CMPUT325: Meta-programming Fundamentals

B. Price and R. Greiner

29th September 2004

Program is Data I

- A Lisp Program≈ an s-expr: `(CAR '(1 2))`
- Lisp interpreter executes the s-expr
- An s-expr is just a nested list structure
- Treated as a data structure, an s-expr can be traversed, composed or decomposed
- A program is just a nested list structure
- Programs can be traversed, composed or decomposed

Program is Data II

- Consider the program

```
(LAMBDA (fn) (funcall fn fn) )
```

- It uses fn as a program (function to be called) and as data (arguments for function)
- We can call this function on a λ

```
(   (LAMBDA (fn) (funcall fn fn))
      '(LAMBA (X) (CAR x) )   )
```

- The λ argument is used as both program and data

$\equiv$ `( (LAMBDA (x) (CAR x))  '(LAMBDA (x) (CAR x)) )`

$\longrightarrow$ `LAMBDA`

Other examples

```
( (LAMBDA (fn) (funcall fn fn))  'ATOM )
→ t
( (LAMBDA (fn) (funcall fn fn))  CAR )
→
The function CAR (the variable) is undefined
( (LAMBDA (fn) (funcall fn fn))  'CAR )
→ Error: CAR expects a list!
( (LAMBDA (fn) (funcall fn fn)) '(LAMBDA (x) x) )
→ (LAMBDA (x) x)
( (LAMBDA (fn) (funcall fn fn)) '(LAMBDA (x) (x x)) )
→ The function x is undefined
( (LAMBDA (fn) (funcall fn fn)) '(LAMBDA (x) (funcall x
→ ... waiting ... waiting ...
```

Modifying Code I

```
(setf (symbol-function 'foo) '(LAMBDA (x) (CADR x)) )
```

- Why do I need symbol-function?
  There is a separate table for the data values and function values of symbols.

```
(setf foo 2)
foo → 2
(symbol-function 'foo)
→ (LAMBDA (x) (CADR x))
(foo '(A B C)) → B
(CONS (CAR (symbol-function 'foo)) '((u y z) y))
→(LAMBDA (u y z) y)
```


Modifying Code II

```
(setf (symbol-function 'bar)
        (CONS (CAR (symbol-function 'foo)) '((u y z) y))
→ (LAMBDA (u y z) y)
(bar 4 (+ 2 3) '(t Q)) →   5
(setf (symbol-function 'N-args)
     '(LAMBDA (x) (LENGTH (CADR (symbol-function x)))))
→
(LAMBDA (x) (LENGTH (SECOND (symbol-function x))))
(N-args 'foo)→  1
(N-args 'bar)→  3
(N-args 'N-args)→  1
```

Compiler vs Interpreter

COMPILER translates Source Program into Executable Object Code

Steps in Compiler-Based System:

1. read program
2. check syntax & type agreement
3. compile
   3.1 produce "object code"
   3.2 discard "source" program
4. run object code

Compiler vs Interpreter

INTERPRETER directly executes (Source) Program

Steps:

1. read next form
   1.1 evaluate (aka "execute") it
   1.2 print value
2. Notes:
   2.1 form may be a program (s-expr)
   2.2 only run-time checks performe

Lisp System

- Many *Lisps* have compilers - both byte-code and native
- Most *Lisps* include INTERPRETERs.
  READ-EVAL-PRINT Loop
- Some Lisp's (s-lisp) call compiler after each read so code is always compiled

LISP Interpretation: EVAL

- Interpretation is based on Evaluation
  which maps S-expr into S-expr

  `(CONS (CAR '(A B)) '(C D))` $\longrightarrow$ `(A C D)`

- Can write a Lisp Function to do it!
  `EVAL` of ⟨`form`⟩ is ⟨`form`⟩'s value.

