Introduction to Programming in Lisp

Supplementary handout for 4th Year AI lectures · D W Murray · Hilary 1991

1 Background

There are two widely used languages for AI, viz. Lisp and Prolog. The latter is *the* language for Logic Programming, but much of the remainder of the work is programmed in Lisp. Lisp is the general language for AI because it allows us to manipulate symbols and ideas in a commonsense manner.

Lisp is an acronym for **Lis**t **P**rocessing, a reference to the basic syntax of the language and aim of the language. The earliest list processing language was in fact IPL developed in the mid 1950's by Simon, Newell and Shaw. Lisp itself was conceived by John McCarthy and students in the late 1950's for use in the newly-named field of artificial intelligence. It caught on quickly in MIT's AI Project, was implemented on the IBM 704 and by 1962 to spread through other AI groups.

AI is still the largest application area for the language, but the removal of many of the flaws of early versions of the language have resulted in its gaining somewhat wider acceptance. One snag with Lisp is that although it started out as a very pure language based on mathematic logic, practical pressures mean that it has grown. There were many dialects which threaten the unity of the language, but recently there was a concerted effort to develop a more standard Lisp, viz. Common Lisp. Other Lisps you may hear of are FranzLisp, MacLisp, InterLisp, Cambridge Lisp, Le Lisp, ...

Some good things about Lisp are:

- Lisp is an early example of an interpreted language (though it can be compiled). Interpreting means that you get instant returns to what you type in without going through the repetive cycle of compilation, linking, execution and debugging. This suits AI workers rather well, who have a need to perform arbitrary symbolic manipulation, and often a need to prototype heuristices rapidly. A requirement of such a language is that it can automatically allocate space for new structures: this also requires a background garbage collector to tidy up memory after you.
- Lisp does not distinguish between program and data. They are syntactically equivalent. This means you can treat code as data, modifying it automatically perhaps.
- Lisp computing environments and interfaces tend to be rather good and promote high productivity.
- It is easy to learn (and forget!) in short time.

2 Getting started

Lisp has its fair share of terminology and jargon — atoms, s(ymbolic)-expressions, lists, bindings, evaluation, interpretation etc. This section will give you some familiarity with these terms and by the end of it you should be able to do some simple coding in Lisp. You will also learn that the way to success at Lisp is to have a thorough knowledge of where **(** and **)** are on the keyboard.

2.1 Preamble

You will find several lisp directories in $\sim labdwm/ai$. Copy these your space. In addition, you will find there an initialization file called init.lsp. This enables you to set up the default editor. It is set for emacs.

At your terminal type "kcl" for Kyoto Common Lisp. You will enter the top-level of the Lisp interpreter which will indicat that it is ready for your scribblings by showing a prompt $>$. To leave kcl type (bye).
The interpreter is ready to receive input. If your input is in a file, it can be loaded by typeing

```
> (load 'myfile.l)
```

Using

```
> (ed 'myfile.l)
```

will get you into the editor. On leaving the editor, you will be back into lisp. I find it convenient to define a function "dev" say which performs the edit-load cycle:

```
(defun dev () (ed 'myfile.l)(load 'myfile.l))
```

2.2 The interpreter

In Lisp the fundamentals object are word-like objects called *atoms*. These build up in sentence-like objects called *lists*. One can build lists of lists, so a hierarchical structure is built into Lisp. Atoms and lists are known collectively as *symbolic expressions*, or *s-expressions* for short. Almost everything in Lisp is an s-expression. Lisp is about manipulating symbolic expressions, just as other languages are tuned to arithmetic manipulation. That doesn't mean you can't do arithmetic in Lisp. It is true that Lisp used to be slow for numerical work, but this is no longer the case. Because of our backgrounds, this section will introduce concepts using arithmetic examples. Don't be deluded into thinking this is what Lisp is really for.

Try adding 6 to 4:

```
> (+ 6 4)
10
>
```

Now try:

```
> (* 6 4)
24
>
```

and

```
> (- 10 8)  ; NB Things right of a semi-colon are ignored
2
>           ; So this is a comment
```

2.3 Some terminology

The commands above are examples of s-expressions. S-expressions can either be of the *atomic* or *list* varieties, the latter being often called *parenthesised* s-expressions. A list contains *elements* which are themselves atoms or lists. In the s-expressions above, the first thing in the list was a *procedure* which indicated what to do with the *arguments*. Built-in Lisp procedures like **plus** are called *primitives*. In contrast to lists, atoms are s-expressions that Lisp cannot break up any more.

2.4 Atoms

In one sense atoms are likely variables in other languages. They can have a value bound to them using the primitive **setf**. Eg., consider this decimalized exposition of Micawberian economics:

```
> (setf income 95)
95
> (setf expenses 100)
100
> (setf credit (- income expenses))
-5
> (setf happy (> credit 0))
nil
```

Each list here starts with the procedure **setf**, which is itself an atom. It and "income", for example, are strictly speaking *symbolic atoms*, or *symbols*. The arguments to **setf** are income and 95. 95 is a *numeric atom* or, as we scientists call it, a *number*.

S-expressions which are to be worked on by the interpreter, or *evaluated*, are call *forms*. The result of the evaluation is the *returned* value. It is becoming clearer the order that the interpreter evaluates s-expressions. It applies an recursive algorithm called **eval**, which looks informally like:
eval

- look at the outermost parenthesized s-expression first.
- **eval** each of its arguments
- apply outermost procedure to arguments

Armed with this information about **eval** you will become immediately wary of producing producing unwanted or unnoticed side-effects when using **setf**.

```
> (setf x 3)
3
> (+ x (setf x (+ 2 3)))
8
> x
5
>
```

So `x` has changed its value – a side effect. It is not too difficult to see what is going on here, but consider the following example.

```
> (setf x 3)
3
> (+ x (setf x (+ 2 x)) x)
13
> x
5
```

If you haven't followed what is happening go back to read the order in which **eval** works.

