CMU-CS-80-146

510.7808
C28r
80-146
c.2

An Object-Oriented Command Language

Richard Snodgrass

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

October 1980

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551.

Abstract

This report describes Cola, an object-oriented command language for Hydra; Hydra is a capability-based operating system that runs on C.mmp, a tightly-coupled multiprocessor. Cola was designed to effect a correspondence between capabilities in Hydra and objects that are supported by the language. Cola is based on Smalltalk in that it uses message-passing as a control structure to allow syntactic freedom in the expression of commands to the system. Cola objects are arranged in a hierarchy, and the message-passing mechanism was designed to exploit this structure by automatically forwarding an unanswered message up the hierarchy. Two ramifications of this mechanism, automatic inheritance and shadowing, are discussed. An evaluation of the design decisions is also given. The second part of the paper discusses the syntax and semantics of the language in more detail, and the appendices contain a few examples of Cola objects.

Acknowledgements

The CL [Reiner 77] provided valuable insight into some of the inherent advantages and disadvantages of a command language under Hydra. Most of the underlying language concepts can be traced directly back to Smalltalk-72 [Goldberg 78]. The concept of classes and instances originated with Simula [Birtwistle 73], and some of the concepts of the class hierarchy (especially the concept of shadowing) were developed independently by Scott Fahlman [Fahlman 79] and were hinted at in the Pygmalion system [Smith 75]. Many people contributed both directly and indirectly through helpful discussions and comments on earlier versions of this paper, and through explanations of the intricacies of Hydra and Smalltalk. I would especially like to thank Gail Kaiser, Joseph Newcomer, Karsten Schwans, Philip Wadler, and William Wulf for their help.

Table of Contents

1. Introduction

Cola is an object-oriented command language which grew out of a need for a comfortable user interface for Hydra [Wulf 81], a capability-based operating system that runs on C.mmp [Wulf 72], a tightly-coupled multiprocessor. Hydra provides a comprehensive set of facilities to the user, yet the previous command language for Hydra (called the *CL*[1]) presented these facilities as an unrelated collection of procedures. The overall character of the operating system was not reflected in the way that the user interacted with the command language. In the design of Cola, much effort was expended to mirror the philosophy of the operating system in the command language. The incorporation of *objects*, similar to Simula classes, in the command language was a result of this objective.

That the concept of class might by a valuable addition to command languages is suggested by an analogy between the development of command languages and general purpose languages. When the hardware is first introduced, one programs in assembly language, due in part to a lack of higher level software, an inadequate understanding of what structuring concepts are useful, and a desire to make the most of a scarce resource. Assembly language programs can be characterized by their utilization of the full power of the hardware by building on only the basic facilities available. One command language analogue to assembly languages is the OS/360 Job Control Language [Mealy 66], which shares these same properties. As with assembly language, anything is possible in JCL, and almost everything is difficult. The first step toward making the machine easier to program was Fortran, which provides a few basic control structures, such as DO-loops and subroutines, as well as a few data structures, such as arrays and COMMON storage. The George 3 Command Language [Ostreicher 67], which includes conditional and looping statements, as well as user-defined macros, embodies some of the advances found in Fortran.

Algol was the next major development in programming languages; concepts such as data types, block structure, and recursive procedures first appeared in this language. Similar ideas can be found in the Burrough's Work Flow Language [Cowan 75], IBM's CMS language [IBM 80], OSL/2 [Alsberg 71], SCL [Brunt 76], and the CL, although they have been adapted for a command language domain[2]. For example, SCL uses the block structure of a control program to limit both the scope of variables (as in Algol) and the scope of operating system resources, and the CL interprets certain names to refer to objects in the file system rather than in main memory. The next major step in programming

[1]The CL [Reiner 77] is a general expression -oriented block structures programming language. The syntax resembles Bliss [Wulf 75], the system implementation language Hydra itself was written in. Some features have been added (such as a capability data type) or altered (such as a more versatile assignment statement) to allow access to specific Hydra facilities.

[2]Lisp has also been taken as a starting point in the design of a command language [Ellis 80, Levine 80]

languages (one that is currently still in progress) is the introduction of abstract data types [Shaw 80]. The concept of class first appeared in Simula [Birtwistle 73], with Euclid [Lampson 77], Alphard [Wulf 76], and CLU [Liskov 77] continuing the emphasis on abstraction and modularization. Despite the advantages inherent in these developments, the concept of abstract data types has not yet appeared in command languages.

A similar analogy between general purpose languages and operating systems also suggests incorporating classes into the command language. In one sense, an operating system is merely a large, complex runtime system for the user's program. This was true before multiprocessing, and applies even more with the advent of operating systems for personal computers, which are usually single language machines [Lampson 79, Redell 80]. The concepts introduced in programming languages tend to be transfered to the runtime systems, as well as the operating systems, which support them [Jones 77]. Thus, there has been a flurry of activity in recent years concerning object-based operating systems [Jones 79, Ousterhout 80, Wilkes 79]. In systems such as these where the concept of object pervades the programming language, its runtime system, and the operating system itself; the command language should provide consistency by also supporting objects.

Part 1: Overview

This document is divided into three parts. The first provides an overview of Cola by examining the implications its design has on its usage as a command language and on the ability to organize objects in a coherent fashion. The second part is more concrete, describing an informal syntax and semantics for the language. The third part, the appendices, contains several examples that are used to illustrate some of the constructs that appear in the language.

The first part begins by motivating the character of Cola through a discussion of the facilities that a command language for Hydra should contain. Cola and the CL are compared, and it is argued that Cola is more successful at incorporating the abstractions present in the operating system in a uniform manner. The third chapter investigates the rich structuring mechanisms for objects available in Cola. These mechanisms facilitate efficient storage of both static knowledge (in the form of data structures) and dynamic knowledge (in the form of control structures). The rest of this part sums up what has been learned through this effort and what still remains to be done.

4

2. Cola as a Command Language

As indicated above, much progress has been made in providing more powerful command languages. There are, however, arguments for eliminating the command language completely, and instead embedding the functionality previously provided by the command language in the programming system, resulting in a programming *environment* with a uniform syntax and semantics [Feiler 80, Habermann 80, Lauesen 73, Sandewall 78, Teitelbaum 79, Teitelman 78]. These arguments include not having to learn a different command language, being able to utilize the control and data structures present in the existing language when writing job control programs, and aiding the standardization task for command languages.

Despite these advantages, such an approach is not always appropriate. Each programming environment must be developed separately for each language that is desired. Also, it is very difficult to write a system in several different languages if each is supported by its own environment. Due to these reasons, plus the lack of existing systems that provide operating system primitives within the language, it is necessary to provide some kind of user-level interface that can interact with all these language systems. Since the operating system is the one entity shared by the users of the various languages, and it is the operating system that is providing the services that the command language refers to, it is appropriate to align the command language as closely as possible with the operating system [Treu 75].

2.1 Overview of Hydra

Cola is a command language that was designed for Hydra [Wulf 81], a capability-based operating system that runs on C.mmp [Wulf 72], a tightly-coupled multi-processor consisting of sixteen DEC PDP-11's. The two most important attributes of Hydra -- supporting a multi-processor and implementing an object-based protection scheme -- are relatively orthogonal. In Hydra, capabilities are protected pointers that refer to resources, both physical and virtual, called *objects*. Each object consists of an array of capabilities (a *clist*) and an array of words (a *datapart*). A particular capability is refered to by specifying the index into a clist; similarly, data is referenced by indexing into the datapart. Since every resource in Hydra is associated with an object, all resources can contain data and capabilities. For instance, the datapart of a procedure object contains information relevant for execution and debugging (such as its trap and interrupt addresses, saved registers, etc.); the clist contains capabilities used by that procedure, including capabilities for *page objects*, which contain the code for the procedure.

In addition to using capabilities to support a very flexible protection scheme [Cohen 75], Hydra provides powerful abstraction mechanisms. Objects are typed, and associated with each type is a set of procedures that can manipulate the representation of objects of that type. The user can define new types by specifying the representation of objects of that type in terms of types that are already defined, and by providing procedures that can perform operations on objects of the new type. In this way, Hydra types are analogous to the abstract data types of Simula or Alphard.

Although capabilities and Hydra objects are quite different entities, the two terms are often used interchangably. Hence, instead of refering to 'the length of the datapart of the object pointed at by the capability called "ACapa",' one would refer to 'the length of the datapart of "ACapa"'. This convention will be followed in the remainder of this paper.

2.2 Objects and Message Passing

Cola is also based on the concept of objects. A *Cola object* is a potentially active piece of knowledge that communicates by sending messages composed of objects [Hewitt 77]. Cola is modeled closely after Smalltalk [Goldberg 78, Shoch 79]. Although it shares many characteristics with conventional languages incorporating abstract data types, Cola is unusual in that it uses message passing as a control structure to allow syntactic freedom in the expression of commands to the system, an important consideration in the design of an interactive language.

As an example of message passing, evaluating ⟨object⟩ + 4 means the message '+ 4'[3] is sent to ⟨object⟩, which interprets the message, and returns another object as a reply. In the procedural view, '+' would be an infix operator defined in the types within which the plus operator was valid. In Cola, the '+' in the message is interpreted by 3 (as an example of ⟨object⟩) as ordinary addition, with the integer 7 returned; for the ⟨object⟩ 'a string', the plus sign is interpreted as string concatenation, with the string 'a string4' returned. Each object can interpret a message in any way it sees fit. Message passing is entirely consistent with, and indeed, supports, the information hiding aspects inherent in abstract data types [Ingalls 78].

