# Lisp notes

Dennis Yurichev

April 13, 2022

# Contents

# Chapter 1

# Cons, car and cdr


Logo by Conrad Barski, M.D.

Everything good you hear about Lisp is true, but in reality, things are even better [1]. To my opinion, any serious programmer should learn some Lisp [2].

Some of my Python code you see in my books and blog posts was actually first written in Lisp (I prefer Scheme/Racket) as prototype/sketch, and then rewritten to Python if everything works as expected – for a wider audience. (My another *sketching* PL is Wolfram Mathematica, which is very *lispy*.)

It's widely considered that a true Lisp hacker should write (toy) Lisp interpreter at least once in a lifetime, to dive deeper into all the internals.

This is true, but what I offer here is a smaller/simpler first step: only functions that handle *cons cells*. I wrote many functions in pure C and that helped me in Lisp learning.

Even more – when I used pure C for some projects, I felt I miss Lisp constructs. So I add the *cell* struct and many Lisp functions. And I felt that I stepped on a well-trodden path: "Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp." (Greenspun's tenth rule of programming.)

Hopefully, my method would help someone else.

GraphViz graphs are generated with helper functions. Download the full source code with it: https://yurichev.org/lisp/lisp.tar.

## 1.1 Cell type

Main cell types are *atoms* and *cons cells*. *Atom* can be integer, string...

Listing 1.1: cell_types.h

```
// AKA NIL AKA #f AKA '()
#define CELL_TYPE_NIL           0
#define CELL_TYPE_ATOM_INT      1
#define CELL_TYPE_ATOM_STRING   2
#define CELL_TYPE_CONS          3
```

## 1.2 Cons cell

*Cons cell* can hold atom *or* two links to other cells (*car* and *cdr*).

Since cons cell is a central Lisp structure, it can be heavily optimized – at least using *union* type.

Not uncommon is storing *cons cell* into 32/64-bit integer (also, in Lisp machines).

Cons cells are present in ML dialects (OCaml, etc) and Haskell: "head :: tail".

Listing 1.2: cons.h

```
struct cell
{
        int type;
        // atoms:
```

[1] https://lispers.org/

[2] Well, maybe you would also need some Perl: https://xkcd.com/224/.

```
        int number;     // if CELL_TYPE_ATOM_INT
        char *string;   // if CELL_TYPE_ATOM_STRING
        // there is also can be - vector/array, hash table...
        // if CELL_TYPE_CONS
        struct cell* car; // AKA head
        struct cell* cdr; // AKA tail
};

struct cell* alloc_cell()
{
        return calloc(1, sizeof (struct cell));
};

// create a cell
struct cell* cons(struct cell* car, struct cell* cdr)
{
        struct cell* new_cell=alloc_cell();
        new_cell->type=CELL_TYPE_CONS;
        new_cell->car=car;
        new_cell->cdr=cdr;
        return new_cell;
};

struct cell* car(struct cell* cell)
{
        if (cell==NULL)
                return NULL;
        if (cell->type==CELL_TYPE_NIL)
                return NULL;

        assert(cell->type==CELL_TYPE_CONS);
        return cell->car;
};

struct cell* cdr(struct cell* cell)
{
        if (cell==NULL)
                return NULL;
        if (cell->type==CELL_TYPE_NIL)
                return NULL;

        assert(cell->type==CELL_TYPE_CONS);
        return cell->cdr;
};

bool CONSP (struct cell* c)
{
        if (c==NULL)
                return false;
        return c->type==CELL_TYPE_CONS;
};

// http://clhs.lisp.se/Body/f_rplaca.htm
// AKA set-car! in Scheme
struct cell* RPLACA (struct cell* cell, struct cell* new_car)
{
        assert (CONSP(cell));
        cell->car=new_car;
        return cell;
};
```

```
// http://clhs.lisp.se/Body/f_rplaca.htm
// AKA set-cdr! in Scheme
struct cell* RPLACD (struct cell* cell, struct cell* new_cdr)
{
        assert (CONSP(cell));
        cell->cdr=new_cdr;
        return cell;
};
```

Function names ending with -P usually returns *boolean*.

## 1.3 Atom

Listing 1.3: atom.h

```
// AKA ATOMP AKA ATOM?
bool ATOM (struct cell* c)
{
        // NIL is also atom:
        if (c==NULL)
                return true;
        if (c->type==CELL_TYPE_NIL)
                return true;

        if (c->type==CELL_TYPE_ATOM_INT)
                return true;
        if (c->type==CELL_TYPE_ATOM_STRING)
                return true;
        return false;
};

struct cell* atom_int (int n)
{
        struct cell* new_cell=alloc_cell();
        new_cell->type=CELL_TYPE_ATOM_INT;
        new_cell->number=n;
        return new_cell;
};

struct cell* atom_string (const char* s)
{
        struct cell* new_cell=alloc_cell();
        new_cell->type=CELL_TYPE_ATOM_STRING;
        new_cell->string=strdup(s);
        return new_cell;
};

// http://clhs.lisp.se/Body/f_nump.htm
bool numberp (struct cell* x)
{
        return x && x->type==CELL_TYPE_ATOM_INT;
};

// http://clhs.lisp.se/Body/f_stgp.htm
bool stringp (struct cell* x)
{
        return x && x->type==CELL_TYPE_ATOM_STRING;
};

// https://docs.racket-lang.org/reference/generic-numbers.html#%28def._%28%28quote._~
    23~25kernel%29._number-~3estring%29%29
```

```
struct cell* number_to_string (struct cell* input)
{
        assert (input->type==CELL_TYPE_ATOM_INT);

        char buf[128];
        snprintf (buf, 128, "%d", input->number);
        return atom_string(buf);
};
```

