A Little *more* Smalltalk

Timothy A. Budd
Oregon State University
Corvallis, Oregon
USA

December 16, 1994

Contents

Part I

Getting Started

Chapter 1

The Object-Oriented Mindset

Describe the basic object-oriented mind set. Similar to the platypus book.

Chapter 2

The Little Smalltalk System

Describe the basic features of the Smalltalk Language, including differences from Smalltalk-80.

2.1 The Little Smalltalk Language

2.1.1 Constants

2.1.2 Variables

2.1.3 Messages

2.1.4 Classes

2.1.5 Methods

2.2 The Real-Eval-Print loop Interface

The read/eval/print loop interface is the simplest of the user interfaces supported by the various versions of Little Smalltalk. In the read/eval/print loop interface expressions are input by the user in response to a prompt. Each expression is evaluated, and the results of executing the expression are printed. An example session might be as follows:

```
-> 3 + 4
7
-> 3 class
SmallInt
->
```

More here....

2.2.1 Creating New Classes and Methods

Classes in Smalltalk are simply global variables with values which are instances of class Class. New classes and methods are formed by sending the messages to the appropriate class object. Figure 2.1 describes the protocol provided by class Class for this purpose.

For example, to create a new class, an existing class object is selected to serve as the parent class. Often this is just the class Object, which is a parent class to all objects in the Little Smalltalk system. The method subclass: (or one of its variations) is then passed to this object. The argument must be a symbol, representing the name of the new class.

```
-> Object subclass: #Foo
subclass created: Foo
->
```

The alternative forms of the subclass: method are used when the class definition requires the use of instance variables and/or class variables.

Messages to class objects are also used to add new methods to a class or to edit existing method definitions. In response to these methods an editing window is created. Users make modifications to the text in the editing window. (The exact specifics of the editing operation are platform specific, and will not be discussed here). When the window is closed the user is prompted whether the method should be compiled. If the user answers yes, then the bytecode compiler (Chapter 9) is invoked to translate the textual form of the method into the internal bytecode representation. If the compilation process is successful then the new method is added to the class protocol.

```
-> Foo addMethod
... editing session omitted ...
compile method ? y
method inserted: nameOfMethod
->
```

Yet other messages can be used to discover information about specific classes. For example, the message listMethods will generate a list of all the methods implemented directly by the class. The message listAllMethods includes those messages inherited from parent classes.

```
-> Object listMethods
=
==
```

SUBCLASS CREATION PROTOCOL

subclass: #Name
: create a new class by subclassing an existing class

subclass: #Name variables: #(a b c)
: create a new class with given instance variables

subclass: #Name variables: #(a b c) classVariables: #(x y)
: create a new class with given instance variables and class variables

METHOD ADDITION AND MODIFICATION PROTOCOL

addMethod
: spawn an editor for creating a new method

viewMethod: #name
: view text of given method

editMethod: #methodName
: edit text of given method

CLASS INFORMATION PROTOCOL

listMethods
: list methods directly implemented by a class

listAllMethods
: list all methods implemented by a class or by parent classes

methods
: return dictionary of methods held by class

superclass
: list parent class of class

subclasses
: produce an indented description of subclasses of class

instanceVariables
: produce an array containing names of instance variables

provides: #methodName
: query if class or superclass provides implementation for method

Figure 2.1: Class Manipulation Commands using the Read-Eval-Print interface

```
class
error:
isKindOf:
isMemberOf:
isNil
notNil
print
printString
question:
~=
```

The message superclass will return the parent class of an object.

```
-> Dictionary superclass
Collection
-> Object superclass
nil
```

The message `subclasses` produces an indented description of the subclass hierarchy. Passing the message to `Object` will produce a display of the entire Little Smalltalk hierarchy.

```
-> Array subclasses
Array
    ByteArray
        String
->
```

The message methods returns the dictionary which maintains all the methods implemented by a class. Methods implemented by the Dictionary protocol (See Chapter 7) can then be used to access specific information. The following, for example, will print out the text of each method defined by a class.

```
-> Object methods do: [ :meth | meth text print ]
... output omitted ...
```

Class methods

Class methods can be manipulated using the same commands used in the creation and modification of ordinary methods. The user simply places the message `class` between the class name and the method being invoked. For example, the command:

```
-> Foo class addMethod
```

can be used to add a new class method to the class `Foo`.

2.2.2 The `fileIn` mechanism

An alternative to adding classes and methods directly at the keyboard is the use of the `fileIn` mechanism. Textual files of commands, prepared using an ordinary editor prior to beginning a Smalltalk session, can be processed and incorporated into the current image. This task is accomplished by sending the class message `fileIn:` to the object `File`. The argument passed along with the message must be a string representing the file name.

```
-> File fileIn: 'testfile.st'
fileIn successful
->
```

A file to be processed in this fashion must have a special format. Each line of the file is a command. The differing commands are indicated by the first character of the line, which must be one of the following four values:

character	command
`+`	smalltalk command
`!`	method addition
`=`	class method addition
any other char	comment line

Lines beginning with the `+` character contain Smalltalk expressions or statements. The remainder of the line is interpreted as a single smalltalk expression. The expression is immediately executed as the line is read, and the resulting output is printed. Often the commands are used to create new classes, using one of the expressions given in Figure 2.1. Other common uses are to immediately execute a instance of a newly created class, for example to evaluate a set of test cases.

Lines beginning with `!` or `=` begin a multiline command. The only other text on the line should be a class name. Successive lines represent the text of a method to be inserted. The end of this text is indicated by a line containing only the single exlaimation character `!`. Once input, the commands will be inserted into the class indicated by the command line, either as a regular method or as a class method.

Lines beginning with any other character are treated as comments. The remainder of the line is read and ignored. Such lines allows programmers to place informative commentary in files being prepared for file-in processing.

Figure 2.2 shows a file in proper file-in format. The file first adds a new method, named `isPalendrome` to the existing class `String`. Next, it creates a new class `Pal`, which will

```
/ A simple palendrome tester
/ first, add a method to class String
!String
isPalendrome | string |
    string <- self select: [:c | c isAlphabetic ].
    string <- string collect: [:c | c lowerCase ].
    ↑ string = string reverse
!
/ next, add a new class with a few test cases
+Object subclass: #Pal
!Pal
test: aString
    aString print.
    ' : ' print.
    aString isPalendrome print.
    Char newline print.
!
!Pal
runTests
    self test: 'A Man, A Plan, A Canal, Panama'.
    self test: 'Rats live on no Evil star'.
    self test: 'This is not a palendrome'
!
+Pal new runTests
/ end of file
```

Figure 2.2: A file ready to be filed in

be used to test this method. Two new methods are then added to this class. Finally, an expression is used to create an instance of the new class, and to execute the second method.

2.2.3 Saving Images

Once new classes or methods have been created they become part of the current smalltalk *image*. These changes are lost once the smalltalk system is terminated unless the user explicitly saves the current image file. This is most easily done by sending the class message saveImage: to the global variable File. The argument should be a string, containing the name of the file on which the image will be saved.

```
-> File saveImage: 'newImage.st'
image saved
->
```

The mechanism used to start the Little Smalltalk system with an image other than the current image will differ from platform to platform. Under Unix, for example, the image file is simply given as an argument on the command line.

```
% st newImage.st
```

On the macintosh, on the other hand, the user double-clicks on the image file to start the application.

Part II

Exploring the Standard Image

Chapter 3

Basic Objects

Chapter 4

Booleans

The methods defined in the class `Boolean` and its two subclasses `True` and `False` provide an excellent illustration of the power of inheritance and the utility of overridden, or virtual, functions. The class `Boolean` exists to solve two general problems. The first is the evaluation of boolean operators, which permit complex expressions to be rendered in a formal fashion. The second is the implementation of the control flow constructs, used to implement the `ifTrue:` and `ifFalse:` statements.

The abstract class `Boolean` exists only as a parent class for the classes `True` and `False`, which in turn exist only to provide behavior to the global variables `true` and `false`. In fact, the method `new` is overridden in each subclass, so that no new instances of these classes can be created. (That is, passing the message `new` to the class `True` will yield the global variable `true`, the only instance of the class.)

4.1 The Abstract Class `Boolean`

The protocol defined by the class `Boolean` is shown in Figure 4.1. Block expressions are used as a mechanism to delay evaluation by almost all the methods implemented by the class boolean. When used in conjunction with logical operators this is sometimes called "short-circuit evaluation".

Take, for example, the logical conjunction operator, which is represented by the method `and:`. A typical expression using this operator might be the following:

```
i < anArray size and:  [ (anArray at:  i) < 10 ]
```

To execute this, the expression "`i < anArray size`" is evaluated, resulting in either the value `true` or `false`. This result is then used as a receiver for the message `and:`, which passes as argument a block expression. At the time the `and:` message is sent, the expression "`(anArray at:  i) < 10`", contained within the block, has not yet been evaluated. In fact,

PROTOCOL FOR EVALUATION OF LOGICAL OPERATORS

`and: aBlock`
: true if receiver and value of block are both true

`or: aBlock`
: true if either receiver or value of block are true

`not`
: true if receiver is false, false if receiver is true

PROTOCOL FOR CONTROL FLOW STATEMENTS

`ifTrue: trueBlock`
: the value of the block if receiver is true, nil otherwise

`ifFalse: falseBlock`
: the value of the block if receiver is false, nil otherwise

`ifTrue: trueBlock ifFalse: falseBlock`
: the value of the first block if receiver is true, second block otherwise

`ifFalse: falseBlock ifTrue: trueBlock`
: the value of the first block if receiver is false, second block otherwise

Figure 4.1: Methods Implemented by the class `Boolean`

if the result of the relational test is false there is no way the entire expression can ever be true, and thus the block need not be evaluated at all. This is just as well, for the message `at:` might very well generate an out-of-bound error message if the first test is false.

Although the application of short circut semantics to logical operators means that they no longer match their mathematical counterparts (in so much as we lose the property of communititivity), there are two reasons why programming languages typically use short-circuit evaluation. The first is efficiency. If the truth or falsity of an expression can be determined after examining only the first term, then one can save execution time by avoiding the evaluation of the second. However, A second reason is even more important from a practical point of view. It is commonly the case that the first term in a logical expression is being used as a guard, as in the example given above. The first term protects the execution of the second term, ensuring that it will not be executed in situations where doing so would raise an error condition.

An important feature to note in conjunction with the control flow messages defined by class `Boolean` is that they conceptually serve both as statements and as expressions. A smalltalk statement might be written as follows:

```
x > 0 ifTrue: [ x <- x - 1 ] ifFalse: [ x <- x + 1 ].
```

The statement will subtract one from the variable `x` if it is positive, or add one to `x` if it is negative. The same statement could be written as follows:

```
x <- x > 0 ifTrue: [ x - 1 ] ifFalse: [ x + 1 ].
```

Here we have factored out the assignment arrow, and the conditional expression is used to build the value that will ultimately be assigned to `x`. There is little performance difference between the two forms, so the selection of one over the other is largely a matter of personal taste.

Almost every function implemented by class `Boolean` makes use of the control flow statement `ifTrue:ifFalse:`, which *must* be redefined in each subclass. For example, the `and:` operator we used earlier is defined as follows:

```
and: aBlock
    ↑ self
        ifTrue: [ aBlock value ]
        ifFalse: [ self ]
```

The `or:` and `not` operators are similar. The control flow statements are also defined in relation to this one statement, as in the following:

```
ifTrue: aBlock
    ↑ self
        ifTrue: [ aBlock value ]
        ifFalse: [ nil ]
```

Such a method, which must be redefined in each subclass of a class, is in many object-oriented languages called a *pure virtual function.*

4.2 The subclasses `True` and `False`

We have seen that at the very least, the subclasses of the class `Boolean` must define the method `ifTrue:ifFalse:`. They do this by selecting one of the two arguments for evaluation, ignoring the second. For example, the method is defined in class `True` as follows:

```
ifTrue: trueBlock ifFalse: falseBlock
    ↑ trueBlock value
```

Once again, the use of blocks provides a delayed evaluation. Because the second argument is never passed the message `value`, the expressions within the block will never be executed.

For the sake of efficiency, the methods `and:` and `or:` are also redefined in subclasses. For example, the definition of the methods `and:` and `or:` in the class `True` is the following:

```
and: aBlock
    ↑ aBlock value

or: aBlock
    ↑ true
```

While the definitions in class `False` as slightly different:

```
and: aBlock
    ↑ false

or: aBlock
    ↑ aBlock value
```

In both cases the expressions used to implement the functions are simpler than their more general counterparts in the class `Boolean`.

4.2.1 Scandiavian versus American Overriding Semantics

The implementation technique used by the `and:` and `or:` operators described in the last section is made possible because Smalltalk uses the "American", or *replacement* semantics for overriding methods. In this interpretation, a method with the same name as a function inherited from the parent class totally replaces, or hides, the function from the parent class. When we execute the operation `and:` on a true or false boolean value, only the method from the subclasses `True` or `False` is executed; the method `and:` in the parent class `Boolean` is not touched.

An alternative semantics, called "Scandiavian", or *refinement* semantics, is found in a few alternative object-oriented languages, chiefly languages of Scandiavian origian (such as Simula [?], or Beta [?]). In these languages the code associated with a method in a subclass is *merged* with the code inherited for the function in the parent class. In this fashion, the code for the parent class is *always* executed.

