Lisp Tutorial

CS157

April 9, 2004

Interpreter

- On Leland machines, type

  ```
  alisp
  ```

- To exit, type

  ```
  (exit)
  ```

- The interpeter runs in a ‘read-eval-print loop’.
- Try out a sample command like addition

  ```
  (+ 1 2 3)  => 6
  ```

  The notation used here is fairly common. The result of executing the Lisp command (+ 1 2 3) results in 6.

Data Types: Atoms

- There are two: atoms and lists.
- Atoms: `a`, `7`, `tom`, `my-age`, `nil`, `T`
  - Evaluate an atom gives the value assigned that atom. Numbers are special--they always evaluate to themselves.

```
7   => 7
my-age   => Error (variable not defined)
```

  - Assign a value to an atom with `setq` and/or `setf`

```
(setq my-age 24) => 24
my-age   => 24
```

  - Reserved: nil and T

```
nil  => nil
t    => t
```

  - Case insensitive: aBc and ABC are the same atom

Data Types: Lists

- Lists: `(a)`, `(+ 6 7)`, `(a (e f g) h)`, `()`, `nil`
  - nil is both an atom and a list. It is equivalent to the empty list: ().
  - Evaluating a list always invokes a function.

    (function-name arg1 … argn)

    ```
    (+ 6 7)  => 13
    (foo 17 18 19) => Error (function not defined)
    (+ my-age 4)   => 28
    ```

  - When evaluating a list, first evaluate all the arguments and then apply the function to all those results.

Primitive Functions 1

- Lisp includes many built-in functions: +, *, -, /, max, min, sqrt
- More importantly, includes built-in list operations.
  - cons builds a list out of data and a tail.

```
(cons 1 nil)  =>  (1)
(cons 1 (cons 2 nil))   => (1 2)
```

  - Notice the tail value signifying ‘nothing’ is nil.
  - Now try using it with atoms that are not reserved.

```
(cons a nil)  => Error: variable a is not defined.
```

    - Why does this give an error?

Primitive Functions: Building Lists

- We need to tell the interpreter not to evaluate the atom a. To do that, we use ‘.

```
(cons ‘a nil)   =>  (a)
```

- ‘ can also be applied to an entire list

```
(cons ‘a ‘(b c d)) => (a b c d)
```

- cons can be irritating: make a list of ‘a, ‘b, ‘c, and ‘d without quoting a list.
  - list and append are functions that take any number of arguments:

```
(list ‘a ‘b ‘c ‘d)   => (a b c d)
(list ‘a ‘(b c d))   =>  (a (b c d))
(append ‘(a b c) ‘(d e f) ‘(g h i))   => (a b c d e f g h i)
```

Backquote and Comma

- Sometimes we want to quote a list except for one or two of its members, e.g.

  (list ‘a ‘1 (/ my-age 2) ‘b ‘4) => (a 1 12 b 4)

- Backquote and comma allow us to write the above without so many quotes:

  `(a 1 ,(/ my-age 2) b 4)

  – Thus the comma within a backquote tells Lisp to evaluate what follows. That is just the opposite of what the ‘ function does normally.

Primitive Functions: Accessing Lists

- Once a list is built, how do we access its members?
  - first and car give you the first element of a list.

```
(first '(1 2 3))   => 1
(first '((a b) 2 3))   =>  (a b)
```

  - last and cdr give you the list minus the first element.

```
(rest '(1 2 3))   => (2 3)
(rest '((a b) 2 3))   => (2 3)
```

  - car and cdr can be combined

```
(caar '((a b) c d e))   =>  a
(cdar '((a b) c d e))   => (b)
```

    Work from the right to the left

Changing Atom Values

- Example

```
(setq mylist '(b c d))   =>  (b c d)
(cons 'a mylist)  =>  (a b c d)
mylist => (b c d)
```

- No side effects! (for the most part)
  - Need to setq to change an atom's value.

- setf is a version of setq that takes a function as its first argument.

  (setf (cadr mylist) 7) => 7

  mylist => (b 7 d)

Equality

- Talk about 2 types of equality, via examples

  (eq ‘a ‘a) => T

  (eq ‘a ‘b) => nil

  (eq ‘(a) ‘(a)) => nil

  (equal ‘a ‘a) => T

  (equal ‘(a) ‘(a)) => T

  (equal ‘(or p q) ‘(or p q)) => T

- (equal x y) is t when (eq x y) is true and when things that look the same are true (sort of).

Sets

- Can treat lists as sets (order not preserved)

```
(union '(a b) '(a d))  => (a b d)
(union '((a) (b)) '((a))) => ((a) (b) (a))
(union '((a) (b)) '((a)) :test #'equal) => ((a) (b))
```

  – The test condition for determining whether 2 items in the set are the same is the function equal.

```
(adjoin 'a '(a b c))  => (a b c)
(set-difference '(a b c) '(b d c)) => (a)
```

- adjoin and set-difference can use `:test #'equal` as well. Can even supply your own function (once we show you how to define one).

