Variables and Efficiency

- Variables are symbolic labels used to refer to data values
  - provided by the programmer, or
  - calculated from functions of data

- Variables allow us to refer to the same data multiple times

- Variables can improve efficiency - consider:
  
  \[
  Y := F(x) \times F(x) \quad \text{vs.} \quad Z := F(x) \\
  Y := Z \times Z
  \]

  - First example computes \( F(x) \) twice; Second example only once

  - Optimizing compilers can often detect simple redundancies, but it is important to be aware of the general principle
Examples in LISP

- How would we optimize the following code in Lisp:

  \[\text{(APPEND (foo x) (foo x))}\]

- Solution 1:

  \[
  \left(\text{(LAMBDA (z) (APPEND z z) (foo x))}\right)
  \]

- Alternatively, equivalently and more transparently

  \[
  \text{(LET ((z (foo x))) (APPEND z z))}
  \]

Using Functions Efficiently

- Consider the append predicate (see last lecture)

  \[
  \text{(DEFUN append (list1 list2) (COND ((NULL list1) list2 ) ( T (CONS (CAR list1) (append (CDR list1) list2)))))}
  \]
Analysis of Append

- What is \( \text{runTime}(\text{append}) \)?
  (Hint: examine reduction operator)

<table>
<thead>
<tr>
<th>(length L1)</th>
<th>(length L2)</th>
<th>#Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>11</td>
</tr>
</tbody>
</table>

- Running time of append is LINEAR in length of 1st arg
  \( \text{runTime(Append)} = O(\text{length}(L1)) \)

- Implication: always call with short list in first position

Efficiency Tricks

- First analysis of recursive structure may not yield an efficient solution

- Additional examination of the recursion can lead to significant improvements
Naive reverse implementation

(reverse '()) → ()
(reverse '(A)) → (A) ;; (APPEND () '(A))
(reverse '(B A)) → (A B) ;; (APPEND '(A) '(B))
(reverse '(C B A)) → (A B C) ;; (APPEND '(A B) '(C))

▶ Analysis
▶ Base case? '()→()
▶ Reduction? (CDR l1)
▶ Composition? (APPEND reduced-problem (LIST (CAR l1)))

▶ Solution based on this analysis (DO NOT IMPLEMENT!):

(DEFUN reverse-1 (l1)
   (COND ((NULL list) nil)
         (t (APPEND (reverse-1 (CDR l1))
                   (LIST (CAR l1)) ))))

Trace of Naive reverse-1

▶ The reverse-1 method starts by successively reducing the
   problem to the base case

(reverse-1 '(a b c d))
Enter reverse-1 (a b c d)
   Enter reverse-1 (b c d)
      Enter reverse-1 (c d)
         Enter reverse-1 (d)
            Enter reverse-1 nil

▶ As recursion unwinds, append is called at each step

Exit reverse-1 ()
Enter append () (d)
Trace of Naive reverse-1 II

Exit append (d)
Exit my-reverse-1 (d)
Enter append (d) (c)
Exit append (d c)
Exit my-reverse-1 (d c)
Enter append (d c) (b)
Exit append (d c b)
Exit my-reverse-1 (d c b)
Enter append (d c b) (a)
Exit append (d c b a)
Exit my-reverse-1 (d c b a)

Complexity of Naive reverse-1

(DEFUN reverse-1 (l1)
  (COND ((NULL l1) nil)
    ( t (APPEND (reverse-1 (CDR l1))
      (LIST (CAR l1)) ))))

▶ Each time reverse-1 completes, APPEND is called
▶ APPEND traverses the entire singly-linked list
  \[ \text{runtime(append)} = O(n) \]
▶ \[ \text{runtime(reverse - 1)} = n+(n-1)+\cdots+1 = \frac{n(n+1)}{2} = O(n^2) \]
LIST as STACK and Accumulators II

- Note: CONS operator is like a stack push and CAR is like stack pop

(SETF STK nil)
STK → ()
(SETF STK (CONS ’A STK))
STK → (A)
(SETF STK (CONS ’B STK))
STK → (B A)
(SETF STK (CONS ’C STK))
STK → (C B A)

- We push items into a lambda parameter named stk

(DEFUN load-stack (items stk)
 (COND ((NULL items) stk)
 ( t (load-stack
 (CDR items)(CONS (CAR items) stk))))))

