Pontifícia Universidade Católica do Rio de Janeiro

Luiz Romário Santana Rios

A survey of function values in imperative programming languages

Dissertação de Mestrado

Dissertation presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Roberto Ierusalimschy

Rio de Janeiro
April 2019

Pontifícia Universidade Católica
do Rio de Janeiro

Luiz Romário Santana Rios

A survey of function values in imperative programming languages

Dissertation presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the undersigned Examination Committee.

Rio de Janeiro, April 25th, 2019

Luiz Romário Santana Rios

Bibliographic data

Santana Rios, Luiz Romário

A survey of function values in imperative programming languages / Luiz Romário Santana Rios; advisor: Roberto Ierusalimschy. – Rio de janeiro: PUC-Rio, Departamento de Informática, 2019.

v., 79 f: il. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia

1. Ciência da Computação – Teses. 2. Linguagens de Programação – Teses. 3. Funções de primeira classe;. 4. Programação funcional;. 5. Programação imperativa;. 6. Programação orientada a objetos;. 7. Design de linguagens de programação;. 8. Levantamento.. I. Ierusalimschy, Roberto. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

To all the people where I find a home.

Acknowledgments

I am very thankful to Roberto for accepting me as a student, for all his advice, for helping me take a step back and re-evaluate the direction I'm going several times, and, last but not least, for suggesting the theme of this dissertation. I learned a lot in this journey and it helped me become a better researcher.

I am also very thankful for all the support I got from the folks at LabLua, especially—but not limited to—Hugo and Ana. Not only they helped me academically, but they also motivated me to keep going forward. I don't think I would be able to finish this work without their help.

Finally, I owe a lot to my loved ones that, despite being far away, always believed in me and were there for me. In particular, thanks to my father, for coming along with me to a city I had never been before; thanks to my siblings, for always giving me life advice and being super excited about my endeavors far from home, and thanks to Lauriza, my partner, for closely following along and giving me the love I needed.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Abstract

Santana Rios, Luiz Romário; Ierusalimschy, Roberto (Advisor). **A survey of function values in imperative programming languages**. Rio de Janeiro, 2019. 79p. Dissertação de mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A programming language is said to have first-class functions when it provides the capability of manipulating functions in the same way as other values, i.e., storing in variables, passing as parameters, etc.. Programming with first-class functions opens the programmer to new forms of abstractions and it's the default in functional programming languages. However, in the realm of imperative languages (including object-oriented languages), each language has different semantics, properties, and terminology for functions—in great part, thanks to their focus on mutability, which isn't present in functional languages.
To help shed light on these differences, we made a survey of the specification of function values in imperative programming languages from many different disciplines. For each language, we illustrate, based on examples, the properties of function values in it, highlighting where it differs from other languages—all this with a consistent terminology in all languages. We provide a reference that compares and contrasts different renditions of functions in one single place and conclude that the design of functions in a language depends on the interaction of its features and constraints with its functions.

Keywords

First-class functions; Functional programming; Imperative programming; Object-oriented programming; Programming language design; Survey.

Resumo

Santana Rios, Luiz Romário; Ierusalimschy, Roberto. **Um levantamento sobre o suporte a funções como valores em linguagens imperativas**. Rio de Janeiro, 2019. 79p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Se diz que uma linguagem de programação tem funções de primeira classe quando ela fornece a capacidade de manipular funções da mesma maneira que outros valores, isto é, guardar em variáveis, passar como parâmetros, etc.. Programar com funções de primeira classe abre o programador a novas formas de abstração e é o padrão em linguagens de programação funcionais. Porém, se tratando de linguagens imperativas (incluindo linguagens orientadas a objeto), cada linguagem tem semântica, propriedades e terminologia diferentes para funções—em grande parte graças a seu foco em mutabilidade, que as linguagens funcionais não têm.
Para esclarecer essas diferenças, nós fizemos um levantamento sobre a especificação de funções como valores em linguagens de programação imperativas de várias disciplinas diferentes. Para cada linguagem, nós ilustramos, nos baseando em exemplos, as propriedades dos valores de funções nela, destacando onde ela difere de outras linguagens—tudo isso usando uma terminologia consistente em todas as linguagens. Nós esperamos oferecer uma referência para desenvolvedores compararem e contrastarem as diferentes versões de funções num só lugar.

Palavras-chave

Funções de primeira classe; Programação funcional; Programação imperativa; Programação orientada a objetos; Design de linguagens de programação; Levantamento.

Table of contents

List of figures

The programs we use to conjure processes are like a sorcerer's spells. They are carefully composed from symbolic expressions in arcane and esoteric programming languages *that prescribe the tasks we want our processes to perform.*

H. Abelson, G. J. Sussman, *Structure and Interpretation of Computer Programs.*

1
Introduction

In programming languages, having functions as values—or just function values—means the language provides one or more ways for the programmer to access functions for purposes other than to directly apply them, such as: passing functions as parameters, storing functions in arrays, etc.. In the imperative programming style, it's not usual to employ functions as values; whenever that's done, it tends to be an advanced technique.

In contrast, functional programming puts a lot of emphasis on functions as values. Functional languages have the following properties:

First-class function values The function values in functional programming languages can be manipulated in all of the same ways any other value can.

Anonymous functions Functions are not necessarily tied to a name and can be declared as an expression that directly denotes a function value. Having function expressions also means that functions can be nested, since function expressions, like any expression, can be present inside functions.

Lexical scoping Functions can access variables created in their outer creation environment. Given that functions, as first-class values, can be moved away from their creation environment, this means that the external variables they access will be available to them even after the end of the scope in which they were created. Functional languages use closures to implement that

These properties enable the programmer to implement various forms of high-level abstracion by use of functions alone. Some of the applications are the following:

– Creating new functions using runtime values and available functions as parameters (e.g. derivative);
– Storing code to call it later (callbacks, observers, etc.);
– Defining part of a function's behavior as a parameter;
– Change the behavior of the program in runtime.

This emphasis on functions as values has later been incorporated in some imperative programming languages, but this later incorporation means that each of these languages took its liberties to approach first-class functions in a way that was most convenient to their language design — unlike functional languages, which more or less agree on which are the properties of function values. But, since designing and implementing functions with the same properties as those in functional languages has many challenges and, many times, the language has to fit function values in their design, these designs can vary wildly—as well as the terminologies accompanying them.

To shed light on these differences, we surveyed the design of function values in several different imperative programming languages—including object-oriented ones. We looked mostly at their standard references and manuals to learn the terminology and properties of their functions, but we also wrote some experimental programs to assess what's possible to do with function values in each of the languages and how.

In this dissertation, we talk in detail about the properties of functions in each of the languages we surveyed, using consistent terminology across all languages. For each of the languages, we explore a set of examples as a device to talk about the properties of functions in it, the shortcomings of their design, why they're designed in the way they are, and, possibly, some workarounds to their shortcomings.

In Chapter 2, we showcase the power of function values using examples written in the Scheme programming language. Chapter 3 discusses how the approach to functions evolved in imperative languages by chronologically looking at a few languages and taking some examples of textbooks from the literature related to the study of programming languages. Chapter 4 will present and justify the languages we chose for the survey and the terminology that's going to be used throughout the whole text. Chapter 5 is where we present each language and talk about the properties of their function values in detail. Chapter 6 presents the conclusions we drew from the survey and talks about future work.

2
The Power of Function Values

Having functions as values enables the programmer to employ new forms of abstraction other than just creating new functions. To showcase the advantage of having first-class, anonymous and lexically-scoped function values, we wrote some examples in the Scheme programming language (1). Our choice is due to Scheme being a very minimalistic language with function values that have all of the aforementioned properties. These properties allow us to focus entirely on examples that illustrate the properties of functions.

2.1
A Brief Scheme Introduction

We made the examples in this chapter as clear as we could, so the reader doesn't need to know Scheme to understand them, but we're going to briefly present the language in this section, since the syntax of languages from the LISP family might be confusing to people who are not used to them. People who have some level of familiarity with Scheme or other LISPs can skip this section.

S-Expressions The syntax of Scheme is entirely based on *s-expressions*, which are parethesis-delimited, space-separated lists. Every Scheme program, including all its function calls, control structures, data, etc., is a set of s-expressions. This makes Scheme's syntax unusual, but it is actually a very simple syntax once one gets used to it.

```
(display "Hello!\n")
(foo 10 20 "bar")
(define numbers '(10 20 30))
```

Listing 2.1: Examples of s-expressions

Declaration and Assignment Variables are declared with the `define` statement by passing it the variable name followed by its initial value; the `set!`

statement assigns a new value to a variable in a similar way. All variables must be declared before assignment.

```
(define greeting "Hi")
(display greeting)  ; Hi

(set! greeting "Hello")
(display greeting)  ; Hello
```

Listing 2.2: Declaring and assigning a variable

Another form of variable declaration is the `let` expression, which takes a set of variables with their values and a body, which is a set of expressions. The syntax is: `let`, then a list containing a set of variable-value pairs, then the body. The difference between `let` and `define` is that `let` explicitly determines the scope of the variables it declares, i.e. the variable declarations in a `let` expression only hold for its body. The value of a let expression is the value of the last expression in its body.

```
(let ((greeting "Hi\n"))
  (display greeting) ; Hi

  (let ((greeting "Hello\n"))
    (display greeting)) ; Hello

  (display greeting)) ; Hi
; greeting is unavailable here
```

Listing 2.3: Nested `let`s, illustrating that the value of a value defined by `let` only holds for its body

Since the use of `let` expressions can make code harder to read than using `define`s, we will favor the latter.

Functions A function call is denoted by an s-expression with the function name as the first element and the arguments as the remainder of the expression. In Listing 2.2, we are using the `display` function to print the `greeting` variable to the standard out.

The `define` statement is also used to define functions, albeit with a different syntax: `define`, then an s-expression containing the function name as the first element and the parameters as the remainder of the list, and, finally, the body of the function.

```
(define (say-hello name)
  (display "Hello, ")
  (display name)
  (display "\n"))

(say-hello "Romário") ; Hello, Romário
```

Listing 2.4: Function definition in Scheme

The body of a function can have multiple statements, but the last expression is the return expression of the function.

```
(define (cube n)
  (display "Cube of ") (display n) (display "\n")
  (define sqr (* n n))
  (* sqr n)) ; Return expression

; Displays "Cube of 10", then 1000
(display (cube 10))
```

Listing 2.5: `cube` has multiple statements in its body, but only the last one is the return expresssion

Arithmetic Arithmetic operations in Scheme are ordinary functions, and, because of that, are in prefix notation. In other words: `+`, `-`, `*`, and `/`, are all just regular function names.

```
(define (average a b)
  (/ (+ a b)
     2))
```

Control Structures Structures like `if` in Scheme are expressions and, as such, have a resulting value. An `if` expression contains, in the following order, a conditional expression, the expression that holds if the condition is true, and, optionally, the expression that holds otherwise.

```
(define (factorial n)
  (if (> n 0)
      (* n (factorial (- n 1)))
      1))
```

Anonymous Functions Anonymous functions (called *lambdas* in Scheme) are declared with the `lambda` keyword and have a very similar format to named function definitions: the `lambda` keyword, followed by the parameter list, followed by the body of the function.

```
(define (call-twice f a)
  (f a) (f a))

 ; Displays "Hello reader" twice
(call-twice
  (lambda (n)
    (display (string-append "Hello " n "\n")))
  "reader")
```

Lists Lists are one of the most pervasive data structures in Scheme. One way of creating a list is to put a single quote character before an s-expression. This transforms the expression in a list we can access.

```
(display '(10 20 30)) ; Prints (10 20 30)
```

However, this form transforms the elements of the s-expression literally, so variable names are not resolved and function calls just become lists inside the list.

```
(define (sqr n) (* n n))
(define x 10)

; Displays literally (x (sqr x)), not (10 100)
(display '(x (sqr x)))
```

The solution is to use `list`. It allows the programmer to construct lists out of values instead of literal expressions.

```
(define (sqr n) (* n n))
(define x 10)

; Displays (10 100)
(display (list x (sqr x)))
```

Accessing the elements of a list is done with the `car` and `cdr` functions: `car` returns the first element of the list, while `cdr` returns the remainder of the list; when the list is unitary, `cdr` returns `'()`, which is the empty list (or null).

```
(define numbers '(10 20 30))

(display (car numbers))
(display (cdr numbers))
(display (car (cdr numbers)))
```

This set of features is enough to understand most of what's going on in a Scheme code. Other features will be presented as needed in the text.

