F.G.H.
5/23/85

LispCourse #30: Conceptual Models for Atoms, Lists *et al.*

Introduction

So far, we've been talking about data objects in Lisp in terms of how they appear to us as programmer's, i.e., in terms of how we type them in and how they are printed on the screen.

> Example: We defined a list as a thing that begins with a "(", followed by one or more atoms or lists, and terminated by a ")".

This way of talking is not wrong, but it does not reflect how Lisp itself "thinks" about the various data objects.

> In particular, Lisp translates the data objects that we type-in into an internal representation. All functions dealing we these data objects then work on this internal representation of the objects. Only when it comes time to print out some result, does Lisp translate the internal; representation back into the external representation (i.e., into the *print name* described in LispCourse #28, page 9).

You can't get very far in Lisp programming without understanding something about the underlying internal representations for atoms, lists, etc.

> For example, the semantics of many Lisp functions can be expressed only in terms of this underlying representation. There are many examples of this below.

There are many levels at which one could describe the internal representation of Lisp data objects.

> For example, the harware actually processes bits that represent numbers and addresses in its memory.
>
> However, here we will discuss internal representaion at a *conceptual* level.

The goal is to provide a conceptual model of Lisp data that provides all of the necessary concepts and mechanism for understanding the semantics of Lisp functions, BUT without going into the grubby details of how the data is actually implemented at the hardware/microcode level.

Everything in Lisp is a *Pointer*

The first lesson is that everything in Lisp is a *pointer*.

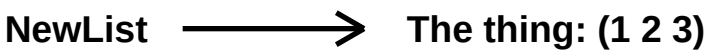
A *pointer* is simply a one-way connection between two data objects. If A points to B, then we can get to B from A, but not vice versa.

When we say that *atom A has the value 5*, Lisp represents this by a pointer between the thing that is the atom **A** and the thing that is the atom **5**.

The atom A points to the atom 5

Similarly, when we represent the fact that *the value of the atom NewList is the list (1 2 3)*, Lisp represents that by a pointer between the atom **NewList** and the thing that represents the list **(1 2 3)**.

The atom NewList points to the list (1 2 3)



Also, when we say *function Foo returns a list*, we actually mean that function Foo returns a pointer to a list.

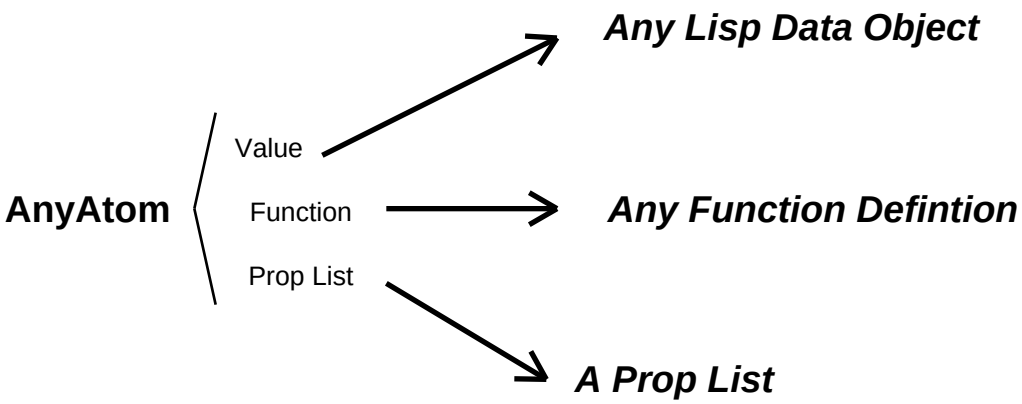
> In some sense, you seldom actually hold a Lisp data object in you hands when you program in Lisp. Its more like you are holding one end of a rope in your hand. The other end of the rope is a Lisp data object. You operate in Lisp by passing around the free end of this rope, e.g., functions return to you a free end of the rope which you then pass onto other functions, etc.

Representing Atoms

In our conceptual model, atoms will be represented by their print name. So to represent the aton *ManyWordAtom*, we simply type use **ManyWordAtom**.

An atom can have three pointers coming from it, representing its *value*, its *prop list* and its *function definition*.

Example:



The SET functions (SET, SETQ, SETQQ) can be used to establish the pointer between an atom and its value.

For example, (SETQ A ’B) sets up a (value) pointer between the atom A and the atom B.

The function DEFINEQ establishes the pointer between an atom and a function definition.

For example, (DEFINEQ (MyFunc (LAMBDA NIL (PLUS 1 3)))) sets up a (function) pointer between the atom MyFunc and the given function definition (LAMBDA ...).

The function SETPROPLIST establishes the pointer between an atom and a prop list object.

For example, (SETPROPLIST 'MyAtom '(Size 5)) sets up a (prop list) pointer between the atom MyAtom and the given prop list.

Note: The functions PUTPROP and GETPROP add to and retrieve from from the prop list pointed to the given atom.

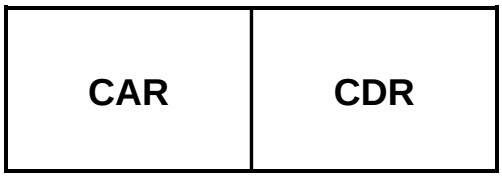
Representing Lists

The CONS cell

Lists are constructed from basic building blocks called ***CONS cells***.

A *CONS cell* can be represented by a box divided into two halves. The left half is called the *CAR* and the right half is called the *CDR*.

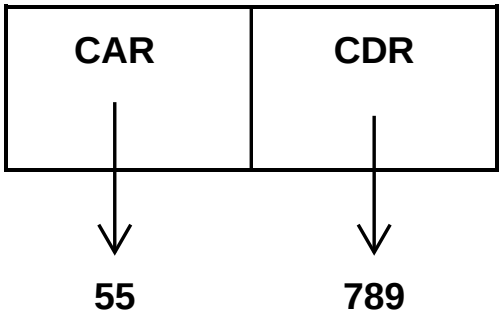
A CONS cell



Both the CAR and the CDR portions of a CONS cell contain pointers to other Lisp data objects.

A CONS cells is printed as a ***dotted pair***, ***(X . Y)***, where *X* is the print name of the thing pointed to by the CAR of the CONS cell and *Y* is the print name of the thing pointed to by the CDR.

The CONS cell: (55 . 789)

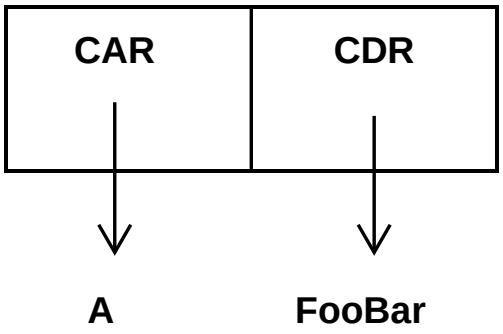


Creating a CONS cell is done using the function **CONS**.

Example: **(CONS 55 789)** prints the result **(55 . 789)** and actually creates a CONS cell like the one shown above.

Second example: **(CONS 'A 'FooBar)** prints the result **(A . FooBar)** and actually creates a CONS cell like the one shown below.

The result of (CONS 'A 'FooBar)



Lists are built from CONS cells

Lists are constructed from CONS cells in the following manner:

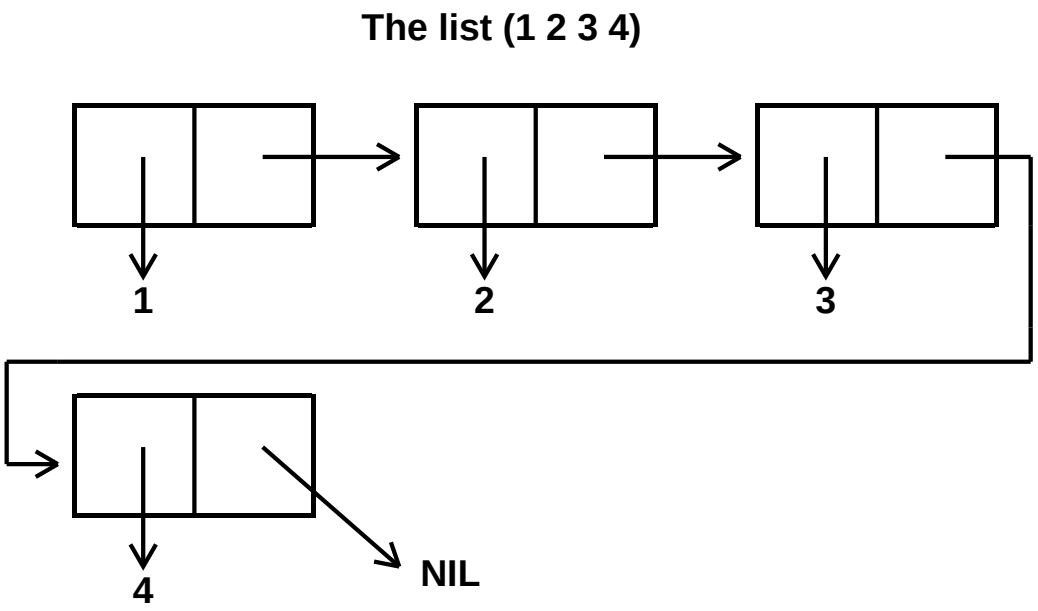
There is a CONS cell for each item in the list.