- Don’t return composed result, pass it forward

- Stk is an accumulator variable returned on last call
Collector Variables or Accumulators

- Collector variable = extra argument in function that represents calculation so far
- When function is done (typically by exhausting another argument), it simply returns collector variable as value of function.
- Here: composition operator is identity function (it simply returns the result)

Using load-stack for my-reverse

- Using "helper function" load-stack to implement my-reverse

(DEFUN my-reverse (l1)
  (load-stack l1 nil))
- Internal definition of "helper function"

(DEFUN my-reverse (l1)
  (LABELS (
    (load-stack (items stk)
      (IF (NULL items)
        stk
        (load-stack (CDR items)
          (CONS (CAR items) stk))))))
  (load-stack l1 nil)))
- This version is $O(n)$!
Trace of efficient my-reverse

1 Enter my-reverse (a b c d)
1 Enter load-stack (a b c d) nil
2 Enter load-stack (b c d) (a)
3 Enter load-stack (c d) (b a)
4 Enter load-stack (d) (c b a)
5 Enter load-stack nil (d c b a)
5 Exit load-stack (d c b a)
4 Exit load-stack (d c b a)
3 Exit load-stack (d c b a)
2 Exit load-stack (d c b a)
1 Exit load-stack (d c b a)
1 Exit my-reverse (d c b a)

▶ Note: this implementation is tail-recursive

Efficiency in General

▶ Q: Is \( \langle fn_1 \rangle \) more efficient than \( \langle fn_2 \rangle \)?
  wrt expected Run Time Cost
  for LARGE problems

▶ Defined in terms of
  # of Function Applications
  as a function of “Size” of Argument(s)

▶ “Size”
  Usually Assymptotic
  “... for sufficiently large lists...”
  wrt LISP: Usually “length of list”
Efficiency Classes I

- **Constant Order** $O(1)$
  
  # of Function Applications is INDEPENDENT of args
  
  ... No recursion
  
  [Eg. \(\text{LAMBDA (x) (CAR (CDR x))}\)]

- **Linear Order** $O(n)$
  
  \(n\) is size of argument
  
  Recursive calls \(\propto\) length of list but CONSTANT work on each call
  
  - (e.g., APPEND ...(CONS (CAR x) (APPEND (CDR x) y))...)

Efficiency Classes II

- **Polynomial Order** $O(n^2), O(n^5), \ldots$
  
  Recursion on length of list, with Linear (poly) work at each level
  
  - (e.g. naive reverse-1 does an append after each call, so $O(n^2)$)

- **Exponential Order** $O(2^n), O(n^n), \ldots$
  
  More than 1 recursive call for each call
  
  - (e.g. naive fibonacci calls self TWICE at each step – stay tuned!)
(power n 0) \rightarrow 1
(power n 1) \rightarrow n
(power n 2) \rightarrow n^2 = n \times n
(power n 3) \rightarrow n^3 = n \times n \times n
(power n 4) \rightarrow n^4 = n \times n \times n \times n
:

\textbf{Analysis}

1. Base case? \((\text{power n 0}) \rightarrow 1\)
2. Reduction? \((- e 1)\)
3. Composition? \((\ast n (\text{power} (- e 1)))\)

\textbf{DEFUN my-power-2 (n e)}

\begin{verbatim}
(IF (= e 0)
  1
  (* n (my-power-2 n (- e 1))))
\end{verbatim}

\textbf{my-power-2} will be called \(e\) times, so it is linear in \(e\): \(O(e)\)
Logarithmic-time Power Function Analysis

(power n 0) → 1
(power n 1) → n
(power n 2) → n^2 = n*n = n*n
(power n 3) → n^3 = n*n*n = n^2 n
(power n 4) → n^4 = n*n*n*n = n^2 2
(power n 5) → n^5 = n*n*n*n*n = n^2 2 n

▶ Analysis
1. Base case? (power n 0) → 1
2. Reduction? If e odd: (- e 1)
   If e even: (/ e 2)
3. Composition?
   If e odd: (* n (power (- e 1))
   If e even: (* (p n e/2) (p n e/2))

Logarithmic-time Power Function Code

▶ Analysis
1. Base case? (power n 0) → 1
2. Reduction? Odd e: (- e 1); Even e: e/2
3. Composition? [see below]

