CSC 244/444 Notes Sept. 10, 2002

A Quick Introduction to Common Lisp

Lisp is a *functional* language well-suited to symbolic AI, based on the λ-calculus and with *list structures* as a very flexible basic data type; programs are themselves list structures, and thus can be created and manipulated just like other data.

This introduction isn't intended to give you all the details, but just enough to get across the essential ideas, and to let you get under way quickly. The best way to make use of it is in front of a computer, trying out all the things mentioned.

Basic characteristics of Lisp and Common Lisp:

- Lisp is primarily intended for manipulating symbolic data in the form of list structures, e.g.,
  `(THREE (BLIND MICE) SEE (HOW (THEY RUN)) !)`

- Arithmetic is included, e.g.,
  `(SQRT (+ (* 3 3) (* 4 4)))` or `(sqrt (+ (* 3 3) (* 4 4)))`
  (Common Lisp is case-insensitive, except in strings between quotes `"` ... `"`, specially delimited symbols like `|Very Unusual Symbol!|`, and for characters prefixed with "\").

- Common Lisp (unlike John McCarthy's original "pure" Lisp) has arrays and record structures; this leads to more understandable and efficient code, much as in typed languages, but there is no *obligatory* typing.

- All computation consists of *expression evaluation*. For instance, typing in the above `(SQRT ...)` expression and hitting the enter key immediately gives 5.

- Lisp is a functional language based on Alonzo Church's λ-calculus (which like Turing machines provides a complete basis for all computable functions). For instance, a function for the square root of the sum of two squares can be expressed as
  `(lambda (x y) (sqrt (+ (* x x) (* y y))))`.
  We can save this function by giving it a name using `defun` (examples will be seen soon), or we can apply it directly, as in
  `((lambda (x y) (sqrt (+ (* x x) (* y y)))) base height)`,
  where `base` and `height` are assumed to be variables that have been assigned some numerical values.

- Since data are list structures and programs are list structures, we can manipulate programs just like data. We can also easily define special-purpose list-structured languages (e.g., for planning or for NLP) and write interpreters for these languages in Lisp.

- List structures are a very flexible data type – it's hard to think of any types of information that *can't* be easily represented as list structures. And as long as we're using list structures as our data types, we can enter data just by supplying list structures – we need not write special reading or structure-building functions. For instance,
  ```
  (SETQ X '(THREE (BLIND MICE) SEE (HOW (THEY RUN)) !))
  ```
  sets the variable `X` to the indicated list structure value; and
  ```
  (READ)
  ```
  will read in any list structre. Similarly printing out list structures requires no special printing functions (the functions `PRINT` and `FORMAT` do almost everything we need). Common Lisp arrays and record structures are also pretty easy to handle in I/O.

- The simplicity and elegance of Lisp, and its unmatched flexibility and ease of use for AI applications, have made it very durable: it is the second-oldest high-level programming language (after FORTRAN), yet is still in common use.

Getting started

Go to the directory where you want to work (and where you'll typically have various program files and data files), and type

```
lisp
```

The system will respond with loading and initialization messages, followed by a prompt like

```
CL-USER(1):
```

You can now type in expressions and have them evaluated. Whenever you've got balanced brackets and hit enter, the value (or an error message) is printed, and then you're prompted for further input.

Getting out

To escape from Common Lisp, type

`(exit)`, or `:exit`

Saying "Hello"

According to N. Wirth (author of ALGOLW and Pascal), a good way to get an initial glimpse of a language is to see a program that prints "hello". Here's the Lisp version (just one quoted symbol!):

```
'hello
```

When you hit enter, you'll get

```
HELLO
```

which is the result of evaluating `'hello`. If you want lower case, you can use the character string

```
"hello"
```

which yields `"hello"` (still in quotes) – i.e., strings evaluate to themselves. If you want the quotes to be invisible, you can use

```
(format t "hello")
```

However, besides getting "hello" without quotes you'll now also get `NIL` on a new line; that's because the `format` function is special, having both an effect and a value: the effect is to print out the material in quotes (plus possibly values of variables – see example below), and the value is `NIL`.

