```
#include "base_fct.h"
#include "list_add.h"
#include "errors.h"

#define BASE_FCT_NB 31


enum {ARCCOSINE, ARCSINE, ARCTANGENT, COSINE, PLUS, MINUS, MULT, DIVIDE, CAR,
      CDR, CONS, EQ, EQUAL, EQBIS, EVAL, EMPTY, ATOM, LOWER, GREATER, LEQ,
      GEQ, MIN, MAX, NEQ, PRINT, READ, SINE, TANGENT, LENGTH, SQRT, END};

struct base_fct_st{
  char name[8];
  list (*fct)(list arg);
};

typedef struct base_fct_st base_fct;

base_fct base_fct_tab[BASE_FCT_NB];

int is_cr_form(char *name)
{
  int i,l;
  l = strlen(name);
  if (name[0] != 'c') return FALSE;
  if (name[l-1] != 'r') return FALSE;
  if (l < 3) return FALSE;
  for(i = 1; i < l-2; i++)
    if ((name[i] != 'a') && (name[i] != 'd')) return FALSE;
  return TRUE;
}

list lisp_cr_form(char *name,list arg)
{
  list r;
  int l;
  int i;
  l = strlen(name);
  if (length(arg) != 1) return argnberror(name,1);
  r = car(arg);
  for(i = l-2; i > 0; i--) {
    if (is_empty(r)) {
      fprintf(stderr,"** %s : cannot take the %s of ",name,name);
      print_list(car(arg));
      printf("\n");
      raise(MYSIGERROR);
      }
    if (name[i] == 'a') r = car(r);
    else r = cdr(r);
  }
  return r;
}

list lisp_plus(list arg)
{
  if (is_empty(arg)) return num_to_list(0);
  if (!is_a_num(car(arg))) argtyperror("** + : not a number : ", car(arg));
  return num_to_list(list_to_num(car(arg)) + list_to_num(lisp_plus(cdr(arg))));
}

list lisp_minus(list arg)
{
  if (is_empty(arg)) return num_to_list(0);
  if (!is_a_num(car(arg))) argtyperror("** - : not a number : ", car(arg));
  if (length(arg) == 1)
    return num_to_list(-list_to_num(car(arg)));
  return num_to_list(list_to_num(car(arg)) - list_to_num(lisp_plus(cdr(arg))));
}

list lisp_mult(list arg)
{
  if (is_empty(arg)) return num_to_list(1);
  if (!is_a_num(car(arg))) argtyperror("** * : not a number : ", car(arg));
  return num_to_list(list_to_num(car(arg)) * list_to_num(lisp_mult(cdr(arg))));
}

list lisp_divide(list arg)
{
  if (is_empty(arg)) return num_to_list(1);
  if (!is_a_num(car(arg))) argtyperror("** / : not a number : ", car(arg));
  if (length(arg) == 1)
    return num_to_list(1/list_to_num(car(arg)));
  return num_to_list(list_to_num(car(arg)) / list_to_num(lisp_mult(cdr(arg))));
}

list lisp_sine(list arg)
{
  if (length(arg) != 1) return argnberror("sin",1);
  return num_to_list(sin(list_to_num(car(arg))));
}


list lisp_tangent(list arg)
{
  if (length(arg) != 1) return argnberror("tan",1);
  return num_to_list(tan(list_to_num(car(arg))));
}

list lisp_cosine(list arg)
{
  if (length(arg) != 1) return argnberror("cos",1);
  return num_to_list(cos(list_to_num(car(arg))));
}

list lisp_arcsine(list arg)
{
  if (length(arg) != 1) return argnberror("asin",1);
  return num_to_list(asin(list_to_num(car(arg))));
}


list lisp_arctangent(list arg)
{
  if (length(arg) != 1) return argnberror("atan",1);
  return num_to_list(atan(list_to_num(car(arg))));
}

list lisp_arccosine(list arg)
{
  if (length(arg) != 1) return argnberror("acos",1);
  return num_to_list(acos(list_to_num(car(arg))));
}


list lisp_car(list arg)
{
  if (length(arg) != 1) return argnberror("car",1);
  return car(car(arg));
}

list lisp_cdr(list arg)
```

```c
{
  if (length(arg) != 1) return argnberror("cdr",1);
  return cdr(car(arg));
}

list lisp_cons(list arg)
{
  if (length(arg) != 2) return argnberror("cons",2);
  return cons(car(arg),cadr(arg));
}

list lisp_eq(list arg)
{
  list l1,l2;
  if (length(arg) != 2) return argnberror("eq",2);
  l1 = car(arg);
  l2 = cadr(arg);
  if (is_empty(l1)) {
    if (is_empty(l1)) return list_true();
    else return list_false();
  }
  if (is_a_num(l1)) {
    if (is_a_num(l2)) {
      if (list_to_num(l1) == list_to_num(l2))
        return list_true();
      else return list_false();
    }
    else return list_false();
  }
  if (is_a_name(l1)) {
    if (is_a_name(l2)) {
      if (strcmp(list_to_string(l1),list_to_string(l2)) == 0)
        return list_true();
      else return list_false();
    }
    else return list_false();
  }
  if (is_a_cell(l2)) {
    if (l1 == l2) return list_true();
    else return list_false();
  }
  return list_false();
}

int list_equal(list l1, list l2)
{
  if (is_empty(l1)) {
    if (is_empty(l1))return TRUE;
    else return FALSE;
  }
  if (is_a_num(l1)) {
    if (is_a_num(l2)) {
      if (list_to_num(l1) == list_to_num(l2))
        return TRUE;
      else return FALSE;
    }
    else return FALSE;
  }
  if (is_a_name(l1)) {
    if (is_a_name(l2)) {
      if (strcmp(list_to_string(l1),list_to_string(l2)) == 0)
        return TRUE;
      else return FALSE;
    }
    else return FALSE;
  }
  if (is_a_cell(l2)) {