2.2 Defining by Specifying: Regions

Our first example is based on a prototyping experiment supported by the ARPA that took place between 1993 and 1994. The goal of the experiment was to implement a simplified version of a tool called *geo-server*, which deals with regions in a 2D geometric space. The experiment compared many different languages at this task and the problem was given to an expert programmer in each of these languages. Each programmer had the freedom to model the regions however they wanted (2).

While the original article was focused in the Haskell implementation, we can easily translate the approach used by the Haskell programmer to Scheme, since it is based on the employment of functions as values.

2.2.1 Regions as Predicates

There are several possible approaches that can be used to define an arbitrary geometric region, but the most direct approach is to define it as a predicate that holds true for a given point if the shape contains that point. For example, given a point p, p is inside a circle with radius r and center in c if the following inequality holds true: $(p_x - c_x)^2 + (p_y - c_y)^2 < r^2$.

There are also several possible approaches to implement this definition in software. In most cases, when faced with such a problem, programmers start thinking about what *data structures* to use in order to represent each of all the possible geometric regions (circles, rectangles, etc.), what *algorithms* will be used to check if a point is in a given region, etc.. However, by employing functions as values, we can represent *any arbitrary region* by just directly using a predicate in the way we described above.

Suppose we use the following definitions for points:

```
(define (point x y) (cons x y))
(define (x-coord p) (car p))
(define (y-coord p) (cdr p))
```

Our circle definition can then be coded as:

```
(define (circle r c)
  (lambda (p)
    (< (+ (sqr (- (x-coord p) (x-coord c)))
          (sqr (- (y-coord p) (y-coord c))))
       (sqr r))))
```

Aside from the syntax, this definition is exactly the same as the mathematical definition of a circle we saw above: the `circle` function takes a radius `r` and a point `c` and returns a predicate that holds true if a given point `p` is inside such a circle. This function *creates*, at run-time, a predicate for one specific circle.

As another example, a rectangle would be very simple to define with this scheme. To check if a given point is inside the boundaries of a rectangle, it's only necessary to check if the coordinates of the points are between all four sides of the rectangle—which, in turn, can be defined by its top-left and bottom-right points. This rectangle definition can then be coded as follows (where `tl` is the **t**op-**l**eft point of the rectangle and `br` the **b**ottom-**r**ight one):

```
(define (rectangle tl br)
  (lambda (p)
    (and (and (> (x-coord p) (x-coord tl))
              (< (x-coord p) (x-coord br)))
         (and (> (y-coord p) (y-coord tl))
              (< (y-coord p) (y-coord br))))))
```

2.2.2 Using Regions

Now, since any region is a predicate, to check whether a point is inside a region or not, all we need to do is to pass the point as an argument. For example, in a REPL[1] prompt, checking whether the point $(30, 40)$ is inside a circle `c` of radius 60 at the center of the plane (i.e. at point $(0, 0)$) looks like this:

[1]Read-eval-print loop.

```
> (define c (circle 60 (point 0 0)))
> (c (point 30 40))
#t
```

Similarly, to check whether point $(20, 30)$ is inside a rectangle with its top-left edge at $(-30, 10)$ and its bottom-right edge at $(0, 20)$, we do the same as before:

```
> (define r (rectangle (point -30 10)
                       (point 0 20)))
> (r (point 20 30))
#f
```

Note that, since any region is just a predicate, checking whether a point is in a region is *exactly the same* for any possible region, however arbitrary it might be. It is also the simplest operation possible: a single function call. This is the highest possible abstraction of such an operation and it is only possible because we are able to create arbitrary predicates, at runtime, for each of the possible shapes we can have.

Plotting the region is very simple with this representation. All we need to do is to paint all the points contained inside the region, i.e., for each pixel in the plotting area, we use the region predicate to check whether the point represented by that pixel is contained in the region; if so, we paint it black.

To illustrate the simplicity of this approach, we implemented a `plot-pgm` function that takes a region and plots it to a 256x256 Plain PGM[2] image file, which required just two nested loops and an if expression inside those loops—file handling and image headers aside (Listing 2.6). Figure 2.1 shows the generated images for the circle `c` and the small rectangle `r`, defined above.

2.2.3 Combining Regions

An annulus is a ring-like region that can be represented by two radiuses (an internal one and an external one) and a single center. It can be constructed as the intersection between the outside of a smaller circle and the inside of a larger circle. Given that a region is just a predicate, the intersection between two regions is the region where both of their predicates hold true, i.e. a boolean *and* operation; similarly, the outside of a region is a predicate that holds true for any point not in the region—in other words, it is a boolean negation of the original predicate.

[2] `http://netpbm.sourceforge.net/doc/pgm.html`, accessed in July 9th, 2019.

```
(define (plot-pgm region)
  (define image (open-output-file "image.pgm"))
  (display "P2 256 256 1" image)
  (newline image)
  (for-range -128 127 (lambda (y)
     (for-range -128 127 (lambda (x)
        (if (region (point x y))
            (display "0 " image)
            (display "1 " image))))))
  (close-output-port image))
```

Listing 2.6: The `plot-pgm` function takes a region and plots it in a PGM image. In order to have the center of the image approximately located at the $(0, 0)$ coordinate, the coordinates go from -128 to 127.

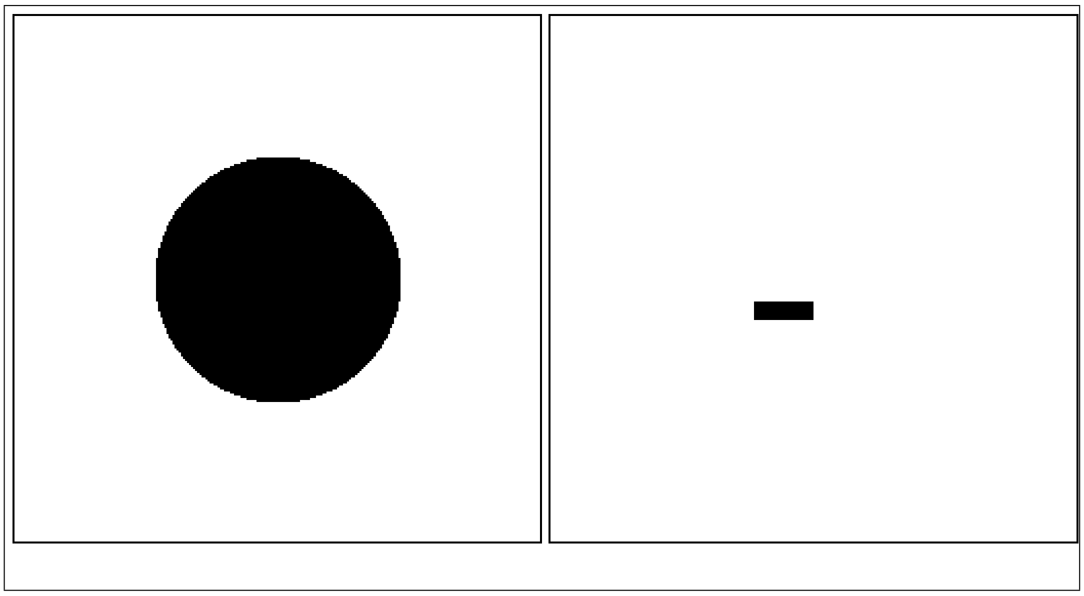
Figure 2.1: The circle `c` and the rectangle `r`.

Having regions as predicates not only allows the *definition* of brand new regions to be as direct and abstract as possible, but the *combination* and *modification* of existing regions becomes trivial, which is reflected in the code:

```
(define (outside region)
  (lambda (p) (not (region p))))

(define (intersect region1 region2)
  (lambda (p) (and (region1 p) (region2 p))))
```

With that, we can define the annulus (Figure 2.2, shows how an annulus looks like when plotted):

```
(define (annulus r1 r2 c)
  (intersect (outside (circle r1 c)) (circle r2 c)))
```

```
> (plot-pgm (annulus 60 120 (point 0 0)))
```

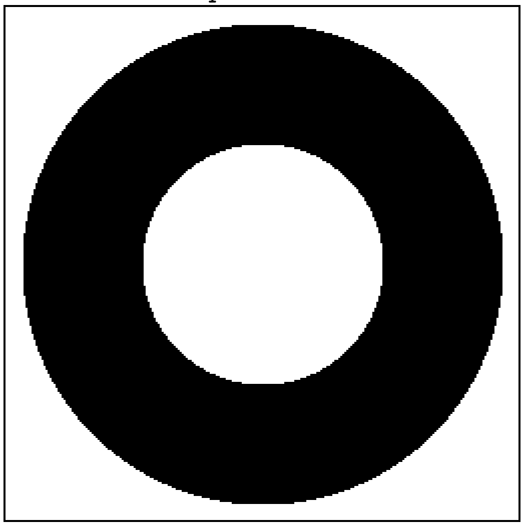

Figure 2.2: Centered annulus with inner radius of 60 and outer radius of 120

Merging two regions is also trivial: it is exactly the union operation:

```
(define (merge region1 region2)
  (lambda (p)
    (or (region1 p) (region2 p))))
```

As an example, Figure 2.3 shows a picture created with `merge`.

All of this is only possible because of the features of functions in Scheme. In particular, having lexical scoping and first-class function values is fundamental to arbitratily create and combine predicates. This level of abstraction simplified a moderately complex task to its very definition. A testament to that is the reaction of the other participants and reviewers. In particular, we quote:

> It is significant that Mr. Domanski, Mr. Banowetz and Dr. Brosgol were all surprised and suspicious when we told them that Haskell prototype P1 (...) is a complete tested executable program. We provided them with a copy of P1 without explaining that it was a program, and based on preconceptions from their past experience, they had studied P1 under the assumption that it was a mixture of requirements specification and top level design. They were convinced it was incomplete because it did not address issues such as data structure design and execution order. (Carlson, Hudak, and Jones (3))

```
> (plot-pgm (merge (circle 20 (point 65 -65))
                   (circle 60 (point 0 0))))
```

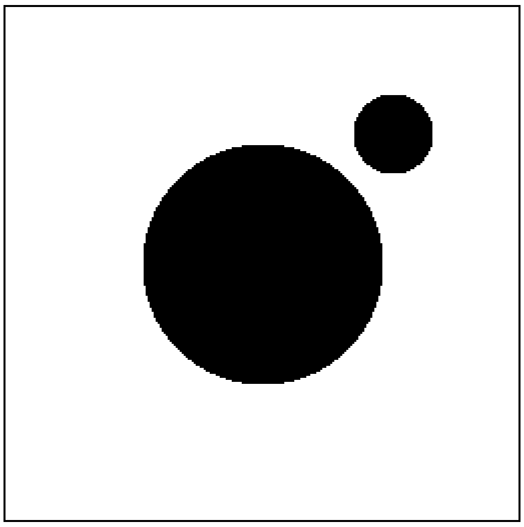
Figure 2.3: A very simplified version of the Lua programming language logo

2.3 Dynamically Composing Functions: Iterators

Suppose we have the following data structures:

```
(define l '(57 674 2 78))
(define v #(78 258 3 57))
```

A list prefixed with # denotes a *vector*. Vectors are more compact than lists because they're allocated in a contiguous region in memory and their elements don't need references to the next element.

These structures are sequences of numbers. So, if we want to, say, print each of the numbers in these sequences, the task would be the same for both `l` and `v`: get each number, print it. But, even though the task is the same for these sequences, `l` is a list and `v` is a vector, and, thus, their elements are accessed in completely different ways: `car` and `cdr` in the case of `l`; `vector-length` and `vector-ref` in the case of `v`. This means that the same task might have to be implemented twice just because of incompatible data types. In Listing 2.7, we implement two functions for that purpose: one to print the elements of a list (`print-list-els`) and another for the elements of a vector (`print-vector-els`).

One solution for this specific example could be keeping the simplest function (i.e. `print-list-els`) and converting `v` to a list using the `vector->list` function each time we want to print the elements of a vector. While it works, there are some problems with this solution: it has a conversion overhead when compared to `print-vector-els`, making printing a vector strictly worse than

```
(define (print-list-els l)
  (if (null? l) '()
      (begin (display (car l))
             (display " ")
             (print-list-els (cdr l)))))

(define (print-vector-els v)
  (let ((i 0))
    (define (loop)
      (if (>= i (vector-length v)) '()
          (begin (display (vector-ref v i))
                 (display " ")
                 (set! i (+ i 1))
                 (loop))))
    (loop)))
```

Listing 2.7: Two different functions that perform the same task on two different datatypes (the `begin` expression allows us to put multiple expressions where a single expression is expected)

printing a list; another more general problem is that a conversion function is not always available between any two types.

