An Introduction to LISP

M. Anthony Kapolka III
CS 340 - Fall 2011
Wilkes University

Church-Turing Thesis

- Church developed the λ (lambda) calculus
- Church's Thesis: any effectively (finite-time) computable function can be written as a λ-abstraction
- Turing developed the Turing Machine
- Turing's Thesis: any effectively (finite-time) computable function can be written as a Turing Machine program

both represent identical sets of functions

Functional Programming

- higher order functions
  - can take functions as its arguments
  - can return a function
- lazy evaluation (done only when needed)
- data abstraction
- equations with pattern matching
- quick to write, easy to verify, naturally parallel

History

- LISt Processing (and not Lots of Incredibly Stupid Parenthesis)
- developed specifically for AI in late 1950s by John McCarthy
- early versions interpreted and SLOW
- LISP is great for knowledge representation

How to Count Parenthesis

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
           (t (CONS (CAR list1)
              (my-append (CDR list1) list2)))))
```

How to Count Parenthesis

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
              (t (CONS (CAR list1)
                  (my-append (CDR list1) list2)))))
```

1

How to Count Parenthesis

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
              (t (CONS (CAR list1)
                  (my-append (CDR list1) list2)))))
```

2

How to Count Parenthesis

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
            (t (CONS (CAR list1)
               (my-append (CDR list1) list2)))))
```

1

How to Count Parenthesis

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
            (t (CONS (CAR list1)
                (my-append (CDR list1) list2)))))
```

2

How to Count Parenthesis

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
              (t (CONS (CAR list1)
                  (my-append (CDR list1) list2)))))
```

3

How to Count Parenthesis

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
            (t (CONS (CAR list1)
                (my-append (CDR list1) list2)))))
```

4

How to Count Parenthesis

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
            (t (CONS (CAR list1)
               (my-append (CDR list1) list2)))))
```

3

How to Count Parenthesis

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
            (t (CONS (CAR list1)
                (my-append (CDR list1) list2)))))
```

4

How to Count Parenthesis

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
          (t (CONS (CAR list1)
             (my-append (CDR list1) list2)))))
```

3

How to Count Parenthesis

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
          (t (CONS (CAR list1)
             (my-append (CDR list1) list2)))))
```

4

How to Count Parenthesis

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
          (t (CONS (CAR list1)
             (my-append (CDR list1) list2)))))
```

5

How to Count Parenthesis

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
          (t (CONS (CAR list1)
             (my-append (CDR list1) list2)))))
```

6

How to Count Parenthesis

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
            (t (CONS (CAR list1)
                (my-append (CDR list1) list2)))))
```

5

How to Count Parenthesis

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
          (t (CONS (CAR list1)
             (my-append (CDR list1) list2)))))
```

6

How to Count Parenthesis

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
          (t (CONS (CAR list1)
             (my-append (CDR list1) list2)))))
```

7

How to Count Parenthesis

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
          (t (CONS (CAR list1)
             (my-append (CDR list1) list2)))))
```

6

How to Count Parenthesis

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
          (t (CONS (CAR list1)
             (my-append (CDR list1) list2)))))
```

5

How to Count Parenthesis

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
          (t (CONS (CAR list1)
             (my-append (CDR list1) list2)))))
```

4

How to Count Parenthesis

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
          (t (CONS (CAR list1)
             (my-append (CDR list1) list2)))))
```

3

How to Count Parenthesis

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
          (t (CONS (CAR list1)
             (my-append (CDR list1) list2)))))
```

2

How to Count Parenthesis

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
          (t (CONS (CAR list1)
              (my-append (CDR list1) list2)))))
```

1

How to Count Parenthesis

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
            (t (CONS (CAR list1)
                (my-append (CDR list1) list2))))))
```

~~1~~ 0

McCarthy's S-expressions

"Symbolic Expression" ➟ allows encoding words & concepts

1. An atomic symbol (string, variable name, number)

2. If e_1 and e_2 are S-expressions, then so is $(e_1 \cdot e_2)$

A list can be implemented as $(e_1 \cdot (e_2 \cdot (\ldots(e_N \cdot \mathrm{NIL})\ldots)))$

NIL is an atomic symbol for the empty list.

(Pure) LISP

- lists (s-expressions) represent information
- programs represented just like data
- only function calls - has no side effects
- McCarthy's LISP had five basic functions:

  ATOM, EQ, CAR, CDR, CONS

ATOM

(ATOM A)

returns T is A is an Atom

else returns F

EQ

