1

Programs as Data

Unlike most other languages, Lisp can treat programs (functions) as data (values) and data (function definitions) as programs.

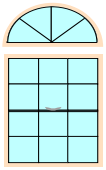
Why Do it?

• To take apart a program for analysis.

• To put together a program at run-time, and then running it.

• To modify the behavior of existing programs. (ie. implement trace, or additional side-effects)

• To merge together programs!
....



2

Functions that Use Programs as Data

• Functions that treat data as programs in Lisp:

`eval` - evaluates a form producing a value
`apply` - evaluates a function definition of one argument
`funcall` - evaluates a function definition of n arguments

• Functions that treat programs as data (for Franz Lisp)

`putd` - associates a function with a symbol
`getd` - retrieves function definition of a symbol
`def, defun`

Of these, only `defun` is available in Common Lisp

3

Examples of Using Programs as Data (Franz Lisp)

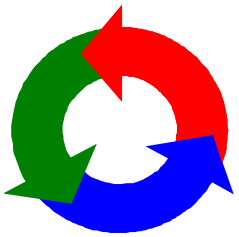
```
--> (def foo  ( lambda (x) (cadr x )))
   foo
--> (foo '(a b c))
   b
--> (getd 'foo)
   (lambda (x) (cadr x))
--> (cons (car (getd 'foo)) '((u y z ) y))
   (lambda (u y z) y)
--> (putd 'bar (cons (car (getd 'foo)) '((u
y z) y)))
   (lambda (u y z) y)
--> (bar 4 (+ 2 3) '(t q))
   5
--> (defun n-args (x)
        (length (cadr (getd x))))
   n-args
-->(n-args 'foo)
   1
-->(n-args 'bar)
   3
-->(n-args 'n-args)
   1
```


4

In Common Lisp, Use `function` Instead of `getf`

```
-->(defun foo (x) (list x x))
FOO

-->(function foo)
(LAMBDA-BLOCK FOO (X) (LIST X X))
```



5

eval

`eval` is a function that accepts an s-expr as argument and evaluates it returning the result.

For example:

```
--> '(cons 'a '(b c))]
   (cons 'a '(b c))
-->   (eval '(cons 'a '(b c)))
   (a b c)
--> 'x
   x
--> (setq x  ' ( list '+ 3 4))
   (list '+ 3 4)
--> x
   (list '+ 3 4)
--> (eval 'x)
   (list '+ 3 4)
--> (eval x)
   (+ 3 4)
--> (eval (eval  'x))
   (+ 3 4)
--> (eval (eval x))
   7
--> (eval '(eval x))
   (+ 3  4)
```


©1993 John Mylopoulos

6


```
--> (setq y 'x)
   x
--> (eval 'y)
   x
--> (eval '(quote y))
   y
--> (eval y)
   (list '+ 3 4)
--> (eval (eval y))
   (+ 3  4)
```


©1993 John Mylopoulos

7

Using `eval`: Extending Lisp

Suppose we want to define an if-then-else construct (`myIf`) such that

```
(myif e1 e2 e3)
```

behaves like

```
(cond (e1 e2) (t e3))
```

The obvious solution,

```
(defun myIf (e1 e2 e3)
   (cond   (e1 e2) (t  e3)))
```

won't work! We don't want the arguments of `myIf` to be evaluated before the body of the function:

```
(myIf (atom x) x (car x))
```

evaluates `(atom x)`, `x`, `(car x)` before applying `myIf`

We need to be able to control the evaluation of function arguments

8

`apply`

`apply` takes a function name and a list of arguments and applies the function to the arguments. Thus,

```
(f  s1 ... sn)
```

is equivalent to

```
(apply 'f '(a1 .. an))
```

, where `a1`, ..., `an` are the values of `s1`,..., `sn` respectively

For example:

```
--> (apply 'cons '(a (b c)))
   (a b c)

--> (apply ' cons  '(x  (c d e)))
   (x c d e)
```

9

funcall

`funcall` takes a function name and a sequence of arguments and applies the function to the arguments.

