PROGRAMMING LANGUAGES

Design, Semantics, and Implementation

Hridesh Rajan

Preface

Perhaps the most important goal of computer science is the study and application of computers to the representation of processes, both natural and artificial, as computational processes so that we can improve upon them for the betterment of society. Programming languages play a crucial role toward this goal. A suitable programming language can enable and aid the representation of computational processes, their efficiency, and their correctness. The selected programming language can also have a significant impact on productivity of developers.

The goal of this book is to help you understand key principles of programming languages – ideas that form the core of most programming languages used in practice. There are two styles for learning computer programming languages: survey-based and experience-based.

- In a survey-based learning style, students typically
  - read about a set of programming languages,
  - compare and contrast these languages, and
  - study variations, strengths, and weaknesses by analysis.
- In an experience-based learning style, a student would
  - implement a set of programming language features,
  - write programs using this programming language implementation, and
  - study variations, strengths, and weaknesses by experience.

This book follows the experience-based learning style. Main differences between the approach taken by this book and other experience-based approaches for teaching principles of programming languages are the following.

- In an experience-based approach, each concept discussed in the book is implemented in a prototype interpreter that is gradually developed. An interpreter is a computer program that consumes another computer program and produces its value. The programming language used to implement the interpreter is called the *defining language* and the programming language that is being implemented is called the *defined language*.

  This book uses the programming language Java as the defining language, i.e. all interpreters in this book are written in the Java programming language. Students using this book build several defined languages over the course of chapters, starting with a functional core language.

  The choice of defining language is intentional — to enable students to focus most of their efforts on realizing the concepts and to leverage upon the prevalance of Java as an introductory programming language in computing curricula.

  The choice of first defined language is also intentional — to enable students to become familiar with functional programming concepts, an extremely important tool for modern multi-paradigm programmers, along with other concepts such as reactive programming and logic programming.

- We cover a broader spectrum of emerging trends in programming languages e.g. concurrency, Big Data, reactive programming (than what is covered in similarly themed material in past).

- We also discuss semantic variations and tradeoffs, as appropriate. These variations are also prototyped in our interpreter.

- We assume very little background, which allows this book to be used in courses with fewer prerequisites.

This book can be used in both undergraduate and graduate courses. Since we assume very little background, it can be used in a course taught as early as the junior year in an undergraduate curriculum.

This book is organized as follows. In the formative chapters, we develop basic concepts in languages including means of computation using primitive values, means of combination such as variable definition and functions,

means of abstraction such as functions and recursive functions. We then look at imperative features such as references, concurrency features such as fork, and reactive features such as event handling. We then look at language features that cover important, and often different, perspectives of thinking about computation such as that in logic programming, flow-based programming. First six chapters should be read in a sequence. Afterward, a course can adopt one or more later chapters as they are developed in a fairly independent manner.

For adopters of early edition: Comments can be sent to the author at his e-mail `hridesh@iastate.edu`. We welcome any feedback that you may have on the material.

Book code: A language implementation for each chapter is available as part of the book code. We invite you to try out examples and look over concrete implementation as you read through the material.

Contents

Chapter 1

Introduction

This book provides an introduction to key concepts in design, semantics, and implementation of computer programming languages, here onwards referred to as programming languages or languages (if that use doesn't cause any ambiguity). It has two goals. First, to prepare you, the reader, to be able to design your own programming language. Second, to prepare you to select an appropriate programming language for a software development tasks out of hundreds of programming languages available today.

The material discussed in this book requires active participation. Each chapter is associated with a working implementation of a programming language and the reader is encouraged to obtain the code corresponding to the chapters and follow along. Just reading the book or reading the code may not be as fruitful.

1.1 Kinds of Programming Languages

There are two kinds of computer programming languages: general-purpose and domain-specific.