But suppose we want a more elaborate greeting, wherein the user is asked for her/his name. Then we could define a hello-function as follows:

```
(defun hello ()
  (format t "Hello, what's your first name?~%")
  (setq *name* (read))
  (format t "Nice to meet you, ~a" *name*) )
```

Note that the first argument of `defun` is a function name, the second is a list of arguments (here, none), and after that we have any number of expressions which will be evaluated in sequence, and which comprise the "body" of the definition. In effect, `defun` just makes a λ-expression out of the argument list and the body, and makes this lambda expression the function-value of the first argument (the function name). In other words, it provides a way of naming a λ-expression. (Note that the function value of a symbol won't interfere with other ordinary values we might associate with it; but generally it's wise to steer away from such multiple uses of a symbol.)

The `~%` in the first format statement means "go to new line". The `setq` expression sets the global value of `*name*` to be whatever Lisp expression is read in by `(read)`. A symbol can have both a global value, and a value local to a particular function call. But it's conventional to use an initial and final "`*`" for global variables, and to avoid using these as local variables. (We'll see local variables in later examples.) The `~a` in the second format statement means "print the value of the next expression given as additional argument of `format`" – and in this case the next argument is `*name*`, and its value is whatever name the user supplied.

When we type in this function definition and hit enter, we get

```
HELLO
```

which is just the name of the function we've defined. (So even an expression that defines a function gets a value.) However, `defun` is primarily evaluated for its effect, not its value, and the effect as indicated is that Lisp will now "remember" the definition of the `hello` function.

To execute the `hello` function, we type

```
(hello)
```

and hit enter, with the result

```
Hello, what's your first name?
Joe      (assumed as user input)
Nice to meet you, JOE
NIL
```

The final `NIL` is the value returned by the `hello` function when it terminates. Of course, we can also use the expression `(hello)` as part of another program, which does not immediately terminate after the greeting.

In general, we can evaluate functions for particular arguments by writing

($\textit{function-name}\ arg_1\ arg_2\ \ldots\ arg_n$)

Since `hello` had no arguments, we wrote `(hello)`. Except for certain functions like `defun`, `setq`, and a few others, *arguments are first evaluated*, and only then is the function applied to the argument values.

Loading Lisp code from a file

If you have placed Lisp code (a series of symbolic expressions) in a file `my-file.lisp`, you can run this code and then continue on-line as follows (assuming you are already in Lisp, having typed `lisp` at some point):

```
(load "myfile.lisp")
```

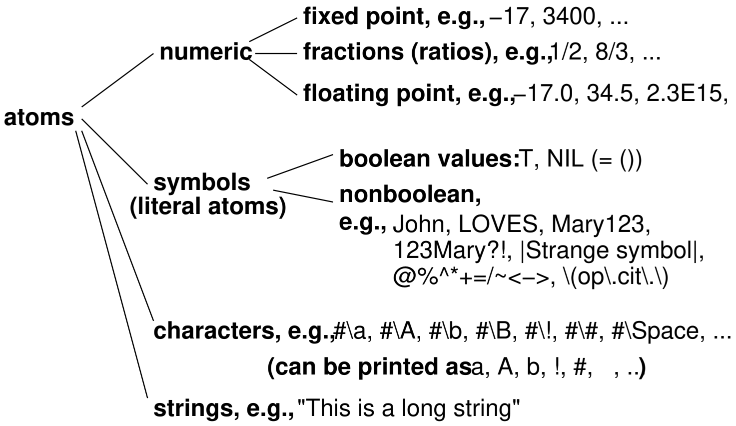
assuming that this file is in the same directory as the one from which you are running Lisp. You can also use the shorter version `(load "myfile")`, i.e., without the "`.lisp`" extension – but this will use a compiled version of the file if one exists. If you are user "`smith`" and the desired file `my-file` is in a directory `lisp-programs`, you could also use

```
(load "/u/smith/lisp-programs/my-file.lisp")
```

(or without the "`.lisp`"), no matter what directory you are currently in. After this is evaluated, you can use functions and constants that were defined in `my-file.lisp`.

