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
Consider the program

\( \text{LAMBDA} (\text{fn}) \ (\text{funcall fn fn}) \)

It uses \( \text{fn} \) as a program (function to be called) and as data (arguments for function)

We can call this function on a \( \lambda \)

\( \text{LAMBDA} (\text{fn}) \ (\text{funcall fn fn}) \)

\( '(\text{LAMBDA} (X) \ (\text{CAR} \ X)) \)

The \( \lambda \) argument is used as both program and data

\[ \equiv \ (\text{LAMBDA} (x) \ (\text{CAR} \ x)) \ ' (\text{LAMBDA} (x) \ (\text{CAR} \ x)) \]

\[ \rightarrow \ \text{LAMBDA} \]

Other examples

\( (\text{LAMBDA} (\text{fn}) \ (\text{funcall fn fn})) \ ' \text{ATOM} \)

\[ \rightarrow t \]

\( (\text{LAMBDA} (\text{fn}) \ (\text{funcall fn fn})) \ \text{CAR} \)

\[ \rightarrow \]

The function \( \text{CAR} \) (the variable) is undefined

\( (\text{LAMBDA} (\text{fn}) \ (\text{funcall fn fn})) \ ' \text{CAR} \)

\[ \rightarrow \ \text{Error: CAR expects a list!} \]

\( (\text{LAMBDA} (\text{fn}) \ (\text{funcall fn fn})) \ ' (\text{LAMBDA} (x) \ x) \)

\[ \rightarrow (\text{LAMBDA} (x) \ x) \]

\( (\text{LAMBDA} (\text{fn}) \ (\text{funcall fn fn})) \ ' (\text{LAMBDA} (x) \ (x \ x)) \)

\[ \rightarrow \ \text{The function } x \ \text{is undefined} \]

\( (\text{LAMBDA} (\text{fn}) \ (\text{funcall fn fn})) \ ' (\text{LAMBDA} (x) \ \text{funcall x}) \)

\[ \rightarrow \ \text{... waiting ... waiting ...} \]
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)) → (A C D)

- Can write a Lisp Function to do it!
  
  EVAL of `<form>` is `<form>`’s value.

  ![form](QUOTE <s - expr>)
  | (CAR <form>)
  | (CAR <form>)
  | (CONS <form> <form>)
  | t | nil

  `(EVAL '(CONS t nil)) → (t)`
  `(EVAL '(CONS (CAR '(A B)) '(C D))) → (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 $\mapsto$ s-expr
  
  (Common Lisp **EVAL** does not accept a context argument)

  $e \iff (\text{eval } 'e \text{ nil})$

  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 $= \text{nil}$
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

\((\text{EVAL } 'x ' ( (y x) (z A) (x P))) ) \rightarrow P\)

\((\text{EVAL } '(\text{CONS} (\text{CAR} x) y) ' ( (x (A B C)) (y (D E))) )\)

\rightarrow (A D E)\)

\((\text{EVAL } '(\text{QUOTE} x) ' ( (y x) (z A) (x P))) )\)

\rightarrow x\)

\(( (\text{LAMBDA} (x c) (\text{EVAL} x c)) 'W ' ( (W A) (X B) ) )\)

\rightarrow A\)

Trick:

\(> (\text{eval } \langle \text{form} \rangle)\) is the same as \(\langle \text{form} \rangle\)

That is: eval cancels quote ...
Extending the Language

▶ Common-Lisp defines
(IF ⟨test □form ⟩⟨true □form ⟩⟨else □form ⟩)
▶ 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) \rightarrow (+ 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) \rightarrow 3

- Result of foo-mac is evaluated!
Macro-functions with Arguments

- Ordinary function installed into the function symbol table
  \[
  \text{(setf (symbol-function 'foo-fun)}
  \text{(function (lambda (a1 a2 a3)}
  \text{ (list a1 a2 a3)))))
  \]
  (foo-fun 'cons 'a nil) →
  (cons a nil) ;; cons quoted!

- Macro-function installed in macro symbol table
  \[
  \text{(setf (macro-function 'foo-mac)}
  \text{(function (lambda (args ctx)}
  \text{ (let ((a1 (second args))}
  \text{ (a2 (third args)) (a3 (fourth args))}
  \text{ (list a1 a2 a3 ))))))}
  \]
  (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:
  \[
  \text{(defmacro name ( a1 ... an ) } \langle \text{form} \rangle)
  \]
- A macro has a list of formal arguments like LAMBDA or DEFUN
- Formal arguments are bound to unevaluated arguments supplied
- The \langle form \rangle is evaluated (\langle form \rangle may use arguments)
- The \textbf{result} of \langle form \rangle is returned in place of the macro
- Lisp evaluates returned \textbf{result}
The defmacro Special Form

- A function evaluates its body form and returns the result

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

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

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

Defining kwote

- Define your own quote function named 'kwote:

```lisp
(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 substitutable 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

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