    if (list_equal(car(l1),car(l2)) == TRUE) {
      if (list_equal(cdr(l1),cdr(l2)) == TRUE)
        return TRUE;
      else return FALSE;
    }
    else return FALSE;
  }
  return FALSE;
}

list lisp_equal(list arg)
{
  if (length(arg) != 2) return argnberror("equal",2);
  if (list_equal(car(arg),cadr(arg)) == TRUE)
    return list_true();
  else return list_false();
}

list lisp_neq(list arg)
{
  if (length(arg) != 2) return argnberror("neq",2);
  if (list_equal(car(arg),cadr(arg)) == FALSE)
    return list_true();
  else return list_false();
}

list lisp_eval(list arg)
{
  if (length(arg) != 1)  { // && (length(arg) !=2)
    return argnberror("eval",1);
  }
  return eval_list(car(arg));
}

list lisp_null(list arg)
{
  if (length(arg) != 1) argnberror("null",1);
  if (is_empty(car(arg))) return list_true();
  return list_false();
}

list lisp_atom(list arg)
{
  if (length(arg) != 1) return argnberror("atom",1);
  if (is_empty(car(arg))) return list_true();
  if (is_a_num(car(arg))) return list_true();
  if (is_a_name(car(arg))) return list_true();
  return list_false();
}

list lisp_lower(list arg)
{
  if (length(arg) != 2) return argnberror("<",2);
  if (!is_a_num(car(arg))) argtyperror("** < : not a number : ", car(arg));
  if (!is_a_num(cadr(arg))) argtyperror("** < : not a number : ", cadr(arg));
  if (list_to_num(car(arg)) < list_to_num(cadr(arg))) return list_true();
  return list_false();
}

list lisp_greater(list arg)
{
  if (length(arg) != 2) return argnberror(">",2);
  if (!is_a_num(car(arg))) argtyperror("** > : not a number : ", car(arg));
  if (!is_a_num(cadr(arg))) argtyperror("** > : not a number : ", cadr(arg));
  if (list_to_num(car(arg)) > list_to_num(cadr(arg))) return list_true();
```

```
  return list_false();
}
list lisp_leq(list arg)
{
  if (length(arg) != 2) return argnberror("<=",2);
  if (!is_a_num(car(arg))) argtyperror("** <= : not a number : ", car(arg));
  if (!is_a_num(cadr(arg))) argtyperror("** <= : not a number : ", cadr(arg));
  if (list_to_num(car(arg)) <= list_to_num(cadr(arg))) return list_true();
  return list_false();
}

list lisp_geq(list arg)
{
  if (length(arg) != 2) return argnberror(">=",2);
  if (!is_a_num(car(arg))) argtyperror("** >= : not a number : ", car(arg));
  if (!is_a_num(cadr(arg))) argtyperror("** >= : not a number : ", cadr(arg));
  if (list_to_num(car(arg)) >= list_to_num(cadr(arg))) return list_true();
  return list_false();
}

list lisp_min(list arg)
{
  list t;
  if (length(arg) < 1) return argnberror("min",1);
  if (!is_a_num(car(arg))) argtyperror("** min : not a number : ", car(arg));
  if (length(arg) == 1) return car(arg);
  t = lisp_min(cdr(arg));
  if (list_to_num(car(arg)) < list_to_num(t)) return car(arg);
  return t;
}

list lisp_max(list arg)
{
  list t;
  if (length(arg) < 1) return argnberror("max",1);
  if (!is_a_num(car(arg))) argtyperror("** max : not a number : ", car(arg));
  if (length(arg) == 1) return car(arg);
  t = lisp_max(cdr(arg));
  if (list_to_num(car(arg)) > list_to_num(t)) return car(arg);
  return t;
}

list lisp_length(list arg)
{
  if (length(arg) != 1) return argnberror("length",1);
  return num_to_list(length(car(arg)));
}

list lisp_print(list arg)
{
  list last;
  list p = arg;
  last = empty_list();
  while (is_a_cell(p)) {
    last = car(p);
    print_list(last);
    p = cdr(p);
  }
  printf("\n");
  return last;
}

list lisp_read(list arg)
{
  if (!is_empty(arg)) return argnberror("read",0);
  return read_list();
}

list lisp_end(list arg)
{
  raise(MYSIGEOF);
  return empty_list();
}

list lisp_sqrt(list arg)
{
  if (length(arg) != 1) return argnberror("srt",1);
  if (!is_a_num(car(arg)))
    return argtyperror("** sqrt : not a number : ",car(arg));
  return num_to_list(sqrt(list_to_num(car(arg))));
}

void init_base_fct()
{

  strcpy(base_fct_tab[PLUS].name,"+");
  base_fct_tab[PLUS].fct = &lisp_plus;

  strcpy(base_fct_tab[MINUS].name,"-");
  base_fct_tab[MINUS].fct = &lisp_minus;

  strcpy(base_fct_tab[MULT].name,"*");
  base_fct_tab[MULT].fct = &lisp_mult;

  strcpy(base_fct_tab[DIVIDE].name,"/");
  base_fct_tab[DIVIDE].fct = &lisp_divide;

  strcpy(base_fct_tab[CAR].name,"car");
  base_fct_tab[CAR].fct = &lisp_car;

  strcpy(base_fct_tab[CDR].name,"cdr");
  base_fct_tab[CDR].fct = &lisp_cdr;

  strcpy(base_fct_tab[CONS].name,"cons");
  base_fct_tab[CONS].fct = &lisp_cons;

  strcpy(base_fct_tab[EQ].name,"eq");
  base_fct_tab[EQ].fct = &lisp_eq;

  strcpy(base_fct_tab[EQUAL].name,"equal");
  base_fct_tab[EQUAL].fct = &lisp_equal;

  strcpy(base_fct_tab[EQBIS].name,"=");
  base_fct_tab[EQBIS].fct = &lisp_eq;

  strcpy(base_fct_tab[EVAL].name,"eval");