A better solution for this kind of problem are *iterators*. An iterator is an object that enables the program to sequentially visit (or iterate through) elements of a container. The advantage of an iterator over regular loops is that they abstract away implementation details about how to get to the next element, allowing the programmer to implement algorithms on top of this abstraction, untying these algorithms from the concrete data structures they're working on.

We define an iterator as a function that takes no arguments and returns the next element of the sequence when called; when the iteration ends, the iterator returns null (i.e. `'()`) for all subsequent calls.

Following that definition, an iterator for a list has to be a function that returns the head of that list, taking it with `car`, and keeps its remainder, with `cdr`. In Listing 2.8, we create an `it-list` function that takes a list and returns a function that keeps a reference to the list it takes, doing what we described at each call—the remainder of the list is kept in that reference.

For the vector, we can make the iterator keep the index of the next element. At each call, we return the element at that index (with `vector-ref`), then increment the index; if the index is equal to the length of the vector (which we check with `vector-length`), we return null. The function that creates such an iterator from a vector is called `it-vector`[3] (Listing 2.9).

[3]As a convention, we prefix every function that constructs an iterator with `it-`.

```
(define (it-list l)
  (lambda ()
    (if (null? l) '()
        (let ((cur (car l)))
          (set! l (cdr l))
          cur))))
```

Listing 2.8: `it-list` takes a list `l` and returns an iterator that, at each call, returns the first element of `l`, keeping only the remainder

```
(define (it-vector v)
  (define i 0)
  (lambda ()
    (if (>= i (vector-length v)) '()
        (let ((cur (vector-ref v i)))
          (set! i (+ i 1))
          cur))))
```

Listing 2.9: The vector iterator stores a reference to the index of the element to be returned in the next call

Having defined these iterator constructors, we now create the `print-els` function, which prints the elements of an iterator (Listing 2.10). It repeatedly calls the iterator it gets, printing each element it gets from each call, then stops when the iterator returns null.

```
(define (print-els it)
  (define it-result (it))
  (if (not (null? it-result))
      (begin (display it-result)
             (display " ")
             (print-els it))))

(print-els (it-list l))   ; 57 674 2 78
(print-els (it-vector v)) ; 78 258 3 57
```

Listing 2.10: `print-els` takes an iterator and recursively takes each of its elements, printing each of them

We have successfully separated a task to be done over a sequence of elements from the underlying data structure of this sequence. The `print-els` function works for both lists and vectors because it leaves the concern of accessing the elements of the data structures to the iterators.

But `print-els` still does more than one thing: it prints the elements of the iterator it gets but also has to deal with the *the process of iteration*. At the current point, writing any function that consumes an iterator would require including the boilerplate to deal with the iteration process.

To generalize the process of iteration, we wrote a `foreach-it` function for our iterators, that does the same thing `print-els` does, but, instead of printing each element of the iterator, it passes the element to an arbitrary function (Listing 2.11). This allows anyone who wants to consume the iterator to pass any function they want to deal with each element of the iterator. With `foreach-it`, the definition of `print-els` becomes trivial.

```
(define (foreach-it it function)
  (define it-result (it))
  (if (not (null? it-result))
      (begin (function it-result)
             (foreach-it it function))))
```

Listing 2.11: The `foreach-it` function is almost the same as `print-els` except that, instead of `display`ing the result of the iterator, it passes that to another `function`

```
(define (print-els it)
  (foreach-it it (lambda (el) (display el)
                              (display " ")))
  (newline))
```

Listing 2.12: After the definition of `foreach-it`, `print-els` becomes a one-liner

Note that all the abstractions that were done so far are based on functions being employed as values: even though an iterator is a function, we almost never call it directly and we don't directly `define` an iterator as a function either—instead, we use functions that return an iterator (`it-list` and `it-vector`). Also note that having functions that can refer to outer functions after their scope ends is fundamental for our iterators to work, since that's how they keep their state: an iterator created by `it-list` uses the `l` variable (taken by `it-list` as an argument) to keep the remainder of the list; one created by `it-vector` keeps the index of the next element in the `i` variable, declared in the body of `it-vector` with a value of `0`.

2.3.1 Composing Iterators

What we have defined so far allows us to create all sorts of iterator-based algorithms without ever needing to call the iterators directly, but we still are only able to perform some task per iterator element with that.

Suppose we want to get an iterator of a sequence of numbers and return a sequence of their squares. We could create a function that takes an iterator and, for each element, squares that element and stores it into either a list or a vector, then returns that container. This would work fairly well and, thanks to iterators, this function is ready to work with lists, vectors, and anything else that implements an iterator.

But the problem with this solution is that we're returning a specific data type and, again, we have to deal with its specificities if we want to get its elements. This becomes clear when we consider that, no matter the type of underlying data structure, the type of the sequence returned by our function would be fixed to what we chose in our implementation. We are taking a value that abstracts away the specificities of the underlying data type and returning a value that doesn't.

To avoid these problems, instead of storing our results into some data structure, we make our function return an iterator that, at each call, calls the original iterator and returns the square of its result. This is the `it-sqr-all` function in Listing 2.13. Instead of worrying about how we will square all numbers of the given iterator, we worry solely about what to do with each specific element. Since the result of `it-sqr-all` is an iterator, we can pass it directly to `print-els` if we want to print the elements right away.

```
(define (it-sqr-all numbers)
  (lambda () (define n (numbers))
             (if (null? n) '()
                 (* n n))))


; Prints 1 4 9
(print-els (it-sqr-all (it-list '(1 2 3))))
```

Listing 2.13: Instead of storing the squares of each element, `it-sqr-all` returns an iterator that calls the iterator whenever it is itself called, then returns the square of the result

With this approach, instead of consuming the iterator to produce a concrete list of the results we want, we produce another iterator that encapsulates the original one, adding an operation on top of it—in this case, squaring. The resulting iterator defines how the result of each call to the underlying iterator is going to be modified at each iteration.

This composition of operations on top of existing iterators can be generalized. When we want to transform each element of an iterator into another given a function that performs such a transformation, we want a *map* operation. We define it as the `it-map` function, which takes an iterator and a function; it returns an iterator that, in each call, applies the function to the result of the underlying iterator.

```
(define (it-map it f)
  (lambda () (define res (it))
             (if (null? res) '()
                 (f res))))


; Prints 1 4 9
(print-els (it-map (it-list '(1 2 3))
                   (lambda (n) (* n n))))
```

Listing 2.14: An iterator created by `it-map` applies the result of each call to `it` to the function `f`

Another general operation that can be performed on top of iterators is *filtering*, which selects a set of elements from a sequence given a predicate. Our filter function is called `it-filter` and, given an iterator and a predicate, it returns an iterator that filters out values that fail on the predicate. To do this, at each call, the filter iterator calls the underlying iterator and checks if the predicate succeeds; if it does, then the iterator returns the value; if it does not, then the iterator calls itself recursively until the predicate succeeds—or until it reaches the end.

```
(define (it-filter it c)
  (define (new-it) (define res (it))
                   (cond ((null? res) '())
                         ((c res) res)
                         (else (new-it))))
  new-it)

; Prints 3 4
(print-els (it-filter (it-list '(3 1 4 2))
                      (lambda (n) (> n 2))))
```

Listing 2.15: The iterator created by `it-filter` calls itself recursively until the predicate `c` succeeds on the value of `it` (`cond` takes a list of conditions and, for each condition, the result if it holds true)

It is now possible to create and combine iterators and manipulate the data coming from them without having to deal with any data structures. As an example, if we want to get the cubes of the integers from 1 to 10, but only the ones larger than 100, we can do it by creating an iterator from a list containing the numbers from 1 to 10 (with `it-list`), then mapping it to get the cubes of each element (using `it-map`), then, finally, filtering it to remove the resulting elements that are larger than 100 (using `it-filter`).

```
; Prints 125 216 343 512 729 1000
(print-els (it-filter
            (it-map
             (it-list '(1 2 3 4 5 6 7 8 9 10))
             (lambda (n) (* n n n)))
            (lambda (n) (> n 100))))
```

Listing 2.16: Printing the cubes of the numbers from 1 to 10 which are larger than 100

2.3.2 More Abstract Iterators

So far, we've been mostly operating on lists in our examples. Sometimes, to illustrate the uniformity iterators bring when dealing with different data structures, we show some vector examples. But our iterators do not need an underlying data structure. As we've said, iterators are, ultimately, just functions that iterate through a sequence, one element at a time; such a sequence does not need to be a concrete list of elements.

In Listing 2.16, we don't need the explicit list of elements from 1 to 10: we just need an iterator that gives us the numbers from 1 to 10. While taking an iterator to a list that contains an explicit sequence of those numbers might work, the problem with this redundancy becomes clear if we start requesting larger sequences—say, all numbers from 1 to 20, or to 50.

In Listing 2.17, we define an `it-range` function, which creates an inclusive range of numbers without the need for a underlying data structure: the only data the iterator carries are the current number and the last number in the range. At each call, the variable holding the current number is incremented and the current number is returned.

As a last example, we show that we don't need to be limited to iterators with a predetermined number of elements. In the case of `it-range`, we can separate the task of counting numbers from the task of getting a determined amount of elements from an iterator; we create one iterator that just counts endlessly, starting from 1 (`it-count`), and another iterator that takes an existing iterator and only gets a determined amount of its first elements (`it-get-n`). In Listing 2.18, we show that this can produce the same results as `it-range`, even though one of the iterators being composed does not end.

```
(define (it-range start end)
  (lambda () (if (> start end) '()
                 (let ((current start))
                   (set! start (+ start 1))
                   current))))

; Prints 125 216 343 512 729 1000
(print-els (it-filter
            (it-map
             (it-range 1 10)
             (lambda (n) (* n n n)))
            (lambda (n) (> n 100))))
```

Listing 2.17: Defining a range iterator with no underlying data structures

The implementation of this iterator architecture is only possible due to the properties of function values in Scheme: if our iterators weren't fist class values, we would not be able to return them from their constructors, or pass them to the functions we defined to manipulate them; with no nesteable anonymous functions (or some other form of function nesting), we wouldn't be able to abstract away the implementation of the iterators from their callers; with no lexical scoping, the iterators wouldn't be able to access the outer variables they use to carry their state. Implementing similar iterators in languages without these properties (like C, for instance) is possible, but is much more foreign, because all these features that are granted by Scheme would have to be taken care of manually by the programmer.

```
(define (it-count)
  (define count 0)
  (lambda ()
    (set! count (+ count 1))
    count))

(define (it-get-n it n)
  (define c 0)
  (lambda ()
    (if (>= c n) '()
        (begin (set! c (+ c 1))
               (it)))))

; Prints 125 216 343 512 729 1000
(print-els (it-filter
            (it-map
             (it-get-n (it-count) 10)
             (lambda (n) (* n n n)))
            (lambda (n) (> n 100))))
```

Listing 2.18: `it-count` counts numbers from 1 to infinity. Together with `it-get-n`, it allows us to iterate through a range of numbers

3
Brief History

The employment of functions as values was not always emphasized in programming languages. This is especially true for imperative languages, that only more recently are turning their attention to functional programming. In this section, we intend to present a little bit of the history of the evolution of functions as values in imperative programming languages. First, we select a few languages to illustrate how functions evolved over time as a feature in imperative languages; then, we look into some programming-language textbooks to illustrate how functions were seen by the community in the past and how that view changed.

3.1
Functions in Imperative Languages

Here, we present some examples to illustrate how the employment of functions in imperative programming languages evolved over time.

3.1.1
Algol 60

Using functions as values already had some support early on in imperative programming languages. Algol 60 was one of the earliest imperative programming languages and it already had some of the features mentioned in the introduction that we considered fundamental to fully take advantage of functions as values:

Nested procedures with proper lexical scoping Algol's original specification says: "Identifiers in the procedure body (...) which are non-local to the body may well be local to the block in the head of which the procedure declaration appears" (4, 5.4.3). In other words, identifiers that are local to the block where a procedure is declared are visible from inside the procedure as if they were local—and, by extension, the same applies to identifiers in outer blocks. Since the body of a procedure is a block and it is possible have a procedure declaration inside a block, nested procedures are supported by Algol.