## 1.4 Lists

Listing 1.4: list.h

```
bool LISTP (struct cell* c)
{
        // NIL is also list
        if (c==NULL)
                return true;
        if (c->type==CELL_TYPE_NIL)
                return true;
        if (c->type==CELL_TYPE_ATOM_INT)
                return false;
        if (c->type==CELL_TYPE_ATOM_STRING)
                return false;
        if (CONSP(c))
        {
                // 1-element list? OK
                if (cdr(c)==NULL)
                        return true;

                // check rest:
                return LISTP(cdr(c));
        };

        assert (!"unknown type");
};

// iterative
int length_i (struct cell* l)
{
        int rt=0;

        // NIL is also (empty) list
        if (l==NULL)
                return 0;

        assert (LISTP(l));

        // traverse list:
        for (struct cell* i=l; i; i=cdr(i))
                rt++;

        return rt;
};

// recursive: fancy looking, but beware of stack overflow
int length_r (struct cell* l)
{
        // NIL is also (empty) list
        if (l==NULL)
```

```
                return 0;

        assert (LISTP(l));

        // is this a cons like (x . NIL) ?
        if (cdr(l)==NULL)
                return 1;

        // it will work fast if written in Lisp,
        // because this code can be converted to tail recursion.
        // but not in pure C, of course.
        return 1+length_r(cdr(l));
};

// shouldn't be used at all...
// but I use it yet another time to demonstrate cons cells
bool length_is_1 (struct cell* l)
{
        if (l==NULL)
                return false;

        assert (CONSP(l));

        if (cdr(l)==NULL)
                return true;
        return false;
};

// return last cons cell
// http://clhs.lisp.se/Body/f_last.htm
// AKA last-pair in Scheme:
// https://docs.racket-lang.org/reference/pairs.html#%28def._%28%28lib._racket%2Flist
    ..rkt%29._last-pair%29%29
/*
2.28. Using just CAR and CDR, is it possible to write a function that returns
the last element of a list, no matter how long the list is? Explain.
( David S. Touretzky -- COMMON LISP: A Gentle Introduction to Symbolic Computation )
*/
struct cell* LAST(struct cell* l)
{
        assert (CONSP(l));
        if (cdr(l)==NULL)
                return l;
        else
                return LAST(cdr(l));
};

// AKA "consing"
struct cell* create_list1 (struct cell* a1)
{
        return cons(a1, NULL);
};

struct cell* create_list2(struct cell* a1, struct cell* a2)
{
        return cons(a1, cons(a2, NULL));
};

struct cell* create_list3(struct cell* a1, struct cell* a2, struct cell* a3)
{
        return cons(a1, cons(a2, cons(a3, NULL)));
```

```
};

struct cell* create_list4(struct cell* a1, struct cell* a2, struct cell* a3, struct
    cell* a4)
{
        return cons(a1, cons(a2, cons(a3, cons(a4, NULL))));
};

struct cell* create_list_of_1_str (const char* a1)
{
        return cons(atom_string(a1), NULL);
};

struct cell* create_list_of_2_strs (const char* a1, const char* a2)
{
        return cons(atom_string(a1), cons(atom_string(a2), NULL));
};

struct cell* create_list_of_3_strs (const char* a1, const char* a2, const char* a3)
{
        return cons(atom_string(a1), cons(atom_string(a2), cons(atom_string(a3), NULL
            )));
};

struct cell* create_list_of_4_strs (const char* a1, const char* a2, const char* a3,
    const char* a4)
{
        return cons(atom_string(a1), cons(atom_string(a2), cons(atom_string(a3), cons
            (atom_string(a4), NULL))));
};

struct cell* create_list_of_5_strs (const char* a1, const char* a2, const char* a3,
    const char* a4, const char* a5)
{
        return cons(atom_string(a1), cons(atom_string(a2), cons(atom_string(a3), cons
            (atom_string(a4), cons(atom_string(a5), NULL)))));
};

struct cell* create_list_of_1_int (const int a1)
{
        return cons(atom_int(a1), NULL);
};

struct cell* create_list_of_3_ints (const int a1, const int a2, const int a3)
{
        return cons(atom_int(a1), cons(atom_int(a2), cons(atom_int(a3), NULL)));
};

struct cell* create_list_of_5_ints (const int a1, const int a2, const int a3, const
    int a4, const int a5)
{
        return cons(atom_int(a1), cons(atom_int(a2), cons(atom_int(a3), cons(atom_int
            (a4), cons(atom_int(a5), NULL)))));
};

// http://clhs.lisp.se/Body/f_mk_lis.htm
// TODO: initial element may be also set!
struct cell* make_list(int n)
{
        struct cell* tmp=NULL;
        for (int i=0; i<n; i++)
```

```
                tmp=cons (NULL, tmp);
        assert (length_i(tmp)==n);
        return tmp;
};

void dotted_pair_test()
{
        struct cell* tmp;

        // dotted pair:
        tmp=cons(atom_int(1), atom_int(2));
        assert_string_repr_is(tmp, false, "(1 . 2)");

        /*
        2.34. Write an expression involving cascaded calls to CONS to construct the
        dotted list (A B C . D).
        David S. Touretzky -- COMMON LISP: A Gentle Introduction to Symbolic
           Computation

        CL-USER> '(a b (c . d))
        (A B (C . D))
        */

        tmp=cons(atom_string("a"),
                cons(atom_string("b"),
                        cons(atom_string("c"), atom_string("d"))));
        // ("a" . ("b" . ("c" . "d")))
        assert_string_repr_is(tmp, true, "(\"a\" . (\"b\" . (\"c\" . \"d\")))");
        dump_graphviz(tmp, "dotted_pair_test");
};


// expression: (a+b)/(x*y)
// list of 3 elements
// 1st element: a+b
// 2nd element: /
// 3rd element: x*y
void nested_tree_test()
{
        struct cell* tmp=create_list3(
                create_list_of_3_strs("a", "+", "b"),
                create_list_of_1_str("/"),
                create_list_of_3_strs("x", "*", "y"));
        // (("a" "+" "b") ("/") ("x" "*" "y"))
        assert_string_repr_is(tmp, false, "((\"a\" \"+\" \"b\") (\"/\") (\"x\" \"*\"
           \"y\"))");
        dump_graphviz(tmp, "nested_tree_test");
};
```

Figure 1.1: dotted_pair_test() output

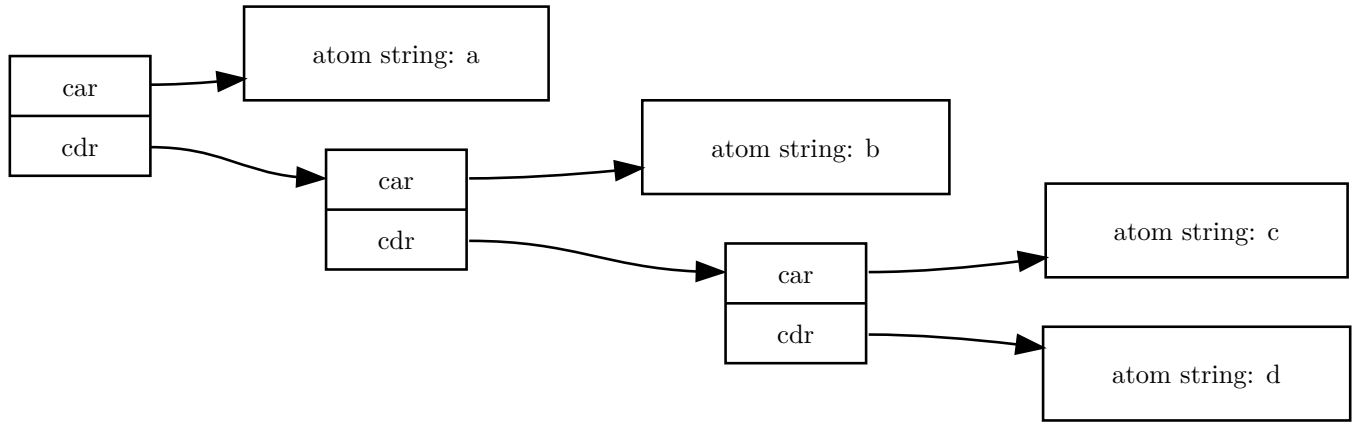
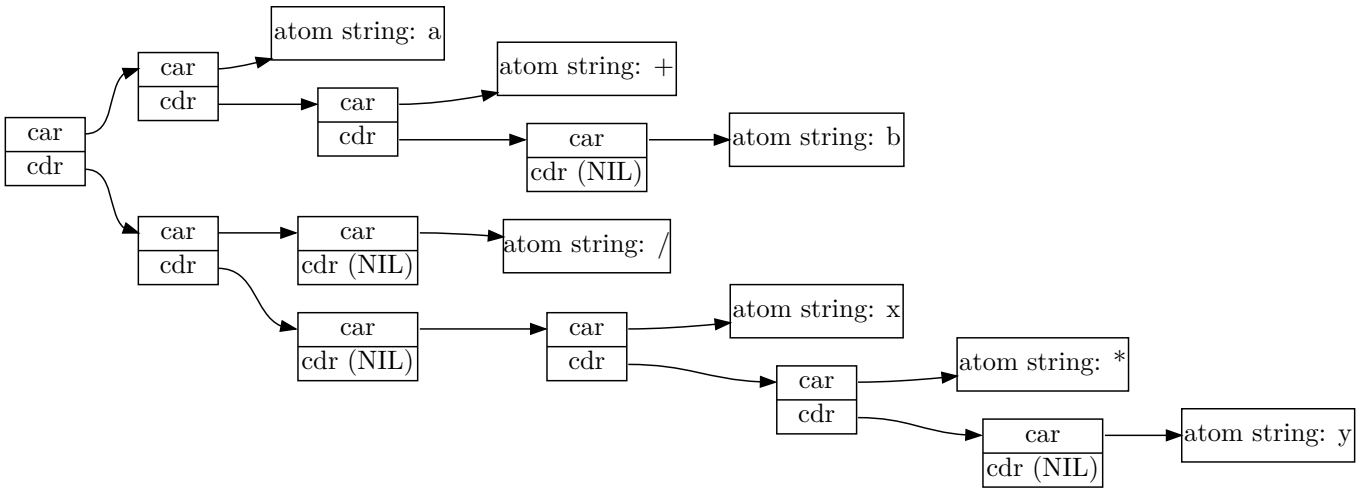



Figure 1.2: nested_tree_test() output




Since any item of list can have any type, there are no tuples in Lisp – lists can be used instead.