Every object in Cola belongs to a *class*, which is analogous to a type in other languages. The class, also an object, defines the messages that all its members can accept, as well as the semantics for each of the messages. It also defines the data structures that can reside in each member. The code that is associated with a class is shared by all the members of that class. Hence, the object 4 is a member of the class INTEGER; the object 'a string' is a member of the class STRING. The class structure is actually much richer than described here; more will be said when the object hierarchy is discussed in chapter 3.

[3]The following typographical conventions will be followed in this report. Names of predefined and user-defined classes (such as INTEGER) will be written in small capitals; messages sent to an object will be in italics, or enclosed in single quotes. Strings will also be enclosed in single quotes; the context should disambiguate the two cases. Examples will be in boldface, and the names of atoms (corresponding to variables in other languages) will be enclosed in double quotes.

2.3 The Cola/Hydra Correspondence

Cola was designed to effect a correspondence between objects (actually, capabilities) in Hydra and objects that are supported by the language: every object (capability) in the user's environment in Hydra is associated with an object in the user's environment in Cola. A system call embedded in a program can cause Hydra to perform an operation on an object referred to by the capability mentioned in the system call. In the same way, a message can be sent to the Cola object associated with that capability to perform the operation. In responding to the message, the Cola object, as a side effect, executes the system call on that capability. Thus, there is no distinction between Hydra objects (capabilities) and Cola objects, resulting in an isomorphism between the two entities[4].

This correspondence is qualitatively different from the view of Hydra presented by the CL. The CL appears to the user as a set of predefined procedures which can operate on capabilities. These procedures correspond directly to the system calls available to Bliss programs running on Hydra [Cohen 76]. Hence the CL is effectively an interpreted Bliss (indeed, if Bliss were an interactive language, then the CL as implemented would have added little functionality.)

Although the use of Bliss as a base for the command language resulted a rather powerful user interface, it suffered from the restriction that communication with the operating system can occur only via system calls. Since objects consist of an array of integers (the datapart) and an array of capabilities (the clist), it would be natural to access objects as arrays. Instead, one retrieves the data of an object by executing a system call (actually, in this case an ad hoc extension was made to the CL to allow array accessing to be done on Hydra capabilities). The CL lacks the ability to *dynamically* define new types at the command level, and to define operations associated with these types which are reflected via system calls to the Hydra objects that are referred to by the command objects.

Cola provides this functionality and, as a result, presents to the user a different perspective on the operating system. Cola integrates the objects supported by the operating system into the language itself (due to the Cola-Hydra object isomorphism), eliminating the cumbersome system call interface. Note that the implementation still uses system calls, but this detail is hidden from the user. Instead, the user interacts with the command language using the abstractions with which she is familiar with, namely those supported by the operating system.

[4] This is not strictly true, since there are Cola objects such as INTEGER which do not have a Hydra analogue. (Ideally, Hydra would handle integers as objects, but the implementation makes small objects inefficient.)

2.4 Non-object-based Operating Systems

The object concept in Cola can be usefully incorporated into command languages for operating systems that are not themselves based on the object model. The function of an operating system is to provide resources for user jobs that can be manipulated by the job by executing system calls. These resources are essentially typed objects (such as files, directories, I/O ports, memory) with operations defined on them (such as print, add entry, send a character, reserve), although they are not always implemented as such in the operating system. It is this thinking that has motivated research in object-based operating systems, and the use of objects in the command language is merely an extension of this concept, independent of the use of objects in the underlying system.

3. The Object Hierarchy

Although the main impetus for this research was the design of a command language for Hydra, the language that evolved out of this effort is interesting in its own right. Cola objects are useful not only as command language surrogates for objects (capabilities), but also have many properties that make them useful in models of computation [Grief 75] and in personal computer languages [Goldberg 79, Kay 77, Warren 79]. Since an object is an active piece of knowledge, one aspect of this research considered structuring objects in ways that have been found to be useful in structuring knowledge [Minsky 75]. Cola uses a hierarchical ordering of classes coupled with an execution semantics and binding mechanism to represent static and dynamic knowledge within the class structure.

3.1 Simula Subclassing

Simula, the first language to incorporate classes, used a *subclassing* mechanism to structure objects. A *subclass* is a refinement of a class: it inherits all of the procedures and data structures of its parent class, and augments these with its own procedures and data structures. Subclasses can also be refined further by their own subclasses, resulting in a tree structure. An *instance* of a class contains the values of all the data structures defined in all of its defining classes. The subclassing mechanism of Simula is a static, compile-time structure that almost completely disappears in the runnable version of the program. Smalltalk-76 has a similar mechanism that is partially interpreted at runtime [Ingalls 78].

There are several advantages inherent in such a scheme. Since the subclass inherits all of the traits of its defining class (its *superclass*), the code for the superclass need not be duplicated. Instead, the subclass uses all of the code it needs from its superclass, and adds traits of its own. Thus the mechanism provides a powerful structuring capability to the language. Other advantages, due to the message passing mechanism, will be discussed shortly.

The disadvantages of subclasses stem from the decision to place all of the data structures in the instance. Information associated with a superclass is replicated in all of the instances of that class. This arrangement complicates the modification of data associated with a class located several levels above the instance, and invokes the traditional consistency problems associated with redundant data (such as how does one make sure that *all* the instances of replicated data have been updated). There is thus an asymmetry in the distribution of data structures and procedures in Simula: procedures are shared by all subclasses of the class containing the procedure; data structures are not shared at all, but exist separately in every instance (note however that the *names* of the data structures are shared in the same manner as procedures). The Cola subclassing mechanism has been designed to allow the sharing of data structures while retaining the advantages mentioned above.

The Cola subclassing mechanism orders all objects in a hierarchical fashion. At the top of the hierarchy is the class (or object, since all classes are objects) called OBJECT. OBJECT is associated with a set of classes (called *subclasses of* OBJECT) through the relation *subclass*. Similarly, each of these classes is associated with object through the relation *superclass*. Subclass is a many-to-one relation -- a class can (and usually does) have many subclasses. Superclass is a one-to-many relation -- a class is restricted to having a unique superclass, but several classes can have the same superclass. Thus the subclass-superclass relation produces a tree structure of classes, as in Simula, with the class OBJECT being the root node.

