1

Variables

- Variables are placeholders for "holding" values.
- Values come from the evaluation of forms
- Variables can be used in Lisp to pass arguments to a function or to aid in efficiency!

©1993 John Mylopoulos

2

Efficiency via Variables

Y := F(x) × F(x)

compute F(x) twice

vs

Z := F(x)
Y := Z × Z

compute F(x) only once!

In Lisp

```
(lambda (x) (append (foo x) (foo x)))
```

vs

```
(lambda (x) ((lambda(z)
               (append z z)) (foo x)))
```

Here `(lambda (z) (append z z))` is a helping function whose argument is bound to the value returned by the form `(foo z)`, thereby avoiding a double evaluation.

Helping functions are auxiliary functions that are used to help a top level function compute its value. Helping functions may call: by other helping functions, themselves or the top-level function

©1993 John Mylopoulos

3

Efficiency Through Variables

Consider the `reverse` function where:

`(reverse '(a b c d e))` is `(e d c b a)`

`reverse` is defined by:

```
(defun reverse (x)
   (cond ((null x)  nil)
         (t  (append   (reverse (cdr x))
                 (list (car x)))  )))
```

Unfortunately, this is very inefficient because `reverse` traverses the (partial) list it is supposed to reverse at each step!

...

1: Traverse list to pick up `e`; Build up `(e)`

2. Pick up `d`; Traverse list `(e)` to build up `(e d)`

3. Traverse list `(e d)` to build up `(e d c)`

...

5: Build up `(e d c b a)` after traversing `(e d c b)`

©1993 John Mylopoulos

4

Trace of inefficient `reverse`

```
-> (reverse '(a b c d e ))

Enter reverse (a b c d e)
   Enter reverse ( b c d e)
      Enter reverse (c d e)
         Enter reverse (d e)
            Enter reverse (e)
               Enter reverse nil
               Exit  reverse nil
               Enter append  nil   ( e)
               Exit  append  ( e)
            Exit  reverse (e )
         Enter append  (e )  ( d)
         Exit  append  (e d )
         Exit  reverse (e d )
      Enter append  (e d )  ( c)
      Exit  append  (e d c )
      Exit  reverse (e d c )
   Enter append  (e d c )      ( b)
   Exit  append  (e d c b )
   Exit  reverse (e d c b )
Enter append  (e d c b)  ( a)
Exit  append  (e d c b a)
Exit  reverse (e d c b a)
```


©1993 John Mylopoulos

5

Efficient `reverse` Definition (in Pseudo Code)

```
answer <- nil  ;  y  <- x
while y ≠ nil do
   answer  <- (cons (car y) answer)
   y       <- (cdr y)
end-do
return (answer)
```

Initially:

```
answer= nil          y=(a b c d e)
```

After 1 iteration

```
answer= (a)          y= (b c d e)
```

After 2 iterations

```
answer= (b a)        y= (c d e)
```

After 3 iterations

```
answer= (c b a)      y= (d e)
```

After 4 iterations

```
answer= (d c b a)    y= (e)
```

After 5 iterations

```
answer= (e d c b a)  y= ()
```


©1993 John Mylopoulos

6

Efficient reverse in FORTRAN-like Lisp

```
(defun  mreverse  (x)
   (prog (y answer)  ;; local variables
      (setf  answer nil)
      (setf y  x)
      loop
      (cond ((null y)(return answer)))
      (setf answer (cons (car y) answer))
      (setf y (cdr y))
      (go loop)
      ))
```

This solution is efficient because it traverses input list only once by building up `answer` while reducing `x`

Uses a collector variable (`answer`) and an auxiliary variable (`y`)

Using `prog` is BAD functional programming style!!!

©1993 John Mylopoulos

7

Collector Variable

A *collector variable* is an extra argument in a (helping) function that represents the calculation so far.

When the (helping) function is done, (typically by exhausting another argument) it simply returns the collector variable as the value of the function.

Efficient `reverse:`

```
(defun  revloop  (y  answer)
 (cond  ((null y)  answer)
        (t (revloop (cdr y)
           (cons (car y) answer)))))

(defun  reverse (x)
   (revloop x nil))
```

Note: `revloop` is a helping function to `reverse`, using tail recursion (composition operator is identity).

This is as efficient as iteration (why?).

©1993 John Mylopoulos

8

Trace of efficient `reverse`

```
-> (reverse '(a b c d e ))

Enter reverse (a b c d e)
   Enter revloop (a b c d e)   nil
      Enter revloop (b c d e)   (a)
         Enter revloop (c d e)   (b a)
            Enter revloop (d e)   (c b a)
               Enter revloop (e)   (d c b a)
            Enter revloop nil (e d c b a)
            Exit  revloop (e d c b a)
               Exit  revloop (e d c b a)
            Exit  revloop (e d c b a)
         Exit  revloop (e d c b a)
      Exit  revloop (e d c b a)
   Exit  revloop (e d c b a)
Exit  reverse (e d c b a)

(e d c b a)
```


©1993 John Mylopoulos