## 1.5 Equality

Listing 1.5: eq.h

```
// http://www.lispworks.com/documentation/lw71/CLHS/Body/f_eq.htm
// obviously, very fast
bool eq(struct cell* x, struct cell* y)
{
        // compare only pointers:
        return x==y;
};

// http://www.lispworks.com/documentation/lw71/CLHS/Body/f_eql.htm
// slower
// AKA eqv in Scheme
bool eql(struct cell* x, struct cell* y)
{
        // true if the same address or if equal atoms
        if (eq(x, y))
                return true;
        if (x->type==y->type)
        {
```

```
                if (CONSP(x))
                        // don't compare CONS cells:
                        return false;

                if (x->type==CELL_TYPE_ATOM_INT)
                        return x->number == y->number;
                if (x->type==CELL_TYPE_ATOM_STRING)
                        return strcmp (x->string, y->string)==0;

                assert (!"unsupported atom type");
        };
        return false;
};

// http://www.lispworks.com/documentation/lw71/CLHS/Body/f_equal.htm
// slowest
bool equal(struct cell* x, struct cell* y)
{
        if (CONSP(x) && CONSP(y))
                return equal(car(x), car(y)) &&
                        equal(cdr(x), cdr(y));

        if (ATOM(x) && ATOM(y) && eql(x, y))
                return true;

        return false;
};
```

All this is not unique to Lisp.

Python has "x is y" which is syntactic sugar for "id(x) == id(y)" [3]. It is comparison of two object's addresses.

## 1.6 range function

Listing 1.6: range.h

```
// create list of integers: [begin...end)
// function from Scheme
struct cell* range(int begin, int end)
{
        struct cell* rt=alloc_cell();
        struct cell* last=rt;

        for (int i=begin; i<end; i++)
        {
                last->type=CELL_TYPE_CONS;
                last->car=atom_int(i);
                if (i+1==end)
                        last->cdr=NULL;
                else
                {
                        last->cdr=alloc_cell();
                        last=last->cdr;
                };
        };
        return rt;
};

// fancy recursive version, shouldn't be used:
struct cell* range_v2(int begin, int end)
```

[3] https://docs.python.org/3/reference/expressions.html#is

```
{
        if (begin==end-1)
                return create_list_of_1_int(begin);

        return append(range(begin, end-1),
                        create_list_of_1_int(end-1));
};

void range_test()
{
        assert_string_repr_is(range(1, 6), false, "(1 2 3 4 5)");
        assert_string_repr_is(range_v2(1, 6), false, "(1 2 3 4 5)");
};
```

## 1.7 Alist

Listing 1.7: assoc.h

```
// http://www.lispworks.com/documentation/lw71/CLHS/Body/m_push.htm
struct cell* push (struct cell* new_head, struct cell* list)
{
        // it's like adding an element before a singly-linked list in pure C:
        return cons(new_head, list);
};

// alist in Lisp is like poor man's dictionary
// this is merely a list of cons-es
// obviously, a hash tables are to be used instead, or binary trees
// but for small dictionaries alists are OK.
// like for configuration files
// also, alist is a good demonstration of how consing works
// https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node153.html
// https://www.gnu.org/software/emacs/manual/html_node/elisp/Association-Lists.html

// http://clhs.lisp.se/Body/f_assocc.htm
struct cell* assoc (struct cell* key, struct cell* list)
{
        if (list==NULL)
                return NULL;
        assert(LISTP(list));
        struct cell* first_cons=car(list);
        assert(CONSP(first_cons));
        if (eql(key, car(first_cons))) // change car to cdr for rassoc
                return first_cons;
        else
                return assoc(key, cdr(list));
};

// http://www.lispworks.com/documentation/HyperSpec/Body/f_acons.htm
struct cell* acons(struct cell* key, struct cell* val, struct cell* dict)
{
        return push(cons(key, val), dict);
};

void assoc_test()
{
        struct cell* dict=NULL;
        // many textbooks, when describing dictionaries, use a list of employees with
            their salaries
        // this text is no exception...
```

```
        dict=push(cons(atom_string("John"),    atom_int(123)), dict);
        dict=push(cons(atom_string("Jake"),    atom_int(456)), dict);
        dict=push(cons(atom_string("Mary"),    atom_int(789)), dict);
        dict=push(cons(atom_string("Steve"),   atom_int(100)), dict);
        dict=push(cons(atom_string("Whoever"), atom_int(991)), dict);

        //print_cell(dict, false); printf ("\n");
        // (("Whoever" . 991) ("Steve" . 100) ("Mary" . 789) ("Jake" . 456) ("John" .
            123))
        assert_string_repr_is(dict, false, "((\"Whoever\" . 991) (\"Steve\" . 100)
           (\"Mary\" . 789) (\"Jake\" . 456) (\"John\" . 123))");

        //print_cell(assoc(atom_string("Mary"), dict), false); printf ("\n");
        // ("Mary" . 789)
        assert_string_repr_is(assoc(atom_string("Mary"), dict), false, "(\"Mary\" .
           789)");

        // update Mary's salary by pushing new cons, "shadowing" previous
        // (set-cdr! or RPLACD should be used instead, to modify the existing cons.)
        // acons can also be used: http://www.lispworks.com/documentation/HyperSpec/
           Body/f_acons.htm
        dict=acons(atom_string("Mary"), atom_int(1000), dict);
        //print_cell(dict, false); printf ("\n");
        // notice two Marys in list:
        // (("Mary" . 1000) ("Whoever" . 991) ("Steve" . 100) ("Mary" . 789) ("Jake"
           . 456) ("John" . 123))
        assert_string_repr_is(dict, false, "((\"Mary\" . 1000) (\"Whoever\" . 991)
           (\"Steve\" . 100) (\"Mary\" . 789) (\"Jake\" . 456) (\"John\" . 123))");
        // but this call fetches only the first Mary:
        //print_cell(assoc(atom_string("Mary"), dict), false); printf ("\n");
        // ("Mary" . 1000)
        assert_string_repr_is(assoc(atom_string("Mary"), dict), false, "(\"Mary\" .
           1000)");
};
```

