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

#include <assert.h>
#include <ctype.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>

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

//======================================================================
// Lisp objects
//======================================================================

// The Lisp object type
enum {
    // Regular objects visible from the user
    TINT = 1,
    TCELL,
    TSYMBOL,
    TPRIMITIVE,
    TFUNCTION,
    TMACRO,
    TENV,
    // The marker that indicates the object has been moved to other location by GC. The new location
    // can be found at the forwarding pointer. Only the functions to do garbage collection set and
    // handle the object of this type. Other functions will never see the object of this type.
    TMOVED,
    // Const objects. They are statically allocated and will never be managed by GC.
    TTRUE,
    TNIL,
    TDOT,
    TCPAREN,
};

// Typedef for the primitive function
struct Obj;
typedef struct Obj *Primitive(void *root, struct Obj **env, struct Obj **args);

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

    // The total size of the object, including "type" field, this field, the contents, and the
    // padding at the end of the object.
    int size;

    // Object values.
    union {
        // Int
        int value;
        // Cell
        struct {
            struct Obj *car;
            struct Obj *cdr;
        };
        // Symbol
        char name[1];
        // Primitive
        Primitive *fn;
        // Function or Macro
        struct {
            struct Obj *params;
            struct Obj *body;
            struct Obj *env;
```

```
72            };
73            // Environment frame. This is a linked list of association lists
74            // containing the mapping from symbols to their value.
75            struct {
76                struct Obj *vars;
77                struct Obj *up;
78            };
79            // Forwarding pointer
80            void *moved;
81        };
82    } Obj;

83    // Constants
84    static Obj *True = &(Obj){ TTRUE };
85    static Obj *Nil = &(Obj){ TNIL };
86    static Obj *Dot = &(Obj){ TDOT };
87    static Obj *Cparen = &(Obj){ TCPAREN };

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

91    //======================================================================
92    // Memory management
93    //======================================================================

94    // The size of the heap in byte
95    #define MEMORY_SIZE 65536

96    // The pointer pointing to the beginning of the current heap
97    static void *memory;

98    // The pointer pointing to the beginning of the old heap
99    static void *from_space;

100   // The number of bytes allocated from the heap
101   static size_t mem_nused = 0;

102   // Flags to debug GC
103   static bool gc_running = false;
104   static bool debug_gc = false;
105   static bool always_gc = false;

106   static void gc(void *root);

107   // Currently we are using Cheney's copying GC algorithm, with which the available memory is split
108   // into two halves and all objects are moved from one half to another every time GC is invoked. That
109   // means the address of the object keeps changing. If you take the address of an object and keep it
110   // in a C variable, dereferencing it could cause SEGV because the address becomes invalid after GC
111   // runs.
112   //
113   // In order to deal with that, all access from C to Lisp objects will go through two levels of
114   // pointer dereferences. The C local variable is pointing to a pointer on the C stack, and the
115   // pointer is pointing to the Lisp object. GC is aware of the pointers in the stack and updates
116   // their contents with the objects' new addresses when GC happens.
117   //
118   // The following is a macro to reserve the area in the C stack for the pointers. The contents of
119   // this area are considered to be GC root.
120   //
121   // Be careful not to bypass the two levels of pointer indirections. If you create a direct pointer
122   // to an object, it'll cause a subtle bug. Such code would work in most cases but fails with SEGV if
123   // GC happens during the execution of the code. Any code that allocates memory may invoke GC.

124   #define ROOT_END ((void *)-1)

125   #define ADD_ROOT(size)                      \
126       void *root_ADD_ROOT_[size + 2];         \
127       root_ADD_ROOT_[0] = root;               \
128       for (int i = 1; i <= size; i++)         \
129           root_ADD_ROOT_[i] = NULL;           \
130       root_ADD_ROOT_[size + 1] = ROOT_END;    \
131       root = root_ADD_ROOT_

132   #define DEFINE1(var1)                       \
133       ADD_ROOT(1);                            \
134       Obj **var1 = (Obj **)(root_ADD_ROOT_ + 1)

135   #define DEFINE2(var1, var2)                 \
136       ADD_ROOT(2);                            \
137       Obj **var1 = (Obj **)(root_ADD_ROOT_ + 1);  \
```