Both forms of inheritance have advantages and disadvantages. The American semantics have a simpler implementation strategy, and permit the sort of uses as found in the `Boolean` class. On the other hand, they leave it uncumbant on the programmer to ensure the method defined in the child class is performing an activity that is somewhat similar to that defined by the parent class. There is nothing, for example, to prevent the programmer from performing

the `or:` operation under the `and:` name in the subclass, and vice-versa. In the scandiavian semantics, on the other hand, the programmer is guaranteed that actions in the parent class will always be executed, and thus a certain miminum level of functionality is ensured.

[note something about "super" being used to simulate refinement].

4.2.2 A Confession Concerning Optimization

The definition of control flow statements and expressions using messages such as `ifTrue:` and the like is an elegant proof that the message passing mechanism is, in theory, sufficiently powerful just by itself for the implementation of the Smalltalk programming langauge. Nevertheless, because boolean expressions are so pervasive in programs, in practice even the optimized expressions defined by the subclasses `True` and `False` are do not execute fast enough for a practical system. For this reason, the bytecode compiler (Chapter 4) treats as a special case all the methods defined by class `Boolean`, as well as the methods `whileTrue:` and `whileFalse:` from class `Block`. Whenever possible (for example, when the arguments are literal blocks) the compiler produces in-line code rather than using the message passing mechanism.

4.3 Other Control Flow Constructs

The second most common form of control flow, after condition statements, is the loop. A loop is formed from two parts; a *condition* that indicates when the loop should terminate, and an *action* that is performed each iteration of the loop. Because the evaluation of both parts must be delayed so that they can be reexamined each iteration, both parts are nested within block expressions. For this reason the loop messages, `whileTrue:` and `whileFalse:`, are implemented as methods in class `Block`, not class `Boolean`.

A typical loop is the following, which computes the product of the numbers from 1 to 100:

```
i <- 1.
prod <- 1.
[ i <= 100 ]
    whileTrue: [ prod <- prod * i. i <- i + 1 ]
```

It is interesting to note that the implementation of `whileTrue:` needs nothing more than recursion and conditional statements.

```
whileTrue: aBlock
    self value
        ifTrue: [ aBlock value. ↑ self whileTrue: aBlock ]
```

The implementation of `whileFalse:` is similar.

A loop which iterates over the values held by a collection is by convention implemented using the `do:` method. The following, for example, might be used to print the values held in a list.

```
'The list contains: ' print.
aList do: [ :ele | ele print. ' ' print ].
Char newline print.
```

The implementation of the `do:` statement depends upon the type of collection being manipulated. These will be discussed further in Chapter 7.

4.3.1 Adding New Control Flow Constructs

Because the control flow statements in Smalltalk are not a "built-in" part of the language, but are rather constructed out of more primitive units (namely blocks and message passing), it is relatively easy to create new control flow statements for special situations. We will demonstrate one example of this, by constructing a switch statement built on top of the cascade construct.

A switch statement first computes a test value, then selects one of many statements to execute depending upon the satisfaction of a number of conditions. The alternative statements can be either tested for exact match against the test value, or a more general test can be performed. Finally, a default case can be provided which will execute an associated statement should no other alternative match.

The following statement will illustrate these possibilities. The class method `on:` creates a new instance of class `Switch`. The switch statement will print one of a number of different messages, depending upon whether the value of the argument number is 2, 3, is even, or is negative. A default case is printed in all other cases.

```
Switch on: aNumber;
    case: 2 do: [ 'number is two' print ];
    case: 3 do: [ 'number is three' print ];
    if: [:x | x isEven ] do: [ 'number ' print. x print. 'is even ' print ];
    if: [:x | x < 0 ]
        do: [ 'number ' print. x print. ' is negative' print ];
    default: [ 'number does not match any condition' print ].
```

To implement the switch statement we first create a new class `Switch`. Instances of class `Switch` will maintain two instance variables, a test value, and a flag indicating if the switch has yet been satisfied. These will be called `test` and `done`, respectively. Thus, the class `Switch` can be created by the following command:

```
-> Object subclass: #Switch variables: #(test done)
```

Next, a class method must be created for the method `on:`. This method creates the new object, and sets the initial values of the instance variables. Within a class method instance variables are accessed by their index value, numbering from 1.

```
on: aValue | newValue |
    newValue <- self new.
    self in: newValue at: 1 put: aValue.  " variable test "
    self in: newValue at: 2 put: false. " variable done "
    ↑ newValue
```

The `if:do:` method takes a one argument test block, and evaluates the block with the current test value. If the result is true, and if an action has not already been performed, then the action block is executed, and the `done` flag is set.

```
if: testBlock do: actionBlock
    (done not and: (testBlock value: test))
        ifTrue: [ done <- true. actionBlock value ]
```

The `case:do:` method could be implemented in a similar fashion, or we could choose to implement it using the `if:do:` method we just defined.

```
case: aValue do: actionBlock
    ↑ self if: [:x | x = aValue ] do: actionBlock
```

Finally, the `default:` method simply needs to test the `done` flag, executing its associated action if no previous action has been successful.

```
default: actionBlock
    done not
        ifTrue: [ done <- true. actionBlock value ]
```

4.4 Case Study – Adding an "Unknown" Boolean

The division of class `Boolean` into the two subclasses `True` and `False` represents the normal mathematical view of logic, where assertions always either true or false. In more general settings, however, we often have three possibilities. Expressions can either be true, or they can be false, or they can be unknown. The creation of an "unknown" boolean value is an interesting exercise in the modification of system-provided classes.

The actual creation of the "unknown" global variable is relatively easy. All we need do is to create a new subclass of Boolean, then insert an instance of this class into the global dictionary.

```
-> Boolean subclass: #Unknown

-> globals at: #unknown put: Unknown new
```

Suppose we define the intended behavior of our unknown values by the following tables:

`and:`	unknown	true	false
unknown	unknown	unknown	unknown
true	unknown	true	false
false	false	false	false

`or:`	unknown	true	false
unknown	unknown	true	unknown
true	true	true	true
false	unknown	true	false

If we examine the default implementation of the `and:` method provided by class `Boolean` (given at the beginning of this chapter), we discover a surprizing fact. The default implementation returns the correct values, even for unknown arguments.

We are not so lucky, however, with the `or:` operator. The implementation of `or:` in class `Boolean` is as follows:

```
or: aBlock
    ↑ self
        ifTrue: [ self ]
        ifFalse: [ aBlock value ]
```

This produces the correct response for eight of the nine values in our truth table. The one case where it fails occurs when the left argument is unknown and the right argument is false, in which case we want the result to be unknown, rather than false. This is easily remidied by changing the implementation of the `or:` operator in class `Unknown` to the following:

```
or: aBlock
    ↑ aBlock value
        ifTrue: [ true ]
        ifFalse: [ self ]
```

It is also necessary to change the implementation of the `not` operator. The negation of an unknown value is no more known than the original.

```
not
    " still unknown "
    ↑ self
```

With the addition of an implementation of `ifTrue:ifFalse:` (simply execute the false branch) and a method to print the result we are finished with our creation of a new form of boolean.

Chapter 5

Magnitudes

The abstract class `Magnatude` is the parent class for objects that can be measured and compared against other values of the same type. The class hierarchy rooted at this class in the Little Smalltalk system is shown in Figure 5.1. The class `Association` and the subclasses of the `Collection` hierarchy represent basic data structures. These will be discussed in Chapters 7 and 8. The numeric subclasses will be discussed in Chapter 6. The remaining two classes, `Char` (character values) and `Symbol`, will be described here.

The class `Magnatude` itself defines only six simple methods, each only one line long, each shown in Figure 5.2. Although simple, these six methods illustrate an important way in which high-level abstract algorithms can provide a framework for solving a problem which can be specialized in different fashions by subclasses.

First consider the three relational operators (>=, > and <=). Notice how each of these is defined in terms of the remaining relational operator < and the equality testing operator = (inherited from class `Object`). A subclass need only redefine these two operators in order for all six relationals to be valid. The functions `min:` and `max:` are used to compute the smallest or largest of a pair of values. The last function `between:and:`, is used to determine if a value lays between a pair of bounds.

The important fact to realize is that there is only *one* copy of the function `between:and:` in the system. Yet this one function can be used in a variety of different ways, depending upon how subclasses have redefined the relational operator <. For example, the relational test is defined to be integer comparison by the integer subclasses. This use is found in the method `includesKey:` in the class `Array`, which is used to determine if an index value is valid for a given array.

```
includesKey: index
    ↑ index between: 1 and: self size
```

A quite different use is found in class `Char`, where the comparison test means character

- Object
  - Magnatude
    - Association
    - Char
    - Collection
      - Array
        - ByteArray
        - String
      - Dictionary
      - Interval
      - List
      - Tree
    - Numeric
      - Integer
        - LargeInt
        - SmallInt
    - Symbol

Figure 5.1: Subclasses of Class `Magnatude`

```
<= arg
    ↑ self < arg or: [ self = arg ]

> arg
    ↑ arg < self

>= arg
    self > arg or: [ self = arg ]

min: arg
    self < arg ifTrue: [ self ] ifFalse: [ arg ]

max: arg
    self < arg ifTrue: [ arg ] ifFalse: [ self ]

between: low and: high
    low <= self and: [ self <= high ]
```

Figure 5.2: The Methods defined by Class `Magnatude`

ordering. The following method is used to decide if a character value represents an upper case letter.

```
isUpperCase
    ↑ self between: $A and: $Z
```

The `String` class is indirectly a subclass of class `Magnatude`. For strings, relational operators mean a test based on lexicographic, or "dictionary" ordering. Thus the following will determine, for example, if a string would appear between the entries for "intaglio" and "intoxicant" if entered in a dictionary.

```
aString between: 'intaglio' and: 'intoxicant'
```

In all these cases the method defining `between:and:` is the same, despite their seeming differences.

5.1 The Class `Char`

The class `Char` is used to represent character values. Character values are most frequently used in the communication between the user and a running program. It is only character values that can printed on a screen, returned as input, be read or written to a file, and so on.

The protocol provided by class `Char` is shown in Figure 5.3. Three class methods are provided. The first reads is the basic method used to create new characters. The integer argument must be a value between 0 and 255, and represents the underlying ASCII representing of the value. The second class method is the fundamental mechanism for reading from the "standard input" (normally the working window). It reads the next input character, freezing execution until the next character is available. It returns the value `nil` when the user signals the end-of-file condition. The third and final class method simply returns the commonly used constant value representing the newline character.

The character testing methods are used to determine the type of value a character represents. These are extensively used, for example, by the bytecode parser during the processing a compiling the text representation of a method into bytecode form. The definition of each of these methods is similar to the `isUpperCase` method given earlier in the chapter.

The transformation of a lower case character into its equivalent upper case value is performed by doing arithmetic on the underlying representation value, as in the following method:

```
upperCase
    self isLowerCase
        ifTrue: [ ↑ Char new: (value - 97) + 65 ]
```

CLASS PROTOCOL

`Char new: anInteger`
: create a new character with given representation value (class method)

`Char input`
: return next character from standard input (class method)

`Char newline`
: return the constant representing a new line character (class method)

CHARACTER TESTING PROTOCOL

`isAlphabetic`
: return true if character represents a letter

`isAlphanumeric`
: return true if character represents a letter or a digit

`isBlank`
: return true if character represents a space, tab or newline character

`isDigit`
: return true if character represents a digit character

`isLowerCase`
: return true if character represents a lower case letter

`isUpperCase`
: return true if character represents an upper case letter

`< aChar`
: return true if representation is less than representation of argument

`= aChar`
: return true if representation is equal to representation of argument

CHARACTER TRANSFORMATION PROTOCOL

`lowerCase`
: if character is upper case, return lower case equivalent

`upperCase`
: if character is lower case, return upper case equivalent

`asString`
: return character as a length one string

`printString`
: return character as a length two string, following $ character

OTHER PROTOCOL

`print`
: display character on standard output device

`value`
: return representation value of character

Figure 5.3: Protocol for the Class `Char`

The remaining character methods are either trivial or are implemented as primitive operations.

5.2 The Class Symbol

A symbol is an array of character values, like a string. Differences between a string and a symbol include the fact that symbols, once created, cannot be modified, and that symbols are unique. That is, no two symbols have the same string values, and an attempt to create a new symbol with a string value equal to an existing symbol will simply yield the original symbol.

Symbols are measurable quantities, and compare using their underlying string representation. In fact, a symbol can be compared against either another symbol or against a string value. This fact is used in the creation and management of symbol values. All symbols are maintained in a tree held by a class variable named `symbols` in the class `Symbol`. The creation of a new symbol is performed by the class method `new:`, which is defined as follows:

```
new: fromString
    ↑ symbols at: fromString
        ifAbsent: [ symbols add: <23 fromString Symbol> ]
```

The collection of existing symbols is first scanned, using the argument (a string) as key. If any value matching the argument is found, the entry in the tree is returned. If no match is found, then a primitive operation (primitive number 23) is invoked to create a new instance of class `Symbol` with the same character values as the argument. (The same primitive is, by the way, used to clone string values). This new symbol is then inserted into the tree of existing symbol values.