The CAR of the CONS cell points to the item.

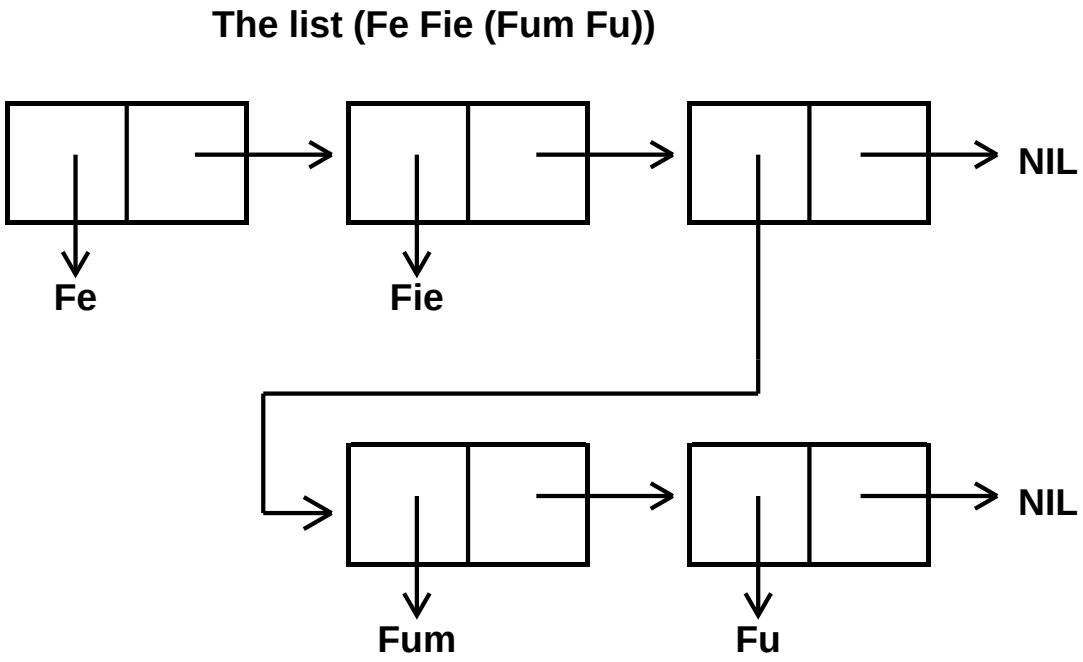
The CDR of the CONS cell points to the CONS cell for the next item in the list.

The CDR of the CONS cell for the last item in the list (i.e., where there is no next item) points to the atom NIL.

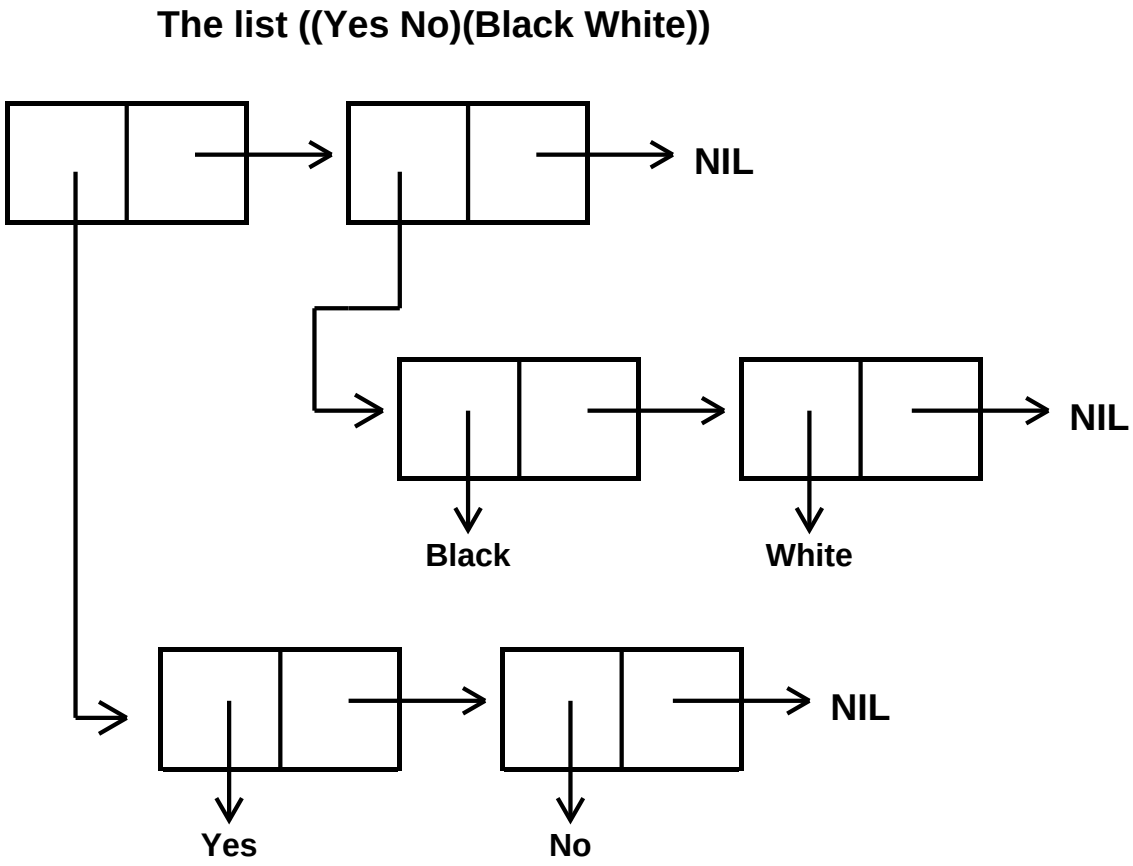
Example: Representation of the list *(1 2 3 4)*



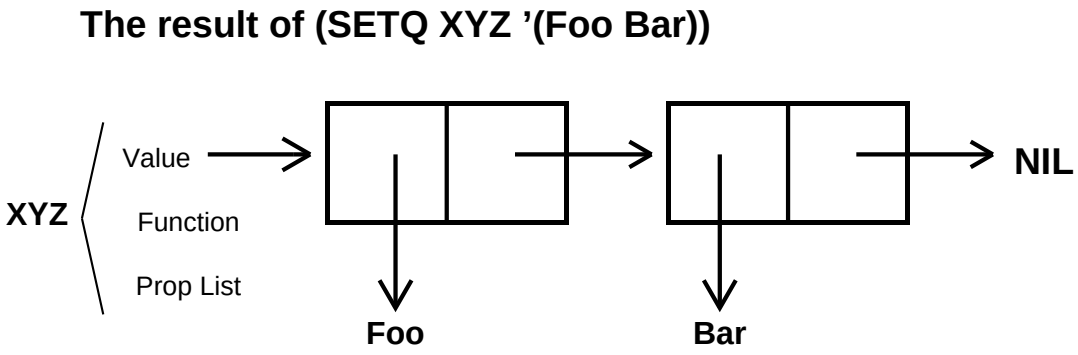
Second example: Representation of the list *(Fe Fie (Fum Fu))*



Third example: Representation of the list *((Yes No)(Black White))*



Fourth example: The result of (SETQ XYZ ’(Foo Bar))



