2-1994

Overview and Specification of the Built-In Types in Little Smalltalk

Gary T. Leavens
Iowa State University

Yoonsik Cheon
Iowa State University

Overview and Specification of the Built-In Types in Little Smalltalk

Gary T. Leavens and Yoonsik Cheon

TR #91-22a
October, 1991 (revised February 1994)

Keywords: Smalltalk, class, subclass, inheritance, instance, method, message passing, specification, subtype, pre-condition, post-condition.
1991 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications — Languages; F.3.1 [*Logics and Meaning of Programs*] Specifying and verifying and reasoning about programs — pre- and post-conditions, specification techniques.

Department of Computer Science
Atanasoff Hall
Iowa Sate University
Ames, Iowa 50011-1040, USA

Contents

1 Notation for Method Specifications **3**

2 Protocol for all Classes **4**
- 2.1 Creation of New Classes . . . 6
- 2.2 Subclass and Superclass Access . . . 6
- 2.3 Instance Method Access . . . 6
- 2.4 Method Creation and Editing . . . 7
- 2.5 Saving and Restoring Class Descriptions . . . 7
- 2.6 Instance Creation . . . 7

3 Protocol for all Objects **8**
- 3.1 Class access and Membership Tests . . . 8
- 3.2 Instance Method Testing . . . 8
- 3.3 Printing and Displaying . . . 8
- 3.4 Object Identity Comparisons . . . 9
- 3.5 Protocol for Mutable Objects . . . 9
  - 3.5.1 Copying . . . 9

4 Smalltalk **10**
- 4.1 Saving System State . . . 10
- 4.2 Terminal I/O . . . 10

5 Primitive Types **10**
- 5.1 Boolean . . . 10
  - 5.1.1 Conditionals . . . 10
  - 5.1.2 Logical connectives . . . 11
- 5.2 Symbol . . . 11
- 5.3 Magnitude . . . 11
- 5.4 Char . . . 12
- 5.5 Numbers . . . 12
  - 5.5.1 Arithmetic Operations . . . 12
  - 5.5.2 Integer Specific operations . . . 12

6 Collections **14**
- 6.1 Protocol for all Collections . . . 14
  - 6.1.1 Testing . . . 14
  - 6.1.2 Iteration . . . 15
  - 6.1.3 Conversions among Collection Types . . . 15
- 6.2 List . . . 15
  - 6.2.1 Adding and Removing Elements . . . 15
  - 6.2.2 Accessing Elements . . . 16
  - 6.2.3 Mapping and Filtering . . . 16
- 6.3 Set . . . 17
  - 6.3.1 Adding and Removing Elements . . . 17
  - 6.3.2 Iteration . . . 17
- 6.4 Array . . . 17
  - 6.4.1 Fetching and Storing . . . 18
  - 6.4.2 Iteration . . . 18
- 6.5 Dictionary . . . 19
- 6.6 String . . . 19

7 Control Structures **20**
7.1 Block 20
7.2 Interval 20

8 Files **21**

9 Guide to the Implementation **21**

This document informally specifies the behavior of most of the key data types in the Little Smalltalk system. This is done partly because the protocols differ from those found in Smalltalk-80[GR83] and also because for version three of Little Smalltalk, the documentation in [Bud87] is hopelessly out of date and the description in [Bud91] lacks detail. Since our specifications emphasize the client's point of view, many specialized or private methods of a class are suppressed. The system itself may be used to discover and examine the behavior of such methods.

In the first section we explain our notation for specifying methods. The next sections give type specifications for groups of related types. Section 2 describes protocol for all classes; i.e., the methods of the type Class. Section 3 gives protocol for all Smalltalk objects; i.e., the methods defined in the type Object, the ultimate supertype of all other types. In Section 4, the type Smalltalk is discussed, whose sole instance is used for saving the state of the interpreter, as well as for terminal I/O. Section 5 specifies methods for such primitive types as Boolean, Char, Number, and Symbol. The type String, along with other collection types (Array, List, etc.), is explained in Section 6. Smalltalk control structures (blocks) are specified in Section 7. The type File is described in Section 8. Section 9 presents a table that can be used to find methods in the implementation of Little Smalltalk.

1 Notation for Method Specifications

Specifications of methods are given informally, but in a stylized manner, following [LG86]. A specification consists of a *header*, a *requires clause*, and an *ensures clause*. The header describes the signature of the operation and gives formal names to the arguments and the result. These names are used in the requires and ensures clauses. The requires clause states a pre-condition on the operation's arguments; that is, a property of the arguments that the caller is expected to satisfy. The requires clause is omitted if the condition is vacuous (i.e., always true). The ensures clause states a post-condition on the results, described in terms of the arguments and the formal result identifier. The post-condition is a property of the result that the method establishes.

Consider the following simple example, the at: operation on arrays.

at: *self: Array, i: Integer* $\rightarrow$ *o: Object*

Requires: i is greater than 0, and less than or equal to the size of *self*.

Ensures: o is the i^{th} element of *self*.

The header says that at: is a message that can be sent to an instance of the class Array or a subtype[1] of Array. The receiver, the object to which the message is sent, is given the name *self*. The at: message also needs an additional argument, which must be an Integer (or a subtype); this additional argument is denoted by i. The object returned by the at: method must be an Object (or one of its subtypes, which is not restrictive), and this object is represented by o. The requires clause says that, the value of i must be a legal index into the array *self*. If this condition is true, the operation behaves as described in the ensures clause. Otherwise, the operation does not need to behave as specified in the ensures clause, and may instead do something else, such as printing an error message. The ensures clause says that the value returned is the i^{th} element of the receiver (*self*). This is a termination semantics, that is to say, the operation must not loop forever or encounter an error when the requires clause is satisfied. If the operation does not return, those cases will be made explicitly described.

A slightly different format is used for operations that mutate their arguments and for those that have no return value. For operations that mutate their arguments, a *modifies clause* is included to state which objects are allowed to be modified. Most operation specifications omit this clause, which means no objects are mutated by that operation. For some operations, especially those that mutate their arguments, the return value is irrelevant. Therefore for such operations, the arrow ($\rightarrow$) and the information about the result is omitted from the header; this is interpreted as follows: if the requires clause is satisfied, the receiving object itself will be returned. For such operations the post-condition is stated in an *effect clause* instead of

[1]A "subtype of Array" is a type whose instances behave like arrays in the sense that each instance of the subtype simulates some array. At the least, an object q simulates r if a sequence of message sends cumulating in a boolean or integer result would yield the same final result for both q and r. See [LW90] and [Lea91] for more details.

an ensures clause. Logically the meaning is the same, however such a post-condition only states side effects, because there is no return object to refer to.

In the specification of an operation that mutates its arguments, it is sometimes necessary to refer the value of an object in two different states; the states before and after the call. It is also necessary to refer the identity of the object (i.e., its address), that is to say, the object itself not its value. These distinctions are made by qualifying formal argument names. Consider, for example, an object named *self.* The value of this object before the call is denoted by **pre**(*self*) while the value after the call is represented by **post**(*self*). The object's identity is denoted by **obj**(*self*). Qualifications are often redundant. This leads us to adopt certain defaults depending on the context in which a name appears. In the modifies clause, one always refers to object identities, so the object qualification is the default. An unqualified formal argument name *arg* is, by default, qualified as **pre**(*arg*). A formal result name *res* is, by default, qualified as **post**(*res*). For example, consider the following description of the at:put: operation for arrays, which stores an object at a given index in the receiver.

at:put: *self: Array, i: Integer, o: Object*

Requires: *i* is greater than 0, and less than or equal to the size of *self.*

Modifies at most: *self.*

Effect: makes *o* the element of **post**(*self*) at index *i.*

The input formal *i* appearing in the requires and effect clauses, is a short form of **pre**(*i*), and the input formal *o* in the effect clause is short for **pre**(*o*). The receiver *self* in the modifies clause is also short for **obj**(*self*).