- Use Subset of *Lisp*:

$$\begin{aligned} \langle form \rangle ::= \;& (\text{QUOTE } \langle s - expr \rangle) \\ & \mid (\text{CAR } \langle form \rangle) \\ & \mid (\text{CDR } \langle form \rangle) \\ & \mid (\text{CONS } \langle form \rangle \langle form \rangle\ ) \\ & \mid \text{t} \mid \text{nil} \end{aligned}$$

`(EVAL '(CONS t nil))` $\longrightarrow$ `(t)`
`(EVAL '(CONS (CAR '(A B)) '(C D)))` $\longrightarrow$ `(A C D)`

EVAL wrt Variables

- Problem: What does x evaluate to in:

```
(EVAL '(CONS x '(B C)) ?
```

- Solution: Specify the **CONTEXT** of the evaluation with an AssocList

```
( (x  foo)  (y  (t nil))  (z  nil) )
```

- `AssocList is a mini data base`
- `EVAL` takes 2 args: form + context

```
(EVAL '(CONS x '(B C))
      '( (x t ) ( y  (t nil)) (z nil))
  → (t B C)
```


EVAL in General

- `EVAL` form + context ⇝ s-expr
(Common Lisp EVAL does not accept a context argument)

```
e  ⇔  (eval 'e nil)
EVAL of e (with nil context) is s-expr
```

- `EVAL` is a function;
Can use like any other function!
- Can take only 1 arg
as if context = `nil`

Examples of EVAL Ia

```
'(CONS 'a '(b c)) → (CONS 'a '(b c))

(EVAL '(CONS 'a '(b c))) → (a b c)

(setq x '(list '+ 3 4))  → (list '+ 3 4)

'x → x

(eval 'x) → (list '+ 3 4)

x → (list '+ 3 4)

(eval (eval 'x))→ (+ 3 4)
```


Examples of EVAL Ib

```
(eval (eval x)) →7

(eval '(eval x))  →(+ 3 4)

(setq y 'x) → x

(eval 'y) → x

(eval '(QUOTE y)) → y

(eval y) →(list '+ 3 4)

(eval (eval y))  →(+ 3 4)
```

Examples of EVAL IIa

```
(EVAL 'x '( (y x) (z A) (x P)) )  →  P

(EVAL '(CONS (CAR x) y)
        '( (x (A B C)) (y  (D E))) )

  → (A D E)

(EVAL  '(QUOTE x)  '( (y x) (z  A) (x  P)) )

  → x

( (LAMBDA (x c) (EVAL x c))
  'W
  '( (W  A) (X  B) ) )
→ A
```


Examples of EVAL IIb

```
( (LAMBDA (x c) (EVAL 'W c))
   'fred
   '( (W A) (X B) ) )

→ A

( (LAMBDA (x c) (EVAL x c))
    '(QUOTE W)   '( (W  A) (X  B) ) )

→ W

( (LAMBDA (x c) (EVAL (EVAL x nil) c))
    '(QUOTE W)   '( (W  A) (X  B) ) )

→ A
Trick:
```

Extending the Language

- Common-Lisp defines
  (IF $\langle test - form \rangle$ $\langle true - form \rangle \langle else - form \rangle$)
- How could we define this in terms of pure Lisp primitives?
- Our first try (DO NOT IMPLEMENT!):

```
(DEFUN my-if (testF trueF falseF)
   (COND (test trueF)
         (t falseF)))
```


Testing Naive IF

- Consider an application:

```
(setf x '(1 2))

(my-if (ATOM x) x (CAR x)) →   1

(setf x 'blah)

(my-if (ATOM x) x (CAR x))
→   error 'blah is not a list
```

- Note `(CAR x)` is always evaluated!

Custom Evaluation of Forms

- Solution: Custom control over evaluation of args

  Eval 1st `arg`
      if true: eval `2nd arg`
      if false: eval `3rd arg`

- We seem to need a new special form
- But Lisp's set of special forms is closed
- Actually, there's another way:

Macro-functions

- Macro-functions get the unevaluated form and context; and return a new form to be evaluated in its place
  - Installing an ordinary function into the function symbol table

```
(setf (symbol-function 'foo-fun)
   (function (lambda ()
      (list '+  1 2))))
(foo-fun) →  (+ 1 2)
```

  - Installing a macro-function into the macro symbol table

```
(setf (macro-function 'foo-mac)
   (function (lambda (args ctx)
         (list '+ 1 2))))
(foo-mac) →  3
```

- Result of `foo-mac` is evaluated!

