Functional Programming in Lisp

Introduction

Functional Programming Features

- Persistence: variables and data structures are immutable (constant)
- Recursion: construct algorithms as recursive functions (vs. loops)
- First-class functions: can be passed to and returned from other functions

Trade-offs of Functional Programming

- Pros
  - Easy (comparatively) to reason about (prove) correctness
  - Compact (fewer LoC)
  - Immutable structures shared between module, threads, etc
- Cons
  - Different way of thinking about programming
  - Sometimes less (constant-factor) efficient

Outline

- Recursion
- First-Class Functions
- Higher-order Functions

Recursion

Recursion: a function or other object defined in terms of itself

- Base case: terminating condition
- Recursion case: reduction towards base

Lisp: DESTRUCTURING-BIND

Destructuring-bind: bind variables to corresponding values draw from a list

```
(destructuring-bind (a b c)
    '(1 2 3)
    (list c b a)
)

;; OUTPUT: (3 2 1)
```

First-Class Functions

First-Class Functions: a programming language has **first-class functions** when it treats functions like any other variable or object. First-class functions can be:

- Bind variables to the function
- Passed as arguments to other functions
- Returned as the result of other functions

Function Closures

Function closure: a function closure or lexical closure is a function and an associated set of variable definitions.

- Ex. Java has classes, C has structs

Closures in Lisp: Local Functions

Labels: defines local functions and executes body using those local functions

```
(labels ((FUNCTION-NAME VARIABLES FUNCTION-BODY)...) LABEL-BODY)
```

Example

```
(let ((a 1))
    (labels ((adder (x)
        (+ x a)))
        (adder 2))
)
```

Closure in Lisp: Anonymous Functions

Lambda: defines an anonymous function:

```
(lambda VARIABLES FUNCTION-BODY)
```

Funcall: Apply a function to the provided arguments

```
(funcall FUNCTION ARGUMENTS...)
```

Example

```
(let ((a 1))
    (funcall (lambda (x)
        (+ x a))
    2)
)
```

Value and Function Namespaces

Value Namespace: record values

- Local: let, let*
- Global: defparamter

Function Namespace: record function definitions

- Local: labels, flet
- Global: defun

Function and Funcall

Function: returns the functional value of a name

```
(function NAME)
;; the function bound to name
```

Funcall: apply a function to the provided arguments

```
(funcall FUNCTION ARGS...)
;; return value of FUNCTION called on ARGS
```

Higher-Order Functions

Higher-Order Functions: a function that takes another function as an argument or returns another function as its result

Common Higher-Order Functions

map: transform elements of a collection fold: combine elements of a collection

Map Function

Map: apply a function to every member of an input sequence, and collect the results into the output sequence.

```
(map 'list
    (lambda (x) (+ 1 x))
    (list 1 2 3)
)
;; Result = (2 3 4)
```

Fold-left Function

Fold-left: apply a binary function to each member of a sequence and the prior result, starting from the left

```
(reduce #'+
    '(1 2 3)
    :initial-value 0)
;; (+ (+ (+ 0 1) 2) 3) = 6
```

Fold-right Function

Fold-right: apply a binary function to each member of a sequence and the prior result, starting from the right

```
(reduce #'+
    '(1 2 3)
    :initial-value 0
    :from-end t)
;; (+ 1 (+ 2 (+ 3 0)))
```