UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B
Spring 1998

P. N. Hilfinger

A Model of Programming Languages*

1 Programming Models

One way to find your way around a programming language is to learn individual answers to numerous questions of the form "How do I get it to do X?". One can gain a certain amount of proficiency by mastering the answers to a very large body of such questions, but it is far better to find a more efficient way, preferably one that will serve to help you find your way around numerous other programming languages as well.

When I started programming, I first learned a couple of machine languages (for the IBM 1620 and the IBM 1401, if you are curious). This approach had various advantages. I could understand constructs in higher-level programming languages (which to me at the time meant FORTRAN, Basic, and Algol 60) by informally translating them into corresponding machine code. The reasons for certain peculiarities in the design of programming languages (such as why integers had limited range) became apparent. The relative speeds of alternative codings of a program fragment became more easily predictable. The weird effects of certain bugs in my programs became less surprising. In short, I adopted a *model* for the execution of programs and used it to explain, understand, and predict my programs' behavior. This model was by no means precise—I didn't actually view my FORTRAN programs as assembly language programs—but it gave me conceptual signposts to guide my understanding.

Unfortunately, this machine model of programs has its drawbacks. It's a pretty substantial jump from some of the constructs used in modern programming languages to their machine-code realizations. Many of the details of how a computer does things are largely irrelevant to understanding a program. For example, one usually makes no use of the the fact that a pointer to a pair in Scheme is actually a number. Another example is that most machines have a finite set of variables known as *registers,* which must be used for certain operations, but which are typically invisible in high-level programming languages. Accordingly, nowadays I usually find myself using a more abstract conceptual model for most purposes. In this Note, I will present a model suitable for Scheme, Java, and C++ programs. You will find many similarities to what you learned in CS 61A (especially the environment models discussed there), and you may wish to review the textbook and handouts for that course.

2 Overview of the Model

The model presented here consists of the following components:

Values are "what data are made of." They include, among other things, integers, characters, booleans (true and false), and pointers (see below). Values, as I use the term, are *immutable;* they never change.

Containers contain values and other containers. Their contents (or *state*) can vary over time as a result of the execution of a program. Among other things, I use the term to refer to what are elsewhere called *variables* and *objects.* Containers may be *simple,* meaning that they contain a single value, or *structured,* meaning that they contain other containers, which are identified by names or indices.

Types are, in effect, tags that can be stuck on values and containers like Post-it™ notes. Every value has such a type, and in Java, so does every container. Types on containers determine the sorts of values they may contain.

Environments are special containers used by the programming language for its local and global variables.

The rest of this Note provides detail.

3 Values

One of the first things you'll find in an official specification of a programming language is a description of the primitive values supported by that language. In Java, for example, you'll find seven kinds of number (types `byte`, `char`, `short`, `int`, `long`, `float`, and `double`), booleans, and pointers. In C and C++, you will also find functions (there are functions in Java, too, but the language doesn't treat them as it does other values), and in Scheme, you will find rational numbers and symbols.

The common features of all values in our model are that they have types (see §5) and they are immutable, that is, they are changeless quantities. We may loosely speak of "changing the value of `x`" when we do an assignment such as '`x = 42`' (or '`(set! x 42)`') but under our model what really happens here is that `x` denotes a *container,* and these assignments remove the previous value from the container and deposit a new one. At first, this may seem to be a confusing, pedantic distinction, but you should come to see its importance, especially when dealing with pointers.

A *pointer* (also known as a *reference*) is a value that designates a container. When I draw diagrams of data structures, I will use rectangular boxes to represent containers and arrows to represent pointers. Two pointer values are the same if they point to the same container. For example, all of the arrows in Figure 1a represent equal pointer values. As shown there, we indicate that a container contains a certain pointer value by drawing the pointer's tail inside the container.

Certain pointer values are known as *null pointers*, and point at nothing. In diagrams, I will represent them with the electrical symbol for ground, or use a box with a diagonal line through it to indicate a container whose value is a null pointer. Figure 1b illustrates these conventions with a "free-floating" null pointer value and two containers with a null pointer value.

4 Containers

A *container* is something that can contain values and other containers. There are two varieties: simple and structured. A *simple container,* represented in diagrams as a plain rectangular box, contains a single value. A *structured container* contains other containers, each with some kind of label; it is represented in diagrams by nested boxes, with various abbreviations. The full diagrammatic form of a structured container consists of a large container box containing zero or more smaller containers[1], each with a *label* or *name,* as in Figure 2a. Figures 2b–d show various alternative depictions that I'll also use. The inner containers are known as *components, elements* (chiefly in arrays), *fields,* or *members.*

An *array* is a kind of container in which the labels on the elements are themselves values in the programming language—typically integers or tuples of integers. Figure 3 shows various alternative depictions of a sample array whose elements are labeled by integers and whose elements are contain numbers.