Data types

The smallest symbolic expressions are *atoms*, and we build larger expressions out of these. Atoms come in several varieties, as indicated in the following tree:



> As an oddball category of literal atoms we should also mention *keywords*; these start with a colon (e.g., `:key`) and have very special uses. A colon in the middle of an atom also has a special meaning: the part before the colon is a *package* name, and the part after it is the name of a function in the package referred to. But we won't go into that here.

Dotted pairs – discussed below

List structures – e.g.,

```
(3 "blind" mice)
(A B (C D E))
NIL   (same as ())
(this (list (is (nested to (depth 5) see?))))
```

> **Note:** In general, expressions are *evaluated* when they occur as an argument (non-initial list element). However, `T`, `NIL`, numbers, characters, strings and keywords evaluate to themselves. In addition, quotation – e.g., `'THIS` or `'(a b)` – prevents evaluation of the expression quoted (in the two examples we just get back `THIS` and `(A B)`). Also as previously noted, certain special functions prevent evaluation of (some of) their arguments (e.g., `defun`, `setq`, etc.). *Initial* list elements are interpreted as functions, unless the list is inside a quoted context. So for instance `(+ x y)` evaluates to the sum of values associated with `x` and `y`, but `'(+ x y)` just evaluates to the expression `(+ x y)`.

Arrays – e.g., we can define an array `A` using

```
(setq A (make-array '(3 4) :initial-contents
                    '((a b c d) 2 3 4)
                      ("this" 5 6 7)
                      (#\C 8 9 10)) ))
```

Retrieving a value: `(setq value (aref A 2 3))` ⇒ 6
Changing a value: `(setf (aref A 2 3) 12)`

> The `:initial-contents` keyword, together with the expression following it supplying the initial contents of the array, is optional. Optional parameters are standardly indicated with keywords, which as mentioned start with a colon and evaluate to themselves.

Hash tables – e.g.,

```
(setq A (make-hash-table))              ; creates it
(setf (gethash 'color A) 'brown)        ; sets value of an entry
(setf (gethash 'name A) 'fred)          ; sets value of an entry
(gethash 'color A)                      ; yields BROWN
(gethash 'name A)                       ; yields FRED
(gethash 'pointy A)                     ; yields NIL
```

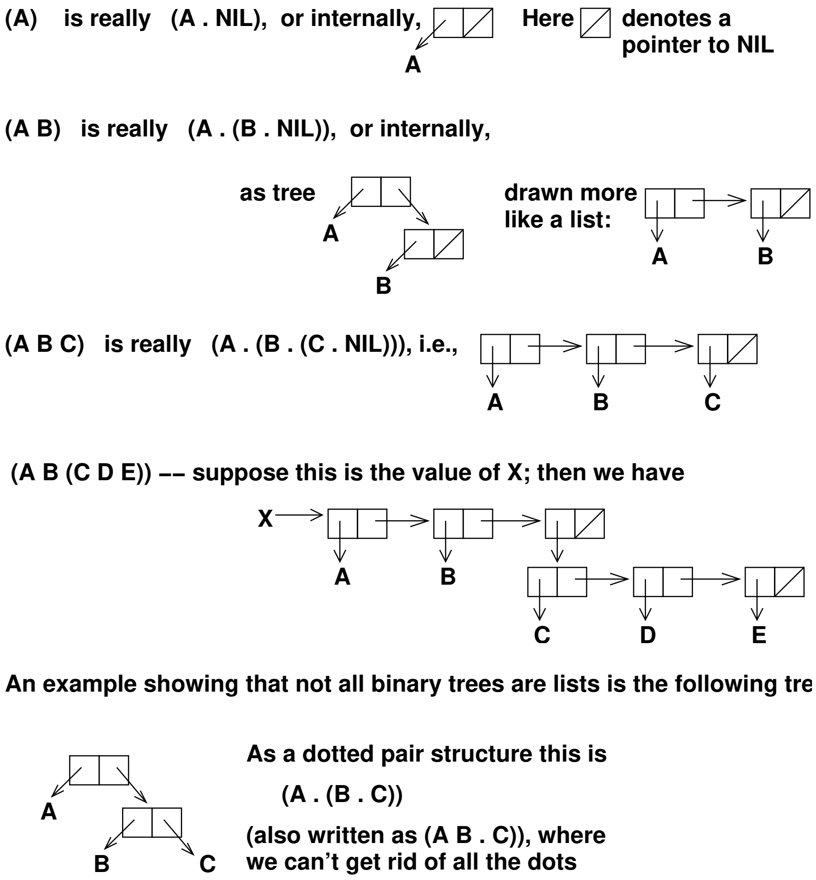
Record structures – e.g.,

```
(defstruct course
    name time credits room instructor)
(setq CSC242 (make-course :name "Artificial Intelligence"
                          :time 'TR3\:00-4\:15
                          :credits 4
                          :room 'CSB601
                          :instructor "John McCarthy"))
(setf (course-instructor CSC242) "Marvin Minsky") ; changes instructor
(course-time CSC242)                              ; yields |TR3:00-4:15|
```

