
48
49     kv = mycalloc(sizeof(KEY_VALUE_PAIR));
50     kv->key = key;
51     kv->val = exp;
52     kv->next = env->kvs;
53     env->kvs = kv;
54 }
55
56 static void
57 set_var(SCM_ENV* env, const char *const key, OBJ* exp)
58 {
59     KEY_VALUE_PAIR *kvs;
60
61     env = env ? env : &root_env;
62
63     kvs = env->kvs;
64     while (kvs){
65         if (kvs->key && key && strcmp(kvs->key, key) == 0){
66             kvs->val = exp;
67             return;
68         }
69         kvs = kvs->next;
70     }
71
72     if (env->parent){
73         set_var(env->parent, key, exp);
74     } else {
75         fprintf(stderr, "Error: Unbound variable: %s\n", key);
76         exit(1);
77     }
78 }
79
80 /**** built-in functions ****/
81
82 static void
83 pair_chk(const OBJ *const exp, const char *const opname, long lno)
84 {
85     if (!exp || exp->ty != PAIR_OBJ){
86         print_obj(exp);
87         if (lno){
88             fprintf(stderr, "line %ld, %s needs a pair.\n", lno, opname);
89         } else {
90             fprintf(stderr, "Error: %s needs a pair.\n", opname);
91         }
92         exit(1);
93     }
94 }

```

```

95
96 static OBJ*
97 car(const OBJ *const exp, long lno)
98 {
99     pair_chk(exp, "car", lno ? lno : exp->lineno);
100     return exp->u.car;
101 }
102
103 static OBJ*
104 cdr(const OBJ *const exp, long lno)
105 {
106     pair_chk(exp, "cdr", lno ? lno : exp->lineno);
107     return exp->cdr;
108 }
109
110 static int
111 id_comp(const OBJ *const exp, const char *const str)
112 {
113     return exp->ty == ID_OBJ && strcmp(exp->u.str, str) == 0;
114 }
115
116 enum {CAR, CDR, DEFINE, LAMBDA, SET, COND, QUOTE,
117       ADD, SUB, MUL, DIV, EQ, GT, DISP,
118       DEFSYN,
119       VER, not_found};
120
121 static int
122 check_builtin_op(OBJ* op)
123 {
124     int ret = not_found;
125
126     if (id_comp(op, "car")) { ret = CAR; }
127     else if (id_comp(op, "cdr")) { ret = CDR; }
128     else if (id_comp(op, "define")) { ret = DEFINE; }
129     else if (id_comp(op, "lambda")) { ret = LAMBDA; }
130     else if (id_comp(op, "set!")) { ret = SET; }
131     else if (id_comp(op, "cond")) { ret = COND; }
132     else if (id_comp(op, "quote")) { ret = QUOTE; }
133     else if (id_comp(op, "+")) { ret = ADD; }
134     else if (id_comp(op, "-")) { ret = SUB; }
135     else if (id_comp(op, "*")) { ret = MUL; }
136     else if (id_comp(op, "/")) { ret = DIV; }
137     else if (id_comp(op, "=")) { ret = EQ; }
138     else if (id_comp(op, ">")) { ret = GT; }
139     else if (id_comp(op, "display")) { ret = DISP; }
140     else if (id_comp(op, "define-syntax")) { ret = DEFSYN; }
141     else if (id_comp(op, "iscm-version")) { ret = VER; }
142
143     return ret;

```

```

144 }
145
146 OBJ*
147 disp(OBJ* params, void* env, long lno)
148 {
149     OBJ* x;
150     x = eval(car(params, lno), env);
151     if (!x){ printf("()"); }
152     else if (x->ty == INT_OBJ){ printf("%ld", x->u.ival); }
153     else if (x->ty == STR_OBJ){ printf(x->u.str); }
154     else { print_obj(x); }
155     return NULL;
156 }
157
158 static void
159 int_chk(const OBJ *const exp)
160 {
161     if (!exp || exp->ty != INT_OBJ){
162         print_obj(exp);
163         fprintf(stderr, "Arithmetic_operator_needs_integer(s).\n");
164         exit(1);
165     }
166 }
167
168 OBJ*
169 four_arithmetic(int op, OBJ* params, void* env, long lno)
170 {
171     OBJ *ret, *x;
172
173     int_chk(x = eval(car(params, lno), env));
174     ret = create_obj_int(x->u.ival);
175
176     params = cdr(params, lno);
177
178     while (params){
179         int_chk(x = eval(car(params, lno), env));
180
181         switch(op){
182             case ADD:
183                 ret->u.ival += x->u.ival; break;
184             case SUB:
185                 ret->u.ival -= x->u.ival; break;
186             case MUL:
187                 ret->u.ival *= x->u.ival; break;
188             case DIV:
189                 if (x->u.ival == 0){
190                     fprintf(stderr, "line_%ld:divided_by_zero.\n", lno);
191                     exit(1);
192                 }

```