```lisp
(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

```lisp
(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
  
  \[
  \begin{align*}
  &\quad (\text{LAMBDA}\ (x)\ 'ok\ )\ (\!0\ 0\!)
  \quad \rightarrow \quad \text{error: divide by zero} \\
  &\quad (\text{NLAMBDA}\ (x)\ 'ok\ )\ (\!0\ 0\!)
  \quad \rightarrow \quad 'ok \\
  &\quad (\text{NLAMBDA}\ (x)\ x\ )\ (\!0\ 0\!)
  \quad \rightarrow \quad (\!0\ 0\!)
  \end{align*}
  \]
- To evaluate arguments, you must explicitly call eval
  
  \[
  (\text{NLAMBDA}\ (x)\ (\text{eval}\ x))\ (\!0\ 0\!)
  \quad \rightarrow \quad \text{error: divide by zero}
  \]
- 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 \(\leadsto\) s-expr
  
  \[
  (f\ s1\ ...\ sn) \iff (\text{APPLY}\ 'f\ '(s1\ ...\ sn)\ \text{nil})
  \]
Examples of APPLY

(APPLY 'CONS '(A (B C)) nil) \rightarrow (A B C)

(APPLY 'CONS '(X (C D E)) '((X .~P)) )
\rightarrow (X C D E)

(APPLY '(LAMBDA (a b) (CONS X b))
 \quad '(Y (C D E)) '((X P)) )
\rightarrow (P C D E)

(APPLY 'APPEND '((A B)(C D E)) '((X P)) )
\rightarrow (A B C D E)}

(APPLY '(LAMBDA (x y) (EQ x y))
 \quad '(A B) nil ) \rightarrow nil

(APPLY '(LAMBDA (x y) (EQ x y))
 \quad '(A B) '((x B)(y B)) ) \rightarrow nil

(APPLY '(LAMBDA (x) (CONS (CAR x) w))
 \quad '((A B C))
 \quad '((x (D E F))(w (G H I))))
\rightarrow (A G H I)

Examples of APPLY II

((LAMBDA (a b) (APPLY 'EQ (LIST a b) nil))
 \quad t t)
\rightarrow t

((LAMBDA (x) (APPLY '(LAMBDA () (NULL nil))
 \quad ()
 \quad '((x T)) )

\quad nil)
\rightarrow t
Examples of APPLY III

\[
(\text{LAMBDA} (x) \\
\quad (\text{APPLY} \\
\quad \quad '(\text{LAMBDA} () (\text{ATOM} x))) \\
\quad \quad () () ) \\
\text{nil})
\rightarrow t
\]

\[
(\text{LAMBDA} (x) \\
\quad (\text{APPLY} '(\text{LAMBDA} (y) (\text{EQ} x y)) \\
\quad \quad '(\text{T} '((x \ T)) )) \\
\text{nil})
\rightarrow t
\]

Examples of APPLY IV

\[
(\text{LAMBDA} (x) \\
\quad (\text{APPLY} (\text{FUNCTION} \\
\quad \quad (\text{LAMBDA} (y) (\text{EQ} x y))) \\
\quad \quad '(\text{T} '((x \ T)) )) \\
\text{nil})
\rightarrow \text{nil}
\]
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 Addition for [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 y))))
  Less-Than  (LAMBDA (x y) (AND (< (FIRST x) (FIRST y))
                                       (< (SECOND x) (SECOND y))))

› Matrix:
  Addition   (LAMBDA (x y) . . .)
  Less-Than  (LAMBDA (x y) . . .)
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 ≠ 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 + 1st is ith 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) ))) )

(quote
(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 computation is necessary
  - When a computation might never halt
- Common Lisp does not directly support laziness (other languages do)
- Easy to add (but first, some examples)

A typical Lisp calculation

```lisp
(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)

```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

\[
\text{(setf } p \text{ (lcons a (lcons b nil)))} \rightarrow (A . \#\text{<DELAYED-COMPUTATION>}) \\
(lcar p) \rightarrow A \\
(lcdr p) \rightarrow (B . \#\text{<DELAYED-COMPUTATION>}) \\
(lcdr (lcdr p)) \rightarrow () \\
\]

\[
\text{(setf } q \text{ (lcons (+ 2 1) (+ 5 6)))} \rightarrow (3 . \#\text{<DELAYED-COMPUTATION>}) \\
(lcar q) \rightarrow 3 \\
(lcdr q) \rightarrow 11 \\
\]

\[
\text{(setf } q \text{ (lcons (+ 2 1) (setf x 5)))} \\
x \rightarrow \text{undefined!} \\
(lcdr q) \rightarrow 5 \\
x \rightarrow 5 \\
\]
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

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
1find-if Function II

- As written, 1find-if may never return

\[
(1\text{find-if}
  \quad (\text{function} \ (\lambda (x) \ (< x 0)))
  \quad (\text{fibset} \ 1 \ 1)) \rightarrow \text{ERROR: STACK OVERFLOW}
\]

- Logic of set generator is decoupled from predicate tests

- Functional model without state or side-effects

\[
\begin{align*}
\text{(setf p (numbers 0))} \\
\text{(lcar (lcdr (lcdr p)))} \rightarrow 2 \\
\text{(lcar p)} \rightarrow 0 \\
\text{(setf q (lcons 9 (lcdr p)))}
\end{align*}
\]

- Infinite sequence is not altered by accessors

Simplified Implementation of Laziness

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

\[
\text{(defmacro delay (form)}
  \quad \text{‘(function} \ (\lambda () ,\text{form)))}
\]

- Define force to restart computation

\[
\text{(defmacro force (delayed-expression)}
  \quad \text{‘(funcall} ,\text{delayed-expression))}
\]

- Lazy list operators

\[
\begin{align*}
\text{(defmacro lcons (car cdr) ‘(cons} ,\text{car (delay} ,\text{cdr)))} \\
\text{(defmacro lcar (cell) ‘(car} ,\text{cell))} \\
\text{(defmacro lcdr (cell) ‘(force} \ (\text{cdr} ,\text{cell)))}
\end{align*}
\]

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