Value or Object? Sometimes, it is not entirely clear how best to apply the model to a certain programming language. For example, we model a pair in Scheme as an object containing two components (`car` and `cdr`). The components of the pair have values, but does pair *as a whole* have a value? Likewise, can we talk about the value in the arrays in Figure 3, or only about the values in the individual elements? The answer is a firm "that depends." We are free to say that the container in Figure 3a has the value `<2.7, 0.18, 2.8>`, and that assigning, say, 0 to the first element of the array replaces its *entire* contents with the value `<0, 0.18, 2.8>`. In a

[1]The case of a structured container with no containers inside it is a bit unusual, I admit, but it does occur.

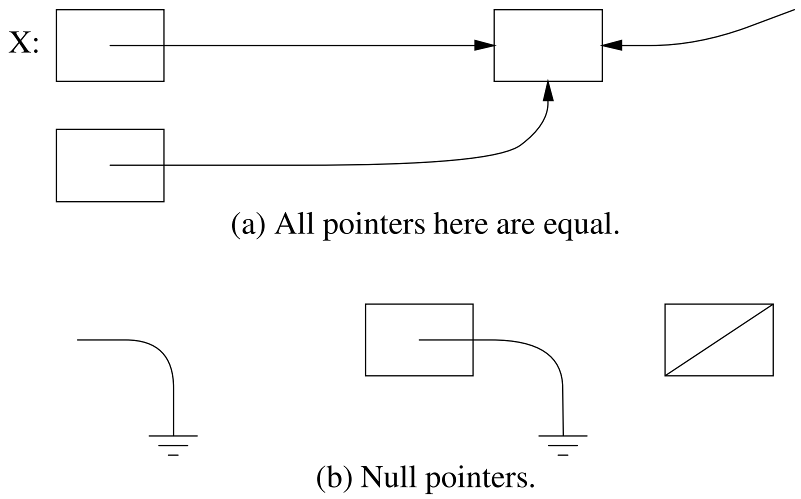

Figure 1: Diagrammatic representations of pointers.

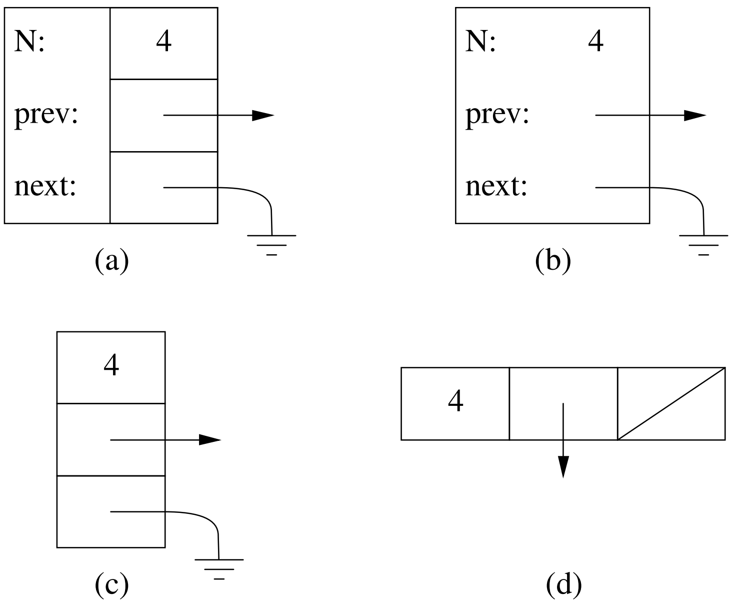

Figure 2: A structured container, depicted in several different ways. Diagrams (c) and (d) assume that the labels are known from context.

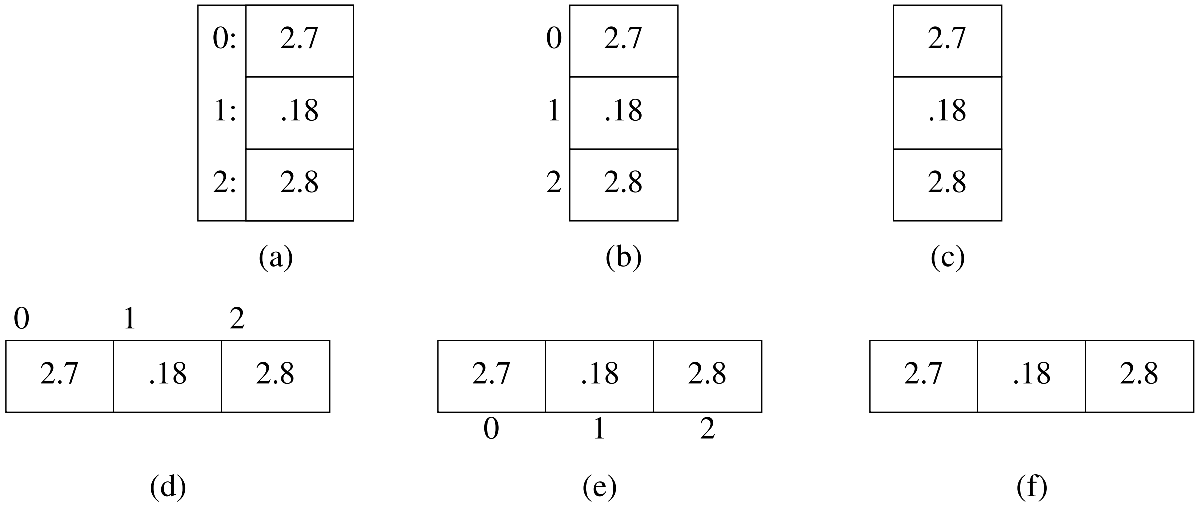

Figure 3: Various depictions of one-dimensional array objects. The full diagram, (a), is included for completeness; it is generally not used for arrays. The diagrams without indices, (c) and (f), assume that the indices are known from context or are unimportant.