2.5 Lists

The s-expressions above like (+ 6 4) are examples of *lists*. Other examples of lists are

```
(a)
(1 2 b 10)
(setf x 3)
(banana orange violin)
((((apple) (banana)) carembola))
(Bitter Lager Stout Mild (Scotch-Whisky Rum))
((((()()()))))
((((intoxicating-beverages))()))
```

How many elements do these have. If you don't agree with one, four, three, two, five, one, two – think again. Note that (Bitter Mild) is not the same list as (Mild Bitter), nor (Bitter Bitter) the same as ((Bitter) (Bitter)).
The second element of the last list is **()** — the empty list. The special atom **nil** is taken as naming this structure. Nil thus has the unique duality of being *both* an atom *and* a list.

2.6 Procedures and Errors

Notice that primitives and procedures need not necessarily take a fixed number of arguments. For example

```
>(+ 1 1 1 1 1 )
5
>(+)
0
```

But some are touchy:

```
> (atom 1)
t
> (atom 1 2)
Error: incorrect number of args to atom
```

Your actual error message might differ, but it should give you the same information. (Later we'll see that **atom** checks if its single argument is an atom.)

3 Lisp is about manipulating symbols

Nearly all the examples seen so far have involved arithmetical manipulation. If this was all that Lisp could do, there would be no point learning it. One might as well have used Fortran. But suppose someone asked you to do juggling acts with lists like (apple banana violin) or (Bitter Lager Stout Mild Spirits)? Tackling those sort of problems in Fortran would be extremely cumbersome. In this section we look more at the ways Lisp handles symbolic computation.

3.1 Quoting

There is a predicate procedure **equal** which tests whether s-expressions are equal. Let's check it out:

```
>(equal Bitter Stout)
Error: eval: Unbound Variable:  Bitter
```

Something is wrong. Of course atom Bitter has no value bound to it (Actually neither does Stout.) A correct example is:

```
> (equal (setf price_of_Bitter 110) (setf price_of_Stout 110))
t
>
```

Suppose we try **equal** with a list:

```
> (equal (Bitter) (Stout))
Error: eval: Undefined procedure Bitter
```

Another error – but why is Bitter an Undefined Procedure? I intended it to be an element in a list. Simply because **eval** sees a "(" and thinks the next item *must* be a *procedure* or *primitive*: but here its definition can't be found. So how can we write lists like `(Bitter)`, `(Bitter Stout)` or whatever and insist that these are just lists with one or two atomic drinks in them? In other words, can we stop **eval** having its wicked way with the lists?
The answer is yes, by **quote**ing:

```
>(equal (quote (Bitter)) (quote (Stout)))
nil
```

In practice, **quote**ing is so common that there is a shorthand:

```
>(equal '(Bitter) '(Bitter))
t
```

When you **quote** something and hand it to **eval**, it evaluates to itself. For example:

```
>'(Bitter Stout (Spirits))
(Bitter Stout (Spirits))
>'(Bitter Stout '(Spirits))
(Bitter Stout '(Spirits))
>'Bitter
Bitter
>'(Bitter 'Bitter (Bitter) '(Bitter))
(Bitter 'Bitter (Bitter) '(Bitter))
>
```

Quoting allows us to **setf** atoms to represent lists.

```
> (setf spirits '(Scotch Rum Vodka))
(Scotch Rum Vodka)
> spirits
(Scotch Rum Vodka)
```

One can also bind atoms to other atoms:

```
> (setf x 'y)
y
> x
y
> (setf y 3)
3
> y
3
> x
y
>
```

You may have wondered why we dont have to quote the `x` in the example above. In fact **setf** itself is a short way of writing **set (quote)**.

```
>(setf x 3)            ; this is same as
3
>(set (quote x) 3)     ; this
3
>
```

Primitive **set** changes the 'value of its value' rather than the 'value of its own argument'. Take care to understand this indirection.

```
>(set 'x 'y)
y
>(set x 4)    ; alters the value of the value of x
4
>y
4
>x
y
>
```

3.2 Primitives car, cdr and cons

Now that you understand **quote**ing, you are in a position to do something more adventurous with symbolic computation.
Operator **car** returns the first element of a list argument

```
>(car '(a b c))
a
```

whereas **cdr** returns the argument list of elements minus its head, as in:

```
>(cdr '(a b c))
(b c)
```

A useful mnemonics may be **cAr** returns first in **A**lphabet but **cDr D**ecapitates. Both operators are non-destructive:

```
> (setf x '(a b c))
(a b c)
> (car x)
a
> x
(a b c)
> (cdr x)
(b c)
> x
(a b c)
>
```

In Common Lisp there are operators **first** and **rest** which ay be used as direct replacements of **car** and **cdr**. However, if you can't remember what **car** and **cdr** stand for, you will find difficulty useing the following operators.

3.3 Primitives cadr, cdar, ETC

If someone asked you for

```
> (car (cdr (car (cdr '((a b c) (d e f))))))
```

your mind's **eval** procedure would (no doubt) instantly return `e`. It is actually more efficient and saves space to use the stuttering extensions of **car** and **cdr**:

```
>(cadadr '((a b c) (d e f)))
e
>
```

Any **c...r** can be built up. However, as huge lists of **car**'s and **cdr**'s form impenetrable code and are frowned on as bad style, so also are long rambling stutters.