```
138     Obj **var2 = (Obj **)(root_ADD_ROOT_ + 2)
139 
140 #define DEFINE3(var1, var2, var3)           \
141     ADD_ROOT(3);                            \
142     Obj **var1 = (Obj **)(root_ADD_ROOT_ + 1);  \
143     Obj **var2 = (Obj **)(root_ADD_ROOT_ + 2);  \
144     Obj **var3 = (Obj **)(root_ADD_ROOT_ + 3)
145 
146 #define DEFINE4(var1, var2, var3, var4)     \
147     ADD_ROOT(4);                            \
148     Obj **var1 = (Obj **)(root_ADD_ROOT_ + 1);  \
149     Obj **var2 = (Obj **)(root_ADD_ROOT_ + 2);  \
150     Obj **var3 = (Obj **)(root_ADD_ROOT_ + 3);  \
151     Obj **var4 = (Obj **)(root_ADD_ROOT_ + 4)
152 
153 // Round up the given value to a multiple of size. Size must be a power of 2. It adds size - 1
154 // first, then zero-ing the least significant bits to make the result a multiple of size. I know
155 // these bit operations may look a little bit tricky, but it's efficient and thus frequently used.
156 static inline size_t roundup(size_t var, size_t size) {
157     return (var + size - 1) & ~(size - 1);
158 }
159 
160 // Allocates memory block. This may start GC if we don't have enough memory.
161 static Obj *alloc(void *root, int type, size_t size) {
162     // The object must be large enough to contain a pointer for the forwarding pointer. Make it
163     // larger if it's smaller than that.
164     size = roundup(size, sizeof(void *));
165 
166     // Add the size of the type tag and size fields.
167     size += offsetof(Obj, value);
168 
169     // Round up the object size to the nearest alignment boundary, so that the next object will be
170     // allocated at the proper alignment boundary. Currently we align the object at the same
171     // boundary as the pointer.
172     size = roundup(size, sizeof(void *));
173 
174     // If the debug flag is on, allocate a new memory space to force all the existing objects to
175     // move to new addresses, to invalidate the old addresses. By doing this the GC behavior becomes
176     // more predictable and repeatable. If there's a memory bug that the C variable has a direct
177     // reference to a Lisp object, the pointer will become invalid by this GC call. Dereferencing
178     // that will immediately cause SEGV.
179     if (always_gc && !gc_running)
180         gc(root);
181 
182     // Otherwise, run GC only when the available memory is not large enough.
183     if (!always_gc && MEMORY_SIZE < mem_nused + size)
184         gc(root);
185 
186     // Terminate the program if we couldn't satisfy the memory request. This can happen if the
187     // requested size was too large or the from-space was filled with too many live objects.
188     if (MEMORY_SIZE < mem_nused + size)
189         error("Memory exhausted");
190 
191     // Allocate the object.
192     Obj *obj = memory + mem_nused;
193     obj->type = type;
194     obj->size = size;
195     mem_nused += size;
196     return obj;
197 }
198 
199 //======================================================================
200 // Garbage collector
201 //======================================================================
202 
203 // Cheney's algorithm uses two pointers to keep track of GC status. At first both pointers point to
204 // the beginning of the to-space. As GC progresses, they are moved towards the end of the
205 // to-space. The objects before "scan1" are the objects that are fully copied. The objects between
206 // "scan1" and "scan2" have already been copied, but may contain pointers to the from-space. "scan2"
207 // points to the beginning of the free space.
208 static Obj *scan1;
209 static Obj *scan2;
210 
211 // Moves one object from the from-space to the to-space. Returns the object's new address. If the
212 // object has already been moved, does nothing but just returns the new address.
213 static inline Obj *forward(Obj *obj) {
214     // If the object's address is not in the from-space, the object is not managed by GC nor it
215     // has already been moved to the to-space.
216     ptrdiff_t offset = (uint8_t *)obj - (uint8_t *)from_space;
217     if (offset < 0 || MEMORY_SIZE <= offset)
```

```
205         return obj;

206     // The pointer is pointing to the from-space, but the object there was a tombstone. Follow the
207     // forwarding pointer to find the new location of the object.
208     if (obj->type == TMOVED)
209         return obj->moved;

210     // Otherwise, the object has not been moved yet. Move it.
211     Obj *newloc = scan2;
212     memcpy(newloc, obj, obj->size);
213     scan2 = (Obj *)((uint8_t *)scan2 + obj->size);