Chapter 6

Numbers

Describe integers

SmallInts – positive values between 1 and 1000.

LargeInts – all other integer values. maintain sign as boolean, values as list of smallints.

Note: a number of large int operations remain to be defined.

6.1 Mixed Type Arithmetic

Describe the double dispatching technique used to resolve mixed type arithmetic expressions.

Chapter 7

Dynamic Collections

The collection classes implement the basic data structures that are fundamental to almost all nontrivial programs. For this reason alone it is important to understand the services provided by these classes. A complicated problem can often be greatly simplified by the selection of appropriate data structures. In addition, the design of these data types provides an excellent illustration of the use of object-oriented techniques in component definition. In particular, much of the functionality provided by the collection classes is implemented in high level, abstract classes, and specialized through deferred methods which are reimplemented in different ways in the various data structures.

The reader should be forwarned the the collection hierarchy in Little Smalltalk differs markedly; even more than most of the language, from the corresponding facilities provided by the various Smalltalk-80 systems. Not only is the structure of the hierarchy different, but even some of the classes and methods with similar names have slightly different meanings in the Smalltalk-80 system. As always, the objective in the design of the Little Smalltalk collection hierarchy was to provide a simple yet useful system, whereas the Smalltalk-80 system is much richer and more complete. We turn this simplicity into an asset, as the missing elements are an abundant source of exercises, both in the development of new data abstractions and the creation of methods for the existing abstractions. Section A.5 in Appendix A outlines some of the more significant differences between the Little Smalltalk collection classes and those in Smalltalk-80.

One major point of difference between the collection classes of Little Smalltalk and those of Smalltalk-80 is the fact that in Little Smalltalk the class Collection is a subclass of Magnitude. The operators = and < are generally defined in set terms for collections, for example set equality and the subset relation. This permits methods from class Magnitude, such as max:, min:, or between:and: (see Chapter 5) to be used with entire collections. For example, for strings the operators = and < can be thought of as defining lexicographic, or "dictionary" ordering. Thus, a test such as:

```
aString between: 'leaguer' and: 'lector'
```

could be used to determine if the value held in the variable aString would appear if entered in a dictionary between the entries for "leaguer" and "lector".

7.1 The Collection Class Hierarchy

There are seven basic classes in the Little Smalltalk collections, as well as the abstract class Collection. The class hierarchy for these classes can be described by the following chart, which also summarizes some of the characteristics of the different forms of collection.

Object	
Magnitude	
Collection	
List	*sequential linear list*
Tree	*binary tree ordered by value*
Array	*fixed size collection, indexed by position*
ByteArray	*array with small integer values*
String	*array with character values*
Dictionary	*indexed collection of key-value pairs*
Interval	*arithmetic progression*

Instances of class List represent a linear sequence of values. Items are usually added to or removed from the front of the list, although other possibilities are also provided. Thus, almost always a list is sequenced by the order of insertion. Elements held in a list need only recognize the equality testing message =. A list is a dynamic data structure, meaning it will grow and shrink as elements are added and removed.

Instances of class Tree represent an ordered collection, maintained in a binary tree. Elements are maintained in order based on their value. Since the tree is ordered, elements maintained by a tree must not only recognize the equality testing message =, but also the comparison message <. A tree is also a dynamic data structure.

There are two major categories of indexed collections. Array values are initially fixed in size (although they can be dynamically increased), and are indexed by integer keys, which range in value from 1 to the size of the collection. Arrays maintain in memory only the values in the collection, and not the keys. The amount of effort needed to access any element in an array is constant, independent of the size of the collection or the position of the value within the collection.

Two special forms of array are supported. A ByteArray is an array in which the values must be integers between zero and 255, and a String is a byte array in which the values are interpreted as printable characters. Instances of `ByteArray` are used chiefly as the means to maintain the bytecode instructions used by the Smalltalk virtual machine (see part IV).

In contrast to the Array subclasses, an alternative approach to defining an indexed data structure is represented by the class Dictionary. A Dictionary is a dynamic collection of key/value pairs. The keys need not be integer, but must be sequenceable. (An alternative, called an IdentityDictionary, which only requires elements to recognize the message ==, is discussed in Chapter xxx.) Both the keys and values are maintained in memory, and access time for any individual element may vary with the size of the collection or the position of the element within the collection.

The last form of collection supported by the Little Smalltalk system is the Interval. An Interval represents an arithmetic sequence; that is, a set of values *start*, $start + step$, $start + 2 \times step$, ..., *stop*. Only the starting value, ending value, and step size are actually maintained by the collection. All other values are generated as they are requested.

To simplify the presentation we have divided the discussion of the collection hierarchy into two chapters. In this chapter we will examine the dynamically sized collections (List, Tree and Dictionary), whereas in the following chapter we will discuss the fixed sized collection Interval and the subclasses of class Array.

7.2 The Abstract Class Collection

Although each of the collection classes define slightly different methods, they have in common the task of managing a group of values. For this reason a number of useful functions can be provided in the abstract class Collection. The protocol provided by class Collection is shown in Figure 7.1. The tasks of creating new elements, the addition of elements, and the removal of elements are all deferred to subclasses. However, the class Collection does provide support for the access of elements, the testing of elements, and the transformation of collections from one form into another.

The methods defined in the Collection class all assume the existance of the iteration method do:, which must be defined in each subclass. The do: method takes as argument a one-argument block, and evaluates the block on each element in turn. In indexed collections the keys are ignored, and only values are examined by the do: message.

Because the do: message is so simple to use, it is easy to overlook the true impact and power of the abstraction. Consider, for example, how iteration is performed in conventional languages, such as C or Pascal. A loop which will perform a certain task on each element of a linked list could be written in C as follows:

```
for (p = list->first; p; p = p->next) {
      ... some task ...
      }
```

For a linked list this is not too difficult, although it does have the disadvantage of exposing the implementation details; such as the fact that the list begins with a field named `first`, and each link is joined to the next by a field named `next`. However, the reader is invited to ponder how a loop can be written which will perform an arbitrary action on each

ACCESS AND ITERATION PROTOCOL

at: value ifAbsent: exceptionBlock
: return object in collection matching value

TESTING PROTOCOL

includes: anElement
: test whether an element is contained in the collection

isEmpty
: test whether the collection contains values

size
: count number of elements held by the collection

= aCollection
: test if two collections are equal

< aCollection
: test if a collection is a subset of another collection

TRANSFORMATION PROTOCOL

asArray
: convert a collection into an array

asString
: convert a collection into an array

asList
: convert a collection into a list

asTree
: convert a collection into a tree

collect: aBlock
: return a collection of transformed values

select: aBlock
: return collection of values that satisfy test

reject: aBlock
: return collection of values that do not satisfy test

Figure 7.1: The protocol provided by class Collection

element contained in, for example, a binary tree. Suffice to say that the task is not easy. But with the do: abstraction, a loop to traverse a binary tree is no more difficult (nor any different) from a loop traversing a list or an array.

7.2.1 Access Protocol

To test equality of two objects the binary method = is used. The default behavior is for this message to mean object identity, however this is redefined in many subclasses. Thus, object equality does not necessarily mean that two objects are exactly the same. The message at:ifAbsent: is used to retrieve the actual object in a collection that matches an argument value, returning the value of the exception block if no such object is found. The default implementation makes use of the do: method, cycling through all elements in the collection. This is redefined in the Tree class, where the the fact that elements are ordered permits a more efficient implementation.

```
at: value ifAbsent: exceptionBlock
        self do: [ :element | element = value ifTrue: [ ↑ element ] ].
        ↑ exceptionBlock value
```

Often the exception block will contain an explicit return statement, which can have the effect returning control to some earlier point and avoiding certain calculations, in much the same way that the return statement nested in the conditional shown here will prematurely terminate the loop, and the remaining values will not be examined.

7.2.2 Testing Protocol

The method includes: is testing to see if a particular value is included in the set of items maintained by the collection. To do so, it makes use of the at:ifAbsent: method just defined. We first ask if the element is in the collection, returning false if not so. The return statement in the exception block shown here is an example of the property discussed at the end of the last section. That is, if the exception block is evaluated, then execution of the includes: statement will be immediately terminated. If we fail to return a false value, then the element must be present in the collection, and a true value is yielded.

```
includes: anObject
        self at: anObject ifAbsent: [ ↑ false ].
        ↑ true
```

The default behavior for the < operator is a set-subset test. That is, one collection is less than or equal to the second if every item in the first is also found in the second.

```
< aCollection
        self do: [ :element | (aCollection includes: element)
```

```
                    ifFalse: [ ↑ false ] ].
        ↑ true
```

Two collections are equal if the subset test works in both directions. Note that this definition is overridden in the Array subclasses.

```
= aCollection
        ↑ self < aCollection and: [ aCollection < self ]
```

The method size tallies the number of items in the collection. It, too, is replaced in the array subclasses with a more efficient version.

```
size            | tally |
        tally ← 0.
        self do: [ :element | tally ← tally + 1 ].
        ↑ talley
```

The default implementation of the method isEmpty first computes the size of the collection, then tests this number of see if it is larger than zero. This may be an inefficient technique, but it is guaranteed to work. Subclasses override either or both the methods size and isEmpty when more efficient implementation techniques are available.

```
isEmpty
        ↑ size size ~= 0
```

7.2.3 Transformation Protocol

The methods provided by the transformation protocol in the class `Collection` are used to transform one collection into another, differing either in form, size, or value.

The method asList converts a collection into an `List`, by creating a new empty List and adding each value one by one. To do this the method addAll: provided by the class List can be used to perform the actual insertion. The new list preserves the ordering, if any, of the original collection.

```
asList
        ↑ List new addAll: self
```

The method asTree is similar.

```
asTree
        ↑ Tree new addAll: self
```

Note that a tree is an ordered collection, while a list is not. Thus, simply inserting a collection of values into a tree implicity sorts the values. One way to order a list of values is to convert it into a tree, then convert the tree back into a list.

```
aList ← aList asTree asList
```

The method asArray, which converts a collection into an array, is complicated by the fact that the size of the array must be declared when the array is created, and the addition of new elements into an array can only be performed if the index value is known. A temporary variable is used to maintain the current index. The method asString is similar.

```
asArray        | newArray index |
        newArray ← Array new: self size.
        index ← 1.
        self do: [ :element | newArray at: index put: element.
                        index ← index + 1 ].
        ↑ newArray
```

The method collect: is used to perform a transformation on every value of a collection. A new collection is formed to hold the transformed values. By default this collection is represented by a list, however this is modified in some subclasses (such as the class Tree). Elements are added to the list using the message addLast, which preserves any ordering the original structure may have possessed.

```
collect: transformBlock | newList |
        newList ← List new.
        self do: [ :element | newList addLast: (transformBlock value: element) ].
        ↑ newList
```

Two methods are used to select subsets of an existing collection. The method select: takes a one-argument block which is assumed to return a boolean value, and returns a new collection containing only those values which yielded a true value when tested by the argument. The default version of select: yields a list, however this function is overridden in several classes to yield other varieties of objects. As with the collect: message, the use of addLast rather than add preserves the ordering of the collection.

```
select: testBlock | newList |
        newList ← List new.
        self do: [ :element | (testBlock value: element)
                        ifTrue: [ newList addLast: element ] ].
        ↑ newList
```

The inverse is the function reject:, which returns those values which yielded a false value. The select: method is, as we have noted, redefined in many subclasses. However, the reject: method can be defined once, by simply inverting the results of the test.

```
reject: testBlock
        ↑ self select: [ :element | (testBlock value: element) not ]
```

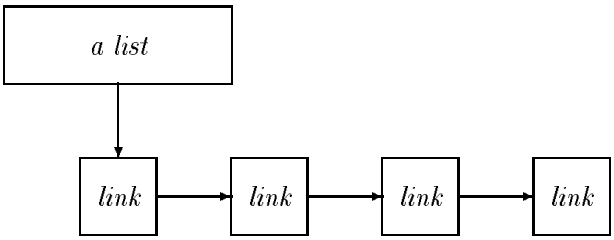
The implementation of reject: illustrates one way in which programming in Smalltalk differs from programming in a strongly-typed object-oriented language, such as C++. At the time we write this method we cannot know the type of result it will yield. It could be a list, a tree, or some other subclass of Collection. This indeterminacy would not be permitted if we were required to declare the return type. Thus, similar techniques cannot be used in the strongly typed language C++. Nevertheless, such methods are exceedingly powerful and useful, and represent a rich source of component reuse in Smalltalk. There are numerous similar examples to be found in the Smalltalk class hierarchy.

7.3 Class List

The sequenced linked-list data abstraction is perhaps the simplest dynamic data-type. Elements can be added either to the front or the end of the structure, and are generally removed from the front (although a method is provided to remove arbitrary elements from the collection). The protocol defined by the class List is shown in Figure 7.2.

7.3.1 Addition Protocol

The actual data values in a linked list are maintained by instances of the class Link. Each Link maintains a single value, and the next link in the chain. The last link holds a nil value in the next field.