3.4 List building with cons,list,append

Operator **cons** takes two arguments – the first may be any data-item, the second *must* be a list – and returns a list which is the second list with the first argument at its head.

```
> (cons 8 nil)
(8)
> (cons 6 '(7 8 9))
(6 7 8 9)
> (cons '(a b) '(c d))
((a b) c d)
```

Notice

```
> (car (cons 3 '(8 9))
3
> (cdr (cons 3 '(8 9))
(8 9)
> (car '(cons 3 '(8 9)))
cons
```

Note that **cons** is also non-destructive. It is also quite expensive in resources. Because it creates a new list, it has to allocate memory, and eventually the garbage-collector has to come round to clear up unused **cons**-memory. A **car** might take one cycle to execute, whereas a **cons**, when garbage collecting is included, might take a 1000.
To build the list (a b c) one could use

```
> (cons 'a (cons 'b (cons 'c nil)))
(a b c)
```

but much easier is:

```
> (list 'a 'b 'c)
(a b c)
```

list takes any number of arguments, evaluates them, and builds a list out of the values.
By contrast, **append**'s arguments *must* evaluate to lists, and a list made up of all the individual elements is returned.

```
> (list 'a '(b c) 'd '(e f g h))
(a (b c) d (e f g h))
>(append '(a) '(b c) '(d) '(e f g h))
(a b c d e f g h)
```

3.5 Primitives length, reverse, subst and last

The following primitive are all self-explanatory.

```
>(length '(Bitter Stout)) ; length's argument must be list
2
>(length ())
0


>(reverse '(SHOT A DUCK!))
(DUCK! A SHOT)
>(reverse '((a b) (l e)))
((l e) (a b))


(subst <new expression> <old atom> <expression to subst in>)
>(subst 'a 'x (subst 'b 'y '(sqrt (+ (* x x ) (* y y)))))
(sqrt (+ (* a a) (* b b)))

>(last '( (apple) carrot banana (rice pudding)))
((rice pudding)) ; makes a list containing last element
```

---

4 Evaluation

In our discussion of **quote** we have already noted that the the Lisp interpreter assumes that the first element of a list is a procedure, and assumes that the remaining elements are arguments to that procedure:

```
( <procedure> <arg-1> ... <arg-n>)
```

Applying operators to arguments is called *evaluation*, and all the interpreter does is run a continuous "read-eval-print" cycle, reading input, evaluating and printing the returned value: We show the code to make your own Lisp top-level later on, but informally the process looks like:

```
while not end-of-file
do
        read an S-expression S
        evaluate S-expression S
        print result
```

So, Lisp evaluates everything and expressions always return a value (except, as noted above, **reset**).
In more detail, **eval** runs as follows.

```
EVAL
if (S is an atom) then {
    if (S is numeric) then {
        result is corresponding base10 number
    }
    else {
        if (S currently has value V) then {
            result is V
        }
        else {
            Error: Unbound variable
        }
    }
else {
    if ( car of S is procedure F ) then {
        determine which arguments are to be evaluated
        Apply EVAL to appropriate arguments
        Apply F to accumulated results
     }
     else {
         Error: undefined procedure F
     }
}
```

5 User-defined procedures

This section deals with defining procedures to do useful commonly-used tasks.
Simple procedure definitions are of the form:

```
(defun <proc-name> (<par-1> <par-2> ... <par-N>)
    (... body of code 1 ...)
     ...
    (... body of code M ...)
)
```

where <proc-name> is an alphanumeric atom, as are the various parameter names.
Note that it is the value returned by the last body of code that is returned as the value of the procedure. For example

```
>
(defun distance (x1 y1 x2 y2)
  (setf p 9) ; First body of code ignored entirely
  (sqrt    ; Start of last body
```

```
    (+ (* (- x1 x2) (- x1 x2))
          (* (- y1 y2) (- y1 y2))
    )
  )        ; end of last body
)

distance
>(distance 1 1 2 2 )
1.414213562373095
>
```

If we switched these bodies of code around, the result would always be 9:

```
>
(defun not-distance (x1 y1 x2 y2)
  (sqrt ; Start of first body
    (+ (* (- x1 x2) (- x1 x2))
          (* (- y1 y2) (- y1 y2))
    )
  )     ; end of first body (ignored)
  (setf p 9) ; Value of last body of code returned
)
not-distance
> (not-distance 1 1 2 2)
9
>
```

In this simple example, the assumed convention is that when the the procedure is called *all* the argument expression are to be evaluated and the results passed over for use in the procedure body.

```
> (distance 0 (+ -2 3) (- 10 8) (* 0.5 4.0))
1.414213562373095
>
```

The formal parameters are bound to their argument values only during the procedure. They are restored after the return to the calling environment. For example:

```
> (setf x1 44)
44
> (distance 1 1 2 2 )
1.414213562373095
> x1
44
>
```

However because p was not a formal its scope extends beyond the procedure. You should find:

```
> p
9
```

The issues of scope are a little tricky. Look at section 5.2 and in Winston and Horn.
Here is an example of a procedure that does symbolic manipulation

```
>(defun xcons (list element) (cons (element list)))
xcons
> (xcons '(b c) 'a)
(a b c)
>
```

5.1 Recursive procedures

Recursion, the ability of a procedure to call itself, is discussed in more detail later. For now, here is an obvious enough example:

```
> (defun ismemberof (item mylist)
     (cond ((null mylist) nil)
           ((equal item (car mylist)) t)
             ( t (ismemberof item (cdr mylist)))
       )
)
ismemberof
(setf loonyleft '(Skinner Heffer Benn))
(Skinner Heffer Benn)
> (setf ravingright '(Ridley Tebbit Fishburn))
(Thatcher Ridley Lawson)
> (ismemberof 'Kinnock loonyleft)
nil
> (ismemberof 'Lawson ravingright)
t
>
```

This example uses the conditional **cond** and the predicate **null** which are discussed in the next section. The example is clear enough though. The conditional checks whether the list `mylist` is empty. If it is not, is the item at the head of mylist? If not, use the tail, **cdr**, of mylist in the next call.

