Common Lisp and Introduction to Functional Programming
Lecture 6: Functional Programming Basics

Yuri Zhykin

Mar 17, 2021

Functions in Mathematics 1/2

- In mathematics, a **function** is a *binary relation* between *two sets* that associates to each element of the first set *exactly one* element of the second set.
- *Intentionally*, functions can be defined as combinations of *primitive operations* and *previously defined functions*:

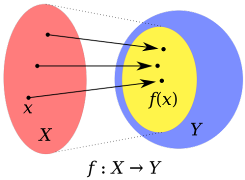
$$f(x) = 10^x$$

- *Extensionally*, functions can be defined by listing function values that correspond to argument values:

$$f(x) = \begin{cases} 10, & x = 1, \\ 100, & x = 2, \\ 1000, & x = 3, \\ \dots & \dots \end{cases}$$

Functions in Mathematics 2/2

- Mathematical functions are often called **maps** or**mappings** between sets.
- Mathematics distinguish the following components of a function definition: **domain**, **codomain**, **image** and **graph**.



- The output of a mathematical function depends only on the input, so one only needs the input value and the function definition itself denoted in some way in order to establish the corresponding output.

Functions in Programming

- Commonly used term **function** in modern programming languages actually means **procedure**.
- **Procedures** are also known in some programming languages as **routines** or **subroutines**.
- **Procedural programming** is a programming paradigm based on the concept of the **procedure call**.
- **Procedure** is a group of instructions that do a clearly defined task and can be "called" multiple times - a very basic means of achieving **modularity** and **code reuse**.
- **Procedure** only vaguely resembles a mathematical function.

State

- **What is the difference between a procedure and a mathematical function?**
- **State** of a program can be defined as the values of all variables (i.e. contents of all storage locations in memory) at any given point in time.
- By definition, the result of mathematical function depends solely on its arguments.
- Procedures (or "functions") in programming languages always depend on state of the program at the moment of procedure (or "function") call.
- In order to determine the result of the procedure call, we need to look at the whole program state, not just the procedure definition and arguments.

Side effects

- **Let's make matters even more complicated!**
- Any useful program **modifies** the global state in order to produce useful results.
- Procedure's result can depend on parts of the state that is modified by any other procedures, including itself, so even sequential calls of the same procedure may produce different results.
- Modifications of variables or structures after they were defined are called **mutations**.

Practical Function Programming

- **What if we wrote procedures in a way that does not depend on state?**
- We call all "functions" **procedures**.
- We agree to write **some** procedures in a way that does not **explicitly** depend on global state, and we call such procedures **functions** or **pure functions**.
- We call any modifications of the state **outside the scope** of a procedure **side effects**.
- We can treat procedures free of side effects as "black boxes" - as soon as we define and verify it, we no longer care about the internals.
- Our program is split into two parts - one that is free of side-effects, and the other that is responsible for all side effects.

Referential Transparency

- An expression is called **referentially transparent** if it can be replaced with its corresponding value (and vice-versa) without changing the program's behavior.

```lisp
(defun the-answer ()
  (print "The answer is: 42.")
  42)

;; (+ 1 (the-answer))
;; (+ 1 42)
```

- Expressions that consist of **pure function** calls are referentially transparent.

Recursion 1/2

- N-th Fibonacci number function:

$$Fib(n) = \begin{cases} 0, & n = 0, \\ 1, & n = 1, \\ Fib(n-1) + Fib(n-2), & \text{otherwise} \end{cases}$$

- Imperative approach using loop:

```
(defun fib (n)
  (let ((a 0)
        (b 1))
    (loop :for i :below n :do
      (multiple-value-setq (a b) (values b (+ a b))))
    a))
```

- Functional approach that follows mathematical definition:

```
(defun fib (n)
  (if (or (= n 0) (= n 1))
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

Recursion 2/2

- Using **recursion** as a general replacement for loops might be inefficient when following mathematical definitions, but most recursive functions can be written in a **tail-recursive** form.
- Function call requires more instructions than a loop iteration.
- **Tail recursion** refers to the recursive call being the last logic instruction in the recursive algorithm.
- Functional approach optimized with tail recursion:

```
(defun fib (n)
  (labels ((%fib (%n a b)
             (if (= %n n)
                 a
                 (%fib (+ %n 1) b (+ a b)))))
    (%fib 0 0 1)))
```

- Some languages (e.g. Lisp) implement **TCO** (**tail call optimization**) that replaces a tail function call with a jump.

The End

Thank you!