Macro-functions with Arguments

- Ordinary function installed into the function symbol table

```
(setf (symbol-function 'foo-fun)
    (function (lambda (a1 a2 a3)
         (list a1  a2  a3)))))
(foo-fun 'cons 'a nil) →
(cons a nil) ;; cons quoted!
```

- Macro-function installed in macro symbol table

```
(setf (macro-function 'foo-mac)
   (function (lambda (args ctx)
      (let ((a1 (second args))
            (a2 (third args)) (a3 (fourth args)))
         (list a1  a2  a3 ))))))
(foo-mac cons 'a nil)  →  (a)
```

- Why skip (first args)? = macro name ⇒infinite loop

The defmacro Special Form

- Powerful and convenient way to extend the language:

  `(defmacro name ( a1 ... an )` ⟨*form*⟩`)`

- A macro has a list of formal arguments like LAMBDA or DEFUN
- Formal arguments are bound to *unevaluated* arguments supplied
- The ⟨*form*⟩ is evaluated (⟨*form*⟩ may use arguments)
- The result of ⟨*form*⟩ is returned in place of the macro
- Lisp evaluates returned result

The defmacro Special Form

- A function evaluates its body form and returns the result

```
(defun mystery-fun ()
   (list '+  1  2 )))
(mystery-fun)
→  (+ 1 2)
```

- A macro evaluates its body form and then evaluates the result

```
(defmacro mystery-mac ()
   (list '+  1  2 ))
(mystery-mac)
→  3
```


Defining kwote

- Define your own quote function named 'kwote:

```
(defmacro kwote (s-expr) (list 'quote s-expr))
(kwote fred) → fred
(list fred) →  error: fred is unbound
```

Understanding kwote

```
(defmacro kwote (s-expr) (list 'quote s-expr))
ENTER EVAL (kwote fred)
  ENTER EVAL-MACRO kwote
    BIND s-expr ←fred
    ENTER EVAL (list 'quote s-expr)
      ENTER EVAL 'quote
      EXIT → quote
      ENTER EVAL s-expr
      EXIT→ fred
    EXIT EVAL list → (quote fred)
  EXIT EVAL-MACRO → (quote fred)
  ENTER EVAL (quote fred)
  EXIT EVAL → fred
EXIT EVAL →fred
```


"backquote" facility

- Concise clean way to handle code generation with arguments
- The backquote ` introduces a "template"
- The comma , introduces substituable parameters
- The substitutions are evaluated once
- Compare versions

```
(defmacro kwote (s-expr) (list 'quote s-expr))
(defmacro kwote (s-expr) `(quote ,s-expr))
(defmacro kwote (s-expr) `',s-expr)
```

defmacro using backquote for arguments

```
(defmacro greet (name) `'( hello ,name ! ))

(greet richard) →
( hello richard ! )  ;; note:  richard unquoted

(greet (/ 0 0) ) →  ( hello (/ 0 0) ! )

(defmacro my-if (testF trueF falseF)
   `(cond (,testF ,trueF)
          (    t      ,falseF)))

(my-if t 'ok (/ 0 0)) →  ok
(my-if nil 'ok (/ 0 0)) →
error: zero divisor
```


Introducing local variables in macros

```
(defmacro
  arithmetic-if (test neg-form zero-form pos-form)
     (let ((var (gensym)))
       `(let ((,var ,test))
          (cond ((< ,var 0) ,neg-form)
                ((= ,var 0) ,zero-form)
                (    t      ,pos-form)))))
```

- gensym creates a new variable name
- This name is guaranteed not to be used already
- It cannot shadow variables in the neg, zero and pos forms

Variable length argument lists

- Like defun, defmacro accepts the &rest keyword

```
(defmacro random-form (&rest args)
   (nth (random (length args)) args) )

(random-form (car '(a b)) (+ 1 2) ) →   A
(random-form (car '(a b)) (+ 1 2) ) →   A
(random-form (car '(a b)) (+ 1 2) ) →   3
(random-form (car '(a b)) (+ 1 2) ) →   A
```

- Also works on this list

```
(random-form 'a x (- 27) (length '(t u x)) ) →
 -27
(random-form 'a x (- 27) (length '(t u x)) )
  →   error: x undefined
```


Comments on Macros

- Macros are not functions: they do not evaluate their arguments
- Macros are not special forms
  - Lisp defines a fixed set of special forms that evaluate arguments in special ways
  - Macros can alter its arguments, but must eventually express its computation in terms of special forms
- Macros may call other macros
- SETF is a macro

NLAMBDA and FEXPR

- Found in "classic" lisps
- No longer supported in common lisp and use is discouraged
  - Interfere with compiler optimization
  - Can interact strangely with dynamic scoping
- NLAMBDA is identical to LAMBDA but does not evaluate arguments before passing them to the body

  ```
  ( (LAMBDA (x) 'ok )  (/ 0 0) ) →
  error: divide by zero
  ( (NLAMBDA (x) 'ok )  (/ 0 0) ) →  'ok
  ( (NLAMBDA (x) x ) (/ 0 0) ) →  (/ 0 0)
  ```