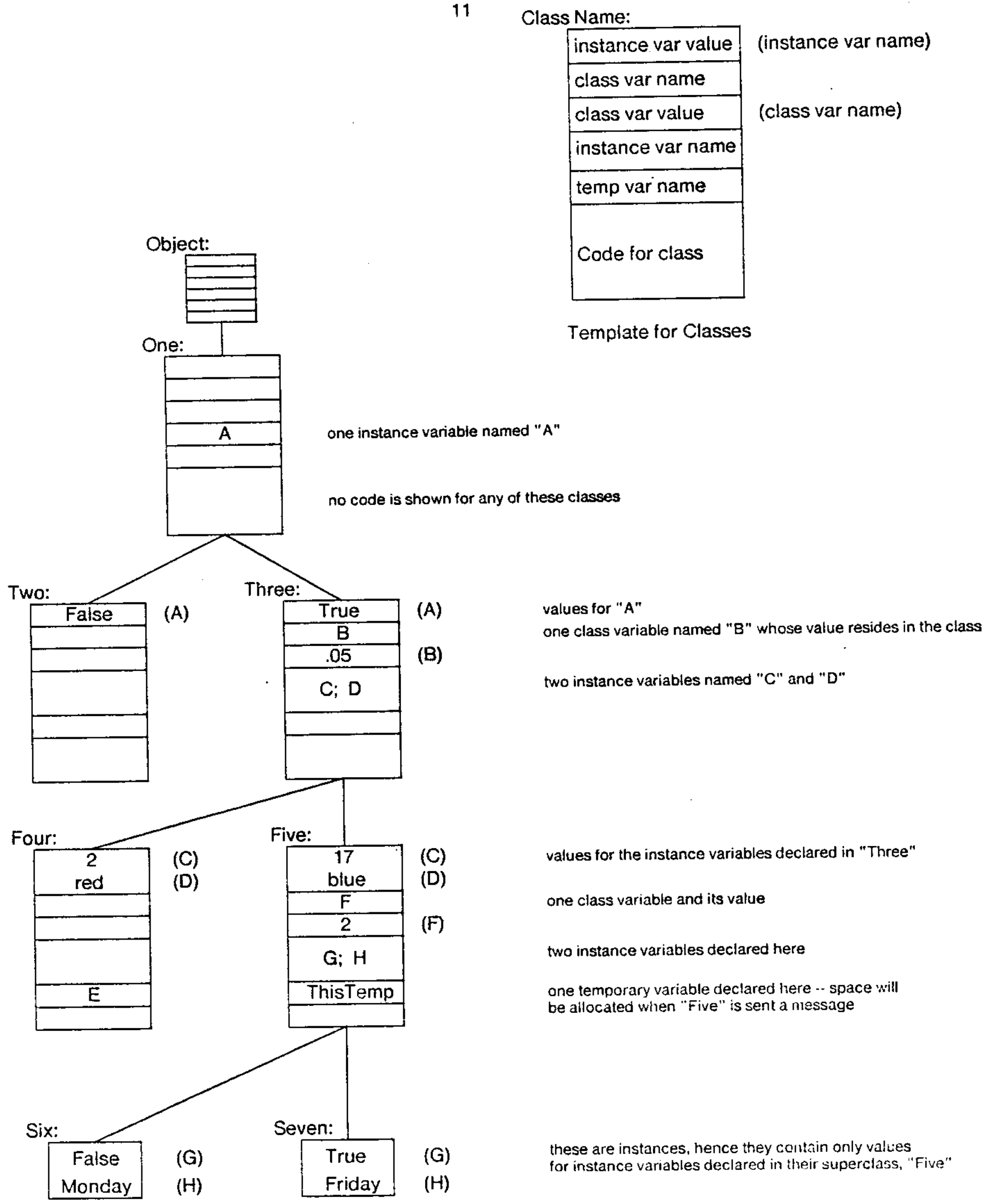
3.2 Naming

Associated with every class are three kinds of variables, permiting flexibility in the placement of the values of the variables within the object hierarchy. These variables are class variables, instance variables, and temporary variables. *Class variables* are named in the class and are associated with values that reside in the class. They correspond to **Own** variables, in that their value is shared by all instantiations of that class. (In Figure 1, "B" is a class variable of "Three", with value ".05".) *Instance variables*, on the other hand, are named in a class, but are associated with values that reside in the *immediate subclasses* of the class. They correspond to the data structures defined in Simula classes, except that the values are stored in the next lower level, rather than in the leaves (i.e., the instances) as in Simula. ("C" and "D" are instance variables of "Three", with values in "Four" and "Five".) Values for *temporary variables* are created on every invocation of the object they are associated with using the traditional stack discipline, and are destroyed when the object returns. They correspond to variables designated as **Local**, **Var**, or **Recursive** in other languages. ("E" and "ThisTemp" are then temporary variables shown in Figure 1.) When a class is defined, the names of these three types of variables are declared. The name of the superclass must also be declared when the class is defined.

3.3 Instances

Instances are objects that differ from classes in only one way: there is no code associated with instances, whereas there must be code associated with classes. This distinction is not necessitated by the logical framework developed so far, but occurs because of the way that classes are defined. The restriction that results is that instances cannot create subclasses or subinstances. Therefore, instances appear as leaf nodes in the hierarchy produced by the subclass-superclass relation. (In Figure 1, "Six" and "Seven" are instances, each containing values for the instance variables declared in "Five".)

Instances correspond to values of variables, where the type of the variable is the superclass of the



Class Name:
instance var value
(instance var name)
class var name
class var value
(class var name)
instance var name
temp var name
Code for class
Template for Classes
Object:
One:
A
one instance variable named "A"
no code is shown for any of these classes
Two:
False
(A)
Three:
True
(A)
B
.05
(B)
C; D
values for "A"
one class variable named "B" whose value resides in the class
two instance variables named "C" and "D"
Four:
2
red
(C)
(D)
E
Five:
17
blue
(C)
(D)
F
2
(F)
G; H
ThisTemp
values for the instance variables declared in "Three"
one class variable and its value
two instance variables declared here
one temporary variable declared here -- space will be allocated when "Five" is sent a message
Six:
False
Monday
(G)
(H)
Seven:
True
Friday
(G)
(H)
these are instances, hence they contain only values for instance variables declared in their superclass, "Five"

Figure 1: A class hierarchy with the data structures shown (the arcs represent the subclass-superclass relation)

instance. Hence the instance 3 has as a superclass the class INTEGER. The atom associated with the instance corresponds to the variable itself.

Since instances do not have any code associated with them, they also do not have either class variables or temporary variables associated with them. Instances contain only the values of instance variables declared in the superclass of the instance.

Most of the statements expressed in the remainder of this chapter apply to all objects; where there exists differences for classes and instances, the differences will be noted. Also, the detailed syntax will be delayed until the second part of this document.

3.4 Execution Semantics and the Binding Mechanism

The control flow is tied to the object hierarchy and allows procedures contained in a class to be shared by its subclasses (the sharing mechanism will be detailed in the next section). When an object is sent a message, the code for that object is invoked. If there is no code associated with the object (i.e., the object is an instance), or if the class does not recognize the message, then the superclass of the object is sent the same message (called *forwarding* the message). This process continues until some class recognizes the message, or the class OBJECT is invoked (OBJECT recognizes every message and responds with some default reply). A class recognizes a message by *returning* a reply.

The binding mechanism is also tied to the object hierarchy and allows data structures to be distributed. The binding mechanism is a combination of static binding (for class and temporary variables) and dynamic binding (for instance variables). When a class is defined, the names of the class, instance, and temporary variables are given, as well as the code to be associated with the class. The binding mechanism for class variables appearing in the class code simply binds the name to the value that resides in the class. Temporary variables are bound to the storage allocated at the time of invocation. The binding of temporary varibles remains in effect until the object returns a reply, at that time the storage is recovered and the binding broken.

The binding mechanism for instance variables is more complicated due to the possible forwarding of a message. If a message is being forwarded from a subclass or instance, then the instance variables are bound to the values which reside in that subclass or instance. If the class was the class that was originally sent the message, then there are no values to bind to the instance variables, and an error occurs if they are referenced.

The object CLASSINVOKED (see section 9.4) returns TRUE if the class itself was sent the message. The object SELF refers to the object that was originally sent the message. It is thus possible to associate different procedures with messages sent to a class and messages sent to a subclass of the class (analogous to *triggers* and *traps*, respectively, in KRL [Bobrow 76].)

The binding mechanism restricts the set of variables that may be referenced by the code in any given class. Since definitions must proceed in a top-down fashion (the superclass must be specified in the definition of a class), it makes no sense for a class to refer to a variable declared in one of its subclasses. Similarly, variables declared in a separate branch of the hierarchy are also inaccessible. To access a variable that is defined in a superclass, the class must send a message to its superclass requesting the value of the variable; this message is forwarded up to the appropriate superclass, which responds with the value of the desired variable. This process also applies to information contained in classes that are above the object originally sent the message, yet below the class currently dealing with the message, since a request for that information can be sent by a class to SELF.

3.5 Automatic Inheritance and Shadowing

One of the ramifications of the execution semantics and the data structuring is the *automatic inheritance* by a class of the ability to respond to all the messages that the successive superclasses of the class can respond to.

As an example, suppose that the classes INTEGER and REAL are subclasses of the class NUMBER. Each of the subclasses handle the messages[5] < *number*, = *number*, and *print*; NUMBER handles the messages > *number*, >= *number*, <> *number*. Each of the subclasses of NUMBER provides a minimal set of operations, and NUMBER augments these operations with ones that can be composed of operations from the minimal set. In this way, NUMBER contains the mechanisms common to INTEGERS and REALS, and each of the INTEGER and REAL classes contain only mechanisms which are unique to them. As an example, suppose that the INTEGER 3 is sent the message '> 4'. Since INTEGERS do not know how to respond to this message, it gets forwarded to NUMBER. NUMBER responds by sending the message '< 4' to SELF, which is defined to be the object that was sent the original message (in this case, the INTEGER 3). This object responds with TRUE, which NUMBER uses to respond to the original message with the reply FALSE. (Note that if the object responds with FALSE, then NUMBER would have to send SELF the message '= 4'.) If a new subclass COMPLEX was defined (that included the ability to respond to < *number*), it would automatically inherit the ability to respond to the message > *number* by forwarding the message up to NUMBER.

This mechanism of automatic inheritance is very useful. Simula [Birtwistle 73] applied a form of automatic inheritance to build up an entire sublanguage suited to the construction of simulation programs. Most knowledge representation systems incorporate the concepts of *prototypes* (i.e., classes), *entities* (subclasses and instances), and *property inheritance* [Winograd 75]. KRL [Bobrow 76], for example, includes a subclassing mechanism with automatic inheritance through the use of

5 *number* is used to represent an instance of the classes INTEGER or REAL.

perspectives. However, sometimes this mechanism is not desired, as in the case of a subclass that needs to respond differently to a message than its superclass. In Cola, one can override automatic inheritance through the use of *shadowing*.

To illustrate shadowing, suppose that the message *realpart* was to be added to all the subclasses of NUMBER (i.e., to INTEGER, REAL, and COMPLEX). Since most subclasses of NUMBER would respond to *realpart* by returning itself (SELF), we decide to add code to NUMBER which does just that. When an instance of the class INTEGER or the class REAL is sent the message *realpart*, it forwards the message up to NUMBER (by not recognizing it), and NUMBER responds with SELF. However, this mechanism is not correct in the case of an instance of COMPLEX responding to *realpart*. The solution is to have COMPLEX *itself* respond to *realpart*, by returning only the real component, *without* forwarding the message up to NUMBER. This enables NUMBER to contain the commonality of its subclasses, and enables any subclass to respond differently to any particular message if it sees fit to do so.

The general mechanism can be summarized as follows: each class represents static knowledge (in the form of class and instance variables) and dynamic knowledge (in the form of the ability to reply to certain messages). Subclasses and instances automatically inherit the knowledge that is found higher in the hierarchical tree, and augment this knowledge with further knowledge. Through the use of shadowing, it is possible for a subclass to respond differently to a particular message than its superclass would have, thus making possible the expression of exceptions without nullifying the knowledge that exists higher in the tree.

4. Summary

Essentially all of Cola has been implemented. The interpreter is written in Bliss/11 [Wulf 75], with Cola code augmenting the low level objects. The static and dynamic structures are very similar to Smalltalk-76 [Ingalls 78], even though they were designed independently. It should be emphasized that Cola is an experimental system and very little effort has gone into tuning the implementation for efficiency.

