CS 5541 (AI): A Few Quick Thoughts on Emacs, Lisp

Rich Maclin
Computer Science Department
University of Minnesota Duluth

Running emacs

- To run, simply type emacs on the command line of an xterm
- To run clisp in the ilisp package type Alt-X run-ilisp, then when prompted to clisp, this will break your windows into two windows (one an editor and one the lisp interpreter)
- You can find many lists of emacs commands online, this is a reasonable one: http://www.physics.ohio-state.edu/~driver/Emacs_Quick_Reference.pdf

Some Useful emacs Commands

Most can be done with emacs menu

- C-x C-f – open or create file
- C-x C-s – save file
- C-x s – save all buffers
- C-x C-w – write file
- C-x C-c – exit emacs
- C-g – cancel current command
- C-x C-b – list all buffers
- C-x b *name* – shift to named buffer
- C-x k – kill buffer

Ilisp menu options

Look for ilisp on menu

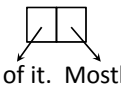
- Load file – load generally a definitions file
- Eval region – evaluate highlighted region in buffer in lisp
- Eval defun – evaluate the surrounding defun statement in the buffer
- Lots of other useful commands

Basics of Lisp

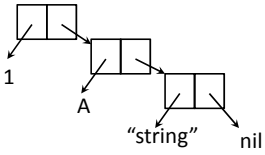
- Basic data types
  - Numbers
  - Symbols 'A is the symbol A
  - Character is #\X – character X
  - #(…) is a vector
  - " … " is a string
  - nil is the null list - as is '()
  - True is anything not nil (can use t)
  - Lists can be lists of any type of object (can mix things together)

Lisp List

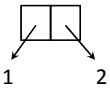
- A list consists of cons cell(s):



- A cons has a first and a rest part of it. Mostly lists of items consists of one cons per item with a pointer to the null list at the end
- '(1 A "string") is



'(1 . 2) is



Lisp List (cont)

- ‘ before (or symbol is equally shorthand for (quote …) command, ‘A is the symbol A
- Can also construct list as (list 1 A “string”)
- Can have arbitrary lists within lists
  ‘(1 (2 A) (3 (“string”) 4) 5)

Running Lisp

- Lisp is generally run as an interpreter (though it can be compiled)
- A Lisp program is generally a file or files defining a set of functions (plus macros, global parameters, etc.)
- The file(s) are loaded into the interpreter and generally called from the command line

Programs in Lisp

- Programs in lisp are written as sets of interacting functions
- Often the program is run by typing a function call at the command line
- The basic syntax of Lisp is
  (*functionName arg1 arg2 arg3* …)
- You can call Lisp functions or write your own

Simple Program

```
(defun factorial (x)
  (if (<= x 0)
      1
      (* x (factorial (- x 1)))
  )
)
```

Type in to define then try calling (factorial 5)

Function Definition (named)

(defun *functionName (arguments)*
  body
)

Name is any reasonable name

Arguments are named pass by value arguments

The body (should) be a single function call where the value of that call is returned as the value of the function, in practice Lisp lets you list multiple function calls in the body and then the value of the function is the last one

Some Basic Functions

(+ *number number* …)
Also -, *, /
(< *number number*)
Also >, <=, >=, ==,
(and *value value* …)
Also or
(not *value*)
(eq *value1 value2*)
Also eql, equal, equalp
(null *item*)

Assignment:
> (setq a 5)
…
> A
5
setf a more powerful version (macro that works on some function calls)

List Manipulation

(setq a '(1 (2 3) 4 5 (6)))

(car a) is 1
(cdr a) is ((2 3) 4 5 (6))
(cadr a) is (2 3)
(caadr a) is 2
(cdadr a) is (3)
(cddr a) is (4 5 (6))
(caddr a) is 4
(cdddr a) is (5 (6))

(nth 0 a) is 1
(nth 3 a) is 5
(nthcdr 3 a) is (5 (6))

first is equivalent to car
rest is equivalent to cdr

(cons a b) creates a cons cell of a b
(list *items*) creates a list of the items *items*
(append *lists*) glues lists together

Multiple Command Structures

Temporary variable declaration (series of variable names, values in lists)

```
(let
   ((x 5)
    (y 6)
   )
   body of let
)
```

let* evaluates arguments in order (can use earlier temporary variable names in later variable values)

Evaluate set of commands
(progn *commands*) – evaluate *commands* in order, return value of last
(prog1 *commands*) – evaluate *commands* in order, return value of first

Some dialects have other commands (e.g., prog2)

Control Structures

(if *expr truestmt falsestmt*) – if *expr* is true evals *truestmt*, otherwise evals falsestmt
(when *expr form1 form2* …) – if *expr* is true evals *form1*, etc. in order
(unless *expr form1 form2* …) – if *expr* is false evals *form1*, etc. in order
(cond
 (*expr1 form11 form12 form13 …)*
 (*expr2 form21 form22 form23 …)*
 (*expr3 form31 form32 form33 …)*
 ..
) – evaluates *expr1*, if true evaluates *form11, form12*, etc. in order, if *expr1* is false evaluates *expr2* and if true evaluates *form21, form22,* etc. in order, in all cases returns the value of the last form evaluated

Loop Structure – Do

```
(do
 ((var1 val1 nextval1)
  (var2 val2 nextval2)
   …
 )
 (endtest result)
 commandlist
)
```

```
(do
 ((i 0 (+ i 1))
  (x nil)
 )
 ((null (nthcdr i lst)) x)
 (push (nth i lst) x)
)
```

do* is the same except the variable list is declared sequentially

Other Loops

(dolist (*varname list*) *commands*) – set the variable name *varname* to each of the items in *list* in turn and evals *commands*

(dotimes (*varname integerarg result*) *commands*) – set the variable name *varname* to each value from 1 to *integerarg*-1 evals *commands*

Lots of others, mapcar, mapcan, and loop!

Another Data Structure - Arrays

(make-array *listofdims*) – lots of optional arguments to control aspects such as initial element

(make-array '(2 2)) makes a 2x2 array of any type of object (initial values are all nil), dimensions are ordered starting from 0

Referring to an array element
(aref *arrayname dim1 dim2* …)

Use setf to set the corresponding value
(setq a (make-array '(2 2)))
(setf (aref a 0 0) 1)

Input/Output

(read) – reads a standard lisp object

(write obj) – writes an object out

Can use versions that understand escape characters (see for example, prin1, print, pprint, princ)

(format *dest controlstring args*) – dest is t for the command line, file handle otherwise, control string is a bit like a C printf string, but we use ~S (any s-expression), ~D (integer), ~A (ascii like strings, chars), ~F (floating point), can use width values after ~ before letter (more for some formats)

(format t "Hi ~S is ~3D for ~4,1F\n" 'a 12 61.353)

Produces

Hi a is 12 for   61.3

Useful Macros

setf – set value of object, can be used on aref, on nth, etc.

incf – increments the value of its argument

decf – decrements value of argument

push – push first arg onto second arg list

pop – remore and return as value first element of arg