```
1   // This software is in the public domain.

2   #include <assert.h>
3   #include <ctype.h>
4   #include <inttypes.h>
5   #include <stdarg.h>
6   #include <stdbool.h>
7   #include <stddef.h>
8   #include <stdio.h>
9   #include <stdlib.h>
10  #include <string.h>

11  //======================================================================
12  // Lisp objects
13  //======================================================================

14  // The Lisp object type
15  enum {
16      // Regular objects visible from the user
17      TINT = 1,
18      TCELL,
19      TSYMBOL,
20      TPRIMITIVE,
21      TFUNCTION,
22      TMACRO,
23      TSPECIAL,
24      TENV,
25  };

26  // Subtypes for TSPECIAL
27  enum {
28      TNIL = 1,
29      TDOT,
30      TCPAREN,
31      TTRUE,
32  };

33  struct Obj;

34  // Typedef for the primitive function.
35  typedef struct Obj *Primitive(struct Obj *env, struct Obj *args);

36  // The object type
37  typedef struct Obj {
38      // The first word of the object represents the type of the
39      // object. Any code that handles object needs to check its type
40      // first, then access the following union members.
41      int type;

42      // Object values.
43      union {
44          // Int
45          int value;
46          // Cell
47          struct {
48              struct Obj *car;
49              struct Obj *cdr;
50          };
51          // Symbol
52          char name[1];
53          // Primitive
54          Primitive *fn;
55          // Function or Macro
56          struct {
57              struct Obj *params;
58              struct Obj *body;
59              struct Obj *env;
60          };
61          // Subtype for special type
62          int subtype;
63          // Environment frame. This is a linked list of association lists
64          // containing the mapping from symbols to their value.
65          struct {
66              struct Obj *vars;
67              struct Obj *up;
68          };
69          // Forwarding pointer
70          void *moved;
71      };
```

```
72  } Obj;

73  // Constants
74  static Obj *Nil;
75  static Obj *Dot;
76  static Obj *Cparen;
77  static Obj *True;

78  // The list containing all symbols. Such data structure is traditionally
79  // called the "obarray", but I avoid using it as a variable name as this
80  // is not an array but a list.
81  static Obj *Symbols;

82  static void error(char *fmt, ...) __attribute((noreturn));

83  //======================================================================
84  // Constructors
85  //======================================================================

86  static Obj *alloc(int type, size_t size) {
87      // Add the size of the type tag.
88      size += offsetof(Obj, value);

89      // Allocate the object.
90      Obj *obj = malloc(size);
91      obj->type = type;
92      return obj;
93  }

94  static Obj *make_int(int value) {
95      Obj *r = alloc(TINT, sizeof(int));
96      r->value = value;
97      return r;
98  }

99  static Obj *make_symbol(char *name) {
100     Obj *sym = alloc(TSYMBOL, strlen(name) + 1);
101     strcpy(sym->name, name);
102     return sym;
103 }

104 static Obj *make_primitive(Primitive *fn) {
105     Obj *r = alloc(TPRIMITIVE, sizeof(Primitive *));
106     r->fn = fn;
107     return r;
108 }

109 static Obj *make_function(int type, Obj *params, Obj *body, Obj *env) {
110     assert(type == TFUNCTION || type == TMACRO);
111     Obj *r = alloc(type, sizeof(Obj *) * 3);
112     r->params = params;
113     r->body = body;
114     r->env = env;
115     return r;
116 }

117 static Obj *make_special(int subtype) {
118     Obj *r = malloc(sizeof(void *) * 2);
119     r->type = TSPECIAL;
120     r->subtype = subtype;
121     return r;
122 }

123 struct Obj *make_env(Obj *vars, Obj *up) {
124     Obj *r = alloc(TENV, sizeof(Obj *) * 2);
125     r->vars = vars;
126     r->up = up;
127     return r;
128 }

129 static Obj *cons(Obj *car, Obj *cdr) {
130     Obj *cell = alloc(TCELL, sizeof(Obj *) * 2);
131     cell->car = car;
132     cell->cdr = cdr;
133     return cell;
134 }

135 // Returns ((x . y) . a)
136 static Obj *acons(Obj *x, Obj *y, Obj *a) {
137     return cons(cons(x, y), a);
```

```
}

//======================================================================
// Parser
//
// This is a hand-written recursive-descendent parser.