More functions

- (length ‘(a b c)) => 3
- (atom ‘a) => T
- (atom ‘(a b c)) => NIL
- (listp ‘a) => NIL
- (listp ‘(a b c)) => T

Representing KIF Sentences

- KIF sentences:

  (<= p (and q r))

  (not (not q))

  (or r (not s) t)

- How do we represent these in Lisp?
  - Simple--use lists.

```
(list '<= 'p (list 'and 'q 'r))
(list 'not (list 'not 'q))
(list 'or 'r (list 'not 's) 't)
```

- What about the empty clause?
- Notice the KIF operators =>, <=, <=>, or, and, not will always be the first element of the list.

Defining functions

- (defun <name> <documentation-string> (<arguments>) <body>)

  (defun square “computes square” (x) (* x x))

  (defun ar (premises conclusion) nil)

- Note that we need not quote any of the arguments to defun. It is taken care of automatically.

- Evaluating a function for some set of arguments results in the last expression evaluated in the function trace.

Branching

- (if <expr> <then-expr> <else-expr>)

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

- (cond ((testa) (form1a) (form2a)…(formNa))

  ((testb) (form1b)…(formNb)) …

  (t (form1k) … (formNk))

```
(cond ((atom p) nil)
            ((listp p) (car p))
          (t nil))
```

- Evaluating cond evaluates the `tests` until one evaluates to true. It then evaluates each of the appropriate `forms`. The last evaluation is the value of the entire cond function.

Logical Functions and Null

- (and <form1> <form2> … <formn>)
  - Evaluates to nil as soon as one of <formi> is nil. Otherwise evaluates to <formn>
- (or <form1> <form2> … <formn>)
  - Evaluates to first non-nil argument. If there are none evaluates to nil.
- (not <form>) and (null <form>) are identical. Usually use the latter when the result should be a list.
  - Evaluate to T iff <form> is nil.

```
(not (+ 1 2 3))  => Nil
(not (and  (eq (+ 1 2) 3) (< 4 3) (/ 5 0)))  => T
```

Iteration

- (let ((<var1> <init1>) (<var2> <init2>) …) <body>)
  - Declares local variables. It is best to declare variables before using them.

- (dotimes (<counter> <limit> <result>) <body>)

```
(let ((sum 0))
    (dotimes (i 10 sum) (setq sum (+ sum i))) ) => 45
```

Iteration2

- (dolist (<var> <initlist> <result) <body>)

```
(let ((sum 0))
      (dolist (i '(1 2 3) sum)
                (setq sum (+ sum i)) ))   => 6
```

- (do ((<var1> <init1> <increment1>)
  (<var2> <init2> <increment2>) …)
  (<termination-test> <result>)
  <body>)
  – Combines let and dolist

Example

```
(defun positive-literals (clause)
  (cond ((atom clause) nil)
        ((listp clause)
            (do ((cs clause (cdr cs))
                 (poslist nil))
                ((null cs) poslist)
              (if (pos-litp (car cs))
                  (setq poslist
                        (cons (car cs) poslist))
  )))))
```

- Result of (positive-literals ‘(or p (not q) (not s))) ?

Recursion

- Nothing new here.

```
(defun power (base exp)
         (if (eq exp 0)
             1
             (* base (power base (- exp 1)))
  ))

(power 3 2)  => 9
```

Mapcar

- (mapcar <function-name> <list>)
  - mapcar applies the function to each element of list and returns a list of the results.

```
(mapcar ‘atom ‘(1 2 (a b) 3)) => (T T nil T)

(defun times2 (x) (* x 2)) => times2
(mapcar ‘times2 ‘(2 3 4)) => (4 6 8)
```

- Other List operations: remove-if, remove-if-not, some, every, search, subseq, length

Strings

- A string in Lisp is technically an array of characters (just like C).
- You'll need to use special operators to work with strings: concatenate, subseq, search.
- The sample code includes enough functionality so that you should not need to worry about strings besides outputting the results of your reasoner.
- The only thing to watch out for is when you are including a quote in the html you send back to the client. Just make sure you use a \ to escape any such quotes. An example appears in the function myfrontpage.

Output

- (print <form>) both prints the evaluation of <form> and returns the evaluation of <form>.

- Useful for debugging

- princ, print1, print all work the basically the same but with minor differences.

Formatted Output

- (format <destination> <control-string> <optional-arguments>)
  - <destination>: `t` prints to the command line
    `nil` prints nothing but returns string
    else prints to the stream <destination>
  - <control-string>: much like printf/sprintf in C. Includes placeholders for arguments.
  - <optional-arguments>: Arguments that fill in the placeholders in the control-string

Formatted Output 2

```
(format t “7 * 6 = ~A” (* 7 6))
  7 * 7 = 42
  => NIL

(format nil “~A” ‘(a b c))
  => “(a b c)”
```

~A: Ascii--any Lisp object

~D: Decimal--numbers

~%: newline

~~: tilde

Debugging: trace

- (trace <func-name1> <func-name2> … <func-namen>), e.g. (trace times2 positive-literals)
  - Every time one of these functions is evaluated, Lisp prints out the function name and the arguments it was given to the terminal. Every time one of these functions exits, Lisp prints out what its return value was.
  - Calling trace multiple times will add to the list of functions.
- To turn off tracing for a function foo and bar, use

  (untrace foo bar)
- To turn off all tracing, use

  (untrace)

References

- Lisp Primer (borrowed content of these slides)
  - http://grimpeur.tamu.edu/~colin/lp/

- Lisp programming instructions
  - http://logic.stanford.edu/classes/cs157/2004/programming/lispserver.html