- To evaluate arguments, you must explicitly call eval

  ```
  ( (NLAMBDA (x) (eval x)) (/ 0 0) ) →error: divide by ze
  ```

- In contrast, macros always pass result to evaluator

EVAL vs. APPLY

- APPLY does for functions what EVAL does for forms.
- APPLY takes a function name, a list of arguments, and a context; and
  applies the function to the arguments (using context as needed).
- EVAL: form + context ⇝ s-expr
  APPLY: function + args + context ⇝ s-expr

  ```
  (f s1 ... sn)  ⇔  (APPLY 'f '(s1 ... sn) nil)
  ```

Examples of APPLY

```
(APPLY 'CONS  '(A (B C))  nil)→ (A B C)
(APPLY 'CONS  '(X (C D E))  '((X .~P)) )
 → (X C D E)
(APPLY '(LAMBDA (a b) (CONS X b)
       '(Y (C D E)) '((X P)) )
 → (P C D E)
(APPLY 'APPEND  '((A B)(C D E))  '((X  P)) )
 → (A B C D E)}
(APPLY '(LAMBDA (x y) (EQ x y))
      '(A B) nil )  → nil
(APPLY '(LAMBDA (x y) (EQ x y))
       '(A B) '((x  B)(y  B)) )  → nil
(APPLY '(LAMBDA (x) (CONS (CAR x) w))
       '((A B C))
       '((x  (D E F))(w  (G H I))) )
→ (A G H I)
```


Examples of APPLY II

```
( (LAMBDA (a b) (APPLY 'EQ (LIST a b) nil))
             t t)
 → t

( (LAMBDA (x) (APPLY '(LAMBDA () (NULL nil))
                              ()
                             '((x  T)) ) )
            nil)
→ t
```

Examples of APPLY III

```
( (LAMBDA (x)
              (APPLY
                 '(LAMBDA () (ATOM x))
                 () () ) )
           nil)
→t
( (LAMBDA (x)
               (APPLY '(LAMBDA (y) (EQ x y))
                 '(T) '((x T)) ) )
            nil)
→t
```


Examples of APPLY IV

```
( (LAMBDA (x)
              (APPLY (FUNCTION
                         (LAMBDA (y) (EQ x y)))
                  '(T) '((x  T)) ) )
            nil)
→nil
```

Application of APPLY to Object Oriented Programming

- "Top Level" Operations:
  - (Add 17 22) Integer Addition
  - (Add 22.3 -4.2E-1) Real Addition
  - (Add (2 . 3) (9 . -7)) Complex Additionfor [2 + 3i] + [9 - 7i]
  - ...
- Same 'Add' operation, but different (Low level) code
- Other datatypes ? eg Matrix, Group, . . .
- Other operations ? eg Times, LessThan, . . .
- Problem: System must determine appropriate Code,dependent on DATA-Types of Args
- Solution: . . .

DataType with Associated Operations

- Integers and Reals:

Addition	(LAMBDA (x y) (+ x y))
Less-Than	(LAMBDA (x y) (< x y))

- Complex-Num:

Addition	(LAMBDA (x y) (CONS (+ (FIRST x) (FIRST y))
	(+ (SECOND x) (SECOND
Less-Than	(LAMBDA (x y) (AND (< (FIRST x) (FIRST y))
	(< (SECOND x) (SECOND

- Matrix:

Addition	(LAMBDA (x y) . . .)
Less-Than	(LAMBDA (x y) . . .)

Object-Oriented Programming II

- Code for add:

```
(DEFUN add (x y)
   (APPLY
      (Find-Addition-Method x y)   ;; implementation
      (LIST x y)                   ;; argument list
      nil))                        ;; context
```

- `Find-Addition-Method`
  1. Determine "data type" of args
     ["Real" for args `22.3`, `-15.2`]
  2. Find method for that operation
     for that data types
     (using default, inheritance, ...)
     ["`(LAMBDA (x y) (+ x y))`" for Real Addition]

APPLY Summary

- `Apply` is defined in Common Lisp
  (Context $\neq$ alist)

- It can take (only) 2 args:
  Function
  List of arguments
  (Context taken to be `nil`)

- Also `Funcall`:
  Like `Apply`, but takes $n + 1$ args:
  First is function;
  $i + 1^{st}$ is i^{th} arg to function.