The major conceptual difference of our notation from Liskov and Guttag's is that we omit from a type's specification specifications for operations defined in the type's supertypes. For instance, since all types are subtypes of the type Object, all have a method class that returns the class of the receiving object. However this method is not repeatedly specified in each type, since repeating it would be redundant and uninformative. It is only specified in the type Object once and for all. Therefore one can think of subtyping as inheritance of *specifications* (as opposed to code inheritance, which is subclassing in Smalltalk). Because of this, it is worthwhile to refer to the specifications of supertypes if the desired operation seems to be missing.

A set of method specifications are collected together to give a specification of an abstract data type. An abstract data type, which will be called a *type* for short, is an abstraction of a set of Smalltalk classes characterized by their behavior. A type is usually implemented by a single Smalltalk class. However it may be implemented by several classes, as in the case of type Boolean which is implemented by three Smalltalk classes, Boolean, True, and False. Figure 1 shows the subtype relationships among the types specified in this paper. Figure 2 gives the actual class hierarchy of the Little Smalltalk system.

2 Protocol for all Classes

A class is a module that implements an abstract data type. The code and other information kept about each class definition is stored in the Smalltalk system as an object, called a *class object.* Since each class definition is represented by a single class object, classes and class objects are identified; hence class objects are often called *classes* for short. Each object in Smalltalk is an instance of a class; hence a class can also be thought of as a set (or class) of objects. Classes (i.e., class objects) are important in Smalltalk because they store Smalltalk code; thus a programmer must master the protocol of class objects to read and write code. Each class object in Smalltalk is named by a global variable that denotes it. The name of this global variable is the name of the class. In what follows we use these names to refer to particular class objects. For example the class Object is named by the global variable Object. Since classes are represented by class objects, these objects must also be instances of some class. That class is called Class. It is the protocol of the class Class that is described in this section.

Classes are related to one another by *subclass* (or inheritance) relationships. Each class is a *proper subclass* of exactly one other class, called its *proper superclass*, and may be a proper superclass of any number of other classes. A class, C, its proper superclass, its proper superclass's proper superclass, and so on are collectively called the *superclasses* (or ancestors) of C. A class C, its proper subclasses, its proper subclass's proper subclasses, and so on are collectively called the *subclasses* (or descendents) of C. See Figure 2 for the class

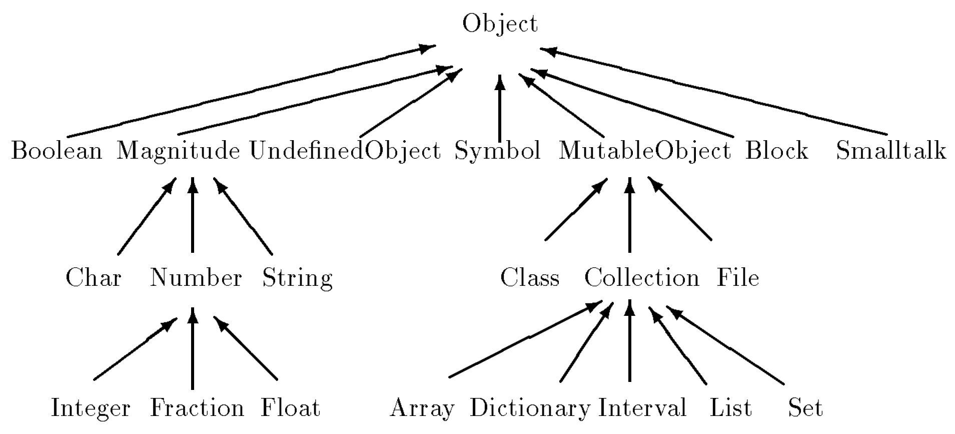

Figure 1: The hierarchy of types

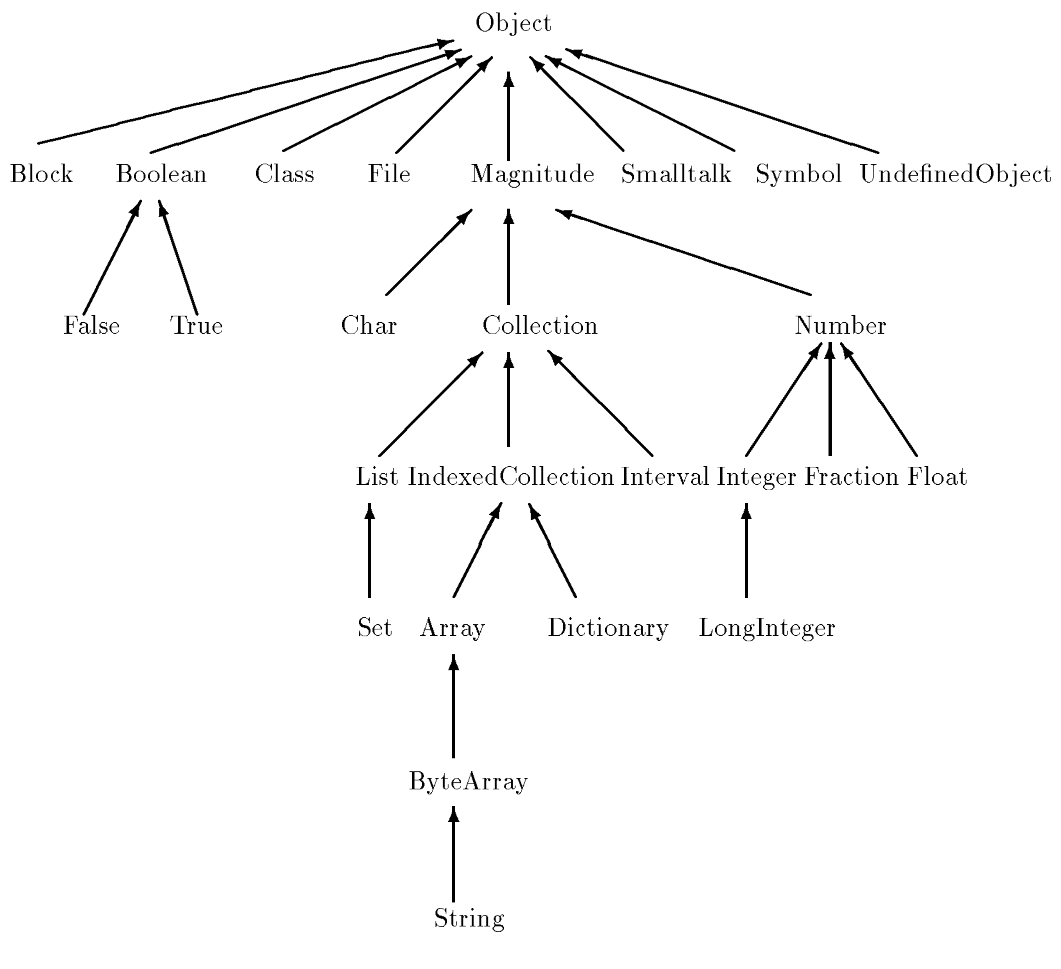

Figure 2: The class hierarchy of the Little Smalltalk

hierarchy of the Little Smalltalk. The operations described below allow one to create new classes, traverse the inheritance hierarchy, access the instance methods defined in a class or its ancestors, create and edit methods, save and restore one's work, and create instances of given class. One can also use the messages in Section 2.2 to traverse the hierarchy of classes.

2.1 Creation of New Classes

New classes can also be created using the methods described in Section 2.5.

addSubClass:instanceVariableNames: *self: Class*, *cn: Symbol*, *ivn: String*

Modifies at most: *self*, the global dictionary *symbols*.

Effect: creates a new class named *cn* as a proper subclass of *self* with its instance variables names those words appearing in *ivn*. (Words in *ivn* are separated by blanks.)