The class List simply holds, in an instance value named elements, the first Link value. Thus, for example, to add a new value to the front of a list simply involves creating a new instance of class `Link`. The value returned by the method is the newly inserted quantity.

```
add: anElement
        elements ← Link value: anElement next: elements.
        ↑ anElement
```

Adding a value to the end of a list is more difficult, as the List data structure does not maintain a pointer to the end of the list. Instead, assuming the list is nonempty, the message is passed to the first link element.

CREATION PROTOCOL

List new
: create new list (Class method)

ADDITION AND MODIFICATION PROTOCOL

add: anElement
: add an element to the front of the list

addLast: anElement
: add an element to the end of the list

addAll: aCollection
: add all elements of a collection to the end of the list

ACCESS AND ITERATION PROTOCOL

do: aBlock
: perform block on each element of the list

reverseDo: aBlock
: perform block on each element, in reverse order

first
: return first element in list

REMOVAL PROTOCOL

remove: anElement
: remove element from list, signal error if not found

remove: anElement ifAbsent: exceptionBlock
: remove element from list, returning exception block if not found

removeFirst
: remove first element from collection, returning updated list

TESTING PROTOCOL

isEmpty
: return true if list is empty

TRANSFORMATION PROTOCOL

copy
: copy elements of list into a new list

reverse
: return new list with elements in reverse order

Figure 7.2: Protocol provided by the class List

```
addLast: anElement
        elements isNil
                ifTrue: [ self add: anElement ]
                ifFalse: [ elements addLast: anElement ].
        ↑ anElement
```

The method addLast in class Link simply passes the element down the list, until the final link is reached, at which point it is inserted. Although the return value is not used by the addLast method in class List, the return operator in class Link is used in order to form a type of recursive call called *tail-recursion*. This form of recursive call is optimized by the virtual machine interpreter, resulting in more efficient use of memory. Tail recursion optimization is useful in situations where the number of recursive calls may be large. Here the number of calls matches the number of elements in the list.

```
addLast: anElement
        next notNil
                ifTrue: [ ↑ next addLast: anElement ]
                ifFalse: [ next ← Link value: anElement ]
```

The method addAll: adds an entire collection to the list. The values are added to the end, thus preserving any ordering that may have existed in the original list.

```
addAll: aCollection
        aCollection do: [ :element | self addLast: element ]
```

To append one list to the end of another without changing either, we can copy the first list, then use addAll: to attach the second.

```
appendedList ← firstList copy addAll: secondList
```

Asymptotic Complexity

Although the add: and addLast: methods appear to be performing similar tasks, there is one very important respect in which they differ. The add: method performs the same sequence of operations regardless of how many items are currently being maintained by the list. The addLast: method, on the other hand, must iterate through each element of the list until it locates the end. Thus, adding an element to the end of a long list will take more time than adding to the end of a shorter list.

We describe this difference using the notation of *asymptotic complexity*. Because of differences in machine speeds, quality of code generation, and so on we cannot say with precision how fast any particular method will execute. But we *can* say that the execution time of the addLast: method is (roughly) proportional to the length of the list. If we let n represent the number of elements in the list, we denote this by saying that the execution

time for the addLast: method is $O(n)$, which is read as "order-n". The execution time for the add: method, on the other hand, is described as $O(1)$, which is read as "order-1" or "order-constant", meaning its execution time is constant (proportional to some constant value).

Sometimes the execution time of a method will vary depending upon the values encountered. An example is the remove: method we will soon investigate, which can be very fast if the element being removed is found near the front of the list, and slower if the element is found near the end. In such cases the asympototic execution time refers to the *worst case* execution time. In the case of remove:, the worst case occurs when every element is examined and none succeed in matching. Thus, the execution time for remove: is $O(n)$.

When operations are combined, the asymptotic complexity refers to the complexity of the entire operation. How this is calculated from the complexities of the individual parts depends upon how the operations are combined. For example, consider the method addAll: discussed in the last section. Let m represent the size of the collection being added, and n represent the initial size of the list to which the collection is being appended. The do: method for lists (as well as for all the collections we will consider) is $O(n)$. But since the activity described in the block is (in the worst case) executed on every element, the asympototic execution time for an entire invocation of do: is the execution time of the do: *times* the execution time of the action. In this case the action, an addLast: operation, is $O(m)$. Thus the total asympototic execution time for addAll: method is $O(n \times m)$.

We will note the asymptotic complexity of various methods we describe, and leave the discovery of yet more complexities as exercises.

7.3.2 Access and Iteration Protocol

The fundamental method do: is simply a wrapper for the invocation of the similarly named function in class Link.

```
do: aBlock
        elements notNil
                ifTrue: [ elements do: aBlock ]
```

The method in class Link is recursive. It first executes the block on the current value, then passes the block on to the next value.

```
do: aBlock
        aBlock value: value.
        next notNil ifTrue: [ ↑ next do: aBlock ]
```

As the do: method examines every element in the list, it's asympototic complexity is, as we have already noted, n times the complexity of whatever task is being performed on each value.

The method reverseDo: examines elements in the reverse order, from last to first. As before, the implementation in class List simply invokes the similarly named method in class Link, which is defined as follows:

```
reverseDo: aBlock
        next notNil ifTrue: [ next reverseDo: aBlock ].
        aBlock value: value
```

The method first is used to return the first element of a list. It signals an error if there are no elements in the list.

```
first
        elements notNil
                ifTrue: [ ↑ elements value ]
                ifFalse: [ ↑ self error: 'attempt to remove from an empty list']
```

7.3.3 Removal Protocol

The most common removal method for linked lists simply removes the first element in the list. The method signals an error if the list is empty, otherwise returning the list with the first element removed. The next method in class Link simply yields the value of the similarly named instance variable.

```
removeFirst
        elements isNil
                ifTrue: [ self error: 'removing first element from empty list' ]
                ifFalse: [ elements ← elements next ]
```

It is possible to remove arbitrary elements from a list. The remove: method removes the first instance of a value that matches the argument, signaling an error if no such element exists. It is implemented by making use of the more general method named remove:ifAbsent:, which explicity provides for the value to be used in the event no matching value is found.

```
remove: anElement
        self remove: anElement
                ifAbsent: [ self error: 'removing element not in collection' ]

remove: anElement ifAbsent: exceptionBlock
        elements isNil
                ifTrue: [ exceptionBlock value ]
                ifFalse: [ elements ← elements remove: anElement
                                ifAbsent: exceptionBlock ]
```

The work is performed in class Link, which recursively runs down the list until the element is encountered or the end is reached.

```
remove: anElement ifAbsent: exceptionBlock
        value = anElement
                ifTrue: [ ↑ next ]
                ifFalse: [ next notNil
                        ifTrue: [ next ← next remove: anElement
                                                ifAbsent: exceptionBlock.
                                ↑ self ]
                        ifFalse: [ ↑ exceptionBlock value ] ]
```

Despite the seeming complexity of this method, it is still only performing a constant amount of work on each element, and in the worst cast must examine each and every element in the list. Thus, the execution time is still described as being $O(n)$.

7.3.4 Testing Protocol

We can avoid counting the elements to see if a list is empty, and instead simply examine the link to the first element. If the link is empty, then there are no values in the list.

```
isEmpty
        ↑ elements isNil
```

The resulting function is $O(1)$, whereas the original version inherited from class Collection is $O(n)$. For long lists this can be a significant savings in execution time.

7.3.5 Transformation Protocol

As we have seen, a copy of list is simply the value returned by passing the message asList to an existing list.

```
copy
        ↑ self asList
```

While this is simple to describe, it is not the most efficient implementation possible. As we have seen, the execution time for addAll: is the length of the list times the length of the new list, which are both n. Thus the copy operation, as implemented here, is $O(n^2)$. Intuitively, it would seem as if it should be possible to produce a copy of a list in fewer operations. Indeed this is true, and is a topic investigated more fully in the exercises.

The reversal of ordering for a list is easily accomplished by iterating over the list structure and adding each new element. As the add operation adds to the front of the list, the result is a reversal.

```
reverse | newList |
        newList ← List new.
        self do: [ :element | newList add: element ].
        ↑ newList
```

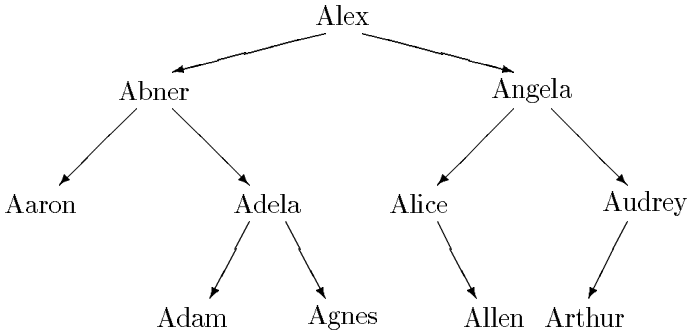
The following chart illustrates the execution of the reverse method on a list consisting of five values.

original list	*new list*
1 3 4 7 9	1
1 3 4 7 9	3 1
1 3 4 7 9	4 3 1
1 3 4 7 9	7 4 3 1
1 3 4 7 9	9 7 4 3 1

7.4 Class Tree

The *tree* data abstraction differs from the list in that values are sequenced not by their time of insertion, but by comparisons between the values themselves. To do this, the class Tree manages an ordered binary tree of values, organized as a binary search tree. Nodes in the tree are implemented by the class Node, which serves an analogous purpose to the class Link in the list abstraction.

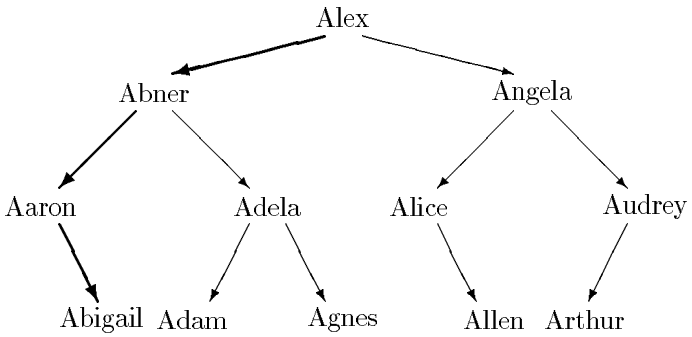
The following binary tree illustrates some of the properties of trees, and the advantages of trees over the linked list abstraction. Each node in the tree maintains a name as a value. Nodes may have as subtrees either a left child, or a right child, or both. Furthermore, the name associated with each node is lexicographically greater than the names in the left child subtree (if any) and lexicographically smaller than the names in the right child subtree (again, if any).



If we traverse the tree in a specific fashion we can access the values in the collection in alphabetical order. This traversal is called a *pre-order* traversal, and can be described as a recursive three-step process:

1. Traverse over the values associated with the left child subtree, if any
2. Process the value of the current node
3. Traverse over the values associated with the right child subtree, if any

Finally, to illustrate the execution advantage of trees over linked lists, consider the task of adding a new value to the collection. Let us add, for example, the name "Abigail". We first compare the name "Abigail" to the root of the tree, which is "Alex". Since "Abigail" is smaller, we move down the left child, which brings us to the name "Abner". Again, "Abigail" is smaller, and so we continue to move left. A comparison between "Aaron" and "Abigail", however, shows that "Abigail" is the larger value, and thus we should insert it into the right subtree. But "Aaron" has no right subtree. So a new node is created, and the right subtree of "Aaron" is changed to point to this new node.



The most important point to note is that in performing the insertion we have compared the value against only three other nodes ("Alex", "Abner" and "Aaron"). These are the nodes that lie on the *path* between the root of the tree and the leaf where the insertion is performed. The majority of nodes in the tree played no part in the insertion process. An important property of binary trees is that if the tree is well balanced, then the length of the path between root and leaf is roughly the logarithm (base 2) of the number of values in the tree.[1] Thus, if certain care is maintained so that trees remain balanced, insertion, testing and removal operations on trees can all be made $O(\log n)$ operations.

The protocol for class Tree, shown in Figure 7.3, is very similar to that of class List. And, as we will see, many of the methods are similar as well, differing only in the methods in class Node, which perform the majority of the work. The root of the binary tree is maintained in an instance variable named, appropriately, root.

[1]This is a big if. If a tree remains well balanced then additions and removals are very fast. If a tree becomes unbalanced, for example leaning entirely to the right or left, then the tree degenerates into a linked list, and operations become $O(n)$ rather than $O(\log n)$. In Chapter ?? we will investigate techniques that can be used to keep trees balanced.