  base_fct_tab[EVAL].fct = &lisp_eval;

  strcpy(base_fct_tab[EMPTY].name,"null");
  base_fct_tab[EMPTY].fct = &lisp_null;

  strcpy(base_fct_tab[ATOM].name,"atom");
  base_fct_tab[ATOM].fct = &lisp_atom;

  // fonctions de comparaison

  strcpy(base_fct_tab[LOWER].name,"<");
  base_fct_tab[LOWER].fct = &lisp_lower;

  strcpy(base_fct_tab[GREATER].name,">");
  base_fct_tab[GREATER].fct = &lisp_greater;
```

```c
  strcpy(base_fct_tab[LEQ].name,"<=");
  base_fct_tab[LEQ].fct = &lisp_leq;

  strcpy(base_fct_tab[GEQ].name,">=");
  base_fct_tab[GEQ].fct = &lisp_geq;

  strcpy(base_fct_tab[NEQ].name,"neq");
  base_fct_tab[NEQ].fct = &lisp_neq;

  strcpy(base_fct_tab[MIN].name,"min");
  base_fct_tab[MIN].fct = &lisp_min;

  strcpy(base_fct_tab[MAX].name,"max");
  base_fct_tab[MAX].fct = &lisp_max;

  strcpy(base_fct_tab[PRINT].name,"print");
  base_fct_tab[PRINT].fct = &lisp_print;

  strcpy(base_fct_tab[READ].name,"read");
  base_fct_tab[READ].fct = &lisp_read;

  strcpy(base_fct_tab[END].name,"end");
  base_fct_tab[END].fct = &lisp_end;

    // fonctions trigo

  strcpy(base_fct_tab[SINE].name,"sin");
  base_fct_tab[SINE].fct = &lisp_sine;

  strcpy(base_fct_tab[COSINE].name,"cos");
  base_fct_tab[COSINE].fct = &lisp_cosine;

  strcpy(base_fct_tab[TANGENT].name,"tan");
  base_fct_tab[TANGENT].fct = &lisp_tangent;

  strcpy(base_fct_tab[ARCSINE].name,"asin");
  base_fct_tab[ARCSINE].fct = &lisp_arcsine;

  strcpy(base_fct_tab[ARCCOSINE].name,"acos");
  base_fct_tab[ARCCOSINE].fct = &lisp_arccosine;

  strcpy(base_fct_tab[ARCTANGENT].name,"atan");
  base_fct_tab[ARCTANGENT].fct = &lisp_arctangent;

  strcpy(base_fct_tab[LENGTH].name,"length");
  base_fct_tab[LENGTH].fct = &lisp_length;

  strcpy(base_fct_tab[SQRT].name,"sqrt");
  base_fct_tab[SQRT].fct = &lisp_sqrt;
}

list apply_base_function(char *name, list arg, environment env, int traced)
{
  int i;
  for(i = 0; i < BASE_FCT_NB; i++) {
    if (strcmp(base_fct_tab[i].name, name) == 0)
      return base_fct_tab[i].fct(arg);
  }
  if (is_cr_form(name)) return lisp_cr_form(name,arg);
  fprintf(stderr,"** eval : not a function : %s\n", name);
  raise(MYSIGERROR);
  return empty_list();
}
```

```c
#include "environment.h"
#include "list_add.h"
#include "errors.h"

environment create_env(int l)
{
  env_item *p;
  p = (env_item *) malloc(sizeof(env_item));
  assert(p != NULL);
  p->name = (char *) malloc(l*sizeof(char));
  assert(p->name != NULL);
  return p;
}

int is_bounded(char *name, environment local_env)
{
  environment p;
  p = local_env;   // On regarde d'abord dans les variables locales
  while (p != EMPTYENV) {
    if (strcmp(name,p->name) == 0)
      return (p->traced)? 2 : 1; // pour éviter le reparcours de tout l'environnement
    p = p->next;
  }
  p = it;   // On regarde ensuite dans les variables globales
  while (p != EMPTYENV) {
    if (strcmp(name,p->name) == 0)
      return (p->traced)? 2 : 1; // pour éviter le reparcours de tout l'environnement
    p = p->next;
  }
  return FALSE;
}

list get_value(char *name, environment local_env)
{
  environment p;
  p = local_env;
  if (strcmp(name,"t") == 0) return list_true();
  while (p != EMPTYENV) {
    if (strcmp(name,p->name) == 0) return p->value;
    p = p->next;
  }
  p = it;
  while (p != EMPTYENV) {
    if (strcmp(name,p->name) == 0) return p->value;
    p = p->next;
  }
  fprintf(stderr,"** eval : undefined variable : %s\n",name);
  raise(MYSIGERROR);
  return NULL;
}

void trace(char *name, environment local_env)
{
  environment p;
  p = local_env;
  if (strcmp(name,"t") == 0) return;
  while (p != EMPTYENV) {
    if (strcmp(name,p->name) == 0) { p->traced = TRUE; return;}
    p = p->next;
  }
  p = it;
  while (p != EMPTYENV) {
    if (strcmp(name,p->name) == 0) { p->traced = TRUE; return;}
    p = p->next;
  }
  argerror("** trace : Undefined variable : report bug to cmenier@ens-lyon.fr\n");
}

void untrace(char *name, environment local_env)
{
  environment p;
  p = local_env;
  if (strcmp(name,"t") == 0) return;
  while (p != EMPTYENV) {
    if (strcmp(name,p->name) == 0) { p->traced = FALSE; return; }
    p = p->next;
  }
  p = it;
  while (p != EMPTYENV) {
    if (strcmp(name,p->name) == 0) { p->traced = FALSE; return; }
    p = p->next;
  }
  argerror("** untrace : Undefined variable : report bug to cmenier@ens-lyon.fr\n");
}

environment set_value(char *name, list l, environment env)
{
  environment p;
  if (strcmp(name,"t") == 0) {
    (void) argerror("** set_value : not a variable : t\n");
    return EMPTYENV;
  }
  p = create_env(strlen(name));
  strcpy(p->name,name);
  p->value = l;   // Copie nécessaire ?