Other data structures:
functions, packages, streams, ...

Dotted pairs and binary trees

It's usually most convenient to think of our data (and programs) as atoms and lists. However, list structures are actually built out of *dotted pairs*. You can think of these as pairs of storage locations (cells) where each location contains a pointer either to an atom or to another dotted pair. In this way we can build up arbitrary binary trees, and these can be used to represent list structures. (But while all list structures can be represented as binary trees, not all binary trees represent list structures; so dotted pairs in principle provide slightly more flexibility than list structures.) Here are examples of how we build list structures out of dotted pairs:



The left part of a dotted pair (i.e., the destination of the left pointer) is called its `CAR`, and the right part its `CDR` (pronounced "could'r"). `CAR` also has the alternative name `first`, since it picks out the first element of a list, as seen in the following.

```
(car '(A))    is the same as  (car '(A . NIL))      which yields A
(cdr '(A))    is the same as  (cdr '(A . NIL))      which yields NIL

(car '(A B))  is the same as  (car '(A . (B . NIL)) which yields A
```

```
(cdr '(A B))  is the same as  (cdr '(A . (B . NIL)) which yields
                                                       (B . NIL),
                                                       i.e., (B)
Similarly  (car '(A B C)) yields A      "first of list"
           (cdr '(A B C)) yields (B C)  "rest of list"
```

There are also functions `second`, `third`, ..., `tenth` for easy access to the first few elements of a list, though we could also use `(car (cdr ...))`, `(car (cdr (cdr ...)))`, etc. Additional "easy access" functions you should know about are `caar`, `cadr`, `cdar`, `cddr`, `caaar`, `...` `cddddr` as well as `nth`, `nthcdr`, and `last` (which does *not* give the last element of a list, but rather the last dotted pair, i.e., a list containing just the last element). Here are some more examples, assuming that we have defined `x` by the indicated `setq`:

```
(setq x '(A B (C D E)))

(car x)                             ==> A
(cdr x)                             ==> (B (C D E))
(car (cdr x)) = (cadr x)            ==> B
(cdr (cdr x)) = (cddr x)            ==> ((C D E))
(car (cdr (cdr x))) = (caddr x)     ==> (C D E)
(cdr (cdr (cdr x))) = (cdddr x)     ==> NIL
(first (third x)) = (caaddr x)      ==> C
(second (third x))                  ==> D
(last (third x))                    ==> (E)
```

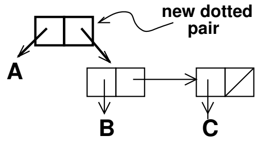
Constructing a dotted pair

`CONS` constructs a new dotted pair and returns a pointer to it. Examples:

```
(cons 'A 'B)      ==>    (A . B)
(cons 'A NIL)    ==>    (A . NIL) = (A)
```

When the second argument of a `CONS` is a list, we can think of its operation as *inserting the first argument at the head of the list given as second argument.* Thus

(CONS 'A '(B C))   yields (A B C)



(CONS '(A B) '(C (D E))) yields ((A B) C (D E))

(CONS '( ) '( )) yields (( )) = (NIL)

etc.