```

193     ret->u.ival /= x->u.ival;
194 }
195
196     params = cdr(params, lno);
197 }
198
199     return ret;
200 }
201
202 static OBJ*
203 def(OBJ* params, void* env, long lno)
204 {
205     OBJ *id, *exp;
206     id = car(params, 0);
207
208     if (id && id->ty == ID_OBJ){
209         exp = eval(car(cdr(params, 0), 0), env);
210         define_var(env, id->u.str, exp);
211     } else {
212         fprintf(stderr, "line_%ld: syntax error in a variable definition.\n", lno);
213         exit(1);
214     }
215
216     return id;
217 }
218
219 static OBJ*
220 set(OBJ* params, void* env, long lno)
221 {
222     OBJ *id, *exp;
223     id = car(params, 0);
224     exp = eval(car(cdr(params, 0), 0), env);
225
226     if (id && id->ty == ID_OBJ){
227         set_var(env, id->u.str, exp);
228     } else {
229         fprintf(stderr, "line_%ld: syntax error in an assignment.\n", lno);
230         exit(1);
231     }
232
233     return exp;
234 }
235
236 static OBJ*
237 eq(const OBJ* params, void* env, long lno)
238 {
239     OBJ *x, *y;
240
241     int_chk(x = eval(car(params, lno), env));

```

```

242 params = cdr(params, lno);
243
244 while (params){
245     int_chk(y = eval(car(params, lno), env));
246     if (x->u.ival != y->u.ival){ return NULL; }
247     params = cdr(params, lno);
248 }
249
250 return create_obj_int(1);
251 }
252
253 static OBJ*
254 gt(const OBJ* params, void* env, long lno)
255 {
256     OBJ *x, *y;
257
258     int_chk(x = eval(car(params, lno), env));
259     int_chk(y = eval(car(cdr(params, lno), 0), env));
260
261     if (x->u.ival > y->u.ival){ return create_obj_int(1); }
262
263     return NULL;
264 }
265
266 static OBJ*
267 cond(OBJ* params, void* env, long lno)
268 {
269     OBJ *ret = NULL;
270     OBJ *clause, *tst, *exps, *exp;
271
272     while(params){
273         clause = car(params, lno);
274         params = cdr(params, lno);
275
276         tst = car(clause, lno);
277         exps = cdr(clause, lno);
278
279         if((tst->ty == ID_OBJ && strcmp(tst->u.str, "else") == 0) || eval(tst, env)){
280             while(exps && (exp = car(exps, lno))){
281                 ret = eval(exp, env);
282                 exps = cdr(exps, lno);
283             }
284             break;
285         }
286     }
287
288     return ret;
289 }
290

```

```

291  **** closure ****/
292  typedef struct {
293      SCM_ENV* env;
294      const OBJ* dummy_args;
295      OBJ* exp;
296  } CLOSURE;
297
298  static OBJ*
299  lambda(OBJ* params, void* parent_env, long lno)
300  {
301      OBJ* obj;
302      CLOSURE *closure;
303
304      obj = mycalloc(sizeof(OBJ));
305      closure = mycalloc(sizeof(CLOSURE));
306
307      closure->env = parent_env ? parent_env: &root_env;
308      closure->dummy_args = car(params, lno);
309      closure->exp = cdr(params, lno);
310
311      obj->ty = FUNC;
312      obj->u.clos = closure;
313
314      return obj;
315  }
316
317  **** hygienic macros ****/
318
319  typedef struct {
320      SCM_ENV* env;
321      const OBJ* literal_part;
322      const OBJ* rules;
323      const OBJ* org_exp;
324  } SYNTAX_RULES;
325
326  static OBJ*
327  compile_macro(OBJ* exp, void* env) /* これを「コンパイル」と呼ぶのは...苦笑() */
328  {
329      OBJ *id, *ret;
330      SYNTAX_RULES *srules;
331
332      id = car(exp, 0);
333      exp = cdr(exp, 0);
334
335      if (!id || id->ty != ID_OBJ || strcmp(id->u.str, "syntax-rules")){
336          fprintf(stderr, "Not a syntax-rules.\n");
337          exit(1);
338      }
339

```