214     // Put a tombstone at the location where the object used to occupy, so that the following call
215     // of forward() can find the object's new location.
216     obj->type = TMOVED;
217     obj->moved = newloc;
218     return newloc;
219 }

220 static void *alloc_semispace() {
221     return mmap(NULL, MEMORY_SIZE, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANON, -1, 0);
222 }

223 // Copies the root objects.
224 static void forward_root_objects(void *root) {
225     Symbols = forward(Symbols);
226     for (void **frame = root; frame; frame = *(void ***)frame)
227         for (int i = 1; frame[i] != ROOT_END; i++)
228             if (frame[i])
229                 frame[i] = forward(frame[i]);
230 }

231 // Implements Cheney's copying garbage collection algorithm.
232 // http://en.wikipedia.org/wiki/Cheney%27s_algorithm
233 static void gc(void *root) {
234     assert(!gc_running);
235     gc_running = true;

236     // Allocate a new semi-space.
237     from_space = memory;
238     memory = alloc_semispace();

239     // Initialize the two pointers for GC. Initially they point to the beginning of the to-space.
240     scan1 = scan2 = memory;

241     // Copy the GC root objects first. This moves the pointer scan2.
242     forward_root_objects(root);

243     // Copy the objects referenced by the GC root objects located between scan1 and scan2. Once it's
244     // finished, all live objects (i.e. objects reachable from the root) will have been copied to
245     // the to-space.
246     while (scan1 < scan2) {
247         switch (scan1->type) {
248         case TINT:
249         case TSYMBOL:
250         case TPRIMITIVE:
251             // Any of the above types does not contain a pointer to a GC-managed object.
252             break;
253         case TCELL:
254             scan1->car = forward(scan1->car);
255             scan1->cdr = forward(scan1->cdr);
256             break;
257         case TFUNCTION:
258         case TMACRO:
259             scan1->params = forward(scan1->params);
260             scan1->body = forward(scan1->body);
261             scan1->env = forward(scan1->env);
262             break;
263         case TENV:
264             scan1->vars = forward(scan1->vars);
265             scan1->up = forward(scan1->up);
266             break;
267         default:
268             error("Bug: copy: unknown type %d", scan1->type);
269         }
270         scan1 = (Obj *)((uint8_t *)scan1 + scan1->size);
271     }

272     // Finish up GC.
273     munmap(from_space, MEMORY_SIZE);
```

```
274         size_t old_nused = mem_nused;
275         mem_nused = (size_t)((uint8_t *)scan1 - (uint8_t *)memory);
276         if (debug_gc)
277             fprintf(stderr, "GC: %zu bytes out of %zu bytes copied.\n", mem_nused, old_nused);
278         gc_running = false;
279     }

280     //======================================================================
281     // Constructors
282     //======================================================================

283     static Obj *make_int(void *root, int value) {
284         Obj *r = alloc(root, TINT, sizeof(int));
285         r->value = value;
286         return r;
287     }

288     static Obj *cons(void *root, Obj **car, Obj **cdr) {
289         Obj *cell = alloc(root, TCELL, sizeof(Obj *) * 2);
290         cell->car = *car;
291         cell->cdr = *cdr;
292         return cell;
293     }

294     static Obj *make_symbol(void *root, char *name) {
295         Obj *sym = alloc(root, TSYMBOL, strlen(name) + 1);
296         strcpy(sym->name, name);
297         return sym;
298     }

299     static Obj *make_primitive(void *root, Primitive *fn) {
300         Obj *r = alloc(root, TPRIMITIVE, sizeof(Primitive *));
301         r->fn = fn;
302         return r;
303     }

304     static Obj *make_function(void *root, Obj **env, int type, Obj **params, Obj **body) {
305         assert(type == TFUNCTION || type == TMACRO);
306         Obj *r = alloc(root, type, sizeof(Obj *) * 3);
307         r->params = *params;
308         r->body = *body;
309         r->env = *env;
310         return r;
311     }

312     struct Obj *make_env(void *root, Obj **vars, Obj **up) {
313         Obj *r = alloc(root, TENV, sizeof(Obj *) * 2);
314         r->vars = *vars;
315         r->up = *up;
316         return r;
317     }

318     // Returns ((x . y) . a)
319     static Obj *acons(void *root, Obj **x, Obj **y, Obj **a) {
320         DEFINE1(cell);
321         *cell = cons(root, x, y);
322         return cons(root, cell, a);
323     }

