Introduction to Common Lisp

Christopher Buckingham
Aston University

October 17, 2017

1 Introduction

This handout systematically introduces the syntax and programming constructs of Lisp. We will review them briefly in the lectures but they should be self-explanatory and help you work through the lab classes.

2 Syllabus

- Idea behind functions and assigning them to variables
- Building blocks of lisp: s-expressions
- Processing lists: Read-eval-print loop
- Defining procedures
- List processing functions
- Equality functions
- The if choice function

3 Learning objectives

At the end of this unit you should be able to:

- understand the fundamental programming principles underlying lisp
- understand how lists represent knowledge structures
- run gcl and load programs into it
- apply some built-in list-processing functions
- generate your own functions

4 References

- Paul Graham (1996). *ANSI Common Lisp*.
- Peter Seibel (2005). *Practical Common Lisp*.
- George F Luger (2005). *Artificial intelligence: Structures and strategies for complex problem solving*.
- Winston & Horn (1989, 3rd edition). *Lisp*.
- Graham, P. (1996). *ANSI Common Lisp*. Prentice Hall.
- See BlackBoard for additional information.

Note: This document shows dialogue with the lisp interpreter in courier font. The "greater-than" sign is the prompt given by the Clisp interpreter.

5 Lisp is functional

Its syntax and philosophy is based on the idea of recursive functions.

Functions return a value when they execute.

The value can thus be assigned to a variable directly, as opposed to indirectly with procedures which change variable values by side effects.

```
(first '(a b c))
```

- first is a function which returns the first element of a list
- Hence it evaluates to *a*.
- If the function was assigned to a variable, the variable would take on the value of *a*.
- *setf* is a lisp function which assigns the value of the second argument to the symbol given by the first argument. Stands for SET Field.

```
>(first '(a b c))
A

>(setf var (first '(a b c)))
A

>var
A
```

6 Syntax

Based on symbolic expressions or *s-expressions*.

6.1 s-expression

Consists of either an atom or a list.

6.1.1 atom

Synctatic units of Lisp

Include both numbers and symbols

Symbolic atoms are composed of letters, numbers, and some non-alphanumberic characters.

Examples:

- 999
- all-my-money
- nil
- *

nil is the only atom which is also considered to be a list: the empty list.

6.1.2 List

Sequence of atoms or other lists.

Sequence separated by blanks.

```
(atom (list) (list (list-in-a-list (list-in-a-list-in-a-list)
 arbitrarily-complex)))
```

6.2 Formal definition of an s-expression

From Luger and Stubblefield.

- An atom is an s-expression
- If S1, S2, ..., Sn are s-expressions, then so is the list (S1 S2 ... Sn).

Recursive definition.
A list is a nonatomic s-expression

6.3 Programs and data are both represented as s-expressions

List syntax exactly the same for programs and data

```
(+ 2 4)
```

List like any other but is also a program: add two numbers and return the result.

7 Read-eval-print loop

```
>(+ 2 4)
6
```

1. Expression typed in at prompt
2. First element of list read by interpreter
3. Assumes it is a function and attempts to apply the function with the rest of the list as the arguments.
4. Applying the function evaluates it
5. Result is printed.

Arguments are evaluated first and then the result applied to the function.

```
(+ (* 4 (/ 12 3)) (/ 6 2))
```

Answer: 19

8 Forms

Forms are s-expressions for evaluation.
Evaluation procedure;

1. If s-expression is a number, return the number
2. If s-expression is an atomic symbol, return the value bound to that symbol, or an error if there is no value
3. If s-expression is a list, evaluate the second through to the last element and pass the results to the function represented by the first argument.