Basic mechanism: expression evaluation

Note that expressions like

```
                                       yielding
                                       ~~~~~~~~
 'hello                                 HELLO
 (load "my-file")                        NIL
 (setq n 5)                               5
 (car '(a b c))                           A
 (cons 3 n)                             (3 . 5)
```

are complete stand-alone programs.
This is in contrast with other programming languages, where expressions analogous to these could only occur within the body of some function or procedure definition (well, except for the "load" statement). However, only the second and third of the above statements have any lasting effect. In particular, the second statements (as you've learned) loads a file, and this file may contain function definitions, etc., that can be used thereafter. The third statement, as discussed before, sets the global value of a variable. Note again that `setq` does *not* evaluate its first argument; it does evaluate the second, but since this is a number, it evaluates to itself. (Recall that the boolean literals `T, NIL` (or `( )`), characters, strings, and keywords also evaluate to themeselves.)

Of course, an extremely important type of expression is `defun`, whose evaluation makes a function available for subsequent use. We now look at some simple examples.

Some simple examples of functions

true-listp – a function that checks whether its argument is a true list at the top level (its elements need not be lists). Note the use of "tail-recursion" (recursion on the `cdr` of the input).

Note: The predefined function `LISTP` just checks if its argument is `nil` or a `cons` (i.e., a dotted pair); e.g., `(listp '(a . b)) = T`. Also note that we can put comments after a semicolon – the Lisp reader ignores semicolons and everything following them.

```
(defun true-listp (x)
;~~~~~~~~~~~~~~~~~~~~
  (or (null x) (and (listp x) (true-listp (cdr x)))) )

e.g., (true-listp '(a (b . c) c)) ==> T
      (true-listp '(a . b)) ==> NIL
      (true-listp 3) ==> NIL
      (true-listp NIL) ==> T
      (true-listp '(a . (b c))) ==> T
```

list-structure – a function that checks whether its argument is a true list structure (i.e., it's not an atom other than NIL and prints without "dots"). Again we use tail-recursion.

```
(defun list-structure (x)
;~~~~~~~~~~~~~~~~~~~~~
   (or (null x)
       (and (not (atom x)); we could use (listp x) instead
            (or (atom (car x)) (list-structure (car x)))
            (list-structure (cdr x)) )))


e.g., (list-structure 'a) ==> NIL
      (list-structure '(a b c)) ==> T
      (list-structure '(a (b . c) c)) ==> NIL
      (list-structure '(a . (b c))) ==> T
```

standard-dev – a function for computing the standard deviation (root mean square deviation from the mean) of a list of numbers. This illustrates the use of the "fell swoop" functions `reduce` and `mapcar`, as well as direct use of a lambda-expression.

```
(defun standard-dev (data)
;~~~~~~~~~~~~~~~~~~~~~~~~~
; data: a list of numbers
; RESULT: the root mean square deviation from the mean of the data
;
  (let ((n (length data)) mean); define 2 local variables, supplying
                               ; initial value = (length data) for n
       (if (zerop n) (return-from standard-dev nil))
       (if (= n 1) (return-from standard-dev 0))
       (setq mean (/ (reduce #'+ data) n))
       (sqrt (/ (reduce #'+
                  (mapcar #'(lambda (x) (expt (- x mean) 2))
                          data ))
                n )))); end of standard-dev

e.g., (standard-dev '()) ==> NIL
      (standard-dev '(1)) ==> 0
      (standard-dev '(2 6)) ==> 2
      (standard-dev '(1 4 1 5 9 2 6 5)) ==> 2.5708704
                                            (cf., 5/sqrt(3) = 2.89 :-)
```