2.2 Subclass and Superclass Access

The methods described below allow one to traverse the class hierarchy.

superClass *self: Class* $\rightarrow$ *c: Class*

Ensures: *c* is the proper superclass of *self*.

subClasses *self: Class* $\rightarrow$ *subs: List*

Ensures: *subs* contains all of the proper subclasses of *self* without repetition and no other classes.

upSuperclassChain: *self: Class*, *b: Block*

Requires: *b* is a one-argument block that takes a class object as its argument.

Modifies at most: the objects modified by the execution of *b*.

Effect: invokes the block *b* with argument *self*, the proper superclass of *self*, and so on for each ancestor of *self* (in that order).

To better traverse the class hierarchy, one can add additional methods to the class Class. For example, the following method would invoke a block on each subclass of the receiver (in depth first order).

```
allSubclassesDo: aBlock
        aBlock value: self.
        self subClasses do: [ :c | c allSubclassesDo: aBlock ]
```

2.3 Instance Method Access

An instance is said to *respond to* a given message if its class defines or inherits a method for that message. This is no guarantee that something useful will happen when that message is sent to it; in some cases an error will be produced. See also messages for method creation and editing in Section 2.4 the methods in Section 3.2, and the methods in Section 5.2.

respondsTo *self: Class* $\rightarrow$ *d: Dictionary*

Ensures: *d* is a new dictionary that describes the messages to which instances of *self* respond. That is, *d* maps each method name (a Symbol) to the method object (a Method) that implements that method. Inherited methods are included.

methods *self: Class* $\rightarrow$ *d: Dictionary*

Ensures: *d* is a new dictionary that describes the messages that are defined in the class *self*. Inherited methods are not included.

viewMethod: *self: Class*, *s: Symbol*

Effect: If instances of *self* respond to the message *s*, then the name of the class where the instance method for *s* is defined and the code that implements that method are printed; otherwise an error message is printed.

methodNamed: *self: Class*, *s: Symbol* $\rightarrow$ *m: Object*

Ensures: If there is a method named *s* implemented in *self* or its superclasses, then *m* is the Method object that implements *s*; otherwise *m* is nil.

2.4 Method Creation and Editing

Method creation in Little Smalltalk is either done interactively (as described in this section) or through the use of external files and the methods described in Section 2.5. One can create a method interactively using the editor named by the global variable editor (which is "emacs" by default at ISU). Once editing is finished, it is compiled by the system into an executable code. If there are any errors, one gets a chance to correct them.

addMethod *self: Class*

Modifies at most: *self.*

Effect: allows you to add a new method to the class *self*, using the editor whose name is the value of the global variable editor.

readMethods *self: Class*

Modifies at most: *self.*

Effect: allows you to add several method to the class *self.* You are prompted as to whether you want to add a method or not, and if so you add it as in addMethod. When you are done, you are again asked whether you want to add a method or not, etc.

editMethod: *self: Class*, *s: Symbol*

Modifies at most: *self.*

Effect: if instances of *self* respond to the message *s*, then you are allowed to edit the method definition named *s*; otherwise an error message is printed.

2.5 Saving and Restoring Class Descriptions

Smalltalk code can be saved to and restored from the operating system's files using the methods described in this section. It is a good idea to save methods to files fairly often; it is especially important to save them before running any tests, because if the Smalltalk system gets into an inconsistent state, you may be forced to restart from files.

The "file out" methods write a description into a file, which can be read back in by the "filing in" methods. Among the methods listed below, some are actually those messages that should be sent to File objects (see Section 8). It is also possible to save the state of the entire system (see Section 4).

fileOut *self: Class*

Effect: write a description of *self* in a file named "*cn*.st", where *cn* is the name of the class *self.*

fileOutOn: *self: Class*, *f: File*

Modifies at most: *f.*

Effect: write a description of *self* into the file *f.*

fileOutMethodsOn: *self: Class*, *f: File*

Modifies at most: *f.*

Effect: write a description of all instance methods of the class *self* into the file *f.*

fileIn: *self: File, fn: String*

Modifies at most: *self*, the global dictionary *symbols*, and existing class objects.

Effect: reads class and/or method descriptions stored in the file named *fn*, and installs them in the system.

fileIn *self: File*

Modifies at most: *self*, the global dictionary *symbols*, and existing class objects.

Effect: reads class or method descriptions stored in the file *self*, and installs them in the system.

2.6 Instance Creation

Most objects (other than class objects) in Smalltalk are created (ultimately) by sending some class object a new message. However, arrays and strings are created with new: message (see Section 6).

new *self: Class* $\rightarrow$ *o: Object*

Requires: *self* is neither Array nor String.

Ensures: *o* is a new instance of the class *self*, and this new instance has, itself, also been sent the message new (unless *self* is Class, in which case the new class object is sent the message initialize instead).

new: *self: Class, i: Integer* $\rightarrow$ *o: Object*

Requires: *self* is either Array or String.

Ensures: *o* is a new instance of *self*, with size *i*.

3 Protocol for all Objects

The type Object is a supertype of all other types, just as the class Object is a superclass of all other classes. Its methods provide a consistent basic functionality and default behavior that is useful for debugging. With these methods one can observe all objects in the Little Smalltalk system, including class objects.

The methods of the type Object described below can be used to test the class of an object, test the instance methods to which an object responds, to print and display objects, and to compare the identity of objects. Also included below are methods of the type MutableObject, which is a fictitious[2] subtype of Object. A mutable object is one whose state can be changed. The methods of type MutableObject can be used to copy mutable objects.

3.1 Class access and Membership Tests

The proper way to test if an object is a member of a certain class is with the messages whose specification is given below. (There are also messages, such as isFraction and isInteger, that should be regarded as private to various classes.)

class *self: Object* $\rightarrow$ *c: Class*

Ensures: *c* is the class of *self*.

isMemberOf: *self: Object, c: Class* $\rightarrow$ *b: Boolean*

Ensures: *b* is *true* if and only if *c* is the class of *self*.

isKindOf: *self: Object, c: Class* $\rightarrow$ *b: Boolean*

Ensures: *b* is *true* if *c* is the class of *self* or a superclass of *self*; *b* is *false* if *c* is not an ancestor of the class of *self*. For example, `1 isKindOf: Object` is *true*, but `1 isKindOf: Boolean` is *false*.

3.2 Instance Method Testing

The messages to which an object responds can be discovered in two ways. One can use the respondsTo: message specified below, or one can first obtain the object's class and use the message respondsTo specified in Section 2.3. Note that the respondsTo: message can be sent to any object, while the respondsTo message are limited to class objects. Furthermore, these messages differ in the number of arguments and the kind of value returned. (The inverse of this kind of test is described in Section 5.2.)

respondsTo: *self: Object, s: Symbol* $\rightarrow$ *b: Boolean*

Ensures: *b* is *true* if *self* responds to the message *s*; otherwise *b* is *false*.

3.3 Printing and Displaying

The message printString is used in preparing formatted output. For debugging the other messages are more useful.

printString *self: Object* $\rightarrow$ *s: String*

[2] That is, MutableObject only exists for purposes of specification and there is no class called MutableObject in Little Smalltalk.

Ensures: s is a string describing *self*. The exact result depends on the class of *self* and its value. (By default, s is just the class name of *self*.)

print *self: Object*

Modifies at most: the standard output.

Effect: prints the string that results from sending *self* the message printString.

display *self: Object*

Modifies at most: the standard output.

Effect: prints a description of *self* on the standard output device. The exact result depends on the class of *self* and its value. (By default, this prints the class name in parentheses and the value of sending *self* the message printString.)