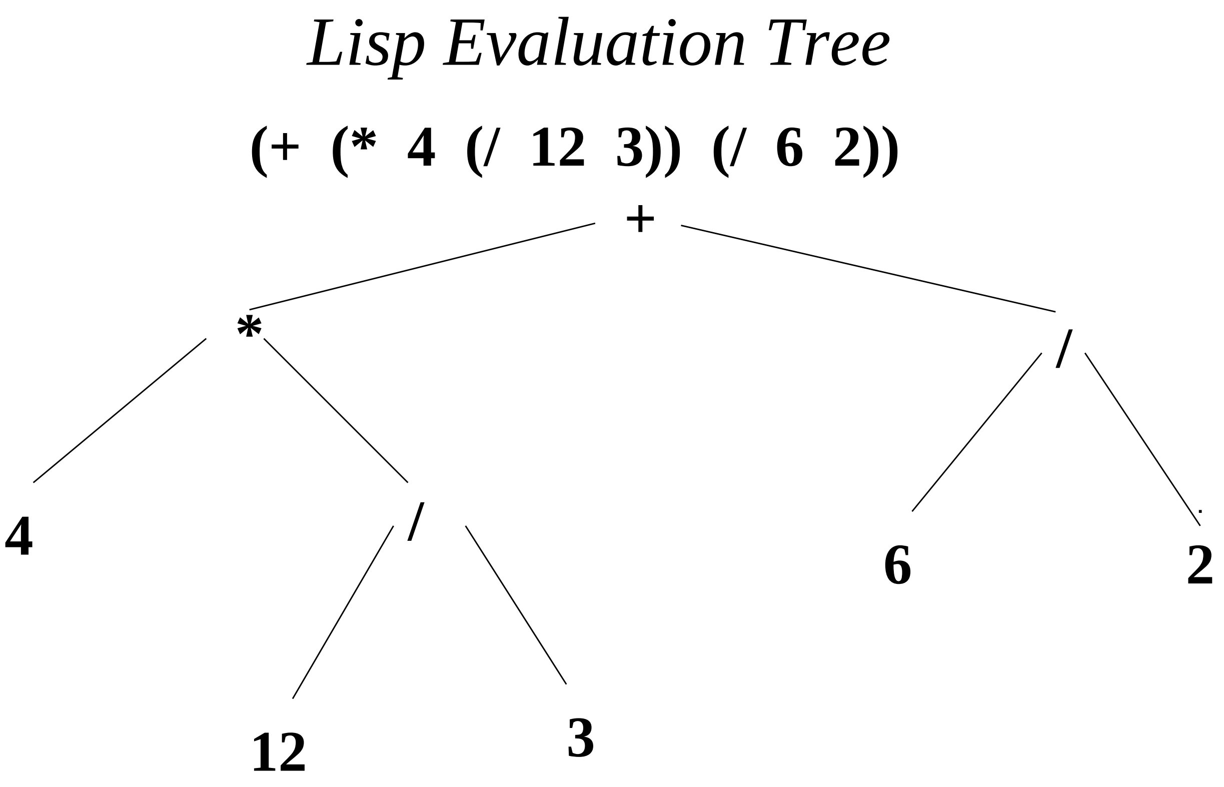

Figure 1: *Evaluation tree for a lisp function*

9 Lists as data, not a program

Often want to use a list as a piece of data rather than evaluating it.
e.g.

```
(jagger (occupation rock-star) (age 58) (sex male))
```

If we type this straight into the interpreter:

```
>(jagger (occupation rock-star) (age 58) (sex male))

Error: The function JAGGER is undefined.
Fast links are on: do (si::use-fast-links nil) for debugging
Error signalled by EVAL.
Broken at EVAL.  Type :H for Help.
>>
```

Evaluation of data is prevented by using a lisp function called *quote*.

9.1 Quote

```
>(quote (+ 4 5))
(+ 4 5)

>(quote (jagger (occupation rock-star) (age 58) (sex male)))
(JAGGER (OCCUPATION ROCK-STAR) (AGE 58) (SEX MALE))
```

This is so frequently required that a short-cut notation for it is the apostrophe.

```
>(quote (+ 4 5))
(+ 4 5)

>'(+ 4 5)
(+ 4 5)
```

Hence to attach a given list as a value to a variable using setf, we need to stop the list being evaluated by using the quote function.

