1

Functional Arguments: Revised Rule for Evaluating a List as a Form

Remember the rule that the `first` of a form determines its type, and therefore the way it will be evaluated

first		kind of form
symbol	special form name	special form
symbol	macro name	macro call
symbol	function name	function invocation
s-expr	lambda expression	function invocation

For some Lisps, such as Scheme, if none of the above, and `first` is an atom then evaluate the atom, and use the value of the atom instead to evaluate the form.

For example, assume that the symbol `boo` has as value the atom `cons`. Then:

```
--> boo
   cons

-->   ( boo   'a   '(b) )
   (a b)
```

In Common Lisp you need to write

```
(funcall boo 'a '(b))
```


©1993 John Mylopoulos

2

Functional Arguments

Generally, it is possible to use a parameter of a function as the car of a form in the body of the function. The values of such variables are **functional arguments**.

Consider the following function `operate` which takes as first argument a function to be applied to the second argument and the `cdr` of the third:

```
(defun operate  (fun a b)
   (funcall fun  a (cdr b) ))
```

then for example:

```
--> (operate  'cons  1  '(b c d))
   (1 c d)
--> (operate 'append '(a b c) '(d e f))
   (a b c e f)
--> (operate
       '(lambda (p q) (cons p (car q)))
                2 '((v w) (x)))
   (2 x)
```


©1993 John Mylopoulos

3

...but:

```
--> (defun boo (a b)    (list a a b))
   boo

-->(boo 1 '(b c))
   (1 1 (B C))

-->(funcall boo 1 '(b c))
(1 B C)

--> (operate  'boo 1 '(b c d))
   (1 1 (c d))

--> (operate boo 1 '(b c d))
   (1 c d)
```


©1993 John Mylopoulos

4

Mapping Functions : Examples

Sometimes we want to apply a function to each element of a list. For example we may wish to add 1 to each element:

```
(defun plus1 (l)
   (cond ((null l) nil)
      (t    (cons (1+    (car  l)))
                     (plus1 (cdr l))) )))

   --> (1+ 5)
      6
   --> (plus1 '(3 5 0))
      (4 6 1)
```

Or, suppose we wish to take the `car` of each element of a list:

```
(defun carall (l)
   (cond ((null l)     nil)
      ( t   (cons ( car  (car l)))
                  (carall  (cdr l))) )))

   --> (carall '((a b) (c d e) (t) (5 x)))
      (a c t 5)
```


©1993 John Mylopoulos

5

Mapping Function - `mapcar`

`mapcar` of function `fn` and list `x` applies `fn` to each element in `x` and returns as value the resulting list.

List	function	result
(3 5 0)	1+	(4 6 1)
((4)(t q))	car	(4 t)
((4) (t q))	cdr	(() (q))
((4)(t q))	listp	(t t)
(a b (c d))	atom	(t t nil)
()	atom	()
(a b c)	(lambda (x)(cons x '(t)))	((a t)(b t)(c t))

`mapcar` may be defined as:

```
(defun  mapcar (fn   l)
   (cond ((null l)    nil)
         (t (cons (funcall fn  (car  l))
                  ( mapcar  fn  (cdr  l) )))))
```


©1993 John Mylopoulos

6

Mapping Function - `anyof`

`anyof` of function `fn` and list `x` determines if any element of `x` satisfies predicate `fn` and returns `t` or `nil` accordingly.

function	list	result
atom	(a b (c d))	t
atom	((4)(t q))	nil
listp	((4) (t q))	t
car	((4) (t))	t
cdr	((4) (t))	nil
atom	()	nil
(lambda (y) (eq y 'a))	(b a c)	t

`anyof` may be defined as:

```
(defun anyof  (fn l )
   (cond ((null l)     nil)
         (( funcall fn  (car l)) t)
            (t    (anyof  fn  (cdr l)))  ))
```

Likewise, `allof` determines if all the elements of a list satisfy a predicate function.

©1993 John Mylopoulos

7

Mapping Functions Returning Lists

Some mapping functions apply a function to each element or sublist of a list and return a list of values. For example:

`mapcar` applies a function to each element of a list, returns a list of values.

`maplist` like `mapcar`, but uses successive sublists (not elements)

`mapcan` like `mapcar`, but removes a layer of parentheses from the resulting list.

For example, suppose

```
--> (mapcar   'foo ' (a b  (c d) e))
   ((x a) (x b) () (x e))
```

(what does `foo` do?). Then

```
--> (mapcan  'foo '(a b (c d) e))
   (x a x b x e)
```

`mapcon` is a cdr version of `mapcan`

©1993 John Mylopoulos

8

...What Does `foo` Do?

```
(defun  foo (x)
   (cond ((atom x)  (list 'x x))
         (t nil)    ))
```


©1993 John Mylopoulos

9

Free Variables interact with Functional Arguments

Using a functional argument for a function with free variables in its body can cause unexpected results in Lisp dialects which adopt dynamic scoping.