  p->next = env;
  p->traced = FALSE;
  return p;
}

environment set_list_value(list l, list v, environment env, int traced)
{
  environment p = env;
  list l_aux,v_aux;
  l_aux = l;
  v_aux = v;
  while ((is_a_cell(l_aux)) && (is_a_cell(v_aux))) {
    if (traced) {
      print_list(car(l_aux));
      printf(" = ");
      print_list(car(v_aux));
      printf(" , ");
    }
    p = set_value(list_to_string(car(l_aux)),car(v_aux),p);
    l_aux = cdr(l_aux);
    v_aux = cdr(v_aux);
  }
  if (traced) printf("\n");
  if (!is_empty(v_aux)) {
    (void) argerror("Too many arguments\n");
    return EMPTYENV;
  }
  if (!is_empty(l_aux)) {
    (void) argerror("Too few arguments\n");
    return EMPTYENV;
  }
  return p;
}

environment set_list_let_value(list l, environment env)
{
  environment p = env;
```

```
  list l_aux;
  l_aux = l;
  while (is_a_cell(l_aux)) {
    if (length(car(l_aux)) != 2) {
      (void) argtyperror("** let : wrong type of arguments : ",car(l_aux));
      return EMPTYENV;
    }
    if (!is_a_var(caar(l_aux))) {
      (void) argtyperror("** let : not a variable : ", caar(l_aux));
      return EMPTYENV;
    }
    p = set_value(list_to_string(car(car(l_aux))),eval(cadar(l_aux), env),p);
    l_aux = cdr(l_aux);
  }
  return p;
}

void free_env(int n, environment env)
{
  int i;
  environment p;
  p = env;
  for(i =0; i < n; i++)
    {
      environment old;
      if (p == EMPTYENV) break;
      old =p;
      p = p->next;
      free(old->name);
      free(old);
    }
}
```

```c
#include "errors.h"
#include <signal.h>
#include "base_fct.h"
#include "keyword_fct.h"
//#include <stdlib.h>

sigset_t sigerror;

void sigerrorhandler(int sig)
{
  sigprocmask(SIG_UNBLOCK,&sigerror,NULL);
  longjmp(try_buf,1);
}

void sigeofhandler(int sig)
{
  printf("See you later alligator!\n");
  exit(1);
}

void init_sig()
{
  struct sigaction sa;
  sa.sa_handler = sigerrorhandler;
  sa.sa_flags = 0;
  sigemptyset(&(sa.sa_mask));
  sigaction(MYSIGERROR,&sa,NULL);

  sigemptyset(&sigerror);
  sigaddset(&sigerror,MYSIGERROR);

  sa.sa_handler = sigeofhandler;
  sa.sa_flags = 0;
  sigemptyset(&(sa.sa_mask));
  sigaction(MYSIGEOF,&sa,NULL);
}

void init()
{
  init_sig();
  init_base_fct();
  init_keyword_fct();
}

list argnberror(char *name, int n)
{
  fprintf(stderr,"** %s : wrong number of arguments : %d\n",name,n);
  raise(MYSIGERROR);
  return empty_list();
}

list argerror(char *message)
{
  fprintf(stderr,"%s",message);
  raise(MYSIGERROR);
  return empty_list();
}

list argtyperror(char *message, list l)
{
  fprintf(stderr, "%s", message);
  print_list(l);
  printf("\n");
  raise(MYSIGERROR);
  return empty_list();
}
```

```c
#include "lisp.h"
#include "environment.h"
#include "errors.h"
#include "list_add.h"
#include "base_fct.h"
#include "keyword_fct.h"

environment it = EMPTYENV;

int traceall = FALSE;

int count;

void init_count()
{
  count = 0;
}

void incr()
{
  count++;
}

void decr()
{
  count--;
}

void print_string_level(char *s)
{
  int i;
  printf("\t");
  for(i = 0; i < count; i++)
    printf(" ");
  printf("%d",count);
  printf("%s",s);
}

list apply(list f, list arg, environment env, int traced);

list eval_wrapped(list l, environment env)
{
  if (is_empty(l) || is_a_num(l))
    return l;
  if (is_a_name(l))
    return get_value(list_to_string(l), env);
  if (is_a_cell(l)) {
    list p, args;
    p = car(l);
    if (is_a_name(p)) {
      char *s;
      s = list_to_string(p);
      if (strcmp(s,"apply") == 0) {
        if (length(l) != 3) return argnberror("apply",2);
        return apply(cadr(l), eval(caddr(l), env), env, traceall);
      }
      if (is_keyword(s))
        return apply_keyword_fct(s, l, env, traceall);
    }
    args = map(cdr(l), eval, env);
    return apply(p, args, env, traceall);
  }
  return NULL;
}

list eval(list l, environment env)
{
  list result;
  if (traceall) {
    print_string_level(" --> ");
    print_list(l);
    printf("\n");
    incr();
  }
  result = eval_wrapped(l, env);
  if (traceall) {
    decr();
    print_string_level(" <-- ");
    print_list(result);
    printf("\n");
  }
  return result;
}

list apply(list f, list arg, environment env, int traced)
{
  int t;
  if (is_a_num(f)) return argerror("**eval : cannot apply a number\n");
  if (is_empty(f)) return argerror("**eval : cannot apply nil\n");
  if (is_a_name(f)) {
    if (t = is_bounded(list_to_string(f), env)) {
      list result;
      if ((t-1) || traced) {
        print_list(f);
        printf("\t --> ");
      }
      result = apply(get_value(list_to_string(f), env), arg, env, (t-1) || traced);
      if ((t-1) || traced) {
        printf("\t<-- ");
        print_list(result);
        printf("\n");
      }
      return result;
    }
    return apply_base_function(list_to_string(f), arg, env, traced);
  }
  if (is_a_name(car(f))) {
    if (strcmp(list_to_string(car(f)), "lambda") == 0) {
      environment local_env;
      if (length(f) != 3) return argnberror("lambda",2);
      if (is_a_var(cadr(f))) {
        local_env = set_value(list_to_string(cadr(f)), arg, env);
        return eval(caddr(f), local_env);
      }
      else {
        local_env = set_list_value(cadr(f), arg, env, traced);
        return eval(caddr(f), local_env);
      }
    }
    return apply(eval(f, env), arg, env, traced);
  }
  else
    return apply(eval(f, env), arg, env, traced);
}

list eval_list(list l)
{
  init_count();
  return eval(l, EMPTYENV);
}
```

```c
#include "lisp.h"
#include "errors.h"
#include "list_add.h"
#include "environment.h"

#define KEYWORD_FCT_NB 14

enum {TRACEALL, TRACE, UNTRACE, QUOTE, IF, DEFINE, SETQ, LET, DE, LAMBDA,
      COND, WHILE, PROGN, FOR};

struct keyword_fct_st{
  char name[12];
  list (*fct)(list l, environment env);
};

typedef struct keyword_fct_st keyword_fct;

keyword_fct keyword_fct_tab[KEYWORD_FCT_NB];

char *lambda_str = "lambda";

list lisp_traceall(list l, environment env)
{
  traceall = TRUE;
  return empty_list();
}

list lisp_trace(list l, environment env)