Effectively, the interpreter *is* evaluating the argument by applying quote to it and returning its own value.

```
>(setf all-my-money '((form (1 6 3)) (fitness (24 4))
                     (trainer ((name pipe)(strike-rate 21)))))
((FORM (1 6 3)) (FITNESS (24 4))
 (TRAINER ((NAME PIPE) (STRIKE-RATE 21))))

>all-my-money
((FORM (1 6 3)) (FITNESS (24 4))
 (TRAINER ((NAME PIPE) (STRIKE-RATE 21))))
```

Note that *setf* was a function that did more than just return a value. It also assigned a value to a symbol.

This effect of setf continued after its termination.

This is a side effect of the function.

Strictly speaking, a procedure that only returns a value is called a function but one that both returns a value and causes some persisting change in the program is called a procedure.

10 Defining your own procedures

defun allows you to define your own procedure.

Recall the program to find the number of matches needed for a knockout tournament with any number of players:

```
(defun matches (players)
    (- players 1))
```

defun is a function where the second member of the list is the function name, the third member is the list of parameters and the fourth is a form (i.e. an s-expression for evaluation).

```
(defun matches        ;; procedure name
    (players)         ;; parameters
  (- players 1))      ;; form which defines procedure steps
```

The general case is given by:

```
(defun <procedure name>
       (<parameter 1>..<parameter n>) ;; many parameters
       <form 1>...<form n>)           ;; many forms
```

In Lisp, all procedures act as functions by returning a value.

The value is the last step in the evaluation of the form.

Hence the *matches* function will return the value given by the subtraction sum.

```
>(defun matches (players) (- players 1))
MATCHES

>(matches 50)
49
```

11 Defining your own "language"

Functions to manipulate the knowledge structures.
You will want to retrieve the trainer's name many times.
Write an accessor function to do it.

11.1 Building the code for a new function

develop and test
bit by bit.

For a function to get the trainer name, it comes down to some simple steps:

```
(setf all-my-money '((form (1 6 3)) (fitness (24 4))
            (trainer ((name pipe)(strike-rate 21)))))
```

If we number what the lisp code in each step returns, we can easily see how the code builds into the function we want.

1. Get the trainer information

   ```
   > (assoc 'trainer all-my-money)

    (TRAINER ((NAME PIPE) (STRIKE-RATE 21)))
   ```

2. From the trainer information (1), get the trainer association list

   ```
   > (second (assoc 'trainer all-my-money))

    ((NAME PIPE) (STRIKE-RATE 21))
   ```

3. From what was returned by (2) above, get the name property

   ```
   > (assoc 'name (second (assoc 'trainer all-my-money)))

    (NAME PIPE)
   ```

4. Return the value of the name property from what (3) returns

   ```
    (second (assoc 'name
                   (second (assoc 'trainer all-my-money))))

   PIPE
   ```

- We know that the function returns the desired information but...
- it must operate on a list with the right structure.
- the specific horse, all-my-money, must be given a generic "horse" list structure.
- So our function will work on any horses that follow this structure.
- Lets now define the function we need, where the parameter will be a horse structure.

```
(defun get-trainer-name (horse)
  (second (assoc 'name (second (assoc 'trainer all-my-money)))))
```

- But this isn't right, yet
- our code was tested on the specific horse, all-my-money.
- We need it to work on *any* horse
- We must change the variable `all-my-money` to the parameter name

```
(defun get-trainer-name (horse)
  (second (assoc 'name (second (assoc 'trainer horse)))))
```

- Now, we can apply the function to any horse
- Call the function with the specific horse's name.

```
> (get-trainer-name all-my-money)

PIPE
```

What about a different horse?

```
(setf desert-orchid
   '((form (1 1 1)) (fitness (24 1))
            (trainer ((name elsworth)(strike-rate 100)))))


(get-trainer-name desert-orchid)

ELSWORTH
```

12 List processing functions

Because lists are the fundamental data structure in Lisp, it is important to be able to manipulate them easily. Hence there is a rich set of tools for creating lists, accessing the information contained within them, and altering that information.

```
>(setf l'(1 2 3))
(1 2 3)

>(third l)
3

>(setf l '(1 2 3))
(1 2 3)

>l
(1 2 3)

>(first l)
1

>(last l)
(3)

>(second l)
2
```

```
>(third l)
3

>(rest l)
(2 3)

>(rest '(1))
NIL

>(first nil)
NIL

>(rest nil)
NIL
```