//======================================================================

static Obj *read(void);

static void error(char *fmt, ...) {
    va_list ap;
    va_start(ap, fmt);
    vfprintf(stderr, fmt, ap);
    fprintf(stderr, "\n");
    va_end(ap);
    exit(1);
}

static int peek(void) {
    int c = getchar();
    ungetc(c, stdin);
    return c;
}

// Skips the input until newline is found. Newline is one of \r, \r\n,
// or \n.
static void skip_line(void) {
    for (;;) {
        int c = getchar();
        if (c == EOF || c == '\n')
            return;
        if (c == '\r') {
            if (peek() == '\n')
                getchar();
            return;
        }
    }
}

// Reads a list. Note that '(' has already been read.
static Obj *read_list(void) {
    Obj *obj = read();
    if (!obj)
        error("Unclosed parenthesis");
    if (obj == Dot)
        error("Stray dot");
    if (obj == Cparen)
        return Nil;
    Obj *head, *tail;
    head = tail = cons(obj, Nil);

    for (;;) {
        Obj *obj = read();
        if (!obj)
            error("Unclosed parenthesis");
        if (obj == Cparen)
            return head;
        if (obj == Dot) {
            tail->cdr = read();
            if (read() != Cparen)
                error("Closed parenthesis expected after dot");
            return head;
        }
        tail->cdr = cons(obj, Nil);
        tail = tail->cdr;
    }
}

// May create a new symbol. If there's a symbol with the same name, it
// will not create a new symbol but return the existing one.
static Obj *intern(char *name) {
    for (Obj *p = Symbols; p != Nil; p = p->cdr)
        if (strcmp(name, p->car->name) == 0)
            return p->car;
    Obj *sym = make_symbol(name);
    Symbols = cons(sym, Symbols);
    return sym;
}
```

```
209  // Reader macro ' (single quote). It reads an expression and returns
210  // (quote <expr>).
211  static Obj *read_quote(void) {
212      Obj *sym = intern("quote");
213      return cons(sym, cons(read(), Nil));
214  }

215  static int read_number(int val) {
216      while (isdigit(peek()))
217          val = val * 10 + (getchar() - '0');
218      return val;
219  }

220  #define SYMBOL_MAX_LEN 200

221  static Obj *read_symbol(char c) {
222      char buf[SYMBOL_MAX_LEN + 1];
223      int len = 1;
224      buf[0] = c;
225      while (isalnum(peek()) || peek() == '-') {
226          if (SYMBOL_MAX_LEN <= len)
227              error("Symbol name too long");
228          buf[len++] = getchar();
229      }
230      buf[len] = '\0';
231      return intern(buf);
232  }

233  static Obj *read(void) {
234      for (;;) {
235          int c = getchar();
236          if (c == ' ' || c == '\n' || c == '\r' || c == '\t')
237              continue;
238          if (c == EOF)
239              return NULL;
240          if (c == ';') {
241              skip_line();
242              continue;
243          }
244          if (c == '(')
245              return read_list();
246          if (c == ')')
247              return Cparen;
248          if (c == '.')
249              return Dot;
250          if (c == '\'')
251              return read_quote();
252          if (isdigit(c))
253              return make_int(read_number(c - '0'));
254          if (c == '-')
255              return make_int(-read_number(0));
256          if (isalpha(c) || strchr("+=!@#$%^&*", c))
257              return read_symbol(c);
258          error("Don't know how to handle %c", c);
259      }
260  }

261  // Prints the given object.