5.2 Scope, binding and free variables

So far, I think all the Lisp we have seen has been independent of dialect. We come now, however, to a discussion of scope, or to what is a variable bound at any particular point in the program.
We have seen that the scope of formal parameters to a procedure is confined to the procedure using them. But what about free variables – variables which are not formal parameters to a procedure, but are nevertheless fiddled with during the procedure. For example:

```
> (defun fiddle (x1)  ; x1 is formal parameter
      (setf x2 3)      ; x2 is free variable
```

```
      (+ x1 3)
)
fiddle
```

A set of bindings will be called an environment. If a language employs *dynamic scoping* then the binding of variables in a procedure is determined by the activation environment, that is the environment in force when the procedure is called. Older dialects of Lisp used this. Common-Lisp, however, uses lexical or static scoping, where the bindings are determined by the definition environment, the environment in force when the procedure using the free variables was defined. Variables defined at the top level and which do not appear in the activation or definition environment are *global* free variables.
Using the example above:

```
> (setf x0 10)
10
> (setf x1 20)
20
> (setf x2 30)
30
> (fiddle x0)
13
> x0
10                ; x0 always unchanged
> x1
20                ; x1 changed in fiddle, but restored
> x2
3                 ; x2 changed for good
>
```

Variables `x1` and `x2` are bound and free, respectively. Clearly, one must treat scope with care, lest you introduce unwanted side effects. One way of introducing local variables whose scope is confined is by using **let** which we discuss now.

5.3 Binding with let

In our fiddle procedure, variable `x2` was a free parameter and was changed for good. So how can we confine the scope of variables which are not free parameters? The old fashioned way was by using **prog**, but it is better in CommonLisp to use **let**, whose syntax is

```
(let ((<local-par1><initial-value1>)
      (<local-par2><initial-value2>)
      ...
      (<local-parM><initial-valueM>))
      <body of let>
)
```

A simple example would be:

```
> (defun newfiddle (x1)  ; x1 is formal parameter
   (let ((x2 0))          ; x2 local and initialized to 0
      (setf x2 3)         ; x2 is local
      (+ x1 3)))
newfiddle
> (setf x0 10)
10
> (setf x1 20)
20
> (setf x2 30)
30
> (newfiddle x0)
13
> x0
10               ; x0 always unchanged
> x1
20               ; x1 changed in fiddle, but restored
> x2
30               ; x2 changed in fiddle, but restored
>
```

In the above example, we used only one local parameter. However, if there were more, it is essential to appreciate that all their values are evaluated before being assigned. That is, the assignment is quasi-parallel, not sequential. This means that one cannot use an earlier value to initialize a later one.

```
(let (( local1 12)
      ( local2 (+ local1 10)) ; BUGGED - local2 won't be 22
      <body of let>
)
```

To do this you must use **let***.
An interesting aside is that **psetf** is a parallel version of **setf**; it does all the evaluations first, then the assignments. So this will switch the values:

```
>(setf a 'aardvark b 'badger) ; note multiple assignments
badger
>(psetf a b b a)
>a
badger
>b
aardvark
```

5.4 Procedure Arguments which are also Procedures

Suppose one wanted to write a program that would use a procedure as a variable, for example, a program that would compute say the sum of some function of the first four integers (straight sum, sum of squares, sum of square-roots etc). As we don't know about iteration yet we will have to sum explicitly. Our first try might be:

```
>(defun sumfour (operator) ; BUGGED!
  (+ (operator 1) (operator 2) (operator 3) (operator 4)))
sumfour
> (sumfour 'sqrt)
6.146264369941972
```

Fine – but is it? Now type in:

```
>(defun operator (n)
(* n n))
operator
```

That is define procedure operator as something that works out the square of a number. Now try:

```
>(sumfour 'sqrt)
30   ; * WRONG ! *
>(sumfour 'Anything_you_fancy)
30
```

Lisp has over-ridden the definition of operator given as a formal parameter with its (our) own internal definition of operator.
To do this properly, we need to wrest the control of procedure evaluation from the interpreter, at least temporarily. Lisp has several procedures to do this.

One such is **apply** which takes two arguments, the first of which should evaluate to a procedure name, and the second to a list of arguments to that procedure. Compare the following:

```
> (cons 'a '(b c))
(a b c)
> (apply 'cons '(a (b c)))
(a b c)
>
```

So the write a bug-free version of our earlier code:

```
>(defun sumfour (operator)
  (+ (apply operator '(1))
     (apply operator '(2))
     (apply operator '(3))
     (apply operator '(4))))
sumfour
>(defun square (n) (* n n))
times
(defun cube (n) (* n n n))
cube
> (sumfour 'cube)
100
> (sumfour 'sqrt)
6.146264369941972
>
```

Note that the second argument to **apply** *must* evaluate to a *list*.

An alternatives to **apply** is **funcall** which expects its arguments one after another.

```
> (apply 'cons '(a (b c)))
(a b c)
> (funcall 'cons 'a '(b c))
(a b c)
>(defun sumfour (operator)
  (+ (funcall operator 1)
     (funcall operator 2)
     (funcall operator 3)
     (funcall operator 4)))
sumfour
```

When we discuss iteration we shall ways of eliminating the unstylish repeated lines.

5.5 Lisp calls by value rather than reference

Lisp is a call by value language. When a procedure is called with an actual parameter, the procedure's formal parameter is passed the instantaneous value of the actual parameter, as opposed to an adress of the actual parameter. Thus the actual parameter and formal parameter can procede independently.

---

6 Predicates, Conditionals and logic

6.1 Truth values

There are no distinct Boolean variables in Lisp, but conventionally **nil** represents false and **t** is true. Both these evaluate to thenselves. In fact, all the built in logical procedures merely distinguish between **nil** and any non-**nil** value. So **t** is just there for convenience.