12.1 cons, append, and list

cons returns a list with the first argument at the head of the list represented by the second argument:

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

Short for **CONS**truct.
append takes any number of lists as arguments and combines them into a single list:

```
>(append '(1 2 3) '(4 5 6) '(7 8 9))
(1 2 3 4 5 6 7 8 9)
```

list makes a list out of its arguments as opposed to combining lists into one, as append does:

```
>(list 1 2 3 '(a b) 'c)
(1 2 3 (A B) C)
```

None of these functions change their arguments.
That is, whatever lists are given as parameters remain unaltered by the functions.
Note that setf can bind more than one variable at once in pairs:

```
>(setf list-1 '(1 2 3) list-2 '(4 5 6))
(4 5 6)

>(setf list-3 (append list-1 list-2))
(1 2 3 4 5 6)

>list-1
(1 2 3)

>list-2
(4 5 6)

>list-3
(1 2 3 4 5 6)
```

So to create a new list made up of appending two old lists, you need to do it using *setf*.

12.2 Assoc

Crucial list accessing function
Operates on “association lists” which are lists of lists where each member list has just two elements.

```
>(setf jagger '((occupation rock-star) (age 58) (sex male)))
((OCCUPATION ROCK-STAR) (AGE 58) (SEX MALE))
```

assoc lets you retrieve any one of the sub-lists using the first member of the sub-list as the key:

```
>(assoc 'age jagger)
(AGE 58)
```

General operation is as follows:

```
(ASSOC <key> <association list>)
```

Now beginning to see how lists can hold knowledge and lisp functions can be used to access that knowledge.

12.3 Length

Returns the length of a list:

```
>(length list-3)
6
```

12.4 Reverse

```
>(reverse list-3)
(6 5 4 3 2 1)

>list-3
(1 2 3 4 5 6)
```

So reverse does not alter the list.

12.5 let

This is a function which allows you to declare local variables. Double-list is a local variable which only exists within the let statement.

```
(let ((<parameter 1><initial value 1>)
      ...
      (<parameter n><initial value n>)
  <form 1>
  ...
  <form n>))
```

13 Global and local variables

Global variables differ from local ones by remaining in the lisp environment after a program has finished executing. They should be distinguished from local ones by having an asterisk in front of their name: *<*global-variable*> and should be used **as rarely as possible**.

EXTREMELY IMPORTANT: Do not allow any procedure to alter the value of a global variable during execution. This is very bad programming practice and should *only* be allowed under exceptional circumstances with watertight reasons.

legitimate uses are for:

1. very large knowledge structures that can't be passed as parameters to a function (stack problems can result)
2. large knowledge structures where it is easier and more efficient to surgically alter the list rather than copy it and change the copy.
3. MORE ON THIS LATER

Global variables can often be created by mistake. e.g. simply typing

```
>(setf global-by-mistake 999)
999

>global-by-mistake
999
```

will set up *global-by-mistake* as a permanent variable within the lisp interpreter. You can tell if it is a global by typing its name directly into the interpreter and seeing what happens. Globals will return whatever value they are bound to whereas locals will not persist and an error message will be returned.

NOTE: Global variables existing by mistake can cause bizarre errors to occur with your program so beware of this. If you suspect such a problem, exit lisp, go back in again, and then reload your program (assuming it is not your file itself which has introduced the global..they are usually created by mistake when typing into the interpreter directly.

Using "let" stops the variable persisting:

```
>(let ((global-by-mistake nil))
  (setf global-by-mistake 999))
999

>global-by-mistake

Error: The variable GLOBAL-BY-MISTAKE is unbound.
Fast links are on: do (si::use-fast-links nil) for debugging
Error signalled by EVAL.
Broken at EVAL.  Type :H for Help.
>>
```

13.1 If you must change global variables

These are ways of altering global variables, in order of best practice, for a function called "change-global":