CREATION PROTOCOL

Tree new
: create a new instance of class Tree (class method)

ADDITION AND MODIFICATION PROTOCOL

add: anElement
: insert a new element into the tree

addAll: aCollection
: add all the elements from a collection into a tree

ACCESS AND ITERATION PROTOCOL

at: anElement ifAbsent: exceptionBlock
: test if an element is contained in the collection

do: aBlock
: perform block on each element of the tree

reverseDo: aBlock
: perform block on each element, in reverse order

first
: return first element in collection

REMOVAL PROTOCOL

removeFirst
: remove first element from collection

remove: element
: remove value from collection, signalling error if not found

remove: element ifAbsent: exceptionBlock
: remove element from tree, returning exception block if not found

TESTING PROTOCOL

isEmpty
: return true if tree is empty

TRANSFORMATION PROTOCOL

copy
: create copy of the collection

collect: transformblock
: form new collection of transformed values

select: testBlock
: return tree of selected items

Figure 7.3: Protocol provided by the class Tree

7.4.1 Addition Protocol

To add a new value to a binary tree either a new node is created (if the tree is empty), or the `add:` message is passed to the root value for insertion. The newly inserted value is returned as the result.

```
add: anElement
        root isNil
                ifTrue: [ root ← Node new: anElement ]
                ifFalse: [ root add: anElement ].
        ↑ anElement
```

The work of the insertion is performed by a method in class Node. Nodes maintain as instance variables three quantities, the value of the current node, and the two left and right subtrees. The add: method recursively walks down until it has encountered a leaf node in the tree, then inserts the value.

```
add: anElement
        value < anElement
                ifTrue: [ right notNil
                        ifTrue: [ right add: anElement ]
                        ifFalse: [ right ← Node new: anElement ] ]
                ifFalse: [ left notNil
                        ifTrue: [ left add: anElement ]
                        ifFalse: [ left ← Node new: anElement ] ]
```

Adding an entire collection of values is simply a matter of iterating over each and adding them one by one.

```
addAll: aCollection
        aCollection do: [ :element | self add: element ]
```

7.4.2 Access and Iteration Protocol

The default implementation of the message at:ifAbsent: cycles through the entire list. Since a tree is ordered, a more efficient implementation can simply follow the path from root to leaf. The method in class `Tree` passes the message on to the topmost node, if the collection is nonempty.

```
at: key ifAbsent: exceptionBlock
        root isNil
                ifTrue: [ ↑ exceptionBlock value ]
                ifFalse: [ ↑ root at: key ifAbsent: exceptionBlock ]
```

Class Node traverses the path from root to leaf. This is similar to the insertion algorithm.

```
at: key ifAbsent: exceptionBlock
        value = key ifTrue: [ ↑ value ].
        value < key
                ifTrue: [ right notNil
                        ifTrue: [ ↑ right at: key ifAbsent: exceptionBlock ]
                        ifFalse: [ ↑ exceptionBlock value ] ]
                ifFalse: [ left notNil
                        ifTrue: [ ↑ left at: key ifAbsent: exceptionBlock ]
                        ifFalse: [ ↑ exceptionBlock value ] ]
```

The do: and reverseDo: messages both simply pass the commands on to the root node, once it has been verified that the tree is nonempty.

```
do: aBlock
        root notNil ifTrue: [ root do: aBlock ]

reverseDo: aBlock
        root notNil ifTrue: [ root reverseDo: aBlock ]
```

The work is performed by the class Node, which simply performs a recursive traversal of the binary tree. For a normal iteration the traversal is left to right, for a reversed iteration it is right to left.

```
do: aBlock
        left notNil ifTrue: [ left do: aBlock ].
        aBlock value: value.
        right notNil ifTrue: [ right do: aBlock ]

reverseDo: aBlock
        right notNil ifTrue: [ right do: aBlock ].
        aBlock value: value.
        left notNil ifTrue: [ left do: aBlock ]
```

The method first is used to return the smallest element maintained by the tree. This is always the leftmost node. A recursive algorithm in class Node walks down the left links until it can go no further, then returns the current value.

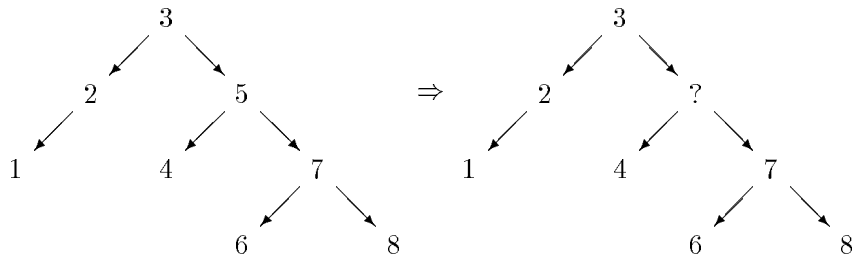
```
first
        left notNil
                ifTrue: [ ↑ left first ]
                ifFalse: [ ↑ value ]
```

The method in class Tree simply ensures that there is at least one element in the collection.

```
first
        root notNil
                ifTrue: [ ↑ root first ]
                ifFalse: [ ↑ self error: 'accessing first element in empty tree']
```

7.4.3 Removal Protocol

Removing an element from a binary tree is more difficult than removing an element from a linear list. To see why, consider the task of removing the value 5 from the following tree. Removing the value 5 leaves a "hole". Which value should be used to fill this void?



Before we describe the solution to this problem we first solve a slightly simpler case, namely the removal of the first element in the collection, which is always the leftmost child of the leftmost subtree. This is implemented in class Node by the following method:

```
removeFirst
        left notNil
                ifTrue: [ left ← left removeFirst ]
                ifFalse: [ ↑ right ]
```

The method is invoked in the class Tree as follows:

```
removeFirst
        root isNil
                ifTrue: [ self error: 'removing first from empty tree']
                ifFalse: [ root ← root removeFirst ]
```

Note that the default value returned by the method in class Node is the current node. By diagramming the trees that result from systematically removing the values 1 to 8 in the

tree given earlier the reader can verify that the function returns the expected result in all cases.

The solution to the general case removal is now solved using the methods first and removeFirst. The algorithm can be expressed in pseudo-code as follows:

when we find the node to delete
    if it has a right child
        set current value to first value in right subtree
        remove first value from right substree
        return current node
    else if no right child, then return left child (if any)

Adding the code to handle the traversal from root to the given node, plus the exceptional case when no matching node is found, results the following method in class Link.

```
remove: key ifAbsent: exceptionBlock
    value = key ifTrue:
        [ right notNil ifTrue:
            [ value ← right first.
            right ← right removeFirst.
            ↑ self ]
        ifFalse: [ ↑ left ] ].
    value < key
        ifTrue: [ right notNil
            ifTrue: [ right ← right remove: key ifAbsent: exceptionBlock ]
            ifFalse: [ ↑ exceptionBlock value ] ]
        ifFalse: [ left notNil
            ifTrue: [ left ← left remove: key ifAbsent: exceptionBlock ]
            ifFalse: [ ↑ exceptionBlock value ] ]
```

Although this is one of the longest methods in the collection hierarchy, all activities are still restricted to simply making changes along a path from root to leaf. Thus in a relatively well balanced tree the complexity will still remain $O(\log n)$.

7.4.4 Testing Protocol

As with the class List, it is possible to determine whether or not a tree is empty without counting the number of elements.

```
isEmpty
    ↑ root isNil
```

7.4.5 Transformation Protocol

The copy, collect: and select: methods are similar to those of class List, differing only in that they construct trees rather than lists.

```
copy
        ↑ self asTree

collect: transformBlock | newTree |
        newTree ← Tree new.
        self do: [ :element | newTree add: (transformBlock value: element) ].
        ↑ newTree

select: testBlock | newTree |
        newTree ← Tree new.
        self do: [ :element | (testBlock value: element)
                ifTrue: [ newTree add: element ] ].
        ↑ newTree
```

7.5 Class Dictionary

The class Dictionary maintains a collection of key/value pairs. Unlike the class Array, where the keys must be integers, in the class Dictionary the key values can be any object. (Most often keys are instances of class Symbol or class String, although this is not universally true). The actual data is maintained by an instance variable named data which holds an instance of class Tree. The key/value pairs themselves are maintained by instances of class Association, which simply holds two data fields for the key and value items. The protocol associated with class Dictionary is shown in Figure 7.4.

7.5.1 Creation Protocol

The Tree data value is initialized in the method new, used to create an instance of class Dictionary. This class method overides the method inherited from class Class. The message in:at:put: is the means normally used in class methods for initializing the instance variables associated with a newly created object. The index represents the index of the instance variable in the object. The method returns the newly created and initialized object.

```
new
        ↑ self in: super new at: 1 put: Tree new
```

CREATION PROTOCOL

Dictionary new
: create a new empty Dictionary (class method)

ADDITION AND MODIFICATION PROTOCOL

at: key put: value
: add value to collection using given key, or modify value at key

ACCESS AND ITERATION PROTOCOL

at: aKey
: return item associated with key, yielding error if none

at: key ifAbsent: exceptionBlock
: return value associated with key, or exception block if not found

do: aBlock
: iterate through all values

binaryDo: binaryBlock
: iterate through all key/value pairs

REMOVAL PROTOCOL

removeKey: aKey
: remove item in collection associated with given key, or yield error

removeKey: aKey ifAbsent: exceptionBlock
: remove value associated with given key

TESTING PROTOCOL

isEmpty
: return true if collection is empty

Figure 7.4: Protocol provided by the class Dictionary

7.5.2 Addition Protocol

Items are added to a Dictionary by means of the method at:put:, which specifies both the key and value. If an entry associated with the key exists already then the value is simply changed. If not, then a new association must be added to the collection. These two tasks are performed by the method at:ifAbsent: in class Tree. The method searches for a value with the given key, executing the exception block if none is found. In this case, the exception block inserts a new association into the tree, and returns. Otherwise, if the at:ifAbsent: method returns a value it must be an instance of Association. In this case the method value: is used to change the value field of the pair.

```
at: key put: value | anAssociation |
        anAssociation ← data at: key
                ifAbsent: [ ↑ data add: (Association key: value: value ) ].
        anAssociation value: value
```

One important fact should be noted. Recall that the tree of data values is maintaining a collection of associations. When the search of the data tree is first performed each association is being compared against a key value. When the addition operation is performed, each association is being compared against another association. This dual use of the = and < operators is accomplished in class Association by explicitly testing the type of the argument value:

```
= aKey
        aKey class == Association
                ifTrue: [ ↑ key = aKey key ]
                ifFalse: [ ↑ key = aKey ]
```

7.5.3 Access and Iteration Protocol

As with the class Array, the method at: simply involks the more general method at:ifAbsent:, which in turn simply uses the similarly named method for the tree data structure. The result, if an exception does not occur, is an instance of class Association. The value message is used to extract the value field from the association.

```
at: key ifAbsent: exceptionBlock | anAssociation |
        anAssociation ← data at: key ifAbsent: [ ↑ exceptionBlock value ].
        ↑ anAssociation value
```

The do: method makes use of the more general binaryDo: method, which iterates over both keys and values. The latter, in turn, is implemented using the do: method for the underlying tree.

```
do: aBlock
        self binaryDo: [ :k :v | aBlock value: v ]

binaryDo: aBlock
        data do: [ :anAssociation |
                aBlock value: anAssociation key value: anAssociation value ]
```

7.5.4 Removal Protocol

To remove an value from a dictionary only the key is specified, using the method removeKey:. This method is implemented in terms of the more general removeKey:ifAbsent:, which in turn simply uses the method remove:ifAbsent: on the underlying tree.

```
removeKey: aKey
        self removeKey: aKey
                ifAbsent: [ self error: 'removing element not in dictionary']

removeKey: aKey ifAbsent: exceptionBlock
        tree remove: aKey ifAbsent: exceptionBlock
```

7.6 Example Application – A Concordance

The development of a concordance application provides an illustration of how the selection of appropriate data structures can greatly simplify the solution of certain problems. A concordance is an alphabetical list of words in a text, that shows the line numbers on which each word occurs.

The creation of a concordance is divided into two steps. First, the concordance is generated (by reading lines from an input stream), and then the result is printed. The class Concordance will reflect this separation, having different methods for creating and printing the concordance. The class Concordance maintains a single instance variable, named dict, which holds a Dictionary. This dictionary will be indexed by string values (the words). The value maintained by the dictionary will be lists of line numbers.

The method word:occursOnLine: is the method used to update the dictionary of words. It is defined as follows:

```
word: word occursOnLine: line
        (dict includes: word)
                ifFalse: [ dict at: word put: List new ].
        ((dict at: word) includes: line)
                ifFalse: [ (dict at: word) addLast: line ]
```

The first task the method performs is to determine whether or not the word is already present in the dictionary, that is, whether or not the word has been seen previously. If not, then a new entry is made in the dictionary with the word as key an an empty list as value. Regardless of the results of the first test, the list indexed by the word is then examined to see if the line number occurs already. This step is necessary since the same word may be used twice on the same line. If not, then the line number is added to the list associated with word. Adding the line number of the end of the list ensures the numbers will be reported in ascending order.