Thus,

```
(f  s1 ... sn)
```

is equivalent to

```
(funcall 'f  'a1 ... 'an)
```

, where `a1`, ..., `an` are the values of `s1`,..., `sn` respectively.

For example,

```
--> (funcall  'cons  ' a  '(b c) )
   (a b c)

--> (funcall  'cons  'x  '(c d e)))
   (x c d e)
```


10

Kinds of Function Definitions

- `(lambda ( ... ) ( ... ))`
  - evaluate arguments, bind, evaluate form
- `(macro ( ... ) ( ... ) )`
  - do not evaluate arguments, bind, eval form, return value

11

Using macros to define `if`

```
(defmacro macro-name lambda-list body )

(defmacro ourIf
          (pred then &optional else)
   (list 'cond (list pred then)
      (list 't else)))

-> (ourIf (> 5 0)     (print 'greater)
                      (print 'lesser) )

   ;; generates
   ;; (cond ((> 5 0)     (print 'greater))
   ;;       (t           (print 'lesser)))
   ;; and then evaluates it

greater
greater
```


©1993 John Mylopoulos

12

Backquote

Unlike `quote`, `backquote` makes it possible to evaluate parts of a (back)quoted expression. A comma indicates which parts of a backquoted expression are to be evaluated:

```
<cl> (setf book '(war and peace)
(WAR AND PEACE)

<cl> '(this is a book)
(THIS IS A BOOK)

<cl> '(this is a ,book)
Error

<cl> `(this is a book)  ;; backquote used
(THIS IS A BOOK)

<cl> `(this is a ,book)
(THIS IS A (WAR AND PEACE))
```


©1993 John Mylopoulos

13

Backquote

A comma followed by @(`,@`) substitutes itself with its list value:

```
<cl> (setf book '(war and peace)
(WAR AND PEACE)

<cl> '(this is a book)
(THIS IS A BOOK)

<cl> `(this is a ,book)
(THIS IS A (WAR AND PEACE))

<cl> `(this is a ,@book)
(THIS IS A WAR AND PEACE)

<cl> `(' ,@book)
(' WAR AND PEACE)
```


14

Using `defmacro` to define `quote`, `list`, `or`

```
-> (defmacro kwote (args) `',args)
   kwote
->(kwote (car x))
   (car x)
->(kwote x)
   x

->(defmacro myList (&rest args)
    `(cond ((null ',args) nil)
           (t (cons ,(car args)
              (myList ,@(cdr args))
 ))))

->(defmacro myOr (arg1 &rest args)
    `(cond (,arg1  ,arg1)
           ((null ',args) nil)
           (t (cons ,(car args)
              (myOr ,(car args)
                    ,@(cdr args))
 ))))
```

15

How Does `myList` Work?

Consider `(myList 'a 'b 'c)` which expands to

```
(COND ((NULL '('A 'B 'C)) NIL)
      (T (CONS 'A (MYLIST 'B 'C))))
```

This works when eval'd but one can *not* compile a function containing this macro because it causes the compiler to go into an infinite recursion.

A much better way of doing it is:

```
(defun betterHelper (x)
  (cond ((null x) nil)
      (t `(cons ,(car x)
           ,(betterHelper (cdr x))))))

(defmacro betterList (&rest args)
  (betterHelper args))
```

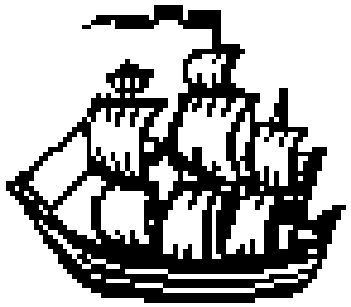
Then `(betterList 'a 'b 'c)` expands to

```
(CONS 'A (CONS 'B (CONS 'C NIL)))
```

which is what you really want. You need the helper function to handle the recursion.

16

...But, There Are Easier Ways...

```
(defun myList1  (&rest  args)   args)
```

this will also define a `list` special form function, without any need for macros, backquotes