3.4 Object Identity Comparisons

Object identity is a useful concept only for mutable types (types whose instances have time-varying state) and should not be used for comparing instances of immutable types. The major exception to this rule is for testing to see if an object is defined or nil. In most circumstances one should use the = message to compare the abstract values of objects instead of comparing their identity (address). However, the specification for = cannot be given here, as it varies from type to type and is not meaningful for some types (e.g., Block).

== *self: Object, o: Object* $\rightarrow$ *b: Boolean*

Ensures: b is *true* if **obj**(*self*) is equal to **obj**(o); otherwise b is *false*. Hence, *true* is returned if and only if *self* and o are the same object.

$\sim\sim$ *self: Object, o: Object* $\rightarrow$ *b: Boolean*

Ensures: b is *false* if **obj**(*self*) is equal to **obj**(o); otherwise b is *true*. Hence, *true* is returned if and only if *self* and o are different objects.

isNil *self: Object* $\rightarrow$ *b: Boolean*

Ensures: b is *true* if and only if **obj**(*self*) is the object nil, which is the sole instance of the class UndefinedObject.

notNil *self: Object* $\rightarrow$ *b: Boolean*

Ensures: b is *false* if and only if **obj**(*self*) is equal to **obj**(nil).

3.5 Protocol for Mutable Objects

The type MutableObject is considered to be a supertype of all mutable types. The methods described below are implemented in class Object in Little Smalltalk, but do not work for objects of all types and hence cannot be specified as methods of type Object. The methods of Section 3.4 are also principally useful for mutable objects, but they also work for immutable objects and so are considered methods of type Object.

3.5.1 Copying

Copying an instance of a mutable type is useful as a way to preserve an object's state from side-effects and to avoid aliasing. The usual way to copy an object is to use the copy operation, which is defined for many types. However, its specification cannot be given here, as it varies from type to type.

shallowCopy *self: MutableObject* $\rightarrow$ *o: MutableObject*

Ensures: o is a new object that has the same value as *self*, o is not the same object as *self* (i.e., **obj**(o) $\neq$ **obj**(*self*)), but o shares all the objects contained in *self* with *self*.

deepCopy *self: MutableObject* $\rightarrow$ *o: MutableObject*

Ensures: o is a new object that has the same value as *self*, is not the same object as *self* (i.e., **obj**(o) $\neq$ **obj**(*self*)), and does not shares any objects contained in *self*. That is, none of the objects contained in o is equal (==) to any of the objects contained in *self*.

4 Smalltalk

The type Smalltalk handles terminal input and output, as well as saving the state of the interpreter. The class Smalltalk has as its only instance the value of the pseudo-variable smalltalk.

4.1 Saving System State

See also the methods in Section 2.5.

saveImage *self: Smalltalk*
Effect: prompt the user for a file name (e.g., "systemImage", typed without the quotes) and save the current system state in that file.

saveImage: *self: Smalltalk, s: String*
Effect: save the current system state in a file named *s*.

4.2 Terminal I/O

getPrompt: *self: Smalltalk, p: String → s: String*
Modifies at most: standard input and output.

Ensures: *s* is a string consisting of a line read from the standard input, after *p* is printed as a prompt on the standard output with no carriage return. The line read from standard input is terminated by a carriage return, which is not included in *s*.

inquire: *self: Smalltalk, p: String → b: Boolean*
Modifies at most: standard input and output.

Ensures: *b* is *true* if the first character on a line read from standard input is either "y" or "Y"; *b* is *false* otherwise. Before reading the line, the string *p* is printed as a prompt (i.e., with no carriage return) on the standard output.

error: *self: Smalltalk, s: String*
Modifies at most: standard error output.

Effect: print *s* as an error message on the standard error output, followed by a trace of the run-time stack, and terminate current execution (so this operation never returns).

5 Primitive Types

The type UndefinedObject has as its only instance the object that is the value of the pseudo-variable nil. It is sometimes useful as a placeholder or a default value.

5.1 Boolean

The type Boolean provides conditionals and logical connectives. It is implemented by three classes named Boolean, True, and False. There are two boolean objects, *true* and *false*, which are bound to the pseudo globals true and false respectively. The class of *true* is True, and that of *false* is False. Both of these classes are subclasses of the class Boolean. Having two classes provides an interesting demonstration of the technique of using the class of an object to select the algorithm used in the implementation of a message.

5.1.1 Conditionals

The messages ifTrue:ifFalse: and ifFalse:ifTrue: can be used both as expressions and statements.

ifTrue:ifFalse: *self: Boolean, c: Block, a: Block → o: Object*
Requires: *c* and *a* take no arguments.

Modifies at most: the objects modified by the execution of *c* or *a*.

Ensures: *o* is the result of evaluating *c* if *self* is *true*; otherwise *o* is that of evaluating *a*.

ifFalse:ifTrue: *self: Boolean, a: Block, c: Block → o: Object*

Requires: *a* and *c* take no arguments.

Modifies at most: the objects modified by the execution of *c* or *a*.

Ensures: *o* is the result of evaluating *a* if *self* is *false*; otherwise *o* is that of evaluating *c*.

ifTrue: *self: Boolean, c: Block*

Requires: *c* takes no arguments

Modifies at most: the objects modified by the execution of *c*.

Effect: Evaluates *c* if *self* is *true*; otherwise does nothing.

ifFalse: *self: Boolean, a: Block*

Requires: *a* takes no arguments

Modifies at most: the objects modified by the execution of *a*.

Effect: Evaluates *a* if *self* is *false*; otherwise does nothing.

5.1.2 Logical connectives

To achieve short-circuit evaluation, the connectives or: and and: take as their second argument a block that should return a boolean.

and: *self: Boolean, aBlock: Block → b: Boolean*

Requires: *aBlock* takes no arguments and returns a Boolean.

Modifies at most: the objects modified by the execution of *aBlock*.

Ensures: *b* is *false* if *self* is *false*; otherwise *b* is the result of evaluating *aBlock*.

or: *self: Boolean, aBlock: Block → b: Boolean*

Requires: *aBlock* takes no arguments and returns a Boolean.

Modifies at most: the objects modified by the execution of *aBlock*.

Ensures: *b* is *true* if *self* is *true*; otherwise *b* is the result of evaluating *aBlock*.

not *self: Boolean → b: Boolean*

Ensures: *b* is *true* if *self* is *false*; otherwise *b* is *false*.

xor: *self: Boolean, b2: Boolean → b: Boolean*

Ensures: *b* is *true* if *self* is not equal to *b2*; otherwise *b* is *false*.

5.2 Symbol

Symbols, such as `#foo:bar:`, are used for message selectors and in dictionaries. They can be created by the compiler from literals (`#aSymbolLiteral`), or by sending the message asSymbol to a String object. Symbols cannot be created using new. Abstractly a symbol is a string of characters. Hence, its identity is determined by its string of characters. As a result, there is only one symbol with a given string of characters. Symbols are immutable.

In addition to the methods described below, the methods apply: and apply:ifError: can be used to send a message when the name of the message is not known until run-time.

respondsTo *self: Symbol → s: Set*

Ensures: *s* is a new set that contains all classes whose instances respond to the message named *self*.

asString *self: Symbol → s: String*

Ensures: *s* is the string of characters that form the abstract value of *self*.

5.3 Magnitude

The type Magnitude defines a general comparison protocol followed by its subtypes: Char, Number, and some subtypes of Collection. The messages <=, <, >=, >, ~= (not equal), =, min:, and max: can be sent to objects of type Magnitude (or a subtype) and have the usual meanings. They should only be considered defined when the second argument has the same subtype as the receiver (e.g., both numbers or both characters). The only unusual message is the following.

between:and: *self: Magnitude, low: Magnitude, high: Magnitude → b: Boolean*

Requires: *self*, *low*, and *high* are instances of classes that have a common ancestor which is a proper subclass of Magnitude.

Ensures: *b* is *true* if *self* is greater than or equal to *low* and *self* is less than or equal to *high*; otherwise *b* is *false*.