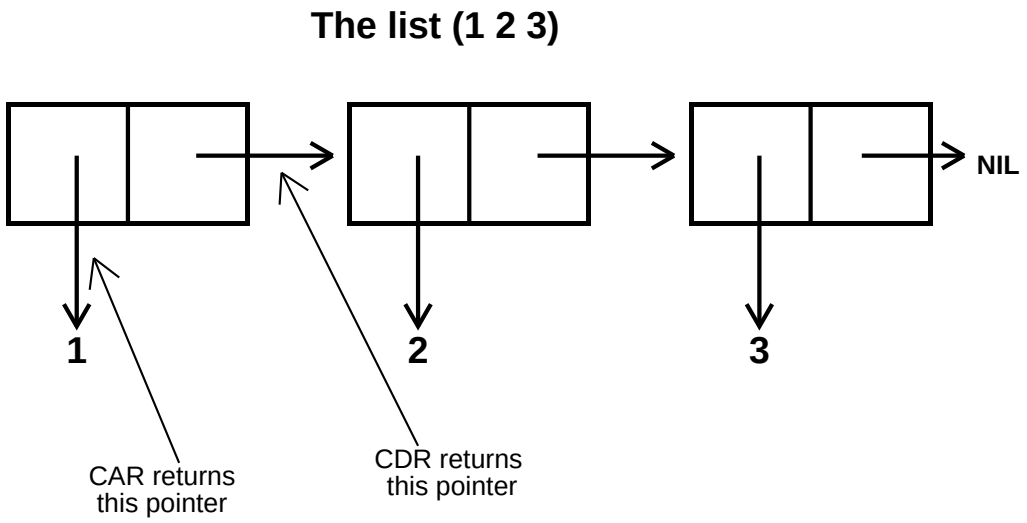
CAR and CDR revisited

CAR and CDR can be defined in terms of CONS cells as follows:

CAR ž returns contents of the CAR part of the CONS cell pointed to by its argument.

CDR ž returns contents of the CDR part of the CONS cell pointed to by its argument.

Note: when the CONS cell is part of list, then the CDR part points to the CONS cell that begins the rest of the list as per our previous definition of CDR.



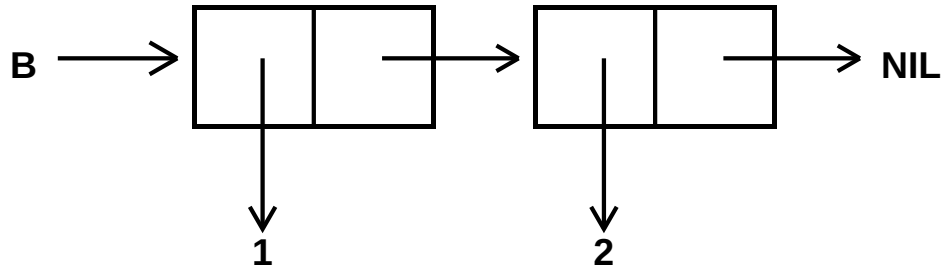
CONSing onto a list and LIST revisited

Recall that CONS creates a new CONS cell with the first argument as its CAR and the second argument as its CDR.

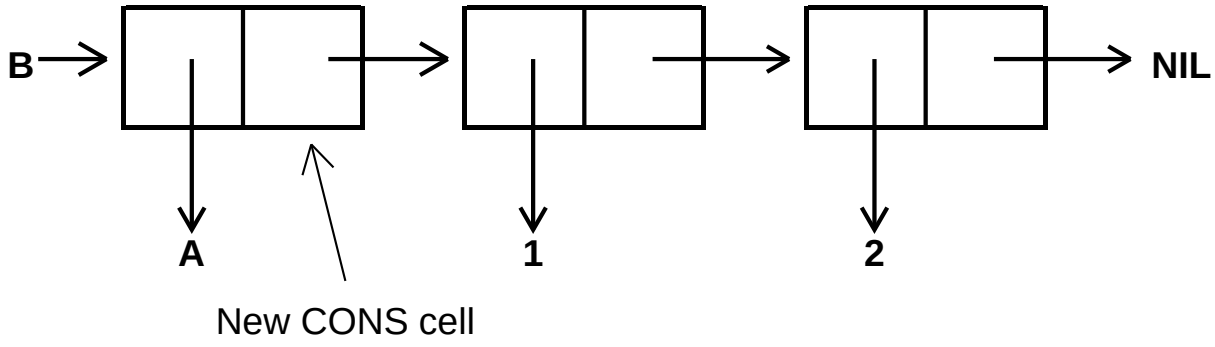
When the second argument is a list (i.e., a pointer to the first CONS cell in a list) then the result is a new list with a different CONS cell at the head of the list.

Example: *(SETQQ B (1 2)) then (SETQ B (CONS 'A B))*

After (SETQQ B (1 2))

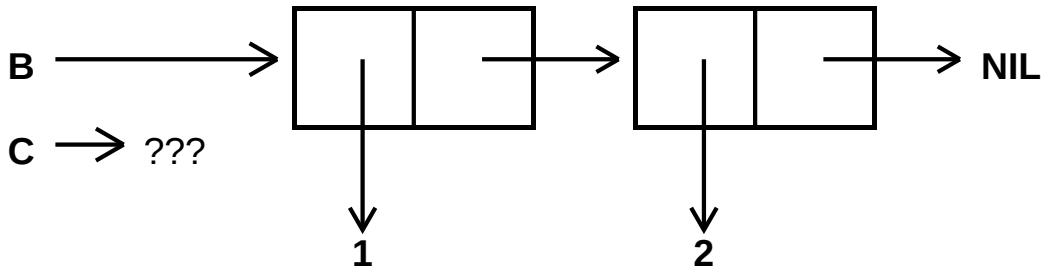


After (SETQ B (CONS 'A B))

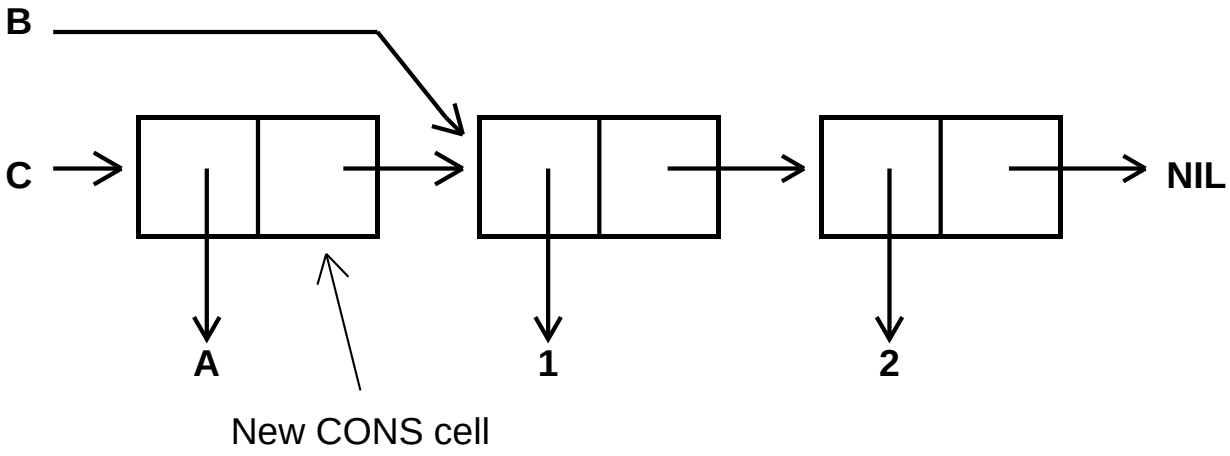


Second example: *(SETQQ B (1 2)) then (SETQ C (CONS 'A B))*

After (SETQQ B (1 2))



After (SETQ C (CONS 'A B))



The function ***LIST*** constructs a list with its arguments as the items in the list. Example: (LIST 1 2 3 '(3 4)) returns the list (1 2 3 (3 4)). LIST works by successive CONSes, i.e., by building the list one CONS cell at a time.

EFS: Define the function NewLIST using the CONS function.