(DEFUN my-power (n e)
  (COND
   ((= e 0) 1)
   ((EVENP e) (LET ((result (my-power n (/ e 2))))
     (* result result))
   (t (* n (my-power n (- e 1))))))

▶ Note: two distinct cases for recursive calls
Fibonacci Function Case Study

\[ \begin{align*}
\text{fib}(1) & \rightarrow 1 \\
\text{fib}(2) & \rightarrow 1 \\
\text{fib}(3) & \rightarrow 2 \\
\text{fib}(4) & \rightarrow 3 \\
\text{fib}(5) & \rightarrow 5 \\
\text{fib}(6) & \rightarrow 8 \\
\text{fib}(7) & \rightarrow 13 \quad ; \quad 13 = 5+8
\end{align*} \]

▶ Analysis

1. Base case? \( \text{fib}(1) \rightarrow 1, \text{fib}(2)\rightarrow1 \)
2. Reduction? \((- n 1) (- n 2)\)
3. Composition? \((+ (\text{fib} (- n 1)) (\text{fib} (- n 2)))\)

Naive Fibonacci

▶ Analysis

1. Base case? \( \text{fib}(1) \rightarrow 1, \text{fib}(2)\rightarrow1 \)
2. Reduction? \((- n 1) (- n 2)\)
3. Composition? \((+ (\text{fib} (- n 1)) (\text{fib} (- n 2)))\)

▶ A naive implementation (DO NOT IMPLEMENT)

\[
\begin{align*}
\text{(DEFUN fib1 (n)} & \\
& \quad (\text{COND ((< n 3) 1)} \\
& \quad \quad \quad (t (+ (fib1 (- n 1)) \\
& \quad \quad \quad (fib1 (- n 2)) ))))
\end{align*}
\]
Partial Trace of Naive Fibonacci

ENTER fib1 6 ;; Each call → 2 subcalls
ENTER fib1 5 ;; runtime(fib − 1) = O(2^n)
ENTER fib1 4
   ENTER fib1 3
      ENTER fib1 2 →1
      ENTER fib1 1→1
   ENTER fib1 2
ENTER fib1 3
   ENTER fib1 2→1
   ENTER fib1 1→1
ENTER fib1 4
   ENTER fib1 3
      ENTER fib1 2→1
      ENTER fib1 1→1
ENTER fib1 2

Linear Fibonacci

► Naive fib-1 generates 2 branches at (essentially) each call
► Build up answer from bottom forwards, using accumulators
  and stop when we have computed n terms
► n is #desired terms, I is a counter, fibI is ith fibonacci term,
  fibPrev is i − 1st fibonacci term

(DEFUN fib2 (n)
   (LABELS ( (fibHelp (n I fibI fibPrev)
      (IF (EQ n I)
         FibI
         (fibHelp n (+ I 1) (+ fibI fibPrev) fibI))
         (fibHelp n 1 1 0))))
Trace of Linear Fibonacci

ENTER: (FIB2 6)
ENTER: (FIBHELP 6 1 1 0)
ENTER: (FIBHELP 6 2 1 1)
ENTER: (FIBHELP 6 3 2 1)
ENTER: (FIBHELP 6 4 3 2)
ENTER: (FIBHELP 6 5 5 3)
ENTER: (FIBHELP 6 6 8 5)
ENTER: FIBHELP ==> 8
ENTER: FIBHELP ==> 8
... 
ENTER: FIBHELP ==> 8
ENTER: FIB2 ==> 8

- tail-recursive structure permits compiler optimization to linear loop

Sublinear Fibonacci I

- Define \( fib(n) \) to return vector \( \begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix} \)

- Base case: \( fib(2) = \begin{pmatrix} f_2 \\ f_1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \)

- Recursion:

\[
fib(n) = \begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} f_{n-1} + f_{n-2} \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f_{n-1} \\ f_{n-2} \end{pmatrix}
\]

- Still linear recursion
Sublinear Fibonacci II

- Sequence of recursive calls has its own shared substructure

\[
\text{fib}(n) = \begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} f_{n-1} + f_{n-2} \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f_{n-1} \\ f_{n-2} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f_{n-2} \\ f_{n-3} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2 \begin{pmatrix} f_{n-2} \\ f_{n-3} \end{pmatrix}
\]

