C7—Lisp Programming

Russell Bradford

1 Introduction

This course is about Lisp. Or rather this course is about how to program in Lisp. The distinction is a great one: Lisp, as a language to use, to program with, to learn, is one of the simplest of all languages. There is no difficult syntax, and every piece of code or data is treated in exactly the same way, with only very few (but important) exceptions.

On the other hand, some of the hardest language semantic questions can be posed and written using Lisp, involving subtle or complex ideas, readily adapting to accommodate every new fad and paradigm.

We shall concentrate on the programming aspect, and perhaps give the merest hint of the semantic underpinning.

1.1 What is Lisp?

The name "Lisp" means "List Processor." The basic object that we manipulate is the list. A list is an ordered collection of objects, each of which may be other lists, or things which are not lists, called "atoms." An atom is any object that does not have a list structure. Thus

```
(the cat sat (on the) mat)
```

is a list of 5 objects, the 4$^{\text{th}}$ being a list of 2 objects. The symbols `the`, `cat`, etc., are atoms. Here is another example of a list

```
((one 1) (two 2) (three 3))
```

which associates a symbol with a numeral (each of which are atoms).

Lists come in all shapes and sizes, for example

```
(((x 10) 2) ((x 5) ((y 2) 1) 1) ((x 2) 3) 2)
```

is a more complicated list, which some people like to think of as a representation for the polynomial $2x^{10} + (y^2 + 1)x^5 + 3x^2 + 2$.

Notice the emphasis on symbols: Lisp is a *symbolic* language, where the data items we handle are symbols. This is in contrast to, say, C or Fortran, where all we have to play which is numbers. Thus Lisp is used heavily for symbolic problems, such as logic, computer algebra, and AI.

There is a special list that stands out amongst all lists, the list that contains no objects:

```
()
```

This object has the name `nil`, and is strange in that it commonly regarded to be both a list and and atom.

This list aside, (we we can liken to the empty set, which has all sorts of exceptional properties) we think of a list in a recursive manner as being composed of two parts, namely the first element, and all the rest. Thus the first element of `( (1 3) (3 4) (4 5) )` is the list `(1 3)`, and the rest is `( (3 4) (4 5) )`. This accords well with our naturally recursive outlook on life: the length of a list is 1 for the first element plus the length of the rest of the list.

These parts have names, the first element of a list is called its `car`, the rest is called its `cdr`. To understand these names we need a little history.

1.2 A Little History

Lisp is very old language, second only to Fortran in the family tree of high level languages. In the late 1950s John McCarthy described a language, named "Lisp 1," that had all the important parts of the current language, in particular the lists, and atoms. He was trying to develop a language that would adequately express the λ-calculus of Alonzo Church that arose in the 1930s.

McCarthy wrote a paper describing a Lisp function that would interpret any Lisp function, including itself. This function, `eval`, is like a universal Turing machine. The Lisp language he described was a universal language that could implement all other languages. In fact, this latter is truer than you might at first think—many modern top end compilers are written in languages which, if not actually Lisp, at least are morally Lisp. The first Fortran compiler for UNIX, f77, was written in Lisp. Those of you who use GNUemacs may be interested to know that it is written mainly in Lisp.

Lisp never looked back. From Lisp 1 there spawned dozens of Lisps, each exploring its own niche of computer science, each choosing a set of ideas to develop, each picking a new set of names for the basic functions. Lisp is not a language, it is a family of languages: a family tree can be constructed showing the major paths of development. Unfortunately, this profusion of dialects, like the Tower of Babel, has lead to a state where any particular Lisp program may or may not run on a particular implementation of Lisp. Or if your program does run, it may do something entirely contrary to your expectation.

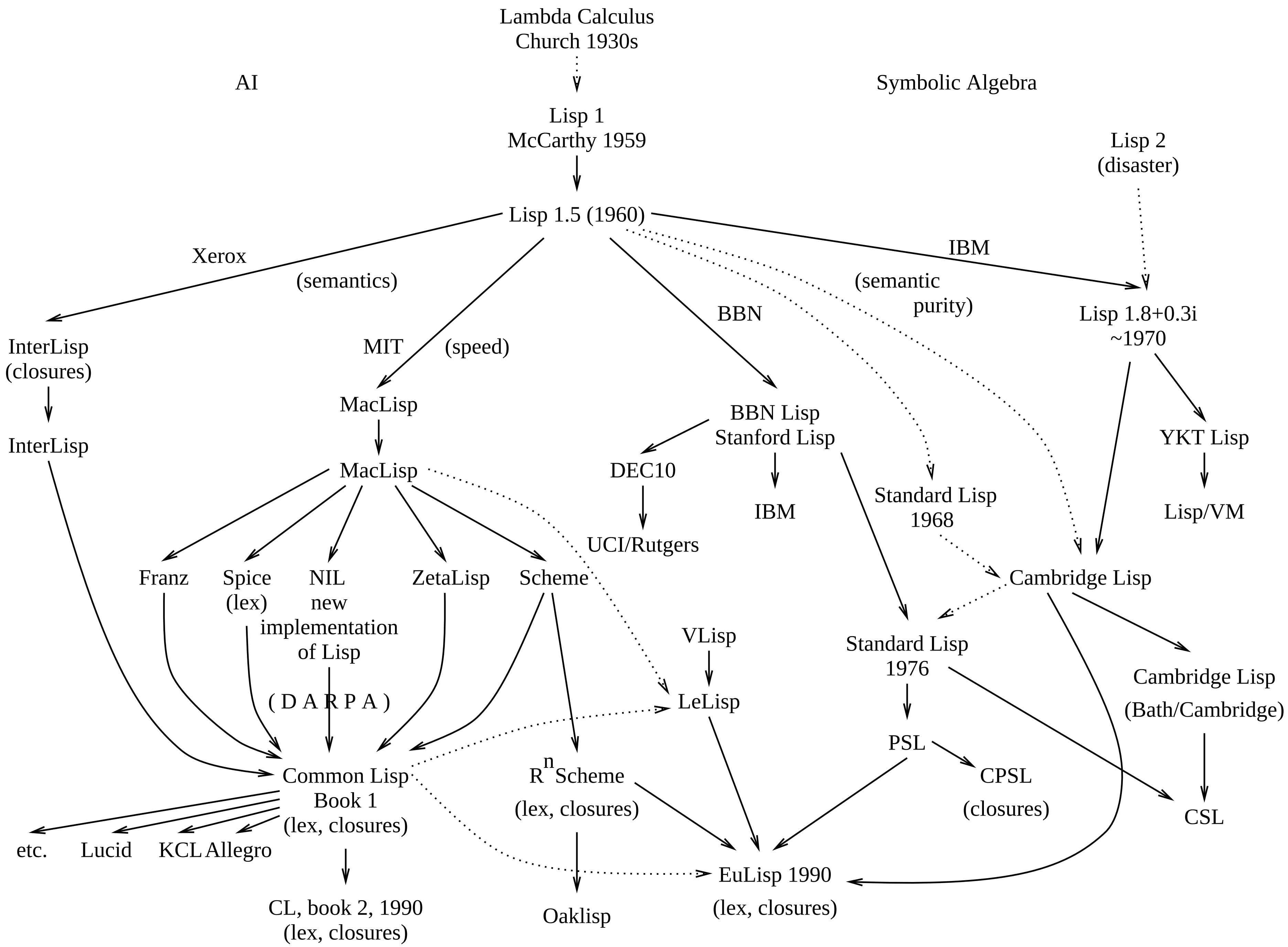
This may seem a disaster to a Fortran programmer, who wants her program to run on any compiler on any machine. But the diversity has led to important developments and cross-fertilization of ideas, as perhaps, objects from one branch are merged with logic programming from another to confront pure functional forms from a third.

There are a few things common to all Lisps: lists and atoms, the function `cons` to construct lists, and the functions `car` and `cdr` to take them apart.

McCarthy implemented his Lisp on an IBM 704 whose instruction format had four sections named `cpr` (contents of the prefix register), `ctr` (contents of the tag register), `car` (contents of the address register), `cdr` (contents of the decrement register). The way he arranged things was when he referred to a list, the car would be in the address register, the cdr in the decrement register. And the names just stuck. Some people claim we keep the names as a homage to McCarthy, others are more cynical.

Beyond these, things get a little vague. For example, the Cambridge Lisp function to add two numbers is called `plus`, while the Common Lisp analogue is named `+`. This sort of variance is rife.

Even the value of the `car` and `cdr` of `nil` varies according to taste. Either it is an error to try to take the `car` (the value of `nil` is an atom), or it is `nil` (the value of `nil` is a list, thus you can take a `car` of it).


Lambda Calculus
Church 1930s
AI
Symbolic Algebra
Lisp 1
McCarthy 1959
Lisp 2
(disaster)
Lisp 1.5 (1960)
Xerox
(semantics)
IBM
(semantic
purity)
BBN
MIT
(speed)
InterLisp
(closures)
InterLisp
MacLisp
MacLisp
Lisp 1.8+0.3i
~1970
BBN Lisp
Stanford Lisp
DEC10
UCI/Rutgers
IBM
Standard Lisp
1968
YKT Lisp
Lisp/VM
Franz
Spice
(lex)
NIL
new
implementation
of Lisp
ZetaLisp
Scheme
Cambridge Lisp
VLisp
LeLisp
Standard Lisp
1976
Cambridge Lisp
(Bath/Cambridge)
(D A R P A)
PSL
CPSL
(closures)
CSL
Common Lisp
Book 1
(lex, closures)
R n Scheme
(lex, closures)
etc.
Lucid
KCL
Allegro
CL, book 2, 1990
(lex, closures)
Oaklisp
EuLisp 1990
(lex, closures)

1.3 The Ball of Mud

A poetic reason for the divergence of Lisp was given by Moses: a language such as APL is like a diamond. It is perfectly symmetric, and shines brightly. However, if you wish to add a new feature to the language, the symmetry is smashed, and the diamond cracks and shatters.

Lisp, on the other hand, is a ball of mud. Essentially shapeless, you can easily add new extensions and ideas, and all you get is a larger ball of mud ready and able to accept more and more. Lisp is infinitely extensible: you can add new functions that have exactly the same importance to the system as the built-in commands, or even redefine the built-in commands. You can, if you feel that way inclined, redefine the whole syntax. Imagine the possibilities if C allowed you to redefine the `while` loop, or Fortran let you introduce exactly that form of `DO` loop that you required for your application.

Over the years many important new ideas have first been tried out using Lisp. Optimizing compilers were first written in Lisp. The idea of tail recursion removal was implemented in Lisp compilers as early as 1962. Only since the late '80 have other languages caught on: C compilers are beginning to spot the advantages of tail recursion. Object oriented systems and object oriented graphics were first developed using Lisp. Similarly were pure functional ideas, and logic programming and rewrite systems. Other languages have subsequently been conceived (Prolog, ML, C++) that are based on these ideas, but they were all prototyped in Lisp.

Lisp typically runs as an interpreter, using a *read-eval-print* loop. This loop sits and waits for input, reads it, evaluates the expression it has read, and then prints the result. (As an aside, the read-eval-print loop is often written in Lisp, and could be redefined into something more exotic, such as a control program for some piece of machinery.) This interactive use is one of the reasons Lisp is so important as a prototyping tool: a function can be tried, and altered, and then re-tried as often as you wish, and function call always refers to the *latest* definition of a function. Thus if you define `foo`, and then `bar` which calls `foo`, you can define `bar` in the knowledge that `foo` will use your new definition. (This is somewhat like the language Forth (which was developed using the idea from Lisp, of course).) It is possible to have only a Lisp compiler, but this removes the interactive developmental feel of the language.

Lisp enjoys the universal Turing property, not only theoretically, but also practically.

1.4 The Language

We shall be using no particular dialect of Lisp in general, except when we need to make specific points about how dialects vary. As a default we will use function names from EuLisp (which are similar to those from Scheme and Common Lisp, but different from those in Cambridge Lisp).

EuLisp is a "modern" Lisp, and the implementation we have is the world's first nearly complete version, and as such it is a little flakey here and there: the implementation is named "FEEL," for "free, and eventually EuLisp." t is coded in both C and EuLisp, the former being a bootstrap for the latter. (This is typical way of porting Lisp: write a small Lisp interpreter in a suitable language, e.g., C, and use this to load and run a Lisp program, typically a compiler, which you then use to compile Lisp source of the Lisp language. When this is done, you are left with a compiled Lisp on your new processor.)

Lists are the basic data structure in Lisp, as we have said. Lisp programs are included, themselves being lists.

```
(+ 2 3)
```

This is a (short) program to compute the sum of 2 and 3. The syntax of a function call is

```
(fun arg arg ... )
```

This is only a slight change from the more usual `fun(arg, arg, ... )` but it allows an immense flexibility, as all function calls, programming structure and data are in exactly the same shape.

