Common Lisp

Introduction

- LISP is an acronym for **LIS**t **P**rocessor.
- In the 1980s there was an attempt to standardize the language. The result is Common LISP (CLISP).
- LISP is usually used as an interpreted language.
- The interpreter runs what is known as a *read-eval-print* loop. That is, it *reads* what you type, *evaluates* it, and then *prints* the result, before providing another prompt.

Cont.

- The two most important kinds of objects in LISP for you to know about are atoms and lists.
- Atoms are represented as sequences of characters of reasonable length. Such as : 34 or join.
- Lists are recursively constructed from atoms. Such as: (a john 34 c3po).
- The interpreter treats any list as containing the name of a function followed by the arguments to the function. Such as: (+ 2 13 45).

Cont.

- These are some primitive functions: +, -, *, /, exp, expt, log, sqrt, sin, cos, tan, max, min.
- If you want the list to be treated as list of atoms without being evaluated, use the single quote mark, ', in front of an item that you do not wish the interpreter to evaluate.
- There are many functions for list manipulation, such as cons, list, append, first, rest

List

- cons :
  - **Format:** (cons <exp1> <exp2>)
- cons creates a new copy of the list returned by <expr2>, and makes the value returned by <expr1> the new first element in this list. However, if <expr2> returns an atom, cons returns the *dotted pair* of the values of the two expressions.
- Example:

```
> (cons 'a '(1 2 3))
  (A 1 2 3)
> (cons 'a 3)
  (A . 3)
```

Cont.

- list :
  - **Format:** (list <exp1> <exp2>...<expN>)
- All the <arg>'s are evaluated and the resulting values are returned as elements of a list.
- Example:

```
> (list 'picard 'riker 'worf 'crusher)
  (PICARD RIKER WORF CRUSHER)
> (list 1 (+ 1 1) (+ 1 1 1) (+ 1 1 1 1))
  (1 2 3 4)
```

Cont.

- append :
  - **Format:** (append <list1> <list2>...<listN> <exp>)
- Each of the arguments is evaluated; all except the last must return a list. If all the arguments evaluate to lists, append creates a new list which has as its elements the elements of all the argument lists. If the last argument is not a list, append returns a dotted pair object.
- Example:

```
> (append '(a b) '(c d))
  (A B C D)
> (append '(1 (2 (3))) (i (j) k))
  (1 (2 (3)) I (J) K)
> (append 'a '(1 2))
  Error: A is not of type LIST
```

Cont.

- first :
  - **Format:** (first <exp>)
- The argument expression must evaluate to a list; first returns the first element of this list. If the list is empty, i.e. is nil, first returns nil.
- Example:

```
> (first '(1 2 3))
  1
> (first '((a (b (c)) d) e (f)))
  (A (B (C)) D)
> (first ())
  NIL
```

Cont.

- rest :
  - **Format:** (rest <exp>)
- The argument expression must evaluate to a list; rest returns a list of all elements in the input except the first one element. If the list is empty, i.e. is nil, rest returns nil.
- Example:

```
> (rest '(1 2 3))
  (2 3)
> (rest '((a (b (c)) d) e (f)))
  (E (F))
> (rest ())
  NIL
```

Cont.

- butlast :
  - **Format:** (butlast <list>) (butlast <list> <int>)
- If butlast is used with a single argument then it will return the list argument with the last element removed.
- If butlast is given an integer second argument, it will remove the number of elements specified from the end of the list.
- Example:

```
> (butlast '(a s d f))
  (A S D)
> (butlast '(a s d f) 2)
  (A S)
> (butlast '(a s d f) 0)
  (A S D F)
```

Cont.

- reverse :
  - **Format:** (reverse <list>)
- Reverse returns a list that contains all the elements of <list> in reversed order.
- Example:

```
> (reverse '(picard riker worf crusher))
  (CRUSHER WORF RIKER PICARD)
> (reverse (reverse '(picard riker worf crusher)))
  (PICARD RIKER WORF CRUSHER)
> (reverse '((this list) (of words)))
  ((OF WORDS) (THIS LIST))
```

Cont.

