Functional Programming & Lisp Processing with Lisp
CS 305
©Denbigh Starkey

1. Introduction

Lisp (**LIS**t **P**rocessor[1]) was one of the pioneer programming languages, designed at MIT for Artificial Intelligence applications, and has continued to be one of the most important languages for AI (although in highly extended forms). It is also, particularly in its original form, the classic example of a functional programming language.

The goal of these notes isn't to teach Lisp as a language, but is to give you a feeling for the power that can be achieved with a very simple set of programming constructs. For example many programmers, if asked to describe a single programming construct that would be in all programming languages, would select the assignment statement or maybe simple loops, but the original Lisp didn't provide either of them.

The main current version of Lisp is Common Lisp, which is a dramatic extension over the original Lisp, and includes structures like loops and assignment statements. I'll be using the Common Lisp interpreter, but I'll be restricting my discussions to Lisp 1.5, the first released version, which is also often called Pure Lisp. In these notes I'll just call it Lisp.

Lisp was developed by McCarthy *et al.* at MIT as a language for symbolic processing, which effectively means for Artificial Intelligence. It is mathematically very pure, since it is based on Alonzo Church's Calculus of Lambda Variations, although this is

[1] For reasons that will become obvious later, many people claim that the acronym should be Lots of Idiotic Silly Parentheses.

less visible than it used to be. For example, one no longer has to use the keyword LAMBDA when defining a function, so the lambda form is mainly hidden.

Any Lisp program is itself a Lisp data object, and so a program can take itself as data and modify itself. I won't be doing this.

2. Lisp Data Types: Atoms and S-Expressions

Lisp has one data type, called the S-expression, and special versions of this called atoms and lists. I'll first define atoms, then use these to define S-expressions, and in the next section I'll use them to define lists.

An atom is any atomic unit. E.g., `33`, `-22.5`, `NIL`, `nil`, and `Denbigh` are all atoms.

An S-expression is defined recursively with:

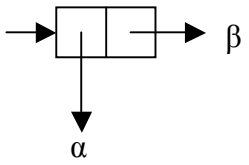
1. An atom is an S-expression
2. If α and β are S-expressions then (α . β) is an s-expression.

E.g., the following are S-expressions:

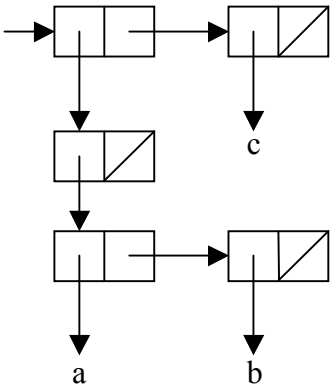
```
33
Denbigh
Starkey
(Denbigh.Starkey)
(Denbigh.(Starkey.nil))
(((a.(b.nil)).nil).(c.nil))
```

(You might, correctly, be getting the impression that nil is an important atom in Lisp. We'll see why in the next section when we get to lists.)

Internally a non-atomic S-expression (α . β) is represented as:

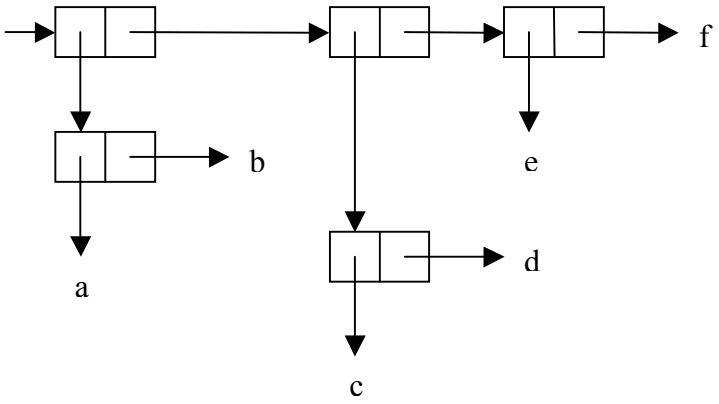


and so, for example, `(((a.(b.nil)).nil).(c . nil))` will be represented as:



Note the shorthand that I have used for nil in this picture where, instead of putting a pointer to nil I have just put a diagonal line in that part of the S-expression.