```

340  ret = mycalloc(sizeof(OBJ));
341  ret->ty = MACRO;
342  ret->u.clos = srules = mycalloc(sizeof(SYNTAX_RULES));
343
344  srules->env = env;
345  srules->literal_part = car(exp, 0);
346  srules->rules = cdr(exp, 0);
347  srules->org_exp = exp;
348
349  return ret;
350 }
351
352 static OBJ*
353 defsyn(OBJ* params, void* env, long lno)
354 {
355     OBJ *id, *m;
356     id = car(params, lno);
357
358     m = compile_macro(car(cdr(params, lno), lno), env);
359
360     if (id && id->ty == ID_OBJ){
361         define_var(env, id->u.str, m);
362     } else {
363         fprintf(stderr, "line %ld: syntax error in a syntax definition.\n", lno);
364         exit(1);
365     }
366
367     return NULL; /* TODO: #<undef>? */
368 }
369
370 SCM_ENV*
371 pattern_match(const OBJ *const pattern, OBJ* exp, OBJ* prec, SCM_ENV* dict)
372 {
373     OBJ *p = NULL, *obj, *l1, *l2;
374     char *s;
375
376     dict = dict ? dict : mycalloc(sizeof(SCM_ENV*));
377
378     /*
379     if (pattern) {
380         printf("pattern: "); print_obj(pattern); puts("");
381     }
382     if (prec) {
383         printf("prec: "); print_obj(prec); puts("");
384     }
385     */
386
387     if (pattern && (p = car(pattern, 0))){
388         if (p->ty == ID_OBJ){

```

```

389 if (strcmp(p->u.str, "...") == 0){
390     if (prec){
391         if (prec->ty == ID_OBJ){
392             s = mycalloc(strlen(prec->u.str) + 5);
393             sprintf(s, "%s□...", prec->u.str);
394             define_var(dict, s, exp);
395             return dict;
396         } else {
397             l1 = mycalloc(sizeof(OBJ));
398             l2 = mycalloc(sizeof(OBJ));
399             l1->ty = PAIR_OBJ;
400             l2->ty = PAIR_OBJ;
401
402             while (exp){
403                 obj = car(exp, 0);
404                 exp = cdr(exp, 0);
405                 if (l1->u.car){
406                     append_to_list(l1, car(obj, 0));
407                 } else {
408                     l1->u.car = car(obj, 0);
409                 }
410                 if (l2->u.car){
411                     append_to_list(l2, car(cdr(obj, 0), 0));
412                 } else {
413                     l2->u.car = car(cdr(obj, 0), 0);
414                 }
415             }
416
417             if (car(prec, 0)->ty != ID_OBJ || car(cdr(prec, 0), 0)->ty != ID_OBJ){
418                 print_obj(prec); puts("");
419                 fprintf(stderr, "error!\n");
420             }
421
422             s = mycalloc(strlen(car(prec, 0)->u.str) + 5);
423             sprintf(s, "%s□...", car(prec, 0)->u.str);
424             define_var(dict, s, l1);
425
426             s = mycalloc(strlen(car(cdr(prec, 0), 0)->u.str) + 5);
427             sprintf(s, "%s□...", car(cdr(prec, 0), 0)->u.str);
428             define_var(dict, s, l2);
429
430             return dict;
431         }
432     } else {
433         fprintf(stderr, "No precedence?\n");
434         exit(1);
435     }
436 } else {
437     define_var(dict, p->u.str, exp ? car(exp, 0) : NULL);

```