Passing procedures as parameters Algol's syntax for procedure declarations says that each of the formal parameters of a procedure can have a `<specifier>`, which determines what kind of parameters they are. One of the possible `<specifier>`s is `procedure`, meaning that any parameter can be a procedure (4, 5.4.1).

However, procedures can neither be returned by other procedures nor stored in variables: in the first case, to return a value, a procedure needs to specify its return type, but there are no procedure *types*, since `procedure` is a *specifier*, not a type; in the second case, the right-hand side of an assignment has to be either arithmetic (i.e. numeric) or boolean (4, 4.2.4)—these were the only first-class values available, since Algol 60 didn't have composite values yet. The only purpose of passing a procedure as an argument to a function is to either call the passed procedure immediately or to pass it further to an inner procedure call.

The specification provides two examples to illustrate the declaration of high-order procedures in Algol (4, 5.4.6). The first example, for instance, (Listing 3.1) computes the sum of a given function from zero up to infinity using the Euler method. This is a very common application of higher-order functions and Algol is fully capable of expressing it. However, these properties do not provide enough abstraction capabilities to, for example, allow the *indefinite* integral to be directly computed and provided as a value. None of what was done in Chapter 2 would be possible either.

```
procedure euler(fct, sum, eps, tim);
    value eps, tim;
    real procedure fct;
    real sum, eps;
    integer tim;
```

Listing 3.1: The `euler` procedure, as expressed by the original comment of the example, "computes the sum of `fct(i)` from `i` from zero up to infinity by means of a suitably refined euler transformation" (here, we show the function without its body, for brevity).

All of this is evidence that, while high-order functions were already present early on, procedures were seen as their own kind of entity, entirely separate from the values in the language.

3.1.2 Pascal

According to Sebesta (5, p. 97), Pascal was the second language design by Niklaus Wirth which was directly based on Algol 60—preceded by Algol-W. Among other features, Pascal retained Algol's high-order procedures. But even though this feature was kept, Wirth didn't think it was relevant enough to warrant a discussion: passing procedures as arguments is not mentioned anywhere in Pascal's first specification document (6)—it can only be implied from the grammar rules; also, in the user's manual, passing procedures and functions as parameters is only briefly mentioned, together with a single example in the same page (7, p. 79).

3.1.3 C

The C programming language was created with simplicity, portability, and efficiency in mind—according to Chapter 0 of *The C Programming Language* (8). This made C a relatively low-level language when compared to contemporaries like Algol and PL/I. C's design of functions reflects those goals: one of C's main ideas is that any program is a collection of functions and that each function is separately compiled and the only form of automatic memory management was the stack (in contrast with Algol 68, which already had garbage collection).

C didn't allow nested functions. While the reasoning for this decision isn't explicitly explained by the language authors, one of the possible reasons could be to make memory management simpĺer to implement. After all, unlike Algol or Pascal, function pointers are first-class values in C; the management of outer variables accessed by a nested function could be difficult to implement in such a scenario. The authors even recommended using external (global) variables to share state between functions:

> Because external variables are globally accessible, they provide an alternative to function arguments and returned values for communicating data between functions. (...)
>
> The third reason for using external variables is their scope and lifetime. Automatic [stack] variables are internal to a function; they come into existence when the routine is entered, and disappear when it is left. External variables, on the other hand, are permanent. They do not come and go, so they retain values from one function invocation to the next. (...) (p. 72 (8))

As the authors themselves point out, this approach was error-prone, but restricting functions to access only their own local variables and global variables makes the implementation of functions much simpler than having nested functions with static scoping.

Another reason was to make it possible to keep each function separate from all other functions, favoring the creation of several simple functions rather than few, large functions, with functions embedded in them. This also makes it possible for functions to be compiled independently of others:

> C has been designed to make functions efficient to use; C programs generally consist of numerous small functions rather than a few big ones. A program may reside on one or more source files in any convenient way; the source files may be compiled separately and loaded together, along with previously compiled functions from libraries. (p. 65 (8))

Despite not allowing nested function, higher-order functions were already in the mind of the creators of C, which is why they provided function pointers. For example, Section 5.12 of the book is about pointers to functions. In that section, the book use sorting to illustrate the use of functions pointers, which is a classic example of high-order functions.

3.1.4 C++

C++ appeared in the early-to-mid-1980's and two main sources of inspiration to it were C and Simula67 (9, p. 3-4).

One form of function value in C++ is the function pointer, which was inherited from C. But, since the first published book, the C++ programming language already supported the overloading of the function call operation (9, p. 183-184), which can turn any class into a callable object and, as we see in Section 5.7.2.4, was the basis for anonymous functions in C++11.

Although brief, the motivation provided for this feature seems to be providing function-like objects that carry data, something that is not possible with function pointers. After defining an iterator type as a class that acts as a function that gets the next element from a container (somewhat similar to what we did in Section 2.3), the author affirms the following:

> An iterator type like this has the advantage over a set of functions doing the same job: it has its own private data for keeping track of the iteration. It is typically also important that many iterators of such a type can be active simultaneously. (p. 184 (9))

Both of these points—keeping the data of a computation private and keeping multiple instances of the same computation active simultaneously—are clearly also motivations for first-class functions with closures.

Since, at the time, C++ still did not have templates, the fact that each class that overloaded the operator call was from a different type prevented any meaningful application of these types as function objects. However, the introduction of templates made it possible to implement high-order template functions that can take any callable object, whether it is a function pointer or a class with the operator call overloaded.

3.1.5
Python

Python was strongly influenced by the ABC programming language, where Guido, the creator and main designer of Python, previously worked on in the late 80s (10). ABC's main focus was to be easy to use by non-programmers who had to work with computers. Python, in its inception, was directly based on Guido's experience on working with ABC, but he also implemented an extensibility model in C, something which ABC never considered to be important. The initial goal of Python was to be an auxiliary language for C programmers if they needed to peform some task where the use of C would be overkill—to "bridge the gap between the shell and C", in Guido's own words (11).

Perhaps because of the connection to C and the focus in non-expert programmers, functions as values were not much of a concern for Python initially and it was only when more experienced programmers started feeling the lack of functional properties in the language that these features started being incorporated. A strong evidence to this idea lies in the creation of the `lambda` construct in Python. While explaining the history of `lambda` in Python in a blogpost from 2009, Guido makes it clear that he wasn't interested in functions as values by the time he created Python:

> I was much more familiar with imperative languages such as C and Algol 68 and although I had made functions first-class objects, I didn't view Python as a functional programming language. However, earlier on, it was clear that users wanted to do much more with lists and functions. (Guido van Rossum (12))

He then explains that, even though Python had no way to construct anonymous functions, users found ways to work around that. The code in Listing 3.2, given by Guido in the blogpost, was a common approach at the time (around late 1993). This made it clear that there was a demand for anonymous functions:

> It was clear that there was a demand for such functionality. However, at the same time, it seemed pretty "hacky" to be specifying anonymous functions as code strings that you had to manually process through exec. Thus, in January, 1994, the map(), filter(), and reduce() functions were added to the standard library. In addition, the lambda operator was introduced for creating anonymous functions (as expressions) in a more straightforward syntax. (Guido van Rossum (12))

```python
def genfunc(args, expr):
    exec("def f(" + args + "): return " + expr)
    return eval("f")

# Sample usage
vals = [1, 2, 3, 4]
newvals = map(genfunc("x", "x*x"), vals)
```

Listing 3.2: Without `lambda`, users resorted to metaprogramming in order to emulate anonymous functions

This pattern of ignoring a functional feature then adding it due to user demand was common during the evolution of Python. Like its contemporaries, Python's design wasn't open to the use of functions as values because its main designer came from a more traditional school of imperative programming, where procedures and values were different entities, and could not see the value in that. However, the users, used to the growing acceptance of functional programming, did see it and demanded more focus to those features.

3.2 Evolution in the Literature

Most of the mainstream literature on programming languages (most notably textbooks) used to make a clear distinction between functions and other values in the language, and the employment of functions as values was reserved to functional programming and mentioned in imperative programming languages as either a nuisance to be aware of or as an advanced or modern application of functions. Around the late-2000s and early-2010, however, that seemed to change and more emphasis was put into the applications of functions as values in imperative languages. In this section, we take a few examples from the literature to illustrate the discussion about how the approach to functions in imperative programming languages changed over the years.

Structure and Interpretation of Computer Programs (13) was a book first released in 1985 that saw a second edition in 1996 (which is the one we use as reference here). This book is a prime example of putting functions front and center in the modelling of problems. From the very first chapter, "Building Abstractions with Procedures", it is made clear that functions are not only powerful, but are actually capable of being the backbone of a program, providing a foundation for anything else. A clear example of that is the "Streams" example, presented in Section 3.5 of the book, which is very similar to our iterators example from Section 2.3.

Given that the Scheme programming language was created in 1975 (and that it had first-class functions with closures since the beginning) (14), it might seem that, in 1996, employing functions as values to build powerful abstractions wouldn't be something new. However, much later than that, having functions as values was treated almost as a corner case in much of the imperative programming camp. For example, in the second edition of *Programming Language Pragmatics* (15), released ten years later, in 2006, Scott does talk about first-class *subroutines*, but presents them as a subtopic of scoping and environments, treating them more as an implementation concern rather than as a tool that programmers can use:

> (...) First-class subroutines in a language with nested scopes introduce an additional level of complexity: they raise the possibility that a reference to a subroutine may outlive the execution of the scope in which that routine was declared.
>
> (...) imperative languages with first-class subroutines must generally adopt alternative mechanisms to avoid the dangling reference problem for closures. C, C++, and Fortran, of course, do not have nested subroutines. Modula-2 allows references to be created only to outermost subroutines (...) (Michael L. Scott (15), p. 140-141)

He focuses on the *problems* first-class function values introduce and how to solve them, not about their applications. The only other mention of first-class functions is in the end of the book, where he talks about functional programming, presenting it as an alternative programming model. The employment of functions as values has no place in imperative programming. In its third edition (2009), the emphasis remains the same, but now the book talks about *object closures*, which was something known to C++ since at least

1986—and it still remains buried as a subtopic of scopes and environments. Only in its fourth edition (2016) the book starts talking about *lambda expressions*.

Other books first released before the late 2000s show the same approach to functions. For instance, Sebesta's *Concepts of Programming Languages* was first released in the late 1980s and released many editions throughout the decades of 1990 and 2000. In its eleventh edition (5), released in 2016, the book dedicates an entire chapter of his book (Chapter 2) to discuss the evolution of major programming languages over time. However, aside from the functional programming section of the chapter, where he uses Lisp to talk about the evolution of functional programming (Section 2.4), and a few brief mentions of terms related to functions as values[1], at no point, in any of the languages, is the evolution of *functions* specifically discussed. Again, the discussion of the employment of functions as values is confined almost exclusively to functional programming. Even the discussion of functional programming in this chapter seems to lack focus on functions as values: first-class functions are only mentioned in the paragraph where the text talks about Scheme.

The sharp distinction between functions and values becomes very clear in Watt's *Programming Language Design Concepts* (16), released in 2004. In Chapter 2, "Values and types", function values are tangentially mentioned *once* (p. 46, function types are not mentioned), while in Chapter 3, "Variables and storage", nothing is mentioned about assigning functions to values. Even Chapter 5, "Procedural abstractions", dedicates only one paragraph and one example to anonymous functions (p. 118).

As a last example, we mention the third edition of *Programming Language Concepts*, by Ghezzi and Jazayeri, released in 1996 (30). As the other books, it does not emphasize the use of functions as values in imperative languages throughout the book: routine parameters are discussed relatively well in the syntax and semantics chapter, but they are treated as a corner case and are not discussed anywhere else; there is one chapter dedicated to functional programming and any discussion about functional features is reserved to this chapter. However, in Section 7.5, inside the chapter about functional programming, the book discusses how C++ can support high-order functions by the use of templates and function objects. This shows that there was already some interest to have functional features in imperative languages at the time.

