Overview

- Announcement
- Lisp Basics

Announcement

- CMUCL to be available on sun.cs.
- You may use GNU Common List (GCL)
  http://www.gnu.org/software/gcl/
  which is available on most Linux platforms.
- There is also a commercial version of Common Lisp which is free to students:
  - Allegro Common Lisp
  - Supports Linux, windows, FreeBSD, Mac OS X

Outline of Writing and Running Lisp

1. Write a program (function definitions) in a file: `blah.lsp`
   ```lisp
   (defun mysq (x)
     (* x x)
   )
   (defun mytest (x)
     (if (> x 10)
       'Blah
       'Poo
     )
   )
   ```
2. Run `lisp /opt/apps/cmucl/bin/lisp`
3. Load function definitions (load "blah.lsp")
4. Run functions
   (mysq 10)
   (mytest 2)

LISP: A Quick Overview

- Components: Atoms, Lists, and Functions.
- Basics: list, math, etc.
- Arrays and SETQ vs. SETF
- Variable binding
- Lexical vs. dynamic scope
- Conditionals, predicates, iterations, etc.
- User-defined function
- Recursion
- Output
Components

Symbolic expression = ATOM or LIST.

- Atom: numbers, variable names, etc.
  
  \[ <\text{letters}|<\text{digits}>\] +
  
  e.g.: 1, 10, foo, bar, this-is-an-atom

- List: functions, list of items
  
  "( " [<list>|<atom>]\* " )"
  
  e.g.: (a), (1 1 3) (4 5 6)

- NIL: it is an atom and at the same time a list.
  
  NIL is the same as ()

- T: true, as opposed to NIL.
  
  See conditionals and predicates.

Basics

- quote: returns a literal (i.e. not evaluated) atom or a list.
  
  ‘(+ 2 3) \rightarrow (+ 2 3)
  
  (quote (+ 2 3)) \rightarrow (+ 2 3)

  Compare with:
  
  (+ 2 3) \rightarrow 5
  
  (eval ‘(+ 2 3)) \rightarrow 5

- Basically, you can think of a quoted atom or list as data, as opposed to instruction, in Lisp.

Evaluation in Lisp

- Lisp basically tries to evaluate everything (atom or a list) that is not quoted.

- If it sees an atom, it treats it as a variable, and tries to find out a value assigned to it.

- If it sees a list, it treats it as a function, where the first element in the list is seen as the function name and the rest function arguments.

- The quote function is used to exactly avoid such behavior (i.e., evaluate by default).

Evaluation in Lisp (cont’d)

- For example, if you typed in (hello (my world)),
  
  1. Lisp will see the first entry in that list as a function and tries to evaluate it using the argument (my world).
  2. But, it needs to evaluate all of the arguments first, so it will try to evaluate (my world).
  3. Since this also looks like a function, Lisp will now try to evaluate function my.
  4. To do that, it needs to evaluate the symbol world. Since it is an atom, Lisp will check if any value is assigned to the symbol world (i.e., treating it as a variable).

- What about ((hello world) (my friend))?
Evaluation in Lisp (cont’d)

- What about \((* 10 \text{b})\)?

- Lisp sees a well-defined function \(*\) and proceeds to evaluate its arguments first.

- It is happy with the number 10, so it proceeds on to evaluate \(\text{b}\).

- Here’s where the problem begins. If you already did something like \((\text{setq} \ \text{b} \ 20)\), then Lisp knows \(\text{b}\) can be evaluated to the value 20, so it will do that and evaluate \(*\) with that and return 200.

- If you haven’t defined \(\text{b}\), Lisp will treat it as an unbound variable, and balk.

- What about \((* 10 ‘\text{a})\)?

Basics: List

- **car**: returns first element (atom or list)
  
  \[
  \text{car} \ ‘(a \ (b \ c)) \rightarrow A \\
  \text{car} \ ‘((b \ c) \ a) \rightarrow (B \ C)
  \]

- **cdr**: returns all except the first element of a list, as a list
  
  \[
  \text{cdr} \ ‘(a \ (b \ c)) \rightarrow ((B \ C)) \\
  \text{cdr} \ ‘((b \ c) \ a) \rightarrow (A)
  \]

List can be seen as trees: atoms at leaves and internal nodes representing lists.

- Once this is understood, the list operations such as **car**, **cdr**, **cons** become easy to understand.