More Examples of Apply

(apply '+ (3 5)) → 8

(funcall '+ 3 5)} → 8

(apply 'car '((a b c))) → a

(funcall 'car '(a b c))→ a

(apply '(lambda (x) (cadr x)) '((a b c))) → b

(funcall '(lambda (x) (cadr x)) '(a b c)) →b

(apply '(lambda (x y) (eq x y)) '(a b))→ nil

(funcall '(lambda (x y) (eq x y)) 'a 'b) → nil

And for your amusement

- What does this code do?

```
( (lambda (arg)
    (list arg
      (list (quote quote) arg)) )

  (quote
    (lambda (arg)
      (list arg
        (list (quote quote) arg)))) )
```

Lazy Computation

- Usually being "lazy" is bad
- When might it be good?
  - Unsure if compuation is necesary
  - When a computation might never halt
- Common Lisp does not directly support laziness (other languages do)
- Easy to add (but first, some examples)

Lazy Computation

- A typical Lisp calculation

  `(setf p (+ 2 3))` → `5`
  `p` → `5`
- Lazy calculations are introduced with "`delay`".
- A delayed computation can be restarted using "`force`".
- Example (not supported directly by Lisp)

  `(setf P (delay (+ 2 3)))` →
  `"#<DELAYED-COMPUTATION>"`
  `(force p)` → `5`

Lazy List Computation

- Lazy computations work well with recursive data-structures
- Define lazy "cons" which delays evaluation of its second argument

```
(setf p (lcons a (lcons b nil)))
→ (A . "#<DELAYED-COMPUTATION>")
(lcar p) →  A
(lcdr p) → (B . "#<DELAYED-COMPUTATION>")
(lcdr (lcdr p)) → ()

(setf q (lcons (+ 2 1) (+ 5 6))) →
(3 . "#<DELAYED-COMPUTATION>")
(lcar q) →  3
(lcdr q) → 11
```


Lazy List Computation

```
(setf q (lcons (+ 2 1) (setf x 5)))
x → undefined!
(lcdr q) → 5
x → 5
```

Infinite Computations

- What does this recursion compute?

```
(defun numbers (x)
   (lcons x (numbers (1+ x))))

(setf p (numbers 0))
(lcar p) →  0
(lcdr p) → (1 . "#<DELAYED-COMPUTATION>")
(lcar (lcdr p)) → 1
(lcar (lcdr (lcdr p))) →  2
(lcar (lcdr (lcdr (lcdr p)))) → 3
(defun fibset (f1 f2)
   (lcons f1  (fibset f2 (+ f1 f2))))
(setf q (fibset 1 1))
(lcar (lcdr (lcdr (lcdr (lcdr q))))) → 5
```


lfind-if Function I

- returns first element of lazy-list satisfying predicate pred

```
(defun lfind-if (pred llist)
 (cond ((funcall
          pred (lcar llist)) (lcar llist))
       (     t    (lfind-if pred (lcdr llist)))))
```

- Find smallest Fibonacci number greater than 342

```
(lfind-if
   (function (lambda (x) (>= x 342)))
   (fibset 1 1)) → 377
(lfind-if
   (function (lambda (x) (>= x 342)))
   (primeset)) → 347
```

lfind-if Function II

- As written, `lfind-if` may never return

```
(lfind-if
   (function (lambda (x) (< x 0)))
   (fibset 1 1)) → ERROR: STACK OVERFLOW
```

- Logic of set generator is decoupled from predicate tests
- Functional model without state or side-effects

```
(setf p (numbers 0))
(lcar (lcdr (lcdr p))) → 2

(lcar p) →  0

(setf q (lcons 9 (lcdr p)))
```

- Infinite sequence is not altered by accessors

Simplified Implementation of Laziness

- Define `delay` to freeze evaluation of expressions *in original lexical context*

```
(defmacro delay (form)
   `(function (lambda () ,form)))
```

- Define `force` to restart computation

```
(defmacro force (delayed-expression)
   `(funcall ,delayed-expression))
```

- Lazy list operators

```
(defmacro lcons (car cdr) `(cons ,car (delay ,cdr)))
(defmacro lcar  (cell)    `(car ,cell))
(defmacro lcdr  (cell)    `(force (cdr ,cell)))
```

- A more complex version might define a type for delayed computations