

Principles and Practice of Functional Programming

Mircea Marin

Viorel Negru

Isabela Drămnesc

Editura Universității de Vest

Timișoara, 2016

Contents

1	Introduction	1
1.1	Preliminary remarks	1
1.2	What is functional programming?	2
1.2.1	Other programming paradigms	4
1.2.2	Declarative versus procedural programming	7
1.3	Characteristics of functional programming	8
1.3.1	Values	8
1.3.2	Variables and environments	10
1.3.3	The structure of function values	14
1.3.4	The evaluation of function calls	14
1.3.5	Special forms	17
1.3.6	Feature 1: Higher-order functions	18
1.3.7	Feature 2: Purity and referential transparency	18
1.3.8	Feature 3: Repetition through recursion	19
1.3.9	Feature 4: Tail-call optimization and garbage collection	21
1.4	The lambda calculus	25
1.4.1	The language	25
1.4.2	Transformation rules	28
1.4.3	Reduction derivations	30
1.5	Programming in the λ -calculus	31
1.5.1	Programs and evaluation strategies	32
1.5.2	Multiple arguments	33
1.5.3	Abstract data types	34
1.5.4	Recursion	38
1.5.5	Enriching the λ -calculus	41
1.6	A history of functional programming	43
1.6.1	Lisp and its dialects	43
1.6.2	ISWIM	47

1.6.3	FP	48
1.6.4	ML	49
1.6.5	Lazy languages: SASL, KRC, and Miranda	50
1.6.6	Other languages	50
1.6.7	Haskell	50
1.6.8	Scala	51
1.7	Summary	52
2	RACKET and DRACKET	55
2.1	Interacting with RACKET	56
2.1.1	The parenthesised prefix notation	58
2.1.2	Identifiers and keywords	62
2.1.3	The definitions area	64
2.2	Block-structured programming	65
2.2.1	Main features	67
2.2.2	Block-structured programming in RACKET	67
2.3	Types of RACKET expressions	68
2.3.1	Values	69
2.3.2	Variables	69
2.3.3	Function calls	70
2.3.4	Special forms	76
2.3.5	Macros	76
2.3.6	Exercises	77
2.4	Primitive datatypes	78
2.4.1	Numbers and numerical functions	78
2.4.2	Symbols and keyword values	80
2.4.3	Booleans and true values	81
2.4.4	Other primitive data types	81
2.4.5	Exercises	82
2.5	Equality and identity tests	84
2.5.1	Referential identity: eq?	84
2.5.2	eqv?, equal?, and =	85
2.6	Conditional expressions and predicates	86
2.6.1	cond, if, when, and unless	86
2.6.2	and, or, and not	88
2.6.3	Exercises	89
2.7	Definitions and declarations	91
2.7.1	Scope and extent	91
2.7.2	let	95
2.7.3	let*	95

2.7.4	<code>letrec</code>	95
2.7.5	Exercises	96
2.8	Multiple return values	98
2.9	Input and output	100
2.9.1	Ports	100
2.9.2	Default ports	101
2.9.3	Reading operations	101
2.9.4	Writing operations	103
2.9.5	Varieties of ports	103
2.10	Imperative programming capabilities	105
2.10.1	<code>set!</code>	106
2.10.2	<code>set!-values</code>	107
2.11	Summary	107
2.12	Exercises	109
3	Composite Datatypes	113
3.1	Generalities	113
3.2	Pairs	115
3.3	Lists	121
3.4	Vectors	125
3.5	Hash tables	128
3.6	Quasiquoted values	132
3.7	Structures	134
3.7.1	The <code>struct</code> form	135
3.7.2	Structure subtypes	138
3.8	Summary	139
3.9	Exercises	140
4	Recursively Specified Data	143
4.1	Defining sets of values with Backus-Naur form	143
4.2	Proofs by structural induction	145
4.3	Structural recursion	146
4.3.1	Recursive functions with numeric arguments	149
4.4	Other patterns of recursion	150
4.4.1	Solving a more general problem	150
4.4.2	Bottom-up transformation of composite values	151
4.4.3	Top-down transformation of composite values	152
4.5	Guidelines for writing recursive definitions	153
4.6	Some interesting examples	154
4.6.1	Lists and related operations	154