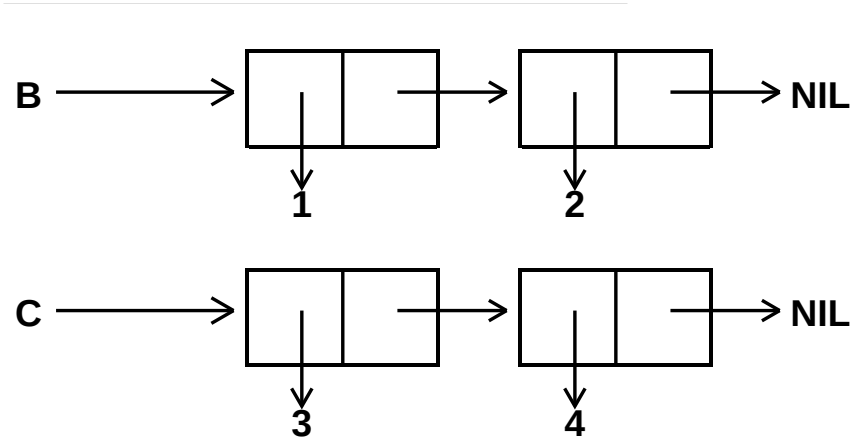
APPEND revisited

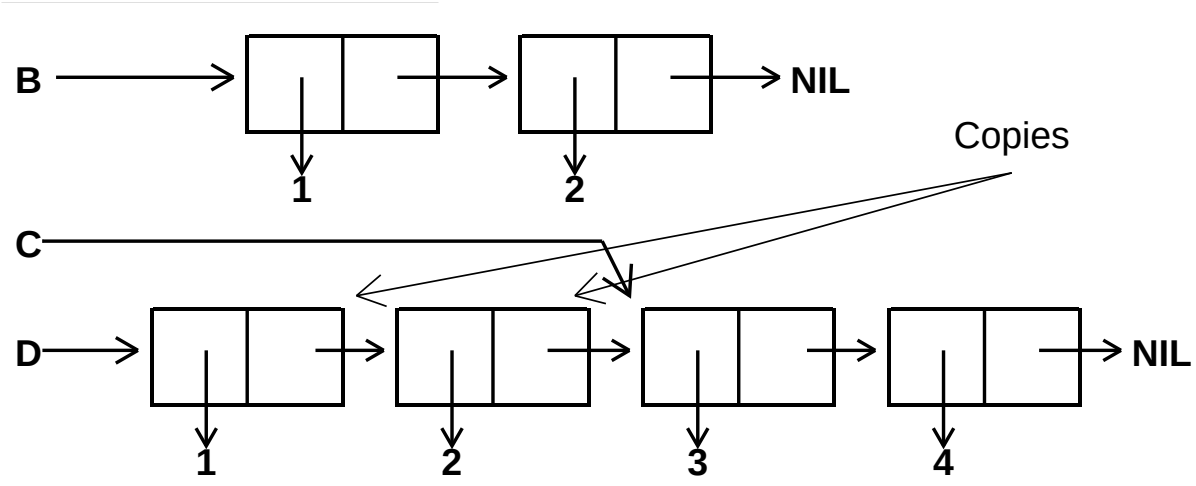
APPEND works very differently from CONS.

In particular, APPEND ***copies*** all but the last list it is appending before it does the append.

It then changes the last CDR cell in each copy to point to the head of the next copy rather than to NIL.

Example: *(SETQQ B (1 2)), (SETQQ C (3 4)) then (SETQ D (APPEND B C))*





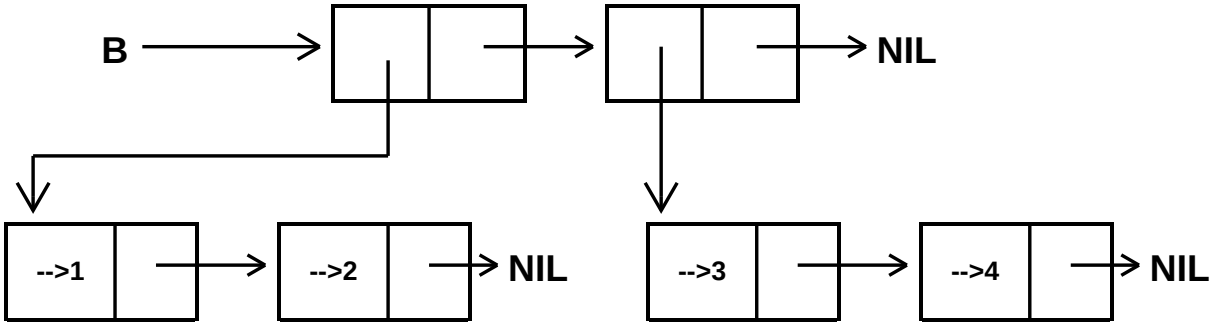
The COPY function

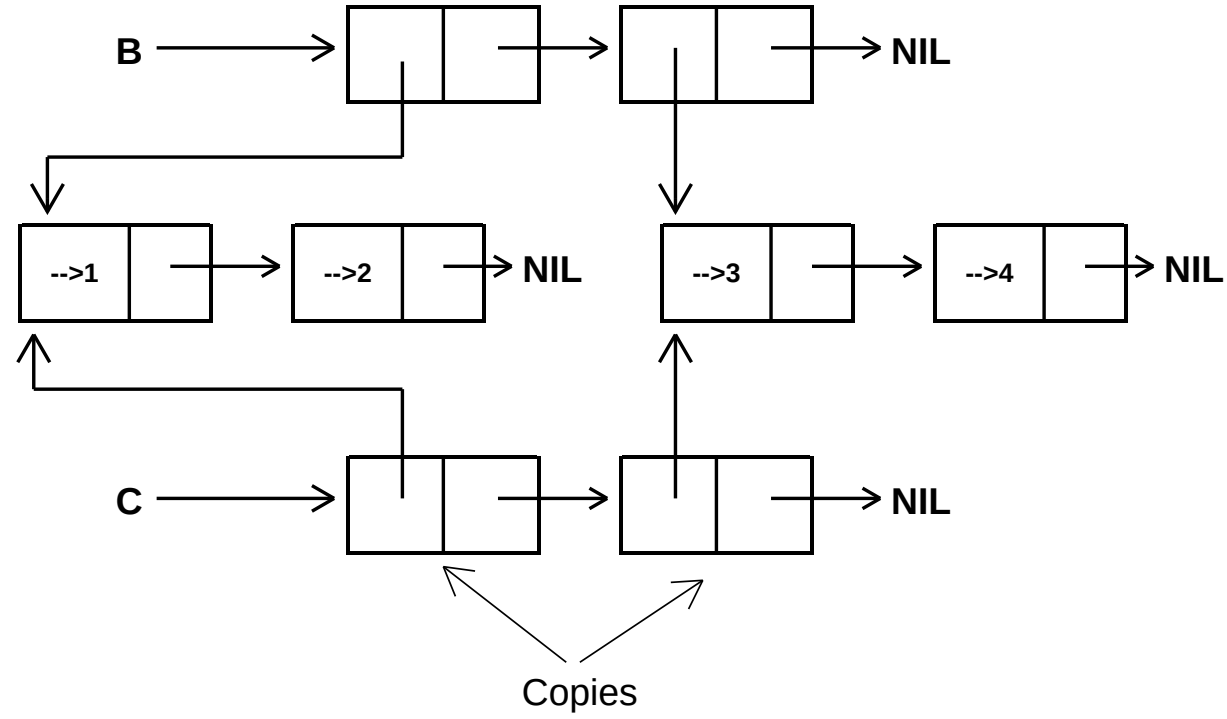
The function **COPY** makes a copy of a list by copying all the CONS cells in the list.

APPEND uses COPY to make its copies.

Note: COPY copies only the top-level of a list. If the CAR of any CONS cell points to another list, then that list is NOT copied.