## 1.8 copy-tree

Listing 1.8: copy_tree.h

```
// AKA deep copy
// http://www.lispworks.com/documentation/lw70/CLHS/Body/f_cp_tre.htm
/*
If tree is not a cons, it is returned; otherwise, the result is a new cons of the
   results of calling copy-tree on the car and cdr of tree. In other words, all
   conses in the tree represented by tree are copied recursively, stopping only when
   non-conses are encountered.
*/
// see also:
// http://www.lispworks.com/documentation/lw70/CLHS/Body/f_cp_lis.htm
// http://www.lispworks.com/documentation/lw70/CLHS/Body/f_cp_ali.htm
struct cell* copy_tree(struct cell* input)
{
        if (input==NULL)
                return NULL;
        if (ATOM(input))
                return input;

        assert (CONSP(input));

        return cons(copy_tree(input->car), copy_tree(input->cdr));
```

```
};

void eq_and_copy_test()
{
        /*
        18.1 (r) Evaluate
                (eql '((a b) ((c)) (d)) '((a b) ((c)) (d)))
        Notice that, although the two lists look the same, they are not eql.
        ( STUART C. SHAPIRO -- COMMON LISP -- An Interactive Approach )
        */

        struct cell* tmp=create_list3(
                //(a b)
                create_list_of_2_strs("a", "b"),
                // ((c))
                create_list1(create_list_of_1_str("c")),
                // (d)
                create_list_of_1_str("d"));

        // second list, construct it from scratch:
        struct cell* tmp2=create_list3(
                //(a b)
                create_list_of_2_strs("a", "b"),
                // ((c))
                create_list1(create_list_of_1_str("c")),
                // (d)
                create_list_of_1_str("d"));

        // tmp and tmp2 were created from scratch twice
        // they have different addresses
        assert (eql(tmp, tmp2)==false);
        // but these lists have the same contents:
        assert (equal(tmp, tmp2)==true);

        tmp=copy_tree(tmp2);
        assert (equal(tmp, tmp2)==true);
        assert (eq(tmp, tmp2)==false);
        assert (eql(tmp, tmp2)==false);
};
```

Again, Python has the same story about shallow/deep copy: https://yurichev.com/news/20211223_Py_ptrs/.

## 1.9 NCONC, append

Listing 1.9: append.h

```
// NCONC: concatenate two lists
// first list is modified. tail will point to the second list
/*
CL-USER> (setq x '(a b c))
CL-USER> (setq y '(d e f))
CL-USER> x
(A B C)
CL-USER> y
(D E F)
CL-USER> (nconc x y)
(A B C D E F)
CL-USER> x
(A B C D E F)
CL-USER> y
(D E F)
```

```
*/
// http://www.lispworks.com/documentation/lw60/CLHS/Body/f_nconc.htm
// AKA append! in SICP
// https://docs.racket-lang.org/srfi/srfi-std/srfi-1.html#append%21
struct cell* NCONC (struct cell* l1, struct cell* l2)
{
        assert (CONSP(l2) && "l2 argument must be list too!");

        if (l1==NULL)
                return l2;

        // find last cons of l1.
        struct cell* last=LAST(l1);
        assert(CONSP(last) && "NCONC: last element of l1 list must be cons cell");
        assert(cdr(last)==NULL); // yet free
        RPLACD (last, l2);
        return l1;
};

// http://clhs.lisp.se/Body/f_append.htm
struct cell* append(struct cell* x, struct cell* y)
{
        trace_2_inputs(__FUNCTION__, "x", x, "y", y);
        if (x==NULL)
        {
                trace_return(__FUNCTION__, y);
                return y;
        };
        if (y==NULL)
        {
                trace_return(__FUNCTION__, x);
                return x;
        };

        // from SICP, tail recursion:
        struct cell* rt=cons(
                car(x),
                append(cdr(x), y));
        /*
        // my naive version:
        struct cell* tmp=copy_tree(x);
        return NCONC(tmp, y);
        */

        trace_return(__FUNCTION__, rt);
        return rt;
};

void append_test()
{
        struct cell* tmp=create_list_of_2_strs("a", "b");
        struct cell* tmp2=create_list_of_2_strs("c", "d");
        struct cell* tmp3=append(tmp, tmp2);
        // must be untouched:
        // ("a" "b")
        assert_string_repr_is(tmp, false, "(\"a\" \"b\")");
        // must be untouched:
        // ("c" "d")
        assert_string_repr_is(tmp2, false, "(\"c\" \"d\")");
        // ("a" "b" "c" "d")
        assert_string_repr_is(tmp3, false, "(\"a\" \"b\" \"c\" \"d\")");
```

```
};
```

## 1.10 reverse

Listing 1.10: reverse.h

```
// http://clhs.lisp.se/Body/f_revers.htm
struct cell* reverse(struct cell* input)
{
        trace_1_input(__FUNCTION__, "input", input);
        if (input==NULL || length_is_1(input))
        {
                trace_return(__FUNCTION__, input);
                return input;
        };

        trace_padding++;
        // recursively...
        struct cell* rt=append(
                reverse(cdr(input)),
                create_list1(car(input)));
        trace_padding--;

        trace_return(__FUNCTION__, rt);
        return rt;
};

/*
from SICP:
=
Exercise 2.39

Complete the following definitions of reverse in terms of fold-right and fold-left:

(define (reverse sequence)
    (fold-right (lambda (x y) <??>) '() sequence))

(define (reverse sequence)
    (fold-left (lambda (x y) <??>) '() sequence))
=

; Racket
; reworked solutions from http://community.schemewiki.org/?sicp-ex-2.39
(define a '(1 2 3 4 5 6))
;(define a '(1))
;(define a '())
(reverse a)
(foldl cons null a)
(foldr (lambda (x y) (append y (list x))) null a)
(foldr (lambda (x y) (append y (cons x null))) null a)
(foldr (lambda (x y) (foldr cons (list x) y)) null a)
(foldr (lambda (x y) (foldr cons (cons x null) y)) null a)
*/

struct cell* reverse_v2(struct cell* input)
{
        trace_1_input(__FUNCTION__, "input", input);

        trace_padding++;
        struct cell* rt=fold_left (cons, NULL, input);
```

```
        trace_padding--;

        trace_return(__FUNCTION__, rt);
        return rt;
};

#ifdef CBMC
// cbmc --trace --function reverse_check 1.c --unwind 20 --unwinding-assertions -
   DCBMC
void reverse_check()
{
        int x,y,z,q,w;
        struct cell* tmp=create_list_of_5_ints(x,y,z,q,w);
        __CPROVER_assert (equal(reverse(tmp), reverse_v2(tmp)), "assert");
};
#endif

void reverse_test()
{
        trace_state=true;
        struct cell* tmp=create_list_of_4_strs("a", "b", "c", "d");
        struct cell* tmp2=reverse(tmp);
        // must be unchanged:
        assert_string_repr_is(tmp, false, "(\"a\" \"b\" \"c\" \"d\")");
        assert_string_repr_is(tmp2, false, "(\"d\" \"c\" \"b\" \"a\")");

        tmp2=reverse_v2(tmp);
        // must be unchanged:
        assert_string_repr_is(tmp, false, "(\"a\" \"b\" \"c\" \"d\")");
        assert_string_repr_is(tmp2, false, "(\"d\" \"c\" \"b\" \"a\")");
        trace_state=false;
};
```

Listing 1.11: Trace: tests

```
reverse() input=("a" "b" "c" "d")
 reverse() input=("b" "c" "d")
  reverse() input=("c" "d")
   reverse() input=("d")
   reverse() -> ("d")
   append() x=("d") y=("c")
   append() x=NIL y=("c")
   append() -> ("c")
   append() -> ("d" "c")
  reverse() -> ("d" "c")
  append() x=("d" "c") y=("b")
  append() x=("c") y=("b")
  append() x=NIL y=("b")
  append() -> ("b")
  append() -> ("c" "b")
  append() -> ("d" "c" "b")
 reverse() -> ("d" "c" "b")
 append() x=("d" "c" "b") y=("a")
 append() x=("c" "b") y=("a")
 append() x=("b") y=("a")
 append() x=NIL y=("a")
 append() -> ("a")
 append() -> ("b" "a")
 append() -> ("c" "b" "a")
 append() -> ("d" "c" "b" "a")
reverse() -> ("d" "c" "b" "a")
```

```
reverse_v2() input=("a" "b" "c" "d")
 fold_left() init=NIL list=("a" "b" "c" "d")
  fold_left() init=("a") list=("b" "c" "d")
   fold_left() init=("b" "a") list=("c" "d")
    fold_left() init=("c" "b" "a") list=("d")
     fold_left() init=("d" "c" "b" "a") list=NIL
     fold_left() -> ("d" "c" "b" "a")
    fold_left() -> ("d" "c" "b" "a")
   fold_left() -> ("d" "c" "b" "a")
  fold_left() -> ("d" "c" "b" "a")
 fold_left() -> ("d" "c" "b" "a")
reverse_v2() -> ("d" "c" "b" "a")
```