The late 2000s and early 2010s seemed like a turning point in the

[1]At the very end of Section 2.17 (p. 114), it is *very briefly* mentioned that Java SE 8 has lambda expressions; Subsection 2.18.6 mentions that functions are first-class values with closures in Lua; Section 2.19 mentions delegates (p. 123), contrasting them with C++'s function pointers.

treatment of functions by the imperative language community. In the case of Sebesta and Scott, the editions released after this point in time included many additions and improvements to the treatment of function values in imperative languages. However the *structure* of those books is still strongly impacted by the view that functions (or subroutines in general) and other values were not of the same nature, which was dominant at the time these books were first published.

On the other hand, books that were first published after that time period had their attention much more clearly turned to functions as values. For example, Sestoft's *Programming Language Concepts* (17), released in 2012, presents itself as a general introductory book about the study of programming languages, but it uses F# (a language in the ML family) as its main language and, in Section 3.5, has a very brief survey of "Higher-Order Functions in the Mainstream", where it brings up the use of functions as values in Java and C#.

This trend of growing acceptance of first-class functions with closures is confirmed in Chapter 5, where we explore the properties of functions in some imperative programming languages. There we see how many of those programming languages recently adopted features which contribute to that goal, such as: anonymous functions, better static scoping, etc..

4
Survey Presentation

In this section, we present our choice of languages for the survey and the terminology we will use throughout the presentation of function values in the next chapter.

4.1
Chosen Languages

To conduct out survey, we chose a set of languages to illustrate how the design of functions varies depending on many aspects of the language—its paradigms, its implementation, etc. In this section, we present each of these languages and justify why we believe they're relevant to the survey.

The goal of this survey is to, ultimately, contrast many different designs of functions from many different programming disciplines. We chose a set of sufficiently known programming languages that are distinct enough, so that we can showcase how function values fit in each of the different paradigms shown here. This set of languages is by no means comprehensive, but we believe it is broad enough to make this contrast.

The languages we chose are: Scheme, Lua, Go, Python, Ruby, Java, C#, C++, and Rust.

4.1.1
Choice Criteria

Scheme is our lingua franca. Its minimalism allows us to focus on the properties of function values without too much interference from other language features.

Lua (18) is also a minimalistic dynamic programming language with functions that share all of the same properties of functions in Scheme; in fact, the semantics of functions in Lua closely resemble Scheme.

The **Go** (19) programming language is a statically-typed, compiled programming language. We chose Go because, even though it is not as dynamic as Scheme, it still has first-class functions with closures. It serves as an illustration of how a more static language can still choose to have the same properties of Scheme regarding functions.

Python (20), on the other hand, illustrates that a language can be dynamic and still not emphasize function values in the same way Scheme does. While Python does have first-class functions with closures, some shortcomings regarding scope and anonymous functions arise from such a lack of emphasis.

Ruby (21) is a dynamic, object-oriented programming language. It has first-class functions and closures, but, since Ruby is inspired by Smalltalk's object-oriented model, the forms of functions it has (blocks, methods, procs) are very particular in design.

Java (22) and **C#** (23) are two similar object-oriented programming languages, where all function definitions are tied to classes. Because of this similarity, these languages had similar solutions to fit anonymous functions as objects in them.

Finally, we chose **C++** (24) and **Rust** (25)(26) because they are two lower-level programming languages that do not have garbage collection, but managed to implement closures. The design of anonymous functions in these languages is directly impacted by the lack of garbage collection.

4.2 Terminology

To uniformize the comparison of the properties of different languages, we decided to refer to language features in our own terms instead of using the language's official terminology. The translation from language terminology to our terminology will be made in the introduction of each of the languages.

Function Anything that is callable and can take arguments in a programming language will be called "function". This, of course, includes regular functions, but it also includes other "function-like" entities that can go by the names of "method", "block", "closure", etc..

Function Value Any value in the language that gives access to a function. If such value can be used and passed around in any context a value can be used, without any restrictions—i.e. if it can be returned from functions, passed as an argument, stored in variables, etc.—we call it a *first-class function value.*

Function Type The type of a function value.

Function Definition A syntactic construct in the language dedicated to the creation of a function tied to a name. Ex.: `define` in Scheme as described in Listing 2.4.

Anonymous Function A function without a name. These are usually function values as well, but not always. They are often also called *lambdas*—as seen in Chapter 2.

Closure The data structure that holds a function and the outer variables referenced by it.

5
Properties of Function Values

Unilke functional programming languages, imperative languages have great variety of function designs due to their different paradigms, goals, and constraints. In this chapter, we will analyze function values in the languages we chose, using examples to illustrate their properties.

In each section, we explain the design of function values in the languages being presented and the reasoning behind that design, showing some examples that highlight the properties we're talking about.

Section 5.1 presents the examples we will use across all languages in Scheme, our lingua franca. Then, section 5.2 presents Lua; Section 5.3 presents Go; Section 5.4 presents Python; Section 5.5 presents Ruby; Section 5.6 presents Java and C#; and, finally, Section 5.7 present C++ and Rust.

5.1
Starting Examples for Surveyed Languages

The following are the examples we are going to introduce each of the languages with. We are using Scheme as our lingua franca.

5.1.1
Example: Counter

Based on our motivational example at Section 2.3, our first example is the counter iterator defined in Listing 2.18:

```
(define (it-count)
  (define count 0)
  (lambda ()
    (set! count (+ count 1))
    count))
```

As simple as it looks, in this example, the programmer does not have to worry about *how* the function returned by `it-count` will store the `count` variable; also, the caller doesn't have to worry about implementation or interface details. This demands, from the language:

– First-class function values;

– Anonymous functions or some form of function nesting, and;
– Lexical scoping, with some way to refer to outer variables after the end of their scope.

After defining the counter constructor, we will create a counter and call it repeatedly.

```
(define c (it-count))
(display (c))  ; 1
(display (c))  ; 2
(display (c))  ; 3
```

Listing 5.1: Calling a counter created by `it-count`

5.1.2 Example: Split Counter

A variation of the counter example (Listing 5.2) returns two different functions from the constructor: an incrementer and an accessor function. This unties the access of the current value from the increment of the iterator.

```
(define (it-counter-split)
  (define count 0)
  ; Incrementer and accessor of count
  ((lambda () (set! count (+ count 1))) .   ; car
   (lambda () count))                       ; cdr
  )

(define counter (it-counter-split))
(define increment (car counter))
(define get (cdr counter))

(increment)
(increment)
(display (get) "\n") ; 2
```

Listing 5.2: Split counter.

The first counter example does not necessarily enforce the sharing of the counter variable in multiple places. This variation is then useful as a stronger demonstration that the lexical scoping in the language makes it possible for two separate functions to share a reference to the same variable even after the scope of this variable ends. In the original counter example, we could, for example, move the variable inside the counter without sharing references to it with anyone else.

5.2 Lua

Lua is an imperative dynamic language. Even though it is a procedural language with a greater focus on control structures than Scheme, the functions have the same properties: all function values are first-class, every function is anonymous and nesteable, and it has proper closures.

5.2.1 Example: Counter

Despite the more imperative syntax, functions in Lua are sematically very close to Scheme's. The counter example makes this clear because it is a word by word translation of the Scheme version:

```
local function mk_counter()
    local count = 0
    return function()
        count = count + 1
        return count
    end
end

local c = mk_counter()
print(c()) -- 1
print(c()) -- 2
print(c()) -- 3
```

The `mk_counter` function creates a `count` variable, a number, and returns an anonymous function that increments `count` and then returns it. The `local` keyword defines a new lexically scoped variable, like Scheme's `define`. It's also worth mentioning that the declaration of `mk_counter` is pure syntactic sugar: declaring a "named" function like that is equivalent to assigning an anonymous function to a variable. More specifically, that declaration would be equivalent to the following[1]:

```
local mk_counter
mk_counter = function()
    -- ...
end
```

[1]Declaring the variable and *then* assigning the function to it (instead of declaring and assigning at the same time, like we're doing with `count`) ensures that `mk_counter` is lexically available inside the function body, which allows recursion.

The split counter example is almost a direct translation of the Scheme version as well. The only relevant difference is that while the Scheme version returns a pair, in Lua, we can return the incrementer and the accessor directly, since Lua functions support multiple returns:

```
local function mk_counter()
    local count = 0
    return function() count = count + 1 end,
           function() return count end
end

local increment, get = mk_counter()
increment()
increment()
print(get()) -- 2
```

Now `mk_counter` returns two functions referring to the same counter: one increments the value and the other returns it. Lexical scoping makes it clear that both functions are referring to the same `count` and garbage collection ensures that `count` remains accessible for as long as both functions are alive.

Lua is an example of a programming language that saw the power of functions as values—which is why it is very similar to Scheme in that regard, despite having a syntax that emphasizes procedural programming. In Listing 5.3 we show that Lua iterators are functions, just like our iterators from Section 2.3, and that built-in structures take advantage of that. With a slight modification to `mk_counter`, so that it has a stopping point, we turn it into `count_to`, which can actually be used in the built-in `for` statement to print all the numbers the iterator returns. All the other iterators in Lua, like `pairs` and `ipairs`, are iterators that work in the same way as this example. This shows that even non-functional programming styles can take advantage of functions as values.

```
local function count_to(n)
    local count = 0
    return function()
        if count < n then
            count = count + 1
            return count
        end
    end
end

for i in count_to(10) do
    print(i)  -- prints 1 to 10
end
```

Listing 5.3: Lua iterators are functions.

5.3 Go

Go is a statically-typed, compiled programming language. Despite not being as dynamic as Scheme, functions in Go have the same properties Scheme functions do, illustrating that more static languages can still have first-class functions with closures.

Terms The only relevant difference between Go's terminology and ours is that the Go language uses *function literal* to refer to an anonymous function.

5.3.1 Example: Counter

The counter example in Go looks very similar to the original example. The differences are the type annotations and the fact that, like C, any Go program has to have a `main` function and define a `main` package.

```
package main
import "fmt"

func mkCounter() func() int {
    count := 0
    return func() int {
        count++
        return count
    }
}

func main() {
    c := mkCounter()
    fmt.Println(c()) // 1
    fmt.Println(c()) // 2
    fmt.Println(c()) // 3
}
```

Listing 5.4: Counter example in Go.

In Go, the type of a variable (and the return type of a function) goes after it. The `mkCounter` function returns a `func() int`, which is a function that takes no parameters and returns an integer. We declare `count` using the `:=` operator, which is a shorthand for a variable declaration followed by an assignment, with automatic inference of the variable type:

```
    var count int;
    count = 0;
```

The split counter example is also straightforward (Listing 5.5).

```
func mkCounter() (func(), func() int) {
    count := 0
    return func() { count++; },
           func() int { return count; }
}

func main() {
    increment, get := mkCounter()
    increment()
    increment()
    fmt.Println(get()) // 2
}
```

Listing 5.5: Split counter.

Go supports multiple returns. To denote that a function returns multiple values, a pair type can be set as the return type—in this case, `(func(), func() int)`, meaning "a pair where the first element is a function that takes and returns nothing and the second, a function that returns an `int`, taking no arguments".

5.3.2 Function Definitions versus Anonymous Functions

In the previous examples, we saw function definitions and anonymous functions. Both of these generate function values and have a very similar syntax—they both use the `func` keyword, for example. But, unlike in Lua, top-level function declarations are not merely syntax sugar for declaring a variable and then assigning a function to it.

It's not possible to use the function declaration syntax inside another function. Were it syntax sugar, the example in Listing 5.6 would be valid. But, at the moment of writing, the use of nested function declarations raises a compiler error, even though there is no explicit mention of that in the manual. It's also not possible to have a variable assignment at the top level because the only statements that are valid in the top level are constants, types and function declarations—anonymous functions are none of these.

Aside from this syntatic nuisance, functions in Go are still first-class values because the function values generated by both function definitions and anonymous functions don't have any difference from the point of view of

```
func mkCounter() func() int {
    count := 0
    func counter() int {
        count++
        return count
    }
    return counter
}
```

Listing 5.6: Invalid nested function code in Go

the programmer. In Listing 5.7, the `compose` function takes two functions as arguments. From the point of view of `compose`, there is no distinction between taking a named function an taking an anonymous one.

```
func compose(
  f func(float64) float64,
  g func(float64) float64) func(float64) float64 {

  return func(x float64) float64 { return f(g(x)) }
}

func main() {
  fmt.Println(
    compose(
      math.Sqrt,
      func(x float64) float64 {return x * x})(10))
}
```

Listing 5.7: `compose` doesn't care if the functions it gets are anonymous or named

5.4 Python

Python is a dynamic, imperative programming language with first-class function values and closures. Despite being a dynamic programming language, Python does not emphasize the employment of functions as values like Scheme does, which leads to some shortcomings regarding scope and anonymous functions.