Example: *(SETQQ B ((1 2) (3 4)) then (SETQ C (COPY B))*





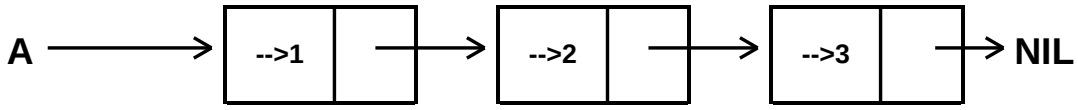
RPLACA, RPLACD, and NCONC -- The destructive functions

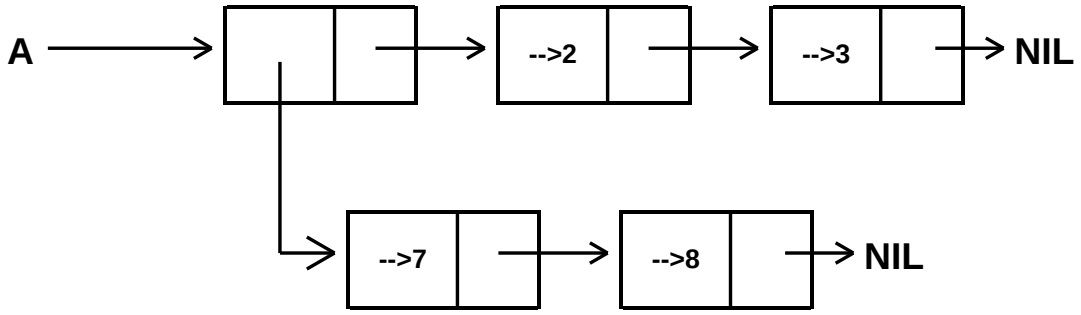
RPLACA, RPLACD, and NCONC are three function that "do surgery on lists." Unlike APPEND, they don't make copies and then carry out actions on the copies, rather they actually change the list they are passed as an argument. Because of this they are dangerous functions and should be used with some care!!!!

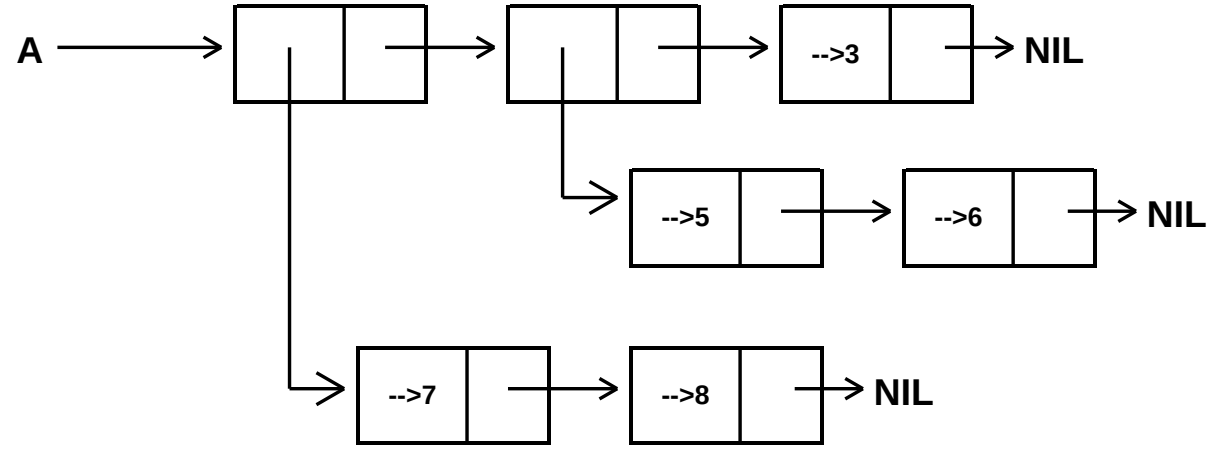
(RPLACA *X Y*)ž replaces the CAR of the CONS cell *X* with a pointer *Y*. RPLACA returns *X*.

Example:

1_(SETQQ A (1 2 3))
(1 2 3)
2_(RPLACA A '(7 8))
((7 8) 2 3)
Note: No SETQ is necessary because the actual list is changed.
3_ A
((7 8) 2 3)
4_ (RPLACA (CDR A) '(5 6))
((5 6) 3)
5_ A
((7 8) (5 6) 3)





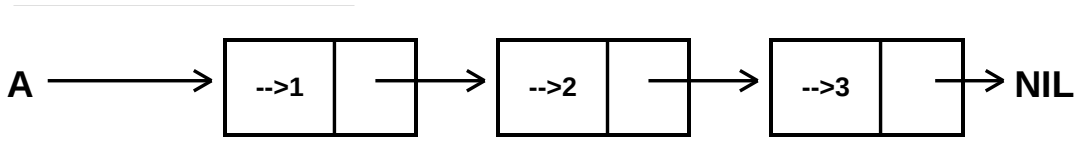


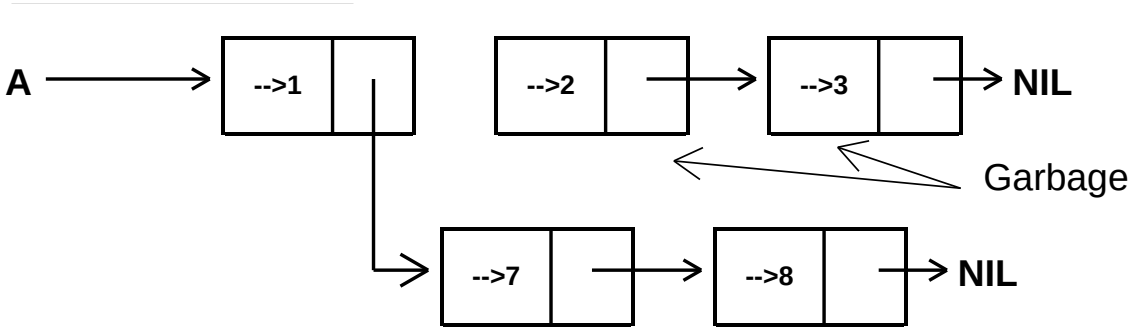
(RPLACD *X Y*)ž replaces the CDR of the CONS cell *X* with a pointer *Y*. RPLACD returns *X*.

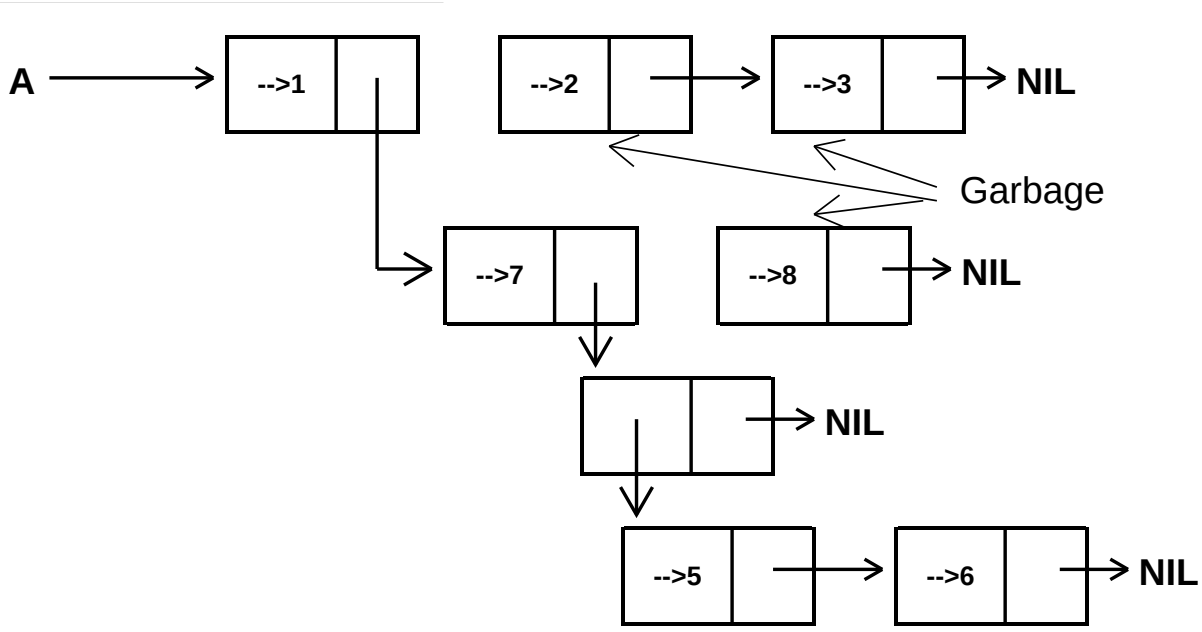
Example:

6_(SETQQ A (1 2 3))
(1 2 3)
7_(RPLACD A ’(7 8))
(1 7 8)
Note: No SETQ is necessary because the actual list is changed.
8_ A
(1 7 8)
9_ (RPLACD (CDR A) ’((5 6)))
(7 (5 6))
10_A

(1 7 (5 6))







(NCONC *X1 X2 X3 ...*)ž Like APPEND, but does not copy the lists before doing the list splicing.