5.4 Char

Characters can be created using literals (such as `$c`) by sending the asCharacter message to a (small) Integer. The type Char is a subtype of `Magnitude`; thus characters can be compared with the usual comparison operations (see Section 5.3).

asInteger *self: Char → i: Integer*

Ensures: *i* is the ASCII code for *self*.

isAlphabetic *self: Char → b: Boolean*

Ensures: *b* is *true* just when *self* is an alphabetic character (a-z or A-Z).

isDigit *self: Char → b: Boolean*

Ensures: *b* is *true* just when *self* is a digit (0-9).

isAlphaNumeric *self: Char → b: Boolean*

Ensures: *b* is *true* just when *self* is an alphabetic character (a-z or A-Z) or a digit (0-9).

isBlank *self: Char → b: Boolean*

Ensures: *b* is *true* just when *self* is the space character (`$␣`).

isUppercase *self: Char → b: Boolean*

Ensures: *b* is *true* just when *self* is an upper case letter (A-Z).

isLowercase *self: Char → b: Boolean*

Ensures: *b* is *true* just when *self* is a lower case letter (a-z).

5.5 Numbers

The type Number has three subtypes: Integer, Fraction, and Float. Integers and floating point numbers can be created using literals. Integers can also be created by sending the message asInteger to a String object. Fractions are created by division of integers using the message /. The numerator and denominator of a fraction can be obtained by using the messages top and bottom.

Definite iteration over numbers is possible by creating intervals using the methods to: or to:by: sent to a number (see Section 7.2). For example the following expression prints the odd numbers from 1 to 10.

```
(1 to: 10 by: 2) do: [:i | i print]
```

The timesRepeat: message (see Section 5.5.2) can also express definite iteration. Protocol for comparing numbers is inherited from the type Magnitude (see Section 5.3).

5.5.1 Arithmetic Operations

The protocol for arithmetic, given in Smalltalk's syntax, is fairly standard. Most of it is implemented in the class Number. See Figure 3.

5.5.2 Integer Specific operations

In addition to the following operations, integers also have several operations that can be used to manipulate bit-fields: bitShift:, bitAnd:, bitOr:, bitXor:, bitAt:, bitInvert, anyMask, and allMask.

even *self: Integer → b: Boolean*

Ensures: *b* is *true* if *self* is divisible by 2; otherwise *b* is *false*.

odd *self: Integer → b: Boolean*

Figure 3: Arithmetic operations

message name	meaning
abs	absolute value
negated	unary negation
positive	test if receiver is greater than or equal to 0
strictlyPositive	test if receiver is greater than 0
negative	test if receiver is less than 0
sign	return -1, 0, or 1 depending on sign of receiver
rounded	integer nearest the receiver
truncated	integer part of receiver
fractionalPart	fractional part of receiver
floor	largest integer not greater than receiver
ceiling	smallest integer not less than receiver
exp	e to power of receiver
ln	logarithm base e
log:	logarithm at base of argument
sqrt	square root of receiver
squared	receiver multiplied by itself
raisedTo:	multiply receiver by itself argument times
reciprocal	1 divided by receiver
+	addition
-	subtraction
*	multiplication
/	division (preserves accuracy)
//	division truncated towards $-\infty$
\\\\	modulo (remainder truncated towards $-\infty$)
rem:	remainder truncated towards 0
quo:	quotient truncated towards 0

Ensures: b is *true* if *self* is not divisible by 2; otherwise b is *false*.

factorial *self: Integer* $\rightarrow$ *i: Integer*

Requires: *self* is not negative.

Ensures: i is the factorial of *self*; that is, $i = self \times (self - 1) \times \cdots \times 2 \times 1$.

timesRepeat: *self: Integer*, *aBlock: Block*

Requires: *self* is not negative and *aBlock* has no arguments.

Modifies at most: the objects modified by the execution of *aBlock*.

Effect: evaluate *aBlock self* times.

gcd: *self: Integer*, *j: Integer* $\rightarrow$ *i: Integer*

Ensures: i is the largest integer that divides both *self* and j.

radix: *self: Integer*, *base: Integer* $\rightarrow$ *s: String*

Requires: *base* is between 2 and 36 (inclusive).

Ensures: s represents the value of *self* as a numeral in base *base* digits.

6 Collections

The type Collection is the supertype of types, like List and Array, whose instances contain a "bunch" of other objects. The major subtypes of Collection are List and Array. Arrays are much like those in Smalltalk-80, but lists are quite different. Other subtypes of Collection are Set, Dictionary, String, and Interval. Common protocol for Dictionary, Array and String is implemented by the type IndexedCollection (See Figure 1 on page 23 for more detail). Protocol for all collections (i.e., those methods defined in the type Collection) is described in the next sub-section. The following sub-sections describe the subtypes.

6.1 Protocol for all Collections

Objects of a subtype of Collection are mutable, and thus Collection is a subtype of MutableObject (see Section 3.5). As such collections can be copied. A deep copy of a collection generally copies each element, while a shallow copy shares elements with the original. By default copy gives a shallow copy

It is best to think of a collection as homogeneous, that is, a collection should contain elements that have a common supertype (e.g., Object) if nothing else. Following this view, one can think of Collection as a type generator, which can be instantiated for various element types; for example, sets of integers or arrays of objects.

The type of a collection of objects that are magnitudes (e.g., lists of integers) is a subtype of Magnitude and hence supports the comparison protocol of the type Magnitude (see Section 5.3). In implementation terms, the comparison protocol that the class Collection inherits from the class Magnitude only works if the elements also support the protocol. The basic "magnitude" of a collection in this view is how many times an element occurs in the collection. For example, if `c1` and `c2` are lists, then `c1 <= c2` if every element of `c1` occurs at least as many times in `c2` as it occurs in `c1`. (See the message occurrencesOf: below.)

6.1.1 Testing

isEmpty *self: Collection* $\rightarrow$ *b: Boolean*

Ensures: b is *true* if *self* contains no elements, and *false* otherwise.

size *self: Collection* $\rightarrow$ *i: Integer*

Ensures: i is the number of elements in *self*.

occurrencesOf: *self: Collection*, *o: Object* $\rightarrow$ *i: Integer*

Ensures: i is the number of elements in *self* that are equal (=) to o.

includes: *self: Collection*, *o: Object* $\rightarrow$ *b: Boolean*

Ensures: b is *true* if *self* contains an element that is equal (=) to o, and *false* otherwise.

6.1.2 Iteration

See also the methods for the type Interval in Section 7.2.

do: *self: Collection, aBlock: Block*

Requires: *aBlock* takes one argument.

Modifies at most: the objects modified by the execution of *aBlock*.

Effect: evaluate *aBlock* for each element of *self*.

select: *self: Collection, aBlock: Block* → *c: Collection*

Requires: *aBlock* takes one argument, returns a Boolean, and has no side-effects.

Ensures: *c* is a new collection with the same class as *self*, that contains just those elements *e* of *self* for which the expression *aBlock* value: *e* is *true* (See Section 7.1).

inject:into: *self: Collection, v: Object, aBlock: Block* → *o: Object*

Requires: *aBlock* takes two arguments and has no side-effects.

Ensures: *o* is the reduction of *self* by *aBlock* with initial value *v*. That is, if *self* has no elements, then *o* is *v*. If *self* has an element *e*, then *o* is *aBlock* value: *x* value: *e*, where *x* is the reduction of the remaining elements of *self*. For example, one can sum the elements of a collection c by the following expression.

```
c inject: 0 into: [ :x :y | x + y ]
```

6.1.3 Conversions among Collection Types

asArray *self: Collection* → *a: Array*

Ensures: *a* is a new Array that contains just the elements of *self*.

asSet *self: Collection* → *s: Set*

Ensures: *s* is a new Set that contains just the elements of *self*, such that no two elements of *s* are equal (=).

sort: *self: Collection, aBlock: Block* → *lst: List*

Requires: *aBlock* takes two arguments, returns a Boolean, and has no side-effects; furthermore, the elements of *self* are totally ordered by *aBlock*.