There are several conclusions to be drawn from this effort. A useful correspondence between the entities supported by the command language (Cola objects) and those supplied by the the operating system (Hydra capabilities) has been achieved. This correspondence is indeed "natural", in that there exist facilities in the language that support typed objects and the notion of operations (messages) that may be performed on these objects just as the operating system does. It is argued that the Cola paradigm can be successfully incorporated into a command language for a conventional operating system, although this premise has not been demonstrated concretely with an implementation.

Arguments for the message passing (verses procedure call) mechanism are less conclusive. At a basic level, this issue does not apply, since a duality exists between message-oriented languages and procedure-oriented languages just as it does in operating systems [Lauer 79]. As an example, the command

```
    A print
```

in Cola (which sends the object "A" the message *print*, causing "A" to print a representation of itself on the terminal) is equivalent to the statement

```
    print(A)
```

in the CL, with the instance (in this case, "A") passed implicitly as a parameter. The entire message forwarding mechanism can be simulated in Simula, although it must be done explicitly using additional procedure calls.

The primary advantage of message passing is that it is simple and does not impose a strict grammar on the language. The latter is usually considered to be a disadvantage in general purpose languages, but is convenient for a casual user interacting with a system at a terminal instead of carefully composing her programs before typing them in. A secondary advantage of message passing is that the mechanism lends itself naturally to multiprocess(or) systems [Yonezawa 77]. Although C.mmp, on which Cola runs, is a multiprocessor, this aspect was not dealt with in the design.

The primary disadvantage of message passing is that it is inefficient when implemented in the obvious fashion. This drawback is not as worrisome as might first be expected, for several reasons.

In a command language, efficiency is not a primary concern, since, on the average, a relatively small number of statements are executed as a result of a command typed by the user [Kernighan 79]. It can be argued that a command language procedure that is unacceptably slow should be rewritten in one of the languages supported by the operating system [Bourne 78]. In addition, the general message passing mechanism can be avoided most of the time in the interpreter (the semantics of the language would, of course, still be defined in terms of message-passing) by applying a few relatively simple transformations to the source before it is interpreted and by designing the interpreter so that it uses the local state to circumvent the message assembly mechanism except for special cases, primarily when an error occurs [Kay 80].

The object hierarchy, coupled with the message-passing and -forwarding mechanism, has been shown to have several useful attributes concerning the structuring of objects. Static knowledge, in the form of class and instance variables, is stored as high in the hierarchy as possible, eliminating redundancy at lower levels. Similarly, dynamic knowledge, encoded in the ability to respond to certain messages, is shared among many classes and instances. The mechanism allows flexibility in the placement of both procedures and data structures within the hierarchy.

One disadvantage is the restriction that the values of instance variables declared in a class must reside in each subclass of the class. This restriction can be removed by allowing more flexibility in the sharing of *names* of data structures within the objext hierarchy. The names of class variables, for instance, are not shared at all, since both the names and the values reside in the class they were declared in. In Cola, the names (but not the values) of instance variables are shared by the immediate subclasses of the class they were declared in. In Simula, the names of instance variables are shared by *all* the subclasses. However, the optimal placement of the value (and the name) of an instance variable depends to a large extent on the semantics desired for the information contained in that variable (and the sharing of the name of the variable), and mechanisms for allowing more variability for the binding and storage of instance variables need to be developed.

There are several other areas where additional research is needed. It is not clear how one would incorporate multiple process(or) concepts into the language. Several possible alternatives seem likely, including utilizing the message-passing mechanism and/or allowing multiple objects to execute concurrently, but the ramifications these various schemes have on the language has not been investigated at all. More work is necessary to make the paradigm of objects communicating via messages a viable one in terms of efficiency (although much has been done in this area by the Learning Research Group at Xerox PARC [Ingalls 78]). Lastly, the techniques used to mirror the abstractions provided by Hydra in the command language should be applied to other operating systems, including more conventional ones, in order to assess the applicability of these concepts.

Part 2: The Language

The first part of this document gave an overview of the Cola paradigm and some implications this paradigm has on the use of Cola as a command language and on the structuring of knowledge in Cola. This part discusses the syntax and semantics of the language in more detail. The first two chapters describe how the language handles messages; the next chapters describe the various commands and predefined objects available in Cola. Much of the first half of this part is devoted to the underlying structure of Cola, which closely resembles Smalltalk [Goldberg 78]. This material has been included primarily for completeness. The third part (the appendices) concludes with several examples of Cola objects.

5. The Interpretation Loop

Cola is an interpreted language. In most interpreted systems, the interpretation loop is very simple:

```
while system-still-running do
    token := getnexttoken()
    execute(token)
```

Although a great deal of the action has been hidden behind the two operations **getnexttoken** and **execute**, the above construct is fairly accurate for most systems. Cola's basic interpretation loop is also very simple:

```
proc Basic-Loop =
    while system-still-running do
        object := getnextobject()
        message := getmessage()
        send(message, object)
```

This very high-level view of Cola illustrates the main control mechanism in Cola: objects receiving messages. The **getnextobject** operation retrieves the object pointed to by the *codepointer* (associated with each context) and increments the *codepointer* to point to the next object. Since the message that is sent to an object is defined to be the stream of objects that follow the given object (see chapter 6), the **getmessage** operation bundles up the rest of the objects into a message.

The third operation, that of sending a message to an object, is considerably more complicated:

```
proc send (message, object) =
    objectflag := true
    while true do
        if object-has-code
            then instantiate(message, object)
                !Does the object return a result?
                if message-is-recognized
                    then object := result
                    else object := super(object)  !go up the tree
                objectflag := false
            else object := super(object)          !an instance
```

First, a few remarks on notation. **Super(object)** returns the superclass of **object**. The **objectflag** (a boolean value local to each context) indicates whether this class was originally sent the message. Comments are preceeded by an '!'.

The **instantiate(object, message)** operation creates a new context with a new *codeptr* and recursively calls **Basic-Loop**.

The interpretation loop has now been specified, except for one small detail. As it now stands, it seems that the system continues to recurse indefinitely, since **Basic-Loop** calls **send**, which calls **Basic-Loop**, etc., etc. This recursion is broken by *returning* a result (see chapter 6) or by running out of code in the object. (Some objects, such as the class OBJECT (see chapter 3), return a reply when they don't recognize a message. Since OBJECT is a superclass of all objects, the recursion is guaranteed to stop at some point.) Returning a result causes the current context to be popped off the execution stack, and the result becomes the next object a message is sent to.

Consider the following stream of objects:

```
3 + 4 ;
```

The instance **3** is sent the message '**+ 4 ;**'. **3** recognizes the message **+**, and calls **send(4, ';')**. **4** does not recognize the message, so it calls its superclass, OBJECT. OBJECT recognizes the message as a message delimiter (see section 7.8), so it replies with SELF, which is defined to be the original object that was sent the message (in this case, the instance **4** -- see section 9.10). This object is the reply of **4** to the message ';', and is received by **3**. **3** adds 4 to itself, and replies with '**7**', which is the expected response when **3** is sent the message '**+ 4 ;**'. An understanding of this example indicates an understanding of the basic paradigm of objects sending messages and receiving replies.

6. Messages

As mentioned previously, objects can receive messages and send replies. To send a message to an object, syntactically follow the object by the message. To send the object 3 the message '+ 4', the user would execute

```
3 + 4
```

When a Cola object is sent a message, there are ways to retrieve objects from the message and to send a reply. All message commands treat the message as a linearly ordered list of objects.

Messages are received by executing one of the following commands:

:	evaluate the message and return the value
get	return the next object in the message

The first two commands excise part of the message, in the sense that the part of the message that is fetched is no longer available to the object. The command

peek	return the next object in the message

does not alter the message in any way. The command

see xyz	if the next object is xyz, then fetch the object and return TRUE, otherwise return FALSE and do not fetch (note that xyz is not evaluated)

conditionally fetches the next object.

To send a reply, use the command

return <value>

If the class does not *return* a reply, then the class' superclass is sent the rest of the message (see section 7.9).

This view of messages is a pervading one in Cola. When Cola is at the top level, it waits for the user to specify an object and a message to be sent to that object. The object's reply is then typed back for the user to see. In effect, the user is merely another "object" that can receive messages and send replies (see section 8.13).

7. Command Syntax and Semantics

When the system is not executing commands, it continually executes the following sequence of tasks:

1. print the prompt, and read characters until a linefeed is typed;
2. assemble the characters in an object called a *string*;
3. evaluate the STRING; and
4. print whatever object the STRING returns.

Cola does these tasks by executing a special object called the *Driver* (see section 9.7). Whenever an error occurs, the appropriate message is printed out, and a new driver is invoked. Thus, at any point in time, there may be many drivers stacked up, each in a particular phase of execution. Performing a forward call on an error ensures that the state of the computation is reserved for interrogation, modification, and re-execution. Systems employing this mechanism (such as InterLisp [Teitelman 78]) often provide sophisticated debuggers which benefit from the information on the stack when the error occurred.

Whenever a new context is entered (i.e., a new driver is invoked, a left parenthesis is processed, etc.) the *level number* is incremented. Whenever a context is exited (i.e., a *return or done* (see below) is executed, or a right parenthesis is processed), the level number is decremented. The prompt consists of the level number followed by a '>'. The level number is printed to aid in determining the current relative position in the execution stack.

7.1 Comments

A comment consists of text delimited on the left by a '!' and on the right by a line terminator. Any characters between the '!' and the carriage return or the line feed are ignored. Multi-line comments are surrounded with '%'s. The '!' and '%' characters inside a STRING are treated normally.