## 1.11 nth, nthcdr, first, second, ..., rest

Listing 1.12: nth_etc.h

```
// http://clhs.lisp.se/Body/f_nth.htm
// AKA list-ref in Scheme
struct cell* nth (struct cell* lst, int n) // starting at 0
{
        assert (LISTP(lst));
        if (n>=length_i(lst))
                return NULL;

        struct cell* tmp=lst;

        // skip n elements
        for (int i=0; i<n; i++)
                tmp=cdr(tmp);

        return car(tmp);
};

// "Returns the tail of list that would be obtained by calling cdr n times in
    succession."
// http://clhs.lisp.se/Body/f_nthcdr.htm
// AKA list-tail in Scheme: https://docs.racket-lang.org/reference/pairs.html#%28def.
    _%28%28quote._~23~25kernel%29._list-tail%29%29
// AKA drop in Scheme: https://docs.racket-lang.org/reference/pairs.html#%28def._
    %28%28lib._racket%2Flist..rkt%29._drop%29%29
struct cell* nthcdr (struct cell* lst, int n) // starting at 0
{
        assert (LISTP(lst));
        if (n>=length_i(lst))
                return NULL;

        struct cell* tmp=lst;

        // skip n elements
        for (int i=0; i<n; i++)
                tmp=cdr(tmp);

        return tmp;
};

void nthcdr_test()
{
        // simulate drop function
```

```
        // drop 2 first elements in this list:
        struct cell* tmp=create_list_of_5_strs("Mary", "had", "a", "little", "lamb");
        //print_cell (nthcdr(tmp, 2), false);
        // ("a" "little" "lamb")
        assert_string_repr_is(nthcdr(tmp, 2), false, "(\"a\" \"little\" \"lamb\")");
};

// http://clhs.lisp.se/Body/f_firstc.htm#first
struct cell* first (struct cell* c)
{
        return car(c);
        // or nth(c, 0)
};

struct cell* second (struct cell* c)
{
        // often abbreviated as cadr()
        return car(cdr(c));
        // or nth(c, 1)
};

struct cell* third (struct cell* c)
{
        // often abbreviated as caddr()
        return car(cdr(cdr(c)));
        // or nth(c, 2)
};

// rest AKA cdr:
struct cell* rest (struct cell* c)
{
        return cdr(c);
};

void nth_test()
{
        struct cell* tmp=create_list_of_3_ints(1, 2, 3);

        assert (eql(
                nth(tmp, 0),
                atom_int(1)));

        assert (eql(
                first(tmp),
                atom_int(1)));

        assert (eql(
                nth(tmp, 1),
                atom_int(2)));

        assert (eql(
                second(tmp),
                atom_int(2)));

        assert (eql(
                nth(tmp, 2),
                atom_int(3)));

        assert (eql(
                third(tmp),
                atom_int(3)));
```

```
};
```

## 1.12 Some helper funclets

Listing 1.13: funclets.h

```
struct cell* my_mul (struct cell* x, struct cell* y)
{
        numberp (x);
        numberp (y);
        return atom_int(x->number * y->number);
};

struct cell* my_add (struct cell* x, struct cell* y)
{
        numberp (x);
        numberp (y);
        return atom_int(x->number + y->number);
};

struct cell* my_add_strings (struct cell* x, struct cell* y)
{
        stringp (x);
        stringp (y);
        char tmp[1024]; // TODO - sum lengths of input strings...
        snprintf(tmp, 1024, "(%s+%s)", x->string, y->string);
        return atom_string(tmp);
};

struct cell* my_sub (struct cell* x, struct cell* y)
{
        numberp (x);
        numberp (y);
        return atom_int(x->number - y->number);
};
```

## 1.13 fold-left/right

Listing 1.14: fold.h

```
/*
idea from larceny-1.3

  (define (fold-left proc initial l)
    (cond ((null? l)
           initial)
          ((pair? l)
           (fold-left proc (proc initial (car l)) (cdr l)))
          (else
           (assertion-violation 'fold-left "non-list" l0))))
*/

struct cell* fold_left (struct cell* (*func) (struct cell*, struct cell*),
                struct cell* init,
                struct cell* list)
{
        trace_2_inputs(__FUNCTION__, "init", init, "list", list);
        if (list==NULL)
```

```
        {
                trace_return(__FUNCTION__, init);
                return init;
        };

        trace_padding++;
        struct cell* rt=fold_left(func, func(car(list), init), cdr(list));
        trace_padding--;

        trace_return(__FUNCTION__, rt);
        return rt;
};

/*
idea from larceny-1.3

  (define (fold-right proc initial l)
    (if (null? l)
        initial
        (proc (car l) (fold-right proc initial (cdr l)))))
*/

struct cell* fold_right (struct cell* (*func) (struct cell*, struct cell*),
                struct cell* init,
                struct cell* list)
{
        trace_2_inputs(__FUNCTION__, "init", init, "list", list);
        if (list==NULL)
        {
                trace_return(__FUNCTION__, init);
                return init;
        };

        trace_padding++;
        struct cell* rt=func(car(list), fold_right(func, init, cdr(list)));
        trace_padding--;

        trace_return(__FUNCTION__, rt);
        return rt;
};

void fold_tests()
{
        trace_state=true;
        // Racket
        // (foldl * 1 '(1 2 3 4 5))
        // 120
        assert (eql(
                fold_left (&my_mul, atom_int(1), range(1, 6)),
                atom_int(120)));

        // Racket:
        // (foldl + 0 '(1 2 3 4 5))
        // 15
        assert (eql(
                fold_left (&my_add, atom_int(0), range(1, 6)),
                atom_int(15)));

        // Racket:
        // (foldl - 6 '(1 2 3 4 5))
        // -3
```

```
        assert (eql(
                fold_left (&my_sub, atom_int(6), range(1, 6)),
                atom_int(-3)));

        // Racket:
        // (foldr - 6 '(1 2 3 4 5))
        // -3
        assert (eql(
                fold_right (&my_sub, atom_int(6), range(1, 6)),
                atom_int(-3)));

        // got idea from https://en.wikipedia.org/wiki/Fold_(higher-order_function)#
           Examples
        /*
        Racket:
        (define (f x y)
          (string-append "(" x "+" y ")"))
        (foldl f "init" '("1" "2" "3" "4" "5"))
        "(5+(4+(3+(2+(1+init)))))"
        (foldr f "init" '("1" "2" "3" "4" "5"))
        "(1+(2+(3+(4+(5+init)))))"
         */
        struct cell* numbers=mapcar(number_to_string, range(1, 6));
        struct cell* tmp;
        tmp=fold_left (&my_add_strings, atom_string("init"), numbers);
        assert_string_repr_is(tmp, false, "\"(5+(4+(3+(2+(1+init)))))\"");

        tmp=fold_right (&my_add_strings, atom_string("init"), numbers);
        assert_string_repr_is(tmp, false, "\"(1+(2+(3+(4+(5+init)))))\"");

        /*
        Racket:
        (foldl cons 6 '(1 2 3 4 5))
        '(5 4 3 2 1 . 6)
        */
        tmp=fold_left (&cons, atom_int(6), range(1, 6));
        assert_string_repr_is(tmp, true, "(5 . (4 . (3 . (2 . (1 . 6)))))");
        assert_string_repr_is(tmp, false, "(5 . (4 . (3 . (2 . (1 . 6)))))");

        /*
        Racket:
        (foldr cons 6 '(1 2 3 4 5))
        '(1 2 3 4 5 . 6)
        */
        tmp=fold_right (&cons, atom_int(6), range(1, 6));
        assert_string_repr_is(tmp, true, "(1 . (2 . (3 . (4 . (5 . 6)))))");
        assert_string_repr_is(tmp, false, "(1 . (2 . (3 . (4 . (5 . 6)))))");
        trace_state=false;
};
```

