Common Lisp and Introduction to Functional Programming
Lecture 7: Higher-order Functions

Yuri Zhykin

Mar 25, 2021

So What Is Functional Programming? 1/3

- **Procedural programming** is an **imperative** programming paradigm in which "procedure" definitions a sequence of imperative statements which update the **state** of the program.
- **Functional programming** is a **declarative** programming paradigm in which "function" definitions are **trees of expressions** that map values to other values.
- **Functional programming** is a programming paradigm where programs are constructed by **applying** and **composing** functions.

So What Is Functional Programming? 2/3

- Functional programming approach builds software by composing **pure functions**, avoiding **shared state**, **mutable data**, and **side-effects**.
- Program state is not modified explicitly but rather "flows" through function calls.
- Functional program can be a single expression consisting of a large number of function invocations.
- State is passed as arguments to functions-components of the program expression and returned as results, which become arguments to other functions-components.

So What Is Functional Programming? 3/3

- Example of a functional "data pipeline":

```lisp
(defun collect-data (sources)
  "Returns raw data."
  ...)

(defun process-data (data)
  "Processes raw data and returns results."
  ...)

(defun analyse-results (results)
  "Analyses results and returns analysis data."
  ...)

(defun show-analysis (analysis)
  "Prints analysis data to standard output."
  ...)

(defun main ()
  (let ((sources (list <source1> <source2> ...)))
    (show-analysis
     (analyse-results (process-data (collect-data sources))))))
```

First-class Functions

- A programming language is said to have **first-class functions** if it treats functions as **first-class values**.
- **First-class values** or language's **first-class citizens** are values that can be
  - passed as arguments to functions,
  - returned as results from functions,
  - assigned to variables,
  - stored in data structures.
- **First-class functions enable passing behaviour as arguments to functions.**

Higher-order Functions 1/2

- **Higher-order function** (a.k.a., **Operator** or **Functional** in mathematics) is a function that takes one or more functions as arguments or returns a function as its result.
- Functions that neither take functions as parameters nor return functions as results, are called **first-order functions**.
- Higher-order functions are necessary in order to have first-class functions.

Higher-order Functions 2/2

- **Differential operator** is a common example of higher-order functions: it takes function as an argument and maps it to its derivative, which is also a function

$$\frac{d}{dx} f(x) = f'(x)$$

- Common Lisp's `funcall` and `apply` are also higher-order functions, since they take functions as their first argument:

```
CL-USER> (funcall (lambda (x) (* x 2)) 5)
10
CL-USER> (apply #'+ '(1 2 3 4 5))
15
```

Functional Composition 1/2

- Main role of functions, similar to procedures in procedural programming, is to group computations into units that have a clearly defined common goal and dependencies - arguments.
- In order to construct programs, one must be able to **apply** and **compose** functions.
- Common Lisp provides operators for applying functions (`funcall`, `apply`), while composing is trivially done via nested expressions:

```
(f1 (f2 a1 a2) (f3 (f4 a3)))
```

Functional Composition 2/2

- **Function composition operator** is a binary operator that takes two functions f and g and returns a function h such that $h(x) = f(g(x))$:

```
(defun compose (&rest fs)
  (if (null fs)
      (lambda (x) x)
      (lambda (x)
        (funcall (car fs) (funcall (apply #'compose (cdr fs)) x)))))
```

- Function composition operator enables very elegant style of programming called **tacit style** or **point-free style**, in which function definitions do not identify their arguments:

```
(defun main ()
  (apply (compose #'show-analysis
                  #'analyse-results
                  #'process-data
                  #'collect-data)
         (list <source1> <source2> ...)))
```

Map, Reduce and Filter

- **Recursive operators** `Map`, `Filter` and `Reduce` are higher-order functions that take an recursively defined object and a function, and apply that function along the structure of the object.
- Common Lisp provides standard functions `mapcar`, `remove-if-not` and `reduce` that operate on lists.
- Similar functions that operate on sequences are available in most modern programming languages (Java Script, Python, Java, etc).
- Just using these three operators may be enough for most sequence processing tasks.

Map, Filter and Reduce: Map

- Function `Map` takes a function $f : A \mapsto B$ and a list A and **returns a new list** B such that

$$\forall a_i \in A, b_i \in B : b_i = f(a_i)$$

- Example:

```
CL-USER> (mapcar (lambda (x) (* x x)) '(1 2 3 4 5))
(1 4 9 16 25)
```

Map, Filter and Reduce: Filter

- Function `filter` takes a function $f : A \mapsto \{True, False\}$ and a List A and **returns a new list** B such that

$$\forall a_i \in A : a_i \in B \iff f(a_i) = True$$

- Example:

```
CL-USER> (remove-if-not #'evenp '(1 2 3 4 5))
(2 4)
```

Map, Filter and Reduce: Reduce

- Function `reduce` takes a function $f : B \times A \mapsto B$, a list $A = a_1, a_2, ..., a_k$ and an initial value b_0 and **returns a value** b_k where

$$b_i = f(b_{i-1}, a_i), i = 1, 2, ..., k$$

- Example:

```
CL-USER> (reduce #'+ '(1 2 3 4 5) :initial-value 0)
15
CL-USER> (+ (+ (+ (+ (+ 0 1) 2) 3) 4) 5)
15
```

Combining Map, Filter and Reduce 1/2

- Typical data pipeline with `Map`, `Filter` and `Reduce` looks like this:

```
;; (aggregate (transform (select (collect ...))))
(reduce (lambda (x y) ...)
        (mapcar (lambda (x) ...)
                (remove-if-not (lambda (x) ...)
                               (collect ...))))
```

- This general pattern fits large number of use cases and sequence processing problems and can be applied almost identically in most of modern programming languages.

Combining Map, Filter and Reduce 2/2

- Example:

```
(flet (;; Return a number of exported functions for a package.
       (count-exported-functions (p)
         (let ((fcount 0))
           (do-external-symbols (s p)
             (when (fboundp s) (incf fcount)))
           fcount))

       ;; Check if a package is an SBCL-supplied one.
       (sbcl-package-p (p)
         (let ((index (search "SB-" (package-name p))))
           (and index (zerop index)))))

  ;; Collect, filter, process and aggregate data.
  (reduce
   #'+
   (mapcar #'count-exported-functions
           (remove-if-not #'sbcl-package-p (list-all-packages)))
   :initial-value 0))
```

The End

Thank you!