Form & Function in Software

Richard P. Gabriel phd mfa
Confusionists and superficial intellectuals...
...move ahead...
...while the ‘deep thinkers’ descend into the darker regions of the status quo...
...or, to express it in a different way, they remain stuck in the mud.

-Paul Feyerabend
(defun factorial (n)
  (cond ((= n 0) 1)
        (t (* n (factorial (- n 1)))))))
(defun eval (form env)
  (cond
   ((null form) nil)
   ((numberp form) form)
   ((stringp form) form)
   ((eq t form) form)
   ((atom form)
    (cond
     ((get form 'APVAL))
     (t (lookup form env))))
   ((eq (car form) 'quote) (car (cdr form)))
   ((eq (car form) 'cond) (evalcond (cdr form) env))
   (t (apply (car form) (evallist (cdr form) env) env))))

(defun apply (fct parms env)
  (cond
   ((atom fct)
    (cond
     (eq fct 'car) (car (car parms)))
     ((eq fct 'cdr) (cdr (car parms)))
     ((eq fct 'cons) (cons (car parms) (car (cdr parms))))
     ((eq fct 'get) (get (car parms) (car (cdr parms))))
     ((eq fct 'atom) (atom (car parms)))
     ((eq fct 'error) (error (string parms)))
     ((eq fct 'eq) (eq (car parms) (car (cdr parms))))
     (t (cond
         ((get fct 'EXPR)
          (apply (get fct 'EXPR) parms env) parms env)
         (t (apply (lookup fct env) parms env)))))
    (eq (car fct) 'lambda)
    (eval (car (cdr fct)))
    (update (car (cdr fct)) parms env))
   (t (apply (eval fct env) parms env))))

(defun evalcond (conds env)
  (cond
   ((null conds) nil)
   ((eval (car (car conds)) env)
    (eval (car (cdr (car conds))) env))
   (t (evalcond (cdr conds) env))))
(defun eval (form env)
  (cond ((null form) nil) ((numberp form) form) ((stringp form) form) ((eq t form) form) ((atom form)
    (cond ((get form 'APVAL)) (t (lookup form env)))
    ((eq (car form) 'quote) (car (cdr form)))
    ((eq (car form) 'cond)
      (evalcond (cdr form) env)) (t (apply (car form) (evallist (cdr form) env) env))))
(defun apply (fct parms env)
  (cond ((atom fct) (cond ((eq fct 'car) (car (car parms))) ((eq fct 'cdr) (cdr (car parms)))
      ((eq fct 'cons) (cons (car parms) (car (cdr parms))))
      ((eq fct 'get) (get (car parms) (car (cdr parms))))
      ((eq fct 'atom) (atom (car parms)))
      ((eq fct 'error) (error (string parms)))
      ((eq fct 'eq) (eq (car parms) (car (cdr parms))))
      (t (cond ((get fct 'EXPR) (apply (get fct 'EXPR) parms env) parms env)
        (t (apply (lookup fct env) parms env)))
      ((eq (car fct) 'lambda) (eval (car (cdr (cdr fct)))
        (update (car (cdr fct)) parms env))
      (t (apply (eval fct env) parms env))))
  (defun evalcond (conds env)
    (cond ((null conds) nil) ((eval (car (car conds)) env) (eval (car (cdr (car conds))) env))
      (t (evalcond (cdr conds) env)))
  (defun evalcond (conds env)
    (cond ((null conds) nil) ((eval (car (car conds)) env) (eval (car (cdr (car conds))) env))
      (t (evalcond (cdr conds) env))))
form and function can be as disjoint as you care to have it
(factorial 10) -> 3628800
(defun eval (form env)
  (cond ((eq form 't) t)
        ((eq form 1) (format t "!~%") nil)
        ((atom form) (lookup form env))
        ((eq (car form) '*)) (format t " Screw you!")
        (eval (caddr form) env))
        ((eq (car form) '=') (= 0 (lookup (caddr form) env)))
        ((eq (car form) '+) (+ (lookup (cadadr form) env) 1))
        ((eq (car form) '-' (lookup (cadadr form) env) 1))
        ((eq (car form) 'cond) (evcond (cdr form) env))
        (t
         (apply (car form) (evlist (cdr form) env) env))))

(defun apply (fn args env)
  (let ((fndef (lookup fn env)))
    (eval (cadr fndef) (update (car fndef) args env))))

(defun evcond (forms env)
  (cond ((null forms) nil)
        ((eval (car (car forms)) env)
         (eval (cadr (car forms)) env))
        (t (evcond (cdr forms) env))))