Listing 1.15: Trace: tests

```
fold_left() init=1 list=(1 2 3 4 5)
 fold_left() init=1 list=(2 3 4 5)
  fold_left() init=2 list=(3 4 5)
   fold_left() init=6 list=(4 5)
    fold_left() init=24 list=(5)
     fold_left() init=120 list=NIL
     fold_left() -> 120
    fold_left() -> 120
   fold_left() -> 120
```

```
  fold_left() -> 120
 fold_left() -> 120
fold_left() -> 120
fold_left() init=0 list=(1 2 3 4 5)
 fold_left() init=1 list=(2 3 4 5)
  fold_left() init=3 list=(3 4 5)
   fold_left() init=6 list=(4 5)
    fold_left() init=10 list=(5)
     fold_left() init=15 list=NIL
     fold_left() -> 15
    fold_left() -> 15
   fold_left() -> 15
  fold_left() -> 15
 fold_left() -> 15
fold_left() -> 15
fold_left() init=6 list=(1 2 3 4 5)
 fold_left() init=-5 list=(2 3 4 5)
  fold_left() init=7 list=(3 4 5)
   fold_left() init=-4 list=(4 5)
    fold_left() init=8 list=(5)
     fold_left() init=-3 list=NIL
     fold_left() -> -3
    fold_left() -> -3
   fold_left() -> -3
  fold_left() -> -3
 fold_left() -> -3
fold_left() -> -3
fold_right() init=6 list=(1 2 3 4 5)
 fold_right() init=6 list=(2 3 4 5)
  fold_right() init=6 list=(3 4 5)
   fold_right() init=6 list=(4 5)
    fold_right() init=6 list=(5)
     fold_right() init=6 list=NIL
     fold_right() -> 6
    fold_right() -> -1
   fold_right() -> 5
  fold_right() -> -2
 fold_right() -> 4
fold_right() -> -3
fold_left() init="init" list=("1" "2" "3" "4" "5")
 fold_left() init="(1+init)" list=("2" "3" "4" "5")
  fold_left() init="(2+(1+init))" list=("3" "4" "5")
   fold_left() init="(3+(2+(1+init)))" list=("4" "5")
    fold_left() init="(4+(3+(2+(1+init))))" list=("5")
     fold_left() init="(5+(4+(3+(2+(1+init)))))" list=NIL
     fold_left() -> "(5+(4+(3+(2+(1+init)))))"
    fold_left() -> "(5+(4+(3+(2+(1+init)))))"
   fold_left() -> "(5+(4+(3+(2+(1+init)))))"
  fold_left() -> "(5+(4+(3+(2+(1+init)))))"
 fold_left() -> "(5+(4+(3+(2+(1+init)))))"
fold_left() -> "(5+(4+(3+(2+(1+init)))))"
fold_right() init="init" list=("1" "2" "3" "4" "5")
 fold_right() init="init" list=("2" "3" "4" "5")
  fold_right() init="init" list=("3" "4" "5")
   fold_right() init="init" list=("4" "5")
    fold_right() init="init" list=("5")
     fold_right() init="init" list=NIL
     fold_right() -> "init"
    fold_right() -> "(5+init)"
   fold_right() -> "(4+(5+init))"
  fold_right() -> "(3+(4+(5+init)))"
```

```
 fold_right() -> "(2+(3+(4+(5+init))))"
fold_right() -> "(1+(2+(3+(4+(5+init)))))"
fold_left() init=6 list=(1 2 3 4 5)
 fold_left() init=(1 . 6) list=(2 3 4 5)
  fold_left() init=(2 . (1 . 6)) list=(3 4 5)
   fold_left() init=(3 . (2 . (1 . 6))) list=(4 5)
    fold_left() init=(4 . (3 . (2 . (1 . 6)))) list=(5)
     fold_left() init=(5 . (4 . (3 . (2 . (1 . 6))))) list=NIL
     fold_left() -> (5 . (4 . (3 . (2 . (1 . 6)))))
    fold_left() -> (5 . (4 . (3 . (2 . (1 . 6)))))
   fold_left() -> (5 . (4 . (3 . (2 . (1 . 6)))))
  fold_left() -> (5 . (4 . (3 . (2 . (1 . 6)))))
 fold_left() -> (5 . (4 . (3 . (2 . (1 . 6)))))
fold_left() -> (5 . (4 . (3 . (2 . (1 . 6)))))
fold_right() init=6 list=(1 2 3 4 5)
 fold_right() init=6 list=(2 3 4 5)
  fold_right() init=6 list=(3 4 5)
   fold_right() init=6 list=(4 5)
    fold_right() init=6 list=(5)
     fold_right() init=6 list=NIL
     fold_right() -> 6
    fold_right() -> (5 . 6)
   fold_right() -> (4 . (5 . 6))
  fold_right() -> (3 . (4 . (5 . 6)))
 fold_right() -> (2 . (3 . (4 . (5 . 6))))
fold_right() -> (1 . (2 . (3 . (4 . (5 . 6)))))
```

## 1.14 mapcar

Listing 1.16: mapcar.h

```
// http://www.lispworks.com/documentation/lw50/CLHS/Body/f_mapc_.htm
// AKA map in Scheme and in many other PLs
struct cell* mapcar(struct cell* (*func) (struct cell*),
                struct cell* list)
{
        assert (LISTP(list));

        struct cell* rt=make_list(length_i(list));
        struct cell* out=rt;

        // traverse list:
        for (struct cell* i=list; i; i=cdr(i))
        {
                RPLACA(out, func(car(i)));
                out=out->cdr;
        };
        return rt;
};

void mapcar_test()
{
        struct cell* tmp=mapcar(number_to_string, range(0, 10));
        assert_string_repr_is(tmp, false, "(\"0\" \"1\" \"2\" \"3\" \"4\" \"5\" \"6\"
            \"7\" \"8\" \"9\")");
};
```

## 1.15 reduce, sum, product, factorial

Listing 1.17: reduce.h

```
// http://clhs.lisp.se/Body/f_reduce.htm
// https://stackoverflow.com/questions/21688283/reduce-function-in-racket
   /25211454#25211454
struct cell* reduce (struct cell* (*func) (struct cell*, struct cell*),
                struct cell* list)
{
        trace_1_input(__FUNCTION__, "list", list);

        if (length_i(list)<=1)
        {
                trace_return(__FUNCTION__, list);
                return list;
        };

        trace_padding++;
        struct cell* rt=fold_left(func, first(list), rest(list));
        trace_padding--;

        trace_return(__FUNCTION__, rt);
        return rt;
};

struct cell* sum (struct cell* list)
{
        trace_1_input(__FUNCTION__, "list", list);

        trace_padding++;
        struct cell* rt=reduce(my_add, list);
        trace_padding--;

        trace_return(__FUNCTION__, rt);
        return rt;
};

struct cell* product (struct cell* list)
{
        trace_1_input(__FUNCTION__, "list", list);

        trace_padding++;
        struct cell* rt=reduce(my_mul, list);
        trace_padding--;

        trace_return(__FUNCTION__, rt);
        return rt;
};

struct cell* factorial (int max)
{
        return product (range(1, max+1));
};

// 11! = 39916800
void factorial_test()
{
        trace_state=true;
        assert (eql(
                factorial(11),
                atom_int(39916800)));
        trace_state=false;
};
```

```
void reduce_test()
{
        trace_state=true;
        assert (eql(
                reduce(my_add, range(1, 6)),
                atom_int(15)));
        assert (eql(
                reduce(my_mul, range(1, 6)),
                atom_int(120)));
        trace_state=false;
};
```