{
  if (length(l) != 2)
    return argnberror("trace",1);
  if (!is_a_var(cadr(l)))
    return argtyperror("** trace : not a variable : ", cadr(l));
  if (!is_bounded(list_to_string(cadr(l)), env))
    return argtyperror("** trace : undefined variable : ",cadr(l));
  trace(list_to_string(cadr(l)), env);
  return cadr(l);
}

list lisp_untrace(list l, environment env)
{
  if (length(l) != 2)
    return argnberror("untrace",1);
  if (!is_a_var(cadr(l)))
    return argtyperror("** untrace : not a variable : ", cadr(l));
  if (!is_bounded(list_to_string(cadr(l)), env))
    return argtyperror("** untrace : undefined variable : ",cadr(l));
  untrace(list_to_string(cadr(l)), env);
  return cadr(l);
}

list lisp_quote(list l, environment env)
{
  if (length(cdr(l)) != 1) return argnberror("quote",1);
  return cadr(l);
}

list lisp_if(list l, environment env)
{
  if (length(cdr(l)) != 3) return argnberror("if",3);
  if (is_false(eval(cadr(l), env)))
    return eval(cadddr(l), env);
  return eval(caddr(l), env);
}

list lisp_define(list l, environment env)
{
  list p, args;
  if (length(l) != 3) return argnberror("define",2);
  p = cadr(l);
  if (!is_a_var(p)) argtyperror("** define : not a variable name : ", p);
  args = eval(caddr(l), env);
  it = set_value(list_to_string(p), args, it);
  return args;
}

list lisp_let(list l, environment env)
{
  environment local_env;
  local_env = set_list_let_value(cadr(l), env);
  return eval(caddr(l), local_env);
}

list lisp_de(list l, environment env)
{
  list args;
  if (length(l) != 4)
    return argnberror("de",3);
  if (!is_a_var(cadr(l)))
    return argtyperror("** de : not a variable : ", cadr(l));
  args = caddr(l);
  while (is_a_cell(args)) {
    if (!is_a_var(car(args)))
      return argtyperror("** de : bad parameter : ", car(args));
    args = cdr(args);
  }
  it = set_value(list_to_string(cadr(l)), cons(string_to_list(lambda_str),cddr(l)), it);
  return cadr(l);
}

list lisp_lambda(list l, environment env)
{
  return l;
}

list lisp_cond(list l, environment env)
{
  list args;
  args = cdr(l);
  while(is_a_cell(args)) {
    list c;
    c = car(args);
    if (length(c) != 2) return argnberror("cond : condition : ",2);
    if (!is_false(eval(car(c), env))) return eval(cadr(c), env);
    args = cdr(args);
  }
  return empty_list();
}

list lisp_while(list l, environment env)
{
  list cond;
  list body;
  list last = empty_list();
  if (length(cdr(l)) != 2) return argnberror(list_to_string(car(l)),2);
  cond = cadr(l);
  body = caddr(l);
  while (!is_false(eval(cond, env)))
    last = eval(body, env);
  return last;
}
```

```c
list lisp_progn(list l, environment env)
{
  list last;
  list args;
  last = empty_list();
  args = cdr(l);
  while(is_a_cell(args)) {
    last = eval(car(args), env);
    args = cdr(args);
  }
  return last;
}

list lisp_for(list l, environment env)
{
  list l1, l2;
  list last;
  list body;
  char *var;
  int i;
  environment local_env;
  if (length(l) != 5) return argnberror("for",4);
  if (!is_a_var(cadr(l)))
    return argtyperror("** for : not a variable : ", cadr(l));
  var = list_to_string(cadr(l));
  l1 = eval(caddr(l), env);
  l2 = eval(cadddr(l), env);
  body = cadddr(cdr(l));
  if (!is_a_num(l1))
    return argtyperror("** for : not a number : ", l1);
  if (!is_a_num(l2))
    return argtyperror("** for : not a number : ", l2);
  last = empty_list();
  for(i = (int) list_to_num(l1); i <= (int) list_to_num(l2); i++)
    {
      local_env = set_value(var, num_to_list(i), env);
      last = eval(body, local_env);
    }
  return last;
}

int is_keyword(char *name)
{
  int i;
  for(i = 0; i < KEYWORD_FCT_NB; i++)
    if (strcmp(keyword_fct_tab[i].name, name) == 0)
      return TRUE;
  return FALSE;
}

void init_keyword_fct()
{
  strcpy(keyword_fct_tab[TRACEALL].name,"traceall");
  keyword_fct_tab[TRACEALL].fct = &lisp_traceall;

  strcpy(keyword_fct_tab[TRACE].name,"trace");
  keyword_fct_tab[TRACE].fct = &lisp_trace;

  strcpy(keyword_fct_tab[UNTRACE].name,"untrace");
  keyword_fct_tab[UNTRACE].fct = &lisp_untrace;

  strcpy(keyword_fct_tab[QUOTE].name,"quote");
  keyword_fct_tab[QUOTE].fct = &lisp_quote;

  strcpy(keyword_fct_tab[IF].name,"if");
  keyword_fct_tab[IF].fct = &lisp_if;

  strcpy(keyword_fct_tab[DEFINE].name,"define");
  keyword_fct_tab[DEFINE].fct = &lisp_define;

  strcpy(keyword_fct_tab[SETQ].name,"setq");
  keyword_fct_tab[SETQ].fct = &lisp_define;

  strcpy(keyword_fct_tab[LET].name,"let");
  keyword_fct_tab[LET].fct = &lisp_let;

  strcpy(keyword_fct_tab[DE].name,"de");
  keyword_fct_tab[DE].fct = &lisp_de;

  strcpy(keyword_fct_tab[LAMBDA].name,"lambda");
  keyword_fct_tab[LAMBDA].fct = &lisp_lambda;

  strcpy(keyword_fct_tab[COND].name,"cond");
  keyword_fct_tab[COND].fct = &lisp_cond;

  strcpy(keyword_fct_tab[WHILE].name,"while");
  keyword_fct_tab[WHILE].fct = &lisp_while;

  strcpy(keyword_fct_tab[PROGN].name,"progn");
  keyword_fct_tab[PROGN].fct = &lisp_progn;

  strcpy(keyword_fct_tab[FOR].name,"for");
  keyword_fct_tab[FOR].fct = &lisp_for;

}

list apply_keyword_fct(char *name, list arg, environment env, int traced)
{
  int i;
  for(i = 0; i < KEYWORD_FCT_NB; i++)
    if (strcmp(keyword_fct_tab[i].name,name) == 0)
      return keyword_fct_tab[i].fct(arg, env);
  return argerror("** eval : apply_keyword_fct : report bugs to cmenier@ens-lyon.fr\n");
}
```

```c
#include "lisp.h"


static object  *
alloc_new_object()
{
    object         *p;
    p = (object *) malloc(sizeof(object));
    assert(p != NULL);
    return p;
}

list empty_list()
{
    static object   obj;
    /* TRICK: To avoid replication! */
    obj.the_type = NIL_OBJECT;
    return &obj;
}

/*
 * Observe that the empty list is a pointer **real object**, which consumes
 * storage resources. An alternative implementation would be to make it a
 * NULL pointer, or maybe a ((void *)1) pointer to make it more explicit and
 * tractable. It is considered that having NIL as a real object makes things
 * more explicit... Well?