- car :
  - **Format:** (car <exp>)
- car is an archaic form of the function first and the two may be used interchangeably. (Historical note: "CAR" is an acronym for "Contents of Address Register" which refers to the way in which linked lists were originally implemented in Lisp.).

Cont.

- cdr :
  - **Format:** (cdr <exp>)
- cdr is an archaic form of the function rest and the two may be used interchangeably. (Historically its name is derived from Contents of Decrement Register. Its pronunciation rhymes with udder.).

Cont.

- caar, cadr, cdar, cddr, etc :
  - **Format:** (c-r <list>)
- The function *cxyr* is a composition of the function *cxr* with *cyr*. So, for example, (cadr foo) is equivalent to (car (cdr foo)), etc. Up to 3 letters, a or d, may appear between the c and the r.
- Example:

```
> (cadr '(Sisko Kira Dax Odo Bashir OBrien))
  KIRA
> (cddr '(Sisko Kira Dax Odo Bashir OBrien))
  (DAX ODO BASHIR OBRIEN)
> (cddar '(Sisko Kira Dax Odo Bashir OBrien))
  Error: SISKO is not of type LIST.
```

Cont.

- length :
  - **Format:** (length <exp>)
- <exp> must evaluate to a sequence (e.g. list, array, vector, string). Returns the length of the given sequence.
- Example:

```
> (length '(1 2 3 4 5))
  5
> (length "1 2 3 4 5")
  9
> (length (make-array 3))
  3
```

Cont.

- listp :
  - **Format:** (listp <exp>)
- Returns T if <exp> is of the data type list; NIL otherwise.
- Example:

```
> (listp '(a s d f))
  T
> (listp 3)
  NIL
> (listp (cons '1 '(2 3 4)))
  T
```

Cont.

- atom :
  - **Format:** (atom <exp>)
- <exp> can be any LISP expression
- The argument expression is evaluated. If the value returned by this evaluation represents a LISP atom, atom returns T, else it returns NIL. Symbols, numbers and strings are all considered LISP atoms.
- Example:

```
> (atom 'hello)
  T
> (atom "hello")
  T
> (atom 4.6434)
  T
> (atom '(hello "hello" 4.6434))
  NIL
```

Cont.

- nth :
  - **Format:** (nth <index> <list>)
- The function nth returns the indexed element of <list>. <index> must be a non-negative integer. 0 indicates the first element of <list>, 1 the second, etc. An index past the end of the list will cause nth to return nil.
- Example:

```
> (nth 0 '(picard riker work crusher))
  PICARD
> (nth 2 '((captain picard)
           (commander riker)
           (lieutenant worf)
           (ensign crusher)))
  (LIEUTENANT WORF)
```

Cont.

- nthcdr :
  - **Format:** (nthcdr <index> <list>)
- The function nth returns the <list> with the first n elements removed. <index> must be a non-negative integer. An index past the end of the list will cause nthcdr to return nil.
- Example:

```
> (nthcdr 0 '(Sisko Kira Dax Odo Bashir OBrien))
  (SISKO KIRA DAX ODO BASHIR OBRIEN)
> (nthcdr 1 '(Sisko Kira Dax Odo Bashir OBrien))
  (KIRA DAX ODO BASHIR OBRIEN)
> (nthcdr 3 '(Sisko Kira Dax Odo Bashir OBrien))
  (ODO BASHIR OBRIEN)
> (nthcdr 2345 '(Sisko Kira Dax Odo Bashir OBrien))
  NIL
```

Cont.

- second, third, etc :
  - **Format:** (second <list>) (third <list>) etc.
- These functions return the obvious element from the given list, or nil if the list is shorter than the selected element would require.
- Example:

```
> (second '(1 2 3 4))
  2
> (fourth '(1 2 3 4))
  4
> (ninth '(1 2 3 4))
  NIL
```

Cont.

- member :
  - **Format:** (member <item> <list> :test <test> :test-not <test-not> :key <key>)
- <test>/<test-not>: A function or lambda expression that can be applied to compare <item> with elements of <list>. <key>: A function or lambda expression that can be applied to elements of <list>.
- The elements of <list> are compared with the <item>. If <test> is not specified, eq is used; otherwise <test> is used. If <item> is found to match an element of <list>, a list containing all the elements from <item> to the end of <list> is returned. Otherwise NIL is returned. If <test-not> is specified, member returns a list beginning with the first UNmatched element of <list>. Specifying a <key> causes member to compare <item> with the result of applying <key> to each element of <list>, rather than to the element itself.