6.2 Predicates

Predicates are procedures with return either **t**, true, or **nil**, false.

atom tests whether something is an atom, **listp** tests whether something is a list.

```
>(atom 'a)
t
>(atom (cdr '(a b c)))
nil
>(listp '(a b c))
t
>(listp nil)
```

```
t
>(atom nil) ; nil is both!
t
>
```

The predicate **null** distinguishes **nil** from other s-expressions

```
>(null nil)
t
>(null '(a b c))
nil
>(null (null (null nil)))
t
>
```

We have already used **equal** which tests whether two s-expressions are alike. For example:

```
>(equal 'a 'a)
t
```

There are many arithmetic predicates. Eg, **numberp** tests whether its argument is a number.

```
>(numberp 6)
t
>(numberp 'a)
nil
>
```

zerop, oddp, evenp, lessp, ¡ , greaterp, ¿ are obvious.

Our earlier procedure ismemberof has a built in version in Lisp, **member**.

```
> (setf greatcomposers '(Purcell Bach Beethoven
                         Mozart Handel Haydn))
(Purcell Bach Beethoven Mozart Handel Haydn)
> (member 'Handel loonyleft)
nil
> (member 'Handel greatcomposers)
(Handel Haydn)
>
```

Note that **member** returns a found member along with the remainder in the list. This value is non-**nil** (and hence true to all predicates) and is more informative than merely returning **t** itself. Note also **member** only checks *elements* of the second list.

```
> (member 'y (x (y) z))
nil
>
```

6.3 Conditionals

Conditional branching is achieved using the **cond** procedure which we saw earlier on. This takes one or more arguments, representing the various options. Each argument should be of the form:

```
(<test for true> <action-1> ... <action-N>)
```

When **cond** finds a test that evaluates to something which is non-**nil**, it takes the actions after it and then leaves the **cond**. Test and actions are of course just s-expressions. It is often useful to make the last test of **cond** an trivially true default. **cond** returns the value of the last action it took, otherwise **nil**.

Consider:

```
> (defun yoghurt_plan (y fy)
     (cond
        ((member y fy)(setf plan '(Take yoghurt)))
        (     t       (setf plan '(Goto Tescos )))
     )
)
yoghurt_plan
> (setf yogs_in_fridge
    '(Banana Rhubarb Bats_of_the_Forest Rodent_and_Kiwi))
> (yoghurt_plan 'Rhubarb yogs_in_fridge)
(Take yoghurt)
> (yoghurt_plan 'Walnut_and_Mouse yogs_in_fridge)
(Goto Tescos)
>
```

and

```
>(defun agegroup (age)
(cond ((numberp age)
            (cond ((< age 0)   'unborn)
                  ((< age 3)   'baby)
                  ((< age 14)  'child)
                  ((<age 110)  'adult)
                  (    t       'Probably_deceased)))
      (    t        'Error_in_usage))
)
agegroup
>(agegroup 12)
child
>
```

Note that it is possible for **cond** to have only a test s-expression and no action s-expressions.

6.4 Logical Predicates

Lisp provides **and**, **or** and **not** with which to perform logic. For example,

```
>( or (atom 8) (atom '(a b c)))
t
>( and (atom 8) (atom '(a b c)))
nil
>(not nil)
t
```

and

```
>(defun even-50-100 (x)
  (and (numberp x) (evenp x) (> x 49) (< x 101)))
even-50-100
>(even-50-100 76)
t
>
```

7 Recursive and Iterative methods

7.1 Recursion

We have noted already that recursion is a major programming technique in Lisp.

We have seen the procedure:

```
> (defun ismemberof (item list)
     (cond ((null list) nil)
           ((equal item (car list)) t)
             ( t (ismemberof item (cdr list)))
       )
)
ismemberof
```

Any recursive procedure should do a little 'work', including checking whether it should terminate, then apply to procedure to something simpler. A classic recursive routine is to compute factorials:

```
>(defun factorial (m)
    (cond
      ((zerop m) 1)
      (   t (* m (factorial (- m 1))))))
```

A more elegant use of recursion appears in the definition of subst which substitutes the value of x for the value of y everywhere it appears in the value of z. The 'work' in this case is substitution at the atomic level, which is achieved by substituting the **car** and the **cdr** of z, then **cons**'ing the result.

```
> (defun subst (x y z)
     (cond
       ((equal y z) x)
       ((atom z) z)
       ((cons (subst x y (car z))(subst x y (cdr z))))))
> (subst 'a 'b '((a b c) b c ((((((b)))))) bbc))
((a a c) a c ((((((a)))))) bbc)
>
```

Recursive programming has the reputation of being difficult. However, it often makes tasks much simpler, because all you have to think about coding a little piece of work. The rest is chucked into the next recursive call. Difficulties often arise though in understanding someone else's recursive expressions, and convincing yourself that so simple a piece of code performs can perform so complicated a task!

7.1.1 A note on tail recursion

A procedure is tail recursive if the value returned is either something computed directly, or the value returned by the recursive call. Because a tail-recursive procedure does nothing with the returned value, it need not remember anything. For example, *ismemberof* defined above is tail recursive, whereas the procedure to find factorials is not. Tail-recursive procedures are efficient. They can be *automatically* converted to non-recursive code.