(defun update (l1 l2 env)
  (cond ((null l1) env)
        (t (update (cdr l1) (cdr l2) (push (list (car l1) (car l2)) env))))

(defun lookup (var env)
  (cadr (assoc var env)))

(defun evlist (l env)
  (mapcar #'(lambda (x) (eval x env)) l))
(factorial 10) ->

Screw you! Screw you! Screw you! Screw you! Screw you!
Screw you! Screw you! Screw you! Screw you! Screw you!!
NIL
the same form can have many functions...
...& the same function can be expressed in many forms
<table>
<thead>
<tr>
<th>form1</th>
<th>form2</th>
</tr>
</thead>
<tbody>
<tr>
<td>form3</td>
<td>form4</td>
</tr>
</tbody>
</table>

Function
but, an interpreter is another form...
almost any number of interpreters can produce the same result
∀form, ∀function, ∃interpreter st
form -[interpreter]--> function
in the real world...
form <-[laws of physics?]--> function
a door must be large enough...
...for what passes through...
...& a table must be flat...
...so what it supports does not slip
such laws are the essential interpreter...
...everything else is contingent
and perhaps in the real world...
form <-[design]- function

constrained
form <- [design] - function

constrained
in the software world...
function
^  
|  
|  
design
form
^  
|  
|  design

function
^  
|  
|  design
are all software interpreters contingent?
Physical Constraints on Computing

- P=NP?
- size and speed of memory
- speed of processors
- speed of communications
- density of computational resources
limited resources
unlimited imagination
...but it’s rarely this desperate...
# Other Forms of Form

<table>
<thead>
<tr>
<th>Model</th>
<th>Purpose</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>procedural</td>
<td>control</td>
<td>Pascal, Algol</td>
</tr>
<tr>
<td>functional</td>
<td>composition</td>
<td>Lisp, Haskell</td>
</tr>
<tr>
<td>logic</td>
<td>constraints</td>
<td>Prolog</td>
</tr>
<tr>
<td>object-oriented</td>
<td>simulation</td>
<td>Smalltalk, Java</td>
</tr>
<tr>
<td>hardware</td>
<td>OS</td>
<td>C, C++</td>
</tr>
<tr>
<td>string</td>
<td>transformation</td>
<td>Perl</td>
</tr>
<tr>
<td>array</td>
<td>collections</td>
<td>APL</td>
</tr>
<tr>
<td>concurrency</td>
<td>events</td>
<td>threading?</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
invent an intellectual structure...
...describing a programming model...
...that makes it easier to program things that we think of that way
00: objects sending messages to each other
Sequence Diagram: example

Caller

- lift-receiver
- begin(dialtone)
- dial(7)
- end(dialtone)
- dial(3)
- dial(5)
- dial(3)
- begin(ringing)

Phone line

- begin(ringing)
- connect
- disconnect

Receiving Party

- lift-receiver
- connect
- replace-receiver
- disconnect
Collaboration Diagram: example

1: lift-receiver
3: dial(7)
5: dial(3)
6: dial(5)
7: dial(3)

2: begin(dialtone)
8: begin(ringing)
4: end(dialtone)
10: connect
12: disconnect
9: lift-receiver
11: replace-receiver
10: connect
12: disconnect

Caller

Phone line

Receiving Party
UML State Diagram - example

Chess game

Start

White’s turn

- checkmate
- stalemate

Black’s turn

- black moves
- white moves

Black wins

Draw

White wins

Topic 10

OOAD

13
Guards, Activities and Actions - example

Vending machine model

- **Idle**
  - **Coins in (amount) / set balance**
  - **Cancel / refund coins**

- **Collecting money**
  - **Coins in (amount) / add to balance**
  - **Select (item)**
  - **[change < 0]**
  - **[change = 0]**
  - **[change > 0]**

- **Do: test item and compute change**
  - **[change = 0]**
  - **[change > 0]**

- **Do: dispense item**

- **Do: make change**
State Generalization: example

Transmission

Neutral

Reverse

push N
push R

push N
push F

Forward

First

Second

Third

upshift
upshift
downshift
downshift
other forms of form
Sony Personal Entertainment Communicator
~50 computers
Boeing 777 Flight Deck
many excellent programs...
...exhibit common local characteristics...
...not the same, but similar...
...and they represent sketches of form...
...giving rise to excellent function, sturdy structure, and palpable beauty
they are called “patterns,”...
...and they are our best hope for a lasting connection between form and function in software
form creates function for the essential interpreter
form creates aesthetics for the contingent interpreter
software is the discipline where form and function are least entangled
last thought:
(factorial 10)