Cont.

- max, min :
  - **Format:** (max <num1> ... <numN>) (min <num1> ... <numN>)
- Returns the numerical maximum (minimum) of the arguments given.
- Example:

```
> (max 1 4 3 15 (* 9 2))
  18
> (min 3 4 (- 7 19) 5 6.0)
  -12
> (max 3)
  3
```

setq and setf

- Setq is useful for changing the values of variables. Such as:

```
>(setq a ‘(1 2 3))
 (1 2 3)
```

- But sometimes one would like to change just part of the value of a variable, setf is what you need. Such as:

```
>(setf (second a) '4)
 4
>a
 (1 4 3)
```

Defining Lisp Function

- Use defun to define your own functions in LISP.

  (defun <name> <parameter-list> <body>)
- Example:

```
>(defun square (x) (* x x))
  SQUARE
>(square 2)
  4
```
- The name of a user-defined function can be any symbol. (Recall: A symbol is any atom that is not a number.)
- It is even possible to redefine LISP's predefined functions such as first, cons, etc. Avoid doing this!

Cont.

- More advanced programming allows the use of &rest, &optional, &key in the parameter list to permit variable numbers of arguments.
- The body of a function can be a single LISP instruction, or it can be an indefinitely long set of instructions.
- The value of the last instruction will be used as the return value.

Cont.

- Parameters are treated as local variables in the function.
- Local variables disappear when the function that uses them is done.

Conditional Control

- There are two:
  - If

    Format: (if <test> <then> <else>)
  - Cond

    Format:

    (cond (<testa> <form1a> <form2a> ... <resulta>)
    (<testb> <form1b> <form2b> ... <resultb>)
    ...
    (<testk> <form1k> <form2k> ... <resultk>))

Cont.

- if :
  - **Format:** (if <test> <then> <else>)
- The test, then, and else expressions can be any evaluable Lisp expressions -- e.g., symbols or lists.
- If the evaluation of the test expression returns anything other than nil, the interpreter evaluates the then expression and returns its value, otherwise it returns the result of evaluating the else expression.
- Example:

```
> (defun absdiff (x y)
        (if (> x y)
            (- x y)
            (- y x))
   )
```

Cont.

- cond :
  - **Format:** (cond (<testa> <form1a> <form2a> ... <resulta>)
    (<testb> <form1b> <form2b> ... <resultb>)
    ...
    (<testk> <form1k> <form2k> ... <resultk>))
- We will call each of the lines (**<**test**>**.....**<**result**>**) a ``cond clause'', or sometimes just “clause” for short.
- In cond clauses, only the test is required, but most commonly cond clauses contain just two elements: test and result.

Cont.

- Instead of nesting if statements as before, the set of conditions (if A B (if C D E)) can be expressed using cond. It would look like this:

```
(cond (A B)
      (C D)
      (t E))
```

- If the evaluation of the test expression at the beginning of a clause returns nil, then the rest of the clause is not evaluated. If the test returns a non-nil value, all the other expressions in that clause are evaluated, and the value of the last one is returned as the value of the entire cond statement.
- Cond and if statements are most powerful in defining functions that must repeat some step or steps a number of times.

Equality Predicates

- Common LISP contains a number of equality predicates. Here are the four most commonly used:
  - **=** : (= x y) is true if and only x and y are numerically equal.
  - **equal** : As a rule of thumb, (equal x y) is true if their printed representations are the same (i.e. if they look the same when printed). Strictly, x and y are equal if and only if they are structurally isomorphic, but for present purposes, the rule of thumb is sufficient.
  - **eq** : (eq x y) is true if and only if they are the same object (in most cases, this means the same object in memory).
  - **eql** : (eql x y) is true if and only if they are either eq or they are numbers of the same type and value.
- Generally = and equal are more widely used than eq and eql.

Cont.