To create the concordance, the method fileIn reads a sequence of lines, converts them to all lowercase, then breaks them into individual words. The method break: will be examined more fully in the next chapter. It takes as argument a string, and returns a list of words, the argument to the break: method being used to define the separator characters between words. In this case the separator characters are spaces, periods, and commas. Once the list of words has been generated, the do: method is used to iterate over the collection, adding each word to the dictionary.

```
fileIn | text lineNumber words |
        lineNumber ← 0.
        [ text ← String input. text notNil ]
                whileTrue: [
                        text ← text collect: [ :c | c lowerCase ].
                        lineNumber ← lineNumber + 1.
                        words ← text break: ' .,'.
                        words do: [ :word | self word: word
                                occursOnLine: lineNumber ] ]
```

Displaying the resulting dictionary simply requires a pair of nested loops:

```
displayDictionary
        dict binaryDo: [ :word :lines |
                word print.
                lines do: [ :line | ' ' print. line print ].
                Char newline print ]
```

The final method main simply puts all the pieces together. It initializes the dictionary, then invokes the fileIn method, and ends by calling the display method.

```
main
        dict ← Dictionary new.
        self fileIn.
        self displayDictionary
```

If we run the program on the text

It was the best of times,
it was the worst of times.

for example, the output, from best to worst, would be:

best 1
it 1 2
of 1 2
the 1 2
times 1 2
was 1 2
worst 2

Exercises

1. Assume the do: method executes in $O(n)$ steps. What is the asymptotic execution time of the following methods as defined in class Collection.
   (a) at:ifAbsent:
   (b) includes:
   (c) size
   (d) $<$
   (e) $=$
2. Assume we have an instance of class Tree. Recall that the at:ifAbsent: method is redefined in this class, so that it uses only $O(\log n)$ execution steps. Hows does this change the asymptotic complexities of the other method listed in the previous question.
3. Is it faster to convert a collection into a list, into a tree or into an array? Compute the asymptotic complexity of the methods asList, asTree, and asArray.
4. Using a large collection of randomly generated values, produce empirical timings for the three transformation methods described in the previous questions. Does the ranking given by a comparison of the asymptotoic analysis seem to hold in practice?
5. The section on lists suggests the following algorithm for generating a list that represents two argument lists appended end to end.

   appendedList ← firstList copy addAll: secondList

   What is the asympototic complexity of this approach? Can you think of a faster algorithm? What restrictions does your faster algorithm require? (For example, does it require that the two argument collection recognize the message reverseDo?)

6. As defined in class List and class Tree, the remove: method eliminates the first value that matches the argument. Suppose, instead, you wished to remove *all* matching values. Write a method removeAll to perform this task. Can you do this without adding any new functionality to class Link or class Node? (Hint: you can break out of loops using a return statement from within an exception block).

7. As noted in the text, the default implementation of copy for lists is $O(n^2)$, due to the inefficiency of the asList method when the argument is itself a list. Can you produce a $O(n)$ implementation of asList? (Hint: make use of reverseDo:). Are any other methods besides copy also speeded up by your new method?

8. What is the asymptotic complexity of the method first sent to an instance of Tree, assuming the tree is relatively well balanced? How about removeFirst?

9. Simulate the removal of the values 1 to 8, in sequence, from the binary tree shown on page 57. Show what the tree looks like after each value has been removed.

10. Show what trees will result from the initial tree shown in page 57 if the values are removed in the sequence 3, 8, 7, 4, 2, 5.

11. Give the asympototic execution time for each method is the Dictionary protocol, assuming the underlying tree value remains relatively well balanced.

Chapter 8

Fixed Sized Collections

In this chapter we continue the investigation of the collection hierarchy in Little Smalltalk that we started in the last chapter. The previous chapter discussed dynamic data structures; collections that grow and shrink as elements are added and removed from the collection. In this chapter we will consider fixed sized collections. In these collections the number of elements is fixed when the collection is created, and does not change during the course of execution.

There are two major categories of fixed sized collections. The biggest category is represented by the class Array and its subclasses ByteArray and String. Arrays are indexed structures, as was the class Dictionary studied in the previous chapter. However, the index values for an array represent a position within the collection. Index values are restricted to integers between 1 and the size of the collection. The benefit of arrays over the more general dictionary data type is that the time to access any element is very fast. In particular, the amount of work needed to access any element in an array is independent of the size of the array or the position of the value. Thus, the asymptotoic complexity of array access is always $O(1)$. With a dictionary, the access time for elements is $O(\log n)$ at best, and can be much worse if the underlying tree data structure which is holding the values becomes unbalanced.

Arrays and Strings are the only collection datatype that can be expressed as Smalltalk constants. An array literal is a sequence of constant values surrounded by the characters `#(` and `)`. The elements of an array literal can be integers, strings, characters, symbols, or other arrays. The following literal exhibits all of these possiblities:

```
#( $c 3 'abc' #abc #(1 2))
```

The class ByteArray is a form of array in which the values being held are restricted to being integers between 0 and 255 (that is, byte values). Such arrays are treated as a special case by the Little Smalltalk system because they can be very efficiently stored in memory – much more efficiently than the general array structure. There are two major uses for

ByteArrays in the Little Smalltalk system. The first is as the holding object for the virtual machine instructions in the compiled representation of methods. We will investigate this representation in more detail in Part ??. The second major use of byte arrays is as the parent class for strings.

A string is a textual value – a sequence of characters. Strings can be printed and stored in files, and are the primary means of communication between a running program and the user.

The second major category of fixed size collections in the Little Smalltalk system is the Interval. As we noted in the last chapter, an interval represents an arithmetic progression. However, the actual values in the interval are never stored in memory. Instead, an algorithm is used which generates the values on demand as they are needed.

In the following sections we will investigate in more detail the implementation of these classes.

8.1 Class Array and Class ByteArray

As we noted at the beginning of this chapter, the class Array is used to represent sequenced collections of values with fixed extent. Elements are referred to using an integer index, which represents the position of the value within the array (the value one being used to represent the first position). Unlike lists, which are also sequenced collections, the access time for each element in an array is independent of the position of the value within the array.

While the index keys must be integer, the values maintained by an array can be any arbitrary object. A subclass, ByteArray, differs from an array in that the values are restricted to being integers in the range zero to 255 (that is, bytes). The class String is a further subclass of ByteArray, in which each byte value is interpreted as a character. The protocol provided by the class Array is shown in Figure 8.1.

8.1.1 Creation Protocol

Array values cannot be created with the message new, instead the class method new: (with an argument) must be employed. The argument specifies the number of elements the array will maintain. This method, like many in the class, is implemented by a primitive operation.

```
new: size
        ↑ < 7 Array size >
```

8.1.2 Access and Iteration Protocol

Access to array values is given by the method at:, which in turn is implemented using the more primitive method at:ifAbsent:, which specifies the action to take if the index is out of bounds. To test the index value the method includesKey: is invoked.

CREATION PROTOCOL

Array new: size
: create a new array with given size (Class method)

ACCESS AND ITERATION PROTOCOL

at: index
: return value designated by index

at: key ifAbsent: exceptionBlock
: return value associated with key or exception value

do: aBlock
: iterate over elements of collection

reverseDo: aBlock
: iterate over elements of collection in reverse order

ADDITION AND MODIFICATION PROTOCOL

at: key put: value
: modify value at given key, error if key is out of range

with: anElement
: return new array one element larger, adding value at end

+ anArray
: return new array with elements of argument array catenated to end

TRANSFORMATION PROTOCOL

copy
: return a duplicate of the array

TESTING PROTOCOL

includes: anElement
: return true if array holds given value

includesKey: index
: return true if key is in range of legal index values

size
: return number of elements maintained by array

= anArray
: return true if elements match those in argument array

< anArray
: return true if array is lexicographically smaller then argument

Figure 8.1: Protocol provided by the class Array

```
at: index
        ↑ self at: index
                ifAbsent: [ self error: 'out of range array index']

at: index ifAbsent: exceptionBlock
        (self includesKey: index)
                ifTrue: [ ↑ < 24 self index > ]
                ifFalse: [ ↑ exceptionBlock value ]
```

To iterate over the elements in a collection the method to: is used to create an arithmetic progression representing the index value. The method do: cycles over these values, which are then used to index the array. The method reverseDo: is similar, only indexing backwards.

```
do: aBlock
        1 to: self size do: [ :i | aBlock value: (self at: i) ]
```

8.1.3 Addition and Modification Protocol

The method at:put: is used to modify the value of an array indicated by a given index value. Once the index is recognized as valid this operation, too, is implemented using a primitive.

```
at: index put: aValue
        (self includesKey: index)
                ifTrue: [ < 5 aValue self index > ]
                ifFalse: [ self error: 'array indexing error']
```

There are two methods used to increase the storage capacity of an array. The first, with: creates a new array which is one element larger than the current, adding the argument value to the last position. By using the expression self class new: to create the new structure, rather than Array new:, the same function can be used both for class Array and for ByteArray and String.

```
with: newItem | newArray size |
        size ← self size.
        newArray ← self class new: size + 1.
        1 to: size do: [ :i | newArray at: i put: (self at: i) ].
        newArray at: size + 1 put: newItem
        ↑ newArray
```

A more general technique is provided by overloading the + operator symbol. The argument is assumed to be a collection that is similar to the array. A new collection is created which is sufficient to hold both collections end to end, and the values are copied into the new structure. Once again, the idiom self class new: is used so that the code will work for strings as well as general arrays.

```
+ aValue | size1 size2 newValue |
        size1 ← self size.
        size2 ← aValue size.
        newValue ← self class new: (size1 + size2).
        1 to: size1 do: [ :i | newValue at: i put: (self at: i) ].
        1 to: size2 do: [ :i | newValue at: (s1 + i) put: (aValue at: i) ].
        ↑ newValue
```

8.1.4 Transformation Protocol

In a manner similar to the copy method for trees and lists, a copy of an array can be produced by simply using the message asArray.

```
copy
        ↑ self asArray
```

8.1.5 Testing Protocol

The method includes: is used to tell if a value is present in a collection. It does this simply by examining each value in turn, comparing it against the argument. (The method inherited from class collection: cannot be used due to the change in the meaning of at:ifAbsent:). The method includesKey: is used to determine if a value represents a legal index into an array. Note that the function size, used in the latter method, is also overridden in class Array in order to provide a more efficient implementation (the size is computed using a primitive operation, rather than by enumerating all the elements).

```
includes: aValue
        self do: [ :element | element = aValue ifTrue: [ ↑ true ] ].
        ↑ false

includesKey: index
        ↑ ((0 < index) and: [ index <= self size ]
```

Two arrays are considered to be equal if they have the same number of elements and the elements are themselves equals. Once again note that the same method is used for strings as well as general arrays.

```
= anArray
        self size = anArray size ifFalse: [ ↑ false ].
        1 to: self size do:
                [ :i | (self at: i) = (anArray at: i)
                        ifFalse: [ ↑ false ] ].
        ↑ true
```

The definition of < is slightly more complicated. The intuition to use is that of strings and dictionary-style ordering. Two values are compared up to their minimum size; if any position differs then the difference is used to determine the comparison. If the values match up to the minimum size, then the smaller array in size is smaller in comparison as well. (The latter occurs when comparing "lab" to "labrodore", for example).

```
< anArray | selfsize argsize |
        selfsize ← self size. argsize ← anArray size.
        1 to: (selfsize min: argsize)
                do: [ :i | (self at: i) = (arg at: i)
                        ifFalse: [ ↑ (self at: i) < (arg at: i) ] ].
        ↑ selfsize < argsize
```

8.1.6 The Class ByteArray

The class ByteArray is a subclass of Array in which the values are restricted to be integers in the range zero to 255 (that is, a byte). Byte arrays are used to maintain the compiled bytecodes executed by the Smalltalk virtual machine (Chapter ??). Byte arrays are also used as the implementation medium for strings.

The class ByteArray implements only three methods, inheriting all other behavior from class Array. All three methods employ primitive operations to do the actual work.

The class method named new: is used to allocate a new byte array with a given size.

```
new: size
        ↑ < 20 ByteArray size >
```

The remaining two functions are the methods at:ifAbsent: and at:put:, which do the actual access to byte values. Note that these differ from the methods in class Array only in the primitive operations they employ.

```
at: index ifAbsent: exceptionBlock
        (self includesKey: index)
                ifTrue: [ ↑ < 21 self index > ]
                ifFalse: [ ↑ exceptionBlock value ]

at: index put: aValue
        (self includesKey: index)
                ifTrue: [ < 22 aValue self index > ]
                ifFalse: [ self error: 'byte array indexing error' ]
```

8.2 Class String

Class String is used to represent sequenced collections of character values, for example literal strings. The class ByteArray is used for the actual storage, and many of the methods in class String are merely wrappers around similarly named methods in that class. Other methods, such as the access protocol provided by the method do:, are simply used as inherited from parent classes without change. Other methods provided by class String are related to the common use of strings as input and output sequences, and still others simplify common transformation operations on strings. The protocol for class String is shown in Figure 8.2.

8.2.1 Creation Protcol

Strings are most commonly created either by passing an existing object the message printString, or by the message new:, which creates a string with a given extent but empty values, or by reading in a line of text from a file. The latter is used, for example, in the read-eval-print interface to Little Smalltalk. It is implemented by a class method named input, which is defined in the following fashion:

```
input | c result |
        result ← ''.
        [ c ← Char input.
        c isNil ifTrue: [ ↑ nil ]. c ~= Char newline ]
                whileTrue: [ result ← result + c printString ].
        ↑ result
```

Characters are read from the input one by one (using the class method input defined in class Char). As long as neither the end of input is encounterd (indicated by a nil value being returned from the character input) nor an end of line, the characters are appending to the end of a result string. When a newline is encountered, the string that has been constructed up to that point (without the newline character) is returned.

8.2.2 Access Protocol

Recall that the message at:, inherited from class Array, is implemented using a more basic message at:ifAbsent:. The latter is, in turn, implemented by invoking the method inherited from the parent class ByteArray. The method in ByteArray will return a small integer value. This is then used to initialize a new character value. The return arrow in the ifAbsent: clause terminates the method prematurely in the event the index is invalid.

```
at: index ifAbsent: exceptionBlock
        Char new: (super at: index ifAbsent: [ ↑ exceptionBlock value ] )
```

The iteration methods do: and reverseDo: are inherited from class Array.