Listing 1.18: Trace: tests for reduce

```
reduce() list=(1 2 3 4 5)
 fold_left() init=1 list=(2 3 4 5)
  fold_left() init=3 list=(3 4 5)
   fold_left() init=6 list=(4 5)
    fold_left() init=10 list=(5)
     fold_left() init=15 list=NIL
     fold_left() -> 15
    fold_left() -> 15
   fold_left() -> 15
  fold_left() -> 15
 fold_left() -> 15
reduce() -> 15
reduce() list=(1 2 3 4 5)
 fold_left() init=1 list=(2 3 4 5)
  fold_left() init=2 list=(3 4 5)
   fold_left() init=6 list=(4 5)
    fold_left() init=24 list=(5)
     fold_left() init=120 list=NIL
     fold_left() -> 120
    fold_left() -> 120
   fold_left() -> 120
  fold_left() -> 120
 fold_left() -> 120
reduce() -> 120
```

Listing 1.19: Trace: test for factorial

```
product() list=(1 2 3 4 5 6 7 8 9 10 11)
 reduce() list=(1 2 3 4 5 6 7 8 9 10 11)
  fold_left() init=1 list=(2 3 4 5 6 7 8 9 10 11)
   fold_left() init=2 list=(3 4 5 6 7 8 9 10 11)
    fold_left() init=6 list=(4 5 6 7 8 9 10 11)
     fold_left() init=24 list=(5 6 7 8 9 10 11)
      fold_left() init=120 list=(6 7 8 9 10 11)
       fold_left() init=720 list=(7 8 9 10 11)
        fold_left() init=5040 list=(8 9 10 11)
         fold_left() init=40320 list=(9 10 11)
          fold_left() init=362880 list=(10 11)
           fold_left() init=3628800 list=(11)
            fold_left() init=39916800 list=NIL
            fold_left() -> 39916800
           fold_left() -> 39916800
          fold_left() -> 39916800
         fold_left() -> 39916800
        fold_left() -> 39916800
       fold_left() -> 39916800
      fold_left() -> 39916800
```

```
     fold_left() -> 39916800
    fold_left() -> 39916800
   fold_left() -> 39916800
  fold_left() -> 39916800
 reduce() -> 39916800
product() -> 39916800
```

## 1.16 flatten

Listing 1.20: flatten.h

```
// https://stackoverflow.com/questions/2680864/how-to-remove-nested-parentheses-in-
   lisp
struct cell* flatten(struct cell* input)
{
        trace_1_input(__FUNCTION__, "input", input);
        if (input==NULL || ATOM(input))
        {
                trace_return(__FUNCTION__, input);
                return input;
        };

        struct cell* rt;

        trace_padding++;
        if (ATOM(car(input)))
                rt=cons (first(input), flatten (rest(input)));
        else
                rt=append (flatten(first(input)), flatten(rest(input)));
        trace_padding--;

        trace_return(__FUNCTION__, rt);
        return rt;
};

void flatten_test()
{
        trace_state=true;
        // (("a" "b") (("c")) ("d"))
        struct cell* tmp1=create_list3(
                //(a b)
                create_list_of_2_strs("a", "b"),
                // ((c))
                create_list1(create_list_of_1_str("c")),
                // (d)
                create_list_of_1_str("d"));
        dump_graphviz(tmp1, "flatten_test_before");
        struct cell* tmp2=flatten(tmp1);
        assert_string_repr_is(tmp2, false, "(\"a\" \"b\" \"c\" \"d\")");
        dump_graphviz(tmp2, "flatten_test_after");
        trace_state=false;
};
```

Listing 1.21: Trace: tests

```
flatten() input=(("a" "b") (("c")) ("d"))
 flatten() input=((("c")) ("d"))
  flatten() input=(("d"))
   flatten() input=NIL
   flatten() -> NIL
```

```
   flatten() input=("d")
    flatten() input=NIL
    flatten() -> NIL
   flatten() -> ("d")
   append() x=("d") y=NIL
   append() -> ("d")
  flatten() -> ("d")
  flatten() input=(("c"))
   flatten() input=NIL
   flatten() -> NIL
   flatten() input=("c")
    flatten() input=NIL
    flatten() -> NIL
   flatten() -> ("c")
   append() x=("c") y=NIL
   append() -> ("c")
  flatten() -> ("c")
  append() x=("c") y=("d")
  append() x=NIL y=("d")
  append() -> ("d")
  append() -> ("c" "d")
 flatten() -> ("c" "d")
 flatten() input=("a" "b")
  flatten() input=("b")
   flatten() input=NIL
   flatten() -> NIL
  flatten() -> ("b")
 flatten() -> ("a" "b")
 append() x=("a" "b") y=("c" "d")
 append() x=("b") y=("c" "d")
 append() x=NIL y=("c" "d")
 append() -> ("c" "d")
 append() -> ("b" "c" "d")
 append() -> ("a" "b" "c" "d")
flatten() -> ("a" "b" "c" "d")
```

Figure 1.3: flatten() test: before run (input)

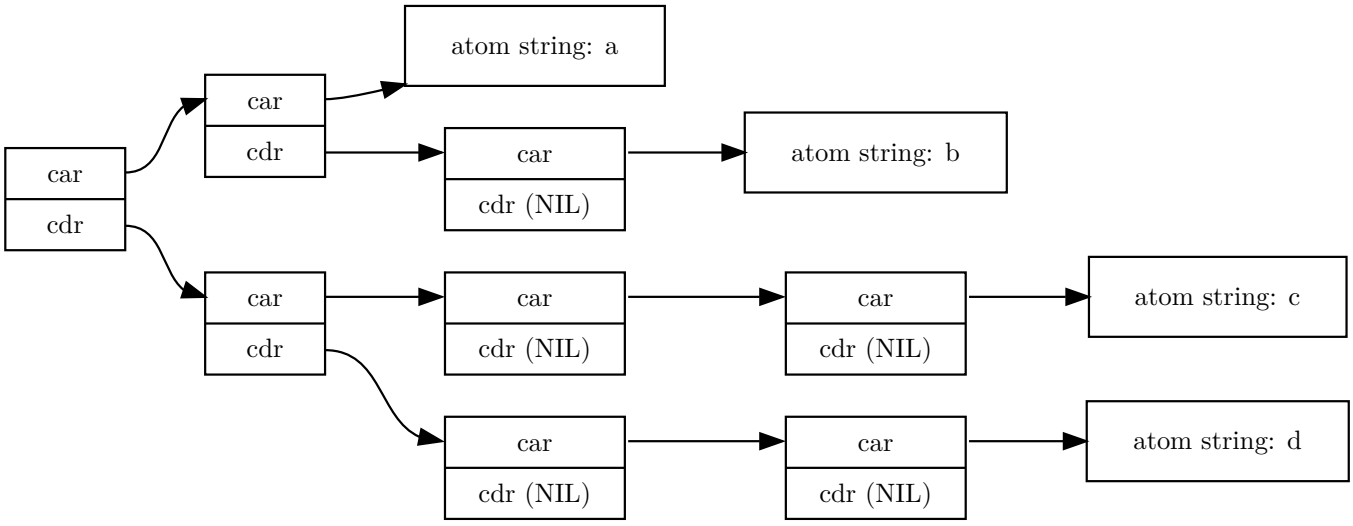
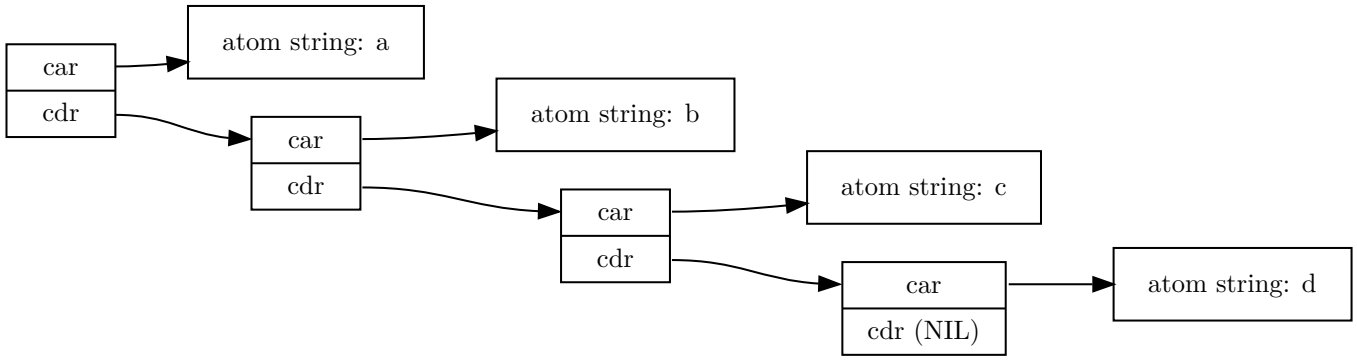