- Here are some examples involving numbers:

```
>(= 3 3.0)
  T
>(= 3/1 6/2)
  T
>(eq 3 3.0)
  NIL
>(eq 3 3)
  T or NIL (depending on implementation of Common LISP)
```

Cont.

```
>(eq 3 6/2)
  T
>(eq 3.0 6/2)
  NIL
>(eql 3.0 3/1)
  NIL
>(eql 3 6/2)
  T
>(equal 3 3)
  T
>(equal 3 3.0)
  T
```

Checking for NIL

- null, not :
  - **Format:** (null <exp>) (not <ex>).
- The predicates null and not act identically. Good programming style dictates that you use null when the semantics of the program suggest interest in whether a list is empty, otherwise use not:
- Example:

```
> (null nil)
  T
> (not nil)
  T
> (null ())
  T
> (not ())                    ;;preferable to use null
  T
```

Logical Operator

- and and or are functions but not predicates since they may return values other than t or nil.
- and returns nil as soon as it finds an argument which evaluates to nil; otherwise it returns the value of its last argument.
- Or returns the result of its first non-nil argument, and does not evaluate the rest of its arguments. If all the arguments evaluate to nil, then or returns nil.

Cont.

- Examples

```
>(and 1 2 3 4)
  4
>(and 1 (cons 'a '(b)) (rest '(a)) (setf y 'hello))
  NIL
>y
  27
>(or nil nil 2 (setf y 'goodbye))
  2
>(or (rest '(a)) (equal 3 4))
  NIL
```

Recursion

- The distinctive feature of a recursive function is that, when called, it may result in more calls to itself. For example:

```
(defun power (x y)
        (if (= y 0)
            1
            (* x (power x (1- y)))
        )
)
```

- Every call for the function has its own local variables.
- Any recursive function will cause a stack overflow if it does not have a proper termination condition.

Cont.

- LISP provides a nice way to watch the recursive process using trace. For example:

```
>(trace power)
  POWER
>(power 3 4)
    1> (POWER 3 4) ;; Your actual output
            2> (POWER 3 3) ;; may vary in format
                    3> (POWER 3 2)
                            4> (POWER 3 1)
                                    5> (POWER 3 0)
                                    <5 (POWER 1)
                            <4 (POWER 3)
                    <3 (POWER 9)
            <2 (POWER 27)
    <1 (POWER 81)
  81
```

- If you want to turn trace off, then (untrace power)

Iteration

- There are different ways of iteration, such as:
  - Iteration Using Dotimes
  - Iteration Using Dolist
  - Iteration Using loop
  - Iteration Using do

Cont.

- dotimes :
  - **Format:** (dotimes (<counter> <limit> <result>) <body>)
- Counter is the name of a local variable that will be initially set to 0,then incremented each time after body is evaluated, until limit is reached; limit must, therefore, evaluate to a positive integer. Result is optional. If it is specified, then when limit is reached, it is evaluated and returned by dotimes. If it is absent, dotimes returns nil. The body of a dotimes statement is just like the body of a defun -- it may be any arbitrarily long sequence of LISP expressions.

Cont.

- dolist :
  - **Format:** (dolist (<next-element> <target-list> <result>) <body>)
- Dolist is very much like dotimes, except that the iteration is controlled by the length of a list, rather than by the value of a count.
- In a dolist statment result and body work just as in dotimes. Next-element is the name of a variable which is intially set to the first element of target-list (which must, therefore be a list). The next time through, next-element is set to the second element of target-list and so on until the end of target-list is reached, at which point result-form is evaluated and returned.

Simple Data Structures in LISP

- There are many simple data structure, such as:
  - Association Lists
  - Property Lists
  - Arrays, Vectors, and Strings
  - structure

Association Lists

- An association list is any list of the following form:

  ((<key1> ...<expressions>)
  (<key2> ...<expressions>)....)

- The keys should be atoms. Following each key, you can put any sequence of LISP expressions.
- For example:

```
>(setf person1 '((first-name john)
                 (last-name smith)
                 (age 23)
                 (children jane jim)))
((FIRST-NAME JOHN) (LAST-NAME SMITH) (AGE 23) (CHILDREN JANE
   JIM))
```

Cont.