The goal of domain-specific programming languages is to provide specialized support for concepts in the domain. These concepts could be data types, relations, operations, etc. For example, the Dot language for Graphviz, a software for graph visualization[1] has support for nodes and edges that are basic graph concepts. Similarly the HTML language, a domain-specific language for web-pages has support for markup related con-

[1] http://www.graphviz.org/doc/info/lang.html

cepts[2]. A related language CSS is a domain-specific language for describing format and layout of web-pages[3]. The SQL language, a domain-specific language for querying databases has support for query, join, etc.

The goal of general-purpose computer programming language is be able to express all computation. Languages such as C, Fortran, LISP, Ada, Scheme, C++, Java, C#, Haskell[4], Scala[5], Swift[6], Clojure[7], Lua[8], Objective C[9], are examples of general-purpose languages, and so are Smalltalk[10], Perl[11], Python[12], Ruby[13], Javascript[14], Dart[15], Go[16], etc...

1.2 Our Goals

In this book, we will study fundamental building blocks that form the basis of both domain-specific and general-purpose languages. Our goal will be to prepare you to design, implement, analyze, and understand both kinds of programming languages.

Now, it is true that many readers will not go on to design their own general-purpose programming language but knowing about principles of programming language design, semantics and implementation is important for at least the following reasons.

1. As you can perhaps imagine from the list of programming languages above, new computer programming languages are continuously introduced to address new needs. As of this writing in Summer of 2015, Apple has recently introduced Swift whereas two new languages Go

[2]http://www.w3.org/html/
[3]http://www.w3.org/Style/CSS/
[4]http://www.haskell.org/
[5]http://www.scala-lang.org
[6]http://developer.apple.com/swift/
[7]http://clojure.org
[8]http://www.lua.org
[9]developer.apple.com/library/mac/navigation
[10]http://smalltalk.org/
[11]http://www.perl.org
[12]http://www.python.org
[13]http://www.ruby-lang.org
[14]ECMAScript Language Specification
[15]http://www.dartlang.org
[16]http://golang.org

and Dart are coming out of Google, the usage of Clojure and Scala is on the rise also, and Typescript and F# are coming out of work at Microsoft. So, it is common for computer software professionals to learn several programming languages during the course of their career. Knowing about principles of languages, and core ideas will enable you to more easily pickup a new language and be productive. You will be able to determine the concepts that transfer over, and those that are unique to the new language (and focus on mastering those new concepts).

2. Perhaps related to the first point, knowing about computer programming languages, perhaps more deeply than others who do not have similar training, gives you skills to be a more productive programmer.

3. Most important benefit may come from the usefulness of these skills toward your future domain-specific language (DSL) design and implementation. We believe that each reader will design a DSL in their lifetime to enhance their own productivity as well as of those in their organization. Knowledge of language design and implementation will greatly benefit such DSL design and implementation efforts.

4. Programming language design, data type design, and application programming interface (API) design are related activities. So skills acquired while studying principles of programming languages will benefit you during these other activities.

5. Knowing about programming language features and their tradeoffs will help you select appropriate computer programming language for the software development task at hand.

1.3 Parts of a Programming Language

According to Sussman and Abelson, programming language features can be broadly classified into the following three categories. An effective language should provide each of these means.

- *Means of computation* These programming language features enable basic computation, e.g. addition, subtraction, multiplication, divi-

sion, string concat, etc... In a nutshell, these features are atomic, indivisible instructions defined by the programming language.

- *Means of combination* These programming language features enable us to take two or more computations and combine them to form a larger piece of computation. Such a combination may introduce an order, or it may introduce a deterministic or non-determinstic choice, among other things. An example of means of combination is a sequence often written as ";" in C, C++, Java-like languages. A sequence combines computation to left and to the right into a larger computation. Another example is the conditional expression often written as "x > 0 ? y : z". A conditional combines three expressions – based on the result of the first expression either the first expression or the second expression is evaluated.