Sublinear Fibonacci III

- On repeated substitution all the way down to the base case:

\[
\begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-2} \begin{pmatrix} f_2 \\ f_1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-2} \begin{pmatrix} 1 \\ 1 \end{pmatrix}
\]

- Examples:
  
  \[
  \begin{pmatrix} f_2 \\ f_1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^0 \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix},
  \]
  
  \[
  \begin{pmatrix} f_3 \\ f_1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1 \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}
  \]
Sublinear Fibonacci IV

- Showed (power n e) has time logarithmic in exponent
- Substituting matrix multiplication for '*' implements matrix power
- Showed Fibonacci can be reduced to matrix exponentiation
- Fibonacci can therefore be computed in logarithmic time

Scope of Variables

- Consider function bindings of variables in ⟨form⟩ given:

(LAMBDA (y z) ⟨form⟩)
  - y refers to 1\textsuperscript{st} arg
  - z refers to 2\textsuperscript{nd} arg

- Consider \textit{nested} functions

(LAMBDA (y z)
  (LAMBDA (x v) ⟨form⟩)
  (CDR y) 'A)))

Variables usable within ⟨form⟩:
  - x is bound to CDR of 1\textsuperscript{st} arg
  - v is bound to value A
  - y is bound to 1\textsuperscript{st} arg
  - z is bound to 2\textsuperscript{nd} arg
Variables

(LAMBDA (y z)
  (LAMBDA (x y) (LIST x y z)) 'A (CDR y))

- is a function that takes 2 args, and evaluates to 3 element list:
  ( A (CDR of 1\textsuperscript{st} arg) (2\textsuperscript{nd} arg) )

- Notation: In inner $\lambda$-expr
  - Variable $x$ and $y$ are BOUND
  - Variable $z$ is FREE
  - $y$’s value is “shadowed” by (CDR $y$)

Dealing with Free Variables

- Def’n: Formal variables of a function are **bound** within the function definition.

- All other variables are **free**.

- Consider function

  (DEFUN foo (z x) (LIST z x y) )

- When (foo 5 t) is called
  - $z \rightarrow 5$, bound
  - $x \rightarrow t$, bound
  - $y$ will be FREE

- Is (foo ⟨f1⟩ ⟨f2⟩) always defined? No
Evaluating foo

Case 1: ( (LAMBDA (x y) (foo 'A (CDR x))) '(5) t)

Eval: "(foo 'A (CDR x))" with x ← (5), y ← t
Eval: "(LIST z x y)" with z ← A, x ← (), y ← t
Returns: (A () t)

Case 2: ( (LAMBDA (x) (foo 'A (CDR x))) '(5) )

Eval: (foo 'A (CDR x)) with x ← (5)
Eval: (LIST z x y) with z←A, x←(), y undefined
Undefined!!

Scope: Dynamic vs Static

Dynamic Scoping:
Value of variable depends on RUN-time situation!
EG: Lisp

Static Scoping:
Value of variable determined by COMPILE-time declaration.
EG: Pascal, Turing, ...

Examples …
Example of Static Scoping

```
Foo
  x ← 5

Bar
  ... x ...
  put x;  {value is "5"}

Glob
  x ← 7
  ... Bar() ...
  put x;  {prints 7}
```

Example of Dynamic Scoping

```
(SETQ x 20) → 20
(SETQ y 10) → 10
(DEFUN plusy (x) (+ x y)) →
  plusy  ;;; y is free
  (plusy 5) → 15
  (+ x y) → 30
(SETQ y 20) → 20
(plusy 5) → 25
```
Contexts

- Identify each variable with a (LIFO) STACK of values.
- Variable’s current “value” is top of stack
- Initializing/Updating Variable’s Stack
  - Initially, each variable’s stack is [undefined]
  - If (SETQ a v), reset top of a’s stack to (value of) v.
  - When entering function fn with args a₁, ..., aₙ, bound to values v₁, ..., vₙ
    PUSH the value of vᵢ onto aᵢ’s stack for each i
  - When exiting function, POP stack of each of function’s variables