- Exercise: draw the tree for \((((((a))))))\)

Basics: List

- Combinations are possible: \(c\text{XXXXr}\) where \(X=(a | d)\)
  
  \[
  \text{cadr} \ ‘(a \ (b \ c)) \equiv (\text{car} \ (\text{cdr} \ ‘(a \ (b \ c)))) \rightarrow (B \ C)
  \]

- **list**: creates a list out of atoms and lists
  
  \[
  \text{list} \ ‘(1 \ 2) ‘((3 \ 5) \ (7 \ 8))) \\
  \rightarrow (A \ (1 \ 2) ((3 \ 5) \ (7 \ 8)))
  \]

- **length**: number of elements in a list
  
  \[
  \text{length} \ ‘(a \ b \ c) \rightarrow 3
  \]

- Some shorthands: **first**, **second**, **third**, ..., **nth**, **rest**
  
  \[
  \text{first} \ ‘(a \ b) \rightarrow A \\
  \text{nth} \ 2 \ ‘(a \ b \ c \ d) \rightarrow B
  \]

- **nth**: number of elements in a list (length ‘(a b c) = 3)
  
  \[
  \text{nth} \ 2 \ ‘(a \ b \ c \ d) \rightarrow B
  \]
Basics: List

- **CONS**: append an atom (or a list) and a list
  
  \[
  \text{cons } \left\langle a \right\rangle \left\langle 1 \ 2 \ 3 \right\rangle \rightarrow \left\langle A \ 1 \ 2 \ 3 \right\rangle \\
  \text{cons } \left\langle a \right\rangle \left\langle 1 \ 2 \ 3 \right\rangle \rightarrow \left\langle (A) \ 1 \ 2 \ 3 \right\rangle
  \]

- **APPEND**: append two lists
  
  \[
  \text{append } \left\langle 1 \ 2 \right\rangle \left\langle 4 \ 5 \right\rangle \rightarrow \left\langle 1 \ 2 \ 4 \ 5 \right\rangle
  \]

Basics: Assignments/Arrays

- **setq**: assignment of value to a symbol
  
  \[
  \text{setq } x 10 \rightarrow 10 \\
  x \rightarrow 10
  \]

- **setf**: can set the value of a symbol (== setq) or location or structure (next slide).

Arrays and SETQ vs. SETF

- **make-array**: create an array
- **aref**: array reference
- **setf**: set value of array element

More Fun with SETF

Replace list element with **SETF**. Note: **SETQ** will not work!

\[
\text{setf } b \left\langle 1 \ 2 \ 3 \right\rangle 4 \rightarrow \left\langle 1 \ 2 \ 3 \ 4 \right\rangle \\
\text{caadr } b \rightarrow 2 \\
\text{setf } \left(\text{caadr } b\right) \text{ 'abcdefg} \rightarrow \text{ABCDEFG} \\
\text{setq } \left(\text{caadr } b\right) \text{ 'abcdefg} \rightarrow \text{Error: (AREF A 2 ...)} \text{ is not a symbol.}
\]
Basics: Math

- $(+ 1 2) \times (3 4) \times (+ (2 3) / (4 5))$ etc.
- $(\text{max} 1 2 3 4 5) \times (\text{min} 4 6 5)$
- $(\text{sqrt} 16) \times (\text{expt} 2 3) \times (\text{round} 3.141592)$

Basics: File Loading

- $(\text{load} \ "\text{filename}\")$

Function

- defun: user defined function
  
  * $(\text{defun mult} (x y) (* x y))$
  
  DEFUN
  
  * $(\text{mult} 10 20)$
  
  200

- Use the let and let* forms:
  
  $(\text{defun mult} (x y)$
  
  $(\text{let} ((t x x) (t y y))$  
  
  $(\text{let}\ (t x x) (t y y))$
  
  $(\ast x y)$
  
  )$

Recursion

- Fibonacci number:
  
  $F(N) = F(N-1) + F(N-2), F(1)=1, F(2)=2.$
  
  $(\text{defun fibo} (x)$
  
  $(\text{cond}$
  
  $(\text{(equal} x 1) 1)$
  
  $(\text{(equal} x 2) 2)$
  
  $(> x 2)$
  
  $(+ (\text{fibo} (- x 1)) (\text{fibo} (- x 2))))$
  
  )$
  
  $(\text{* (fibo} 4)$

  5
  
  $(\text{* (fibo} 5)$

  8

Binding

You can bind variables anywhere in a program with the let or let* special forms to create a local context.