- LISP provides a function, assoc, to retrieve information easily from association lists given a retrieval key.
- For example:

```
>(assoc 'age person1)
 (AGE 23)
>(assoc 'children person1)
 (CHILDREN JANE JIM)
```

- Setf can be used to change particular values.
- Assoc will return nil if the key is not found.
- it is very easy to add new key-expression sublists, again using setf. For example:

```
>(setf person1 (cons '(sex male) person1))
 ((SEX MALE) (FIRST-NAME JOHN) (LAST-NAME SMITH) (AGE 24)
  (CHILDREN JANE JIM))
```

Property Lists

- An alternative way to attach data to symbols is to use Common LISP's property list feature. For each symbol, the LISP interpreter maintains a list of properties which can be accessed with the function get.

```
>(get 'mary 'age)
  NIL
>(setf (get 'mary 'age) 45)
  45
>(get 'mary 'age)
  45
```

- Additional properties can be added in the same way.
- If, for some reason, you need to see all the properties a symbol has, you can do so:

```
>(symbol-plist 'mary)
```

Arrays and Vectors

- An array is a special type of data object in LISP. Arrays are created using the make-array function.
- To make an array it is necessary to specify the size and dimensions.
- The simplest case is an array of one dimension, also called a vector.

```
>(setf my-vector (make-array '(3)))
  #(NIL NIL NIL)
```

- These elements can be accessed and changed using aref. Indexing of arrays starts with 0.

```
>(aref my-vector 2)
  NIL
```

Cont.

- You may use the :initial-contents keyword to initialize the content.

```
>(make-array '(2 3 4) :initial-contents
                '(((a b c d) (e f g h) (i j k l))
                  ((m n o p) (q r s t) (u v w x))))
#3A(((A B C D) (E F G H) (I J K L)) ((M N O P) (Q R S T)
 (U V W X)))
```

Strings

- A string in LISP is represented by characters surrounded by double quotes: ".

```
>"This is a string“
 "This is a string"
```

- Notice that the string may contain spaces, and that the distinction between upper and lowercase letters is preserved. A string is completely opaque to the interpreter and may contain punctuation marks and even new lines.

Structure

- Defstruct allows you to create your own data structures and automatically produces functions for accessing the data.
- Structures have names and “slots.” The slots are used for storing specific values.

```
>(defstruct employee
            age
            first-name
            last-name
            sex
            children)
EMPLOYEE
```

Cont.

- Defstruct automatically generates a function to make instances of the named structure. In this example the function is called make-employee, and in general the name of the instance constructor function is make-defstructname.
- Each slot is provided an automatic access function, by joining the structure name with the slot name:

```
>(employee-age employee1)
  NIL
```

Cont.

- It is also possible to assign values to the slots of a particular instance at the time the instance is made, simply by preceding the slot name with a colon, and following it with the value for that slot:

```
>(setf employee2 (make-employee
                                    :age 34
                                    :last-name 'farquharson
                                    :first-name 'alice
                                    :sex 'female))
#S(EMPLOYEE AGE 34 FIRST-NAME ALICE LAST-NAME
  FARQUHARSON SEX FEMALE CHILDREN NIL)
```

- Defstruct also allows you to specify default values for given slots. Here is an example:

```
>(defstruct trekkie (sex 'male) (intelligence 'high) age)
  TREKKIE
```

Input and Output

- Terminal input and output is controlled with variants of print and read. More sophisticated output is available using format. Input and output using system files is achieved using the same functions and associating a file with an input or output “stream”.

Basic Printing

- print, prin1, princ and terpri.
- The simplest uses of print, prin1, and princ involve a single argument. Terpri, which produces a newline, can be called with no arguments.
- All these are functions. In addition to causing output, they return values. With print, princ, and prin1, the value returned is always the result of evaluating the first argument. Terpri always returns nil.

Cont.

- Examples:

```
>(print 'this)
  THIS            ;; printed
  THIS            ;; value returned
>(print (+ 1 2))
   3              ;; printed
   3              ;; returned
>(+ (print 1) (print 2))
   1              ;; first print
   2              ;; second print
   3              ;; returns sum
>(+ (prin1 1) (prin1 2))
   12
   3
```

- Print is thus equivalent to terpri followed by prin1

Cont.