4.6.2	Applicative programming with lists	157
4.6.3	Sets	161
4.6.4	Frames	163
4.6.5	Environments	164
4.6.6	Binary trees	165
4.6.7	Binary search trees	167
4.7	Summary	170
4.8	Exercises	170
4.8.1	Defining new datatypes and operations on them .	170
4.8.2	Recursive functions on lists. Applicative programming with lists	177
4.8.3	Recursive functions on numbers	188
5	Advanced Uses of Functions	191
5.1	The general form of function definitions	191
5.1.1	Variadic functions	191
5.1.2	Definitions of curried functions	193
5.2	Functions that take function arguments	194
5.2.1	Composition	194
5.2.2	Repeated function application	195
5.3	Functions that return functions	196
5.3.1	Functions with local state	196
5.3.2	Partial application and currying	200
5.4	Top-down and bottom-up design	203
5.4.1	Newton's root finding algorithm	203
5.4.2	The MapReduce programming model	205
5.5	Localized functionality	208
5.6	Object-oriented programming	211
5.6.1	Writing code in object-oriented style	211
5.7	Dynamic programming	216
5.7.1	Limiting the size of saved data	218
5.8	Summary	221
5.9	Exercises	222
6	Tail Recursion	225
6.1	Accumulator-style programming	226
6.1.1	Primitive recursive functions	229
6.1.2	Course-of-value recursion	230
6.2	Iteration	232
6.2.1	Repeating an action a number of times	233

6.2.2	Repeating an action for all elements of a list . . .	234
6.3	Limitations of tail recursion	235
6.4	Summary	235
6.5	Exercises	236
7	Modules	239
7.1	Basic notions	239
7.2	An illustrated example	240
7.3	The syntax of a module declaration	241
7.3.1	Submodules	243
7.3.2	Main submodules	245
8	Regular Expressions and Pattern Matching	247
8.1	Programming with regular expressions	247
8.1.1	Text strings	248
8.1.2	Regular expressions	248
8.1.3	Matching with regular expressions	250
8.1.4	Basic assertions	252
8.1.5	Characters and character classes	252
8.1.6	Quantifiers	254
8.1.7	Clusters	255
8.1.8	Alternation	257
8.1.9	Backtracking	258
8.1.10	Assertions for looking ahead and behind	259
8.2	Pattern matching	260
8.2.1	The <code>match</code> form	260
8.2.2	Other matching forms	262
8.3	Exercises	264
9	Programming with Streams	269
9.1	The representation of streams	269
9.1.1	Programming guidelines	272
9.2	Operations on streams	273
9.2.1	<code>s-cdr</code>	273
9.2.2	<code>s-filter</code>	274
9.2.3	<code>s-map</code>	274
9.2.4	<code>s-map2</code>	275
9.2.5	<code>s-merge</code>	275
9.2.6	<code>iterate</code>	276
9.3	Recursive streams and stream diagrams	277

9.3.1	Stream diagrams	281
9.4	Streams for infinite sets	282
9.4.1	The cartesian product $\mathbb{N} \times \mathbb{N}$	283
9.4.2	The set of Pythagorean triples	285
9.5	Applications	286
9.5.1	The stream of prime numbers	286
9.5.2	Generating random numbers	287
9.5.3	Generate-and-test methods	288
9.5.4	Symbolic manipulation of power series	290
9.6	Exercises	293
10	Macros	301
10.1	Pattern-based macros	302
10.1.1	Matching sequences	305
10.1.2	Identifier macros	307
10.1.3	Macro-generating macros	308
10.2	General macro transformers	308
10.2.1	Syntax objects	309
10.2.2	Macro transformers	310
10.3	Exercises	310
	Bibliography	313

1

Introduction

This chapter describes the main features of functional programming, compares it with other programming paradigms, indicates the core language of functional programming and its enrichments, and finishes with a historic account of the development of functional programming.

1.1 Preliminary remarks

The simplest way to describe computations is by expressions. For example, if we wish to compute the sum of areas of three circles with radii 2, 3 and 5, respectively, we would write the expression

$$3.14159265 \cdot 2^2 + 3.14159265 \cdot 3^2 + 3.14159265 \cdot 5^2 \quad (1.1)$$

and give it to a tool who knows how to compute its value. Such a tool could be an *interpreter*, and the computation of the value is called **evaluation**.