Maintaining Contexts

Evaluate:

```
((LAMBDA (x y)
  ((LAMBDA (z x) (LIST z x y))
    'a (CDR x) )
  )
 '(A B C) '(D E F) ) with x←[], y←[], z←[]
ENTER λ₁(x y) with x←[(A B C)], y←[(D E F)], z←[]
ENTER λ₂(z x) with x←[(B C)(A B C)], y←[(D E F)], z
EXIT λ₂ with (A (B C) (D E F))
EXIT λ₁ (A (B C) (D E F))
```
Examples of Tracing

(DEFUN foo (x y) (APPEND x (bar y)))
(DEFUN bar (p) (IF (NULL p) x (foo y (CDR p))))

Evaluate (FOO '(A) '(B C))

enter FOO { X←(A), Y←(B C) }
enter BAR { P←(B C) }
enter FOO { X←(B C), Y←(C) }
enter BAR { P←(C) }
enter FOO { X←(C), Y←() }
enter BAR { P←() }
return (C)
return (C C)
return (C C)
return (B C C C)
return (B C C C)
return (A B C C C)

Functional Arguments – Revisited

- Can take a function as argument
treat it as an s-expr
“apply” it

- Dynamic vs Static Scoping
QUOTE vs FUNCTION
Successor Function

- '1+' generates the numeric successor of its argument

\[
(1+ 0) \rightarrow 1 \\
(1+ 1) \rightarrow 2 \\
(1+ 1.5) \rightarrow 2.5 \\
(1+ (\sqrt{2})) \rightarrow 2.41421374 \\
(1+ (/ 3 9)) \rightarrow \frac{4}{3}
\]

Mapping Function: plus1

- Applies a function to each element of list.

- Eg 1: Add 1 to each element:

\[
(\text{DEFUN plus1 (list)} \\
(\text{IF (NULL list)} \\
\text{nil} \\
(\text{CONS (1+ (\text{CAR list}))} \\
(\text{plus1 (\text{CDR list}})))) \\
(\text{plus1 \ (list 3 -10 (\sqrt{2}) (/ 4 7)}) \\
\rightarrow (4 -9 2.4142137 11/7)
\]
Mapping Function: carAll

- Eg 2: Take CAR of each element:

\[
(\text{DEFUN carAll} \ (\text{list})\\
\quad (\text{IF} \ (\text{NULL} \ \text{list})\\
\quad \quad \text{nil}\\
\quad \quad (\text{CONS} \ (\text{CAR} \ (\text{CAR} \ \text{list})))\\
\quad \quad \quad (\text{carAll} \ (\text{CDR} \ \text{list})))\))\\
\text{(CarAll ')((A B) (C D E) (t) (5 A)))}\\
\rightarrow (A \ C \ t \ 5)
\]

Mapping Function – MAPCAR

- Each mapping function has
  - a recursive loop over list elements
  - applying some specific function to each element

- Use higher-order function to define common parts!

- Pass in list and function to apply

\[
(\text{DEFUN MAPCAR} \ (\text{list} \ \text{fn})\\
\quad (\text{IF} \ (\text{NULL} \ \text{list})\\
\quad \quad \text{nil}\\
\quad \quad (\text{CONS} \ (\text{funcall} \ \text{fn} \ (\text{CAR} \ \text{list}))\\
\quad \quad \quad (\text{MAPCAR} \ (\text{CDR} \ \text{list}) \ \text{fn})))\))\\
\]

- MAPCAR is built into Common Lisp
MAPCAR Examples

(MAPCAR '(3 5 0) '1+) → (4 6 1)
(MAPCAR '( (4) (t Q) ) 'CAR) → (4 t)
(MAPCAR '( (4) (t Q) ) 'CDR) → ( () (Q))
(MAPCAR '( (4) (t) ) 'LISTP) → (T T)
(MAPCAR '( A B (C D) ) 'ATOM → (T T nil)
(MAPCAR '() 'ATOM) → ()
(MAPCAR '(A B C) '(LAMBDA (x) (CONS x '(t) ))) →
( (A t) (B t) (C t) )

Mapping Function – AnyOf

- True if any element of list x satisfies the predicate function fn
(Note carefully: list argument is named x)

(DEFUN AnyOf (fn x)
  (COND ((NULL x) nil)
        ((funcall fn (CAR x)) t)
        ( t (AnyOf fn (CDR x))) ))

- An alternative definition emphasizing readability (might lose tail-recursion)