When you type `(+ 2 3)`, or any other expression, at a Lisp interpreter, it regards that expression as one to be evaluated. It does this by first evaluating the arguments (which may themselves be arbitrary expressions), and then applying the function to the evaluated arguments. With a couple of noteworthy exceptions, *everything* in Lisp is done this way. Thus

```
(+ (+ 1 2 3) (* 2 8))
```

evaluates `(+ 1 2 3)` to get 6, and `(* 2 4)` to get 8, and then evaluates `(+ 6 8)` to return 14. Notice, also, that `+` can take a variable number of arguments. Many functions are like this, notably in the area of arithmetic. In Cambridge Lisp this would be written

```
(plus (plus 1 2 3) (times 2 8))
```

which is just the same, but more verbose.

Functions like `+` or `plus` act on atoms (numbers) and return atoms. What about a function of a list? Well, `length` is the name of a function that takes a list and returns its length. So `length` applied to the list `(2 4 6)` returns 3. But it we simply type

```
(length (2 4 6))
```

at the interpreter, the following happens: the interpreter wishes to evaluate the argument `(2 4 6)`. Recursively, it find the value of 4 to be 4, and of 6 to be 6. Then it tries to apply the function `2` to the arguments 4 and 6. Unless we have been particularly perverse, there is no function of that name, and the interpreter complains

```
Trapped non-continuable invalid-operator!
  message: invalid operator
  error-value: 2
  args: (3 4)
  op: 2
```

in FEEL and

```
Error: 2 is invalid as a function.
Error signalled by EVAL.
```

in KCL, both indicating you tried to use `2` as a function.

(Aside: the Lisps now go into a "condition handler" or "error handler". These are used for debugging—for now, use `!exit` in FEEL or `:q` in Common Lisp to get back to the top level read-eval-print loop.)

The problem is how to convince the interpreter that this occurence of `(2 4 6)` is to be understood as a three-element list, not a function to be evaluated.

Atoms such as 1, 2, and 3 evaluate to themselves, whereas an atom like "`+`" have a value which can be any Lisp object.

(Another aside: atoms like `+` and `*` have no special syntactic significance, they have exactly the status of a atom like `wombat`; this is entirely unlike C or ML, where they are specially treated.)

Suppose, somehow, the symbol `two` has been associated with the value 4. Then

```
two
```

returns 4, and

```
(+ 2 two)
```

is 6.

To help bootstrap ourselves, and allow us to input unevaluated symbols and lists there is a function with name `quote` which returns its argument *unevaluated*. Thus the value of

```
(quote (2 4 6))
```

is `(2 4 6)`. The value of `(quote two)` is the symbol `two`.

The function `quote` is one of the above-mentioned exceptions, and is so important that there is a syntactic abbreviation for it, the quote (`'`) symbol. So `'two` is the same as `(quote two)` (which evaluates the the symbol `two`), and `'(2 4 6)` is the same as `(quote (2 4 6))` (which evaluates to the list `(2 4 6)`). This is purely a syntactic convention, and quite often the reader in the read-eval-print loop will transform a `'` into the functional form before the evaluator sees it.

Now we can compute the length of a list

```
(length '(2 4 6))
```

returns 3. The function `car` returns the car of a list, `cdr` returns the cdr.

```
(car (cdr '(1 (2 3) 4)))
```

is 2. And so on.

Notice that

```
'(+ 1 2 3)
```

returns the list of length 4

```
(+ 1 2 3)
```

It is instructive to predict result of evaluating the expression

```
''two
```

1.5 Books

In essence, any or no book is recommended for this course. Any book is enough to give a good idea of what Lisp looks like, and how to write programs, but no book gives a world view ranging over all the Lisps you are likely to meet. Here are some names anyway

1. "Lisp." Winston and Horne. This has a grubby programming style, not favoured by purists.

2. "Artificial Intelligence Programming." Charniak and McDermot. Only if you want to do AI next year.
3. "Structure and Interpretation of Computer Programs." Abelson and Sussman. This book describes *Scheme*, an important but exceptional Lisp.
4. Cambridge Lisp Manual. We shall try to keep a copy in the teaching lab.
5. Common Lisp, Guy Steele Jr. A description of the soon-to-be Common Lisp ANSI standard. Look at this, and marvel at its thickness. The latest edition is now 1000 pages long.
6. Any book with "Lisp" in its title.

1.6 Running Lisp

On the machines here we have several Lisps, notably the implementation of EuLisp and and implementation of Common Lisp. EuLisp is run by the command `feel`, Common Lisp by `kcl`. The version of Common Lisp we have is *Kyoto* Common Lisp, implemented by Yuasa and Hagiya of Kyoto University. They are both also available on ss1.

Beware: these Lisps are large (on the order of 2Mb image size), so don't be surprised if you can't get several of them running at once.

After some introductory stuff FEEL gives a prompt like

```
user!0>
```

Now you type some Lisp expression, when it evaluates it, and prints the value followed by a new prompt.

Common Lisp has a prompt

```
>
```

for the same purpose.

To exit from FEEL or KCL, use control-D.

The command "`lisp`" gives you Cambridge Lisp, with prompt `Input:`. Arithmetic functions in Cambridge Lisp are named `plus`, `difference`, and the like, whereas in Common/EuLisp they are +, and −. To exit Cambridge Lisp, use `fin`.

There are also a selection of other Lisps available, namely

`scheme`	a comprehensive Scheme interpreter
`siod`	a small Scheme subset
`xlisp`	a small Common Lisp subset
`xscheme`	a substantial Scheme subset
`oaklisp`	a Scheme-like Lisp

To exit these use `^D` for siod and xlisp, `^Aq` for scheme, and `(exit)` for Oaklisp.

Also GNUemacs has a built-in Lisp interpreter—in fact GNUemacs is mostly implemented in Lisp.

2 Basics

We shall now take a little more organized walk through Lisp, starting with the basic data items.

2.1 Lists and Atoms

Let us start from the indivisible elements that make up the Lisp universe. These are the *atoms*. We can classify atoms into two types, those which evaluate to themselves, and those that don't. What this means is that an atom of the first class, for example the integer `23` has value the integer 23. Also the string `"hello world"` has value `"hello world"`. Thus, when the Lisp evaluator sees an example of this type, it returns what you gave it.

On the other hand, the class of *symbols* are completely different. Symbols have values, just like in other languages, but the values can be any Lisp object. When the Lisp evaluator see a symbol it returns "the" value of the symbol. (Of course, which value is dependent on where you are, which recursive invocation of a function you are executing, and so on.)

Thus the value of `two` might be the integer 2, or it might be the string `"forty-two"`. What is more interesting, is that the value of a symbol may be a symbol. Perhaps the value of `twoplustwo` is `four`. And then maybe the value of the value of `twoplustwo` is 4. Unlike C, Lisp does not have a type associated with a symbol ("strong typing is for weak minds"). Any symbol can have as value any Lisp object—in fact the *objects* have types, say integer or string or whatever, but the symbols don't.

Having collected a few atoms together we want a convenient method for bunching them together. To do this we use *lists*.

A list is an ordered sequence of zero or more Lisp objects. This definition is very powerful, as it allows the members of a list to be arbitrary lists and atoms.

(Aside: some lists contain themselves as members—there is no axiom of foundation to help us here! However, such lists need a trifle more sophistication to understand than we currently have, so we may assume for now that all lists are finite.)

Lists are denoted by an open parenthesis followed by a space-separated sequence of its members, followed by a close parenthesis.

Here is an example:

```
(+ 2 4)
```

This list has three elements, firstly the symbol `+`, then the atoms `2` and `4`.

It is very important to realise that lists do *not* evaluate to themselves. A list presented to the interpreter is viewed as a function call to be evaluated, the first element of the list being a function to be applied the other elements of the list, which are the arguments. A list is evaluated in the following manner: firstly, each argument is (recursively) evaluated, then the function described by the first element is applied to the results. The value of this function call is the value of the list. (The *order* of evaluation of arguments is Lisp dependent.)

Thus `(+ 2 3)` evaluates to the expected result since 2 and 3 evaluate to themselves, and `+` names a function which adds its arguments.

If the symbol `four` has the value 4, and the symbol `twoplustwo` has the value `four`, then the expression `(+ four 5)` has value 9, but `(+ twoplustwo 6)` causes an error, as the function named by `+` does not know how to add the symbol `four` to the number 6.

The list

```
(* (+ 3 4) (- 5 2))
```

has value 21.

The empty list `()` has a slightly special status in that it is commonly regarded as both a list of zero elements, and as an atom (as it is "indivisible"). The value of `()` is itself, namely `()`. If you like, it it where the two universes meet. The symbol `nil` is reserved to hold this value. Some implementations of Lisp blur the distinction between the symbol `nil` and its value `()`, allowing then to be syntactically identical. Others have `nil` to be a symbol with constant value `()`. We shall occasionally use "`nil`" with the syntactic blur, but note that this is the "wrong" way of doing things.

Additionally, some implementations prefer `nil` to be a list, and allow `car` and `cdr` to be applied (with result `nil` in both cases); others take the atomic view that `nil` cannot be further divided, and it is an error to try to take the `car` or `cdr` of `nil`.

2.2 The Truth

The symbol `t`, like `nil`, has a special meaning. It is a symbol with constant value `t`. It is not really self-evaluating, but has much the same properties—it's just that its value is itself! So `(length (list t t t))` is valid, and has value 3.

The `()` and `t` are customarily taken as the boolean false and true: the function `equal`, which compares two objects to see if they are the same, will return `t` if it decides they are the same, otherwise `()`. For example, `(equal 2 3)` is `()`, whereas `(equal '(1 2) (list 1 2))` is `t`.

On the other hand, it also customary to regard every non-`()` value to represent true, just as in C, where 0 is false, and everything else it true, but -1 is a "favoured" representation for true.

Thus the function `member` takes an object and a list, and checks whether the object is a member of that list. It returns `()` if not, but if it is, it returns the remainder of the list from that object onwards. This being non-`()`, denotes true, but returns some extra information, viz., where in the list this object lives. On the whole Lisp functions return as much information as possible, even the boolean functions.

This means `(member 'a '(1 2 3))` is `()`, but `(member 'a '(b a d))` is `(a d)`. Notice that `(member 'a '(1 (a b) 2))` is `()`.

2.3 Pairs and Lists

Lists in Lisp can be arbitrarily long (with the usual memory constraints). To achieve this flexibility, they are often implemented using *pairs*. A pair is an object that contains just two objects, named the car and the cdr. The cdr is often another pair, whose cdr is another pair, whose cdr is another pair, and so on. Thus we get a linked list of pairs, where we store information in the car, and a link in the cdr. A list is simply such a chain.

We create pairs using `cons` (for *construct*). This takes two arguments, and returns a pair containing those arguments. So `(cons 'a 1)` is the pair `(a . 1)`. The notation used for a pair is

```
(car . cdr)
```

In the read-eval-print loop the printer chooses to format pairs specially: if you want to print a pair use the following algorithm

1. Print an open parenthesis.
2. Print the car.
3. If the cdr is a pair, make it the current pair, print a space, and go to step 2.

4. If the cdr is not null, print a space, a dot, a space, and the cdr.
5. Print a closing parenthesis.

Using this algorithm, if the printer gets the pair

```
(a . (1 . (b . (2 . ()))))
```

it prints it as

```
(a 1 b 2)
```

This is because the above pair representation is precisely how the list `(a 1 b 2)` is stored. Again, it is merely a convenience and convention in the reader and printer that the dots and parentheses are dropped. The pair

```
((1 . (2 . ())) . (3 . 4))
```

is printed as

```
((1 2) 3 . 4)
```

Notice how `()` is used as an end of list marker.

We can tack objects on to the start of a list using `cons`

```
(cons 'a '(1 2 3))
```

returns `(a 1 2 3)`: this is internally `(a . (1 . (2 . (3 . ()))))`. The function `list` is defined as a repeated application of `cons`: `(list obj1 obj2 obj3 ... objn)` is equivalent to

```
(cons obj1 (cons obj2 (cons obj3 ... (cons objn ()) ... )))
```

A list that is terminated by `()` is a *proper list,* and one that is terminated by a non-`()` value is an *improper list.* Thus `(1 2 3)` (i.e., `(1 . (2 . (3 . ())))`) is proper, but `(1 2 . 3)` (i.e., `(1 . (2 . 3))`) is improper.

The three functions `cons`, `car`, and `cdr` are the building (and unbuilding) blocks of Lisp.

2.4 A Few Functions

Here is a random selection of interesting functions, just to get ourselves going.