- Princ and prin1 are the same except in the way they print strings. Princ does not print the quote marks around a string:

```
>(prin1 "this string")
  "this string“          ;; printed
  "this string“          ;; returned
>(princ "this string")
   this string           ;; no quotes can be more readable
    "this string“        ;; string returned
```

Format

- For sophisticated and easy to read output, format is more useful than other basic printing.

  (format <destination> <control-string> <optional-arguments>)

- The full use of a destination will be introduced further below; for most basic uses, the destination should be specified as t or nil. The control string is a string containing characters to be printed, as well as control sequences. Every control sequence begins with a tilde: ~. The control sequences may require extra arguments to be evaluated, which must be provided as optional arguments to format.

Cont.

- With t as the specified destination, and no control sequences in the control-string, format outputs the string in a manner similar to princ, and returns nil.

```
>(format t "this")
  this
  NIL
```

- With nil as destination and no control sequences, format simply returns the string.

```
>(format nil "this")
 "this"
```

Cont.

- Inserting ~% in the control string causes a newline to be output.
- ~s indicates that an argument is to be evaluated and the result inserted at that point. Each ~s in the control string must match up to an optional argument appearing after the control string.

```
>(format t "~%This number ~s ~%is bigger than this ~s ~%" 5 3 )

This number 5
is bigger than this 3
NIL
```

Reading

- Input from the keyboard is controlled using read, read-line, and read-char.
- Read expects to receive a well-formed LISP expression, i.e. an atom, list or string. It will not return a value until a complete expression has been entered -- in other words all opening parentheses or quotes must be matched.
- Read-line always returns a string. Read-line will take in everything until the return key is pressed and return a string containing all the characters typed.
- Read-char reads and returns a single character.

Input and Output to Files

- All input and output in Common LISP is handled through a special type of object called a stream.
- When a stream is not specified, Common LISP's default behavior is to send output and receive input from a stream bound to the constant *terminal-io* corresponding to the computer's keyboard and monitor.
- print, format, read and the other functions mentioned allow optional specification of a different stream for input or output

Cont.

- To open a stream use the following:

  (with-open-file (<stream> <filename>) <body>)
- Unless specified otherwise, with-open-file assumes that the stream is an input stream.
- Output stream is:

  >(with-open-file (outfile "foo" :direction :output)
- Or

  >(with-open-file (outfile "foo" :direction :output :if-exists :append)
- Examples:

  (read infile)
  (prin1 '(here is an example) outfile))
  (format outfile "~%This is text.~%"))

Functions, Lambda Expressions, and Macros

- Eval: Eval implements the second stage of the LISP interpreter's read-eval-print loop. Any LISP expression can be passed to eval for immediate evaluation.

```
> (eval '(+ 1 2 3))
6
```

- The Lambda Expression is the heart of Lisp's notion of a function. The term comes from Alonzo Church's ``lambda calculus'' -- a development of mathematical logic. You can think of a lambda expression as an anonymous function. Just like a function it has a list of parameters and a block of code specifying operations on those parameters.

Cont.

- Example:

```
> (setf product '(lambda (x y) (* x y)))
  (LAMBDA (X Y) (* X Y))
> product
  (LAMBDA (X Y) (* X Y))
```

- Lambda expressions can be used in conjunction with apply to mimic function calls:

```
> (apply product '(3 4))
  12
```

Cont.

- FUNCALL is similar to APPLY.
- The MAPCAR form is an "iterator" that applies a function repeatedly, to each element of a list and returns a list of the results. For example:

```
> (MAPCAR 'ATOM '(DOG (CAT HORSE) FISH))
  (T NIL T)
```

- Backquoted lists allow evaluation of items that are preceded by commas.

```
>`(2 ,(+ 3 4))
  (2 7)
> `(4 ,(+ 1 2) '(3 ,(+ 1 2)) 5)
  (4 3 '(3 3) 5)
```

Cont.

- Macro definitions are similar to function definitions, but there are some crucial differences. A macro is a piece of code that creates another lisp object for evaluation. This process is called ``macro expansion''. Macro expansion happens before any arguments are evaluated.

```
> (defmacro 2plus (x) (+ x 2))
  2PLUS
> (setf a 3) ;; setf is a macro too!
  3
> (2plus a)
  5
```