7.2 Iteration

In earlier versions of Lisp a primitive called **prog** was much used for iteration as it allowed, first, the introduction of explicit jumps, and, secondly, it allowed local scoping. It is best to forget **prog** entirely and use **do** for iteration.

do has the amazing syntax:

```
(do
   ((<par1> <initial value1> <update form 1>)
    (<par2> <initial value2> <update form 2>)
    ...
    (<parN> <initial valueN> <update form N>))
    (<termination test>
       <zero or more intermediate forms>
          <result>)
     <body>)
```

This awesome mess needs explanation, but that is best done by example. The first evaluates m to the power n.

```
>
(defun powerof (m n)
(do ((result 1)                    ; evaluate then assign
     (exponent n))                 ; ditto
```

```
    ((zerop exponent) result)        ;termination, 0 forms, result
    (setf result (* m result))       ;body
    (setf exponent (- exponent 1)))) ;more body
powerof
>(powerof 2 3)
8
```

Going through the **do** syntax step by step:

```
First, evaluate all the initial values (in parallel)
Then, assign the parameters to corresponding values (temporarily)
Evaluate test list: (<termination test> <forms> <result>)
If (test-list as a whole is non-nil) {
    If (car of test-form is true) {
Evaluate rest of forms in test list
        and return the last result as the value of do.
    }
    else {
        Evaluate the body forms  in the order they appear.
        Bind the parameters to the results of the update forms
    }
}
else { ‘‘Test form is nil’’
    return nil
}
```

We can see now that a more succinct way of writing powerof would be:

```
(defun powerof (m n)
(do ((result 1 (* m result)) ; Bind, assign , update
     (expon  n (- expon 1))) ; bind, assign, update
    ((zerop expon) result)))
powerof
```

The next example reverses a list:

```
(defun myreverse(mylist)
(do
   ((x mylist (cdr x)) ; bind, assign, update
    (result nil (cons (car x) result))) ; bind,assign,update
   ((null x) result))) ; test, no forms, result
myreverse
>(setf bus64B '(Cornmarket StGiles Summertown Kidlington))
(Cornmarket StGiles Summertown Kidlington)
> (myreverse bus64B)
(Kidlington Summertown StGiles Cornmarket)
>
```

Note that the assignment of values is done in parallel. There is another primitive, **do***, which will make the assignments serially.

7.2.1 prog1, progn

These allow the sequential execution of several expressions, and are handy in certain circumstances.

```
>(prog1 (setf a 'x) (setf b 'y) (setf a 'z)) ; returns
x                                             ; value of 1st
>(progn (setf a 'x) (setf b 'y) (setf a 'z)) ; returns
z                                             ; value of last
```

7.2.2 Iterative application of functions

There are some special Lisp functions which have iteration built into them. They are useful for applying the same function to everything in a list.

```
> (mapcar 'evenp '(0 1 2 3 4 5 6 7 8 9))
(t nil t nil t nil t nil t nil)
>
>(mapcar 'times '(1 2 3 4) '(5 6 7 8))
(5 12 21 32)
> (mapcar 'listp '(1 2 3 (1 2) a (a b) () nil ))
(nil nil nil t nil t t t)
>
```

This next example is rather more complicated, showing how the total number of atoms embraced by a list can counted:

```
> (defun ca (lst)
(cond ((null lst) 0)
      ((atom lst) 1)
      (t (apply 'plus (mapcar 'ca lst)))))
ca
> (ca '( a (b c) d (e (f))))
6
>
```

There are several other mapping procedures.

Often the result of using **mapcar** is a "wrapped-up" list, eg ((a b c) nil (d e f) (g) nil nil (h)). Such lists can be unwrapped by using the "apply-append" combination:

```
> (apply 'append '( (a b c) nil (d e f) (g) nil nil (h)))
(a b c d e f g h)
>
```

There is a primitive, **mapcan** which behave rather like **mapcar** followed by the "apply-append" combination. Suppose we have a routine **parents** that returns a list of the known parents of a person. One could then write a routine to discover all the ancestors of a person:

```
>(defun ancestors (person)
    (let (( p (parents person))) ; initialize local list p
      (cond ((null p) nil)
            (t (append p (apply 'append (mapcar 'ancestors p)))))))
```

Here you start out by finding the list of parents of a person. Then you use **mapcar** to recover lists of parents of those parents, and so on, which is unwrapped by "apply-append", and then stuck on the back of the list of parents. Using **mapcan**, this is better written as:

```
>(defun ancestors (person)
    (let (( p (parents person))) ; initialize local list p
      (cond ((null p) nil)
            (t (append p (mapcan 'ancestors p))))))
```

8 Properties, association and defstruct

So far we have treated atoms very much like variables in other languages. In Lisp, though, atoms can have *properties*. Properties are established by **setf**, retrieved by **get** and removed by **remprop**. For example:

```
> (get 'bert 'gender)
male

> (setf (get 'mabel 'gender) 'female)
female
> (get 'mabel 'gender)
female

>(remprop 'mabel 'gender) ; mabel no longer has
                          ; a gender property.
>
```

Properties are useful for

- One dimensional arrrays with atoms as indices
- Clusters of attribute associate with an item (eg., "frames")
- Arbitrary interconnections between data

8.1 Association Lists

An association list describes attributes of a particular object:

```
>(setf my-car '((make VWGolf)
                (cc   1292)
                (colour maroon)
                (date 1982)
                (Reg  AVM645Y)))
((make VWGolf) (cc 1292) (colour maroon)
 (date 1982) (Reg AVM645Y))
>(assoc 'date my-car)
(date 1982)
```

8.2 Data abstraction and defstruct

If we had to remember the details of whether atoms had properties or association lists we would rapidly get bogged down. Suppose for cars we made the association list a property, but for vans we had (perversely) made properties:

```
>(setf (get 'my-car 'car-details) '((make VWGolf)
                                    (cc   1292)
                                    (colour maroon)
                                    (date 1982)
                                    (Reg  AVM645Y)))
((make VWGolf) (cc 1292) (colour maroon) (date 1982) (Reg AVM645Y))

>(setf (get 'joes-van 'make) 'Ford)
Ford
>(setf (get 'joes-van 'colour) 'green)
green
... etc ...
```

As it stands to get at the details we would have to type different things for cars and vans

```
>(cadr (assoc 'colour (get my-car 'my-car-details)))
maroon
>(get 'joes-van 'colour)
green
```

This is very tedious. Instead one should have procedures that access the data as it is stored:

```
>
(defun car-colour (car)
(cadr (assoc 'colour (get car 'car-details))))

>(defun van-colour (van)
(get van 'colour)

> (car-colour my-car)
maroon
> (van-colour joes-van)
green
```

One would also have procedure to "make" new cars, to delete cars, to change details and so on. This process is called data abstraction.