```

438     return pattern_match(cdr(pattern, 0),
439                          exp ? cdr(exp, 0) : NULL,
440                          p, dict);
441 }
442 } else if (p->ty == PAIR_OBJ){
443     if (pattern_match(car(pattern, 0), car(exp, 0), NULL, dict)){
444         return pattern_match(cdr(pattern, 0), cdr(exp, 0), p, dict);
445     }
446 } else {
447     /* TODO: 定数パターン*/
448 }
449 } else if (!exp){
450     return dict;
451 }
452
453 return NULL;
454 }
455
456 OBJ*
457 exec_template(SCM_ENV* dict, const OBJ *const template, OBJ* prec, OBJ* obj)
458 {
459     char *s;
460     OBJ *p, *exp;
461     KEY_VALUE_PAIR *kvs;
462
463     if (!dict){
464         return NULL;
465     }
466
467     /*
468     if (!obj){
469         puts("dict:");
470         kvs = dict->kvs;
471         while (kvs){
472             printf("%s => ", kvs->key);
473             print_obj(kvs->val);
474             puts("");
475             kvs = kvs->next;
476         }
477     }
478     */
479
480     if (template && (p = car(template, 0))){
481         if (p->ty == ID_OBJ){
482             if (strcmp(p->u.str, "...") == 0){
483                 s = mycalloc(strlen(prec->u.str + 5));
484                 sprintf(s, "%s...", prec->u.str);
485                 if (!lookup(dict, s, &exp)){
486                     fprintf(stderr, "error!\n");

```

```

487     }
488     set_cdr(obj, exp);
489     return obj;
490 } else {
491     if (!lookup(dict, p->u.str, &exp)){
492         exp = p;
493     }
494 }
495 } else {
496     exp = exec_template(dict, p, NULL, NULL);
497 }
498
499 if (obj){
500     append_to_list(obj, exp);
501 } else {
502     obj = mycalloc(sizeof(OBJ));
503     obj->ty = PAIR_OBJ;
504     obj->u.car = exp;
505 }
506
507 obj = exec_template(dict, cdr(template, 0), p, obj);
508 }
509
510 return obj;
511 }
512
513 OBJ*
514 pattern_chk(const OBJ *const rule, OBJ* exp)
515 {
516     SCM_ENV *dict;
517     dict = pattern_match(car(rule, 0), exp, NULL, NULL);
518     return exec_template(dict, car(cdr(rule, 0), 0), NULL, NULL);
519 }
520
521 OBJ*
522 expand_macro(OBJ* key, OBJ* exp)
523 {
524     OBJ *obj = NULL;
525     const OBJ *rule, *rules;
526     SYNTAX_RULES *srules;
527
528     srules = key->u.clos;
529     rules = srules->rules;
530
531     while (rules){
532         rule = car(rules, 0);
533         rules = cdr(rules, 0);
534         if((obj = pattern_chk(rule, exp)))
535             break;

```

```

536 }
537
538 if (!obj){
539     fprintf(stderr, "line_%ld: pattern_match_failed.\n", exp->lineno);
540     exit(1);
541 }
542
543 printf("@:"); print_obj(obj); puts("");
544 return eval(obj, srules->env);
545 }
546
547 /**** eval ****/
548
549 static OBJ version_str_obj = {STR_OBJ, {ISCM_VERSION}, NULL};
550
551 OBJ*
552 eval(OBJ *const exp, void* env)
553 {
554     OBJ* ret = NULL;
555     OBJ *obj, *op;
556     long lno;
557
558     /* print_obj(exp); */
559
560     if (!exp || exp->ty == INT_OBJ || exp->ty == STR_OBJ ||
561         exp->ty == FUNC || exp->ty == BUILTIN_FUNC || exp->ty == MACRO){
562         return exp;
563     }
564
565     if (exp && exp->ty == ID_OBJ){
566         if (lookup(env, exp->u.str, &obj)){
567             return eval(obj, env);
568         } else if (check_builtin_op(exp) != not_found) {
569             ret = mycalloc(sizeof(OBJ));
570             ret->ty = BUILTIN_FUNC;
571             ret->u.ival = check_builtin_op(exp);
572             return ret;
573         } else {
574             fprintf(stderr, "Unbound_variable: %s\n", exp->u.str);
575             exit(1);
576         }
577     }
578
579     op = eval(car(exp, 0), env);
580
581     if (op && op->ty == BUILTIN_FUNC){
582         lno = exp->lineno;
583
584         switch (op->u.ival){

```