- let and let* : lexical scope (local context)
  
  $(\text{let} ((\text{local} v a r \ \text{list}) \ \text{BODY})$  
  
  $(\text{let*} ((x 10) (y (\ast 2 x)) (z)) \ \text{BODY})$

- Either just a variable or (variable default-value).

- With let*, values from previous vars can be used to define new value.
  
  $(\text{let*} ((x 10) (y (\ast 2 x)) z)) \ \text{BODY})$
Use of Local Scope

- Always use (let ... ) or (let* ...) be your first (and only) statement in your function, if you are writing something complex which is not like a mathematical function in its usual sense.

- Think of it as declaring local variables in C/C++.

(defun func-name (arg1 arg2)
  (let (localx localy localz)
    (expression1 args)
    (expression2 args)
    (expression3 args)
    (expression4 args)
    (expression5 args)
  )
)

Binding: Example

* (let ((a 3)) (+ a 1))
4
* (let ((a 2)
  (b 3)
  (c 0))
  (setq c (+ a b))
c)
5
* (setq c 4)
4
* (let ((c 5)) c)
5
* c
4

Lexical Scope

Return value according to the lexical scope where it was defined.

* (setq regular 5)
5
* (defun check-regular () regular)
CHECK-REGULAR
* (check-regular)
5
* (let ((regular 6)) (check-regular))
5

Dynamic Scope

Use the defvar to define a special variable that is dynamically scoped. (Just think of it as defining a global variable.)

* (defvar *special* 5)
* SPECIAL*
* (defun check-special () *special*)
CHECK-SPECIAL
* (check-special)
5
* (let ((*special* 6)) (check-special))
6
* *special*
5
*(let ((x 23)) (check-special))
5
Group (or Block) of Commands

`progn` returns the result of the last element, but evaluates all s-expressions in the argument list.

- `(progn (setq a 123) (* 5 10)) → 50
  a → 123`

A better way of writing it is:

```lisp
(progn
  (setq a 123)
  (* 5 10)
)
```

How Not to Define a Block

A common mistake is to define a block using just bare parentheses, instead of using the function `(progn ...)`:

```
(setq x 10)
(setq y 20)
(* x y)
```

It looks fine, but as mentioned earlier, Lisp will interpret this list as a function that has a name `(setq x 10)` and two argument `(setq y 20)` and `(* x y)`. So, don't do this!

Conditionals: the Ps.

`p` is for predicate:

- `numberp`, `listp`, `symbolp`, `zerop`, ...
- common comparisons: `<`, `>`,
- `equal`: if the values are the same.
- `eq`: if the memory locations are the same.
- `and`, `or`, `not`: logical operators.

Returns either `NIL` or `T`.

Control Flow

IF STATEMENT
```
(if (> 2 3) ; condition
  (+ 4 5) ; when true
  (* 4 5) ; when false
)
```