7.2 Assignment

```
a := 3
```

This statement causes "a" to be considered an object of class INTEGER, with value 3. (See section 7.3 for further elaboration.) Any object can be assigned to a variable; in general, there is no type checking or type declarations (however, see sections 8.7, 8.8, and 8.9).

The result of an assignment statement is the value of the right hand side, so the statement

```
a := b := c
```

is valid (see section 8.1)[6].

7.3 Literals

Literals are converted into instances of predefined classes by Cola. When the object 3 (an INTEGER literal) is mentioned, for instance, an instantiation of the object INTEGER is created with the value three. INTEGER literals can be of two types: decimal and octal. Octal literals are preceeded with a '#' sign. INTEGER literals may be negative.

There are several other types of literals (STRING, VECTOR, and WORDVEC are examples) that will be discussed when their particular class is introduced.

7.4 Control Structures

a) Do statement

do <expr> (...)	the (...) is evaluated <expr> times (the <expr> is evaluated once)
do 5 (...)	
do (...)	do 1 is the default

b) Repeat statement

repeat (...)	repeats until a done or return statement is executed

c) Done statement

done with <expr>	terminates a loop with the value <expr>
done with 5	
done	*with nil* is the default -- the object NIL is discussed later

d) Return Statement

Return <expr>	returns from an invocation with <expr> as the reply
Return 3 + 4	replies with the object 7

[6] BLISS also returns the right hand side, but Planner (a language incorporating actors) returns the left hand side of an assignment. There are arguments supporting both views.

7.5 Conditionals

The syntax for a conditional statement is

```
( <if clause> then ( <then clause> ) <else clause> )
```

The <if clause> is simply an expression. If the expression returns FALSE, then the <else clause> is executed. Otherwise, the <then clause> is executed, and the <else clause> is skipped. Conditionals return a value, and may be nested; the statement

```
a := ((3 > 5) then (5) (4 < 6) then (7) 4)
```

assigns 7 to a (the parentheses are necessary in this case for a correct parse of the statement).

The conditional statement lends itself well to chaining. A construction that is common in Cola programs is[7]

```
( see a then ( . . . )
  see b then ( . . . )
  see c then ( . . . )
  return <var>
 )
```

In this statement, the <else clause> of the first conditional is

```
see b then ( . . . )
see c then ( . . . )
return <var>
```

Notice that the nested parentheses can be omitted; the fully parenthesized version would be

```
(see a then ( . . . )
        (see b then ( . . . )
                (see c then ( . . . )
                        return <var>
                )
        )
)
```

As another example:

```
see a
```

is equivalent to

```
(peek = a then (get; return true) return false)
```

7.6 Integer Expressions

Integer expressions in Cola are similar to those in other programming languages. The operations allowed are +, -, *, / (integer division), *mod*, =, <, >, and ↑ (integer exponentiation).

[7] See is a predefined object used for receiving messages; see chapter 6.

Due to the paradigm of objects sending and receiving messages (see chapter 6 for further elaboration), evaluation is right to left with no precedence. The statement

```
3 + 9 / 6
```

sends the message '+ 9 / 6' to 3. This results in the message '/ 6' being sent to the INTEGER 9, which returns the INTEGER 1, that gets added to 3, resulting in the INTEGER 4. To change the order of evaluation, use parentheses. Hence, for left to right evaluation of the above statement, execute

```
(3 + 9) / 6
```

instead (which returns the INTEGER 2).

7.7 Input/Output

Some of the input/output facilities of Cola have already been discussed. When an object followed by a message is typed on the keyboard, the object picks up the message by executing ':', *get*, *peek*, or *see*. When the object returns a reply to the top level, the reply is printed. Terminals (and more generally files) are objects that have useful side effects when sent messages (for a more detailed explanation of files and terminals, see sections 8.12 and 8.13).

To request input from the keyboard, use the statement

```
tty read
```

This returns a STRING consisting of the characters typed on the keyboard ending with the line terminator. For single character input, use

```
tty getchar
```

This returns a STRING instance whose length is 1 containing the next character typed on the keyboard[8].

The message

```
output
```

is a general message that can be sent to any object[9]. For instance, if "abc" is an instance of INTEGER with value 17 and "crlf" is a STRING composed of a carriage return and a linefeed and + is the concatenation operator for STRINGS (see section 8.7), then

```
tty write 'The answer is ' +  abc output + crlf;
```

prints

```
The answer is 17
```

The following class makes output much easier (see section 8.3 for more information on OBJECT):

[8] Since this object temporarily enables single character input (rather than the normal line-oriented input), there can be unexpected interactions with the normal buffered line editing (back space or line cancel) provided by the operating system.

[9] User-defined classes should reply to *output* in some meaningful way by returning an appropriate STRING.

```
    type := object "/ /
            ( repeat ( tty write : output; see , then ( )  return)
             )";
```

Thus, the above statement can be rewritten:

```
    type 'The answer is ', abc, crlf;
```

7.8 Delimiters

There are several different kinds of delimiters, depending on the context. Hopefully the following explanation makes it clear where a particular delimiter applies.

Delimiter	Application
<space>	delimits atoms, but is otherwise ignored
<comma>	delimits atoms; returns NIL if invoked
<period>	delimits atoms; returns NIL if invoked
(	treated like a space; Cola matches left and right parentheses
<carriage return>	treated like a space
;	marks the end of the message (*get*, *peek*, *see*, or : cannot read past a semicolon)
)	treated like a semicolon, except Cola checks for balanced parentheses
<line feed>	treated like a semicolon, used for an end-of-command when entering commands from the keyboard
", '	enclose STRINGS, see section 8.7
[ ]	enclose WORDVECS, see section 8.8
{ }	enclose VECTORS, see section 8.9

7.9 Returning a Value

The code for a class can stop executing in one of three ways: returning a value, returning NIL, or ending with an object with no message.

A value is returned by using the *return* command. This command sends the rest of the message to the object that is returned.

If a class returns NIL (a predefined object; see section 8.6), then the message is forwarded to the class' superclass. Returning NIL is equivalent to not recognizing the message.

If the evaluation of the code in a class finally results in an object with no corresponding message, then the same thing happens as if that object was returned by the class, except in one instance. If a class (call it "a") was on the right side of an assignment statement, and that class recognized part of the message and then returned another object (call it "b"), then "b" would be assigned to the atom on the left side of the assignment statement. However, if "a" merely ended with the object "b" (i.e., did not return the object), then "b" would be sent the rest of the message, and the object that "b" returned (call it "c") would be assigned to the atom. See the PROTECTEDCAPA class in Appendix II for an example.

This distinction is a small one, and is not important for most applications. However, it can be very useful at times to RETURN a value and at other times not to RETURN a value.

7.10 Defining Subclasses and Instances

In Simula, classes are defined statically, at compile time, and instances are instantiated dynamically, at runtime, using information provided by the compiler. In Smalltalk [Goldberg 78], classes and instances are defined interactively, although there is no facility for initializing at definition time the instance and class variables residing in a subclass (since values of instance variables do not reside in the subclass). Cola simplifies this process by giving the class complete control over the creation of subclasses and instances.

To create an instance of a class, the class (call it "a") executes the object

```
    defineinstance
```

This causes the system to create an instance (call it "b") of the current class ("a"). The system then sends the rest of the original message to the newly created instance ("b"). Since "b" contains no code, the message is forwarded to "a". The class knows that a new instance is executing because the object

```
    newinstance
```

now returns TRUE (at any other time this object returns FALSE). The class "a" then initializes the instance variables (which reside, of course, in "b"), using the message, and returns. An instance of "a" has thus been created and initialized. See the appendices for examples.

To create a subclass of a class, the class (again called "a") sends the object

```
    definesubclass
```

a message consisting of a STRING in the following form

```
"<temporary variables> / <instance variables> / <class variables>
                    ( <body> ) "
```

where the (optional) variable names are separated by spaces and the slashes and parentheses are required. This causes the system to create a subclass (call it "c") of "a" with the specified variables and code. The system then sends the rest of the original message to the newly created subclass ("c"). The subclass "c" knows that it has just been created, since the object

```
classdefined
```

now returns. "c" initializes its class variables using this message and returns TRUE. A subclass of "a" has thus been created and initialized.

8. Predefined Classes

Only a few of the many predefined classes have been mentioned so far. A discussion of the more important of them follows.

8.1 Atom

Atoms have a *name* and a *value* and are equivalent to identifiers in other languages. The name of an atom consists of a letter, an '&', or a '$', followed by any number of letters or digits or '&' or '$'. Since the name is a STRING, there is no limit to the number of characters in a name, and all are significant. Case folding is performed on the name. Atoms respond to the messages

name	returns the STRING that is the atom's name
: =	assigns a new value to the atom[10]
is <object>	returns TRUE if the superclass of the atom's value is <object>, otherwise returns FALSE
is ?	returns the name of the superclass of the atom's value
anything else	returns the value of the atom

8.2 Integer

The class INTEGER is a subclass of the class OBJECT. The messages that this class accepts are given in section 7.6. INTEGERS also accept the message *output*, and respond with a STRING of the value in decimal. INTEGERS have a range of [-32768, 32767]. Although INTEGER literals can be either octal or decimal, the internal representation is the same.

8.3 Object

The class OBJECT is a superclass of every other Cola object (see chapter 3). OBJECT returns FALSE when it receives the message *Null*. OBJECT responds to everything else with SELF.