- *Means of abstraction* These programming language features enable us to create a proxy, a name, that can then be used to refer to a complex piece of computation. An example of means of abstraction is a function in mathematics, e.g. `f(x) = x`^{2} which gives us a name `f` that we can then use to refer to the computation that happens inside the function body. This function abstraction is also adopted in most computer programming languages.

In the next few chapters, we will gradually build a general-purpose programming language. We will begin this work by discussing some means of computation, support a number of them in our language. We will then gradually add means of combination and abstraction to arrive at a complete core by the end of the fifth chapter. This core can then be used to build your own domain-specific programming language or to enrich with more expressive extensions. At each step we provide a number of exercises that we hope will be helpful in solidifying your understanding of the material.

Exercise

1.3.1. Write down the name of programming languages that you have used or heard about. Try to identify key design goals for each of these languages.

1.3.2. Identify an application domain that is of interest to you. Search and list one or more domain-specific programming languages that are designed to make software development for that domain easier.

1.3.3. For your favorite programming language identify means of computation, means of combination, and means of abstraction.

1.3.4. For a recently released programming language, make a list of all features that the language makes available. Then, try to categorize these features into means of computation, means of combination, and means of abstraction.

1.4 Mathematical Concepts and Notations

Now, we introduce some concepts and notations that we will use throughout this book. A reader familiar with these concepts may also want to skim through this section to refresh their memory, and to make a note of notational differences, if any.

Functions

We will write functions as follows, where f is the name of the function, D is the *domain* of the function, and R is the *range* of the function.

```
f :  D -> R
```

Sets and Operations on Sets

We will use explicit enumeration to define a set, e.g. $\{a, b\}$ to define a set containing two elements a and b. We may also use comprehension to define a set, e.g. $\{a \mid a \in B\}$ to define a set that contains all elements a that are also in the set B. The notation $a \in B$ will be used to mean a is a member of (or element of) B. We will use the notation $\emptyset$ to refer to the set containing no elements. We will use the notation $A \subseteq B$ to mean that A is a subset of B. We will use the notation $A \setminus B$ to denote the difference of set A and B, which is a set that has elements from A that are not in B.

1.5 Java: A Brief Introduction

The rest of this book assumes familiarity with the Java programming language. If you haven't programmed in Java before, we would encourage you to refer to several excellent textbooks and tutorial material on this programming language. Below we provide a short introduction to the programming language. A reader familiar with Java may also want to skim through this section to refresh their memory about object-oriented terms.

Java is an object-oriented programming language, i.e. basic elements of a Java program are "objects". For example, a program may have objects such as an apple, a second apple, an orange, a boy, a girl, a bicycle, etc. An object can have properties. Apples and oranges could have weight, color, etc. Boys and girls could have the property age, name, etc. An object may also be capable of performing certain operations. For example, boys and girls may be able to eat an apple, ride a bicycle. There could be several objects in a program that are similar in nature in that they have the same set of properties, but not the same. For example, two apple objects could both have weight and color as properties with different values say '0.15 lb' and '0.17 lb' and 'red' and 'green'. An object-oriented programmer uses a 'class' as a template for objects that have the same set of properties. For example, one would write a class `Apple` in Java as follows.

```
1 class Apple {
2   double weight;
3   boolean edible = true;
4   boolean isEdible() {
5     return edible;
6   }
7 }
```

This form above is called a *class declaration*. The act of writing the code for a class is referred to as *declaring a class*. In the class declaration above, notice two statements about `weight` and `edible`. The statements on line 2 and 3 says that all 'Apple' objects will have a property 'weight' and a property 'edible'. These two statements are called *field declarations*. The two forms of field declarations are shown on line 2 and line 3. Both start with declaring that this field's legal values will be limited to those that can be taken by a `double` and a `boolean`. A boolean can only take two legal values: `true` and `false`. So, by stating `boolean edible`, this

field declaration says that this field edible must be either true or false. The rest of the field declaration on line 3 gives an initial value to the field; it says that each apple will start out with the field (property) edible as true. A class declaration can also include *method declaration* that represents actions that can be performed on an object of that class. For example, in the class declaration for `Apple` on lines 4-6 there is a method declaration `isEdible`. A method declaration has several parts: return value (what does the operation produce when it is done), name (what is it called), arguments (what does it consume). The declaration of the method `isEdible` says that it produces a value that is limited by a `boolean`, its name is 'isEdible' and it does not consume anything (denoted by "()"). We will soon see examples of method declaration that consume values as well.

Given the class declaration of `Apple`, we can create apple objects. An example appears below that creates two objects `apple1` and `apple2`.

```
1  Apple apple1 = new Apple();
2  Apple apple2 = new Apple();
```

To allow programmers to reuse code, object-oriented languages also allows a class to inherit all of the functionalities from some other class. For example, we could choose to write a more general class `Fruit` as follows.

```
1 class Fruit {
2  double weight;
3  boolean edible;
4  boolean isEdible() {
5    return edible ;
6  }
7 }
```

Notice that this class doesn't give an initial value to the field `edible` to support both edible and inedible fruits.

Then, the `Apple` class can inherit all of the functionalities from the `Fruit` class as follows.

```
1 class Apple extends Fruit {
2 }
```

This version of the `Apple` class also has two fields `weight` and `edible` and the method `isEdible` —even though it doesn't explicitly declare them— because it inherits those properties from the `Fruit` class. In object-oriented

terminology `Apple` will be called a *subclass*, and `Fruit` a *superclass*. A class in Java can have exactly one superclass.

This definition of `Apple` is not sufficient though, it doesn't define apples to be edible. We can fix that by defining a *constructor* that gives an initial value to the edible field as follows.

```
1 class Apple extends Fruit {
2  Apple () {
3     edible = true;
4  }
5 }
```

A constructor is a special kind of operation. It runs exactly once, when objects are being created. In Java constructors are operations that do not produce any value. In this case, the constructor for `Apple` changes the field `edible` so that apples are considered edible. The following listing defines another class to represent objects of `Wahoo`, which are not edible.

```
1 class Wahoo extends Fruit {
2  Wahoo () {
3     edible = false;
4  }
5 }
```

We can also create an `Orange` class as a template for creating all oranges in our program. This class can also inherit `weight` and `edible` fields from its parent class `Fruit`.

```
1 class Orange extends Fruit {
2  boolean peeled = false;
3  Orange () {
4     edible = true;
5  }
6 }
```

The `Orange` class also declares a new field `peeled` in addition to all of the fields that it inherited from the class `Fruit`. It also provides a constructor.

A superclass such as `Fruit` need not declare all the functionalities. It may just leave precise meaning of some operations undefined. Such a class is referred to as an *abstract superclass*. For example, in the following redefinition of the class `Fruit`, on line 1 the keyword `abstract` says this class is not fully defined.

```
1 abstract class Fruit {
2  double weight;
3  boolean edible;
4  boolean isEdible() {
5    return edible;
6  }
7  abstract boolean isSweet();
8 }
```

In the class declaration on line 7, the method declaration `isSweet` is not fully defined. A subclass will provide an adequate definition of this method. The new definition of the class `Apple` now provides a method declaration on lines 2-4 that provides a concrete definition of the method `isSweet`.

```
1 class Apple extends Fruit {
2  Apple () {
3    edible = true;
4  }
5  boolean isSweet() {
6    return true;
7  }
8 }
9 class Orange extends Fruit {
10  boolean peeled = false;
11  Orange () {
12    edible = true;
13  }
14  boolean isSweet() {
15    return false;
16  }
17 }
```

The class `Orange` also provides a concrete definition of the method `isSweet`, which is different from `Apple`'s definition.