Expressions are combinations of values, variables, operators, functions, and special forms that the interpreter knows how to handle (according to its particular rules of precedence and of association). For example, to evaluate (1.1), we expect the interpreter to know the meaning of the constants 2, 3, 5, and 3.14159265, of the arithmetic operators for multiplication, addition, and raising to a power, and the fact that addition is a left-associative operation. The computation of the value would be a sequence of elementary reduction steps

$$\begin{aligned} & 3.14159265 \cdot 2^2 + 3.14159265 \cdot 3^2 + 3.14159265 \cdot 5^2 \\ \Rightarrow & \underline{3.14159265 \cdot 4} + 3.14159265 \cdot 3^2 + 3.14159265 \cdot 5^2 \\ \Rightarrow & 12.5663706 + 3.14159265 \cdot \underline{3^2} + 3.14159265 \cdot 5^2 \\ \Rightarrow & 12.5663706 + \underline{3.14159265 \cdot 9} + 3.14159265 \cdot 5^2 \end{aligned}$$

$$\begin{aligned} &\Rightarrow \underline{12.5663706} + \underline{28.27433385} + \underline{3.14159265} \cdot 5^2 \\ &\Rightarrow \underline{40.84069445} + \underline{3.14159265} \cdot \underline{5^2} \Rightarrow 40.84069445 + \underline{3.14159265} \cdot 25 \\ &\Rightarrow \underline{40.84069445} + \underline{28.14159265} \Rightarrow 68.9822871 \end{aligned}$$

The underlined expressions are those selected to be reduced, and are called *redexes*. The evaluation stops when all redexes have been reduced.

Expressions made only of values and predefined functions and operators can easily become cumbersome. One such example is expression (1.1), where we wasted effort to write several times similar functionality:

1. The numeric approximation 3.14159265 of the value of π is long and possibly hard to remember.
2. The code for the computation of the area of a circle occurs repeatedly, for different values of the radius.

This way of writing code is bad practice. The **Principle of Abstraction** states the following: “Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by *abstracting out* the varying parts.” [37]

Programming languages support this principle with various definitional mechanisms. A definitional mechanism allows to name objects of interest, and to refer to them later, whenever they are relevant for further computations. For example, a language like C++ allows to name values and functions, like:

```
float pi = 3.14159265;
float circleArea(float r) { return pi*r*r; }
```

and to replace expression (1.1) with the equivalent expression

$$\text{circleArea}(2) + \text{circleArea}(3) + \text{circleArea}(5) \quad (1.2)$$

As we shall see, functional programming is exactly about this: computing by evaluating expressions.

1.2 What is functional programming?

Functional programming is a **programming paradigm** which views every computation as the evaluation of an expression. Programs consist of naming constructs, mainly function definitions, which simplify the writing of expressions.

This is not the only way of viewing a computation: other paradigms propose other ways of thinking. In general, a programming paradigm is a style or “way” of programming that prescribes a number of programming constructs and how to use them in order to model and solve efficiently a wide range of problems.

To fully master the benefits of a programming paradigm, one should know not only the behaviour of its basic programming constructs but also the general principles that guide the thinking of a good programmer, and how to apply them wisely. This is similar to what characterizes a good chess player: In addition to knowing the basic rules to move the pieces, he must also know playing strategies with high chances to win.

Software engineering recognizes four major programming paradigms:

1. Imperative (or procedural) programming,
2. Object-oriented programming (OOP),
3. Functional programming (FP), and
4. Logic programming (LP).

With the exception of logic programming, all the other programming paradigms claim to be suitable for a very large range of applications. Every seasoned programmer or software engineer should know (1) the concepts and principles that characterize every programming paradigm, and (2) the programming languages that support them best. Based on this knowledge, he should be able to make the right choice of a programming language and programming style when faced with the implementation of a particular application.

Often, the best solution to a problem is a combination of different programming styles. Most programming languages are *multi-paradigm*: they have programming constructs that support well different ways of thinking. However, most programming languages are designed to support best a particular programming paradigm, and the separate study of every programming paradigm is a very important experience.

The rest of this section is organized as follows. First, we put things into perspective by giving a concise description of the other three major programming paradigms: imperative programming, object-oriented programming, and logic programming. Next, we indicate the characteristic features of functional programming. Afterwards, we describe the λ -calculus, a very simple computational model implemented by almost all functional programming languages. Finally, we conclude with a historic account of the development of the functional programming style.

1.2.1 Other programming paradigms

Imperative programming

Imperative programming views every computation as a sequence of commands that act on an implicit program state, until we reach a state containing the desired result. Historically, this is the first proposed programming paradigm: The notion of implicit state was inspired by hardware, to reflect the values of variables stored in the memory of the computer. The order of commands is important, and imperative programming languages typically provide programming abstractions for control flow (e.g., *block* constructs, conditional statements, and *while* loops) to permit a precise and deterministic control over the program state. This way of thinking is similar to descriptions of everyday routines, such as food recipes which prescribe the steps to be performed to achieve a desired result. The main programming abstraction is the *procedure*, which abstracts the actions of one or more commands, and can be called as a single command. For this reason, imperative programming is also called *procedural programming*.