 */


int is_empty(list l)
{
    return (l->the_type == NIL_OBJECT);
}

int is_a_num(list l)
{
    return (l->the_type == NUM_OBJECT);
}

int is_a_name(list l)
{
    return (l->the_type == NAME_OBJECT);
}

int is_a_cell(list l)
{
    return (l->the_type == CELL_OBJECT);
}

list num_to_list(double n)
{
  object         *p;
    p = alloc_new_object();
    p->the_type = NUM_OBJECT;
    (p->the_value).as_num = n;
    return p;
}

list string_to_list(char *s)
{
    object         *p;
    p = alloc_new_object();
    p->the_type = NAME_OBJECT;
    (p->the_value).as_name = s;
    return p;
}

list car(list l)
{
    if (is_empty(l)) {
        fprintf(stderr, "Cannot take the car of nil!\n");
        raise(MYSIGERROR);
    }
    if ((l->the_type) != CELL_OBJECT) {
        fprintf(stderr, "Cannot take the car of a number or a name!\n");
        raise(MYSIGERROR);
    }
    return (l->the_value).as_cell.the_car;
}

list cdr(list l)
{
    if (is_empty(l))
        return (empty_list());
    if ((l->the_type) != CELL_OBJECT) {
        fprintf(stderr, "Cannot take the cdr of a number or a name!\n");
        raise(MYSIGERROR);
    }
    return (l->the_value).as_cell.the_cdr;
}


list cons(list x, list l)
{
    object         *p;
    p = alloc_new_object();

    p->the_type = CELL_OBJECT;
    (p->the_value).as_cell.the_car = x;
    (p->the_value).as_cell.the_cdr = l;

    return p;
}

double string_to_num(char *s)
{
  int i = 0;
  double r, a;
  int e;
  double sign = 1;
  double signexp = 1;
  r = 0;
  a = 1;
  e = 0;
  if (s[0] == '+') {i++; sign = 1;}
  else if (s[0] == '-') {i++; sign = -1;}
  while (s[i] != '\0') {
    if (s[i] == '.') {i++; break;}
    if ((s[i] == 'e') || (s[i] == 'E')) break;
    r = 10*r + s[i] - '0';
    i++;
  }
  while (s[i] != '\0') {
    a = 0.1 * a;
    if ((s[i] == 'e') || (s[i] == 'E')) {i++; break;}
    r = r + a * (s[i] - '0');
    i++;
  }
  if(s[i] != '0') {
    if (s[i] == '+') {i++; signexp = 1;}
    else if(s[i] == '-') {i++; signexp = -1;}
  }
```

```c
  while (s[i] != '\0') {
    e = 10 * e + s[i] - '\0';
    i++;
  }
  return sign * r * pow(10, signexp * e);
}
```

```
#include "lisp.h"
#include "list_add.h"
#include "errors.h"

char *t_str = "t";

list caar(list l)
{
  return car(car(l));
}

list cadr(list l)
{
  return car(cdr(l));
}

list cddr(list l)
{
  return cdr(cdr(l));
}

list caddr(list l)
{
  return car(cdr(cdr(l)));
}

list cadddr(list l)
{
  return car(cdr(cdr(cdr(l))));
}

list cadar(list l)
{
  return car(cdr(car(l)));
}

int length(list l)
{
  list p;
  int i = 0;
  p = l;
  while (is_a_cell(p)) {
    i++;
    p = cdr(p);
  }
  return i;
}

char *list_to_string(list l)
{
  if (is_a_name (l))
    return l->the_value.as_name;

  argerror("This is not a string !\n");
  return NULL;
}

double list_to_num(list l)
{
  if (is_a_num(l))
    return l->the_value.as_num;
  argerror("This is not a number !\n");
  return 0;
}

int is_true(list l)
{
  if (is_a_name(l))
    if (strcmp(list_to_string(l), "t") == 0)
      return TRUE;
    else return FALSE;
  else return FALSE;
}

int is_false(list l)
{
  return is_empty(l);
}

list list_true()
{
  static object t;
  t.the_type = NAME_OBJECT;
  t.the_value.as_name = t_str;
  return &t;
}

list list_false()
{
  return empty_list();
}

/*
   void set_car(list l, list new_car)
   {
   if (is_a_cell (l))
   l->the_value.as_cell.the_car = new_car;
   else {
   fprintf (stderr, "Setting the car of something that is not a cell!\n");
   raise (MYSIGERROR);

   }
   }

   void set_cdr (list l, list new_cdr)
   {
   if (is_a_cell (l))
   l->the_value.as_cell.the_cdr = new_cdr;
   else {
   fprintf (stderr, "Setting the cdr of something that is not a cell!\n");
   raise (MYSIGERROR);

   }
   }

   list copy_list (list l)
   {
   list p, ancre; // a traduire en angliche

   if (is_empty (l))
   return empty_list();

   ancre = cons (car (l), empty_list ());
   p = ancre;
   while (!is_empty (cdr (l))) {
   set_cdr (p, cons (car (cdr (l)), empty_list ()));
   l = cdr (l);
   p = cdr (p);
   }
   return ancre;
   }
*/
```

```
int is_a_var(list l)
{
  if (!is_a_name(l)) return FALSE;
  if (strcmp(list_to_string(l),"t") == 0) return FALSE;
  return TRUE;
}

list map(list l, list (*f)(list, environment), environment env)  // Version récursive => ver
sion itérative ....