2.4.1 Some arithmetic

`(+ n1 n2 ...)` Form the sum of `n1`, `n2`, … (Cambridge Lisp: `plus`)

`(- n1 n2)` Subtract `n2` from `n1`. (Cambridge Lisp: `difference`)

`(* n1 n2 ...)` Form the product. (Cambridge Lisp: `times`)

(/ n1 n2) The integer quotient if both are integers, otherwise a floating point divide. (Common Lisp: much more complicated, no exact equivalent, Cambridge Lisp: `quotient`)

`(expt n1 n2)` The exponential.

`(gcd n1 n2)` The GCD of integers.

`(zerop n)` This is `t` if `n` is zero, otherwise `()`.

`(> n1 n2)` This is `t` if `n1` is greater than `n2`, otherwise `()`. (Cambridge Lisp `greaterp`.)

`(< n1 n2)` This is `t` if `n1` is less than `n2`, otherwise `()`. (Cambridge Lisp `lessp`.)

Integer arithmetic in Lisp is often unrestricted in size: this means we can compute 2^{100} and see all of the 31 digits of the answer. These integers are given the colourful name of `bignums`, and are often internally implemented as vectors of "digits," where the digits are often base 10^8, or $2^3$1, say, something that will fit nicely into a single word. In a good system the cross-over from native small integers to bignums is invisible: we may mix them in any fashion. Amongst the implementations we have currently, KCL, Cambridge Lisp, Scheme and Oaklisp support bignums.

KCL claims

```
>(expt 2 1000)
10715086071862673209484250490600018105614048117055336074437503883703510511249361
22493198378815695858127594672917553146825187145285692314043598457757469857480393
45677748242309854210746050623711418779541821530464749835819412673987675591655439
4607706291457119647768654216766042983165262438683720566806937 6
```

There are also a wide range of trigonometric and logarithmic functions.

2.4.2 List stuff

`(null x)` Returns `t` if `x` is the empty list, otherwise `()`.

`(atom x)` Returns `t` if `x` is an atom, otherwise `()`.

`(consp x)` Returns `t` of `x` is a pair, otherwise `()`

`(cons x y)` Return the pair `(x . y)`.

`(car x)` Return the `car` of the list `x`.

`(cdr x)` Return the `cdr` of the list `x`.

`(list x1 x2 ...)` Make the list `(x1 x2 ...)`.

`(length x)` The length of the list `x`.

`(reverse x)` The reverse of the top level of the list `x`.

`(last x)` Return the last element of the list `x`

`(append x1 x2)` the list `x2` is appended to the end of list `x1`.

The use of a “p” at the end of some function names is also a little historical. The “p” is meant to imply *predicate*, i.e., something that returns true or false. However, its use is somewhat erratic: the names in Scheme are slightly different, such as `null?` for `null`, `atom?` for `atom`, and so on.

Exercise: what is the difference between the result of `(append '(1 2 3) '(a b c))` and the result of `(cons '(1 2 3) '(a b c))`?

There are also functions named `caar`, `cadr`, `cdar`, and so on, up to (typically) a depth of `c....r`, where each dot it either a `a` or `d`. These are shorthand for the repeated application of the appropriate `car` or `cdr` functions: `(cdar x)` is equivalent to `(cdr (car x))`.

2.4.3 Miscellany

`(quote x)` Return the *unevaluated* `x`. As we have said before, `quote` is not really a function.

`(print x)` Return (the value of) `x`. There is a side-effect in that the value of `x` is printed.

`(newline)` Return `()`. A side-effect is to print a newline. (Common Lisp, Cambridge Lisp: `terpri`.)

`(mapcar fun x)` Return the list which the result of applying the function `fun` to each element of `x` in turn. (Cambridge Lisp: the order of arguments differs, `(mapcar x fun)`)

Functions are first-class object in Lisp, `mapcar` is a good example of a function that has an argument that is a function. Later we will see how to create and return functions as values. The EuLisp call

```
(mapcar atom '(cat sat (on the) mat))
```

returns `(t t () t)`.

2.4.4 Logical Connectives

The “functions” `or` and `and` are two more exceptions to the “always evaluate” rule. Their semantics are forced upon them by the need to conform with other languages. For example, in C, the expression `e1 || e2 || e3` is evaluated by first computing `e1`. If this is a true, then return true, otherwise compute `e2`, and so on. So, the function application `(or e1 e2 e3)` progresses by evaluating `e1` alone, deciding if the value of the `or` is determined yet (which it is if `e1` evaluates to something non-`()`), and only then evaluating `e2`.

`(or x1 x2 ...)` Evaluate `x1`, `x2`, ...in left to right order. Whenever any value is non-`()` (i.e., true) return this immediately as the result of the `or`. If all are `nil`, return `()`.

`(and x1 x2 ...)` Evaluate `x1`, `x2`, ...in left to right order. Whenever any value is `()` (i.e., false) return `()` immediately. If all are non-`()`, return the last value as the result of the `and`.

`(not x)` If `x` is `()` (false), return `t`. Otherwise return `()`. This function is identical to `null`.

2.5 Defining your own

Just as expected, we can define our own functions in Lisp. These have just the same status in the eyes of the interpreter as any other built-in function; indeed we can often re-define any of the built-in functions to be whatever we wish (but it is unwise to do so—doing this may have unexpected consequences elsewhere in other functions that you might not realise are dependent on the function you are hacking).

We define a function as follows

```
(defun sumsqs (n m) (+ (* n n) (* m m)))
```

The "function" `defun` is almost the last of the exceptions to the evaluation rule. It takes the symbol `sumsqs` and makes it the name of a function of two arguments. The arguments have local names `n` and `m` in the body of the function, which computes the sum of the squares of the arguments. (This exception is a notational convenience: it would be quite possible to have `(defun 'sumsqs '(n m) '(+ (* n n) (* m m)))`, but the quoting of all the arguments is irritating.)

The symbol `sumsqs` is the value of the `defun`. The Cambridge Lisp variant is

```
(de sumsqs (n m) (plus (times n n) (times m m)))
```

i.e., the function defining function is named `de`. In Scheme we have the parenthesis in slightly different places:

```
(define (sumsqs n m) (+ (* n n) (* m m)))
```

and use "`define`". This makes the definition more reminiscent of the use of the function (in fact the truth is deeper than this).

In general, we have

```
(defun name (arg1 arg2 ...) expr1 expr2 ... exprn)
```

which defines a function with name `name`, with arguments `arg1`, `arg2`, ..., that when called evaluates `expr1` to `exprn` in turn, and the value of `exprn` is the value returned by the call.

To define a function of zero arguments is simple:

```
(defun foo () (print "hello") 23)
```

i.e., the arglist is `()`. Then `(foo)` is the function call. Note that this is different from just "`foo`" where we are given the value of the symbol `foo` (which may be a function), *not* the result of evaluating the function call.

To define a function that takes no arguments is even simpler:

```
(defun foo () (print "hello"))
```

i.e., the argument list is `()`. Then `(foo)` is the function call. Note that this is different from just `foo`, which gives the value of the symbol `foo` (which may be the function just defined).

Once we have defined a function, we can use it immediately:

```
(sumsq (+ 1 2) (sumsq 3 4))
```

is 634. We can use it in new function definitions

```
(defun sqs4 (a b c d) (+ (sumsqs a b) (sumsqs c d)))
```

and `sqs4` has the expected properties, e.g., `(sqs4 1 2 3 4)` is 30.

However, and this is one of the important features of Lisp, if we now *re-define* `sumsqs` by, say,

```
(defun sumsqs (u v) (- (* u u) (* v v)))
```

now every reference to `sumsqs`, new or old, will refer to the new function definition. Now `(sqs4 1 2 3 4)` is -10.

This is achieved by functions being referred to by *name*. The functional "value" of a symbol is the body of the function that we want to evaluate. So when we execute `sqs4`, it refers to a function named `sumsqs`, and to evaluate that, it looks up the current functional value associated with the symbol, and executes the code it finds there.

We have put "value" above in quotes: this is one of the major points of divergence in the Lisp family tree. Early in the development of Lisp there was a split in the decision of where to keep the function that was associated with a symbol. One way was to treat a function just as any other object in the system. The value of a symbol can be a list, or an atom, or whatever, or it can be a function. Lisps that choose this view of the world are called *single valued*, or Lisp-1s.

EuLisp, Scheme and Cambridge Lisp are Lisp-1s. If you have defined `sqs4` as above, and then simply type

```
sqs4
```

at the interpreter, the interpreter sees a symbol, and just as for any other symbol it retrieves the value, which in EuLisp is

```
#<interpreted-function: (lambda (n m) (+ (* n n) (* m m))) @ standard>
```

This in the form that (interpreted) functions take. They are structures, with the special marker `lambda`, a list of arguments, and the procedure body. The name `lambda` is derived from the λ-calculus, where the notation $\lambda\ a\ b\ c\ d.\mathrm{sumsqs}(a, b) + \mathrm{sumsqs}(c, d)$ might be employed. Notice that this function is anonymous: there is no name (like `sqs4`) attached to it. This is quite general—an object in Lisp is independent of the particular symbol of which it happens to be a value. You wouldn't expect the number 2 to be required to note that it was the value of the symbol `grobbit`.

On the other hand, in the other branch of the family tree, it was decided that functions and other values were completely different entities, and should be kept in different places—this, allegedly, was to improve speed (at the expense of semantic uniformity). These became the Lisp-2s, or the *two-valued* Lisps, so named as every symbol now had two values associated with it, namely the normal value it had as a symbol, plus the functional value, which can be completely unrelated. These are stored in the so-called *value cell*, and the *function cell*.

Common Lisp is a 2-Lisp. For the function `sqs4` defined above, if we were to try

```
sqs4
```

we would likely get

```
Error: The variable SQS4 is unbound.
```

This is because the interpreter looks in the value cell, and finds nothing there. If we want the *function* associated with the symbol we have to use

```
(symbol-function 'sumsqs)
```

when we get

```
(LAMBDA-BLOCK SUMSQS (N M) (+ (* N N) (* M M)))
```

There is a shorthand for `symbol-function`, the hash-quote: `#'`. This is used in the same way as `'`, but the effect is completely different. So `#'sqs4` is the lambda-block expression above.

In Common Lisp, if you have a function contained in a value cell (maybe this function was the return value of another function), to apply this function you must use `funcall`, or `apply`. This awkwardness is due to the semantic hiccough caused by the 2-cell property.

Suppose the symbol `foo` has a function as its value. In a 2-Lisp the application

```
(funcall foo 1 2 3)
```

will take the function that is the value of `foo` (i.e., is in the value cell of `foo`), and apply it to the arguments 1, 2, and 3. The function `funcall` is generally not present in a 1-Lisp, as the equivalent function application would simply be `(foo 1 2 3)`.

The function `apply` is similar, but requires the arguments as a single list:

```
(apply foo '(1 2 3))
```

This function is usually present in 1-Lisps, too.

With a 2-Lisp the symbol `foo` in `(foo bar baz)` is treated differently from `bar` and `baz`. The latter two are evaluated by inspecting their value cells; the former by looking at the function cell. In a 1-Lisp, all three symbols have just the same import. The function to apply is in a value cell, just as are the values to which want to apply the function. In fact, most 1-Lisps allow the object in the first place to be an arbitrary expression, just as the arguments can be arbitrary expressions:

```
((if (> x y) sin cos) 42)
```

will compute sin or cos of 42 according to the relative sizes of `x` and `y`. A 2-Lisp does not have this simplicity of expression, it would have to be

```
(funcall (if (> x y) #'sin #'cos) 42))
```

Only a symbol that names a function (or a selection of special lists) may appear in the functional position of a list that is being evaluated.

The example of `mapcar` we gave above is for a 1-Lisp. If we were to try the expression `(mapcar atom '(cat sat (on the) mat))` on a 2-Lisp the result will most probably be an error: the value cell of the symbol `atom` is being inspected. What we need is

```
(mapcar #'atom '(cat sat (on the) mat))
```

as this ensures the *function* `atom` is used.

This separation of 1-Lisps and 2-Lisps has become a religious war over the decades, each side firmly dug-in. Examples of 1-Lisps are Cambridge Lisp, EuLisp, the IBM Lisps, and Scheme. The 2-Lisps are the Common Lisps, and the Common Lisp-lookalikes; also PSL. Standard Lisp avoided the issue by not specifying which way an implementation should go: a Standard Lisp programmer must be careful not to use constructs that assume either!

2.6 If...

Having discovered how to define functions, we must now cover the question of program control: this means the introduction of `if`.

Lisp does include `if`, but this is an instance of a more general structure, named `cond` (for conditional).

The form of a `cond` is as follows:

```
(cond (test1 expr1)
      (test2 expr2)
      ...
      (testn exprn))
```

This is a multi-way if statement. The function `cond`, like `defun`, does not evaluate its arguments (again, this is a notational convenience). When evaluated, it first evaluates `test1`. If this is true (i.e, non-`()`) the expression `expr1` is evaluated, and the result of this is the value of the `cond`.