262  static void print(Obj *obj) {
263      switch (obj->type) {
264      case TINT:
265          printf("%d", obj->value);
266          return;
267      case TCELL:
268          printf("(");
269          for (;;) {
270              print(obj->car);
271              if (obj->cdr == Nil)
272                  break;
273              if (obj->cdr->type != TCELL) {
274                  printf(" . ");
275                  print(obj->cdr);
276                  break;
277              }
278              printf(" ");
279              obj = obj->cdr;
280          }
281          printf(")");
282          return;
283      case TSYMBOL:
```

```
284             printf("%s", obj->name);
285             return;
286         case TPRIMITIVE:
287             printf("<primitive>");
288             return;
289         case TFUNCTION:
290             printf("<function>");
291             return;
292         case TMACRO:
293             printf("<macro>");
294             return;
295         case TSPECIAL:
296             if (obj == Nil)
297                 printf("()");
298             else if (obj == True)
299                 printf("t");
300             else
301                 error("Bug: print: Unknown subtype: %d", obj->subtype);
302             return;
303         default:
304             error("Bug: print: Unknown tag type: %d", obj->type);
305         }
306     }

307     static int list_length(Obj *list) {
308         int len = 0;
309         for (;;) {
310             if (list == Nil)
311                 return len;
312             if (list->type != TCELL)
313                 error("length: cannot handle dotted list");
314             list = list->cdr;
315             len++;
316         }
317     }

318     //======================================================================
319     // Evaluator
320     //======================================================================

321     static Obj *eval(Obj *env, Obj *obj);

322     static void add_variable(Obj *env, Obj *sym, Obj *val) {
323         env->vars = acons(sym, val, env->vars);
324     }

325     // Returns a newly created environment frame.
326     static Obj *push_env(Obj *env, Obj *vars, Obj *values) {
327         if (list_length(vars) != list_length(values))
328             error("Cannot apply function: number of argument does not match");
329         Obj *map = Nil;
330         for (Obj *p = vars, *q = values; p != Nil; p = p->cdr, q = q->cdr) {
331             Obj *sym = p->car;
332             Obj *val = q->car;
333             map = acons(sym, val, map);
334         }
335         return make_env(map, env);
336     }

337     // Evaluates the list elements from head and returns the last return
338     // value.
339     static Obj *progn(Obj *env, Obj *list) {
340         Obj *r = NULL;
341         for (Obj *lp = list; lp != Nil; lp = lp->cdr)
342             r = eval(env, lp->car);
343         return r;
344     }

345     // Evaluates all the list elements and returns their return values as a
346     // new list.
347     static Obj *eval_list(Obj *env, Obj *list) {
348         Obj *head = NULL;
349         Obj *tail = NULL;
350         for (Obj *lp = list; lp != Nil; lp = lp->cdr) {
351             Obj *tmp = eval(env, lp->car);
352             if (head == NULL) {
353                 head = tail = cons(tmp, Nil);
354             } else {
355                 tail->cdr = cons(tmp, Nil);
356                 tail = tail->cdr;
```

```
357             }
358         }
359         if (head == NULL)
360             return Nil;
361         return head;
362     }

363     static bool is_list(Obj *obj) {
364       return obj == Nil || obj->type == TCELL;
365     }

366     // Apply fn with args.
367     static Obj *apply(Obj *env, Obj *fn, Obj *args) {
368         if (!is_list(args))
369             error("argument must be a list");
370         if (fn->type == TPRIMITIVE)
371             return fn->fn(env, args);
372         if (fn->type == TFUNCTION) {
373             Obj *body = fn->body;
374             Obj *params = fn->params;
375             Obj *eargs = eval_list(env, args);
376             Obj *newenv = push_env(fn->env, params, eargs);
377             return progn(newenv, body);
378         }
379         error("not supported");
380     }

381     // Searches for a variable by symbol. Returns null if not found.
382     static Obj *find(Obj *env, Obj *sym) {
383         for (Obj *p = env; p; p = p->up) {
384             for (Obj *cell = p->vars; cell != Nil; cell = cell->cdr) {
385                 Obj *bind = cell->car;
386                 if (sym == bind->car)
387                     return bind;
388             }
389         }
390         return NULL;
391     }

392     // Expands the given macro application form.