For example, consider two alternative definitions of the function `memq`. Both `memq`'s are functions of two arguments that return `t` if the first argument is an element of the second argument (a list) using the previously defined `anyof`:

- `memq` with first argument named `at`:

```
(defun  memq (at l)
   (anyof '(lambda (i) (eq i at)) l  ))
```

- `memq` with first argument named `x`:

```
(defun  memq (x l)
   (anyof '(lambda (i) (eq i x))  l  ))
```

Note that `at` and `x` are free variables within the inner lambda expression.

©1993 John Mylopoulos

10

Free Variables interact with Functional Arguments

Assume that `anyof` is defined as before:

```
(defun anyof  (fn x)
   (cond ((null x)    nil)
      ((funcall fn  (car x)) t)
      (t   (anyof  fn  (cdr x)))  ))
```

- `memq` with first argument named `at`

```
(defun  memq (at l)
 (anyof '(lambda (i)(eq i at)) l))
```

```
--> (memq 'a '(a b c))
```

Enter `memq`: `at/[a]  l/[(a b c)]  i/[]`
`fn/[]  x/[]`

Enter `anyof`: `at/[a]  l/[(a b c)] i/[]`
`fn/[(lambda (i) (eq i at))] x/[(a b c)]`

Enter lambda expr: `at/[a]  l/[(a b c)] i/[a]`
`fn/[(lambda (i) (eq i at))]  x/[(a b c)]`

Exit `lambda` expr with `t`
Exit `anyof` with `t`
Exit `memq` with `t`

`t`

©1993 John Mylopoulos

11

Free Variables interact with Functional Arguments

Assume that `anyof` is defined as before:

```
(defun anyof  (fn x)
   (cond ((null x)     nil)
         ((funcall fn  (car x)) t)
         (t    (anyof  fn  (cdr x)))  ))
```

• `memq` with first argument named `x`

```
(defun  memq (x l)
   (anyof '(lambda (i) (eq i x)) l))
```

```
--> (memq 'a '(a b c))
```

Enter `memq`: `l/[(a b c)]  i/[]`
`fn/[]  x/[a]`
Enter `anyof`: `l/[(a b c)]  i/[]`
`fn/[(lambda (i) (eq i x))]`
`x/[(a b c) a]`
Enter lambda expr: `l/[(a b c)]  i/[a]`
`fn/[(lambda (i) (eq i x ))]`
`x/[(a b c)  a]`

Evaluate `(eq i x)`: **??? ERROR/nil???**
`x` is `(a b c)` instead of expected `a`

(Common Lisp gives an error because of rules limiting the use of free variables)

©1993 John Mylopoulos

12

The "FunArg" Problem: Free Variables Interact with Functional Arguments

```
(defun anyof  (fn x )
      (cond ((null l)     nil)
            ((funcall fn  (car x)) t)
            (t    (anyof  fn  (cdr x)))  ))
```

• `memq` with first argument named `at`:

```
(defun  memq (at l)
   (anyof '(lambda (i) (eq i at)) l))
```

• `memq` with first argument named `x`:

```
(defun  memq (x l)
   (anyof '(lambda (i) (eq i x)) l))
```

These two `memq`'s are different functions! In the second case, what we mean is that we want the free variable `x` to be evaluated statically (based on the program definition) rather than dynamically (based on the run-time environment).

©1993 John Mylopoulos

13

`function` is for Static Scoping

Back to `memq` with first argument named `x`:

```
(defun  memq (x l)
 (anyof (function  (lambda (i) (eq i x)))
                                    l  ))
```

In general, `function` causes free variables to be evaluated statically

```
(function (lambda (a1 ... an) form))
```

Hence: the value of `x` in `memq`'s

```
    (function (lambda (i) (eq i  x )))
```

when evaluated by `anyof`, is the value of `memq`'s first argument.

This is the desired result!!

`quote` causes free variables to be evaluated dynamically.

`function` causes free variables to be evaluated statically.

14

Common Lisp Example

```
>(setq dog 2)
2

>(defun huh (d fn) (funcall fn d))
HUH

>(let ((dog 10))
 (huh 3 (quote (lambda (x) (* dog x))) ))
6

>(let ((dog 10))
    (huh 3 (function
             (lambda (x) (* dog x))) ))
30
```

Dynamic scoping leads to `dog`'s global value (`2`); static scoping leads to its statically bound value (`10`)