For example, the pseudocode of an imperative implementation for the computation of the minimum element of a list L of n numbers L is

<pre>minList(L, n) $r \leftarrow L[0]$ $i \leftarrow 1$ while $i < n$ do $r \leftarrow \min(r, L[i])$ end while return r</pre>	<pre>min(a, b) if $a < b$ then $m \leftarrow a$ else $m \leftarrow b$ endif $i \leftarrow i + 1$ return m</pre>
---	--

The minimum is computed by the procedure call `minList( $L, n$ )`. The auxiliary procedure `min( $a, b$ )` groups together three commands:

1. a conditional statement whose overall effect is to store the minimum of a and b in the state variable m ,
2. an assignment which increments the value of i by 1, and
3. the command which returns the value of m .

Procedures versus mathematical functions.

The main difference between procedures and mathematical functions is the following:

- A **mathematical function** computes a result only from the values of its input arguments. It does not have any side effects on the implicit state of the program, such as: changing the value of a non-parameter variable, writing to a file, etc. Therefore, we can not distinguish different calls of the same mathematical function with the same input arguments: They always have the same behaviour. For this reason, we say that mathematical functions are **referentially transparent**.
- A **procedure** groups a set of commands whose actions may have side effects. Therefore, different calls of the same procedure with the same input arguments may have different effects.

An example of a procedure which is not a mathematical function is `min`: the procedure call `min(a, b)` computes the minimum of a and b , but it also has the side effect of incrementing the value of i by 1.

The imperative programming style is problematic because it is focused on simulating computations on a particular computer architecture instead of providing a computational model closer to human thinking. Efficiency is the primary concern, rather than the suitability of the language for software development.

Representative languages for imperative programming are Fortran [4], Algol, Pascal, Basic, and C.

Object-oriented programming

Object-oriented programming (OOP) advocates thinking in terms of *objects* communicating with each other by message passing. Objects consist of *data fields* (attributes that characterize their state) and associated procedures known as *methods* which characterize their computing capabilities. There are two main kinds of OOP languages: *class-based* (e.g., Simula, Smalltalk, C++ and Java) where objects are instances of classes, and *prototype-based* (e.g., JavaScript) where objects are clones of existing objects that serve as prototypes. Computations are performed by method calls addressed to objects. This is different from imperative programming: instead of changing an implicit state by executing com-

mands, OOP changes the states of objects by sending them messages consisting of method invocations. OOP can be viewed as an improvement of imperative programming with several new features, including:

- *Encapsulation*: The facility to bundle data with methods (or other functions) operating on that data.
- *Dynamic dispatch*: When a method is invoked on an object, the object itself determines what code gets executed by looking up the method at run time in a table associated with the object.
- *Inheritance*: Objects defined by classes can inherit fields and behaviour (that is, methods) from pre-existing classes, superclasses, or parent classes.

Logic Programming

Logic programming (LP) is very much different from the other three programming paradigms. It is based on the following way of thinking:

Answer questions by search according to a fixed, predictable strategy.

This way of thinking is useful for solving problems related to the extraction of knowledge from basic facts and relations. The programmer must describe what he knows as facts and rules collected in a program, and leave to the compiler or interpreter of the programming language the task of answering the questions that he may ask afterwards. LP is considered a declarative programming style because programming consists only in declaring what we know; *how* to answer arbitrary questions is predefined by the built-in search strategy of the language, e.g., SLD resolution in Prolog, which is a representative language for LP.

For example, a logical model for the computation of the minimum element of list of numeric values is given by the following facts and rules of inference:

- **Fact**: The minimum element of a singleton list made of number m is m .
- **Rules of inference**:
 1. The minimum of x and y is x if $x \leq y$.
 2. The minimum of x and y is y if $y < x$.

3. The minimum of a list starting with x, y followed by sublist t is m if m is the minimum of x and n , where n is the minimum of the list with first element y followed by sublist t .

If we consider the predicates $\text{min}(x, y, z)$ with the intended reading “the minimum of x and y is z ,” and $\text{minList}(L, n)$ with the intended reading “the minimum element of list L is n ,” then the previous facts and rules of inference can be written as follows in the language of first-order logic:

$$\forall x, y. (x \leq y \Rightarrow \text{min}(x, y, x)).$$

$$\forall x, y. (y < x \Rightarrow \text{min}(x, y, y)).$$

$$\forall m. \text{minList}([m], m).$$

$$\forall x, y, t, m, n. (\text{minList}([y|t], n) \wedge \text{min}(x, n, m) \Rightarrow \text{minList}([x, y|t], m)).$$

and as follows in Prolog:

```
min(X,Y,X) :- X =< Y.
```

```
min(X,Y,Y) :- Y < X.
```

```
minList([M],M).
```

```
minList([X,Y|T],M) :- minList([Y|T],N), min(X,N,M).
```

1.2.2 Declarative versus procedural programming

An interesting classification of programming paradigms is by looking how they separate the concerns of specifying **what we know** from **how to solve** a concrete problem.