393     static Obj *macroexpand(Obj *env, Obj *obj) {
394         if (obj->type != TCELL || obj->car->type != TSYMBOL)
395             return obj;
396         Obj *bind = find(env, obj->car);
397         if (!bind || bind->cdr->type != TMACRO)
398             return obj;
399         Obj *args = obj->cdr;
400         Obj *body = bind->cdr->body;
401         Obj *params = bind->cdr->params;
402         Obj *newenv = push_env(env, params, args);
403         return progn(newenv, body);
404     }

405     // Evaluates the S expression.
406     static Obj *eval(Obj *env, Obj *obj) {
407         switch (obj->type) {
408         case TINT:
409         case TPRIMITIVE:
410         case TFUNCTION:
411         case TSPECIAL:
412             // Self-evaluating objects
413             return obj;
414         case TSYMBOL: {
415             // Variable
416             Obj *bind = find(env, obj);
417             if (!bind)
418                 error("Undefined symbol: %s", obj->name);
419             return bind->cdr;
420         }
421         case TCELL: {
422             // Function application form
423             Obj *expanded = macroexpand(env, obj);
424             if (expanded != obj)
425                 return eval(env, expanded);
426             Obj *fn = eval(env, obj->car);
427             Obj *args = obj->cdr;
428             if (fn->type != TPRIMITIVE && fn->type != TFUNCTION)
429                 error("The head of a list must be a function");
430             return apply(env, fn, args);
431         }
```

```
432         default:
433             error("Bug: eval: Unknown tag type: %d", obj->type);
434         }
435     }
436 
437     //======================================================================
438     // Functions and special forms
439     //======================================================================
440 
441     // 'expr
442     static Obj *prim_quote(Obj *env, Obj *list) {
443         if (list_length(list) != 1)
444             error("Malformed quote");
445         return list->car;
446     }
447 
448     // (list expr ...)
449     static Obj *prim_list(Obj *env, Obj *list) {
450         return eval_list(env, list);
451     }
452 
453     // (setq <symbol> expr)
454     static Obj *prim_setq(Obj *env, Obj *list) {
455         if (list_length(list) != 2 || list->car->type != TSYMBOL)
456             error("Malformed setq");
457         Obj *bind = find(env, list->car);
458         if (!bind)
459             error("Unbound variable %s", list->car->name);
460         Obj *value = eval(env, list->cdr->car);
461         bind->cdr = value;
462         return value;
463     }
464 
465     // (+ <integer> ...)
466     static Obj *prim_plus(Obj *env, Obj *list) {
467         int sum = 0;
468         for (Obj *args = eval_list(env, list); args != Nil; args = args->cdr) {
469             if (args->car->type != TINT)
470                 error("+ takes only numbers");
471             sum += args->car->value;
472         }
473         return make_int(sum);
474     }
475 
476     static Obj *handle_function(Obj *env, Obj *list, int type) {
477         if (list->type != TCELL || !is_list(list->car) || list->cdr->type != TCELL)
478             error("Malformed lambda");
479         for (Obj *p = list->car; p != Nil; p = p->cdr) {
480             if (p->car->type != TSYMBOL)
481                 error("Parameter must be a symbol");
482             if (!is_list(p->cdr))
483                 error("Parameter list is not a flat list");
484         }
485         Obj *car = list->car;
486         Obj *cdr = list->cdr;
487         return make_function(type, car, cdr, env);
488     }
489 
490     // (lambda (<symbol> ...) expr ...)
491     static Obj *prim_lambda(Obj *env, Obj *list) {
492         return handle_function(env, list, TFUNCTION);
493     }
494 
495     static Obj *handle_defun(Obj *env, Obj *list, int type) {
496         if (list->car->type != TSYMBOL || list->cdr->type != TCELL)
497             error("Malformed defun");
498         Obj *sym = list->car;
499         Obj *rest = list->cdr;
500         Obj *fn = handle_function(env, rest, type);
501         add_variable(env, sym, fn);
502         return fn;
503     }
504 
505     // (defun <symbol> (<symbol> ...) expr ...)