[10] If the new value is a Cola class, then NIL is returned, otherwise the new value is returned. This semantics allows expressions such as a: = b: = c, which assigns c to a and b.

8.4 False

FALSE is a subclass of the class OBJECT. The message that this class accepts are *and*, *or*, *xor*, *eqv*, *not*, and *then*.

8.5 True

TRUE is a subclass of the class OBJECT. TRUE accepts the messages *xor, not, or, and, eqv,* and *then*.

8.6 Nil

NIL is the empty value in Cola. NIL returns TRUE when given the message *null*. NIL prints out nothing when sent the message *output*.

8.7 String

STRINGS enable the user to manipulate character data. STRINGS are declared either with the statement[11]

```
    a := string 15
```

which defines "a" to be a STRING of length 15 , initialized to NIL characters, or the statement

```
    b := 'abc'
```

which defines "b" to be a STRING of length 3, initialized to 'abc'. STRINGS can be delimited by double quotes, allowing single quotes to appear in the STRING, or by single quotes, allowing double quotes to appear in the STRING. Single and double quotes may not appear in the same STRING. STRINGS can be concatenated using the ' + ' operator; the statements

```
    a := 'abc';
    b := 'def';
    c := b + a;
```

result in a STRING of length 6, with the characters 'abcdef'. Substring selection is also possible:

```
    b[1 to 2] := c[2 to 3]
```

results in the STRING 'bcf'. As many characters as are needed to fill the slice are removed from "c"; if "c" is not long enough, then only the characters in "c" are transferred. Other operations on STRINGS include lexical ordering (> and <)[12], equivalence (=), *value* (returns the ASCII value of the first character in the STRING), and *length* (<string> *length*). Notice that the first character in <string> is

[11]Since STRINGS can be expanded (using the ' + ' operator), this construct is not really necessary, but is useful when initializing STRINGS containing control codes. If the length is not specified, it is assumed to be 0.

[12]A STRING (or slice) of length 1 can also be compared with an INTEGER, but not vice versa.

<string>[1]. <string>[*] is equivalent to <string>[1 to <string> length]; <string>[3 to *] is equivalent to <string>[3 to <string> length]. STRINGS have no maximum length (within the bounds of available memory restrictions, of course). If the message *eval* is sent to a STRING, then the STRING is evaluated in the current context. Thus the following command

```
        'c := 3' eval;
```

is equivalent to

```
        c := 3;
```

See section 9.7 for another example of *eval*.

8.8 Wordvec

WORDVECS enable the user to handle vectors of INTEGERS. To initialize a WORDVEC, use either of the statements[13]

```
        <var> := wordvec <size>
```

or

```
        <var> := [ <integer> <integer> . . . <integer> ]
```

The statement

```
        q := [ 10 20 30 60 70 #40 #60 ]
```

creates a WORDVEC of size 7 with the specified contents and assigns it to "q". Selections (including *) are used to access part of a WORDVEC, analogous to STRINGS. The operation *length*, as well as concatenation, also applies analogously.

8.9 Vector

VECTORS are similar to WORDVECS and STRINGS, in that all are linear, unbounded data structures allowing selection and concatenation. However, whereas WORDVECS are composed of INTEGERS, and STRINGS are composed of characters, VECTORS are composed of arbitrary Cola objects. To initialize a VECTOR, use either

```
        <var> := vector <size>
```

or

```
        <var> := { <object> <object> . . . <object> }
```

Selections (including *all*) are used to access part of a VECTOR, analogously to STRINGS and WORDVECS. VECTORS also respond to the message *length* and concatenation. Any Cola object can be stored into a single element of a VECTOR, and a VECTOR slice can be stored as an element of another VECTOR.

---

[13] If the <size> is omitted, it is assumed to be 0.

8.10 Capa

The class CAPA includes several operations that apply to all capabilities [Cohen 76]. The syntax for these operations is similar to record accessing in Pascal (A and C denote instances of the class CAPA, and x and y are INTEGERS):

C . data[x to y]	access the data part -- return a WORDVEC
C . data length	return an INTEGER specifying the length of the data part
C . clist[x to y]	access the CAPA part -- return a VECTOR of capabilities
C . clist length	return an INTEGER specifying the length of the clist
C . index	return an INTEGER specifying the slot in Cola's clist (see section 2.1) where this CAPA resides (used in KCall (see section 8.14))
C . clist[x to y] vacate	remove the CAPAs in the selected slots in the clist
C = A	returns TRUE if the CAPAs are identically the same (i.e., refer to the same object), otherwise returns FALSE

In addition to operations that apply to all CAPAS, there are operations that apply only to those CAPAS of a certain type. Each type in Hydra is represented by a subclass of the class CAPA in Cola. The type specific operations are enumerated below. Note that every capability in the user's universe is represented by an instance of one of the subclasses of the class CAPA.

To create a new CAPA, execute

```
capa <type>
```

which returns a new CAPA of the indicated type.

8.11 Catalogue

This is a user-defined type (rather than a kernel type), and the operations are performed by the Catalogue Subsystem [Almes 78, Wulf 81]. The valid operations are (S is a STRING instance, and C is an instance of the class CATALOGUE, a subclass of the class CAPA):

C[S]	look up entry S in CATALOGUE C -- return a CAPA of the correct type
C[S] . capa	same as C[S]
C[S] . protect	return TRUE if the protect bit was set for this entry, otherwise return FALSE

(The above expressions can be used as the left side of an assignment statement.)

C[S] vacate	remove entry S in CATALOGUE C

C is catalogue	returns TRUE

8.12 File

FILES enable the user to do input/output with objects supported by the Hydra File System [Reiner 77]. FILES are regarded as sequential streams of characters, and can only be read or written (but not simultaneously). Hence, no direct access to FILES is provided.

To initialize a FILE, use

```
    <var> := file input <capa>
```

or

```
    <var> := file output <capa>
```

that initializes <var> to be either an input or an output FILE connected to a Hydra Superfile referenced by <capa>. As an example, the statement

```
    MyFile := file input Pub["Snodgrass"]["TryFile"];
```

creates a new FILE instance, called "MyFile", which can perform input on the Hydra Superfile called TryFile ("Pub" is a predefined CATALOGUE).

The operations that are possible on FILES are (F is a FILE instance, and S is a STRING):

F close	closes the FILE, so that it is impossible to read from it or write to it until it is initialized again
F opened	returns FALSE if the FILE is closed, otherwise returns TRUE
F input	returns TRUE if the FILE is an input FILE, otherwise returns FALSE
F output	returns TRUE if the FILE is an output FILE, otherwise returns FALSE
F read	if F is an input FILE, returns a STRING delimited by an escape from the FILE (Line separators are converted to a single carriage return, and any linefeed that may be present after a carriage return is ignored.). Escapes are converted to linefeeds on input. If there are no more characters in the FILE, a STRING consisting of a single linefeed is returned.
F getchar	if F is an input FILE, returns a STRING of length 1 containing the next character in the FILE
F write S	if F is an output FILE, writes the STRING S onto the FILE
F eof	if F is an input FILE, returns TRUE if the entire FILE has been read, otherwise return FALSE; if F is an output FILE, always returns TRUE

8.13 Terminal I/O

A terminal can be viewed as a type of FILE (see section 8.12). However, terminals are different from FILES in several ways:

1. terminals can perform input and output simultaneously;
2. terminals should respond to special characters (typically, line editing commands) that cause the "input half" to modify the "output half";
3. terminals have special characters associated with them which can signal a running process; and
4. terminals have echoing characteristics that require special handling.

To initialize a 'tty-file', execute

```
    <var> := file terminal <capa>
```

where <capa> is a port CAPA (a type provided by the kernel) [Cohen 76] with the first two channels (channels 0 and 1) connected to a terminal. TTY-files can respond to all messages that normal FILES do. Note that since tty-files can both input and output, they can accept the messages *read, getchar*, and *write*, and will respond to *input*, *output*, and *eof* with TRUE. <escape> characters are illegal in terminal input; use linefeed instead.

Since tty-files need to go into single character mode to do *getchar*'s, erratic behavior may occur if line editing commands are typed while executing *getchar*.

TTY-files can also respond to the following messages (where T is a tty-file):

T prompt	equivalent to *T read*, except that the prompt (the level number followed by '>') is printed at the beginning of each line (after each carriage return) (this operation is used in DRIVER (see section 9.7))
T is terminal	returns TRUE
T flush	Causes the characters in the internal buffer associated with the tty-file to be printed immediately (the buffer is flushed automatically if a *T read* or *T getchar* is executed)

There is one predefined tty-file, called *tty*, which should suffice for performing input and output from the terminal.

8.14 Kernel Call

Kernel calls (K-calls) [Cohen 76] are implemented in a very primitive fashion. The syntax is

```
Kcall <arg1>, <arg2>, . . . . , <argn>;
```

or

```
(Kcall <arg1>, <arg2>, . . . . , <argn>)
```

where $\langle arg_i \rangle$ is either a CAPA instance, an INTEGER, or a selection of a WORDVEC[14]. <arg1> must be an INTEGER; it specifies the kcall index (i.e., which system call to execute). The semantics of the rest of the arguments depends on their type:

INTEGER	*simply pushed on the stack (i.e., call by value)*
CAPA	*index is pushed on the stack (i.e., call by reference)*
WORDVEC	*memory address is pushed on the stack (i.e., call by reference) Actually, the contents of the* WORDVEC *are copied to the stack page, and a pointer to this block is pushed onto the stack. When the* KCALL *returns, the block is copied back into the* WORDVEC. *(Note that a selection will give unexpected results if the* WORDVEC *is being written into by the kernel during the* KCALL; *in that case a complete* WORDVEC *should be given.)*

No type-checking is done on the arguments -- if there are invalid arguments, the KCALL will signal an error, which is printed out. The result of the kernel call (an INTEGER) will be replied by the KCALL object.

[14] A selection is a subscripted WORDVEC of more than one element, since a single element of a WORDVEC is an INTEGER.

9. Utility objects

Several objects perform tasks which are useful to Cola users. Some of these objects and their functions are given below. Examples of most of these objects can be found in the appendices.

9.1 At(@)

The '@' operator is a dereferencing operator. If the message sent to @ is <string>, then the atom whose name is <string> is invoked. If the message is <atom>, then the value of <atom> is invoked.

9.2 Attach

This utility "attaches" a Cola class (its second argument) to a predefined class (its first argument) provided by the Cola kernel. If the predefined class code does not recognize a message, then the Cola class attached to it is sent the message. If ATTACH is only given one argument (a predefined class), then the Cola class that is attached to the class is "unattached". This utility is provided to facilitate the construction of the Cola environment from the bare kernel, and should only be used by the maintainer of the system.

9.3 ClassDefined

This object returns TRUE if this subclass has just been created by executing DEFINESUBCLASS; otherwise the object returns FALSE (see section 7.10).

9.4 ClassInvoked

This object returns TRUE if the class itself was sent a message, rather than an instance of the class (the same code is executed, in either case). Usually this boolean is tested before creating a new subclass or instance of this class (see the appendices for examples).

9.5 DefineInstance

This object creates a new instance of the class that is currently executing, and sends the rest of the original message to the new instance. When the new instance is executing, the object NEWINSTANCE returns TRUE (and CLASSINVOKED returns FALSE).

9.6 DefineSubClass

This object creates a new subclass of the class that is currently executing, using the message (a STRING) as code for the new subclass. The message must be of the form

```
"<temp vars> / <instance vars> / <class vars> ( <code> )"
```

where the variables are separated by spaces, and the slashes and parentheses are required (see section 7.10).

9.7 Driver

DRIVER adds another level to the context stack, which begins with an initial invocation of DRIVER. The definition of DRIVER is

```
Driver := object "// (repeat (tty write
                tty prompt eval output; cr))";
```

It may be instructive to compare this code for this object with the sequence of tasks listed in the beginning of chapter 7. (CR is an object that prints a carriage return-line feed on the terminal.)

9.8 Instances

This object accepts a class as a message, and returns a VECTOR containing as elements all the instances of a given class. This utility is useful when one wants to send a message to all the instances of a class. The first element of the VECTOR contains the instance last created.

It is interesting to consider what happens when the following statement is executed:

```
a := instances vector
```

First, an empty VECTOR is created. Then, the VECTOR instances are placed, one by one, in the VECTOR that was just created. Since the first element of the VECTOR contains the instance last created, this element is the VECTOR itself!

9.9 NewInstance

This object returns TRUE if this instance has just been created by executing DEFINEINSTANCE; otherwise the object returns FALSE (see section 7.10).

9.10 Self

SELF refers to the object that originally received the message. This object is necessary to implement shadowing (see section 3.5).

9.11 Show

SHOW accepts a class as a message, and returns the code of that class as a STRING[15].

9.12 Six12

SIX12 invokes Six12, the BLISS/11 debugger. Exiting the debugger via the Six12 command 'go<cr>' returns the atom NIL; exiting via the Six12 command 'return <expr><cr>' returns <expr>.

9.13 SubClasses

This object accepts a class as a message, and returns a VECTOR containing all the subclasses of the given class. This utility is useful when one wants to send a message to all the subclasses of a class. The first element of the VECTOR contains the last subclass that was created.

9.14 Super

SUPER refers to the superclass of the currently executing class or instance. This object is useful if an object wants to send a message to its superclass (see section 3.5).

[15] This utility may be viewed as the inverse of definesubclass, although this is an oversimplification.

Bibliography

[Almes 78] G. Almes and G. Robertson.
An Extensible File System for Hydra.
Technical Report, Carnegie-Mellon University, Computer Science Department, February, 1978.
Available as CMU-CS-78-102.

[Alsberg 71] P. Alsberg.
OSL/2, An Operating System Language.
PhD thesis, Center for Advanced Computation, University of Illinois at Urbana-Champaign, 1971.

[Birtwistle 73] G.M. Birtwistle, O-J Dahl, B. Myhrtag and K. Nygaard.
Simula Begin.
Auerbach Publishers, Inc., Philadephia, PA, 1973.

[Bobrow 76] D.G. Bobrow and T. Winograd.
An Overview of KRL, A Knowledge Representation Language.
Technical Report, Xerox PARC, July, 1976.
Available as CSL-76-4.

[Bourne 78] S.R. Bourne.
The Unix Shell.
The Bell System Technical Journal 57(6, Part 2):1971-1990, July-August, 1978.

[Brunt 76] R.F. Brunt and D.E. Tuffs.
A User-Oriented Approach to Control Languages.
Software--Practice and Experience 6:93-108, 1976.

[Cohen 75] E. Cohen and D. Jefferson.
Protection in the Hydra Operating System.
In *Fifth Symposium on Operating System Principles*, pages 141-16. ACM, Austin, TX, November, 1975.

[Cohen 76] E. Cohen, et al.
Hydra Kernel Reference Manual.
Technical Report, Carnegie-Mellon University, Computer Science Department, November, 1976.

[Cowan 75] R.M. Cowan.
Burroughs B6700/B7700 Work Flow Language.
In C. Unger (editor), *Command Languages*, pages 153-171. North Holland, 1975.

[Ellis 80] J.R. Ellis.
A LISP Shell.
SIGPlan Notices 15(5):24-34, May, 1980.

[Fahlman 79] S. E. Fahlman.
Netl, A System for Representing and Using Real-World Knowledge.
MIT Press, Cambridge, MA, 1979.

[Feiler 80] P.H. Feiler and R. Medina-Mora.
An Incremental Programming Environment.
Technical Report, Carnegie-Mellon University, Computer Science Department, April, 1980.
Available as CMU-CS-80-126.

[Goldberg 78] A. Goldberg and A.C. Kay, eds.
Smalltalk-72 Instruction Manual
Xerox PARC, Palo Alto, CA, 1978.

[Goldberg 79] A. Goldberg and D. Robson.
A Metaphor for User Interface Design.
In *Proceedings of the 12th Hawaii International Conference on System Science*, pages 148-157. 1979.

[Grief 75] I. Grief and C. Hewitt.
Actor Semantics of PLANNER-73.
In *Proceedings of the Second Conference on Principle of Programming Languages*. January, 1975.

[Habermann 80] A.N. Habermann.
An Overview of the Gandalf Project.
In *CMU Computer Science Research Review 1978-1979*. Carnegie-Mellon University, Computer Science Department, 1980.

[Hewitt 77] C. Hewitt.
Viewing Control Structures as Patterns of Passing Messages.
Artificial Intelligence 8:323-364, 1977.

[IBM 80] IBM.
IBM Virtual Machine Facility/370: CMS Command and Macro Reference Manual
1980.
Order GC20-1818.

[Ingalls 78] D. Ingalls.
The Smalltalk-76 Programming System: Design and Implementation.
In *Proceedings of the Fifth Conference on Principles of Programming Languages*, pages 9-16. ACM, January, 1978.

[Jones 77] A.K. Jones.
The Narrowing Gap Between Language Systems and Operating Systems.
In *Proceedings of the IFIP Conference*. 1977.

[Jones 79] A.K. Jones, R. Chansler, Jr., I. Durham, K. Schwans and S. Vegdahl.
StarOS, a Multiprocessor Operating System for the Support of Task Forces.
In *Proceedings of the Seventh Symposium on Operating System Principles*, pages 117-127. Pacific Grove, CA, December, 1979.

[Kay 77] A.C. Kay and A. Goldberg.
Personal Dynamic Media.
Computer 10(3):31-41, March, 1977.

[Kay 80] A.C. Kay.
Personal communication.
February, 1980.

[Kernighan 79] B.W. Kernighan and J.R. Mashey.
The Unix Programming Environment.
Software--Practice and Experience 9:1-15, 1979.

[Lampson 77] B.W. Lampson, J.J. Horning, R.L. Lampson, J.G. Mitchell and G.L. Popek.
Report on the Programming Language Euclid.
SIGPlan Notices 12(2), February, 1977.

[Lampson 79] B.W. Lampson and R. Sproull.
An Open Operating System for a Single-User Machine.
In *Proceedings of the Seventh Symposium on Operating System Principles*, pages 98-105. Association for Computing Machinery, Pacific Grove, CA, December, 1979.

[Lauer 79] H.C. Lauer and R.M. Needham.
On the Duality of Operating System Structures.
In *Proceedings of the Second International Symposium on Operating Systems*. IRIA, October, 1979.
Reprinted in Operating Systems Review, 13, 2, April, 1979, pp. 3-19.

[Lauesen 73] S. Lauesen.
Program Control of Operating Systems.
BIT 13:323-337, 1973.

[Levine 80] J. Levine.
Why a Lisp-based command language?.
SIGPlan Notices 15(5):49-53, May, 1980.

[Liskov 77] B. Liskov, A. Snyder, R. Atkinson and C. Schaffert.
Abstraction Mechanisms in CLU.
Communications of the ACM 20(8):564-576, August, 1977.