Example:

11_(SETQQ B (1 2))

(1 2)

12_(SETQQ C (3 4))

(3 4)

13_ (NCONC B C)

(1 2 3 4)

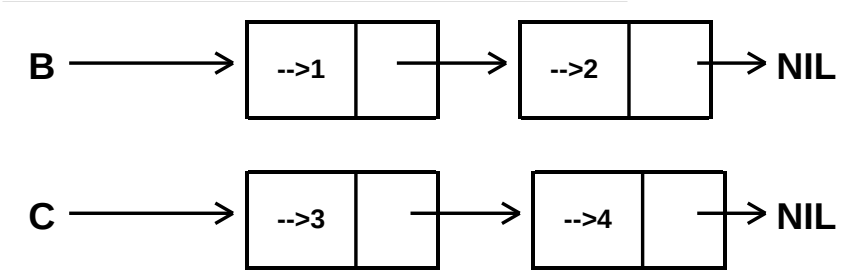
Note: No SETQ is necessary because the actual B list is changed.

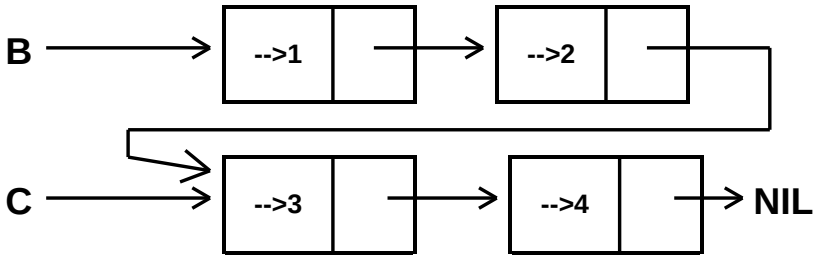
14_ B

(1 2 3 4)

15_ C

(3 4)





Some dangers in using the destructive functions

If you are not careful, RPLACA, RPLACD, and NCONC can cause many strange things to happen to your list structures.

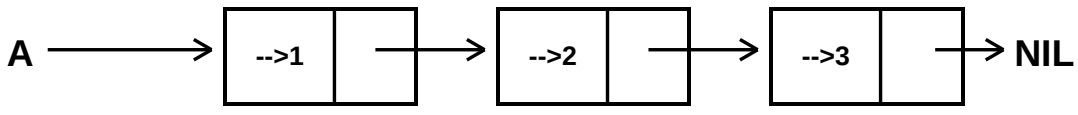
Two of typical strnage things are changing things you didn't mean to change and circular list structures.

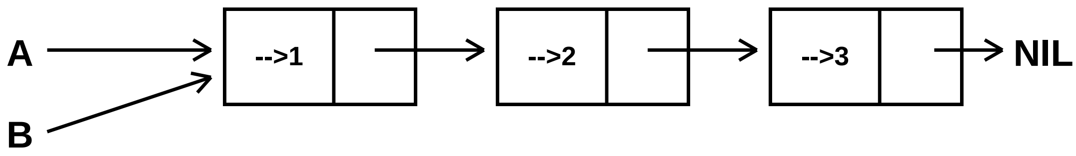
Example: Changing what you didn't mean to change.

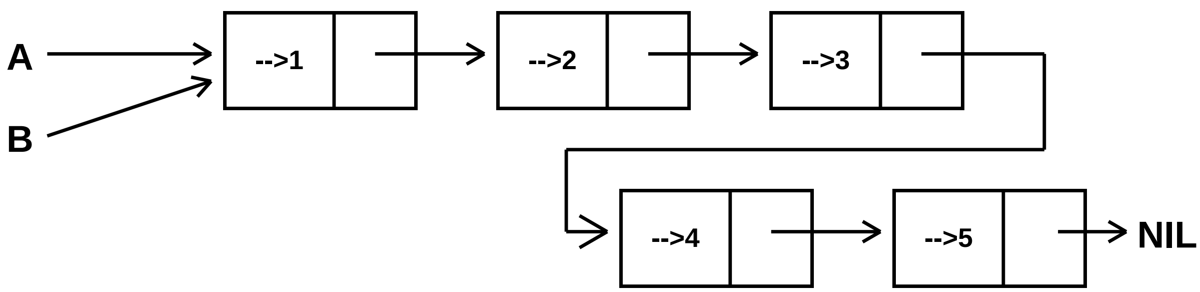
Plan: you are going to change A, so you hold on to the original by SETQing B to the original value of A.

23_(SETQQ A (1 2 3))
(1 2 3)
24_ (SETQ B A)
(1 2 3)
25_ (NCONC A '(4 5))
(1 2 3 4 5)
26_ A
(1 2 3 4 5)
27_ B
(1 2 3 4 5)

Note that the NCONC effects the value of B even though it is not mentioned at all in the NCONC function call.







Second example: Circular lists.

43_(SETQQ B (1 2))

(1 2)

44_(SETQQ C (3 4))

(3 4)

45_ (NCONC B C B)

*(1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3
4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3
4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3
4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3
4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3
4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3
4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3
4 1 2 3 4 ... forever!!!!!!*

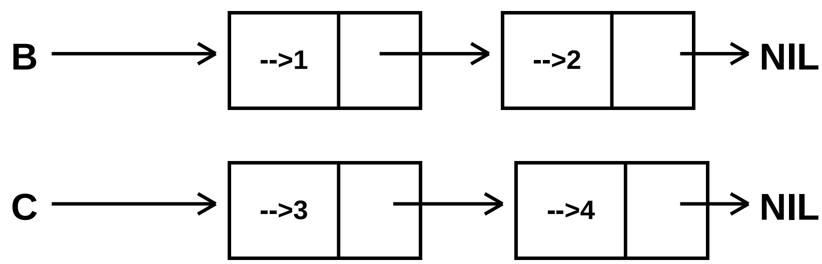
46_ B

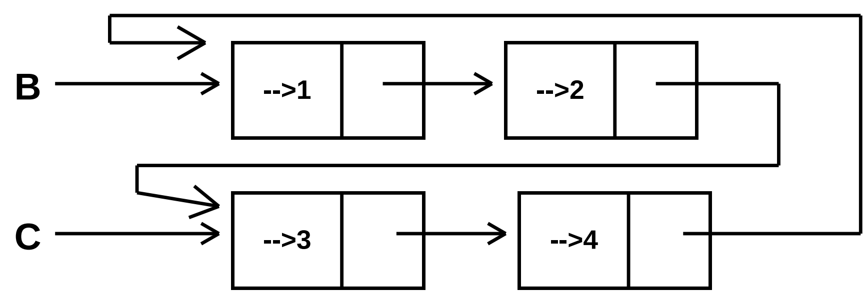
*(1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3
4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3
4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3
4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3
4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3*

4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3
4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3
4 1 2 3 4 ... forever!!!!!!

46_ C

(3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1
2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1
2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1
2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1
2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1
2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1
2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1
2 ... forever!!!!!!





Representing Datatypes and Arrays

Datatypes and Arrays can be modelled as a fixed-length one-dimensional vector of cells, each of which contains a pointer to some Lisp data object. *Note: the cells are NOT CONS cells because they contain only a single pointer.*

For arrays, the cells are index by number. For Datatypes, the cells are indexed by field name.

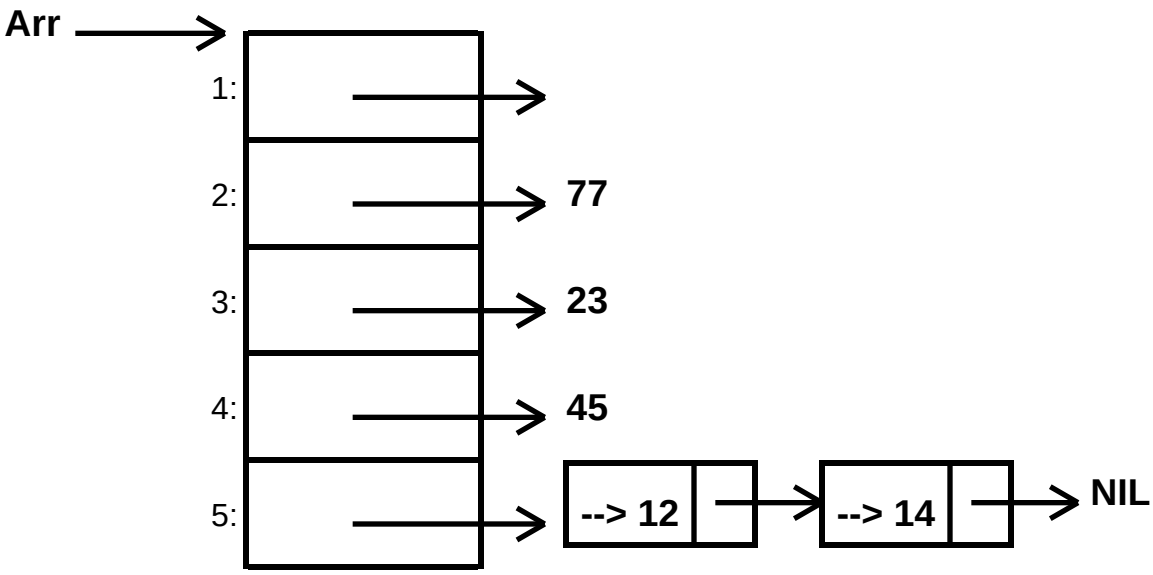
Example:

55_(SETQ Arr (ARRAY 5))

{Array}#6,12345

56_(FOR X IN '(55 77 23 45 (12 14)) AS I FROM 1 DO (SETA Arr I X))

NIL



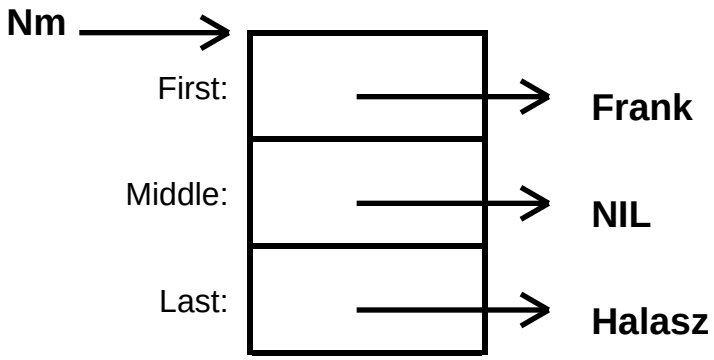
Second example:

34_(DATATYPE LC.Name (First Last Middle))

LC.Name

35_(SETQ Nm (CREATE LC.Name First _ Frank Last _ Halasz))

{LC.Name}#34,12378



Representing Strings

The internal representation of a string has two parts: a ***string pointer*** and a ***character vector***.

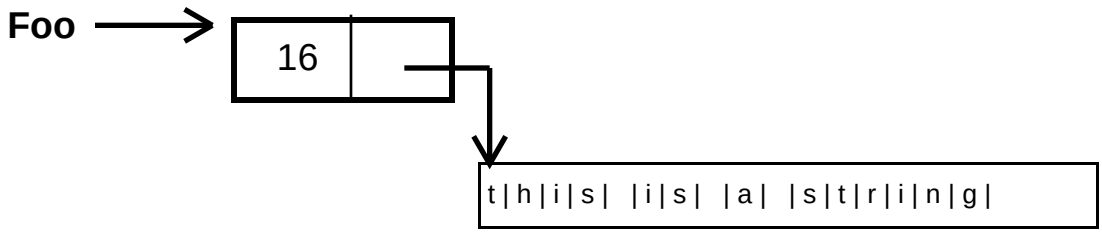
The *string pointer* is a cell (not a CONS cell) containing two things:

1. The number of characters in the string.
2. A pointer to the start of the characters for this string in the character vector.

The *character vector* is just a sequence of characters.

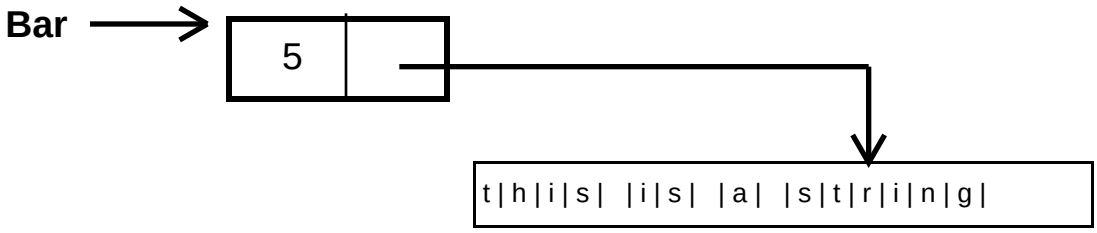
A pointer to a string is a pointer to the string's string pointer.

Example: **(SETQ Foo "this is a string")**



There can be more characters in the character vector that there in the string.

Example: **(SETQ Bar (SUBSTRING Foo 11))** returns *"string"*.



Some functions that create and return strings create only a new string pointer and use an old character vector. Other functions create both a new string pointer and a new character vector.

As shown above, SUBSTRING creates just a new string pointer and uses the old charactre vector.

MKSTRING and ALLOCSTRING create both string pointers and character vectors.

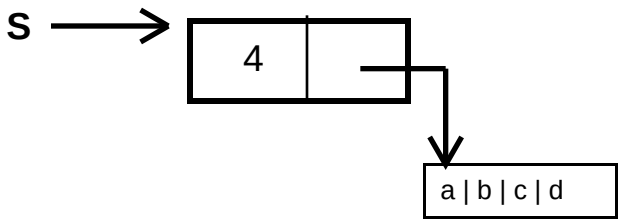
There are destructive string functions analogous to RPLACA, etc. for lists. These are GNC, GLC, and RPLSTRING.

GNC ž returns the first character of a string (as an atom) and then removes that character from the string by updating the string pointer. If the argument is not a string, it is made into a string using its print name.

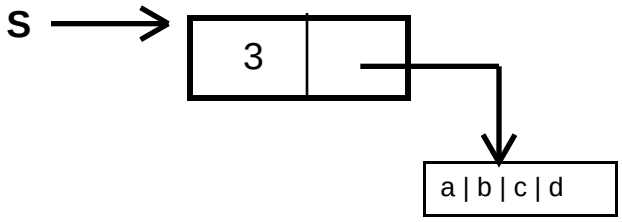
Example:

12_(SETQ S "abcd")
"abcd"
13_(GNC S)
a
14_S
"bcd"
15_ (GNC S)
b
16_ S
"cd"

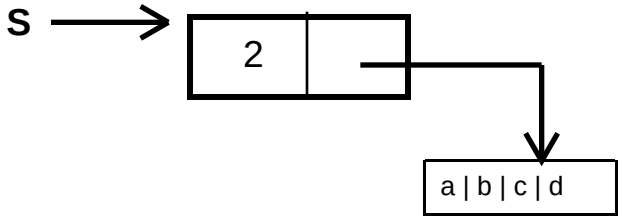
After (SETQ S "abcd")



After first (GNC S)



After second (GNC S)



GLC ž returns the last character of a string (as an atom) and then removes that character from the string by updating the string pointer. If the argument is not a string, it is made into a string using its print name.

Example:

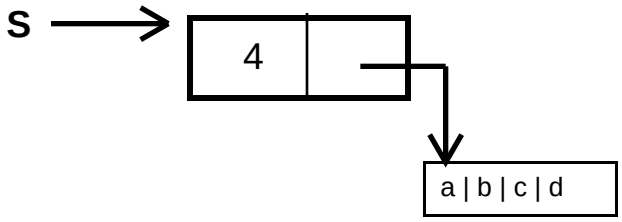
12_(SETQ S "abcd")
"abcd"
13_(GLC S)
d
14_S
"abc"
15_ (GLC S)

c

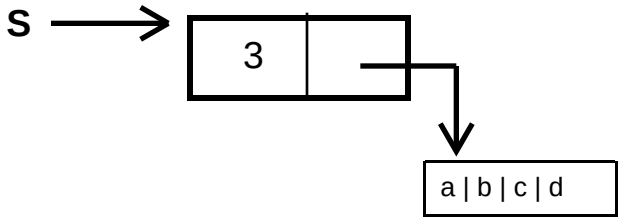
16_ S

"ab"

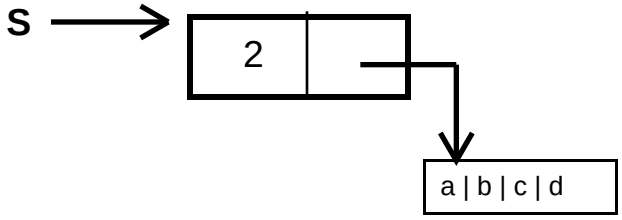
After (SETQ S "abcd")



After first (GLC S)



After second (GLC S)



(RPLSTRING *String1 N String2*) ž changes the character vector of *String1* to include the characters in *String2*, starting at position *N* in the *String1*. *N* can be negative just as in SUBSTRING. If *String1* and/or *String1* are not strings, they are made into strings using their print names.

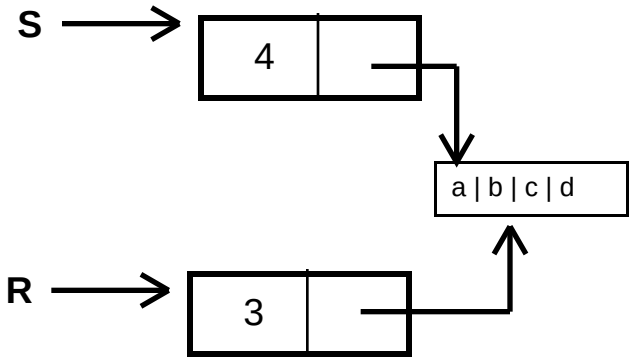
Example:

12_(SETQ S "abcd")

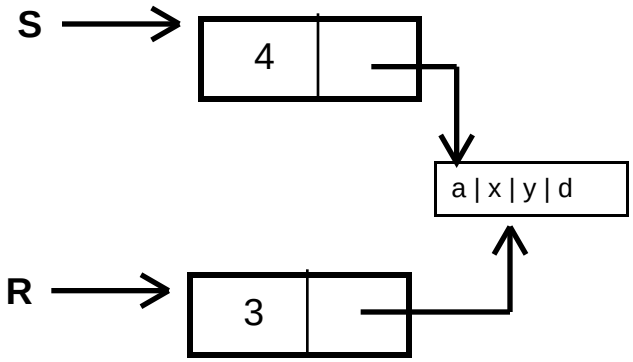
"abcd"

13_(SETQ R (SUBSTRING S 2))
"bcd"
14_ (RPLSTRING S 2 "xy")
"axyd"
15_ S
"axyd"
Note: since RPLSTRING is destructive, R is messed up too.
16_ R
"xyd"

After (SETQ S "abcd") and (SETQ R (SUBSTRING S 1))



After (RPLSTRING S 2 "xy")



Sameness and Equality in Lisp: EQ versus EQUAL

Consider the following:

1_(SETQ A (LIST 1 2 3))
(1 2 3)
2_ (SETQ B (LIST 1 2 3))
(1 2 3)

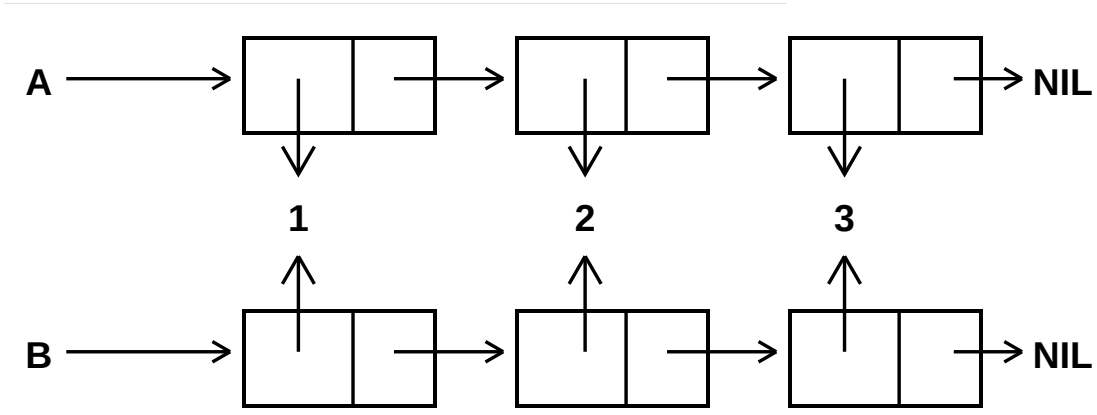
Do A and B have the same value? Are the values of A and B equal?

By most peoples definition, the answer is *Yes*. But in Lisp, the answer is it depends what you mean by *equal*.

The values of A and B are equal in the sense that they are both lists containing the identical sequence of items.

The values of A and B are NOT equal in the sense that they are made up of entirely different CONS cells. This is because the function LITS works by creating new CONS cells to make a list out of its arguments (see above).

The situation is clear if you diagram the two lists.



Contrast the previous situation with the following situation:

1_(SETQ A (LIST 1 2 3))
(1 2 3)
2_ (SETQ B A)
(1 2 3)

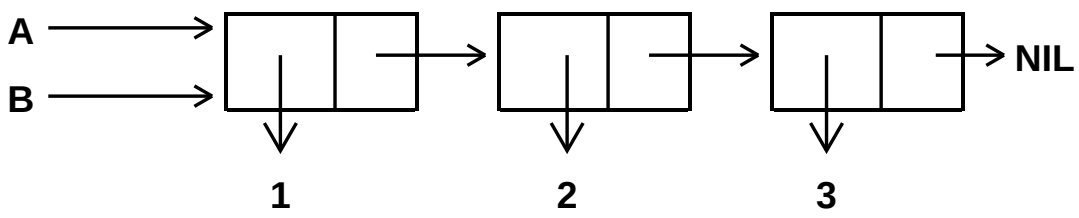
Do A and B have the same value? Are the values of A and B equal?

By most peoples definition, the answer is *Yes*. And in Lisp, the answer is *Yes*.

The values of A and B are equal in the sense that they are both lists containing the identical sequence of items.

The values of A and B are also equal in the sense that they are made up of exaclty the same CONS cells. This is because the function SETQ just sets the value of its first argument to be its second argument without creating any new structures (see above).

The situation is clear if you diagram the list.



Lisp has two concepts of sameness or equality:

1) containing the same "information"

2) being exactly the same data objects

Data objects in Lisp can be equal in the first sense without being equal in the second sense. The opposite is NOT true ÿ identical data object always contain the same "information".

EQ ž returns T if its two arguments are pointers to the exact same internal data object. NIL, otherwise.

EQUAL ž returns T if its two arguments are the same type of data object and contain the *same information*. NIL, otherwise.

Same information is determined as follows:

1) The two arguments are EQ

2) The two arguments are equal numbers

3) The two arguments are strings that are STREQUAL (have the same sequence of characters).

4) The two arguments are lists, where the CARs are EQUAL and the CDRs are EQUAL.

Basically, two lists are EQUAL if they contain the same set of atoms within the same list format.

Examples:

The results of (LIST 1 2 3) and (LIST 1 2 3) are EQUAL but not EQ.

The results of (LIST 1 (LIST 2 3)) and (LIST 1 (LIST 2 3)) are EQUAL but not EQ.

The results of (MKSTRING 'AB) and (MKSTRING 'AB) are EQUAL but not EQ.

Following (SETQ A B), (EQ A B) and (EQUAL A B) return T.

If A points to a list, (EQ (CDDR A) (CDR (CDR A))) is T.

If A points to a list, (EQ (CDR (CONS 1 A)) A) is T.

If A points to a list, (EQ (RPLACA A 1) A) is T.

If A points to a list, (EQ (CONS 22 A)(CONS 22 A)) is NIL, but (EQUAL (CONS 22 A)(CONS 22 A)) is T.

(EQ 'Atom 'Atom) is always T.

For integers less than 65,000, (EQ *SmallInteger SmallInteger*) is T.

For larger integers and for floating numbers, (EQ *Number Number*) is generally NIL, but (EQUAL *Number Number*) is T.

As a general rule, use EQUAL unless you know you want to test for the same exact data structure or you are comparing atoms.

References

CONS cells and lists represented as CONS cells is covered in:

Winston & Horn, Chapter 9

Touretzky, Chapter 2

RPLACA, et al. are covered in Section 2.5 of the IRM.

GNC et al. are covered in Section 2.6 of the IRM.

EQ et al. are covered in Section 2.2 of the IRM and in Touretzky, page 155 and in Winston and Horn, page 142.

Exercises

Attached.