Ensures: *lst* contains just the elements of *self*, sorted so that for all $i < j$ that are indexes of *lst*, *aBlock* value: (*lst* at: *i*) value: (*lst* at: *j*) returns *true*.

sort *self: Collection* → *lst: List*

Requires: the elements of *self* are totally ordered by <=.

Ensures: *lst* contains just the elements of *self* in increasing order.

6.2 List

A List object is, abstractly, a mutable, ordered sequence of objects. Elements can be added and taken off the list at both ends; there is a first element and a last element in every non-empty list. The objects in the sequence are the list's elements. Empty instances of List are created by sending the new message to the class object List.

6.2.1 Adding and Removing Elements

The various ways that elements can be added to a list can be exploited to implement stacks, queues, priority queues, and other data structures. Note, however, that there is no removeLast operation that corresponds to the removeFirst operation.

add: *self: List, o: Object*

Modifies at most: *self*.

Effect: makes **post**(*self*) contain *o* as its last element; the other elements of **post**(*self*) are those of **pre**(*self*), in their original order.

addLast: *self: List, o: Object*

Modifies at most: *self.*

Effect: makes **post**(*self*) contain *o* as its last element; the other elements of **post**(*self*) are those of **pre**(*self*), in their original order.

addFirst: *self: List, o: Object*

Modifies at most: *self.*

Effect: makes **post**(*self*) contain *o* as its first element; the remaining elements of **post**(*self*) are those of **pre**(*self*), in their original order.

addAll: *self: List, c: Collection*

Modifies at most: *self.*

Effect: makes **post**(*self*) contain all the elements of *c* as its last elements (in the ordering of *c*, if *c* has an order), and the elements of **pre**(*self*) as its first elements, in their original order.

add:ordered: *self: List, o: Object, b: Block*

Requires: *b* takes two arguments, returns a Boolean, and has no side-effects.

Modifies at most: *self.*

Effect: makes **post**(*self*) contain all the elements of **pre**(*self*) in their original order, except that *o* appears in **post**(*self*) just before the first element of **pre**(*self*) *v* such that the expression *b* value: *v* value: o is *false*. For example, if `lst` is a list with elements 1, 1, and 3 (in that order), the result of

```
lst add: 2 ordered: [ :x :y | x < y ]
```

is to insert 2 just before the 3 in `lst`, so that the elements of `lst` are (in order): 1, 1, 2, and 3.

remove: *self: List, o: Object*

Modifies at most: *self.*

Effect: makes **post**(*self*) contain all the elements of **pre**(*self*) in their original order, except for the first occurrence of an element that is equal (=) to *o*.

removeFirst *self: List*

Modifies at most: *self.*

Effect: makes **post**(*self*) contain all the elements of **pre**(*self*) in their original order, except for the first element of **pre**(*self*).

6.2.2 Accessing Elements

Surprisingly, there is only one operation available to access elements directly. The other way to access the elements of a list is by using the iteration operations described below and in Section 6.1.2.

first *self: List* $\rightarrow$ *o: Object*

Ensures: *o* is the first element of *self.*

6.2.3 Mapping and Filtering

In addition to those operations described in Section 6.1.2, List also defines following iteration operations.

collect: *self: List, aBlock: Block* $\rightarrow$ *lst: List*

Requires: *aBlock* takes one argument.

Modifies at most: the objects modified by the execution of *aBlock*.

Ensures: *lst* is a new list that contains for each element *e* of *self* the result of the expression *aBlock* value: *e*, where the results appear in the same order as the elements of **pre**(*self*). (The argument *aBlock* is invoked on the elements in the order that they appear in **pre**(*self*).)

reject: *self: List, aBlock: Block* $\rightarrow$ *lst: List*

Requires: *aBlock* takes one argument, returns a Boolean, and has no side-effects.

Ensures: *lst* is a new list that contains the elements *e* of *self* such that the result of the expression *aBlock* value: *e* is *false*, The elements appear in their original order.

reverseDo: *self: List, aBlock: Block*

Requires: *aBlock* takes one argument.

Modifies at most: the objects modified by the execution of *aBlock*.

Effect: evaluate *aBlock* once for each element of *self*, starting with the last element and working back to the first.

6.3 Set

The abstract value of a `Set` instance is a set of objects with distinct abstract values. The elements of a `Set` are compared using the equality (=) message. That is, two objects x and y have the same abstract value if and only if the result of the equality (=) message is *true*. It is assumed that equality (=) can be applied to any two objects and that it is an equivalence relation on all objects.

Instances of class Set are partially ordered by set inclusion; that is, s_1 is a subset of s_2 if for each element e_1 of s_1, there is some element in s_2 with the same abstract value (i.e., that is equal (=) to e_1). Two instances of Set have the same abstract value if each is a subset of the other. Furthermore s_1 is a proper subset of s_2 if s_1 is a subset but they do not have the same abstract value. The = message tests sets for the same abstract value, while the `<=` message tests the subset relation.

New instances of `Set` are created with by sending the `new` message to the class object `Set`, which returns an empty set.

6.3.1 Adding and Removing Elements

add: *self: Set, o: Object*

Modifies at most: *self.*

Effect: makes **post**(*self*) contain just the elements of **pre**(*self*) and *o*.

addAll: *self: Set, c: Collection*

Modifies at most: *self.*

Effect: makes **post**(*self*) contain just the elements of **pre**(*self*) plus all the elements of *c*.

remove: *self: Set, o: Object*

Modifies at most: *self.*

Effect: makes **post**(*self*) contain the elements of **pre**(*self*) except *o*.

6.3.2 Iteration

In addition to those messages described in Section 6.1.2, Set also defines following iteration protocol.

collect: *self: Set, b: Block* $\rightarrow$ *s: Set*

Requires: *b* takes one argument.

Modifies at most: the objects modified by the execution of *b*.

Ensures: *s* is a new set that contains for each element *e* of *self* the value of the expression *aBlock* value: *e*. (The set *s* is constructed by invoking *b* once for each element of *self* in arbitrary order.)

reject: *self: Set, b: Block* $\rightarrow$ *s: Set*

Requires: *b* is a one-argument block that returns a Boolean and has no side-effects.

Ensures: *s* is a new set that contains just those elements, *e*, of *self* for which *b* value: *e* is *false*. Note that this is the opposite of select:.

6.4 Array

The abstract value of an Array is an ordered sequences of elements. This ordering is used to index the elements. The first element has index 1, the second 2, and so on. A legal index is thus the one between 1 and the size of the collection (inclusive).

An array is created using a literal of the form `#(1 aSymbol 3.3)` or by the new: message sent to the class object Array where the argument is the desired size.

6.4.1 Fetching and Storing

Some messages not described below give the programmer more control over error conditions through the use of zero-argument exception blocks. These include at:ifAbsent: and indexOf:ifAbsent:.

at: *self: Array, i: Integer → o: Object*

Requires: *i* is greater than 0, and less than or equal to the size of *self.*

Ensures: *o* is the i^{th} element of *self.*

indexOf: *self: Array, b: Block → i: Integer*

Requires: *b* takes one argument, returns a Boolean, and has no side-effects; furthermore *self* contains some element *x* such that *b* value: *x* is true.

Ensures: *i* is the smallest index of *self* such that the expression *b* value: (*self* at: *i*) returns *true*.

at:put: *self: Array, i: Integer, o: Object*

Requires: *i* is greater than 0, and less than or equal to the size of *self.*

Modifies at most: *self.*

Effect: makes *o* the element of **post**(*self*) at index *i*; the other elements of **pre**(*self*) are unchanged.

with: *self: Array, o: Object → a: Array*

Ensures: *a* is a new array that has as its elements the elements of *self* in order, followed by *o*.

addAll: *self: Array, a2: Array*

Requires: *a2* has no more elements than *self*.