5.4.1 Limited Anonymous Functions

Anonymous functions in Python (called *lambdas*) are denoted with the `lambda` keyword and have only a single expression in their body:

```
f = lambda x: x * x
print(f(10)) # 100
```

While they are first-class function values, their use is very limited due to their body being only one expression. This limitation means that we cannot use an anonymous function to implement the counter example: the counter function needs to increment an external variable, but the increment operation is a statement, not an expression, so it's invalid syntax to increment a variable inside an anonymous function.

Due to those limitations, even the Python developers recommend avoiding anonymous functions for anything other than very simple expressions:

> (...) my usual course is to avoid using `lambda`.
>
> One reason for my preference is that `lambda` is quite limited in the functions it can define. The result has to be computable as a single expression, which means you can't have multiway `if... elif... else` comparisons or `try... except` statements. If you try to do too much in a `lambda` statement, you'll end up with an overly complicated expression that's hard to read. (Guido van Rossum and Python development team (27), p. 19)

5.4.2 Scoping Problems

While anonymous functions are very limited in Python, regular function definitions can still be nested and have closures, so the counter example would still be quite similar to the original version in Scheme, only with a named function rather than an anonymous one.

```
def mk_counter():
  count = 0
  def counter():
    count += 1
    return count
  return counter
```

Listing 5.8: Naïve attempt to implement counter

In Listing 5.8, the program has valid syntax, but Python will complain that the local variable `count` is referenced before assignment.

To understand where this problem is coming from, let us look again at the Scheme version of the counter factory example (Listing 5.1.1). In it, `count` is declared with `define` then assigned inside the counter function with `set!`. In Scheme, *declaration* and *assignment* have separate syntaxes. But Python chose to make the assignment operator take the role of both assignment and declaration, so only one syntax for these two operations is available. Python chose to make undeclared variables (i.e. previously unassigned variables) local by default[2], so the first assignment you make to a variable in a scope is syntactically equivalent to a variable declaration.

The increment of the `count` variable is equivalent to:

```
count = count + 1
```

This means that, in Listing 5.8, when we increment `count` inside of `counter`, we're actually telling Python to declare a *new* variable called `count`, local to `counter` with the value of `count` plus one—i.e. we're creating a new variable and initializing it with a variable that does not exist yet.

To solve that problem, Python 3 introduced the `nonlocal` keyword. It tells Python that one or more variables in the body of a function are actually references to outer variables. Using `nonlocal`, we can tell Python that the `count` variable inside `counter` is a reference to the an outer variable, not a new variable declaration (Listing 5.9).

Before Python 3, `nonlocal` was absent, so there were other ways to accomplish the ability to access (and modify) outer variables, but those were either some workaround to allow the modification of the external value without the assignment operation or the use of classes instead(28)—even the creator of

[2]This is not the only approach when there's no distinction between assignment and declaration: Ruby, for example, chose to declare a new variable by default, except if there's an outer variable of the same name

```
def mk_counter():
  count = 0
  def counter():
    nonlocal count
    count += 1
    return count
  return counter

c = mk_counter()
print(c()) # 1
print(c()) # 2
print(c()) # 3
```

Listing 5.9: `nonlocal` tells that `count` refers to the outer count

the language originally didn't see much reason to care about functions referring to nested scopes. Despite the adoption of a feature that makes the use of function variable more natural, the fact that it only appeared in Python 3 shows that the language does not emphasize the use of function values.

Still, even though the use of `nonlocal` might be unintuitive at first, the code at Listing 5.9 is pretty similar to the original example and functions in Python are still as capable as functions in Scheme. Implementing the split counter example in Python (Listing 5.10) confirms this: it is very close to the Scheme implementation of the same example.

```
def mk_counter():
    count = 0
    def increment():
        nonlocal count
        count += 1

    return increment, lambda: count

increment, get = mk_counter()
increment()
increment()
print(get()) # 2
```

Listing 5.10: We still need to implement the `increment` function as a nested function, but the getter can be an anonymous function

While Python does have shortcomings in regard to employing functions as values that demonstrate this is not a primary concern for Python developers, these shortcomings have been mitigated in more recent versions.

5.5 Ruby

Ruby is a dynamic object-oriented programming language that is inspired by Smalltalk's model of object orientation, i.e. everything is an object, every operation is a message send to an object. Given its Smalltalk roots, Ruby has a very particular approach to functions when compared to the other programming languages in this survey.

To be able to create a Ruby version of the counter example, we need to, first, understand Ruby's function-like primitives. So we decided to explain them first, then leave the examples for last.

5.5.1 Functions in Ruby: Methods, Blocks, Procs

Ruby has three kinds of function-like primitives: methods, blocks, and procs. A method is a kind of function that is associated to an object; a block is a kind of anonymous function that can be passed to method calls.

Methods are defined with the `def` keyword followed by the function name and the parameters in a parens-delimited, comma-separated list; blocks are defined with the `do` keyword, followed by the parameters in a comma-separated list delimited by vertical bars; for both of these, the body of the function is a sequence of statements where the last expression is the return value.

```ruby
# method
def cube(n)
  n * n * n
end

puts cube(10) # 100

# using block
[1, 2, 3].each do |n|
  puts cube(n) # 1, 8, 27
end
```

Listing 5.11: Methods and blocks in Ruby

Neither methods nor blocks are first-class values: the programmer cannot access their values, only call them. If, for example, we want to pass the `cube` method from Listing 5.11 as an argument to another method, passing just `cube` would be a syntax error: any direct mention to a method has to be a call. Similarly, trying to directly return a block from a method is not valid syntax, because blocks are part of the method call syntax and cannot be denoted independently—the only way to call a method directly is to use the `yield` keyword inside the receiving method (Listing 5.13).

```
def mk_counter
  count = 0
  do
    count += 1
    count
  end
end
```

Listing 5.12: This code is invalid because blocks are tied to a method call

```
def use_block
  puts 'Hello'
  yield
  puts 'Goodbye'
end

use_block do
  puts "It's me!"
end
```

Listing 5.13: Immediately using a block inside a method (prints "Hello", then "It's me", and finally, "Goodbye")

To pass methods and blocks around as values, they have to be converted into procs. A proc is an instance of the `Proc` class, and is a value that can hold any function in Ruby. So, even though methods and blocks are not first-class function values, they can be converted into procs, which are first-class.

To convert a method to a proc, there is the `method` method: it takes the name of the method as a symbol and retuns a proc holding that method.

```
def foo(a)
  puts a
end

m = method(:foo)
m.call('bar') # bar
```

Listing 5.14: `method` turns a method in a proc

As for blocks, there are several ways to convert them to procs: there are methods that, given a block, create a proc (`lambda`, `Proc.new`, etc.) and there are also named block parameters, which are method parameters that take a block passed in a method call and transform it into a value.

```
l = lambda do |x|
  x + 1
end

p1 = Proc.new do |x|
  x + 2
end

p2 = proc do |x|
  x + 3
end

def bla(&foo)
   foo
end

p3 = bla do |x|
  x + 4
end
```

Listing 5.15: Several different ways to turn blocks into values in Ruby (`l`, `p1`, `p2`, and `p3` are all instances of `Proc`)

Summary In the remainder of the text, we are going to call methods as functions, blocks as anonymous functions and procs as function values. Summarizing functions in Ruby: the language has named and anonymouns functions. None of the two are first-class function values, but they can be converted into function values in many different ways.

5.5.2 Example: Counter

Now that we discussed Ruby's functional primitives, we can create the Ruby version of the counter example. To do that, we have to create a function that declares the count value of the counter, creates an anonymous function that refers to that value and returns the function. In Listing 5.16, we use the `lambda` function.

Let us remember that `lambda` is not primitive: it is just a function that, given an anonymous function, constructs a function value. Also notice that

```
def mk_counter
  count = 0
  lambda do
    count += 1
    count
  end
end

c = mk_counter
puts c.call # 1
puts c.call # 2
puts c.call # 3
```

Listing 5.16: Counter factory using a lambda

lexical scoping works just as expected: we can seamlessly refer to `count` in the body of the anonymous function.

To further illustrate the capabilities of Ruby's functions, let us create the split counter example in Ruby. We need to return two anonymous functions, both referring to the `count` variable. All we need to do is, again, convert the anonymous functions to function values using `lambda` (Listing 5.17). Despite this detail, this example is very similar to the original one in Scheme.

```
def mk_counter
  count = 0
  [lambda do
     count += 1
     nil
   end,
   lambda do
     count
   end]
end

increment, get = mk_counter
increment.call
increment.call
puts get.call # 2
```

Listing 5.17: Split counter

This confirms that even though Ruby has an unusual design of functions, it has the same capabilities and properties as Scheme, only split in more primitive elements.

5.6 Java and C#

Java and C# are two object-oriented programming languages. All functions are necessarily defined inside a class and are either tied to the objects which are instances of that class or to the class itself. These functions are called *methods*; while Java and C# are not the only languages that have methods, methods are the only kind of function they have—besides anonymous functions.

In Java, before version 8, methods were the only kind of functions present. In C#, anonymous functions appeared in their current form in version 3.0[3]. The introduction of anonymous functions by both of these languages directly contrasts with the notion of function definitions being tied to a class, so their designers had to find a way to fit them into the language design.

Since Java and C# are very similar, we are using Java as the main language in this section. The presentation of C# will focus only on the differences between these languages.

5.6.1 Java

Java is a class-based object-oriented programming language. Any function definition has to be directly associated with a class. Functions defined in a class are called *methods* and are associated with the instances of that class—except when they're are marked `static`, in which case they're tied to the class and are called *static methods*. Java's anonymous functions are called *lambda expressions*.

Since version 8, Java has anonymous functions. Their syntax has two forms: one containing a full function body and an abbreviated form with a single statement (Listing 5.18).

In our counter example (Subsection 5.1.1), we create a function that returns an anonymous function. We cannot do that in Java for two main reasons. First, Java does not have a proper function type. Anonymous functions have to be converted to an object before they can be used in any way. Second, Java does not have closures and relies on copying outer values when referencing them inside an anonymous function. Before we implement a Java version of

[3]Anonymous Functions (C# Programming Guide): `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/anonymous-functions`

```
// anonymous function
n -> {
  if (n == 0) {
    return 0;
  }
  return 1/n;
}

// abbreviated anonymous function
x -> x * x * x
```

Listing 5.18: The two forms of anonymous functions in Java

the counter, we need to understand the reasons behind these limitations and how to work around them.

5.6.1.1 Fitting Function Values in a Strictly Object-Oriented Language

Anonymous functions in Java are not first-class values. They cannot be called directly and can only appear in an assignment, as the argument of a function call or in a conversion context—like a type cast or the return statement. To be able to use an anonymous function in any way, the programmer needs to convert it to a *functional interface*, which is an interface with a single function in it. An interface is a type that defines a blueprint containing a set of functions; a class is said to *implement* the interface when it implements all the functions present in it.

```
class HelloAnon {
    interface Greeter {
        public void greet(String name);
    }

    public static void main(String args[]) {
        Greeter g =
            name ->
            System.out.println("Hello, " + name);
        g.greet("Romário"); // Hello, Romário
    }
}
```

Listing 5.19: The anonymous function being attributed to `g` can only be called if we convert it to a functional interface—in this case, `Greeter`

The necessity of this conversion has to do with the fact that Java is object-oriented. Aside from primitive values, all Java values are an instance of some class. So, to create something that resembles a function value without violating this principle, instead of making a new, distinct function type, Java chose to convert the anonymous function to a functional interface by creating a class that implements the interface, using the body of the lambda as the body of the single method in that class.

Anonymous Classes Before having anonymous functions, Java already had *anonymous classes*, which are analogous, but define classes instead of functions. Like anonymous functions, they can only be created if a compatible interface type is specified. For example, in Listing 5.19, we attribute an anonymous function to an instance of `Greeter`, an interface with a single function called `greet`. Using an anonymous class with this same single function (Listing 5.20) has the exact same effect.

```
Greeter g =
    new Greeter() {
        public void greet(String name) {
            System.out.println(
                "Hello, " + name);
        }
    };
g.greet("Romário"); // Hello, Romário
```

Listing 5.20: Using an anonymous class has the same effect of using an anonymous function

This illustrates how the implementation of anonymous functions took advantage of the already existing anonymous classes and shows that anonymous functions are just syntatic sugar for anonymous classes with a single method.