324     //======================================================================
325     // Parser
326     //
327     // This is a hand-written recursive-descendent parser.
328     //======================================================================

329     #define SYMBOL_MAX_LEN 200
330     const char symbol_chars[] = "~!@#$%^&*-_=+:/?<>";

331     static Obj *read_expr(void *root);

332     static int peek(void) {
333         int c = getchar();
334         ungetc(c, stdin);
335         return c;
336     }

337     // Destructively reverses the given list.
338     static Obj *reverse(Obj *p) {
339         Obj *ret = Nil;
340         while (p != Nil) {
```

```
341         Obj *head = p;
342         p = p->cdr;
343         head->cdr = ret;
344         ret = head;
345       }
346       return ret;
347     }

348     // Skips the input until newline is found. Newline is one of \r, \r\n or \n.
349     static void skip_line(void) {
350       for (;;) {
351         int c = getchar();
352         if (c == EOF || c == '\n')
353           return;
354         if (c == '\r') {
355           if (peek() == '\n')
356             getchar();
357           return;
358         }
359       }
360     }

361     // Reads a list. Note that '(' has already been read.
362     static Obj *read_list(void *root) {
363       DEFINE3(obj, head, last);
364       *head = Nil;
365       for (;;) {
366         *obj = read_expr(root);
367         if (!*obj)
368           error("Unclosed parenthesis");
369         if (*obj == Cparen)
370           return reverse(*head);
371         if (*obj == Dot) {
372           *last = read_expr(root);
373           if (read_expr(root) != Cparen)
374             error("Closed parenthesis expected after dot");
375           Obj *ret = reverse(*head);
376           (*head)->cdr = *last;
377           return ret;
378         }
379         *head = cons(root, obj, head);
380       }
381     }

382     // May create a new symbol. If there's a symbol with the same name, it will not create a new symbol
383     // but return the existing one.
384     static Obj *intern(void *root, char *name) {
385       for (Obj *p = Symbols; p != Nil; p = p->cdr)
386         if (strcmp(name, p->car->name) == 0)
387           return p->car;
388       DEFINE1(sym);
389       *sym = make_symbol(root, name);
390       Symbols = cons(root, sym, &Symbols);
391       return *sym;
392     }

393     // Reader marcro ' (single quote). It reads an expression and returns (quote <expr>).
394     static Obj *read_quote(void *root) {
395       DEFINE2(sym, tmp);
396       *sym = intern(root, "quote");
397       *tmp = read_expr(root);
398       *tmp = cons(root, tmp, &Nil);
399       *tmp = cons(root, sym, tmp);
400       return *tmp;
401     }

402     static int read_number(int val) {
403       while (isdigit(peek()))
404         val = val * 10 + (getchar() - '0');
405       return val;
406     }

407     static Obj *read_symbol(void *root, char c) {
408       char buf[SYMBOL_MAX_LEN + 1];
409       buf[0] = c;
410       int len = 1;
411       while (isalnum(peek()) || strchr(symbol_chars, peek())) {
412         if (SYMBOL_MAX_LEN <= len)
413           error("Symbol name too long");
414         buf[len++] = getchar();
```

```
415         }
416         buf[len] = '\0';
417         return intern(root, buf);
418     }

419     static Obj *read_expr(void *root) {
420         for (;;) {
421             int c = getchar();
422             if (c == ' ' || c == '\n' || c == '\r' || c == '\t')
423                 continue;
424             if (c == EOF)
425                 return NULL;
426             if (c == ';') {
427                 skip_line();
428                 continue;
429             }
430             if (c == '(')
431                 return read_list(root);
432             if (c == ')')
433                 return Cparen;
434             if (c == '.')
435                 return Dot;
436             if (c == '\'')
437                 return read_quote(root);
438             if (isdigit(c))
439                 return make_int(root, read_number(c - '0'));
440             if (c == '-' && isdigit(peek()))
441                 return make_int(root, -read_number(0));
442             if (isalpha(c) || strchr(symbol_chars, c))
443                 return read_symbol(root, c);
444             error("Don't know how to handle %c", c);
445         }
446     }