{
  list ancre;

  if (is_empty(l)) return l;
  ancre = cons(f(car(l), env), map(cdr(l), f, env));
  /*  ancre = copy_list (l);
      p = ancre;
      while (!is_empty (p)) {
      set_car (p, f(car (p), env));
      }*/
  return ancre;
}

void free_list(list l)
{
  list p;
  list old;
  p = l;
  if (is_a_cell(p)){
    free_list(car(p));
    free_list(cdr(p));
  }
  if (is_a_name(p))
    free(p->the_value.as_name);
  free(p);
}
```

```c
/* Copyright (c) 1998 by Lip Ens-Lyon.  Tous droits reverve's */

/***
   NOM
     main
   UTILISATION

   COMMENTAIRE

   HISTORIQUE
     bouge - Dec 9, 1998: Created.
***/

#include "lisp.h"
#include "setjmp.h"
#include "errors.h"

int main()
{
  list result;
  init();
  for (;;) {
    (void) setjmp(try_buf);
    printf(">\t");
    fflush(stdout);
    result = eval_list(read_list());
    printf("= ");
    print_list(result);
    print_newline();
  }
}
```

```
/* Copyright (c) 1998 by Lip Ens-Lyon.  Tous droits reverve's */

/***
   NOM
     print
   UTILISATION

   COMMENTAIRE

   HISTORIQUE
     bouge - Dec 10, 1998: Created.
***/

#include "lisp.h"

static void     print_list_internal(list);

void
print_list(list l)
{
    switch (l->the_type) {
        case NIL_OBJECT:
        printf("()");
        return;
    case NUM_OBJECT:
        printf("%g", (l->the_value).as_num);
        return;
    case NAME_OBJECT:
        printf("%s", (l->the_value).as_name);
        return;
    case CELL_OBJECT:
        printf("(");
        print_list_internal(l);
        return;
    default:
        assert(FALSE);
    }
}

static void
print_list_internal(list l)
{
    if (is_empty(l)) {
        printf(")");
        return;
    }
    if (is_empty(cdr(l))) {
        print_list(car(l));
        printf(")");
        return;
    }
    print_list(car(l));
    printf(" ");
    print_list_internal(cdr(l));
    return;
}


void
print_newline()
{
    printf("\n");
}
```

```c
/* Copyright (c) 1998 by Lip Ens-Lyon.  Tous droits reverve's */

/***
   NOM
     read
   UTILISATION

   COMMENTAIRE

   HISTORIQUE
     bouge - Dec 7, 1998: Created.
***/

#include "lisp.h"

list            read_list();

static          list
reverse_list(list l)
{
    list            acc;
    acc = empty_list();
    while (!is_empty(l)) {
        acc = cons(car(l), acc);
        l = cdr(l);
    }
    return acc;
}

/*
 * The grammar actually needs a lookahead of 1 token.  This is implemented by
 * a pair of look/get functions.  This could alternatively be simulated as
 * follows. The current token would be "pre-read" and passed to the
 * read_token_list(token) parser as a parameter. Then, the read_list topevel
 * call would merely be read_token_list(get_token()).
 */

list
read_list()
{
    token           t;
    t = get_token();

    switch (t.the_type) {
    case NUM_TOKEN:
        return num_to_list(t.the_value.as_num);
    case STRING_TOKEN:
        {
            char           *s;
            s = t.the_value.as_string;
            if (strcasecmp(s, "nil") == 0)
                return empty_list();
            else
                return string_to_list(t.the_value.as_string);
        }
    case RPAR_TOKEN:
        return read_list();
    case QUOTE_TOKEN:
        {
            list            l;
            l = read_list();
            return cons(string_to_list("quote"),
                        cons(l, empty_list()));
        }
    case LPAR_TOKEN:
        {
            list            l;
            l = empty_list();
            for (;;) {

                if (look_token().the_type == RPAR_TOKEN) {
                    (void) get_token();
                    /* Just eat it! */
                    return reverse_list(l);
                    /*
                     * List l has actually been built from tail to head!
                     */
                }
                l = cons(read_list(), l);
            }
        }
    default:
        assert(FALSE);

    }

}
```

```
#ifndef __BASE_FCT
#define __BASE_FCT

#include "lisp.h"
#include "environment.h"

void init_base_fct();

list apply_base_function(char *name, list arg, environment env, int traced);

#endif
```

```c
#ifndef __ENVIRONMENT
#define __ENVIRONMENT

#include "lisp.h"

struct env_item_struct{
  list value;
  char *name;
  int traced;
  struct env_item_struct *next;
};

#define EMPTYENV (environment) NULL

int is_bounded(char *name, environment local_env);

list get_value(char *name, environment local_env);

void trace(char *name, environment local_env);

void untrace(char *name, environment local_env);

environment set_value(char *name, list l, environment env);

environment set_list_value(list l, list v, environment env, int traced);

environment set_list_let_value(list l, environment env);

void free_env(int n, environment env);

#endif
```

```
#ifndef __ERRORS
#define __ERRORS

#include "setjmp.h"
#include "lisp.h"

void init_sig();

void init();

void try();

list argnberror(char *name, int n);

list argerror(char *message);

list argtyperror(char *message, list l);

jmp_buf try_buf;

#endif
```

```
#ifndef __KEYWORD_FCT
#define __KEYWORD_FCT

int is_keyword(char *name);

void init_keyword_fct();

list apply_keyword_fct(char *name, list arg, environment env, int traced);

#endif __KEYWORD_FCT
```

```c
/* Copyright (c) 1998 by Lip Ens-Lyon.  Tous droits reverve's */

/***
   NOM
     read
   UTILISATION

   COMMENTAIRE

   HISTORIQUE
     bouge - Dec 7, 1998: Created.