programming language with a lot of functions that deal with entire arrays, this would be useful. In Java, however, we don't happen to need the concept of "the value of an array object."

5 Types

The term "type" has numerous meanings. One may say that a type is a set of values (e.g., "the type `int` is the set of all values between -2^{31} and $2^{31} - 1$, inclusive.") Or we may say that a type is a programming language construct that defines a set of values and the operations on them. For the purposes of this model, however, I'm just going to assume that a type is a sort of "tag" that is attached to values and (possibly) containers. Every value has a unique type. This does *not* necessarily reflect reality directly. For example, in typical Java implementations, the value representing the character `'A'` is indistinguishable from the integer value 65 of type `short`. These implementations actually use other means to distinguish the two than putting some kind of marker on the values. For us programmers, however, this is an invisible detail

Any given programming language provides some particular set of these type tags. Most provide a way for the programmer to introduce new ones. Few programming languages, however, provide a direct way to look at the tag on a value (for various reasons, among them the fact that it might not really be there!).

When containers have tags (they don't have to; in Scheme, for example, they often don't), these tags generally determine the possible values that may be contained. In the simplest case, a container labeled with type `T` may only contain values of type `T`. In Java (and C, C++, FORTRAN, and numerous other languages), this is the case for all the numeric types. If you want to store a value of type `short` into a container of type `int`, then you must first *coerce* (a technical term, meaning *convert*) the `short` into an `int`. As it happens, that particular operation is often merely notional; it doesn't require any machine instructions to perform, but we can still talk that way.

In more complex cases, the type tag on a container may indicate that the values it contains may one of a whole set of possible types. In this case, we say that the allowable types on values are *subtypes* of the container's type. As a special case, any type is a subtype of itself; say that one type is a *proper subtype* of another to mean that it is an unequal subtype.

If type C is a subtype of type P, and V is a value whose type tag is C, we say that "V *is a* P" or "V *is an instance of a* P." Unfortunately, this terminology makes it a little difficult to say that V "really is a" a P and not one of its proper subtypes, so in this class I'll say that "the type of V is exactly P" when I want to say that.

All this discussion should make it clear that the tag on a value can differ from the tag on a container that holds that value. This possibility causes endless confusion, because of the rather loose terminology that arose in the days before object-oriented programming (it is object-oriented programming that gives rise to cases where the confusion occurs). For example, the following program fragment introduces a variable (container) called `x` and says that the container's type is `P`:

```
P x;
```

Programmers are accustomed to speak of "the type of `x`." But what does this mean: the type of the value contained in `x` or the type of the container itself (i.e., `P`)?

We will use the phrase "the *static type* of `x`" to mean the type of the container, and the phrase "the *dynamic type* of `x`" to mean the type of the value contained in `x`. This is an extremely important distinction! Object-oriented programming in C++ or Java will be a source of unending confusion to you until you understand it.

6 Environments

In order to direct a computer to manipulate something, you have to be able to mention that thing in your program. Programming languages therefore provide various ways to *denote* values (literal constants, such as `42` or `'Q'`) and to *name* containers. Within our model, we can imagine that at any given time, there is a set of containers, which I will call the *current environment,* that allows the program to get at anything it is supposed to be able to reach. In Java (and in most other languages as well) the current environment cannot itself be named or manipulated directly by a program; it's just used whenever the program mentions the name of something that is supposed to be a container. The containers in this set are called *frames*. The named component containers inside them are what we usual call local variables, fields, parameters, and so forth. You have already seen this concept in CS 61A, and might want to review the material from that course.

When we have to talk about environments, I'll just use the same container notation used in previous sections. Occasionally, I will make use of "free-floating" labeled containers, such as to indicate that `X` is a variable, but that it is not important to the discussion what frame it sits in.

7 Important Concepts

This Note summarizes quite a few rather important concepts. Early on in a programming course, this may all seem rather abstract and vague. Therefore, you will do well to review the concepts in this Note from time to time throughout the semester. See if you can "attach" them to programming languages you already know and look out for their appearance while learning Java. Be particularly sure to understand the following terms and phrases: value, container, simple container, structured container, component (element), pointer (reference), type, static type, dynamic type, subtype, proper subtype, "V is a T," "type of a value," " type of a container," coercion, environment, and frame.

X: $\boxed{42}$