5.6.1.2 Emulating Closures in Java

In Java, anonymous functions are only able to refer to outer variables that are *final*—that is, constant. This is because anonymous functions in Java capture outer variables by value, not by reference. Allowing the modification of a copied value inside an anonymous function would be misleading in this case, because, in most of the languages presented thus far (including Scheme), anonymous functions capture a *reference* to the outer variables they use; modifying a variable inside an anonymous function should also change it outside of the function. In Java, if it was possible to reference non-final variables, the modifications done inside an anonymous function wouldn't be replicated outside of it; an user expecting that behavior would have their program behave in an unexpected way without any warnings from the compiler.

However, aside from variables of primitive types, any variable in Java is a reference to an instance of some class. When an anonymous function refers to a reference variable, what needs to be constant is the reference itself, not the referred object. If the object functions that change it or public variables, an anonymous function can change this object by accessing these functions or public variables. Therefore, not having proper closures is a bigger problem for primitive values; since most variables in Java are not of a primitive type, this limitation is not as impactful as it seems.

5.6.1.3
Example: Counter

In the counter example, we need the anonymous function to modify a primitive value. While Java cannot modify a variable of a primitive type directly, we can "box" inside a class, so that the anonymous function actually captures a reference to an object of that class, allowing us to modify the primitive value the referred object contains.

In Listing 5.21, we create a `mkCounter` function that returns an instance of the `Counter` interface. Inside `mkCounter` we create the `CountData` class to box the `count` variable, then create a new instance of `CountData` called `data`. We return an anonymous functions that can, each time it is called, increment the `count` variable by referring to the data variable; since we are not modifying the `data` reference itself, we are not violating any language rules. Thus, the example works as expected.

Another possible approach for this problem is to use an anonymous class instead of an anonymous function. We already have the `Counter` interface defined, so all we need to do is to return an anonymous class that implements `Counter`. On top of that, since an anonymous class is a fully fledged class, we can merge the `CountData` class from Listing 5.21 and return just the anonymous class. In Listing 5.22 we can see that this approach leads to much less boilerplate in comparison to using an anonymous function. It is clear that the limitations of anonymous functions made them inadequate for this problem.

```
class CounterProgram {
    interface Counter {
        public int count();
    }

    static Counter mkCounter() {
        class CountData {
            int count = 0;
        }

        CountData data = new CountData();
        return () -> data.count++;
    }

    public static void main(String args[]) {
        Counter c = mkCounter();
        System.out.println(c.count()); // 0
        System.out.println(c.count()); // 1
        System.out.println(c.count()); // 2
    }
}
```

Listing 5.21: To be able to modify the `count` variable, we put it inside the nested `CountData` class, so that the function captures a reference instead of the primitive `count` variable

5.6.2 C#

C# is also a class-based object-oriented programming language. It is very similar to Java, so we are not going to get into a lot of detail about it. Instead, we are just going to highlight what C# does differently from Java.

5.6.2.1 Example: Counter

C# solves the two problems that make the counter example cumbersome in Java: it introduces proper closures and dedicated function-like types called *delegates*.

In Listing 5.23, our `MakeCounter` function returns an anonymous function that refers to the `count` variable. We don't need to box it like in Java: anonymous functions in C# are able to modify any external variable. Another improvement is that `Counter` here looks more like a function type than the equivalent interface in Java, since the declaration of a delegate only defines its return type and the parameters it takes.

```
    static Counter mkCounter() {
        return new Counter() {
            int c = 0;

            public int count() {
                return c++;
            }
        };
    }
```

Listing 5.22: In comparison to Listing 5.21, much less boilerplate is necessary if we use an anonymous class

But despite delegates making this less obvious, anonymous functions in C# are not function values. To use an anonymous function in any way, the programmer still needs to convert it to a delegate (Listing 5.24). This is very similar to Java, where the programmer needs to convert the anonymous function into an instance of a functional interface.

Even though C# tried to make a better design of anonymous functions than Java, due to the nature of the languages, they feel into the same design decisions to make anonymous functions compatible with object-orientation.

```
using System;

delegate int Counter();

class Hello {
    static Counter MakeCounter() {
        int count = 0;
        return () => count++;
    }

    public static void Main() {
        var c = MakeCounter();
        Console.WriteLine(c()); // 0
        Console.WriteLine(c()); // 1
        Console.WriteLine(c()); // 2
    }
}
```

Listing 5.23: In C#, we can modify outer variables from inside an anonymous function and we now define a delegate (`Counter`) instead of an interface

```
using System;

class TryToCallAnon {
    public static void Main(string[] args) {
        Console.WriteLine((x => x * x * x)(10));
        var cubederiv = x => 3 * x * x;
        Console.WriteLine(cubederiv(10));
    }
}
```

Listing 5.24: This code is invalid because anonymous functions need to be converted to a concrete delegate type

5.7 C++ and Rust

All the presented languages up until this point have garbage collection, so the lifetime of captured variables is automatically managed and it does not concern the programmer. In this section, we present C++ and Rust, two languages that, due to their focus on performance and control, do not have garbage collection. However, they have anonymous functions with closures. We will present how each of these languages accomplishes that.

5.7.1 Languages Presentation

C++ is a programming language that focuses on performance, emphasizing low-cost abstractions and avoiding overhead that wasn't explicitly asked by the user—"what you don't use, you don't pay" (29). It introduced anonymous functions in its C++11 revision, which effectively added support for nested functions with lexical scoping—before that, functions were first-class values, like in C, but they couldn't be nested.

Rust is another language focused on performance and low-cost abstractions, following the similar principles to C++ in this regard, but it also emphasizes the safety and correctness of the programs, as well as concurrency. It introduces mechanisms that allow the compiler to statically ensure that a variable will not be accessed after its lifetime ends. These checks are applied to the outer variables accessed by an anonymous function as well.

Their design of anonymous functions are different, but one important aspect they have in common is giving the programmer control over how the outer variables referenced by the functions are captured.

5.7.2 C++

5.7.2.1 Capturing Variables

In garbage collected programming languages, the capture of the actual reference to an outer variable by a function's closure is transparent to the programmer. Since memory management is not visible (or is an advanced feature), the programmer doesn't need to care where in the memory a variable is and how it is going to be disposed of.

In C++, on the other hand, memory has to be explicitly handled. For this reason, making the capture of variables entirely transparent would go against that, since each variable in C++ might require a different kind of handling depending on when it's used and who owns what. The concept of *ownership* of resources is key when managing memory in C++, so the programmer needs to be able to determine who owns what. This is why, in C++ anonymous functions, the programmer can explicitly determine how variables are going to be captured.

Every anonymous function in C++ has a *capture list* which determines how the variables referenced by the function are going to be captured. The capture list is a bracket-delimited list containing the variables that are going to be captured and how they are going to be captured; optionally, it is also possible to determine a *default capture* that determines how any variable referenced by the function is going to be captured. Listing 5.25 shows an anonymous function capturing an external variable by reference.

Some examples of capture lists:

– `[=]` captures all variables by value
– `[&]` captures all variables by reference
– `[foo, &bar]` captures `foo` by value and `bar` by reference
– `[=, &foo]` captures all variables by value except for `foo`
– `[&, bar]` captures all variables by reference except for `bar`

```
#include <iostream>

int main() {
    int c = 0;
    auto c_plus = [&](int x) { c += x; };

    c_plus(5); c_plus(2);
    std::cout << c << "\n"; // 7
}
```

Listing 5.25: The capture list of `c_plus` determines that `c` is capture by reference

5.7.2.2
Example: Counter

To create the counter example in C++ using anonymous functions, the counter function needs to capture an outer count variable and keep it after its scope ends. One way to do that is to, inside the counter constructor, allocate the count number dynamically, using a reference-counted pointer to refer to it, then return an anonymous function that refers to the pointer and captures it by copy (Listing 5.26). This pointer is made in such a way that copying it ensures that the memory to which it points will remain alive until there are no longer references to it.

```
#include <iostream>
#include <memory>

auto mk_counter() {
    auto count = std::make_shared<int>(0);
    return [=](){ return (*count)++; };
}

int main() {
    auto c = mk_counter();
    std::cout << c() << "\n"; // 0
    std::cout << c() << "\n"; // 1
    std::cout << c() << "\n"; // 2
}
```

Listing 5.26: Making a reference-counted pointer ensures `count` will remain valid after `mk_count` ends

Since we do not share the count variable with anyone else in this example, another solution is to make `count` as an `int` variable (instead of a pointer) and copy it, capturing it by value. By default, variables captured by value are immutable but that can be changed by adding the `mutable` keyword to the anonymous function (Listing 5.27). The advantage is that the indirection of dealing with a pointer is gone and so is the overhead of memory management incurred by the reference counting.

```
auto mk_counter() {
    int count = 0;
    return [=]() mutable { return count++; };
}
```

Listing 5.27: It's possible to modify variables copied by value with the `mutable` keyword

For the split counter example, however, the shared pointer is necessary, because both functions need to point to the same count (Listing 5.28). Copying the pointer to both anonymous functions will ensure the value they point to will only be deleted after *both* functions no longer exist.

```
#include <iostream>
#include <memory>
#include <utility>

auto mk_counter() {
    auto count = std::make_shared<int>(0);
    return std::make_pair(
        [=]() { (*count)++; },
        [=]() { return *count;}
    );
}

int main() {
    auto c = mk_counter();
    c.first();
    c.first();
    std::cout << c.second() << "\n"; // 2
}
```

Listing 5.28: `std::make_shared` is now necessary because both the incrementer and the accessor function need to point to the same count variable

5.7.2.3 The Type of an Anonymous Function

The `auto` keyword is used for type inference: when it is at the return type of a function, it means the return type will be deduced from the body of the function; when it appears at a variable declaration, it means the type of the variable will be deduced from the value of the right hand side expression.

In Listing 5.26, the `mk_counter` function is declared with `auto`. In `main`, the `c` variable where the counter function is stored is also declared with `auto`. This is not just for convenience. The only way we can make a function that returns an anonymous function or a variable that holds it is by the use of inference, because anonymous functions don't have an expressible type. While they do have a concrete type, the programmer is unable to explicitly express it in the code.

C++ inherited C's function types and still has function pointers, but these pointers only refer to the global function definitions; function pointers and anonymous functions are completely different kinds of function values.

If we need to explicitly express an anonymous function as an object of some type, we can use `std::function`. However, that's not the concrete type of an anonymous function: it is a wrapper that uniformizes many different function values into the same type and it's able to wrap anonymous functions as well as named functions. Such a wrapper incurs some non-negligible overhead, which might not be desired if, for example, the anonymous function is called several times.

```
#include <functional>
std::function<int()> mk_counter() {
    auto count = std::make_shared<int>(0);
    return [=](){ return (*count)++; };
}
```

Listing 5.29: Anonymous functions are convertible to `std::function`, but that's not their actual type

5.7.2.4 Function Objects

To understand why each anonymous function has its own unique type, we have to learn about function objects.

As stated before, C++ has function pointers. While these function pointers are first-class function values, there's no way to refer to non-global outer variables with function pointers, since functions cannot be nested—so something like the counter example would not be possible with them.

To allow "function-like" objects that can carry context data with them, C++ makes it possible for any class to overload `operator()` (read "operator call"). Functions that do that are called *function objects*[4] and it is possible to use the same call syntax used to call functions on them. We can even create a new version of our counter by creating a `CounterFunction` class that contains `count` as a variable and overloads `operator()` by making it increment `count` and return its new value (Listing 5.30).

This ties in with anonymous functions because their concrete type is exactly that: an anonymous, internal class with an overloaded `operator()`; the captured variables are turned into variables inside the class. What we do in the `mk_counter` function in Listing 5.26 is equivalent to defining a `CounterFunction` class inside `mk_counter` containing a reference-counted pointer and defining the `operator()` overload as the body of the anonymous function (Listing 5.31).