(DEFUN AnyOf-2 (fn list)
  (AND (NOT (NULL list))
       (OR (funcall fn (FIRST list))
           (AnyOf-2 fn (REST list))))
AnyOf Examples

(AnyOf 'ATOM '(A B (C D))) → t
(AnyOf 'ATOM '((4) (t Q))) → nil
(AnyOf 'LISTP '((4) (t Q))) → t
(AnyOf 'CAR '((4) (t))) → t
(AnyOf 'CDR '((4) (t))) → nil
(AnyOf 'ATOM ()) → nil
(AnyOf '(LAMBDA (y) (EQ y 'A)) '(B A C)) → t

Mapping Functions

- Apply function to each [element | sublist] of list, returning list of values.
  - MapCar applies function to each element of list, returning list of values.
  - MapList — like MAPCAR, but uses successive SUBLISTS (not elements)
  - MapCan, MapCon ... destructive (Not PURE lisp)

- Apply function to each [element | sublist] of list, returning nil.
  (used for side effect – eg printing values. Not PURE lisp)
  - MapC — like MapCar, but returns nil
  - MapL — like MapList, but returns nil

- “Boolean” Functions (not in Common Lisp)
  - ANYOF determines if any element satisfies predicate.
  - ALLOF determines if all elements satisfy predicate.
Using functions with free variables can cause problems

We might expect \texttt{memq} to return \texttt{t} if \texttt{at} is in \texttt{list}

\begin{verbatim}
(DEFUN memq (at list)
  (AnyOf '(LAMBDA (i) (EQ i at)) list ))
\end{verbatim}

Not necessarily true:

Note: \texttt{at} is inside a quoted expression
→ it is not scoped in the context of \texttt{defun memq}

Therefore \texttt{at} is a \textit{Free Variable} within inner $\lambda$-expr.

\section*{MEMQ with DYNAMIC Scoping}

In a Lisp with dynamic scoping
(e.g. Franz lisp but not Common Lisp),
variables are resolved by checking bindings upwards along the stack

\begin{verbatim}
(DEFUN memq (at l)
  (AnyOf '(LAMBDA (i) (EQ i at)) l ))
\end{verbatim}

The \texttt{at} in the $\lambda$ is unbound within the $\lambda$

But, \texttt{memq} calls \texttt{AnyOf} which calls $\lambda$

\begin{verbatim}
(DEFUN AnyOf (fn x)
  (COND ((NULL x) nil)
        ((funcall fn (CAR x)) t)
        ( t (AnyOf fn (CDR x))))))
\end{verbatim}

The \texttt{at} binding created by \texttt{memq} will resolve \texttt{at} in $\lambda$
Tracing MEMQ with DYNAMIC Scoping I

(memq 'a '(b a c))
Enter memq {at←a, l←(b a c)}
Enter AnyOf {fn←(LAMBDA (i) (EQ i at))
   x←(b a c) } Enter λ(fn) {i← b} EVAL (EQ i at) {i←b, at←a} ↠nil

▶ Here, at is resolved against the binding made further up the stack ... so computation continues normally

MEMQ with DYNAMIC Scoping II

▶ Now rename at to x, but the x in λ is still free
(DEFUN memq (x i)
   (AnyOf ' (LAMBDA(i) (EQ i x)) i))

▶ Recall AnyOf uses parameter x as well
(DEFUN AnyOf (fn x)
   (COND ((NULL x) nil)
       ((funcall fn (CAR x)) t)
       ( t (AnyOf fn (CDR x)))))

▶ Again: memq calls AnyOf which calls λ

▶ Here, AnyOf has left closest binding to λ of x on the stack
(memq 'a '(b a c))
Enter memq {x ←a, l ←(b a c)}
Enter AnyOf {fn ←(LAMBDA (i) (EQ i x)), x ←(b a c)}
Enter λ { i ←b }
EVAL (EQ i x ) {i ←b, x ←(b a c)}
→ERROR, as x is (b a c)

▶ The λ retrieves closest x on the stack, which is bound by AnyOf
▶ The λ requires x to be a executable expression: error!

FunArg Problem

▶ If Dynamic Scoping,

(LAMBDA (at L) (AnyOf L (LAMBDA (i) (EQ i at))) )
(LAMBDA (x L) (AnyOf L (LAMBDA (i) (EQ i x))) )
can have completely different results,
as x and at are free within λ

▶ Want x evaluated STATICALLY (based on program definition)
Not DYNAMICALLY (based on run-time environ.)