Figure 1.4: flatten() test: after run (output)




## 1.17 list_max

Listing 1.22: list_max.h

```
bool less_or_eq(struct cell* x, struct cell* y)
{
        // can be extended to bigger numerical tower: reals, ratios... even strings
            ...
        assert(ATOM(x));
        assert(ATOM(y));

        assert(x->type==CELL_TYPE_ATOM_INT);
        assert(y->type==CELL_TYPE_ATOM_INT);

        return x->number <= y->number;
};

struct cell* max_item(struct cell* x, struct cell* y)
{
        trace_2_inputs(__FUNCTION__, "x", x, "y", y);

        struct cell* rt=less_or_eq(x, y) ? y : x;

        trace_return(__FUNCTION__, rt);
        return rt;
};

struct cell* list_max(struct cell* list)
{
        assert (list);

        trace_1_input(__FUNCTION__, "list", list);

        // will also work:
        //if (length_i(list)==1)
        //        return car(list);
        if (length_is_1(list))
        {
                trace_return(__FUNCTION__, car(list));
                return car(list);
        };

        trace_padding++;
        // tail-recursive:
        struct cell* rt=max_item(car(list), list_max(cdr(list)));
        trace_padding--;
```

```
        trace_return(__FUNCTION__, rt);
        return rt;
};

// fancy version:
struct cell* list_max_v2(struct cell* list)
{
        assert (list);

        trace_1_input(__FUNCTION__, "list", list);

        trace_padding++;
        struct cell* rt=reduce(max_item, list);
        trace_padding--;

        trace_return(__FUNCTION__, rt);
        return rt;
};

#ifdef CBMC
// cbmc --trace --function list_max_check 1.c --unwind 20 --unwinding-assertions -
    DCBMC
void list_max_check()
{
        int x,y,z,q,w;
        struct cell* tmp=create_list_of_5_ints(x,y,z,q,w);
        __CPROVER_assert (eql(list_max(tmp), list_max_v2(tmp)), "assert");
};
#endif

void list_max_test()
{
        trace_state=true;
        struct cell* lst=create_list_of_5_ints(432, 0, -1, 999, 4);
        struct cell* tmp;

        tmp=list_max(lst);
        assert (eql(tmp, atom_int(999)));

        tmp=list_max_v2(lst);
        assert (eql(tmp, atom_int(999)));
        trace_state=false;
};
```

Listing 1.23: Trace: tests

```
list_max() list=(432 0 -1 999 4)
 list_max() list=(0 -1 999 4)
  list_max() list=(-1 999 4)
   list_max() list=(999 4)
    list_max() list=(4)
    list_max() -> 4
    max_item() x=999 y=4
    max_item() -> 999
   list_max() -> 999
   max_item() x=-1 y=999
   max_item() -> 999
  list_max() -> 999
  max_item() x=0 y=999
  max_item() -> 999
```

```
 list_max() -> 999
 max_item() x=432 y=999
 max_item() -> 999
list_max() -> 999

list_max_v2() list=(432 0 -1 999 4)
 reduce() list=(432 0 -1 999 4)
  fold_left() init=432 list=(0 -1 999 4)
   max_item() x=0 y=432
   max_item() -> 432
   fold_left() init=432 list=(-1 999 4)
    max_item() x=-1 y=432
    max_item() -> 432
    fold_left() init=432 list=(999 4)
     max_item() x=999 y=432
     max_item() -> 999
     fold_left() init=999 list=(4)
      max_item() x=4 y=999
      max_item() -> 999
      fold_left() init=999 list=NIL
      fold_left() -> 999
     fold_left() -> 999
    fold_left() -> 999
   fold_left() -> 999
  fold_left() -> 999
 reduce() -> 999
list_max_v2() -> 999
```

## 1.18 Further reading

I enjoyed reading source code of some embedded Lisp in xedit editor in the guts of NetBSD: here (click 'download').

Some toy Lisps:
https://github.com/robpike/lisp
http://norvig.com/lispy.html
https://github.com/shinh/sedlisp
https://www.cliki.net/AWK%20Lisp
https://www.metalevel.at/lisprolog/

## 1.19 Want to hack my code?

My pure C source code doesn't have deallocating functions. It's easy to implement it, recursively. But you'll get into trouble if a sublist references in two lists. Here you'll need a garbage collector.

Next step is a (toy) interpreter – add parser [4], eval() [5], etc...

Download the code: https://yurichev.org/lisp/lisp.tar.

## 1.20 Other notes

Donald Knuth's TAOCP has something about *cons cells* and garbage cells, but without mention of Lisp: "2.3.5. Lists and Garbage Collection".

My hypothesis is that such lists were in use before Lisp evolvment. Maybe John McCarthy build Lisp on top of existing data structure?

## 1.21 Symbols

Symbols are like *interned* strings in Python. Of course, when you compare strings in Python, you only compare their addresses. This is fast.

[4] Like I did in my toy MK85 SMT solver: https://sat-smt.codes/
[5] Like I did in toy decompiler: https://sat-smt.codes/

In Lisp, all symbols are unique. So their addresses are also unique. So they can be compared with *EQ*, which compares only addresses.

This is fun, but within a single pure C module, similar strings are also have the same address in memory. This is because if you have two constant strings "hello" and "hello", compiler would need only to store it once. [For more examples, see my "Reverse Engineering for Beginners" [6].]

For example, this code:

```
#include <stdio.h>

void main()
{
        printf ("%p\n", "hello");
        printf ("%p\n", "world");
        printf ("%p\n", "hello");
        printf ("%d\n", "hello" == "hello");
};
```

... may print:

```
0x55eb45529004
0x55eb4552900e
0x55eb45529004
1
```

... but another module within the same executable may have another "hello" string with another address.

[6] https://beginners.re/