447     // Prints the given object.
448     static void print(Obj *obj) {
449         switch (obj->type) {
450         case TCELL:
451             printf("(");
452             for (;;) {
453                 print(obj->car);
454                 if (obj->cdr == Nil)
455                     break;
456                 if (obj->cdr->type != TCELL) {
457                     printf(" . ");
458                     print(obj->cdr);
459                     break;
460                 }
461                 printf(" ");
462                 obj = obj->cdr;
463             }
464             printf(")");
465             return;

466     #define CASE(type, ...)                         \
467         case type:                                  \
468             printf(__VA_ARGS__);                    \
469             return
470         CASE(TINT, "%d", obj->value);
471         CASE(TSYMBOL, "%s", obj->name);
472         CASE(TPRIMITIVE, "<primitive>");
473         CASE(TFUNCTION, "<function>");
474         CASE(TMACRO, "<macro>");
475         CASE(TMOVED, "<moved>");
476         CASE(TTRUE, "t");
477         CASE(TNIL, "()");
478     #undef CASE
479         default:
480             error("Bug: print: Unknown tag type: %d", obj->type);
481         }
482     }

483     // Returns the length of the given list. -1 if it's not a proper list.
484     static int length(Obj *list) {
485         int len = 0;
486         for (; list->type == TCELL; list = list->cdr)
487             len++;
488         return list == Nil ? len : -1;
489     }
```

```
490 //======================================================================
491 // Evaluator
492 //======================================================================

493 static Obj *eval(void *root, Obj **env, Obj **obj);

494 static void add_variable(void *root, Obj **env, Obj **sym, Obj **val) {
495     DEFINE2(vars, tmp);
496     *vars = (*env)->vars;
497     *tmp = acons(root, sym, val, vars);
498     (*env)->vars = *tmp;
499 }

500 // Returns a newly created environment frame.
501 static Obj *push_env(void *root, Obj **env, Obj **vars, Obj **vals) {
502     DEFINE3(map, sym, val);
503     *map = Nil;
504     for (; (*vars)->type == TCELL; *vars = (*vars)->cdr, *vals = (*vals)->cdr) {
505         if ((*vals)->type != TCELL)
506             error("Cannot apply function: number of argument does not match");
507         *sym = (*vars)->car;
508         *val = (*vals)->car;
509         *map = acons(root, sym, val, map);
510     }
511     if (*vars != Nil)
512         *map = acons(root, vars, vals, map);
513     return make_env(root, map, env);
514 }

515 // Evaluates the list elements from head and returns the last return value.
516 static Obj *progn(void *root, Obj **env, Obj **list) {
517     DEFINE2(lp, r);
518     for (*lp = *list; *lp != Nil; *lp = (*lp)->cdr) {
519         *r = (*lp)->car;
520         *r = eval(root, env, r);
521     }
522     return *r;
523 }

524 // Evaluates all the list elements and returns their return values as a new list.
525 static Obj *eval_list(void *root, Obj **env, Obj **list) {
526     DEFINE4(head, lp, expr, result);
527     *head = Nil;
528     for (lp = list; *lp != Nil; *lp = (*lp)->cdr) {
529         *expr = (*lp)->car;
530         *result = eval(root, env, expr);
531         *head = cons(root, result, head);
532     }
533     return reverse(*head);
534 }

535 static bool is_list(Obj *obj) {
536     return obj == Nil || obj->type == TCELL;
537 }

538 static Obj *apply_func(void *root, Obj **env, Obj **fn, Obj **args) {
539     DEFINE3(params, newenv, body);
540     *params = (*fn)->params;
541     *newenv = (*fn)->env;
542     *newenv = push_env(root, newenv, params, args);
543     *body = (*fn)->body;
544     return progn(root, newenv, body);
545 }

546 // Apply fn with args.
547 static Obj *apply(void *root, Obj **env, Obj **fn, Obj **args) {
548     if (!is_list(*args))
549         error("argument must be a list");
550     if ((*fn)->type == TPRIMITIVE)
551         return (*fn)->fn(root, env, args);
552     if ((*fn)->type == TFUNCTION) {
553         DEFINE1(eargs);
554         *eargs = eval_list(root, env, args);
555         return apply_func(root, env, fn, eargs);
556     }
557     error("not supported");
558 }

559 // Searches for a variable by symbol. Returns null if not found.
560 static Obj *find(Obj **env, Obj *sym) {
```