▶ Older Lisp’s usually evaluates free variables DYNAMICALLY.

▶ To get STATIC evaluation use new special form: FUNCTION
MEMQ without DYNAMIC Scoping

- Dynamic scoping can introduce subtle and hard-to-find errors
- In Lisp's without dynamic scoping (e.g., Modern Common Lisp), the \( x \) in quoted \( \lambda \) is still unbound

```
(DEFUN memq (x l)
  (AnyOf '(LAMBDA(i) (EQ i x)) l))
```

- Without dynamic scoping, \( x \) cannot be resolved on the stack

Tracing memq without Dynamic Scoping

```
(memq 'a '(b a c))
Enter memq \{x←a, l←(b a c)\}
Enter AnyOf \{fn←(LAMBDA (i) (EQ i x)), x←(b a c) \}
Enter \( \lambda \) \{ i←b \}
EVAL (EQ i x ) \{i←b,x←(b a c)\}
\( \sim \)ERROR, as \( x \) undefined!
```

- \( x \) cannot be resolved
QUOTE is for Dynamic Scoping

- Dynamic Scoping: free variables isolated by quote

\[
\text{(DEFUN memq1 (x l) (
    (AnyOf (QUOTE (LAMBDA (i) (EQ i x)))
    l))}
\]

- In Lisps that support dynamic scoping, free variables are evaluated DYNAMICALLY

- Hence: value of \( x \) in memq1's is value of Any0f's 2\(^{nd} \) arg.

\[
\text{(QUOTE (LAMBDA (i) (EQ i x)))}
\]

- FunArg problem!

FUNCTION Specifies Static Scoping

- Static Scoping

\[
\text{(DEFUN memq2 (x l) (
    (AnyOf (FUNCTION (LAMBDA (i) (EQ i x)))
    l))}
\]

- Free variables are evaluated STATICALLY
  - bindings are taken from the environment where \( \lambda \) was defined

- As it "sees" the \( x \) in memq2, that is the value it will take
Function Special Form

- FUNCTION behaves exactly like QUOTE except wrt evaluation of free variables:

- FUNCTION \(\approx\) STATIC EVALUATION
  [based on (compile-time) function definition]

- QUOTE \(\approx\) DYNAMIC EVALUATION
  [based on current (run-time) context]

- Lisp's Compiler can compile
  (function (LAMBDA (...) ...))

MEMQ with STATIC scoping

- In both Lisps with dynamic scoping and those without, the FUNCTION form introduces static scoping

  (DEFUN memq (x l)
    (AnyOf (FUNCTION (LAMBDA (i) (EQ i x))) l ))

- The \(x\) in the \(\lambda\) is resolved in the scope of memq
  so it is bound to the first parameter of memq

- Again, memq calls AnyOf which calls the \(\lambda\)

  (DEFUN AnyOf (fn x)
    (COND ((NULL x) nil)
      ((funcall fn (CAR x)) t)
      ( t (AnyOf fn (CDR x))) ))

- But, the \(x\) in AnyOf cannot interfere with the \(x\) in \(\lambda\)
Factory Method Example I

- In the absence of some global definition or binding higher up on the stack

```lisp
(defun dynamic-funs (x)
  (list (quote (lambda () x))
        (quote (lambda (y) (setq x y))))
(setf funs (dynamic-funs 6))
(funcall (first funs)) → variable x unbound
```

Factory Method Example II

- If a global definition exists, it can be used

```lisp
(defun dynamic-funs (x)
  (list (quote (lambda () x))
        (quote (lambda (y) (setq x y))))
(setf x nil)
(setq funs (dynamic-funs 6))
(funcall (first funs)) → nil
(funcall (second funs) 5) → 5
(funcall (first funs)) → 5
```
Even in Lisp's with static binding, function is necessary to tell the compiler that static scoping is desired for an expression

```
(defun static-funs (x)
    (list (function (lambda () x))
           (function (lambda (y) (setq x y)))))
(setq funs (static-funs 6))
(funcall (first funs)) → 6
(funcall (second funs) 43) → 43
(funcall (first funs)) → 43
```

Note: it is possible to create "objects" this way that have local data protected by accessor methods