```

585     case CAR:   ret = car(eval(car(cdr(exp, 0), 0), env), lno); break;
586     case CDR:   ret = cdr(eval(car(cdr(exp, 0), 0), env), lno); break;
587     case DEFINE: ret = def(cdr(exp, 0), env, lno);           break;
588     case SET:   ret = set(cdr(exp, 0), env, lno);           break;
589     case COND:  ret = cond(cdr(exp, 0), env, lno);          break;
590     case LAMBDA: ret = lambda(cdr(exp, 0), env, lno);        break;
591     case QUOTE: ret = car(cdr(exp, 0), 0);                  break;
592     case ADD:   ret = four_arithmetic(ADD, cdr(exp, 0), env, lno); break;
593     case SUB:   ret = four_arithmetic(SUB, cdr(exp, 0), env, lno); break;
594     case MUL:   ret = four_arithmetic(MUL, cdr(exp, 0), env, lno); break;
595     case DIV:   ret = four_arithmetic(DIV, cdr(exp, 0), env, lno); break;
596     case EQ:    ret = eq(cdr(exp, 0), env, lno);           break;
597     case GT:    ret = gt(cdr(exp, 0), env, lno);           break;
598     case DISP:  ret = disp(cdr(exp, 0), env, lno);          break;
599     case DEFSYN: ret = defsyn(cdr(exp, 0), env, lno);       break;
600     case VER:   ret = &version_str_obj;                     break;
601     default:    fprintf(stderr, "Operator not found.\n");   break;
602   }
603   } else if (op && op->ty == FUNC){
604     ret = funcall(op, cdr(exp, 0), env, exp->lineno);
605   } else if (op && op->ty == MACRO){
606     ret = expand_macro(op, exp);
607   } else {
608     print_obj(exp); puts("");
609     print_obj(op); puts("");
610     fprintf(stderr, "Error: not an operator.\n");
611     exit(1);
612   }
613
614   return ret;
615 }
616
617 /**** function call ****/
618
619 static SCM_ENV*
620 setup_parameter_env(const OBJ *dummy_args, OBJ *params, void* env, long lno)
621 {
622   OBJ *k, *v, *obj, *x;
623   SCM_ENV *ret;
624   ret = mycalloc(sizeof(SCM_ENV));
625
626   if (dummy_args && dummy_args->ty == ID_OBJ){
627     obj = create_pair(create_obj_id("quote"));
628     define_var(ret, dummy_args->u.str, obj);
629
630     x = NULL;
631
632     while (params && params->ty == PAIR_OBJ){
633       v = eval(car(params, lno), env);

```

```

634
635     if (x){
636         x->cdr = create_pair(v);
637         x = x->cdr;
638     } else {
639         obj->cdr = create_pair(x = create_pair(v));
640     }
641
642     params = cdr(params, lno);
643 }
644 } else {
645     while (dummy_args && dummy_args->ty == PAIR_OBJ &&
646           params && params->ty == PAIR_OBJ){
647         k = car(dummy_args, 0);
648         v = eval(car(params, lno), env);
649         define_var(ret, k->u.str, v);
650         dummy_args = cdr(dummy_args, 0);
651         params = cdr(params, lno);
652     }
653 }
654
655 return ret;
656 }
657
658 static OBJ*
659 funcall(OBJ* op, OBJ* params, void* env, long lno)
660 {
661     OBJ *ret = NULL, *exp;
662     CLOSURE* closure;
663     SCM_ENV* local_env;
664     SCM_ENV* org_parent;
665
666     closure = op->u.clos;
667
668     local_env = setup_parameter_env(closure->dummy_args, params, env, lno);
669
670     org_parent = closure->env;
671     closure->env = local_env;
672     local_env->parent = org_parent;
673
674     exp = closure->exp;
675     while (exp){
676         ret = eval(car(exp, 0), closure->env);
677         exp = cdr(exp, 0);
678     }
679
680     return ret;
681 }

```