***/

#ifndef __LISP
#define __LISP

#define MYSIGEOF SIGUSR1
#define MYSIGERROR SIGUSR2

#define TRUE 1
#define FALSE 0

#include <assert.h>

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <ctype.h>
#include <strings.h>
#include <malloc.h>
#include <math.h>


enum type_token_enum {
    LPAR_TOKEN, RPAR_TOKEN, QUOTE_TOKEN, NUM_TOKEN, STRING_TOKEN
};

typedef enum type_token_enum type_token;

struct token_struct {
    type_token      the_type;
    union {
        double             as_num;
        char           *as_string;
    }               the_value;
};

typedef struct token_struct token;

token get_token();

token look_token();

enum type_object_enum {
    NIL_OBJECT, NUM_OBJECT, NAME_OBJECT, CELL_OBJECT
};

struct cell_struct;
struct object_struct;

typedef struct cell_struct cell;
typedef struct object_struct object;
typedef enum type_object_enum type_object;

typedef object *list;

/*
 * Warning: a "list" is a pointer to an object. A object may be either NIL,
 * or an INTeger, or a NAME (string), or a CELL made of two pointers to
 * objects (that is, lists, its car and its cdr). This is thus different from
 * the notion of a CAML list, where the car is an object and the cdr a list.
 */

struct cell_struct {
    object         *the_car;
    object         *the_cdr;
};

struct object_struct {
    type_object     the_type;
    union {
        double          as_num;
        char           *as_name;

        cell            as_cell;
    }               the_value;
};

struct env_item_struct;

typedef struct env_item_struct env_item;

typedef env_item * environment;

extern environment it;

extern int traceall;

extern list empty_list();

extern int  is_empty(list);

extern int  is_a_num(list);

extern int  is_a_name(list);

extern int  is_a_cell(list);

extern list num_to_list(double);

extern list string_to_list(char *);

extern list car(list);

extern list cdr(list);

extern list cons(list, list);

extern void print_newline();

extern void print_list(list);

extern list read_list();

extern list eval_list(list);

extern list eval(list, struct env_item_struct *);

double string_to_num(char *s);

#endif
```

```
#ifndef __LIST_ADD
#define __LIST_ADD

#include "environment.h"

list caar(list l);

list cadr(list l);

list cddr(list l);

list caddr(list l);

list cadddr(list l);

list cadar(list l);

int length(list l);

char *list_to_string(list l);

double list_to_num(list l);

int is_true(list l);

int is_false(list l);

list list_true();

list list_false();

int is_a_var(list l);

list map(list l, list (*f)(list, environment), environment env);

void free_list(list l);

#endif
```

```
%{

  /* Fonctions de lecture des tokens réécrites avec lex
     Laure Danthony, Ménier Clément
  */

#include "lisp.h"
#include "errors.h"

static int is_empty_token = TRUE;

token x;

%}

%%
[\n\t ] ;
[(] {x.the_type = LPAR_TOKEN; return 0;}
[)] {x.the_type = RPAR_TOKEN; return 0;}
[\'] {x.the_type = QUOTE_TOKEN; return 0;}
[-+]?([0-9]+|([0-9]*\.[0-9]+))([eE][-+]?[0-9]+)? {
  x.the_type = NUM_TOKEN;
  x.the_value.as_num = string_to_num(yytext);
  return 0;}
[^\"()\'\n\t ]+ {
  char *s;
  x.the_type = STRING_TOKEN;
  s = (char *) malloc(yyleng + 1);
  assert(s != NULL);
  strcpy(s, yytext);
  x.the_value.as_string = s;
  return 0;}
\".+\" {
  char *s;
  x.the_type = STRING_TOKEN;
  s = (char *) malloc(yyleng - 1);
  assert(s != NULL);
  yytext[yyleng - 1] = '\0';
  strcpy(s, yytext + 1);
  x.the_value.as_string = s;
  return 0;}
. {(void) argerror("Not a valid command \n"); return 1;}
%%

token look_token()
{
  if (is_empty_token == FALSE) return x;
  is_empty_token = FALSE;
  yylex();
  return x;
}

token get_token()
{
  token t;
  t = look_token();
  is_empty_token = TRUE;
  return t;
}
```

```
.PHONY: all indent print clean

source:= $(wildcard *.c)

include:= $(wildcard *.h)

object:= $(subst .c,.o,$(source)) token.o

exe:=lisp

all: $(exe)

lisp: $(object) token.o lisp.h
	gcc -g -Wall $(object) -o $(exe) -ll -lm

$(object): %.o: %.c $(include)
	gcc -g -Wall -c $<

token.c: token.lex
	rm -f token.c; lex -t token.lex >token.c

token.o: token.c token.lex
	gcc -g -Wall -c token.c

lint:
	lint $(source)

indent:
	for x in $(source) $(include); do indent $$x; done

print:
	enscript -Ec --columns=2  --borders --media=A4 --landscape --encoding=latin1 $(sourc
e) $(include) token.lex Makefile -o listing.ps
##	lpr listing.ps

tar:
	rm -f token.c ; tar -czvf lisp_c.tgz *.c *.h token.lex idees.txt Makefile

clean:
	rm *.o *~ \#*\# *.BAK *.ps $(exe) token.c
```