Modifies at most: *self.*

Effect: makes the first s_2 elements of **post**(*self*) the elements of *a2* in the order of *a2*, where s_2 is the size of *a2*; the elements of **post**(*self*) whose indexes are greater than s_2, if any, are unchanged from **pre**(*self*).

exchange:and: *self: Array, i: Integer, j: Integer*

Requires: *i* and *j* are greater than 0, and less than or equal to the size of *self.*

Modifies at most: *self.*

Effect: the element of **post**(*self*) at index *i* is that of **pre**(*self*) at index *j* and vice versa; the other elements of **pre**(*self*) are unchanged.

6.4.2 Iteration

As a subtype of Collection, Array also supports all the iteration operations described in Section 6.1.2. In addition to the above, there are also several other interesting operations that have been omitted for lack of space. The messages binaryDo: and binaryInject:into: enable one to operate on both elements and indexes. The messages with:do: and with:ifAbsent:do: allow one to operate on the elements of two arrays at once.

collect: *self: Array, aBlock: Block → a: Array*

Requires: *aBlock* takes one argument.

Modifies at most: the objects modified by the execution of *aBlock*.

Ensures: *a* is a new array that contains for each element *e* of *self* the result of the expression *aBlock* value: *e*, where the results appear in the same order as the elements of **pre**(*self*). (The argument *aBlock* is invoked on the elements in the order they appear in **pre**(*self*).)

reverseDo: *self: Array, aBlock: Block*

Requires: *aBlock* takes one argument.

Modifies at most: the objects modified by the execution of *aBlock*.

Effect: evaluates *aBlock* once for each element of *self*, starting with element at the greatest index, and working back to the element with index 1.

6.5 Dictionary

Abstractly, a Dictionary object is a mutable mapping from keys to values. The mapping is generally partial. The elements of a dictionary are the values, i.e., the range of the map. As a subtype of Collection, Dictionary also supports the iteration operations described in Section 6.1.2. Some messages not described below give the programmer more control over error conditions through the use of zero-argument exception blocks, for example, at:ifAbsent: and indexOf:ifAbsent:.

at: *self: Dictionary, key: Object* $\rightarrow$ *v: Object*

Requires: *self* is defined at *key*.

Ensures: *v* is the value associated with *key* in *self*.

indexOf: *self: Dictionary, b: Block* $\rightarrow$ *key: Object*

Requires: *b* takes one argument, returns a Boolean, and has no side-effects; furthermore, *self* contains some element x such that *b* value: x is true.

Ensures: *key* is one of the keys of *self* such that *b* value: (*self* at: *key*) returns *true*.

includesKey: *self: Dictionary, key: Object* $\rightarrow$ *b: Boolean*

Ensures: *b* is *true* if *self* is defined at *key*, otherwise *b* is *false*.

at:put: *self: Dictionary, key: Object, value: Object*

Modifies at most: *self*.

Effect: makes *value* the element of **post**(*self*) at key *key*; the rest of the mapping is unchanged.

addAll: *self: Dictionary, d2: Dictionary*

Modifies at most: *self*.

Effect: makes **post**(*self*) map the keys of *d2* to the values associated by *d2*; other keys defined in **pre**(*self*) are mapped to their values in **pre**(*self*).

removeKey: *self: Dictionary, key: Object*

Requires: *self* is defined at the key *key*.

Modifies at most: *self*.

Effect: makes **post**(*self*) not defined at *key*; the mapping is unchanged for all other keys.

6.6 String

A String object is an ordered collection of characters. Strings are mutable, and are a subtype of the type array of characters. As such they inherit copying protocol (see Section 3.5) and the fetching and storing protocol of Array (see Section 6.4.1), but with the restriction that only Char objects can be fetched and stored.

As a subtype of Collection whose elements are magnitudes, String supports a lexicographic ordering messages of type Magnitude (see Section 5.3), and the iteration messages described in Section 6.1.2. The iteration protocol of Arrays also applies (see Section 6.4.2).

Strings cannot be created by sending the new message to the class object String. Instead they are created by literals (`'a String, don''t laugh!'`).

, *self: String, s2: String* $\rightarrow$ *s: String*

Ensures: *s* is a new String that contains the elements of *self* (in their original order), followed by the elements of *s2*. This is a string concatenation. For example,

```
'high' , 'brow'
```

returns a string containing the characters “highbrow” in that order.

copyFrom:to: *self: String, f: Integer, t: Integer* $\rightarrow$ *s: String*

Requires: f is no less than 0 and no greater than the size of *self*, and t is no less than $f - 1$ and no greater than the size of *self*.

Ensures: *s* is a new String containing, in order, the characters of *self* starting at index f to index t (inclusive). This is a substring operation.

words: *self: String, aBlock: Block* $\rightarrow$ *a: Array*

Requires: *aBlock* takes one character argument and returns a Boolean.

Ensures: *a* is a new Array that contains as elements substrings of *self* that are consecutive runs of characters of *self* such that *aBlock* value: *c* returns *true*, in the order in which these runs appear in *self*. For example,

```
'hi###there' words: [:c | c ~= $#]
```

returns an array with two elements the strings `'hi'` and `'there'`.

7 Control Structures

Conditionals, which are messages sent to Booleans, are described in Section 5.1.1.

7.1 Block

The class Block provides the essential control structures of Smalltalk. Technically a block is a closure; that is, a block contains some parameterized code and an environment used to look up variables that occur in the block but which are not parameters.

Blocks may have up to three arguments. Zero argument blocks are invoked with the value message. The main use of a zero-argument block is to delay evaluation of code. One argument blocks are invoked with the value: message while two argument blocks are with the value:value: message. Finally, the value:value:value: message is used to invoke a three-argument blocks are invoked with the value:value:value: message. For example, the following expressions all have a value of 7.

```
[ 5 + 2 ] value
[ :x | x + 2 ] value: 5
[ :x :y | x + y ] value: 5 value: 2
[ :x :y :z | x + y + (z - z)] value: 5 value: 2 value: 8
```

In addition to the following, there are also messages (fork, forkWith:, and newProcess) that can be used to create multiple threads of control (parallelism).

whileTrue: *self: Block, body: Block*

Requires: *self* is a zero-argument block that returns a Boolean, and *body* is a zero-argument block.

Modifies at most: the objects modified by the execution of *self* and *body*.

Effect: evaluate *self*, if the result is *false*, return; otherwise evaluate *body* and repeat this process. For example,

```
i <- 1.3
[i <= 4] whileTrue: [ i print.  i <- i+1]
```

prints the numbers 1.3, 2.3, and 3.3 in that order.

whileFalse: *self: Block, body: Block*

Requires: *self* is a zero-argument block that returns a Boolean, and *body* is a zero-argument block.

Modifies at most: the objects modified by the execution of *self* and *body*.

Effect: evaluate *self*, if the result is *true*, return; otherwise evaluate *body* and repeat this process.

whileTrue *self: Block*

Requires: *self* is a zero-argument block.

Modifies at most: the objects modified by the execution of *self*.

Effect: evaluate *self* repeatedly, so that this operation never returns.

7.2 Interval

The type Interval is a subtype of Collection. An Interval object is, abstractly, an arithmetic sequence of numbers, either ascending or descending. Each Interval has a lower bound, an upper bound, and a step size. Intervals with step size 1 are created by sending the message to: to a number with an number argument;

for example, the expression `1.3 to: 4` creates an interval with lower bound 1.3, upper bound 4, and step size 1. The "elements" of this interval are 1.3, 2.3, and 3.3. In general, the elements are numbers of the form $l + (n \times s)$ where l is the lower bound, s is the step size, and n is an integer that ranges from 0 up to the largest integer N such that $l + ((N+1) \times s)$ is strictly greater than the upper bound if the step size is positive or strictly less than the upper bound if the step size is negative.