```
(EQ A B)
```

returns T is A and B are the same
else returns F

CAR

(CAR list)

returns the first item in list

(CAR `(a b c)) ➡ a

(CAR `((a b) (c d)) ➡ (a b)

CDR

(CDR list)

returns everything but the first item in the list

(CDR `(a b c)) ➡ (b c)

(CDR `((a b) (c d)) ➡ ((c d))

CONS

(CONS head list)

returns a new list with head appended as the first item to the list

(CONS 0 `(1 2 3)) ➡ (0 1 2 3)

(CONS `(a b) `(c d)) ➡ ((a b) c d)

(Pure) LISP

McCarthy used those five basic functions to write APPLY, EVAL, and EVALQUOTE ➟ this created a LISP interpreter!

APPLY

(APPLY first second) evaluates each of its two arguments and then applies the first (a function) to the second (a list of args)

(APPLY ` (CAR) (` (a b c))) ➡ a

EVAL

(EVAL X) evaluates (looks up) X and then calls APPLY on this value:

(SET `x `(CONS `A `(B C)))

(EVAL x) ➟ (APPLY (CONS `A `(B C)))

➟ (CONS A (B C)) ➟ (A B C)

EVALQUOTE

EVALQUOTE was effectively a LISP interpreter doing terminal I/O and invoking EVAL (which invokes APPLY.)

LISP

From primitive operations, define rest of the language

- (* 7 9) ➡ 63   (- (+ 3 7) 7) ➡ 10
- (> (* 5 6) (+ 4 5)) ➡ T
- (list 1 2 3 4 5) ➡ (1 2 3 4 5)
- (nth 0 ` (a b c d)) ➡ a
- (length ` (a b c d)) ➡ 4
- (member 3 ` (1 2 3 4 5)) ➡ (3 4 5)

Built in functions

- math functions on integers, rationals, reals, complex numbers
- looping and program control functions
- list & string manipulation
- input & output
- system calls

User-defined functions

```
(defun <function name> (<formal params>) <function body>)

(defun square (x) (* x x))

(defun hypotenuse (a b) ( SQRT (+ (square a) (square b))))
```

User-defined functions

(defun <function name> (<formal params>) <function body>)

```
(defun absolute-value (x)
  (COND ((< x 0) (- x))
        ((>= x 0) x)))
```

User-defined functions

(defun <function name> (<formal params>) <function body>)

```
(defun absolute-value (x)
    (COND ((< x 0) (- x))
          (t x)))
```

here t acts as true, the default action

my-length

```
(defun my-length (my-list)
  (COND ((null my-list) 0)
  (t (+ (my-length (CDR my-list)) 1))))
```

(my-length ((1 2) 2 (1 (4 (5))))) ➡ ???

my-length

```
(defun my-length (my-list)
  (COND ((null my-list) 0)
  (t (+ (my-length (CDR my-list)) 1))))
```

(my-length ((1 2) 2 (1 (4 (5))))) ➟ 3

my-nth

```
(defun my-nth (N my-list)
(COND ((= 0 N) (CAR my-list))
(t (my-nth (- n 1) (CDR my-list)))))
```

my-append

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
               (t (CONS (CAR list1)
                    (my-append (CDR list1) list2))))))
```

mimics built in APPEND function.

(my-append ` (1 2) ` (3 4 5)) ➡ ???

my-append

```
(defun my-append (list1 list2)
    (COND ((null list1) (list2)
                  (t (CONS (CAR list1)
                        (my-append (CDR list1) list2))))))
```

(my-append `(1 2) `(3 4 5)) ➟ (1 2 3 4 5)

something upsetting

```
> (f 4)
5
> (f 4)
6
> (f 4)
7
```

something upsetting

```
> (f 4)
5
> (f 4)
6
> (f 4)
7
```

Why is this upsetting?

something upsetting

```
> (f 4)
5
> (f 4)
6
> (f 4)
7
```

Why is this upsetting?

f is a function!

something upsetting

```
> (f 4)
5
> (f 4)
6
> (f 4)
7
```

Why is this upsetting?

f is a function!

Side Effects!!

set

```
> (f 4)
5
> (f 4)
6
> (f 4)
7
```

```
(set ('inc 0))    ; inc := 0

(defun f (x)
   (set 'inc (+ inc 1))
   (+ x inc))
```

set, setq

binds global variables

```
(set `x 0)

(setq x 0)
```

a function

```
(defun qr (a b c)
    (setq temp (SQRT (- (* b b) (* 4 a c))))
        (list (/ (+ (- b) temp) (*2 a))
              (/ (- (- b) temp) (* 2 a))))
```

a function

```
(defun qr (a b c)
    (setq temp (SQRT (- (* b b) (* 4 a c))))
        (list (/ (+ (- b) temp) (*2 a))
            (/ (- (- b) temp) (* 2 a))))
```

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

let

binds local variables

(let (<local vars>) <expressions>)

a better function

```
(defun qr (a b c)
(let (temp)
    (setq temp (SQRT (- (* b b) (* 4 a c))))
        (list (/ (+ (- b) temp) (*2 a))
              (/ (- (- b) temp) (* 2 a)))))
```

temp is declared to be local with let

value assigned to temp with setq

an even better function

```
(defun qr (a b c)
(let ((temp (SQRT (- (* b b) (* 4 a c))))
      (denom (* 2 a)))
        (list (/ (+ (- b) temp) denom)
              (/ (- (- b) temp) denom))))
```

temp & denom is declared & assigned with let

On dilbert - choices!

- **sbcl** - Steel Bank Common List - sbcl.org
  - Fork of Carnegie Melon University Common Lisp (cmucl)
  - Steel (as in Andrew Carnegie) and Bank (as in Andrew Mellon)
- **clisp** - Gnu Common Lisp - www.gnu.org/software/clisp/
- **ecl** - Embedded Common Lisp - common-lisp.net/project/ecl/
  - Stand alone or embed in C applications

see: www.cliki.net

```lisp
(load "average.lisp")
T

(average '(10 20 30 40))
25

(average2 '(1 2 3 4 5))
3

(quit)
```

```lisp
(load “farmer.lisp”)

(solve-fwgc '(e e e e) '(w w w w))

((E E E E) (W E W E) (E E W E)
 (W W W E) (E W E E) (W W E W)
 (E W E W) (W W W W))
```