[Mealy 66] G.H.Mealy.
The functional structure of OS/360.
IBM Systems Journal 5(2), 1966.

[Minsky 75] M. Minsky.
A framework for representing knowledge.
In P. Winston (editor), *The Psychology of Computer Vision*, pages 211-277. McGraw-Hill, New York, 1975.

[Ostreicher 67] M.D. Ostreicher, M.J. Bailey and J.I. Strauss.
GEORGE 3--A General Purpose Timesharing and Operating System.
Communications of the ACM 10(11):685-693, November, 1967.

[Ousterhout 80] J. Ousterhout, D. Scelza and P. Sindu.
Medusa: An Experiment in Distributed Operating System Structure.
Communications of the ACM 23(2):92-104, February, 1980.

[Redell 80] D. Redell, Y. Dalal, T. Horsley, H. Lauer, W. Lynch, P. McJones, H. Murray and S. Purcell.
Pilot: An Operating System for a Personal Computer.
Communications of the ACM 23(2):81-91, February, 1980.

[Reiner 77] A. Reiner and J. Newcomer, eds.
The Hydra Users Manual
Carnegie-Mellon University, Computer Science Department, 1977.

[Sandewall 78] E. Sandewall.
Programming in the Interactive Environment:The LISP Experience.
Computing Surveys 10(1):35-72, March, 1978.

[Shaw 80] M. Shaw.
The Impact of Abstraction Concerns on Modern Programming Languages.
Proceedings of the IEEE 9(68), September, 1980.

[Shoch 79] J.F. Shoch.
An Overview of the Programming Language Smalltalk-72.
SIGPlan Notices 14(9):64-73, September, 1979.

[Smith 75] D. Smith.
Pygmalion: A Creative Programming Environment.
PhD thesis, Stanford Artificial Intelligence Laboratory, Stanford Computer Science Department, June, 1975.
Available as STAN-CS-75-499.

[Teitelbaum 79] T. Teitelbaum.
The Cornell Program Synthesizer: A Microcomputer Implementation of PL/CS.
Technical Report, Cornell University, July, 1979.
Available as TR 79-370.

[Teitelman 78] W. Teitelman.
INTERLISP Reference Manual
Xerox PARC, 1978.

[Treu 75] S. Treu.
Interactive Command Language Design Based on Required Mental Work.
International Journal of Man-Machine Studies 7:135-149, 1975.

[Warren 79] S. Warren and D. Abbe.
Rosetta Smalltalk: A Conversational, Extensible Microcomputer Language.
In *Proceedings of the Second Symposium on Small Systems.* ACM SIGPC, Dallas, TX, October, 1979.

[Wilkes 79] M.V. Wilkes and R.M. Needham.
The Cambridge CAP Computer and Its Operating System.
Elsevier-North Holland, 1979.

[Winograd 75] T. Winograd.
Breaking the complexity barier (again).
SIGPlan Notices 10(1):13-30, January, 1975.

[Wulf 72]	W.A. Wulf and C.G. Bell. C.mmp--a multi-mini-processor. In *Proceedings of the 1972 Fall Joint Computer Conference*, pages 765-777. AFIPS Press, 1972.
[Wulf 75]	W.A. Wulf, R.K. Johnsson, C.B. Weinstock, S.D. Hobbs and C.M. Geschke. *The Design of an Optimizing Compiler.* Elsevier North-Holland, Inc., New York, N.Y., 1975.
[Wulf 76]	W.A. Wulf, R. London and M. Shaw. *Abstraction and Verification in Alphard, Introduction to Language and Methodology.* Technical Report, Carnegie-Mellon University, Computer Science Department, June, 1976.
[Wulf 81]	W.A. Wulf, R. Levin and S.P. Harbison. *C.mmp/Hydra: An Experimental Computer System.* McGraw-Hill, New York, 1981.
[Yonezawa 77]	A. Yonezawa and C. Hewitt. Modeling Distributed Systems. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 370-376. ACM, MIT, August, 1977.

Part 3: Examples

There are three examples given in this part. The RECTANGLE class illustrates the basic concepts of objects and instances, including sending messages, creating instances, and using class and instance variables. The PROTECTEDCAPA class illustrates how one may define classes that are versions of previously defined data types with added functionality. The BOOTSTRAP class is an example of an object that doesn't have any instances. This type of class is the Cola equivalent to procedures or macros in other command languages.

I. Rectangle Class

This object demonstrates how one defines an object and its code and how one uses instance and class variables. First the class definition will be given, then examples of sending messages to the class, followed by an explanation of these examples.

```
rectangle := object "/ positionx positiony / mysize
                (see newsize then (return mysize := :)
                 newinstance then (positionx := : ;
                                        positiony := : ;
                                        return self)
                 classinvoked then (defineinstance)
                 see xposition then (return positionx)
                 see yposition then (return positiony)
                 see size then (return mysize)
                 see output then (return '(' +
                                 (xposition output) +
                                 ', ' +
                                 (yposition output) +
                                 ')' )
                )";
```

Examples (Cola's responses are in italics):

```
12>rectangle newsize 3
3
12>a := rectangle 3 4
(3, 4)
12>b := rectangle 6 7
(6, 7)
12>( (a size = b size) then ("same size") "not same size")
same size
12>a xposition + b yposition
10
12>
```

The class RECTANGLE is a subclass of the class OBJECT (a predefined class in Cola). RECTANGLE has two instance variables, positionx and positiony, and one class variable, size. Several messages may be sent to RECTANGLE. The message

```
    newsize <integer>
```

sets the class variable size to the value of the INTEGER given in the message. Since size is a class variable, all instances have the same value for size. The message

```
    rectangle  <integer> <integer>
```

causes CLASSINVOKED to be TRUE. RECTANGLE then makes a new instance of itself, sets NEWINSTANCE to TRUE, and executes RECTANGLE again. Since NEWINSTANCE is TRUE, positionx is set to the value of the first INTEGER, and positiony is set to the value of the second INTEGER. The instance that was just generated is then returned. The third assignment statement

```
    a := rectangle 3 4;
```

makes "a" an instance of RECTANGLE, with a position (3, 4). The fourth statement makes "b" another instance of RECTANGLE, with a different position: (6, 7). Notice that both "a" and "b" have size 3, since size is a class variable. The fifth statement outputs the text 'same size'. The last statement illustrates sending messages to instances. The instance "a" is sent the message 'xposition + ...'. RECTANGLE is invoked with the values of positionx, positiony, and size set to 3, 4, and 3, respectively. The condition *see xposition* is TRUE, so positionx, or 3, is returned. This instance (of the class INTEGER) grabs the ' + ' and evaluates the next part of the message. This causes "b" to be sent the message 'yposition'. RECTANGLE is invoked a second time, but with positionx, positiony, and size set to 6, 7, and 3, respectively. The condition *see yposition* is satisfied, so positiony, or 7, is returned. The INTEGER 3 adds 7 to itself and returns the INTEGER 10, which responds to the message *output* (in DRIVER) by returning the STRING '10'.

II. Protectedcapa Class

This example[16] illustrates how one may define classes that are versions of previously defined data types with added functionality. The data type extended in this example is the standard CAPA class, with the additional property that it can be "protected" by sending it the message *protect*. Once protected, an instance of this class cannot be accessed except to print its type and to be unprotected. An unprotected instance of this class acts exactly like a normal CAPA.

```
protectedcapa := object "/ mycapa protectbit /
                (ClassInvoked then (defineinstance)
                NewInstance then (mycapa := : ;
                                  protectbit := false ;
                                  return self )
                see protect then (protectbit := true)
                see unprotect then (protectbit := false)
                see output then (return 'protected capa')
                protectbit then (error 'capa is protected')
                      mycapa
                )";
```

Examples of the PROTECTEDCAPA in action[17]:

```
6>a := protectedcapa &SysDirectory
capability
6>a is protectedcapa
true
6>a output
protected capa
6>a protect
true
6>a isprotected
true
6>a["Public"]
Error: capa is protected
9>a unprotect
false
9>a["Public"]
capability
9>
```

Note that PROTECTEDCAPA ends with *mycapa*, rather than returning *mycapa*, in order to send *mycapa* the rest of the message (see section 7.9). The ERROR object prints the error string and invokes DRIVER, leaving the context where the error occurred on the stack.

[16]suggested by Joseph Newcomer

[17]&SysDirectory is a predefined CATALOGUE.

III. Bootstrap Class

This class is an example of an object which does not have any instances. BOOTSTRAP is sent either a FILE instance or a CAPA instance which refers to a FILE. It then reads the FILE, page by page (each page is delimited by an <escape> character), evaluating each page in turn. If the FILE contains class definitions, then these classes will be defined when BOOTSTRAP completes.

```
Bootstrap := object 'myfile astring //
   (myfile := ::
   (myfile is capa then (myfile := file input myfile));
   repeat (astring := myfile read;
          myfile eof then (myfile close;
                          return "Bootstrap completed")
                 astring eval; tty write "x"; tty flush
          )
   )'
```

To understand why this class is called BOOTSTRAP, assume that it was not yet defined, and suppose that a CAPA called "Startfilecapa" references a FILE containing

```
Bootstrap := object 'myfile ...
           .
           .
           .
   )';
Bootstrap a
<escape>
<other class definitions>
.
.
.
<escape>
```

Then the following commands would read in the FILE, defining BOOTSTRAP (as well as the other classes in the FILE) along the way:

```
a := file input Startfilecapa; a read eval
```