The primitive **defstruct** is a powerful tool for data abstraction. A *structure* has *fields* and *field values*. One can set them up with default values:

```
>(defstruct (car)
    (make nil)
    (cc   nil)
    (colour nil)
    (is-a 'vehicle))
car
>
```

This has an amazing effect. First *selector* procedures are created automatically, viz: **car-make**, **car-cc**, **car-colour** and **car-is-a**. Secondly, a constructor procedure **make-car** is created. Thirdly, **defstruct** generalizes **setf** so we can alter values.

```
> (setf my-car (make-car))
> (car-cc my-car)
nil
>(setf my-car (make-car :make 'VWGolf :cc 1292 :colour 'maroon))
> (car-cc my-car)
1292
>(setf (car-is-a my-car) 'wreck)
>(car-is-a my-car)
wreck
```

9 Lambda

Sometimes it is useful to define procedures which have no name. These are implemented using the lambda notation. (In fact, lambda underlies all Lisp procedures.)

Often when writing a program one will create little procedures that are used but once. Eg, one might write a procedure to compute the fourth power of something which does nothing but `(* x x x x)`. It is about as easy to type `(* x x x x)` as `(fourth x)`, and involves less storage.

Often, too, one writes procedures to do highly specific things which are used only once. For example red-atom-p tests whether something is an atom and has the colour red.

```
> (setf balls
   '(football rugbyball pingpongball cricketball
     mayball oddball tennisball))
>(setf (get 'cricketball 'colour) 'red)
red
>(setf (get 'tennisball 'colour) 'white)
white
... etc ..

> (defun red-atom-p (a)
     (and (atom a) (equal (get a 'colour ) 'red)))
red-atom-p
> (mapcar
   'red-atom-p
   balls)
(nil nil nil t nil nil nil)
>
```

It seems wasteful to have to define a function just for this one off task, and to have the code elsewhere.

A partial remedy might be the following:

```
>(mapcar
  (defun red-atom-p (a) (and (atom a) (equal (get a 'colour ) 'red)))
  balls)
(nil nil nil t nil nil nil)
```

But this still defines a procedue name. This can be eradicated using **lambda** to name the procedure:

```
>(mapcar
  '(lambda (a) (and (atom a) (equal (get a 'colour ) 'red)))
  balls)
(nil nil nil t nil nil nil)
```

Using procedures is at the bottom line just a way of associating formal parameters with a chunk of code, and the procedure name just a convenient label to attach to the code. If we are not interested in saving the procedure we can write an in line procedure using **lambda**.

9.1 Using #' instead of '

When you use a function as an argument to another function where the argument needs **quote**ing, it is considered good practive to use **function** rather than **quote**, or #' instead of '.

The section of code above is short for

```
>(mapcar
  (quote (lambda (a) (and (atom a) (equal (get a 'colour ) 'red))))
  balls)
(nil nil nil t nil nil nil)
```

This call to **quote** is better made to **function**

```
>(mapcar
 (function (lambda (a) (and (atom a) (equal (get a 'colour ) 'red))))
 balls)
(nil nil nil t nil nil nil)
```

which in turn can reshortened to #'

```
>(mapcar
  #'(lambda (a) (and (atom a) (equal (get a 'colour ) 'red)))
  balls)
(nil nil nil t nil nil nil)
```

Actually, this is more than good practice; it affects **lambda**'s adherence with *lexical closure*. If #' is used, lexical scoping is obeyed, whereas if only ' is used dynamical scoping rules are obeyed. All this is of import if you insist on using free variables.

9.2 Primitives remove-if and remove-if-not

These are quite obvious:

```
>(remove-if-not 'red-atom-p balls)
(cricketball)
```

We could use **lambda** here too:

```
>(remove-if-not
  \#'(lambda (a) (and (atom a) (equal (get a 'colour ) 'red)))
  balls)
(cricketball)

>(remove-if
  \#'(lambda (a) (and (atom a) (equal (get a 'colour ) 'red)))
  balls)
(football rugbyball pingpongball mayball oddball tennisball)
```

9.3 Using lambda for variable binding

The **lambda** function is useful if we want to change the value of a variable for a short duration.

Suppose that we had a a function like **sumfour** above, which computed the sum of some specified function of the first four integers, but that in 99 cases out of 100 we were always going to want the sum of squares as opposed to the sum of square-roots, cubes or whatever. It would be nice to have this square function as a default value, but still to be able to change it to something else like squareroot if we wished.

One way of coping this would be to let the internal variable function (`operator`) access a global variable, which we change and then re-change.

```
>(setf operator 'square)
...
...
(setf hold operator)
(setf operator 'cube) ; or whatever
(sumfour)             ; sum of cubes just for here
(setf operator hold)  ; reset default to squares
>
```

Somewhat more elegant(?!?) is to use **lambda**.

```
>((lambda (operator)
  (sumfour)) ; cubes
        'cube)
)
>
```

10 I/O, I/O it's off to work we ...

One of the first things that people want to do in most languages is print "Hello World" to prove that the compiler and linker are working. The urge to do this in Lisp is lessened because the interpreter **evals** as you go along, but for those who are shaking for a wiff of reading and printing ...