The procedural programming paradigms require the programmer to specify *how* to solve a problem, by indicating the flow of control which modifies repeatedly a global program state (in imperative programming) or the local states of communicating objects (in OOP) until we reach a situation that yields the desired solution.

In contrast, the declarative programming paradigms require the programmer to focus on specifying *what* he knows, and leave the task of finding the desired answers or solutions to the implementation of a language. Typically, the declarative programming languages have a built-in strategy to make use of the knowledge encoded in programs.

LP and FP are declarative programming paradigms because:

- In logic programming, the programming activity resumes to specifying what we know, by encoding knowledge as rules and facts. Solving problems is reduced to finding answers to queries, and this

job is the business of the search engine of the language, which implements a predefined search strategy (e.g., SLDNF-resolution in Prolog) which is guaranteed to find all possible answers.

- In functional programming, the programming activity resumes to specifying what we know, by encoding knowledge as function definitions. Solving a problem is modelled as finding the value of an expression, and this job is the business of an evaluator that implements a predefined reduction strategy.

Although declarative programming seems to make programmers' life easier by relieving them from the details of how to compute the solutions, for efficiency reasons it is very important to understand the predefined strategy of the language. In particular, the performance of programs written in LP and FP can be greatly improved if we follow a few programming guidelines, such as writing tail-recursive specifications.

1.3 Characteristics of functional programming

As already mentioned, functional programming (FP) is based on the principle

Compute by evaluating expressions.

The evaluation of an expression e_0 to a value v can be viewed as a sequence of reduction steps $e_0 \Rightarrow e_1 \Rightarrow \dots \Rightarrow e_n = v$, where every step $e_i \Rightarrow e_{i+1}$ selects a subexpression of e_i , called a *redex*, and replaces it with a simpler one. When the redex coincides with e_i , we may write $e_i \rightarrow e_{i+1}$ instead of $e_i \Rightarrow e_{i+1}$.

The simplest expressions are combinations of *values* and *variables*. Before going any further, it is important to clarify what are values, what are variables, and what data structures are used to get their meaning.

1.3.1 Values

Every programming language has a predefined collection of datatypes to work with. Typical examples are integers, booleans, lists of integers, etc. The elements of a datatype are called values.

Most datatypes have literal representations of their values. For example, the literal representation of number two in Java is 2. There may be, however, datatypes with no literal representation for their values.

In functional programming, **all values are first-class objects**. This means that, regardless of the datatype, any value can be named, passed as input argument to a function call, returned as result of a function call, or stored as component of a compound value (e.g., as element of a list, set, or vector).

An outstanding feature of functional programming is that **functions are values**: they can be named, passed as arguments or returned as results of function calls, or stored as elements of compound values. Before explaining what this means, let's introduce some notation to describe functions. Back in 1928, Alonzo Church proposed the lambda notation $\lambda(x).e$ with the intended reading "the mathematical function which, for input argument x , returns the value of expression e ." A slight generalization of Church's notation is the expression

$$\lambda(x_1 \dots x_n).e$$

with the intended reading "the mathematical function which, for input arguments $x_1, \dots, x_n$, returns the value of expression e ."

EXAMPLE 1 Assume we wish to define the function `sum` which, for input arguments x and y , computes their sum. With this notation, we could define

`sum` = $\lambda(x, y).x + y$

Similarly, we could define

`prod` = $\lambda(x, y).x * y$

to refer to the function that multiplies the values of its input arguments. Since functions are values, we can store them in a compound value, such as a list. For example we can define the list of functions

`L` = [`sum`, `prod`]

NOTE: The functions for arithmetic operations (addition, multiplication, etc.) are predefined in almost all programming languages. Also, almost all functional programming languages have built-in support to work with lists. In this example we write $[v_1 \dots v_n]$ for the list with component values $v_1, \dots, v_n$, and assume the predefined operation `get(L, i)` returns the i -th element of list `L`.

Next, we could define a function `f(L, a, b)` which takes as input a list `L` of two functions and computes the list of values obtained by applying the functions from list `L` to arguments `a` and `b`: