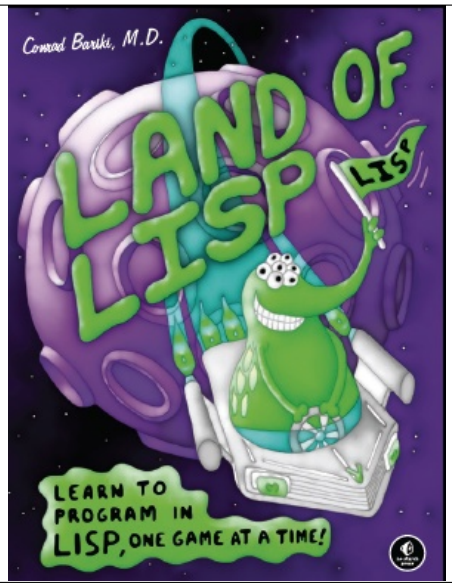
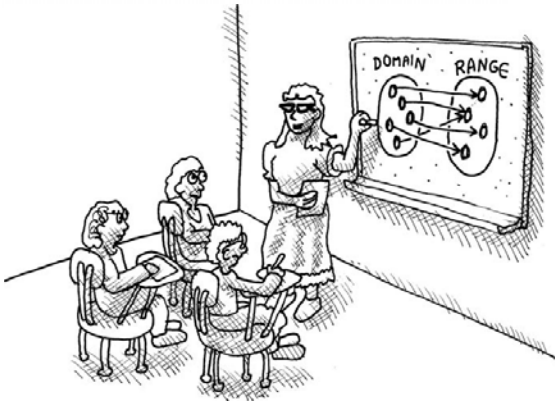


Ch. 14

Ch. 14 - Ramping Lisp Up A Notch With Functional Programming



Functional Programming

- What is functional programming?
- Anatomy of a program written in the Functional Style
- Higher-Order programming
- Why functional programming is crazy
- Why functional programming is fanastic
- What you've learned

Lambda

- Lambda is a special form (like a Macro) - it doesn't evaluate its parameters first
- However, the actual value that lambda returns is a regular Lisp function!
- Functions with lambda instead of a name are called anonymous functions

```
>(mapcar
     (lambda (n)
          (/ n 2))
          '(2 4 6))
(1 2 3)
```


Why Lambda is So Important

- The ability to pass around functions as if they were just plain old pieces of data is incredibly valuable.
- This opens up all kinds of conceptual possibilities in the design of your programs!
- The name for the *style of programming* that relies heavily on passing functions as values is called ***higher-order functional programming.***

Properties of Mathematical Functions

The function

1. always returns the same result, as long as the same arguments are passed into it. (This is often referred to as *referential transparency*.)
2. never references non-local variables (those defined outside the function), unless we are certain that these variables will remain constant.
3. No variables are modified (or *mutated*, as functional programmers like to say) by the function.
4. The purpose of the function is to do nothing other than to return a result.
5. doesn't do anything that is visible to the outside world, such as pop up a dialog box on the screen or make your computer go "Bing!"
6. doesn't take information from an outside source, such as the keyboard or the hard drive.

Lisp Functions Can Obey The Same Rules

- Example, the sine function

```
> (sin 0.5)            ; sine function in Lisp
0.47942555
```

- [1] The `sin` function always returns the same result, as long as you always pass the same argument (in this case, 0.5) into it.
- [5,6] It doesn't do anything to interact with the outside world. ([2,3] or alter variable values)
- [4] Its entire purpose in life is to return the sine as a value.

Therefore, `sin` obeys all the rules in the preceding list.

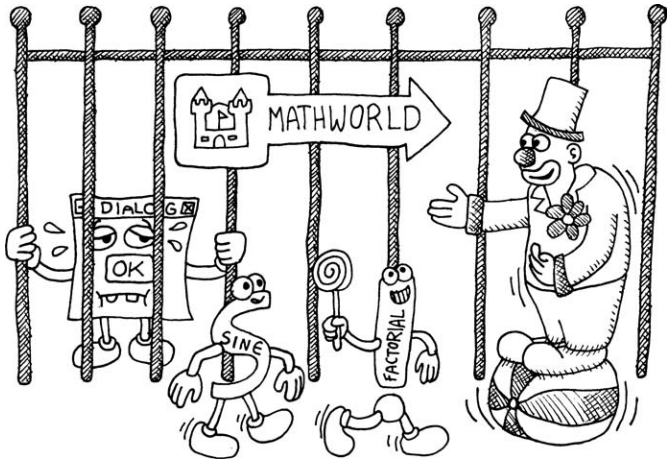
- "Clearly, it would be impossible to write *all* the code in a computer program in the functional style.
- For instance, one of the rules stipulates that the computer isn't allowed to go "Bing!"— *who would want to use a computer if it didn't go "Bing!" once in a while?*"

Central Philosophy of Functional Programming

You should break your program into two parts:

1. The first, and biggest part, should be completely functional and free of side effects. This is the clean part of your program.
2. The second, smaller part of your program is the part that has all the side effects, interacting with the user and the rest of the outside world. This imperative code is dirty and should be kept as small as possible.

Separating Clean and Dirty Code

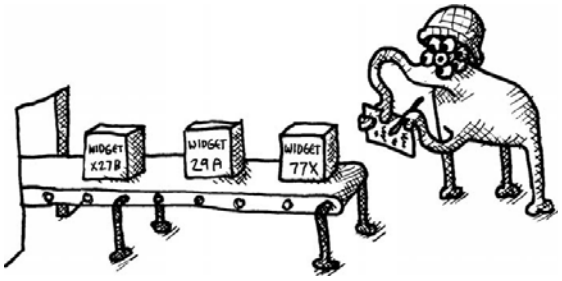


Anatomy of a Program Written in the Functional Style

What do most programs in the world actually do?

Keep track of widgets!

Lets write a program to do that.



Here's our entire example program, written in the functional style:

```
;;; the clean, functional part
(defun add-widget (database widget)
      "Add a widget (2nd arg)  to a database (1st arg)."
      (cons widget database))

;;; the dirty, nonfunctional part
(defparameter *database* nil "create an empty database")

(defun main-loop ()
   "loop to add widgets to *database* with a prompt"
   (loop                  ; infinite loop, ^C to exit
      (princ "Please enter the name of a new widget:" )
      (setf *database* (add-widget *database* (read)))
      (format t "The database contains the following:
            ~a~%" *database*)
 ))
```

Isn't This Horribly Inefficient?

- It seems that we are *copying the entire database* each time we add a new widget!
- Actually, Lisp creates a new cons cell for the widget only!
- The rest of the database stays the same and the new widget points to the previous database.

FP Efficiency

- Sharing of structures can be done safely, since one of the tenets of functional programming is to never modify old pieces of data.
- Lets test our code:

```
> (main-loop)
Please enter the name of a new widget: Frombulator
The database contains the following: (FROMBULATOR)
Please enter the name of a new widget: Double-Zingomat
The database contains the following: (DOUBLE-ZINGOMAT
FROMBULATOR)
...
```

;;; Remember that you can hit CTRL-C to exit the

;;; infinite loop in this example

Higher-Order Programming

- Code composition - combining different chunks of code to perform a single action.
- This can be a stumbling block for new functional programmers to master.
- It is powerful because functions can take functions as parameters.
- Nesting functions is easy. Why make a new variable to hold a value when it will only be used right away in another function?

Code Composition With Imperative Code

- Suppose we want to add two to every number in the following list:

```
> (defparameter *my-list* '(4 7 2 3))
*MY-LIST*
```

- We will need to
  1. write code to traverse the list, and
  2. write code to add two to a number
- The following code is for demonstration purposes only. A Lisper would not write code like this!!!!

```
> (loop for n below (length *my-list*) ; loop through list
      do (setf (nth n *my-list*)      ; set each value to
               (+ (nth n *my-list*) 2)))  ; value + 2
NIL
> *my-list*
 (6 9 4 5)                        ; mission accomplished!!
```


Iterative Style

- Positive
  - Code can be very efficient
  - Clearly composes the iteration and change in value
- Negative
  - The original list is destroyed – what if the list was needed elsewhere.
  - We had to create an extra variable n, to keep track of our position in the list.

Functional Versions

```
;; First version - create a new function to do the job
> (defun add-two (lst)
      (when list           ; repeat until the arg is empty
            (cons (+ 2 (car lst))
                  (add-two (cdr lst)))))
ADD-TWO
> (add-two '(4 7 2 3))
(6 9 4 5)
```

```
;; Better Version – use built-in Lisp function mapcar to iterate
> (mapcar (lambda (x)    ; and anonymous lambda function
            (+ x 2))          ; to add 2 to each number
          '(4 7 2 3))
(6 9 4 5)
```

Why Functional Programming is Crazy

- We can separate clean functional code from dirty code with side-effects
- Performance can be a concern
  - Memory allocation
  - Stack growth due to recursion
- Improve performance by
  - Reduces memory requirements by using shared structures between different pieces of data in our programs
  - Reduce stack space, memory needs, and/or redundant computation using memoization, tail call optimization, lazy evaluation, and higher-order programming

Why Functional Programming is Fantastic

- Reduces bugs
- Programs are more compact
- Functional programs are more elegant

Summary

- Programs written in the functional style always give the same result when they are given the same values in their arguments.
- Functional programs do not contain side effects. Their whole purpose in life is to only calculate a value to return.
- Programs that are not functional usually read like a cookbook, with statements like, “First do this, and then do that.” This style is called imperative programming.
- A good strategy for writing Lisp programs is to break them into a clean, functional part and a dirty, imperative part.
- Functional programs can be written quickly, are more compact, and tend to have fewer bugs.