CREATION PROTOCOL

String input
: return line of text from standard input (class method)

String new: size
: yield new string with given size (class method)

ACCESS AND ITERATION PROTOCOL

at: index ifAbsent: exceptionBlock
: return value associated with key or exception value

ADDITION AND MODIFICATION PROTOCOL

at: index put: value
: modify value at given key, error if key is out of range

TRANSFORMATION PROTOCOL

asSymbol
: return symbol with same name as string

reverse
: return string in reverse order

collect: transformationBlock
: transform a string element by element, returning new string

select: testBlock
: select elements of string which satisfy condition

break: separators
: break string into words, using separators, returns a list

OTHER PROTOCOL

print
: print text on standard output area

copy
: make a clone of receiver string

doIt
: execute string as Smalltalk command

Figure 8.2: Protocol provided by the class String

8.2.3 Modification Protocol

New values cannot be added to string quantities, however existing positions within a string can be modified. This is accomplished by the method at:put:, which is again simply a wrapper for the method in class ByteArray. The argument is converted from a character value into a small integer (using the method value in class Char) and the integer value inserted into the byte array.

```
at: index put: aChar
        super at: index put: aChar value
```

8.2.4 Transformation Protocol

A number of techniques are used to transform a string into a new form. The message asSymbol converts the text of the string into a symbol value. To do this it simply invokes the class method from class Symbol.

```
asSymbol
        ↑ Symbol new: self
```

The message reverse was defined already in class List. A similar message can be constructed for class String by converting the string into a list, reversing, then converting back into a string.

```
reverse
        ↑ self asList reverse asString
```

We have seen the messages select: and collect: already in the abstract class Collection. These methods are overridden in the String class. When used with string values, they first construct the requested collection, then convert the result (which by default is a list), back into a string.

```
collect: transformationBlock
        ↑ (super collect: transformationBlock) asString

select: testBlock
        ↑ (super select: testBlock) asString
```

The method select: is often used, for example, to remove spaces or punchuation from a string, while the method collect: is used to transform a string character by character. To determine if a string represents a palendrome, for example, one might remove all spaces, convert all text into lower case, then reverse the string, and finally ask whether the results are the same.

```
        str ← 'A Man, A Plan, A Canal, Panama'.
        str ← str select: [ :c | c isAlphabetic ].
        str ← str collect: [ :c | c lowerCase ].
        str = str reverse
true
```

The method break: is used to break a string into a sequence of words. The argument represents a list of separator characters, usually spaces, tabs, and puncuation. The separators are removed, and the remaining words returned as a list of strings. This method is often used to break a line of input into parts. We have seen a use of this method already in the concordance program in the last chapter.

```
break: separators | words word |
        word ← ''.
        words ← List new.
        self do: [ :c |
                (separators includes: c)
                        ifTrue: [ (word size > 0) "found a word"
                                ifTrue: [ words addLast: word. word ← '' ] ]
                        ifFalse: [ word ← word + c printString ] ]
        "maybe there is a last word"
        (word size > 0) ifTrue: [ words addLast: word ].
        ↑ words
```

8.2.5 Other Protocol

The doIt method takes a string and executes it as a Smalltalk expression, returning the value of the expression. This method is at the heart of the read-eval-print loop interface for the Smalltalk system. (See Chapter ??). To accomplish this, the function first appends a unary message header and a return arrow to the expression, thereby forming a simple method body. It then invokes the method parser (see Chapter ??), translating the text of the expression into a compiled method. (Although the class Undefined is used to compile the method, it is not inserted into the protocol associated with the class). Assuming no errors were encountered during the parsing process, which is indicated by the fact that the result is non-nil, then a new context if formed and the method executed. The details of forming a context and executing methods will be discussed later in Chapter ??.

```
doIt | aMethod |
        aMethod ← Undefined parseMethod: 'doIt ↑' + self.
        ↑ meth notNil
                ifTrue: [ ↑ Context new
                        perform: aMethod withArguments: (Array new: 1) ]
```

8.3 Class Interval

The class Interval represents an arithmetic sequence. Instances of Interval are constructed by passing the message to: or to:by: to numbers. These messages construct and initialize the new interval:

```
to: limit
        ↑ Interval from: self to: limit by: 1

to: limit by: step
        ↑ Interval from: self to: limit by: step
```

Besides the class method used in the initialization of newly created intervals, there is only one other method implemented by class Interval, which is the iteration method do:.

```
do: | current |
        current ← low.
        (step < 0)
                ifTrue: [
                        [ current >= high ] whileTrue:
                                [ aBlock value: current.
                                current ← current + step ] ]
                ifFalse: [
                        [ current <= high ] whileTrue:
                                [ aBlock value: current.
                                current ← current + step ] ]
```

For reasons of efficiency, the common case involving integer limits and a step size of 1 is implemented directly in the class SmallInt, thereby avoiding the creation of an Interval value.

```
to: limit do: aBlock | i |
        i ← self.
        [ i <= limit ] whileTrue: [ aBlock value: i. i ← i + 1 ]
```

Exercises

1. Give the asymptotic complexity of each of the array methods shown in Figure 8.1.
2. Give the asymptotic complexity of each of the string methods shown in Figure 8.2.

Chapter 9

The Bytecode Compiler

The purpose of the bytecode compiler is to translate a textual description of a method into the internal bytecode description that can be executed by the Smalltalk virtual machine. All expressions evaluated by the Little Smalltalk system must pass through the bytecode compiler. For example, we saw in Chapter 8 that strings can be evaluated as expressions by passing them the message `doIt`, and in Chapter ?? we noted how this was used in the Real-Eval-Print loop interface to the Little Smalltalk system. The `doIt` message, in turn, simply places a method heading and a return operator around the expression, thereby transforming the expression into a simple method description. This method is then examined by the bytecode compiler, and finally the resulting bytecode representation is executed.

9.1 The Little Smalltalk Grammar

The Smalltalk language itself is exceedingly simple. Figure 9.1 gives a grammer for the Little Smalltalk dialect of Smalltalk-80, presented in the format called BNF (Backus-Naur form). Nonterminals are written in a *slanted font*, while terminal symbols are described either in **bold** or within quote marks. The symbol ϵ is used to represent "nothing", that is the absence of any symbols. Curly brace characters are used to represent items that can be repeated zero or more times.

The parsing approach used by the bytecode compiler is a method called "recursive descent parsing". In this technique nonterminals in the grammer are matched with functions designed to recognize valid instances of the nonterminal. Thus, the structure of the compiler mirrors almost exactly the structure of the grammer.

The parser itself is entirely represented as methods of a class named Parser. The purpose of the parser is not only to recognize valid instances of the grammer (reporting compiler error messages when errors are found), but also to produce an internal representation of the method. This internal representation is a tree-like structure, formed out of instances

method	::=	*methodHeader methodVariables body*
methodHeader	::=	**identifier**
	|	**binarySym identifier**
	|	**keyword identifier** { **keyword identifier** }
methodVariables	::=	ϵ
	|	"|" **identifier** "|"
body	::=	*statementList*
statementList	::=	*statement* { "." *statement* }
statement	::=	"↑" *expression*
	|	*expression*
expression	::=	**identifier** ← *expression*
	|	*term cascade*
term	::=	"(" *expression* ")"
	|	*block*
	|	"<" **integerConstant** { *term* } ">"
	|	**identifier**
	|	*literal*
literal	::=	**integerConstant**
	|	"−" **integerConstant**
	|	**characterConstant**
	|	**symbol**
	|	**string**
	|	*arrayLiteral*
arrayLiteral	::=	"#(" { *literal* } ")"
block	::=	"[" *blockArguments statementList* "]"
blockArguments	::=	ϵ
	|	**colonIdentifier** { **colonIdentifier** } "|"
cascade	::=	*keywordContinuation* { ";" *keywordContinuation* }
keywordContinuation	::=	*binaryContinuation* { **keyword** *term binaryContinuation* }
binaryContinuation	::=	*unaryContinuation* { **binarySym** *term unaryContinuation* }
unaryContinuation	::=	{ identifier }

Figure 9.1: Grammar for Little Smalltalk

of a class called `ParserNode` and its various subclasses. Once parsing is completed and the method has been converted into the parse tree representation, a second pass then walks over the parse tree and converts the various statements and expressions into the bytecode format.

The next sections will describe, in turn, the various phases of the compilation process. Section 9.2 will describe how the individual characters are converted into tokens which are meaningful to the parser. Section 9.3 will describe the actual parsing process itself, while section 9.4 will conclude with a discussion of the generation of the bytecode instructions.

9.2 Lexical Processing

The purpose of lexical processing is to convert a sequence of characters (that is, the input) into a stream of tokens, which are the fundamental units of parsing. Tokens can be roughly thought of as being anlogous to words in the English language. Tokens in Smalltalk represent names, constants, and syntactic constructs such as parenthesis or square brackets.

The lexical processing routines of the parser manipulate four instance variables held by the class `Parser`. The variable `text` contains the text of the method being analyzed. The variable `index` contains the starting position of the next token that will be analyzed. That is, all positions smaller than `index` have been processed, and `index` represents the next character position that will be considered when a new token is requested.

The value of the current character being considered is returned by the method named `currentChar`, defined as follows:

```
currentChar
    ↑ text at: index ifAbsent: [ nil ]
```

Advancing to the next character is performed by the method `nextChar`, which also returns the value of the character. The function `nextChar` returns a blank on end of input, whereas `currentChar` returns a `nil` value. This is because `nextChar` is frequently used within a loop, such as scanning the characters that compose an identifier. Returning a blank character simplifies the execution of such loops, since one must only test, for example, for a non-alphabetic character.

```
nextChar
    index <- index + 1.
    ↑ text at: index ifAbsent: [ $  ]
```

The parser instance variable `token` contains the actual token value generated by the last request for a new token. Tokens are always represented by a string. Thus, for example, integers are simply the string containing the digit values. The variable `tokenType` contains

the first character of the token, or blank when the end of input is reached. This is used by various routines to determine the type of the current token.

The fundamental routine for generating the lexical token is the method `nextlex`. In this routine, the method `skipBlanks` is first invoked to skip over blank spaces, tabs, newlines, and comments. The next non-blank character is then placed into the variable `tokenType`. If the end of input has not been encountered, the token is then processed either as an integer, a name, or a binary symbol, depending upon weather it begins with a digit, a letter, or some other character.

```
nextLex
    self skipBlanks.
    tokenType <- self currentChar.
    tokenType isNil   " end of input "
        ifTrue: [ tokenType <- $  . token <- nil. ↑ nil ].
    tokenType isDigit ifTrue: [ ↑ self lexInteger ].
    tokenType isAlphabetic ifTrue: [ ↑ self lexAlphabetic ].
    ↑ self lexBinary
```

A pair of mutually recursive routines are used to skip blank spaces and comments. The first, `skipBlanks`, removes blank characters. Note that the method `isBlank` in class `Char` returns true both on actual spaces, and on tab characters and newline characters. If the next non-blank character is the comment character, then the method `skipComment` is invoked as the final action.[1]

```
skipBlanks  | cc |
    [ cc <- self currentChar.
      cc notNil and: [ cc isBlank ] ]
            whileTrue: [ index <- index + 1 ].
    (cc notNil and: [ cc = $‘‘ ] )
        ifTrue: [ self skipComment ]
```

The `skipComment` function examines each character in turn until the end of comment character is found. This method is notable for the fact that almost all actions are performed in the conditional portion of the loop, while the body is empty. This includes a premature return operator, which will be executed should the end of input be discovered prior to the end of the comment. Once the end of comment is discovered, the routine `skipBlanks` is then executed once again to remove any blank characters following the comment.

```
skipComment  | cc |
```

[1] Note: the latex macro package I'm using doesn allow me to properly set either the sequences $" or $', so I'm using back quotes here and in a later method to represent these.

```
    [ index <- index + 1.
      cc <- self currentChar.
      cc isNil
        ifTrue: [ ↑ self error: 'unterminated comment'].
      cc ~= $'' ] whileTrue: [ nil ].
    self nextChar. self skipBlanks
```

We return now to the processing if input tokens, which was performed by the trio of routines `lexInteger`, `lexAlphabetic`, and `lexBinary`. The first of these, `lexInteger`, simply loops as long as the current character is an integer character. It is only necessary to record the start of the digit sequence. When the end is known, the string method `from:to:` is used to copy the integer string into the token buffer.

```
lexInteger    | start |
    start <- index.
    [ self nextChar isDigit ]
        whileTrue: [ nil ].
    token <- text from: start to: index - 1
```

In a similar manner, the method `lexAlphabetic` simply marks the beginning and ending positions of the next token, then once again uses the method `from:to:` to extract a substring from the input string.

```
lexAlphabetic | cc start |
    start <- index.
    [ (cc <- self nextChar) isAlphabetic ]
            whileTrue: [ nil ].
        " add any trailing colons "
    cc = $: ifTrue: [ self nextChar ].
    token <- text from: start to: index - 1
```