```
561     for (Obj *p = *env; p != Nil; p = p->up) {
562         for (Obj *cell = p->vars; cell != Nil; cell = cell->cdr) {
563             Obj *bind = cell->car;
564             if (sym == bind->car)
565                 return bind;
566         }
567     }
568     return NULL;
569 }

570 // Expands the given macro application form.
571 static Obj *macroexpand(void *root, Obj **env, Obj **obj) {
572     if ((*obj)->type != TCELL || (*obj)->car->type != TSYMBOL)
573         return *obj;
574     DEFINE3(bind, macro, args);
575     *bind = find(env, (*obj)->car);
576     if (!*bind || (*bind)->cdr->type != TMACRO)
577         return *obj;
578     *macro = (*bind)->cdr;
579     *args = (*obj)->cdr;
580     return apply_func(root, env, macro, args);
581 }

582 // Evaluates the S expression.
583 static Obj *eval(void *root, Obj **env, Obj **obj) {
584     switch ((*obj)->type) {
585     case TINT:
586     case TPRIMITIVE:
587     case TFUNCTION:
588     case TTRUE:
589     case TNIL:
590         // Self-evaluating objects
591         return *obj;
592     case TSYMBOL: {
593         // Variable
594         Obj *bind = find(env, *obj);
595         if (!bind)
596             error("Undefined symbol: %s", (*obj)->name);
597         return bind->cdr;
598     }
599     case TCELL: {
600         // Function application form
601         DEFINE3(fn, expanded, args);
602         *expanded = macroexpand(root, env, obj);
603         if (*expanded != *obj)
604             return eval(root, env, expanded);
605         *fn = (*obj)->car;
606         *fn = eval(root, env, fn);
607         *args = (*obj)->cdr;
608         if ((*fn)->type != TPRIMITIVE && (*fn)->type != TFUNCTION)
609             error("The head of a list must be a function");
610         return apply(root, env, fn, args);
611     }
612     default:
613         error("Bug: eval: Unknown tag type: %d", (*obj)->type);
614     }
615 }

616 //======================================================================
617 // Primitive functions and special forms
618 //======================================================================

619 // 'expr
620 static Obj *prim_quote(void *root, Obj **env, Obj **list) {
621     if (length(*list) != 1)
622         error("Malformed quote");
623     return (*list)->car;
624 }

625 // (cons expr expr)
626 static Obj *prim_cons(void *root, Obj **env, Obj **list) {
627     if (length(*list) != 2)
628         error("Malformed cons");
629     Obj *cell = eval_list(root, env, list);
630     cell->cdr = cell->cdr->car;
631     return cell;
632 }

633 // (car <cell>)
634 static Obj *prim_car(void *root, Obj **env, Obj **list) {
```

```
635     Obj *args = eval_list(root, env, list);
636     if (args->car->type != TCELL || args->cdr != Nil)
637         error("Malformed car");
638     return args->car->car;
639 }

640 // (cdr <cell>)
641 static Obj *prim_cdr(void *root, Obj **env, Obj **list) {
642     Obj *args = eval_list(root, env, list);
643     if (args->car->type != TCELL || args->cdr != Nil)
644         error("Malformed cdr");
645     return args->car->cdr;
646 }

647 // (setq <symbol> expr)
648 static Obj *prim_setq(void *root, Obj **env, Obj **list) {
649     if (length(*list) != 2 || (*list)->car->type != TSYMBOL)
650         error("Malformed setq");
651     DEFINE2(bind, value);
652     *bind = find(env, (*list)->car);
653     if (!*bind)
654         error("Unbound variable %s", (*list)->car->name);
655     *value = (*list)->cdr->car;
656     *value = eval(root, env, value);
657     (*bind)->cdr = *value;
658     return *value;
659 }

660 // (setcar <cell> expr)
661 static Obj *prim_setcar(void *root, Obj **env, Obj **list) {
662     DEFINE1(args);
663     *args = eval_list(root, env, list);
664     if (length(*args) != 2 || (*args)->car->type != TCELL)
665         error("Malformed setcar");
666     (*args)->car->car = (*args)->cdr->car;
667     return (*args)->car;
668 }

669 // (while cond expr ...)
670 static Obj *prim_while(void *root, Obj **env, Obj **list) {
671     if (length(*list) < 2)
672         error("Malformed while");
673     DEFINE2(cond, exprs);
674     *cond = (*list)->car;
675     while (eval(root, env, cond) != Nil) {
676         *exprs = (*list)->cdr;
677         eval_list(root, env, exprs);
678     }
679     return Nil;
680 }