[4]They used to be called *functors* by the C++ community, until they realized this would confuse people with a background in functional programming.

```
#include <iostream>

class CounterFunction {
    int count = 0;

public:
    int operator()() { return count++; }
};

int main() {
    CounterFunction c;
    std::cout << c() << "\n"; // 0
    std::cout << c() << "\n"; // 1
    std::cout << c() << "\n"; // 2
}
```

Listing 5.30: An object of type `CounterFunction` can be called like a function

```
auto mk_counter() {
    auto count = std::make_shared<int>(0);

    class CounterFunction {
    public:
        std::shared_ptr<int> count;
        int operator()() { return (*count)++; }
    };

    return CounterFunction{count};
}
```

Listing 5.31: This example is sematically equivalent to Listing 5.26

This pattern of creating function objects to emulate closures is pretty common in the C++ community, so the C++11 standard just used it as the backbone of C++'s anonymous functions; this prevented the effort of trying to design something new and made anonymous functions compatible with existing code that expects function objects.

5.7.3 Rust

To recapitulate: Rust, like C++, is a language with a focus on performance and low-cost abstractions, has no garbage collection and it has anonymous functions with closures. Additionally, Rust has a strong focus on memory safety and concurrency, so it has features in its type system that prevent unsafe use of memory (that may lead to double-frees or leaks) and data races in compile time.

Terms Anonymous functions in Rust are called *closures*.

5.7.3.1 Borrow Checker

One of the most important features that prevents misuse of memory in Rust is the borrow checker, which is the mechanism that checks if all references to any variable are made when the variable is still alive.

By default, passing an argument to a function *moves* the value inside the function, making it no longer available after the function is done (Listing 5.32). To allow passing arguments to functions without losing them, Rust has two types of pointers: immutable (denoted by an `&`) and mutable (denoted by `& mut`). Passing a pointer to a function is called a *borrow* in Rust, because the idea is that a function borrows a reference to a variable, then gives it back when it's done (Listing 5.33).

```
fn hello(name: String) {
    println!("Hello, {}", name)
}

fn main() {
    let name = String::from("Romário");
    hello(name);
    // println!("{}", name);
    //                ^ 'name' no longer available
}
```

Listing 5.32: Values are moved by default in functions

The main rules are: there can be any number of immutable borrows, but only one mutable borrow at any given time; the borrower should not outlive the borrowed variable. If there's any violation of these rules, the borrow checker fails and there's a compilation error.

5.7.3.2 Capturing Variables

Like C++, Rust also allows the programmer to determine how a variable referenced in the body of an anonymous function will be captured, but anonymous functions do not have capture lists like in C++. Instead, there are two kinds of anonymous functions: *borrowing* and *moving*. They have the same

```
fn hello(name: &String) {
    println!("Hello, {}", &name)
}

fn change_name(name: &mut String) {
    name.push_str(" Rios")
}

fn main() {
    let mut name = String::from("Romário");
    change_name(&mut name);
    hello(&name);
}
```

Listing 5.33: Immutable and mutable borrows

syntax, but the difference is that moving anonymous functions are prefixed with `move`.

The concept of borrowing is relevant to anonymous functions in Rust because capturing outer variables has similar semantics to those of taking function arguments, so the borrow rules also apply to captures.

5.7.3.3 Example: Counter

```
fn mk_counter() -> impl FnMut() -> u64 {
    let mut count = 0u64;
    move || -> u64 {
        count += 1;
        count
    }
}

fn main() {
    let mut c = mk_counter();
    println!("{}", c()); // 1
    println!("{}", c()); // 2
    println!("{}", c()); // 3
}
```

Listing 5.34: `count` is moved inside the counter

In our counter example (Listing 5.34), we define a `count` variable and then return a moving anonymous function in the body of `mk_counter`. External variables will be captured by *moving* them inside the closure. A borrowing anonymous function would not work in this case because borrowing `count` would mean than the anonymous function would outlive it, violating the borrow checker (Listing 5.35).

```
fn mk_counter() -> impl FnMut() -> u64 {
    let mut count = 0u64;
    || -> u64 {     // may outlive 'count'
        count += 1;
        count
    }
}
```

Listing 5.35: The function being returned *borrows* `count`, but will outlive it, violating the borrow checker

5.7.3.4 Typing of Anonymous Functions

In Listing 5.34, the return type of `mk_counter` is `impl FnMut() -> u64`. `FnMut` is called a function *trait* and it encompasses all functions that modify the variables they take either by argument (borrowing) or by capture.

We have to use a trait (and not a concrete type) in the return value because, like in C++, anonymous functions in Rust do not have expressible types. The `impl` keyword means that `mk_counter` will return a concrete type that is of the trait `FnMut() -> u64`; this concrete type will be determined at compile time for the expression of the anonymous function being returned.

6
Conclusion

Seeing functions as regular values is a very powerful concept and allows many forms of abstractions. This power is what made some languages, like Lua (Section 5.2), Go (Section 5.3), and many others, to follow the focus on functions set by languages like Scheme. However, each programming language can follow a different paradigm, have different constraints and have some history that leads them to different choices regarding the design of functions.

This survey, as it contrasted a wide range of languages, drew many parallels between them that help justify the variety of different function designs among imperative programming languages. It's a demonstration that the design of functions in a programming language depends on the interaction of the features already present in it.

The Intersection of Roles between Closures and Objects The goal of the counter factory example (Section 2.3) is to have a way of constructing some value that, when called repeatedly, produces a sequence of numbers. To do that, such a value needs to keep its state, i.e. the value to be returned in the next call. In Scheme, we create a function that returns a nested anonymous function that, in turn, accesses an outer variable created in its same scope, taking advantage of closures to create a function with an associated state. But, in object-oriented languages, classes can take that role: each class has a set of functions and their associated state is an object of that class. This is evidenced by languages that support object-orientation had anonymous functions added later in their life, like Java (Section 5.6) and C++ (Section 5.7), used their previous existing class infrastructure to implement closures for anonymous functions. This could also be why programmers of object-oriented languagens didn't see the need for functions with proper closures: classes already cover that usecase.

Fitting Anonymous Functions into Later Versions of a Language Three of the languages studied in this survey, Java, C#, and C++ didn't have anonymous functions initially, since objects partially cover the roles anonymous functions would perform. They were a late addition and this means they had to accomodate the design of these entities to what was already in place, mainly because they need to avoid incompatibilities between anonymous functions, a new element, and all the other elements in these languages. That's why anonymous function values in Java are just an instance of an anonymous class, as well as in C++.

Closures vs. Garbage Collection From Section 5.1 to Section 5.6, all presented programming languages are garbage collected. It's clear that not having to worry about the lifetime of captured variables, making the capture process completely transparent, makes handling functions that capture external variables much more straightforward, as the programmer only needs to worry about passing the function value around—disposing of the closure is a concern of the language. That, however, doesn't mean that it's impossible to have closures without garbage collection, as we've seen in C++ and Rust in Section 5.7. The caveat is that now the capture of variables *is* a concern of the programmer, and that has to be taken into consideration while passing function values around. But while garbage collection is not mandatory for closures, some form of *automatic memory management* definitely is, in practice: all these languages have one or more forms of reference counted pointers, for example.

6.1 Future Work

In Section 4.2, it's implied that programming languages use different terms for the same feature. A big problem while talking about functions in programming language is which definition to use and what each term exactly means. For example: the term "closure" is used by many languages to denote what we refer to by "function value". We believe a survey on the terminology of programming languages, tracking points of divergence and the etymology of each term would be very relevant in clearing that up.

Another potentitally relevant venue of research is the history of functions in imperative programming languages and how they gradually adopted functional patterns over time. We touched on this in Chapter 3, especially in Section 3.1, where we discussed the timeline of a small number of languages, but the chapter only glosses over the history to discuss the general trend of growing acceptance of functional features in imperative languages. A more comprehensive look at the timeline of the evolution of functions is necessary to understand some questions that were left unanswered—for example, why the early 2010s were such a turning point for functions.

Bibliography

[1] JR., G. L. S.; SUSSMAN, G. J. ; OTHERS. **Revised Report on the Algorithmic Language Scheme**. Technical report, February 1998.

[2] HUDAK, P.; JONES, M. P.. **Haskell vs. ada vs. c++ vs. awk vs. ... an experiment in software prototyping productivity**. July 1994.

[3] CARLSON, W.; HUDAK, P. ; JONES, M.. **An experiment using haskell to prototype "geometric region servers" for navy command and control**. Research Report 1031, Nov. 1993.

[4] BACKUS, J. W.; BAUER, F. L.; GREEN, J.; KATZ, C.; MCCARTHY, J.; NAUR, P.; PERLIS, A. J.; RUTISHAUSER, H.; SAMELSON, K.; VAUQUOIS, B.; WEGSTEIN, J. H.; VAN WIJNGAARDEN, A. ; WOODGER, M.. **Revised report on the algorithmic language algol 60**. Technical report, International Federation for Information Processing, 1962.

[5] SEBESTA, R. W.. **Concepts of Programming Languages**. Pearson, 11th edition, 2016.

[6] WIRTH, N.. **The programming language pascal**. ACTA INFORMATICA, 1971.

[7] JENSEN, K.; WIRTH, N.. **Pascal, User Manual and Report**, volumen 18 de **Lecture Notes in Computer Science**. Springer-Verlag, 1974.

[8] KERNIGHAN, B. W.; RITCHIE, D. M.. **The C Programming Language**. Prentice Hall, 1968.

[9] STROUSTRUP, B.. **The C++ Programming Language**. Addison-Wesley, 1986.

[10] VENNERS, B.. **The making of python. a conversation with guido van rossum, part i (`https://www.artima.com/intv/pythonP.html`)**, 2003. Accessed in June 22nd, 2019.

[11] VENNERS, B.. **Python's design goals. a conversation with guido van rossum, part ii (`https://www.artima.com/intv/pyscaleP.html`)**, 2003. Accessed in June 22nd, 2019.

[12] VAN ROSSUM, G.. **Origins of python's "functional" features** (`https://python-history.blogspot.com/2009/04/origins-of-pythons-functional-features.html`), 2009. Accessed in July 3rd, 2019.

[13] ABELSON, H.; SUSSMAN, G. J.. **Structure and Interpretation of Computer Programs**. The MIT Press, 2nd edition, 1996.

[14] SUSSMAN, G. J.; JR., G. L. S.. **Scheme: an interpreter for extended lambda calculus**. Technical Report 349, Massachusetts Institute of Technology, Dec. 1975.

[15] SCOTT, M. L.. **Programming Language Pragmatics**. Morgan Kaufmann, second edition edition, 2006.

[16] WATT, D. A.. **Programming Language Design Concepts**. John Wiley & Sons, Ltd, 2004.

[17] SESTOFT, P.. **Programming Language Concepts**. Springer, 2012.

[18] LUA.ORG. **Lua 5.3 Reference Manual**, 2017.

[19] GOOGLE. **The Go Programming Language Specification** (`https://golang.org/ref/spec`), May 2018. Accessed in January 9th, 2019.

[20] VAN ROSSUM, G.; DEVELOPMENT TEAM, P.. **The Python Language Reference**, release 3.7.1 edition, Nov. 2018.

[21] BRITT, J.; OTHERS. **Help and documentation for the Ruby programming language** (`https://ruby-doc.org/`), 2018. Accessed in April 18th, 2019.

[22] GOSLING, J.; JOY, B.; STEELE, G.; BRACHA, G. ; BUCKLEY, A.. **The Java® Language Specification: Java SE 8 Edition**. Oracle America, Inc., Feb. 2015.

[23] (TG2), E. T. C. . T. T. G. .. **C# language specification**. Technical report, Ecma International, Dec. 2017.

[24] SMITH, R.. **Working draft, standard for programming language c++**. Technical report, ISO/IEC, March 2017.

[25] TEAM, T. R.. **The Rust Programming Language** (`https://doc.rust-lang.org/stable/book/2018-edition/`), 2018. Accessed in April 18th, 2019.

[26] TEAM, T. R.. **The Rust Reference** (`https://doc.rust-lang.org/reference/`), 2018. Accessed in April 18th, 2019.

[27] VAN ROSSUM, G.; DEVELOPMENT TEAM, P.. **Functional Programming HOWTO**, release 3.7.1 edition, Nov. 2018.

[28] YEE, K.-P.. **PEP 3104 – Access to Names in Outer Scopes**. Technical report, Python Software Foundation, Oct. 2006.

[29] STROUSTRUP, B.. **The design and evolution of C++**. ACM Press/Addison-Wesley Publishing Co., 1994.

[30] GHEZZI, C.; JAZAYERI, M.. **Programming Language Concepts**. John Wiley & Sons, third edition, 1996.