The "for loop" of other languages is modeled by the do: message sent to an Interval object For example,

```
(1.3 to: 4) do: [ :i | i print ]
```

prints the numbers 1.3, 2.3, and 3.3 in that order. Less involved "for loops" can also be constructed by sending an integer the timesRepeat: message with a zero-argument block as an argument (see Section 5.5.2). Collections also provide direct ways to iterate over their elements, using the message do: and others described in Section 6.1.2.

The lower bound, the upper bound, and the step size of an Interval object can be modified by the following methods.

lower: *self: Interval, n: Number*

Modifies at most: *self*

Ensures: n is the lower bound of **post**(*self*).

upper: *self: Interval, n: Number*

Modifies at most: *self*

Ensures: n is the upper bound of **post**(*self*).

step: *self: Interval, n: Number*

Modifies at most: *self*

Ensures: n is the step size of **post**(*self*).

8 Files

A File object is created by new message. The name of the file associated with a File object is set by name: (where the argument must be a string), and accessed by name. A file can be opened in three different modes: read ('r'), write ('w'), and read-and-write ('r+w'). The message open: (with one of the mode strings as its argument) is sent to a file object to open it; if mode is 'w', a new file named that of the receiver is created in the current directory, otherwise, the existing file named that of the receiver is opened for reading in the current directory. If no such file exists when the open-mode is 'r', an error occurs. A File object is closed by close message. The method readUntil:doing: can be used for complex reading operations (such as parsing).

The main methods for reading and writing to files are described below. See also Section 2.5 for messages that save and read classes and method definitions into and out of files.

getString *self: File → s: Object*

Requires: *self* was opened with read mode ('r' or 'r+w').

Modifies at most: *self*

Ensures: s is the next line of *self* (terminated by a carriage return) if such is present before the end-of-file; otherwise s is nil.

print: *self: File, s: String*

Requires: *self* was opened with write mode ('w' or 'r+w').

Modifies at most: *self*

Effect: appends s at the end of *self.*

9 Guide to the Implementation

Table 1 lists types specified in this document; it shows which classes implement which operations by methods. The left-hand column gives names of types specified above. The next column, gives the names of the class or classes that implements each type. For example, the type Integer is implemented by the classes Integer and LongInteger. Operations are mentioned in the column *spec & impl* next to the classes that implement them, provided the operations are also specified above. For example, the operation radix: is implemented by the class Integer, and we also give a specification for it above under the type Integer. The column *re-implemented* gives those operations that are specified for a supertype, implemented by the corresponding superclass, and that are also re-implemented (over-ridden) by the subclass. For example, the abs operation is specified in Integer's supertype Number, implemented by a method in LongInteger's superclass Number, and that method is re-implemented in the class LongInteger.

Notice that we omit from a type's specification operations specified by its supertypes. Therefore, to get all the operations defined for a type, one may need to traverse its supertype chain given in Figure 1 on page 5. For example, the operation abs is defined for Integer objects, because it is specified in Integer's supertype Number. In the implementation, the class Integer inherits the method abs from its superclass Number (see Figure 2 on page 5), because abs is not mentioned in the column labeled *re-implemented*.

Operations listed in the column *must be impl* are specified by a type but not implemented by the corresponding class; these operations must be implemented in subclasses of the given class. For example, we specified do: as an operation defined on objects of type Collection, but the class Collection does not implement do:. A slightly different case is the operation =, which is implemented in the class Object, even though its implementation should usually be over-ridden.

Several operations not specified in this document but implemented in Little Smalltalk are listed below the column with header *not specified*.

Acknowledgements

Thanks to Krishna Kishore Dhara and the students in "Principles of Programming Languages" and "Programming Languages I" at Iowa State for helping debug these specifications and for suggestions on improving their presentation.

References

[Bud87] Timothy Budd. *A Little Smalltalk*. Addison-Wesley, Reading, Mass., 1987.

[Bud91] Timothy Budd. *Object-Oriented Programming*. Addison-Wesley, New York, N.Y., 1991.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80, The Language and its Implementation*. Addison-Wesley Publishing Co., Reading, Mass., 1983.

[Lea91] Gary T. Leavens. Modular Specification and Verification of Object-Oriented Programs. *IEEE Software*, 8(4):72–80, July 1991.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. The MIT Press, Cambridge, Mass., 1986.

[LW90] Gary T. Leavens and William E. Weihl. Reasoning about Object-oriented Programs that use Subtypes (extended abstract). *ACM SIGPLAN Notices*, 25(10):212–223, October 1990. *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).

Table 1: Specified types and their implementations

type	impl classes	operations			
		spec & impl	re-implemented	must be impl	not specified
Array	IndexedCollection	at: indexOf: addAll: at:ifAbsent: indexOf:ifAbsent:	asArray		
	Array	at:put: with: collect: select: reverseDo: exchange:and:	size deepCopy shallowCopy do:		=< with:do: with:ifAbsent:do:
Block	Block	value: value:value: value:value:value: whileTrue whileTrue: whileFalse:			
Boolean	Boolean	ifTrue: ifFalse: ifFalse:ifTrue: and: or:		ifTrue:ifFalse	
	True	ifTrue:ifFalse: not xor	printString		
	False	ifTrue:ifFalse: not xor	printString		
Char	Char	asInteger isAlphabetic isDigit isBlank isUppercase isLowercase isAlphaNumeric	printString < ==		digitValue asString
Class	Class	addSubClass:instnaceVariable-Names: superClass subClasses methods upSuperclassChain: respondsTo viewMethod: methodNamed: addMethod readMethods editMethod: fileOut fileOutOn: fileOutMethodsOn: new new:	printString display		name name: instanceSize variables
Collection	Collection	isEmpty size includes: occurrencesOf: inject:into: asArray asSet sort sort:	printString display	do:	= <
Dictionary	IndexedCollection	at: indexOf: addAll: collect: select: do:	asArray		indexOf:ifAbsent:
	Dictionary	includesKey: at:put: removeKey:	display		at:ifAbsent: hash removeKey:ifAbsent:
File	File	getString print: open: close readUntil:doing: fileIn fileIn:			open mode mode: delete name name:
Float	Float	integerPart rounded	printString exp truncated + − ∗ / = < quo ln		
Fraction	Fraction	with:over	abs reciprocal printString truncated ln + − ∗ / < =		top bottom
Integer	Integer	radix: timesRepeat: even odd lcm: gcd: factorial + − ∗ / = < >	printString quo truncated		
	LongInteger		abs negated quo printString negative + − ∗ = <		
Interval	Interval	lower: upper: do: step:			
List	List	add: addLast: addAll: addFirst: add:ordered: remove: removeFirst first collect: do: reverseDo: reject: select:	size		links
Magnitude	Magnitude	<= >= > < = ˜= min max: between:and:			
MutableObject	Object	shallowCopy deepCopy			
Number	Number	abs negated positive strictlyPositive negative sign rounded tuncated fractionalPart floor ceiling exp ln log: sqrt raisedTo: reciprocal + − ∗ / // \\ rem: quo:	= <	generality coerce	
Object	Object	class, isMemberOf: isKindOf: respondsTo: print printString display == ˜˜ isNil notNil		=	hash, basicAt: basicAt:put: basicSize
Set	List	addAll: remove: do: collect: select: reject:	size		links
	Set	add:			
Smalltalk	Smalltalk	getPrompt: inquire: error: saveImage saveImage:			
String	Magnitude	<= >= ˜= min: max: between:and:			
	String	, = < word: copyFrom:to:	size print printString		hash
Symbol	Symbol	respondsTo asString apply: apply:ifError:	printString		
UndefinedObject	UndefinedObject		printString isNil notNil		