We can similarly create a Person class and a Boy class and have Boy inherit all of the functionality from the Person class.

```
1 class Person {
2  int age;
3  String name;
```

```
4   boolean willYouEat (Fruit f) {
5     return true;
6   }
7  }
8  class Boy extends Person {
9   boolean willYouEat (Fruit f) {
10    if (f. isEdible ())
11      return true;
12    else
13      return false;
14  }
15 }
```

The declaration of the `Person` class has a method declaration for an operation `willYouEat`. This method produces a value that is limited by values that `boolean` can take, it consumes a value `f` that is a `Fruit`. By default, a person always eats the fruit.

The subclass `Boy` inherits the functionality from the superclass `Person`, but it also has the method declaration `willYouEat` on lines 9-14. Notice that this method declaration has the same name and same arguments (Fruit f) as the corresponding method in the superclass. Such methods are said to *override* corresponding methods in the superclass. The statements on line 10-13 define the logic of this *overriding method*. They say that, unlike a person that unconditionally eats a fruit, a boy will eat a fruit only if it is edible. The statement on line 10, `f.isEdible()` is called a *method call*. It should be thought of as asking the object `f` to complete operation `isEdible` and provide the result of that operation. A method call is the primary mean of requesting an object to perform an operation. In a method call, the object that is being requested is referred to as the *receiver* object.

When a method (e.g. `willYouEat`) is called, the body of some method declaration (e.g. lines 5 in class `Person`, or line 10 - 13 in class `Boy`) runs. The process of determining which method declaration runs is called *dispatch*. In the Java language most specific method declaration is run. To illustrate all of this, consider a new class `FruitProgram` below that combines all of the classes that we have discussed so far.

```
1  class FruitProgram {
2   static void main (String [] args) {
3     Person p = new Boy();
```

```
4      Fruit f = new Apple();
5      p.willYouEat(f);
6    }
7  }
```

In Java, by convention, a full program must have a class that provides a method with the signature matching line 2 in the code above. When you run this Java program it runs the statement on line 3 that creates a `Boy` object `p`, an `Apple` object `f`, and then calls the method `willYouEat` on the receiver object `p` on line 5. This will cause the method declaration `willYouEat` in the class `Boy` to run (see previous page), which in turn will cause the method declaration `isEdible` in the class `Apple` to run.

Object-oriented languages often distinguish between "what" an object does and "how" it does it. This allows "how" to be changed, provided that the object continues to do the same kinds of things. The programming language feature *interfaces* allows a Java programmer to express "what". This example defines an interface `Food`.

```
1  interface Food {
2    boolean isEdible ();
3  }
```

The interface defines what it means for an object to be "Food": it must provide an operation `isEdible`. We can rewrite the class `Fruit`.

```
1  abstract class Fruit implements Food {
2    double weight;
3    boolean edible;
4    boolean isEdible() {
5      return edible;
6    }
7    abstract boolean isSweet();
8  }
```

This change in line 1 now says that a fruit is-a kind of `Food`. This change allows the class `Person` to become more general.

```
1  class Person {
2    int age;
3    String name;
4    boolean willYouEat (Food f) {
```

```
5     return true;
6   }
7  }
8  class Boy extends Person {
9   boolean willYouEat (Food f) {
10    if (f.isEdible ())
11      return true;
12    else
13      return false;
14  }
15 }
```

On line 4 and line 9, the two method declarations now declare `Food` as argument, which means that we can apply them to objects of other kinds besides `Fruit` (provided that those objects are of kind `Food`).

To summarize, object-oriented programs in Java are a collection of objects. The properties of an object are defined by its class declarations. A class declaration can define properties as field declarations, operations as method declarations, and initializing operations as constructor declarations. A class can inherit functionality from a superclass, which allows code reuse. The functionality of a program can be divided into "what" (interfaces) and "how" (class declarations). This division makes programs more extensible.

1.6 Further Reading

- See "Structure and Interpretation of Computer Programs" by Gerald Jay Sussman and Hal Abelson available at the URL `https://mitpress.mit.edu/sicp/full-text/book/book.html`, Chapter 1.1 for more discussion on elements of programming