If the value of `test1` was false (i.e., `()`), the expression `test2` is evaluated. If this is true, `expr2` is evaluated, and is the value of the `cond`. Otherwise we continue down the list of tests until we find one that evaluates to true. If it happens that `all` of the tests return `()`, the value of the `cond` is `()`.

An example:

```
(cond ((= n 0) "zero")
      ((= n 1) "one")
      ((= n 2) "two")
      (t "many"))
```

This form returns the strings `"zero"`, `"one"`, or `"two"` according to whether `n` is 0, 1, or 2. If `n` is non of these, the `cond` gets to the `t` test, which evaluates to `t`, and returns the string `"many"`. This is a typical use of `cond`, with a "catch-all" clause at the end signalled by `t` which ensures the last clause is evaluated as a default. (Scheme users have the symbol "`else`" for their convenience.)

The tests and the expressions can be arbitrary Lisp forms. In fact, the `cond` is still more general:

```
(cond (test1 expr1a expr1b ... expr1k)
      ...
      (testn exprna exprnb ... exprnl))
```

there can be many expressions after each test. When a clause is selected, the `expr`s are evaluated in order left to right, and the value of the last is the value of the clause and the whole `cond`.

The form `if` is also provided in EuLisp and Common Lisp. This looks like

```
(if test trueclause falseclause)
```

which is more akin to the if-statements in other languages. The `test` is evaluated; if it returns true, then `trueclause` is evaluated and is the value of the `if`. Otherwise the `falseclause` is evaluated and is the value of the `if`.

The test expressions in `cond` and `if` can be arbitrary expressions: in general they will be combinations of function calls and uses of `and`, `or`, and `not`.

```
(cond ((or (> x 10)
```

```
        (< x 0))
    (some-chance))
   ((cond ((= y 0) t)
          ((= z 5) (< y 5))
          (t nil))
    (possibly-ok))
   (t (completely-confused)))
```

2.7 More Function Definition

We know how to define function of 0, 1, 2, and more arguments, but Lisp is more flexible than this, and allows us to define functions like `list` that take an *arbitrary* number of arguments. These are the so-called *n-ary* functions. Consider the following EuLisp definition of a n-ary function:

```
(defun doubleit l
   (if (atom l) ()
       (cons (* 2 (car l))
             (apply doubleit (cdr l)))))
```

This takes an arbitrary number of arguments which are numbers and returns a list with those values doubled. Thus `(doubleit 1 2 3)` returns `(2 4 6)`; `(doubleit 23)` returns `(46)`. The first point to note is the argument list `l`. This is in contrast to the function of *one* argument that would be defined by `(defun doubleit (l) ...)`.

The value of `l` within `doubleit` is a list of the arguments that were passed to the function: for `(doubleit 1 2 3)` `l` gets the list `(1 2 3)`. In `(doubleit 23)`, `l` is `(1)`, for just `(doubleit)`, `l` is `()`, and so on.

The function definition works like this: first check that we were given some arguments using `atom` on `l`. If none, return the empty list. Otherwise we have a list of things to be doubled, so we double the first one and `cons` is on the the recursively doubled rest of the list. Notice that we use `apply` here for the recursive call. This is because if we tried `(doubleit (cdr l))` we are passing just one argument (the list `(cdr l)`) to `doubleit`, and this is not what we want. We use `apply` to "spread" the arguments to `doubleit`. (Recall `(apply foo '(a b))` is equivalent to `(foo a b)`, so `(apply doubleit (cdr l))` is equivalent to calling `doubleit` on the spread-out elements of `(cdr l)`.)

Here is an alternative definition of `doubleit`:

```
(defun doubleit l
   (doubleit-aux l))

(defun doubleit-aux (l)
   (if (atom l) ()
       (cons (* 2 (car l))
             (doubleit-aux (cdr l)))))
```

We avoid the use of `apply` here by the use of a `helper` or `auxiliary` function to do the recursion. This function, `doubleit-aux`, takes just one argument, which is the list of the arguments that were passed to `doubleit`.

Study these two definitions of `doubleit` well, until you are convinced you understand them.

2.8 The "Program" "Feature"

Sometimes we have a place where we require more than one Lisp function to do what we want, where the syntax allows only one. For example, in the `if` form we are allowed exactly one expression in the `trueclause` position (to allow us to distinguish it from the `falseclause` position). To circumvent this difficulty, there is a form, `progn` which guarantees to evaluate its arguments in left to right order:

```
(progn expr1 expr2 ... exprn)
```

Thus, to evaluate a `progn`, we evaluate `expr1` to `exprn` in that order, and the value of the `progn` is the value of the last expression. This is akin to the construct in C of allowing a block of statements in curly brackets after, say, an `if`.

The name "progn" is an abbreviation for "*prog*ram of *n*o arguments," piece of history.

Thus we may write

```
(if (< amount 0)
    (progn (print "attempt to subtract from account")
           (call-error 42))
    (add-to-account amount))
```

which prints a message and calls an error routine if `amount` is negative, but otherwise adds `amount` into an account.

(Common Lisp has several other `progn`-type constructs, including `(prog1 expr1 expr2 ... exprn)`, which evaluates `expr1` to `exprn` and then returns the value of `expr1` as the value of the `prog1`. Also, amazingly, there is a `prog2` which does the above, but returning the value of `expr2`.)

Quite often we would like some extra local variables when we are evaluating a piece of code, maybe to modularise our ideas, or prevent re-computation of a difficult value. For this purpose we have `let`.

```
(let ((var1 val1)
      (var2 val2)
      ...
      (varn valn))
     expr1
     expr2
     ...
     exprn)
```

This says, "evaluate the expressions `val1` to `valn`, and let the symbols `var1` to `varn` have those values for the extent of the `let`. Now evaluate `expr1` to `exprn` in that order, and return the value of `exprn` as the value of the `let`. Outside of the `let`, the symbols revert to their old values, if they had one, or become undefined if they didn't. The `vals` may be arbitrary Lisp expressions, as may be the `exprs`. This is like `progn`, but with the declaration of local variables.

In Cambridge Lisp there is an order construct, named `prog` (this is also present in Common Lisp). The form of `prog` is

```
(prog ((var1 val1)
       (var2 val2)
       ...
       (varn valn))
```

```
      expr1
      expr2
      ...
      exprn)
```

This is much the same as `let`, but the value of the `prog` is `()` unless somewhere in the body of the `prog` there is a `return` statement. A `return` is simply

```
(return expr)
```

and has the effect of immediately exiting the smallest closing `prog` without evaluation of the following `expri`, and having the `prog` return the value of the `expr`.

So

```
(prog ((n 1) (m 2))
      (return (plus n m))
      (print "hi"))
```

returns the value 3, and does not print "hi".

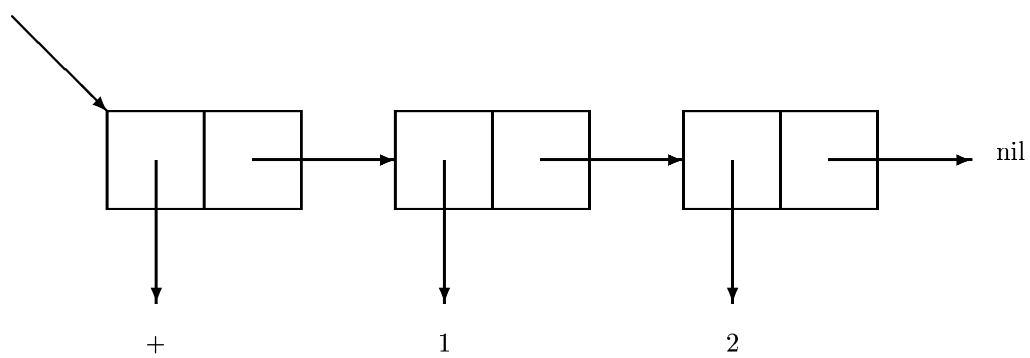
2.9 Were you created equal?

We have used `=` on numbers above: in fact, `=` is a function that is only defined on numbers. If we want to test equality of general Lisp objects we have to think harder: what do we meant by equality? To resolve this we must dig a little deeper into implementation.

2.9.1 `Eq`

All objects in Lisp are represented as pointers. There are Lisps that don't adhere to this to the letter, but to think of things in this manner is instructive. A symbol is a pointer to some store somewhere that contains the information associated with that symbol, such as its name and value. A list (or, more precisely, a pair) is a pointer to a pair of words that contain the `car` and the `cdr`. We know the `car` and `cdr` will fit into exactly two words, as their contents are pointers to other Lisp objects. A string is a pointer to some store that contains the characters in the string. And so on.

For example, the list `(+ 2 3)`:



As we recall, a list is actually a linked list of pairs, terminated by `nil`. The function `eq` compares two pointers, and returns `t` when they have the same value, i.e., refer to the same location in memory.

When the interpreter reads a symbol, it checks whether it has already seen that symbol; if so the reader returns the (pointer to) the previous symbol. If not, the symbol is stored in an internal table (a process called *interning*), and a new reference returned. This ensures all symbols that look the same are the same (. . . , well the truth is more complicated). Also we get

```
(eq 'foo 'foo)
```

it `t` for any symbol.

On the other hand, the function `cons` always makes a new pair, which is consequently not `eq` to any other current Lisp object. Thus

```
(eq (cons 1 2) (cons 1 2))
```

is always `()`, even though the objects produced look identical, viz., `(1 . 2)`. But, now,

```
(let ((cc (cons 1 2)))
     (eq cc cc))
```

is `t`, as the value of the two `cc`s are the same pointer.

Testing for `eq`ality on integers is a little variable. Often, a Lisp implementation will treat small integers specially, where "small" typically means "will fit comfortably in a word." Such implementations could use native machine arithmetic for such integers. Bignums are usually implemented using vectors of integers: every time a new bignum is needed, a new vector is allocated, even if somewhere deep in the system that number already exists—of course it is impractical to compare yourself against all the possibly thousands of other bignums already allocated, it is much easier just to make a new vector. The consequence of this is that two bignums that are numerically equal may not be `eq`. The function `=` is provided for this, and for other numeric testing.

Other implementations allocate a few "common" integers in advance, and treat them the symbols above. For example, KCL pre-allocates the integers from −1024 to 1023. This has the effect that `(eq 1023 1023)` is `t`, but `(eq 1024 1024)` is `nil`.

The point of `eq` is that it is a very fast test. If two objects are `eq`, then they are identical. If they are not `eq`, we must work a little harder to discover whether they are "the same."

2.9.2 `Equal`

But what do we mean by "the same"? The function `equal` has a plausible definition: if the objects are `eq`, return `t`. If they are both numbers return `t` if they are numerically equal. If they are both strings, return `t` if they contain the same characters. If they are both pairs, return `t` of both the `car`s are `equal`, and both the `cdr`s are `equal`.

Now

```
(equal (cons 'a 'b) (cons 'a 'b))
```

is `t`, as the `car`s of the `cons`es are `eq`, as are the `cdr`s.

The function `equal` is a general, tree descending equality. Common Lisp supplies many other variants of the equality operator, the most useful being `eql`, which is a form of `eq` that is also guaranteed to compare numbers correctly.

3 Programming

3.1 Delegation

Much of Lisp programming is done by recursion. This is a natural development from the central datatype, the list. Typically we do not know, and do not need to know how long a list is: we find a common need is to combine the elements of a list in some fashion, or to make a new list from applying a function to each member of an old list in turn.

Both can be solved neatly and efficiently by recursion. For example, suppose we want to compute the sum of a list of pairs of integers. So `(summit '((1 . 2) (3 . 4) (5 . 6)))` is 21. One Lisp function to do that is as follows:

```
(defun summit (l)
  (if (null l) 0
      (+ (caar l) (cdar l) (summit (cdr l)))))
```

This has the natural interpretation that the sum of the list is the what you get by adding the sum of the first pair to the sum of the rest of the list.

Similarly, to create a list of products of the pairs we want the list created by `cons`ing the product of the first pair on to the list of products of the rest. This translates immediately into

```
(defun prods (l)
  (if (null l) ()
      (cons (* (caar l) (cdar l))
            (prods (cdr l)))))
```

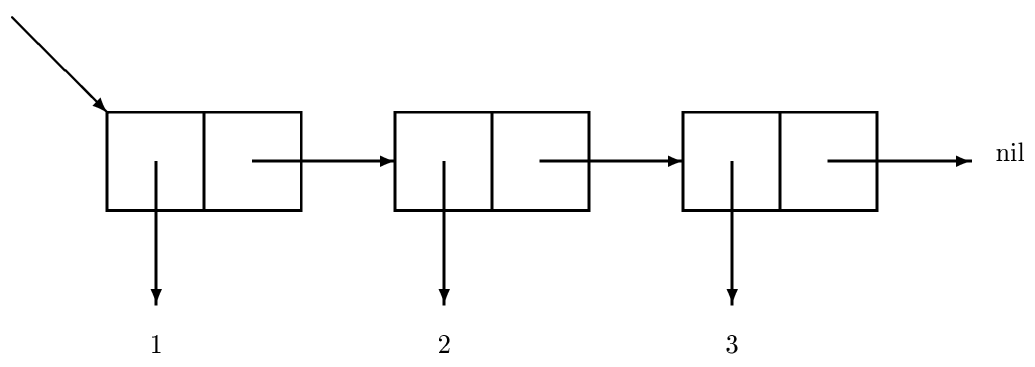
Only the end of the recursion, the base case, needs to be thought out carefully.

3.2 Tail Recursion

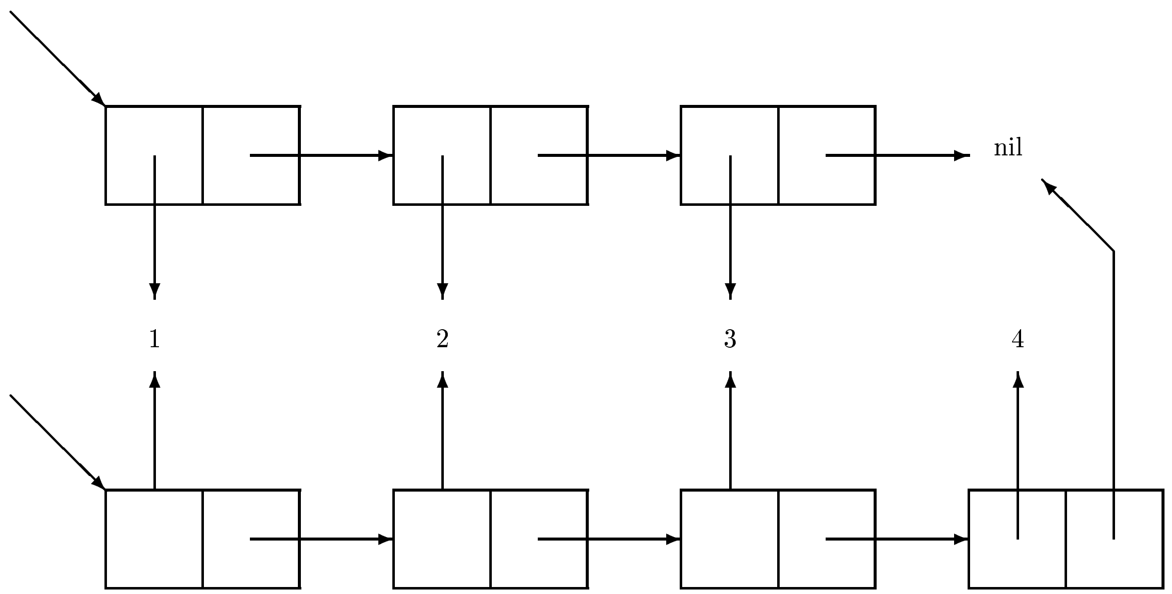
Consider this function that reverses a list

```
(defun rev1 (l)
  (if (null l) ()
      (append (rev1 (cdr l)) (list (car l)))))
```

The function `append` works by making a copy of the first argument and consing it onto the second. The reason for this is clear if we think of the list as a linked chain of pairs. The list `(1 2 3)` looks like



To append the list (4) we must create a new list



while retaining the old list in case other objects have a hold on it (e.g., it is a sublist of another list). Thus each call of `append` uses a number of new `cons` cells equal to the length of the list being appended to.

Now, in the recursive definition of reverse above, we call `append` once for each element in the list: this means we use $O(n^2)$ new `cons` cells for a list of length n. Clearly this is wasteful, as each partial result is discarded almost immediately after it has been created.

A much better implementation of reverse uses *tail recursion*.

```
(defun rev2 (l)
  (revaux l nil))

(defun revaux (l sofar)
  (if (null l) sofar
      (revaux (cdr l)
              (cons (car l) sofar))))
```

Tail recursion is where the result of the function is simply a call to the function itself, probably with altered arguments. An intelligent compiler will spot this, and replace a recursive call (using a stack frame) by a simple re-assignment of the argument variables, and a jump to the top of the function. This allows us to have arbitrarily deep recursions, as no stack is used. Here are two definitions of factorial, the latter being tail recursive:

```
(defun fact1 (n)
  (if (< n 2) 1
      (* n (fact1 (- n 1)))))

(defun fact2 (n)
  (factaux n 1))

(defun factaux (n sofar)
  (if (< n 2) sofar
      (factaux (- n 1) (* n sofar)))))
```

(We note that tail recursion is applicable in any language that allows recursion, and Lisp has being using it since the mid 60's. However, it is only in the late 80's did other language compiler writers notice that they could do it too.)

In `rev2`, we use tail recursion to good effect. The reverse of `l` is accumulated in `sofar`, and we use exactly one `cons` for each element of the list. A trace of `revaux` shows `l` getting shorter and shorter, while `sofar` gets longer and longer.

Many functions benefit from rewriting in this manner.

3.3 A load of Garbage

Now is a good time to mention garbage collection. We said above that each invocation of `append` in `rev1` creates a new copy of the partial reverse, and the old copy is thrown away. The old list (which is a linked list of cons pairs, remember) becomes inaccessible, as we no longer have a pointer to its head. The cells simply lie about uselessly, cluttering up memory. And, of course, we soon will run out of memory if nothing further is done about this. The dead cells are called *garbage*.

Lisp is a tidy language, and likes to recycle and reuse idle cells, and to do this it uses a sub-program called a *garbage collector*. Garbage collectors come in many shapes and sizes, but they all follow the same general principles: when the Lisp program runs out of memory, it halts and invokes the GC. The GC visits all of the cells that are reachable from active variables, locations on the stack and so on—every cell that is still being used by the program—and compacts them together into a contiguous block. Everything else is dead, and can therefore by reused by the Lisp program. When the GC is finished it hands control back to the calling program, which gets its bit of memory, and continues as if nothing had happened.

The visible effect of this is that the program pauses for a while, and then continues after a few seconds. In practice, this is no problem, until we try a program that uses almost all the available memory, when the space reclaimed gets smaller and smaller. When this happens, we spend more and more time GCing, and less and less time doing useful work. Occasionally we get a *store jam*, where the whole of memory is used: such a program will not finish in *any* language!

To avoid such potentially embarrassing pauses, techniques have been developed such as the use of reference counts: whenever a cell is used a count in it is increased by 1. When a reference to a cell is dropped, the Lisp decrements the count, and if the count is now zero, it knows that this cell is unused, and can put it on the free-list for later reuse. This spreads the time of GC over each individual access to a cell. However, the overhead is relatively high, and there are problems with circular lists.

A current topic of research is *concurrent* GC, where you have a separate processor dedicated to scavenging dead cells: the problem to avoid is memory contention when the running program wants a cell that the GC is inspecting.

(Often, a GC is written in Lisp—the ideal language for traversing tree structures!)

3.4 Making Associations

A common want in a programming application is some method of associating one value with another. A typical example is the hash table: we have a key, and the table gives us a value from that key. The Common Lisp and EuLisp standards table have hash tables built in, but there is another technique that can be used in all Lisps, the *association list*.

An association list, put simply, is a list of pairs. The keys are the cars of the pairs, and the values are the cdrs of the pairs. For example, if `nametono` has the value

```
( (zero . 0) (one . 1) (two . 2) (three . 3) (four . 4)
```

```
  (five . 5) (six . 6) (seven . 7) (eight . 8) (nine . 9) )
```

then this forms a useful way of getting from the name of a digit to the digit itself. There is a useful function defined to help us use association lists:

> `(assoc x l)` searches the association list `l` until it finds the first pair with car `equal` to `x`. If successful, it returns that pair; if not it returns `()`. (Common Lisp uses `eql`.)

Thus, for EuLisp, `(assoc 'four nametono)` is 4, but `(assoc 'ten nametono)` is `()`. (Bug alert: in the current implementation, you need to write `(assoc 'ten nametono equal)`, i.e., including the function to be used as a comparator. You may also have `(assoc 'ten nametono eq)`, and so on.)

Association lists are useful in mapping objects to objects. However, if the first object is a symbol, there is another concept we can use called the *property* list. A property list is (morally speaking) an association list that is attached to a symbol. This property list is entirely disjoint from the value and/or function cells: it is sometimes referred to as being in the *property* cell. (N.B., this somewhat belies the names 1-Lisp and 2-Lisp!)

We access the property list through the functions `(setter get)` and `get`, and `symbol-props` (Common Lisp: `symbol-plist`). The function call `(symbol-props 'x)` returns the current property list associated with the symbol `x`. Of course, `(symbol-props x)` will get the property list of the symbol which is the value of `x`, assuming that the value of `x` was a symbol. Trying to manipulate property lists of non-symbols will cause an error. (Bug alert: in the current implementation of EuLisp, you need `(import plists)` to load the relevant code before the above functions will work.)

(Aside on `setter`. There are many function in Lisp that access data structures in some way. Thus `car` gets the car of a pair, and `cdr` the cdr; later we will see `vector-ref` to access elements of a vector. For each accessor there is generally an associated updator function, i.e., a function that updates the cell referred to by the accessor. Thus `(setter vector-ref)` is a function that updates a vector element, and so on. The usage is `((setter vector-ref) vec index newval)`. Having a function as a value `(setter whatever)` is used just as a function.)

We use `(setter get)` to add to the property list:

```
((setter get) 'x 'one 1)
```

puts the value 1 under the key `one` on the property list of `x`. The key for property lists must be a symbol. Successively, `((setter get) 'x 'two 2)` and then

```
(symbol-props 'x)
```

will return `(two 2 one 1)` (this is not actually a kosher association list, but contains equivalent information). If, now we try `((setter get) 'x 'one 10)`, the property list is updated and becomes `(two 2 one 10)`.

The function `get` does an `assoc` on property lists:

```
(get 'x 'one)
```

is 10. If `get` does not find the key, it returns `()`. (This makes it difficult to distinguish between keys whose value is `()`, and keys which are not there.)

To remove a property, use `(remprop symbol key)`: this deletes the property with key `key` from the symbol.

Property lists are a kind of dual to association lists: an association list collects together a table for a concept, and uses a symbol (key) to index to a value. On the other hand, a plist spreads the table out amongst the symbols, and then uses the concept to index for the value. Symbolically, this is concept → (key → value) in contrast to key → (concept → value)

An example: we have the `nametono` alist above. The "concept" is a map from names to numbers, and we associate the map with the single symbol `nametono`. To get a number, we use the name as a key. For a plist, we might have executed

```
((setter get) 'one 'nametono 1)
((setter get) 'two 'nametono 2)
...
((setter get) 'nine 'nametono 9)
```

where the concept is now the key into the plist. The calls

```
(assoc 'two nametono)
```

and

```
(get 'two 'nametono)
```

have the same result.

It is a coincidence that the symbol `nametono` happened to occur both as the name of a variable holding the alist, and as a symbol in its own right which happens to be used as a key off some plists.

Given we have the two mechanisms, which is the better? As usual, it all depends on the situation. If you wish to keep a small alist of ideas in one place, and be able to manipulate the whole alist, then an association list is better. On the other hand, a large alist can be slow to search, while it is rare for a plist to be large, making it very fast to search. However, the table of associations is spread over a possibly indeterminate range of symbols, making it difficult to use as a single entity.

Common Lisp usage of property lists is at variance with the above. To look at the plist, use `symbol-plist` in place of `symbol-props`; the accessors and removers are the same, `get` and `remprop`. To put a new property, use the construct

```
(setf (get 'x 'one) 10)
```

The function `setf` is much more general than this, but for now regard this as an idiom:

```
(setf (get symbol key) value)
```

where EuLisp has `((setter get) symbol key value)`.

Cambridge Lisp has one further way of associating a value with a symbol: `flag`. This function uses the plist to indicate values in the following way:

```
(flag '(x y) 'used)
```

causes the symbols `x` and `y` to be *flagged* with the symbol `used`. The first argument can be a list of symbols of any positive length: each symbol in the list is flagged.

Now, if we use

```
(flagp 'x 'used)
```

we get `t`, but `(flagp 'z 'used)` is `nil`. To delete a flag, use `(remflag '(x) 'used)`. Now `(flagp 'x 'used)` is `nil`.

The value is stored on the property list as an atom (in contrast to the other properties, which are pairs). If we try

```
((setter get) 'num 'one 1)
((setter get) 'num 'two 2)
(flag '(num) 'numbered)
```

then the value of `(plist 'num)` is `(numbered (two . 2) (one . 1))`.

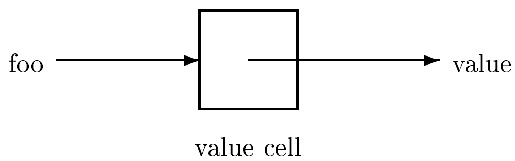
Notice that the Common Lisp usage cannot be extended in this manner due to the way property lists are stored.

3.5 Binding and Assignment

Thus far in this course we have learnt of the various ways a value can be associated with a symbol: value cell, function cell, property list. However, we have said only a little about how to create these associations.

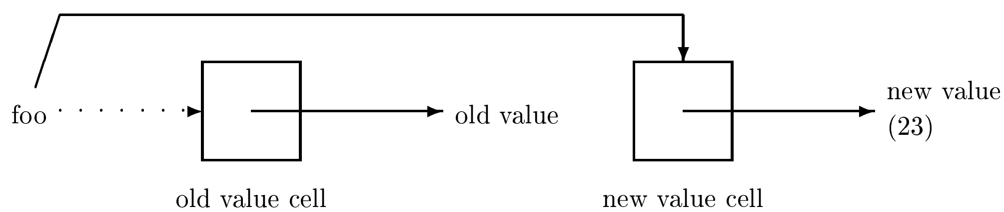
To update the function cell of a symbol, we use `defun` or `de`. To change the property list, we have `put`. But the process of altering the value cell is more interesting.

We have used `prog` or `let` to make local bindings of values to symbols: think of a symbol



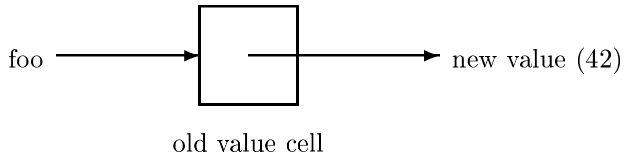
where we have the value cell of the symbol `foo` pointing to some Lisp object.

If we use `(let ((foo 23)) ...)` then in the body of the `prog` we have a new association of symbol to value cell



but the old association still exists. The process of making a new cell–new value association is called *binding*. It is much more useful than the usual (in other languages) method of assignment to a variable in that no information is destroyed. Thus when we leave the scope of the `let`, the old association can be resumed, i.e., the symbol regains its old value.

Lisp has a function that allows us to subvert the binding mechanism: it is the assignment function `set`. If we do `(set 'foo 42)` to the original binding this happens



and now the value of `foo` is 42 (in the scope of this binding). This is akin to the usual assignment in other languages. In fact, there is a special non-evaluating "function" `setq` (for "set quote") so that `(setq foo 123)` is equivalent to `(set 'foo 123)`. The function `set` is a deviation from the semantic purity of Lisp, and some people try to avoid its use at all costs—though others think that the costs are too large, involving cortorted use of local functions. In fact, Scheme and EuLisp deny the use of `set` completely, but allow `setq` (Scheme: `set!`. The exclamation mark indicates we are doing a destructive operation—overwriting the old value).

The distinction between assignment and binding is very important, particularly when we come to the question of scope.

3.6 The Scope of a Binding

Consider the program

```
(setq n 23)

(defun foo () (print n))

(defun bar () (let ((n 42)) (print n) (foo))
```

so `n` has the global value 23.

The question of scope addresses to resolution of the free variable `n` in `foo`. The instance of `n` in `bar` is clear: running `bar` will print 42. Also, running `foo` directly we expect to see the value 23. However, what about the invocation of `foo` from within `bar`?

There are two choices for `n` in `foo`. Either it refers to the global variable (with value 23), or the locally bound variable (with value 42). These two choices are called the *lexical* and `dynamic` scope.

The scope is lexical when we derive the meaning of a reference by its textual context. So, for lexical scope, the `n` in `foo` refers to the global variable with value 23, as it is the "nearest" textual binding of `n`.

On the other hand, we have dynamic scope when the reference is determined at run time from the latest binding that was executed. So, for dynamic scope, the `n` in `foo` refers to the local variable that was bound in `bar` to the value 42.

Both kinds of scoping are useful, and we would like some mechanism for employing both. In the past Lisps have been a little vague on this subject, but recently it has become clear that a distinction must be made. After all, we get very different program behaviour using the two schemes. Older Lisps tend to be dynamically scoped, and newer ones lexically: this is due to a shift in style and need as functional ideas grow in importance. (Functional programming requires a "one symbol, one value" concept in that we can determine from examination of the text of a program what reference a particular symbol means. Clearly this is lexical scoping.)

Cambridge Lisp is (roughly speaking) dynamically scoped as a default, whereas EuLisp and Common Lisp are lexically scoped as a default. In Common Lisp if we want a symbol to be dynamically scoped, we must declare it to be such.

```
(proclaim '(special n m))
```

declares n and m to be *special* variables (which is the CL parlance for dynamically scoped). In the presence of this declaration, n above would be taken to mean the *dynamic* version. Without such a declaration, n and m would be lexically scoped, and n reveals its *lexical* value. The use of declarations is a bit of a kludge, as the meaning of a symbol varies according to whether it is used before or after a declaration. This allows the possibility of having a symbol being lexical in some areas of a program, but dynamic in others. This can be nothing other than confusing.

EuLisp has a neat method for determining which scope to use for a symbol. If we require the lexical scope, simply use the symbol: thus "n" gives the lexical value. If we want the dynamic scope, we must explicitly ask for it,

```
(dynamic n)
```

is the dynamic value. Thus we can see explicitly which scope is required. There are other analogous functions, such as `dynamic-let` and `dynamic-setq` to rebind and re-assign the dynamic variable.

3.6.1 Examples of Lexical Scope

Consider the following Common Lisp program

```
(setq n 23)

(let ((n 0))
  (defun countme () (setq n (+ n 1)))
)
```

The function definition is contained within the `let`, so the free lexical variable n inside `countme` refers to the n from the `let`, not to the global definition.

When we use the function: `(countme)` returns 1, as the value of the function is the value of the `setq` which is 1. Typing "n", we are asking for the global value, which is still 23. However, we have re-assigned the n local to `countme`, so the next use `(countme)` returns 2. And so on. This function counts the number of times it is used. The value of (the global) n remains unaffected.

The local version of n is inaccessible to the rest of the world, providing us with a simple form of data-hiding.

Here is a more sophisticated example in EuLisp

```
(defun makeadder (n) (lambda (m) (+ m n)))

(setq add4 (makeadder 4))
```

Here the `lambda` expression is an anonymous function: for example `(lambda (x) (+ x 1))` is a function that adds 1 to its argument.

The call to `(makeadder 4)` returns a function in which n is lexically bound to 4. If, now, we try `(add4 5)`, the result is 9. But, further, `(setq add6 (makeadder 6))`, then `(+ (add4 3) (add6 10))` is 25. The instance of n in each `lambda` which is the result of calling `makeadder` is a separate binding (made at the time the `lambda` is created), and is unaffected by anyone else's use of the same name.

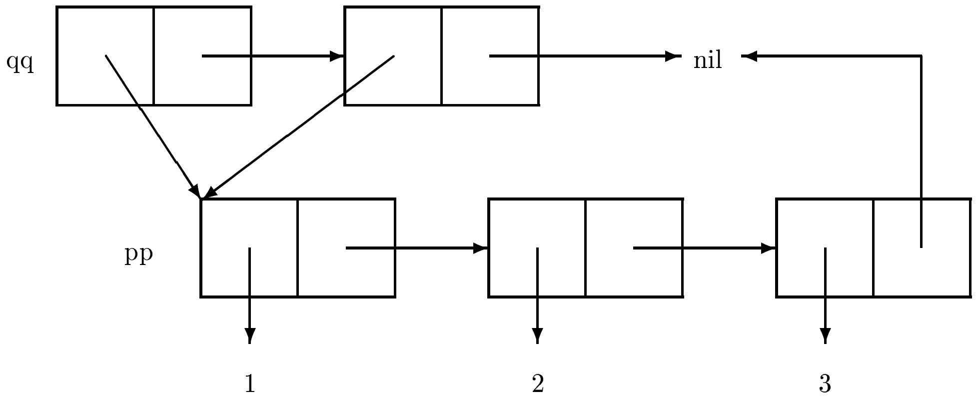
3.7 Structure Sharing

A list is a sequence of pointers (to pairs). In particular, there is no restriction on them being *distinct* pointers. Suppose

```
(setq pp (list 1 2 3))

(setq qq (list pp pp))
```

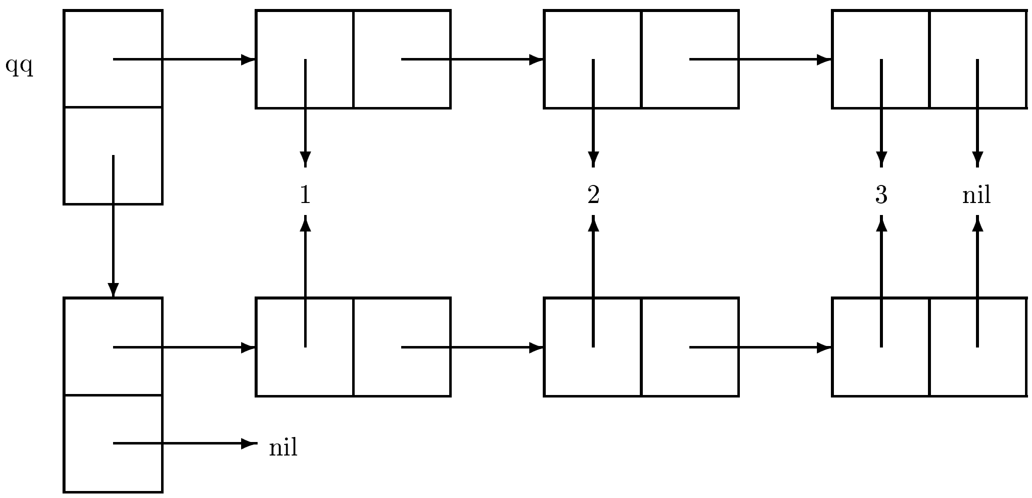
then `qq` has value `((1 2 3) (1 2 3))`:



Although the sublist `(1 2 3)` appears twice within `qq`, it is a single list of pairs. Contrast this with the alternative

```
(setq qq (list (list 1 2 3) (list 1 2 3)))
```

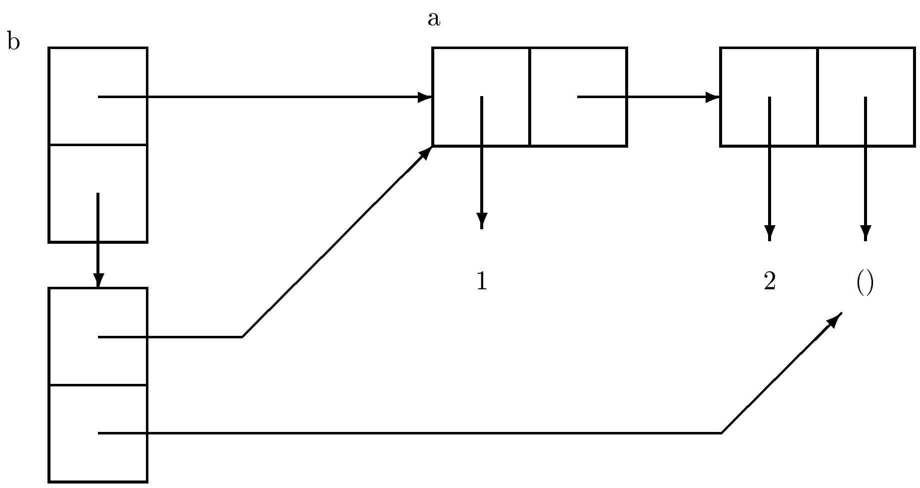
where the car of `qq` is a separate set of conses from the cdr:



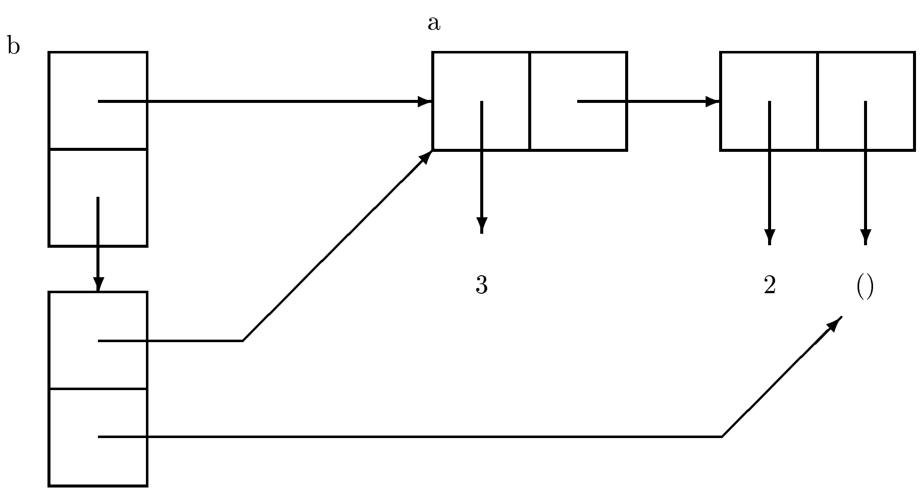
Consider the following:

```
(setq a (cons 1 2))
(setq b (list a a))
```

So `a` is `(1 . 2)`, and `b` is `((1 . 2) (1 . 2))`.



The function `(setter car)` destructively overwrites the car of a pair: after `((setter car) a 3)`, the value of `a` is now `(3 . 2)`. But now consider the value of `b`:



By overwriting the car of `a`, we have affected the value of `b`, it is now `((3 . 2) (3 . 2))`. Moreover, it has affected `b` in each pair that was a copy of `a`. In fact, as the diagram shows, they were not copies of the pair `(1 . 2)`, but actually multiple reference to the same pair.

(Compare the action of a list appending function that makes a copy of its first argument (as does `append`), against the action of an appending function that destructively overwrites the last cdr of the first argument (as does `nconc`). If `a` has the value `(1 2)`, what do you think the values of `(append a a)`, and `(nconc a a)` will be?)

These are simple examples of *structure sharing* in Lisp. Structure sharing allows us to express much larger objects than we might expect: if we merge all identical branches of a tree, we can save on the store we previously required. For a large example, this saving can be immense.

A practical case can be found in computer algebra. A polynomial can be represented in Lisp as a list: $ax^n + bx^{n-1} + \cdots$ might be

```
(((x . n) . a) ((x . n-1) . b) ...)
```

which is to say the car is the pair (leading power . leading coefficient), where the leading power is a pair (variable . degree). The cdr of the list is the *reductum* of the polynomial, i.e., the rest of the polynomial. An integer is represented by a Lisp integer.

Thus, $3x^2+4x+5$ becomes `(((x . 2) . 3) ((x . 1) . 4) . 5)`. We add polynomials using the function `addf`

```
(de addf (u v)
  (cond ((zerop u) v)
        ((zerop v) u)
        ((and (atom u) (atom v)) (+ u v))  % both integers
        ((atom u)
           (cons (car v) (addf (cdr v) u)))   % v not an integer
        ((atom v)
           (cons (car u) (addf (cdr u) v)))   % u not an integer
        ((> (cdaar u) (cdaar v))       % both not integers
           (cons (car u) (addf (cdr u) v)))
        ((> (cdaar v) (cdaar u))
           (cons (car v) (addf (cdr v) u)))
        ((zerop (plus (cdar u) (cdar v)))  % the leading coefficients
           (addf (cdr u) (cdr v)))
        (t (cons (cons (cons 'x (cdaar u)) % degree
                       (+ (cdar u) (cdar v)))    % coeff
                 (addf (cdr u) (cdr v))))))
```

This function looks complicated, but is quite simple if worked through carefully: to start with, the sum of anything and zero is that thing. If both are integers, their sum is easy. Otherwise one of u or v (or both) is not an integer. We use the property of polynomials that

$$(ax^n + bx^{n-1} + \cdots) + (rx^m + sx^{m-1} + \cdots) = ax^n + \big((bx^{n-1} + \cdots) + (rx^m + sx^{m-1} + \cdots)\big).$$

In particular, when $n < m$, we can add the two polynomials on the right by recursion.

This is what `addf` does. We look at the degrees of u and v: if the degree of one is larger than the other (taking care not to try to take the car of an integer, which has degree 0) we pull off its leading term, and stick it back on to the recursively computed sum of rest.

If both degrees are the same, we check that the leading coefficients do not cancel, and do a simple recursion using the sum of the reducta.

Try applying this function to x^3+x+1 and $2x^2$, viz., `(((x . 3) . 1) ((x . 1) . 1) . 1)` and `(((x . 2) . 2) . 0)`. We get a recursive call of `(((x . 1) . 1) . 1)` and `(((x . 2) . 2) . 0)`, then a call on `0` and `(((x . 1) . 1) . 1)`. This returns immediately with `(((x . 1) . 1) . 1)`, and notice that it returns the structure `(((x . 1) . 1) . 1)` directly, not a copy of it (i.e., the pointer to the structure that already exists). Crawling back up the stack, we cons on `((x . 2) . 2)`, and then `((x . 3) . 1)`, giving a final result

```
(((x . 3) . 1) ((x . 2) . 2) ((x . 1) . 1) . 1)
```

or $x^3 + 2x^2 + x + 1$ Now, the number of new cons cells it took to make this answer was exactly two. Every other cons in the result is one of the old ones, reused. Each of the `((x . 3) . 1)`, `((x . 2) . 2)`, and the final `(((x . 1) . 1) . 1)` are all shared with the input structure.

In a big computation, this sharing of structure is essential in the successful shoehorning of the answer into the available memory. Also, by reusing structure, we are not creating as many garbage cells as we might have at first supposed, thus saving us time in reduced garbage collection time, too.

4 Advanced

4.1 Macros

Macros are a relatively benign addition to C. We can define symbols and trivial functions that are textually replaced by their expansion in the body of code when we call the compiler. In Lisp, macros have status often equal to that of functions, and can have very subtle semantics.

Macros are used when we would like to name a short piece of code for the sake of abstraction: e.g., in `addf` above we could define `ldeg` to be the same as `cdaar`; this adds meaning to the program in that we see from the name in the text that we want the leading degree of a polynomial, not just a random part of a random structure. This also allows us an extra level of abstraction by keeping the implementational detail of polynomials hidden behind the name `ldeg`—we might want to change the representation of a polynomial, and all we need do is change the definition of `ldeg`.

We could define `ldeg` as a function, but we might not want the expense of an extra function call, but want the "`ldeg`" to be replaced by "`ldeg`" directly. More generally, a macro allows a Lisp expression to be arbitrarily manipulated before being evaluated. Lisp macros work at the Lisp expression level, not at the character level, as in C.

We define a macro like this

```
(defmacro ldeg (l) (list 'cdaar l))
```

(Cambridge Lisp: use `dm`. In Scheme there is no prevailing standard for macros.) The body of a macro is an expression that is evaluated to give the required macro expansion. When we use `ldeg`, e.g., `(ldeg '(((x . 3) . 2) . 1))`, the macro first expands to `(cdaar '(((x . 3) . 2) . 1))`, which has value `3`.

A simplistic way of thinking of macros is to regard them as functions whose bodies are evaluated twice. Firstly to do the expansion, secondly to compute the value of the expansion. However, the arguments of the macro are not expanded at all. If we use `(ldeg u)`, the macro expands to `(cdaar u)`, which, when evaluated will hand us the leading degree of the polynomial that is the value of `u`.

Define push:

```
(defmacro push (x l)
  (list 'setq l (list 'cons x l)))
```

which takes an object and a symbol whose value is a list, and pushes the object on to the front of the list. So when we use

```
(setq stack '(3 2 1))

(push 4 stack)
```

the macro expands to `(setq stack (cons 4 stack))`, which has the desired effect when it is evaluated.

Consider, however, the call `(push 4 '(1 2 3))`. We might hope to get the list `(4 1 2 3)`. But consider the macroexpansion:

```
(setq '(1 2 3) (cons 4 '(1 2 3)))
```

This will almost certainly flag an error: `setq` demands that its first argument is a symbol.

This kind of usage (where we cons up an expression to be executed) is so common that there is a shorthand provided, called the `backquote` (‘). We can’t define `push` as

```
(defmacro push (x l)
  '(setq l (cons x l)))
```

in the hope that the body evaluates to the right `setq` and `cons` combination, since, for this definition, `(push 4 stack)` will expand to `(setq l (cons x l))`. When we come to evaluate this expression, more likely than not, the symbols `l` and `x` are undefined, or at least have nothing to do with 4 and `stack`.

The backquote is like the forward quote, but allows escapes inside that are treated specially:

```
(defmacro push (x l)
  `(setq ,l (cons ,x ,l)))
```

Within a backquoted expression objects are quoted, except when they appear after a comma `,` operator. The comma is a sort of “anti-quote,” meaning that the object after the comma are *not* quoted. Now `(push 4 stack)` expands to `(setq stack (cons 4 stack))`, since the value of `x` is 4, and the value of `l` is `stack`.

So we might have defined `ldeg` by

```
(defmacro ldeg (p)
  `(cdaar ,p))
```

The backquote is a template that is filled in with the comma expressions.

What about

```
(defmacro def (fun arglist body)
  `(setq ,fun (lambda ,arglist ,body)))
```

This is a simplified (no implied `progn` in the body) form of `defun` for a 1-Lisp. If we try

```
(def foo (n) (+ n 1))
```

the macro expands to `(setq foo (lambda (n) (` n 1)))+, which makes the value of `foo` the specified function.

The ‘ is a piece of syntax, just like ’, and expands in the reader to the `backquote` function; the `,` likewise. It is important to note that commas are evaluated at expand time: so

```
`(1 ',x 2)
```

expands to `(1 (quote 123) 2)` if `x` has the value 123.

For more adventurous macro writing, there is also the splicing operator `,@`. Applied to a sublist inside a backquoted list this splices its argument into the superlist.

```
(setq x '(a b))

`(1 ,x 2 ,@x 3)
```

has value `(1 (a b) 2 a b 3)`.

As always, 1-Lisps and 2-Lisps do things differently. Typically a 1-Lisp saves a macro in the value cell (thus a symbol cannot simultaneously name both function and a macro), where a 2-Lisp keeps it elsewhere. Common Lisp requires the use of a macro-cell, so a single symbol can mean either a value, a function, or a macro, according to context. (What with property cells, and several other cells too, calling CL a "2-Lisp" is somewhat understated!)

4.1.1 Destructuring

A very powerful use of macros involves *destructuring*. This is a simple form of pattern matching of the argument list of the macro. For example, in Common Lisp, we can define

```
(defmacro foo ((a b) c) `(cons ,a ,b ,c))
```

and use `foo` like `(foo (1 2) 3)`, when `a` is bound to 1, `b` to 2, and `c` to 3. The arguments in the call are matched up against the arguments in the definition, and appropriate bindings are made.

The most useful instance of destructuring is in the following example:

```
(defmacro myif (test . cases)
  (cond ((= (length cases) 1)
         `(cond (,test ,(car cases))))
        (t
         `(cond (,test ,(car cases))
                (t     ,(cadr cases))))))
```

The first argument if `myif` is bound to `test`, and all the remaining arguments are bound in a single list to `cases`.

The macro expands into a two case `cond` or a one case `cond` according to the length of `cases`. We can test a macro using `macroexpand`:

```
(macroexpand '(myif 1 2 3))
```

results in `(cond (1 2) (t 3))`. (The local variable of `test` in the macro call is `1`, and the value of `cases` is `(2 3)`.) The call `(macroexpand '(myif 1 2))` expands to `(cond (1 2))`.

Another example

```
(defmacro mylet (bindings . body)
  (prog ((vars (mapcar #'car bindings))
         (vals (mapcar #'cadr bindings)))
    (return
      `((lambda ,vars ,@body) ,@vals))))
```

Cambridge Lisp supports (in the default system) a little destructuring, namely

```
(dm if args
  (prog ((test (car args))
         (cases (cdr args)))
    (return
```

```
      (cond ((equal (length cases) 1)
             `(cond (,test ,(car cases))))
            (t
             `(cond (,test ,(car cases))
                    (t     ,(cadr cases)))))))
```

where the entire argument list is bound to `args`.

4.2 Exotica

We now come so a miscellany of bits of Lisp, bits that did not fit elsewhere, and bits that are not "in the spirit of Lisp."

4.2.1 Gensym

Consider a three way if (Fortran users note!)

```
(arithmetic-if test neg-form zero-form pos-form)
```

where we do `pos-form` if the value of `test` is positive, `zero-form` if it is zero, and `neg-form` when it is negative.

Using macros, this is easy:

```
(defmacro arithmetic-if (test neg zero pos)
  `(let ((var ,test))
     (cond ((< var 0) ,neg)
           ((= var 0) ,zero)
           (t         ,pos))))
```

Here we have a local variable `var` to hold the value of test (we do not want to evaluate `test` more than once). This works fine until we try

```
(arithmetic-if (foo var) (+ var 1) var (- var 1))
```

This expands into

```
(let ((var (foo var)))
  (cond ((< var 0) (+ var 1))
        ((= var 0) var)
        (t         (- var 1))))
```

which almost certainly does not do what we want. The problem, of course, is the name clash. We need to define the macro in such a way that the symbol in the `let` is different from any in the expansion of the body. To do this, Lisp supplies the function `gensym`.

This returns a symbol guaranteed different from every other symbol currently in the system. Also it is not put into the system's table of symbols: the only way to get at this symbol is indirectly, like this:

```
(setq symb (gensym))

(set symb 23)
```

This sets the value of the gensym to be 23.

We can now write `arithmetic-if`:

```
(defmacro arithmetic-if (test neg zero pos)
  (let ((var (gensym)))
    `(let ((,var ,test))
       (cond ((< ,var 0) ,neg)
             ((= ,var 0) ,zero)
             (t          ,pos)))))
```

Now the example expands to

```
(let ((g1234 (foo var)))
  ((< g1234 0) (+ var 1))
  ((= g1234 0) var)
  (t           (- var 1))))
```

where `g1234` might be a printed representation of the gensym.

Gensyms are useful whenever we need a new symbol which must be different from any other we have already.

4.2.2 Vectors

Lisp has lists. For those who can't cope with a flexible, dynamic data structure, we also have vectors. Vectors in Lisp have the functionality of vectors in other languages. The EuLisp

```
(setq v (make-vector 5))
```

makes a vector of 5 slots, numbered 0 to 4 (as a C vector). The value of `v` is now

```
#(() () () ())
```

(each slot is initialised to `()`). To get a value of a slot, use

```
(vector-ref v 3)
```

to access the slot numbered 3 in `v`. To put a value use the idiom

```
((setter vector-ref) v 3 hi)
```

which puts the symbol in slot numbered 3. The value of `v` is now

```
#(() () () hi ())
```

(Aside: the EuLisp `setter` idiom. The `setter` function is a function that takes an accessor function as argument, and returns the corresponding updator function. So `(setter vector-ref)` is a function to update a vector. There are many other `setter`s.)

As for everything else in Lisp, a "vector" is really a pointer to a block of memory, and when we pass a vector as an argument to a function, we are passing a pointer to the block of memory. This means that the local instance of the vector we get in the body of the function is, in fact, a pointer to the original vector. The effect of this is that if we update some slot of a vector in the function, this actually updates the original.

```
(defun foo (vec)
  ((setter vector-ref) vec 0 'zero))

(foo v)
```

then v has value #(zero () () hi ()).

Common Lisp goes way over the top with vectors: it calls them arrays, and they can be multidimensional. Simple usage is as follows.

```
(setq v (make-array 5))
```

makes an array of 5 slots, numbered 0 to 4. The value of v is #(NIL NIL NIL NIL NIL). Access values by

```
(aref v 3)
```

and set values by

```
(setf (aref v 3) 'hi)
```

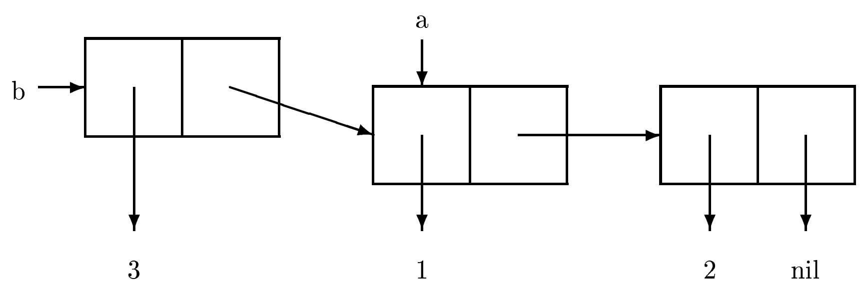
Now is the time to describe the general use of setter and setf.

4.2.3 Ever Increasing Circles

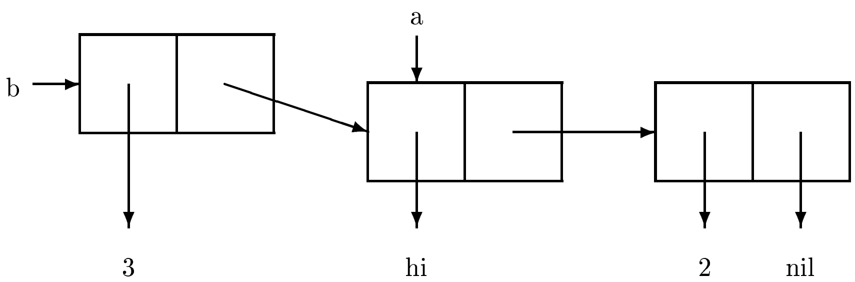
A member of a list can be any Lisp object, including itself. We can make self-referential objects by using the (Cambridge Lisp) functions rplaca and rplacd. The former destructively replaces the pointer that is the value of a car of a list by another pointer.

```
(setq a '(1 2))
(setq b (cons 3 a))
(rplaca a 'hi)
```

The value of a is now (hi 2). but, also, the value of b is (3 hi 2). To see the reason why think of the boxes model



The rplaca replaces the pointer to 1 by a pointer to hi.



Thus `rplac`s must be used with extreme care when we employ shared structure. The other function, `rplacd`, does the same for the cdr.

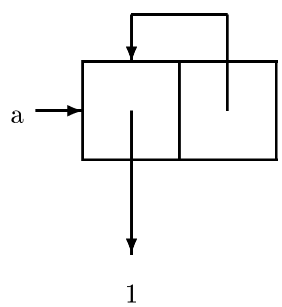
The real fun starts when we rplac the list into itself.

```
(setq a '(1 2))
(rplacd a a)
```

Don't do this! The value of `rplacd` is the last argument. In this case, it is the value of `a`, which after begin `rplacd`'d looks like

```
(1 1 1 1 1 ...
```

ad infinitum. The value of `a` looks like



and the printer loops forever. The print function `printl` can cope with circular structures, and prints the value as

```
%l1=(1 . %l1)
```

where the `%l1=` labels a pointer, and `%l1` refers back to it.

Such pointer bashing can be useful in its place, allowing us to create "infinite" structures, but they are best avoided until you know exactly what is going on.

In EuLisp there is a general mechanism for updating, namely `setter`. This, as mentioned above, takes a accessor function such as `vector-ref`, `car` or `cdr`, and returns a function that updates the place referred to by the accessor. We have seen `(setter vector-ref)` is a function that updates a vector, but also there is `(setter car)` and `(setter cdr)`. The former is a function that updates a `car` of a list, namely `rplaca`. So

```
((setter car) a 'hi)
```

in EuLisp is the `(rplaca a 'hi)` of Cambridge Lisp.

In Common Lisp there is a similar mechanism for destructive updating, namely `setf`. The call

```
(setf form value)
```

finds the object referred to by `form`, and destructively replaces it by `value`.

```
(setf (car a) 23)
```

replaces the car of `a` by 23. This is `rplaca`.

```
(setf (cdr a) a)
```

is `rplacd`. We have already seen `(setf (get 'foo 'bar) 'baz)` to update a property on a symbol, and `(setf (aref v 3) 'hi)` to update a vector. Here is a simpler example:

```
(setf 'x '(1 2))
```

this is the same as `(set 'x '(1 2))`. The function `setf` is used in many other places, for example

```
(setf (symbol-function 'foo)
      #'(lambda (n) (+ n 1)))
```

will replace the contents of the function cell of `foo`.

4.3 The Evaluator

We finish on the function that lies at the heart of Lisp: `eval`. This takes an arbitrary Lisp expression, and returns the result of evaluating that expression.

```
(eval '(plus 2 3))
```

returns 5. Eval takes an expression: if it is a symbol, it retrieves the relevant value; if it is a list, it regards it as a function call, determines which function is to be used, evaluates the arguments of the function, and applies the function to the arguments. It also takes note of things like `quote`, `cond`, and macro expansion.

The argument of `eval` is evaluated as is normal for a function. The result is regarded as a piece of program to be executed, which is what `eval` does. We can regard it as a function that takes a piece of text and breathes life into it to make a living chunk of running code.

Consider the following

```
(eval (reverse '('quote print)))
```

The argument of `eval` is call to `reverse`. To evaluate the `reverse`, we look at *its* argument, which is a quoted expression containing two pieces of data:

```
'('quote print)
```

this has value `('quote print)`. This is what `reverse` gets: it replies with the list `(print 'quote)`. When `eval` receives the latter list it regards it as code to be executed. The action it takes is to print the symbol "`quote`". Somehow, the `print`, which was hidden behind several quotes, and started life most definitely as data, became activated as program.

This action is central to Lisp: a list `(plus 2 3)` can be data, or it can be program—its all up to the programmer (or perhaps the program) how it is interpreted.

Here is an outline for a definition of `eval` in a 1-Lisp (adapted from Abelson and Sussman).

```
(de eval (exp env)
  (cond ((self-evaluating exp) exp)
        ((quoted exp) (text-of-quotation exp))
        ((variable exp) (lookup-variable exp env))
        ((definition exp) (eval-definition exp env))
        ((assignment exp) (eval-assignment exp env))
        ((lambdap exp) (make-procedure exp env))
        ((conditional exp) (make-cond (clauses exp) env))
        ((application exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (t (error "Unrecognised expression"))))
```

The argument `exp` is the expression to be evaluated, the `env` is the *environment*, which, roughly speaking, is the current set of bindings of variables.

Self-evaluating expressions are left as they are, while quoted expressions are easy to evaluate, just return the expression after the quote. (The functions above are mostly trivial, thus `self-evaluating` might be `(lambda (exp) (or (numberp exp) (string exp)))`; perhaps `quoted` is `(lambda (exp) (if (atom exp) nil (eq (car exp) 'quote)))`, and `text-of-quotation` could be `(lambda (exp) (cadr exp))`, and so on).

If we have a variable, we need to look up its value; we do so by referring to the environment `env`, which may contain bindings that shadow the global value of a variable.

Assignments (`setq`) and definitions (`de`) are quite similar, only the definition needs to wrap a lambda about the body. These need the environment, as they potentially overwrite bindings.

A lambda expression is recognised by its car being `lambda`. The `make-procedure` procedure packages up (in a cons) the lambda-body and the environment. When the lambda is called we will need to refer back to the environment in which the lambda was defined for the lexical bindings of the variables it contains.

Conditionals (i.e., `cond`) need a modicum of special treatment to avoid evaluating all the arguments.

A function application is any non-atomic expression that is not covered by one of the above cases. We proceed by finding the function to be applied: this is the value of the car of the expression. The car can be an arbitrary expression, and so we call `eval` to determine its value. The arguments are all evaluated in the current environment and set in a list by `list-of-values` ready for the use by `apply`. The definition of `list-of-values` is a simple recursive call

```
(de list-of-values (exps env)
  (cond ((null exps) nil)
        (t (cons (eval (car exps) env)
                 (list-of-values (cdr exps))))))
```

Apply works by extending the current environment by binding the parameters of the function to their arguments and doing a primitive function application (below the level of Lisp).

The pair `eval-assignment` and `eval-definition` determine the value to be assigned, and then modify the environment to reflect the new values for the variables that have been changed.

The environment can represented quite simply as an association list:

```
( (x . 1) (y . 2) (z . 3) )
```

reading from the car, this says that the current bindings for `x`, `y` and `z` have values 1, 2, and 3. Or rather, the environment is a *list* of association lists. Each association list is called a *frame*.

```
( ( (x . 1) (y . 2) (z . 3) )
  ( (z . 1) (x . 2) (w . 4) )
)
```

The current frame is the car; in the previous frame, which would correspond to an earlier function application, `z` has value 1, `x` has value 2, and `w`. In this outer frame, `y` has no value. Adding a binding to the current frame is a cons on to the front of the outermost frame; deleting a frame (as happened when we move out of the extent of a binding is a cdr.

In the inner frame, if we reference `w` we do not find it in the current frame. When this happens, we search though the next closest frame, and so on out until we find as binding. If there is no binding in any frame, then the reference is in error. This method, called *deep binding*, has the drawback that we may have to search through many frames to find a binding for a given variable. This can be slow. Other strategies exist, such as *shallow binding* and Padget's method, that address this problem.

The above definitions can be fleshed out into a fully lexically scoped Lisp interpreter (see Abelson and Sussman). Notice that it would be relatively easy to convert to a 2-Lisp: the definition and application functions would need to be changed, but otherwise things are much the same. In fact this would be a good way of comparing, say 1-Lisps and 2-Lisps, or lexical and dynamic scoping: it is only a matter of code.

4.4 The End

Conceptually, the read-eval-print loop of the interpreter is

```
(loop
  (print (eval (read)))
)
```

The value of `read` is a Lisp expression. This is passed to `eval`, which evaluates it; then `print` prints it. And so on.

The function eval is where McCarthy started: with it, we can write Lisp in Lisp, and change our semantics to our hearts content. This reflective (or maybe introspective) property is why Lisp is still such an important and useful language, despite its age.