681 // (gensym)
682 static Obj *prim_gensym(void *root, Obj **env, Obj **list) {
683   static int count = 0;
684   char buf[10];
685   snprintf(buf, sizeof(buf), "G__%d", count++);
686   return make_symbol(root, buf);
687 }

688 // (+ <integer> ...)
689 static Obj *prim_plus(void *root, Obj **env, Obj **list) {
690     int sum = 0;
691     for (Obj *args = eval_list(root, env, list); args != Nil; args = args->cdr) {
692         if (args->car->type != TINT)
693             error("+ takes only numbers");
694         sum += args->car->value;
695     }
696     return make_int(root, sum);
697 }

698 // (- <integer> ...)
699 static Obj *prim_minus(void *root, Obj **env, Obj **list) {
700     Obj *args = eval_list(root, env, list);
701     for (Obj *p = args; p != Nil; p = p->cdr)
702         if (p->car->type != TINT)
703             error("- takes only numbers");
704     if (args->cdr == Nil)
705         return make_int(root, -args->car->value);
706     int r = args->car->value;
707     for (Obj *p = args->cdr; p != Nil; p = p->cdr)
```

```
708         r -= p->car->value;
709     return make_int(root, r);
710 }

711 // (< <integer> <integer>)
712 static Obj *prim_lt(void *root, Obj **env, Obj **list) {
713     Obj *args = eval_list(root, env, list);
714     if (length(args) != 2)
715         error("malformed <");
716     Obj *x = args->car;
717     Obj *y = args->cdr->car;
718     if (x->type != TINT || y->type != TINT)
719         error("< takes only numbers");
720     return x->value < y->value ? True : Nil;
721 }

722 static Obj *handle_function(void *root, Obj **env, Obj **list, int type) {
723     if ((*list)->type != TCELL || !is_list((*list)->car) || (*list)->cdr->type != TCELL)
724         error("Malformed lambda");
725     Obj *p = (*list)->car;
726     for (; p->type == TCELL; p = p->cdr)
727         if (p->car->type != TSYMBOL)
728             error("Parameter must be a symbol");
729     if (p != Nil && p->type != TSYMBOL)
730         error("Parameter must be a symbol");
731     DEFINE2(params, body);
732     *params = (*list)->car;
733     *body = (*list)->cdr;
734     return make_function(root, env, type, params, body);
735 }

736 // (lambda (<symbol> ...) expr ...)
737 static Obj *prim_lambda(void *root, Obj **env, Obj **list) {
738     return handle_function(root, env, list, TFUNCTION);
739 }

740 static Obj *handle_defun(void *root, Obj **env, Obj **list, int type) {
741     if ((*list)->car->type != TSYMBOL || (*list)->cdr->type != TCELL)
742         error("Malformed defun");
743     DEFINE3(fn, sym, rest);
744     *sym = (*list)->car;
745     *rest = (*list)->cdr;
746     *fn = handle_function(root, env, rest, type);
747     add_variable(root, env, sym, fn);
748     return *fn;
749 }

750 // (defun <symbol> (<symbol> ...) expr ...)
751 static Obj *prim_defun(void *root, Obj **env, Obj **list) {
752     return handle_defun(root, env, list, TFUNCTION);
753 }

754 // (define <symbol> expr)
755 static Obj *prim_define(void *root, Obj **env, Obj **list) {
756     if (length(*list) != 2 || (*list)->car->type != TSYMBOL)
757         error("Malformed define");
758     DEFINE2(sym, value);
759     *sym = (*list)->car;
760     *value = (*list)->cdr->car;
761     *value = eval(root, env, value);
762     add_variable(root, env, sym, value);
763     return *value;
764 }

765 // (defmacro <symbol> (<symbol> ...) expr ...)
766 static Obj *prim_defmacro(void *root, Obj **env, Obj **list) {
767     return handle_defun(root, env, list, TMACRO);
768 }

769 // (macroexpand expr)
770 static Obj *prim_macroexpand(void *root, Obj **env, Obj **list) {
771     if (length(*list) != 1)
772         error("Malformed macroexpand");
773     DEFINE1(body);
774     *body = (*list)->car;
775     return macroexpand(root, env, body);
776 }