SWITCH STATEMENT
```
(cond ((testp1) (return-value1)) ; condition 1
  ((testp2) (return-value2)) ; condition 2
  ((testp3) (return-value3)) ; condition 3
  (t (default-value)) ; default
)
Iterations

DOTIMES
(dotimes (index-var upper-bound result-var) BODY)

* (dotimes (k 1 val) (setq val k))
0

* (dotimes (k 10 val) (setq val k))
9

Also find out more about dolist, do, and loop.

Output

- print: print a string
  (print "hello")

- format: format a string; (format dest string args)
  dest: determines what to return — t: return NIL, NIL: return string.
  ~% : insert CR
  ~S : S-expression
  ~A : ascii
  ~D : integer
  ~widthD : blank space e.g. ~5D
  ~F : floating point
  ~width,decimalF : width and decimal point

Format: examples

* (format t "Hello, world!")
Hello, world!
NIL

* (format nil "Hello, world!")
"Hello, world!"

Format: examples

* (format nil 
  "The list is ~s and ~% the text is ~a"
  (list 'a 'b 'c)
  "This is a string"
)
"The list is (A B C) and the text is This is a string"
Format: examples

* (format nil "One: ~d~%Two:~f~%Three:~5,2f" (/ 4 3) (/ 4 3))
"One: 12
Two: 1.3333334
Three: 1.33"

Dealing with Errors

* a <--- erroneous input

Error in KERNEL::UNBOUND-SYMBOL-ERROR-HANDLER: the variable A is unbound.

Restarts:
0: [ABORT] Return to Top-Level.

Debug (type H for help)

(EVAL A)

Source: Error finding source:
Error in function DEBUG::GET-FILE-TOP-LEVEL-FORM: Source file no longer exists:
target:code/eval.lisp.
0] q <--- go back to top level

Overview

• Some more LISP stuff: user input, trace, more setf, etc.
• Symbolic Differentiation:
  [q] does it need intelligence?
• Expression Simplification

READ: User Input

READ: keyboard input from user

*(read)
hello
HELLO

*(if (equal (read) 'hello)
  'good
  'bad
)
hello
GOOD
Symbolic Differentiation

Basics: given variable \( x \), functions \( f(x), g(x) \), and constant (i.e. number) \( a \):

1. \[
\frac{da}{dx} = 0, \quad \frac{d(a \times x)}{dx} = a
\]

2. \[
\frac{d(f + g)}{dx} = \frac{df}{dx} + \frac{dg}{dx}
\]

3. \[
\frac{d(f \times g)}{dx} = \frac{df}{dx} \times g + f \times \frac{dg}{dx}
\]

The operators can be extended to: binary minus (e.g. \(- x 1\)), unary minus (e.g. \(- x\)), division (e.g. \(/ x 10\)), etc.

Describing in LISP (I)

\[
(\text{deriv } \text{<expression>} \text{ <variable>})
\]

1. \[
\frac{da}{dx} = 0, \quad \frac{d(a \times x)}{dx} = a
\]

\[
(\text{deriv } '10 'x) \rightarrow 0
\]

\[
(\text{deriv } '((* 10 x) 'x) \rightarrow 10
\]
Describing in LISP (II)

(deriv <expression> <variable>)

1.
\[
\frac{d}{dx}(f + g) = \frac{df}{dx} + \frac{dg}{dx}
\]
(deriv '(+ (* x 10) (+ 25 x)) 'x)
== (list '+
    (deriv '(* x 10) 'x)
    (deriv '(+ 25 x) 'x)
)

Describing in LISP (III)

(deriv <expression> <variable>)

1.
\[
\frac{d}{dx}(f \times g) = \frac{df}{dx} \times g + f \times \frac{dg}{dx}
\]
(deriv '(* (+ 14 x) (* x 17)) 'x)
== (list '+
    (list ' * (deriv '(* 14 x) 'x) '(* x 17))
    (list ' * '(+ 14 x) (deriv '(* x 17) 'x))
)

DERIV: the core function

Pseudo code (basically a recursion):
(defun deriv (expression var) BODY)

1. if expression is the same as var return 1
2. if expression is a number return 0
3. if (first expression) is '+, return
   '+ (deriv (second expression) var)
   (deriv (third expression) var)
4. and so on.

Main Function: DERIV

You can make separate functions for each operator:
(defun deriv (expr var)
  (if (atom expr)
      (if (equal expr var) 1 0)
      (cond ((eq '+ (first expr)) ; PLUS
              (derivplus expr var))
            ((eq ' * (first expr)) ; MULT
              (derivmult expr var))
            (t ; Invalid
              (error "Invalid Expression!")))
    )
)

**Calling DERIV from DERIVPLUS**

Then, you can call `deriv` from `derivplus`, etc.

```lisp
(defun derivplus (expr var)
  (list '+
        (deriv (second expr) var)
        (deriv (third expr) var)
    )
)
```

**Expression Simplification**

Problem: a lot of nested expression containing

```
(* 1 x) (* x 1) (+ 0 x) (+ x 0) (+ 3 4) ...
```

which are just `x`, `x`, `x`, `x`, and 7.

Use simplification rules:

1. `(+ <number> <number>)`: return the evaluated value
2. `(* <number> <number>)`: return the evaluated value
3. `(+ 0 <expr>) (+ <expr> 0)`: return `<expr`
4. `(* 1 <expr>) (* <expr> 1)`: return `<expr`
5. `(- (- <expr>))`: return `<expr`

HINT: look at the raw result and see what can be reduced.

**SPLUS: Simplify (+ x y)**

```lisp
(defun splus (x y)
  (if (numberp x)
      (if (numberp y)
          (+ x y)
          (if (zerop x)
              y
              (list '+ x y)
          )
      )
    (if (and (numberp y) (zerop y))
        x
        (list '+ x y)
    )
)
```

**Programming Exercise**

- Symbolic differentiation: details TBA.