10.1 read and print

```
>(setf x (read)) ; causes the tty to wait for input
(a b c)           *typed by you*
(a b c)
>x
(a b c)
>
```

Function **read** takes no arguments. Function **print** takes one. It starts with a newline and ends with a space.

```
>(defun printcubes ()
     (do (( m 0 ( + 1 m)))
     (nil)
     (print (* m m m))))
printcubes
>(printcubes)
0
1
8
27
64
125
...
```

read and **print** only deal with s-expression, so they are not much good for text.

```
> (defun sroots ()
   (print '(Plz enter a number))
   (print (sqrt (read))))
sroots
>(sroots)
(Plz enter a number) 36
6.0
>
```

This looks a bit ugly because of the () round the message.

We need a way of printing odd things like spaces,)'s etc. is the escape character, in that it allows the character following to escape usual treatment. However, if a name requires many such characters it can become awkward. Instead, Lisp allows the special phrase to be surrounded by vertical bars — – a multiple escape character. Whichever you use, **print** returns it with bars.

```
>'ab\ cd\(    ; this is continuous but with a space in it.
|ab cd(|       ; returns with equivalent but using bars
>'|ab cd|
|ab cd|
>
```

The primitive **print** is a bit restricting. Others output can be obtained with

- **format** Eg

  ```
  (format t ''Hello!~%'')
  ```

  where the ~% throws a newline.

- **terpri** which throws a newline.

10.2 Strings

Most of Lisp is an s-expression, but a string is a different data type surrounded by doublequotes.

```
> (setf sentence ''Most of Lisp is an  s-expression,
but a string is a different data type surrounded
by doublequotes.'')
''Most of Lisp is an s-expression, but a string is
a different data type surrounded by doublequotes.''
> sentence
''Most of Lisp is an s-expression, but etc ...''
>
```

There are some things to note about strings:

- Strings are not atoms and cannot have properties.
- Strings evaluate to themselves and cannot have bound values.
- Strings use less space than atoms.
- Strings are *vectors*, or 1D arrays, of characters.
- Boths vectors and lists are *sequences*

10.3 Ports and redirected i/o

Use with-open-file to redirect output or to read from a file:

```
> (with-open-file (myoutput-stream
                   ''/usr/users/dwm/ai/stuff''
                   :direction :output)
  ...
  (print '(a b c d) myoutput-stream)
)
```

Por

```
> (with-open-file (myinput-stream
                   ''/usr/users/dwm/ai/stuff''
                   :direction :output)
  ...
  (read myinput-stream)
  ...
)
```

10.4 The read-eval-print loop

We are able already to write the top level Lisp interpreter:

```
>(prog ()
read-eval-print
    (princ ''MyLispPrompt!>'')
     (print (eval (read)))
     (terpri)
     (go read-eval-print))
MyLispPrompt!>(setf Lisplist '(New top level))
(New top level)
MyLispPrompt!>Lisplist
(New top level)
MyLispPrompt!> (defun .....
...
```

11 Optional parameters, Macros and Backquotes

So far as we have seen, procedures in Lisp have a fixed number of arguments and evaluate then all. It is possible though to define procedures with optional parameters. This would be useful, for example, in the case of our procedure *sumfour* which was to usually sum squares, but occasionally might be used to sum cubes or whatever. Another example might be adding punctuation on the end of a sentence. Usually, it will be a full-stop, occasionally it will be an exclamation mark!

```
>(defun punctuate (sentence)
(append sentence '(\.))) ; NB use of escape \

>(punctuate '(I am a gumby))
(I am a gumby |.|)

>(defun punctuate (sentence &optional pmark)
     (cond ((not pmark) (append sentence '(\.)))
           (    t       (append sentence (list pmark)))))
```

```
punctuate

>(punctuate '(This is the end))
(This is the end |.|)

>(punctuate '(I hate Lisp) '!)
(I hate Lisp !)
```

One can have any number of optional arguments, or use a `&rest` parameter:

```
>(defun punctuate (sentence &rest end-bits)
     (cond ((null end-bits) (append sentence '(\.)))
           (    t       (append sentence end-bits))))

> (punctuate '(I hate Lisp) 'and 'I 'hate 'Prolog '!)
(I hate Lisp and I hate Prolog !)
```

Lisp procedures always evaluate their arguments. A good example of the awkward consequences of this follows. Suppose you were forever using:

```
(cond ( <test> <do this if test true> )
      (   t    <do this if test false>))
```

You might get the urge to write a procedure:

```
(defun fork (test true-result false-result) ; Disaster!
(cond ( test true-result)
      (   t  false-result)))
```

This looks good, but would be a disaster because fork would evaluate *all* its arguments. So both conditional results would be evaluation *volens nolens*. This will result in a waste of time, but also might lead to unimaginable side effects.

Macros, by contrast do not evaluate their arguments. They are defined by the syntax:

```
(defmacro <macro-name> <parameter-list> <body>)
```

Contrast the following:

```
>(defun myproc (parameter) (print parameter))
myproc
>(defmacro mymacro (parameter) (print parameter))
mymacro

>(setf sentence '(Every good boy deserves fireworks))

>(myproc sentence)
(Every good boy deserves fireworks)

>(mymacro sentence)
sentence
```

12 Debugging

```
> (dribble 'file)
```

echoes all screen output to a file.

```
> (trace function)
```

traces entering and exiting a function

```
(break '' This is break point 3 '')
```

puts in a break point. If you type :h kcl will print break commands that can be used. Use :r to resume. Note that any error will make you go down to a lower level, where the prompt is multiple arrows. Using :r moes you up a level.

```
(time (myroutine))
```

times a routine.