777 // (println expr)
778 static Obj *prim_println(void *root, Obj **env, Obj **list) {
```

```
779         DEFINE1(tmp);
780         *tmp = (*list)->car;
781         print(eval(root, env, tmp));
782         printf("\n");
783         return Nil;
784     }

785     // (if expr expr expr ...)
786     static Obj *prim_if(void *root, Obj **env, Obj **list) {
787         if (length(*list) < 2)
788             error("Malformed if");
789         DEFINE3(cond, then, els);
790         *cond = (*list)->car;
791         *cond = eval(root, env, cond);
792         if (*cond != Nil) {
793             *then = (*list)->cdr->car;
794             return eval(root, env, then);
795         }
796         *els = (*list)->cdr->cdr;
797         return *els == Nil ? Nil : progn(root, env, els);
798     }

799     // (= <integer> <integer>)
800     static Obj *prim_num_eq(void *root, Obj **env, Obj **list) {
801         if (length(*list) != 2)
802             error("Malformed =");
803         Obj *values = eval_list(root, env, list);
804         Obj *x = values->car;
805         Obj *y = values->cdr->car;
806         if (x->type != TINT || y->type != TINT)
807             error("= only takes numbers");
808         return x->value == y->value ? True : Nil;
809     }

810     // (eq expr expr)
811     static Obj *prim_eq(void *root, Obj **env, Obj **list) {
812         if (length(*list) != 2)
813             error("Malformed eq");
814         Obj *values = eval_list(root, env, list);
815         return values->car == values->cdr->car ? True : Nil;
816     }

817     static void add_primitive(void *root, Obj **env, char *name, Primitive *fn) {
818         DEFINE2(sym, prim);
819         *sym = intern(root, name);
820         *prim = make_primitive(root, fn);
821         add_variable(root, env, sym, prim);
822     }

823     static void define_constants(void *root, Obj **env) {
824         DEFINE1(sym);
825         *sym = intern(root, "t");
826         add_variable(root, env, sym, &True);
827     }

828     static void define_primitives(void *root, Obj **env) {
829         add_primitive(root, env, "quote", prim_quote);
830         add_primitive(root, env, "cons", prim_cons);
831         add_primitive(root, env, "car", prim_car);
832         add_primitive(root, env, "cdr", prim_cdr);
833         add_primitive(root, env, "setq", prim_setq);
834         add_primitive(root, env, "setcar", prim_setcar);
835         add_primitive(root, env, "while", prim_while);
836         add_primitive(root, env, "gensym", prim_gensym);
837         add_primitive(root, env, "+", prim_plus);
838         add_primitive(root, env, "-", prim_minus);
839         add_primitive(root, env, "<", prim_lt);
840         add_primitive(root, env, "define", prim_define);
841         add_primitive(root, env, "defun", prim_defun);
842         add_primitive(root, env, "defmacro", prim_defmacro);
843         add_primitive(root, env, "macroexpand", prim_macroexpand);
844         add_primitive(root, env, "lambda", prim_lambda);
845         add_primitive(root, env, "if", prim_if);
846         add_primitive(root, env, "=", prim_num_eq);
847         add_primitive(root, env, "eq", prim_eq);
848         add_primitive(root, env, "println", prim_println);
849     }

850     //======================================================================
851     // Entry point
```

```
852 //======================================================================

853 // Returns true if the environment variable is defined and not the empty string.
854 static bool getEnvFlag(char *name) {
855     char *val = getenv(name);
856     return val && val[0];
857 }

858 int main(int argc, char **argv) {
859     // Debug flags
860     debug_gc = getEnvFlag("MINILISP_DEBUG_GC");
861     always_gc = getEnvFlag("MINILISP_ALWAYS_GC");

862     // Memory allocation
863     memory = alloc_semispace();

864     // Constants and primitives
865     Symbols = Nil;
866     void *root = NULL;
867     DEFINE2(env, expr);
868     *env = make_env(root, &Nil, &Nil);
869     define_constants(root, env);
870     define_primitives(root, env);

871     // The main loop
872     for (;;) {
873         *expr = read_expr(root);
874         if (!*expr)
875             return 0;
876         if (*expr == Cparen)
877             error("Stray close parenthesis");
878         if (*expr == Dot)
879             error("Stray dot");
880         print(eval(root, env, expr));
881         printf("\n");
882     }
883 }
```