One more example, and then I'll move on to lists. The internal representation of `((a.b).((c.d).(e.f)))` is



As a piece of history, which I'll get back to later, when Lisp was implemented on the IBM 7090 that system had an address register and a decrement register, which were used to store the pointers to the two parts of a non-atomic s-expression. So with a general s-expression (α . β), the pointer to α was the **c**ontents of the **a**ddress **r**egister (CAR was the assembly instruction) and the pointer to β was the **c**ontents of the **d**ecrement **r**egister (CDR). These, as we'll see later, became two of the five function names in Lisp.

3. Lisp Data Types: Lists

A list is the most common data type in Lisp, but it is just a shorthand for a certain form of s-expression. The list with *n* elements:

$(e_1\ e_2\ \ldots\ e_n)$

is a shorthand for:

(*e*1 . (*e*2 ... (*en* . nil)...))

where there are *n* right parentheses at the end. E.g., the following pairs are equivalent:

```
(a b c)          (a.(b.(c.nil)))
(((a b)) c)      (((a.(b.nil)).nil).(c.nil))
()               nil
```

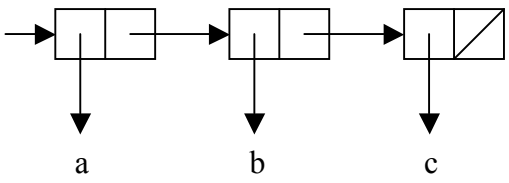
The first one is a vanilla list of three elements (note the three right parentheses at the end of the s-expression).

The second one can be broken down to see what is going on.

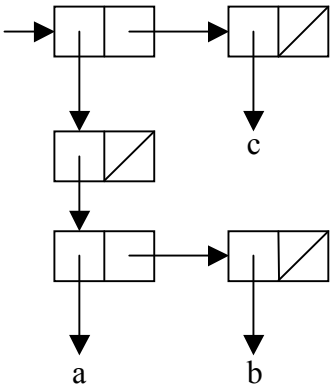
(a) `((a.(b.nil))` is the list of two elements `(a b).`
(b) Any one element list (α) is (α.nil), so `((a b))` is the list containing one element, the list `(a b)`, so it is `(((a.(b.nil)).nil)`, substituting in the expression from `(a)` for α.
(c) Finally at the highest level we have a two element list consisting of the elements `((a b))` and `c`.

The third one is the special case of the empty list. It is the atom nil, although it can also be represented as an empty pair of parens, `()`.

The internal representation of the first example is just:



The internal representation of the second, `(((a b)) c)` was given in the previous section. Repeating it here for simplicity, make sure that you understand it.



4. Lisp Built-In Functions, car, cdr, cons, eq, and atom

Lisp only had five built-in functions, and everything else had to be built on top of them. Two (`car` and `cdr`[2]) give access to the two halves of a non-atomic S-expression, `cons` constructs more complex S-expressions from simple ones, `eq` is a predicate (i.e., returns true or false) which checks to see whether or not two atoms are equal, and `atom` is a predicate that determines whether its argument is atomic or non-atomic. In this section I'll use Lisp pseudocode (also called the Lisp metalanguage) to describe these functions, later I'll convert to real Lisp notation. However, looking ahead briefly, a call like `cons[a; (b.c)]` to create `(a.(b.c))` will actually be written `(cons 'a '(b.c))` or `(cons (quote a) (quote (b.c)))`. So, a bit more formally:

$$\text{car}[s] = \begin{cases} \alpha & \text{if } s = (\alpha.\beta) \\ \text{undef} & \text{otherwise} \end{cases}$$

$$\text{cdr}[s] = \begin{cases} \beta & \text{if } s = (\alpha.\beta) \\ \text{undef} & \text{otherwise} \end{cases}$$

$$\text{cons}[s; t] = (s\ .\ t)$$

$$\text{eq}[s; t] = \begin{cases} T & \text{if } s = t \text{ and } s,t \text{ both atomic} \\ \text{nil} & \text{if } s \neq t \text{ and } s,t \text{ both atomic} \\ \text{undef} & \text{otherwise} \end{cases}$$

$$\text{atom}[s] = \begin{cases} T & \text{if } s \text{ is atomic} \\ \text{nil} & \text{if } s \text{ is nonatomic} \end{cases}$$

[2] The closest that I can get is that `cdr` rhymes with gooder.

Note the names car and cdr for the first and last halves of an S-expression, which came from the registers on the 7090 as I discussed earlier. Modern Lisps tend to supply aliases like head and tail for these two functions.

One minor point before I give examples of these is that F is represented in Lisp with the atom nil.

Some examples:

```
car[((a.b).(c.(d.e)))] = (a.b)
cdr[((a.b).(c.(d.e)))] = (c.(d.e))
car[(a b c)] = car[(a.(b.(c.nil)))] = a
cdr[(a b c)] = cdr[(a.(b.(c.nil)))] = (b.(c.nil)) = (b c)
eq[a; a] = T
eq[a; b] = nil
```

car[Denbigh] is undefined

cdr[Denbigh] is undefined

eq[(a.b); (a.b)] is undefined (although it will usually be T, but don't rely on this)

One important thing to extract from this is that if we take the car of a non-empty list then we get the first element of the list, and if we take the cdr we will get the rest of the list. A couple more examples of each of these:

```
car[((a b) c (d e))] = (a b)
cdr[((a b) c (d e))] = (c (d e))
car[()] = car[nil] = undefined
cdr[()] = cdr[nil] = undefined
```

Also, if we cons something onto a list we will add it on as a new element at the front of the list. E.g.:

```
cons[a; (b c d)] = cons[a; (b.(c.(d.nil)))]
                           = (a.(b.(c.(d.nil))))
                           = (a b c d)
cons((a b); (c d)] = ((a b) c d)
```

5. User-Defined Functions in Lisp

There are two ways to define functions in Lisp, composition and conditional statements.

Composition: Say that we want to write a function, which we'll call cadr, to find the second element of a list l that contains at least two elements. Well cdr[l] returns a list containing everything except the first element, so car[cdr[l]] returns the first element of that, which is the second element of the original list. So we just need to define:

```
cadr[l] = car[cdr[l]]
```

Lisp systems predefine a lot of the common combinations of `car` and `cdr`, using a string of `a`'s and `d`'s between `c` and `r`. So, for example, `caddar[l]` would be `car[cdr[cdr[car[l]]]]`, which is the third element of the list that is the first element of `l`. For example, `caddar[((a b c d) (e f) g)]` is `c`.

Conditional expressions: The pseudocode conditional expression has the form:

```
[p1 → e1;
 p2 → e2;
 ...
 pn → en]
```

where each of the p_i is a predicate (evaluates to T or NIL) and each of the e_i is any expression, including conditional expressions. The interpretation of the conditional expression is that we work down the predicates, evaluating them until we find one, say p_i, that is true, and return e_i as the value of the expression. I.e., this is equivalent to

```
if (p1) e1;
elsif (p2) e2;
...
elsif (pn) en;
```

Now a problem is that if none of the left hand side predicates are true the conditional expression is undefined. To avoid this, the last left hand side predicate is always T, so that if all of the previous ones are false the last right hand side predicate will be returned. So this is like an elseif structure where the final else clause is required.

It is now time to do some more complex definitions:

Lisp gives us eq, which determines whether or not two atoms are equal, but isn't guaranteed to work if one or both of its arguments is non-atomic. We will often need a more general predicate, equal, which determines whether any pair of s-expressions are equal. The pseudocode to do this is:

```
equal[s; t] = [atom[s] → [atom[t] → eq[s, t];
                          T → nil];
               atom[t] → nil;
               equal[car[s]; car[t]] → equal[cdr[s]; cdr[t]];
               T → nil]
```

Trace through this. It says that is s is an atom then we return the value of the next two line conditional expression. It says that if *t* is also an atom then use `eq` to determine whether the two atoms are equal. If *t* isn't an atom then it can't be equal to the atom *s*. If, however, *s* wasn't an atom then if t is an atom we know that they can't be equal, so return `nil`. If neither are atoms we recursively check to see whether the two `car`s are the same. If they are, then equal will be `T` or `nil` based on whether the `cdr`s are the same. Finally, if they aren't atoms and the `car`s aren't the same, then we hit the `T`, and return `nil` because *s* and *t* can't be equal. All Lisp systems provide the predefined function `equal`, with this definition.

We could now use this definition to define a new function `null`, which determines whether a list is empty or not. Remembering that the empty list is the atom `nil`, we can just say:

```
null[l] = equal[l, nil]
```

or we could define it without using equal, with an efficiency gain, with:

```
null[l] = [atom[l] → eq[l, nil];
           T → nil]
```

For a first function definition for lists, I'll use `append`, which concatenates two list together. E.g., `append[(a b c); (d e)] = (a b c d e)`. To write this we are going to need conditional expressions and recursion, and the strategy that we'll use is one that we'll use in nearly every list processing function definition:

1. handle a case where a list is empty (we can test for this using null)
2. if it isn't empty then use `car` to get the first element of the list, and then recursively call the function using the rest of the list, which we get from `cdr`.

In this case say that we are trying to append two lists, `l` and `m`. Using this strategy, and concentrating on `l`, then if `l` is the empty list and we append it onto `m` then the result is just `m`. If l isn't empty then use recursion to `append` the `cdr` of `l` onto `m`, and then put the `car` of `l` onto the front. In pseudocode this becomes:

```
append[l; m] = [null[l] → m;
                T → cons[car[l]; append[cdr[l]; m]]]
```

A final example: We want to perform a high level reverse of a list: For example, `reverse[((a b) (c d e) f g)] = (g f (c d e) (a b))`, where the four elements have been reversed.

Using our basic strategy, if the list is empty, then just return the empty list, `nil`. If it isn't empty, then grab the first element with `car`, reverse the rest (the `cdr`) recursively, and then find a way to add the first element on at the back. That last part is slightly tricky, but if we make a list that contains the first element then we can use `append` to join the reverse of the `cdr` of the list and the list containing the `car`. I.e.,

```
reverse[l] = [null[l] → nil;
              T → append[reverse[cdr[l]]; cons[car[l]; nil]]]
```

6. Lisp Notation and Definitions

As I've mentioned several times, I've so far just being using a clean pseudocode to define everything. A real Lisp program is itself a list, and so it must have a Lisp-like notation. So I need to make some changes:

1. The function call `f[x; y; ...]` becomes `(f x y ...)`.
2. The conditional statement

```
[p1 → e1;
 p2 → e2;
 ...
 pn → en]
```

   becomes:

```
(cond (p1 e1)
      (p2 e2)
      ...
      (pn en))
```

3. The function definition `f[x; y; ...] = defn` becomes

```
(defun f (x y ...) defn)
```

Where the defun approach is that used in Common Lisp, not the original Lambda approach. For example, say that we want to write the definitions of `null`[3] and `append` that we had in the last section. They will become:

```
(defun null (l)
    (cond ((atom l) (eq l nil))
          (T nil)))

(defun append (l m)
    (cond ((null l) m)
          (T (cons (car l) (append (cdr l) m)))))
```

The definition of `equal` looks a bit messier because of the embedded conditional statement, but it is just following the same rules:

```
(defun equal (s t)
    (cond ((atom s) (cond ((atom t) (eq s t))
                          (T nil))
          ((equal (car s) (car t))
                  (equal (cdr s) (cdr t)))
          (T nil)))
```

A final change that we need to make occurs when we supply a value to test a function. Until now I've been just putting in the arguments directly, and so with pseudocode I'd say for example, append[(a b); (c d e)]. If we said

```
(append (a b) (c d e))
```

then it would attempt to compute append[a[b]; c[d; e]]. So we need to quote arguments like this with either the annoying (original) form:

```
(append (quote (a b)) (quote (c d e)))
```

[3] I'll use the second definition from that section.

or the shorthand (which everyone uses)

```
(append '(a b) '(c d e))
```

7. Another Example – reversall

Say that we want to write a function to reverse a list at all levels. E.g., we want `reverseall[((a b c) ((d e) f) g)]` to return `(g (f (e d)) (c b a))]`. I'll look at three possible ways to do this:

```
(defun revall1 (l)
    (cond
        ((null l) l)
        ((atom (car l)) (append (revall1 (cdr l))
                                (cons (car l) NIL)))
        (T (append (revall2 (cdr l))
                   (cons (revall1 (car l)) NIL)))
    )
)

(defun revall2 (l)
    (cond
        ((null l) l)
        ((atom l) l)
        (T (append (revall2 (cdr l))
                   (cons (revall2 (car l)) NIL)))
    )
)


(defun revall3 (l)
    (cond
        ((atom l) l)
        (T (append (revall3 (cdr l))
                   (cons (revall3 (car l)) NIL)))
    )
)
```

In pseudocode revall1 is:

```
revall1[l] = [null[l] → l;
              atom[car[l]] → append[revall1[cdr[l]]; cons[car[l]; nil]];
               T → append[revall1[cdr[l]]; cons[revall1[car[l]]; nil]]]
```

This is, in some ways, the obvious solution. Working through it,

1. If the list is empty, just return it as its reverse.
2. If the list is, say `(a (b c) d)` then we want to recursively `revall1` the `cdr` to get `(d (c b))` and then append `(a)` onto the end. We get the list containing `a` by consing it to `nil`.
3. Otherwise, if the `car` is non-atomic, then we want to recursively `revall1` it, and then append it onto the reverse of the `cdr` as before.

In pseudocode `reverse2` is:

```
reverse2[l] = [null[l] → l;
               atom[l] → l;
               T → append[reverse2[cdr[l]]; cons[reverse2[car[l]]; nil]]]]
```

This is a bit more sophisticated in terms of how it uses the recursion. E.g., if we have `(a (b c) d)` as before, then it won't immediately detect the atomicity of `a`, but will append `(d (c b))` to the list containing `reverse1[a]`. Then it handles the atomic non-null atom argument in the next recursive pass.

In pseudocode `reverse3` is:

```
reverse1[l] = [atom[l] → l;
               T → append[reverse3[cdr[l]]; cons[reverse3[car[l]]; nil]]]]
```

This just combines the first two conditions from `reverse2`. If `l` is null then it is the atom `nil`, so the `atom` test handles both cases.

8. Running a Lisp Program Using Common Lisp

On Unix/Linux systems the Common Lisp compiler is called `clisp`. So the simplest approach is to type `clisp` at the prompt, type in a definition, and test it.

To get out of `clisp` say `(exit)` or `(quit)`. None of the obvious words like exit or quit without parentheses will work.

A much better way to use clisp is to type your function definitions into a file named, say, `mylispfuns.l`, and then after calling `clisp` say

```
(load 'mylispfuns.l)
```

which reads in the definitions from the file.

E.g., if I have a file `reverse.l` which contains:

```
(defun reverse1 (l)
    (cond
        ((null l) l)
        ((atom (car l)) (append (reverse1 (cdr l))
                                (cons (car l) NIL)))
        (T (append (reverse2 (cdr l))
                   (cons (reverse1 (car l)) NIL)))
    )
)

(defun reverse2 (l)
    (cond
        ((null l) l)
        ((atom l) l)
        (T (append (reverse2 (cdr l))
                   (cons (reverse2 (car l)) NIL)))
    )
)
```

as I discussed in the last section, but omitting `revall3`, then a typical `clisp` session is shown below. I've bolded user input to distinguish it from system output.

```
% clisp
  i i i i i i i       ooooo    o        ooooooo   ooooo   ooooo
  I I I I I I I      8     8   8           8     8     o  8    8
  I  \ `+' /  I      8         8           8     8        8    8
   \  `-+-'  /       8         8           8      ooooo   8oooo
    `-__|__-'        8         8           8           8  8
        |            8     o   8           8     o     8  8
  ------+------       ooooo    8ooooooo  ooo8ooo   ooooo   8

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2001


[1]> (load 'revall.l)
;; Loading file revall.l ...
;; Loading of file revall.l is finished.
T
[2]> (revall1 '((a (b c)) d (e f)))
((F E) D ((C B) A))
[3]> (revall2 '((a (b c)) d (e f)))
((F E) D ((C B) A))
[4]> (revall3 '((a (b c)) d (e f)))

*** - EVAL: the function REVALL3 is undefined
1. Break [5]> (defun revall3 (l)
(cond
    ((atom l) l)
    (T (append (revall3 (cdr l)) (cons (revall3 (car l)) nil)))
))
REVALL3
1. Break [5]> (revall3 '((a (b c)) d (e f)))
((F E) D ((C B) A))
1. Break [5]> (quit)
Bye.
%
```

Here I have successfully loaded `revall.l`, which gave me the functions `revall1` and `revall2`, which I tested. Then when I tried to call `revall3` it couldn't (not surprisingly) find it, so I show that I can directly type in a definition, and then I successfully test that.

9. Modern Lisp Programming

As I stated at the beginning, the primary goal of these notes hasn't been to teach Lisp, particularly in its current forms, but has been to give you a feel for the fact that not all programming languages need to be based on Algol-like[4] structures. Pure Lisp reduces programming to its simplest functional forms, with conditional statements, composition, and recursion, which together provide a complete programming environment (i.e., if something is programmable then it can be programmed with just this minimal set of features). There are some major advantages to programming in an environment this simple, including the elimination of side effects, and making it much less difficult to prove the correctness of your programs.

[4] Which includes Algol's descendants like C or Java.

From a practical standpoint, however, most Lisp programmers come to the language with previous experience in other languages, and so feel lost without their familiar tools like while loops, and so use languages like Common Lisp which place these structures on top of the underlying pure Lisp. You will find, however, that you can be much more accomplished as a Common Lisp programmer if you understand the pure Lisp that it is based on.

9. Exercises

1. Give the full s-expression and the internal representation for the list `(((a b) c) d ((e)) (f))`

2. Write a Lisp 1.5 program that finds every odd element in a list. You can assume that the list contains at least one element. Test your program using `clisp` or any other Lisp system.

3. Write a Lisp 1.5 program that finds every even element in a list. E.g., `evens[(a b c d e)]` = `(b d)`, `evens[(a)]` = `()`, `evens[()]` = `()`. Test your program using `clisp` or any other Lisp system.

4. Write a Lisp 1.5 program that returns the last two elements of a list. You can assume that the list contains at least two elements. For example, `last2[(a b (c d) e)]` = `((c d) e)`, `last2[((a b) (c d))]` = `((a b) (c d))`. Test your program using `clisp` or any other Lisp system.

5. Write a Lisp 1.5 program that returns a list containing the first and last elements of a list. You can assume that the list contains at least two elements. For example, `firstlast[(a b (c d) e)]` = `(a e)`, `firstlast((a b) (c d))]` = `((a b) (c d))`. Test your program using `clisp` or any other Lisp system.