506     static Obj *prim_defun(Obj *env, Obj *list) {
507         return handle_defun(env, list, TFUNCTION);
508     }
509 
510     // (define <symbol> expr)
511     static Obj *prim_define(Obj *env, Obj *list) {
```

```
502     if (list_length(list) != 2 || list->car->type != TSYMBOL)
503         error("Malformed define");
504     Obj *sym = list->car;
505     Obj *value = eval(env, list->cdr->car);
506     add_variable(env, sym, value);
507     return value;
508 }

509 // (defmacro <symbol> (<symbol> ...) expr ...)
510 static Obj *prim_defmacro(Obj *env, Obj *list) {
511     return handle_defun(env, list, TMACRO);
512 }

513 // (macroexpand expr)
514 static Obj *prim_macroexpand(Obj *env, Obj *list) {
515     if (list_length(list) != 1)
516         error("Malformed macroexpand");
517     Obj *body = list->car;
518     return macroexpand(env, body);
519 }

520 // (println expr)
521 static Obj *prim_println(Obj *env, Obj *list) {
522     print(eval(env, list->car));
523     printf("\n");
524     return Nil;
525 }

526 // (if expr expr expr ...)
527 static Obj *prim_if(Obj *env, Obj *list) {
528     if (list_length(list) < 2)
529         error("Malformed if");
530     Obj *cond = eval(env, list->car);
531     if (cond != Nil) {
532         Obj *then = list->cdr->car;
533         return eval(env, then);
534     }
535     Obj *els = list->cdr->cdr;
536     return els == Nil ? Nil : progn(env, els);
537 }

538 // (= <integer> <integer>)
539 static Obj *prim_num_eq(Obj *env, Obj *list) {
540     if (list_length(list) != 2)
541         error("Malformed =");
542     Obj *values = eval_list(env, list);
543     Obj *x = values->car;
544     Obj *y = values->cdr->car;
545     if (x->type != TINT || y->type != TINT)
546         error("= only takes numbers");
547     return x->value == y->value ? True : Nil;
548 }

549 // (exit)
550 static Obj *prim_exit(Obj *env, Obj *list) {
551     exit(0);
552 }

553 static void add_primitive(Obj *env, char *name, Primitive *fn) {
554     Obj *sym = intern(name);
555     Obj *prim = make_primitive(fn);
556     add_variable(env, sym, prim);
557 }

558 static void define_constants(Obj *env) {
559     Obj *sym = intern("t");
560     add_variable(env, sym, True);
561 }

562 static void define_primitives(Obj *env) {
563     add_primitive(env, "quote", prim_quote);
564     add_primitive(env, "list", prim_list);
565     add_primitive(env, "setq", prim_setq);
566     add_primitive(env, "+", prim_plus);
567     add_primitive(env, "define", prim_define);
568     add_primitive(env, "defun", prim_defun);
569     add_primitive(env, "defmacro", prim_defmacro);
570     add_primitive(env, "macroexpand", prim_macroexpand);
571     add_primitive(env, "lambda", prim_lambda);
572     add_primitive(env, "if", prim_if);
```

```
573         add_primitive(env, "=", prim_num_eq);
574         add_primitive(env, "println", prim_println);
575         add_primitive(env, "exit", prim_exit);
576     }

577     //======================================================================
578     // Entry point
579     //======================================================================

580     int main(int argc, char **argv) {
581         // Constants and primitives
582         Nil = make_special(TNIL);
583         Dot = make_special(TDOT);
584         Cparen = make_special(TCPAREN);
585         True = make_special(TTRUE);
586         Symbols = Nil;

587         Obj *env = make_env(Nil, NULL);

588         define_constants(env);
589         define_primitives(env);

590         // The main loop
591         for (;;) {
592             Obj *expr = read();
593             if (!expr)
594                 return 0;
595             if (expr == Cparen)
596                 error("Stray close parenthesis");
597             if (expr == Dot)
598                 error("Stray dot");
599             print(eval(env, expr));
600             printf("\n");
601         }
602     }
```