1. (setf global change-global)
   (a) The global variable is visibly seen in the code as being set to a value returned by the function.
2. (change-global global)
   (a) global is passed into the function so you can see that the function is relevant to that global.
   (b) In general, parameters should *not* be changed by the function
   (c) If the function *is* changing the parameter, *explain this effect with a comment before the function definition*
   (d)
   ```
   ;; This function changes the value of the
   ;; parameter called global
   ;;
   ;; ---- lots of other comments explaining
   ;; ---- the function's general purpose and
   ;; ---- behaviour
   ;;
   (defun change-global (global)
     ----
     ----)
   ```
3. (change-global)
   (a) global parameter is changed but there is no way of knowing from the function call that it operates on the variable called global.
   (b) causes major debugging problems if the global is being changed incorrectly
   (c) you won't easily find the functions that cause the change

14 dolist

Function which allows you to apply operations to each member of the list.

```
(dolist (var list-form)
  body-form*)
```

1. `var` is the variable name that has each element of the list returned by `list-form` passed to it.
2. `dolist` is a very rare function in that it allows a variable to be set up and used without being declared first.
3. the variable only has a scope within the body-form of the function.
4. `list-form` must return a list.
5. `body-form` defines what happens to each member of the list: it operates on `var` in some way.

```
;;
;;
(defun double (l)
  (let ((double-list nil))
    (dolist (element l double-list)
            (setf double-list (cons (* 2 element) double-list)))))
```

Note:

1. element is the variable name given to each member of the list as the list is processed
2. l is the list being processed
3. double-list is the result to be returned by the dolist function
4. If there is no result form, then the dolist function returns nil

Let's see how it works:

```
>(double '(1 2 3))
(6 4 2)
```

If we leave off the result form (i.e. double-list) so that the first line looks like *dolist (element l)*, then dolist and thus also the function returns nil.

15 Boolean operators

Equality operators (Table adapted from Winston & Horn).

equal	true if arguments are the same expression
eql	true if arguments are the same symbol or number
eq	true if arguments are the same symbol
=	true if arguments are the same number

Table 1: Equality predicates

Each version tests equality first by the applying the one below and then applying itself. For example:
"equal" first applies "eql" and, if that fails, it tests to see if its arguments are lists with the same members.
"eql" first tries '"eq" and, if that fails, tests to see if its argument are the same symbol or number.
And so on.

15.1 Member

Returns the rest of the list with the element at the head.

```
>(member 't '(i n t e l l i g e n t))
(T E L L I G E N T)
```

In Lisp, nil is false and anything else is taken to be true.

```
>(equal (* 4 2) 8)
T

>(equal (* 4 2) 9)
NIL

>(member 't '(i n t e l l i g e n t))
(T E L L I G E N T)

>(member 'u '(i n t e l l i g e n t))
NIL
```

16 The *if* choice function

```
(if <test> <then form> <else form>)

(defun biggest (x y)
  (if (> x y)
      x
    y))

>(biggest 3 5)
5

>(biggest 5 3)
5
```

Note that there is only one form for the if part and one form for the else part. So if you want to carry out multiple steps, you have to use a function which allows one form to consist of many subforms.

```
;; Note: else part of the if statement consists of two forms, not just
;; the one allowed.
;;
(defun bigger (x y)
  (if (> x y)
      (format t "~%first argument is bigger")
    (format t "~%second argument is bigger")
    (format t "~%and here is a second form for the else bit~%")))

>(bigger 3 4)

Error: Too many arguments.
Error signalled by IF.
Broken at IF.  Type :H for Help.
```

The function required is *progn* which can contain any number of forms:

```
(defun bigger (x y)
  (if (> x y)
      (format t "~%first argument is bigger")
    (progn (format t "~%second argument is bigger")
           (format t "~%and here is a second form for the else bit~%"))))

>(bigger 3 4)

second argument is bigger
and here is a second form for the else bit
NIL
```

progn returns the value of its **last** form whereas a similar function called *prog1* returns the value of its first form. So if we add a second form to the “then” part of the if statement using progn and make this form just a single number, it should be returned by the progn:

```
(defun bigger (x y)
  (if (> x y)
      (progn (format t "~%first argument is bigger")
             1)
    (progn (format t "~%second argument is bigger")
           (format t "~%and here is a second form for the else bit~%"))))

;; now call it in lisp

>(bigger 4 3)

first argument is bigger
1
```