A binary symbol is one or two adjacent characters that are not syntactic characters, digits or letters. They are recognized by the following function:

```
lexBinary    | c d |
    c <- self currentChar.
    token <- c printString.
    d <- self nextChar.
    (self charIsSyntax: c) ifTrue: [ ↑ token ].
    (((d isBlank
        or: [ d isDigit])
        or: [ d isAlphabetic ])
```

```
        or: [ self charIsSyntax: d])
            ifTrue: [ ↑ token ].
    token <- token + d printString.
    self nextChar
```

The function `charIsSyntax` simply tests the argument against the list of syntactic characters:

```
charIsSyntax: c
    ↑ '.()[]''#↑$;' includes: c
```

9.3 The Parser

In the previous section we were introduced to four of the nine instance variables maintained by the class `Parser`. Three of the remaining five are arrays, which hold the names of arguments, instance variables, and temporary variables associated with the method under consideration. These values are initialized by the method `text:instanceVars:`, which must be executed on a new parser before processing can commence. Note that the argument array is initialized to size one, with the first argument being the value "self". All methods internally have this value implicitly as a first argument. The eighth instance variable is a number, representing the maximum number of temporary variables used by the method. It is initially zero.

```
text: aString instanceVars: anArray
    text <- aString.
    index <- 1.
    argNames <- Array new: 1.
    argNames at: 1 put: #self.
    instNames <- anArray.
    tempNames <- Array new: 0.
    maxTemps <- 0
```

The nineth and final instance variable maintained by the parser is an error return block. This is used to terminate the parsing process when a compiler error is detected. The error return block is created in the method `parse`, which is the main interface to the bytecode compiler. The `parse` method, if successful, returns an instance of `Method` containing the translated bytecode instructions.

```
parse    | encoder |
    " note -- must call text:instanceVars: first "
```

```
    errBlock <- [ ↑ nil ].
    self nextLex.
    encoder <- Encoder new.
    encoder name: self readMethodName.
    self readMethodVariables.
    self readBody compile: encoder block: false.
    ↑ encoder method: maxTemps text: text
```

The error block is used by the method `error:` which, as we have seen already, is invoked when a parsing or compilation error has been encountered. The method informs the programmer of the nature of the error, then executes the return block. Because the return block contains a return operator, it immediate transfers control back to the method which transmitted the `parse` message, returning a nil value.

```
error: aString
    'compiler error ' print.
    aString print.
    Char newline print.
    errBlock value
```

As noted in the grammar shown in Figure 9.1, a method consists of three parts. These are the method header, containing the method name and argument names, an optional list of temporary method variables and a method body. These three values are reflected in the three parsing calls made in the function `parse`, given earlier. The first of these, `readMethodName`, determines what type of method is being parsed, and returns the name of the method. As a side effect the argument names are placed into the names array. The method accomplishing this is as follows:

```
readMethodName   | name |
    self tokenIsName    " unary method "
        ifTrue: [ name <- token. self nextLex. ↑ name ].
    self tokenIsBinary    " binary method "
        ifTrue: [ name <- token. self nextLex.
            self tokenIsName
                ifFalse: [ self error: 'missing argument'].
                self addArgName: token asSymbol.
                self nextLex. ↑ name ].
    self tokenIsKeyword
        ifFalse: [ self error: 'invalid method header'].
    name <- ''.
    [ self tokenIsKeyword ]
        whileTrue: [ name <- name + token. self nextLex.
```

```
            self tokenIsName
                ifFalse: [ self error: 'missing argument'].
                self addArgName: token asSymbol.
                self nextLex ].
    ↑ name
```

A general property of the recursive descent parsing technique is that functions which recognize a nonterminal, such as the routine `readMethodName`, always call `nextLex` when they are successful. In this way upon a successful return the token buffer always contains the *next* token.

The routine `addArgName:` checks to see if a new argument has been used previous as an instance variable or a previous argument before inserting it into the name table.

```
addArgName: name
    ((instNames includes: name)
        or: [ argNames includes: name ])
        ifTrue: [ self error: 'doubly defined argument name '].
    argNames <- argNames with: name
```

The routines `tokenIsName` and `tokenIsKeyword` determine the type of the current token. The variable `tokenType` contains the first character, and can be used to determine whether the token is an identifier. A name must begin and end with identifier characters, whereas a keyword must end with a colon. A token is a binary symbol if it is not a name, a keyword, or a syntactic character.

```
tokenIsName
    tokenType isAlphabetic ifFalse: [ ↑ false ].
    ↑ (token at: token size) isAlphanumeric

tokenIsKeyword
    tokenType isAlphabetic ifFalse: [ ↑ false ].
    ↑ (token at: token size) = $:

tokenIsBinary
    (((token isNil
        or: [ self tokenIsName])
        or: [ self tokenIsKeyword])
        or: [ self charIsSyntax: tokenType ]) ifTrue: [ ↑ false ].
    ↑ true
```

The next major section of a method is the optional list of temporary method variables. This consists of a list of identifiers between a pair of vertical bars.

```
readMethodVariables
    tokenType = $| ifFalse: [ ↑ nil ].
    self nextLex.
    [ self tokenIsName ]
        whileTrue: [ self addTempName: token asSymbol. self nextLex ].
    tokenType = $|
        ifTrue: [ self nextLex ]
        ifFalse: [ self error: 'illegal method variable declaration']
```

Like the routine `addTempName:`, the routine `addTempName:` first checks to see if the new name has been used as an argument, instance variable, or another temporary variable before inserting it into the list of names. It also sets the counter which is keeping track of the maximum number of temporary variables used by the method.

```
addTempName: name
    (((argNames includes: name)
        or: [ instNames includes: name ] )
        or: [ tempNames includes: name ] )
        ifTrue: [ self error: 'doubly defined name '].
    tempNames <- tempNames with: name.
    maxTemps <- maxTemps max: tempNames size
```

The heart of a method is, of course, the list of statements that indicate the actions to be performed when the method is executed. This is known as the method *body*. We see in this method the first creation of one of the parser nodes which are used to represent the internal representation of a program. The routine `readStatementList` returns a list of statement nodes, which are inserted into a newly created instance of the class `BodyNode`.

```
readBody
    ↑ BodyNode new statements: self readStatementList
```

The recognition of statement lists is somewhat complicated by the fact that the statement terminating period is optional both after the final statement within a block and the final statement of a method.

```
readStatementList   | list |
    list <- List new.
    [ list add: self readStatement.
      tokenType notNil and: [ tokenType = $. ] ]
        whileTrue: [ self nextLex.
            (token isNil or: [ tokenType = $] ] )
```

```
                ifTrue: [ ↑ list ] ].
    ↑ list
```

A statement is either a return statement or an expression.

```
readStatement
    tokenType = $↑
        ifTrue: [ self nextLex.
            ↑ ReturnNode new expression: self readExpression ].
    ↑ self readExpression
```

The recognition of expressions is made more complex by the fact that both assignment statements and other expressions can begin with an identifier character. Thus, it is not until the following character is read that the type of expression can be determined.

The routine `readCascade:` takes as argument the term which is to the left of the cascade. (In a moment, when we encounter cascades in their other context, where they have no left term, this value will be zero). If the expression begins with an identifier, this term will always be simply the identifier node. Otherwise, the routine `readTerm` is invoked to parse the term value.

```
readExpression   | node |
    self tokenIsName ifFalse: [ ↑ self readCascade: self readTerm ].
    node <- self nameNode: token asSymbol. self nextLex.
    (token notNil and: [ token = '<-' ])
        ifTrue: [ node assignable
                ifFalse: [ self error: 'illegal assignment'].
            self nextLex.
            ↑ AssignNode new target: node expression: self readExpression ].
    ↑ self readCascade: node
```

A term can be a parenthesized expression, a block, a primitive, an identifier, or a literal.

```
readTerm   | node |
    token isNil
        ifTrue: [ self error: 'unexpected end of input' ].
    tokenType = $(
        ifTrue: [ self nextLex. node <- self readExpression.
            tokenType = $)
                ifFalse: [ self error: 'unbalanced parenthesis' ].
            self nextLex. ↑ node ].
    tokenType = $[ ifTrue: [ ↑ self readBlock ].
    tokenType = $< ifTrue: [ ↑ self readPrimitive ].
```

```
    self tokenIsName
        ifTrue: [ node <- self nameNode: token asSymbol.
            self nextLex. ↑ node ].
    ↑ LiteralNode new value: self readLiteral
```

Only nodes representing temporary names or instance variables are assignable. It is not legal to assign to argument names nor, in Little Smalltalk, to global variables. (Global variables can be modified by using the message `at:put:` on the global dictionary.) All valid names, however, can be read as expressions. Nodes representing identifiers as expression are constructed by the method `nameNode`. When a global identifier is used as an expression the value of the node at the time of compilation is used as a constant. The `nameNode:` routine must also check for the pseudo-variable "super" being used as a receiver. This variable is really simply a pseudonum for "self", and thus creates an instance of an `ArgumentNode`.

```
nameNode: name
    " make a new name node "
    name == #super
        ifTrue: [ ↑ ArgumentNode new position: 0 ].
    (1 to: tempNames size) do: [:i |
        (name == (tempNames at: i))
            ifTrue: [ ↑ TemporaryNode new position: i ] ].
    (1 to: argNames size) do: [:i |
        (name == (argNames at: i))
            ifTrue: [ ↑ ArgumentNode new position: i ] ].
    (1 to: instNames size) do: [:i |
        (name == (instNames at: i))
            ifTrue: [ ↑ InstNode new position: i ] ].
    ↑ LiteralNode new;
        value: (globals at: name
            ifAbsent: [ ↑ self error:
                'unrecognized name:' + name printString ])
```

Other literals include character constants, strings, integers, and symbols.

```
readLiteral   | node |
    tokenType = $$
        ifTrue: [ node <- self currentChar.
            self nextChar. self nextLex. ↑ node ].
    tokenType isDigit
        ifTrue: [ ↑ self readInteger ].
    token = '-'
        ifTrue: [ self nextLex. ↑ self readInteger negated ].
```

```
    tokenType = $‘
        ifTrue: [ ↑ self readString ].
    tokenType = $#
        ifTrue: [ ↑ self readSymbol ].
    self error: 'invalid literal'
```

It is at this point that an integer token is converted into the integer value. To convert an ASCII digit character into its corresponding numeric value the ASCII representation of zero (which is 48) is subtracted from the ASCII representation of the character. Note that an integer constant will automatically become a large integer (see Chapter 2) should the value become overly large.

```
readInteger  | value |
    tokenType isDigit ifFalse: [ self error: 'integer expected' ].
    value <- 0.
    token do: [:c | value <- value * 10 + (c value - 48) ].
    self nextLex.
    ↑ value
```

The processing of strings is similar to the processing of identifiers. The initial and final positions of the string are determined, then the message `from:to:` is used to copy the string from the source text. To place a single quote mark within a string constant two quote marks are used. This is processed by a recursive call to `readString`.

```
readString  | first last cc |
    first <- index.
    [ cc <- self currentChar.
      cc isNil ifTrue: [ self error: 'unterminated string constant'].
      cc ~= $‘ ] whileTrue: [ index <- index + 1 ].
    last <- index - 1.
    self nextChar = $‘
        ifTrue: [ self nextChar.
            ↑ (text from: first to: index - 2) + self readString ].
    self nextLex.
    ↑ text from: first to: last
```

The processing of symbols is complicated by the fact that the sharp sign is used both to indicate a simple symbol and a literal array. If an array is not specified, then the next lexeme is converted into a token value.

```
readSymbol   | cc |
    cc <- self currentChar.
```

```
    (cc isNil or: [ cc isBlank])
        ifTrue: [ self error: 'invalid symbol'].
    cc = $( ifTrue: [ ↑ self readArray ].
    (self charIsSyntax: cc)
        ifTrue: [ self error: 'invalid symbol'].
    self nextLex.
    cc <- Symbol new: token. self nextLex.
    ↑ cc
```

An array consists of a sequence of literals. Within an array, an identifier is treated as a symbol value.

```
readArray    | value |
    self nextChar. self nextLex. value <- Array new: 0.
    [ tokenType ~= $) ]
        whileTrue: [ value <- value with: self arrayLiteral ].
    self nextLex.
    ↑ value

arrayLiteral  | node |
    tokenType isAlphabetic
        ifTrue: [ node <- Symbol new: token. self nextLex. ↑ node ].
    ↑ self readLiteral
```

We return now to the processing of primitive expressions. A primitive consists of an integer representing the primitive number, followed by a series of zero or more terms which represent the primitive arguments.

```
readPrimitive  | num args |
    self nextLex.
    num <- self readInteger.
    args <- List new.
    [ tokenType ~= $> ]
        whileTrue: [ args add: self readTerm ].
    self nextLex.
    ↑ PrimitiveNode new number: num arguments: args
```

The final type of term to be described is the block. A block consists of an optional list of block arguments, followed by a list of statements. Since the temporary names used within the block are valid only within the block scope, the array of temporary names is saved prior to the beginning of processing for the block, and restored following this block. A block must

remember the starting index of the first block argument in the temporary array. But this is easily computed, as it is the size of the temporary array prior to the block being processed.

```
readBlock    | stmts saveTemps |
    saveTemps <- tempNames.
    self nextLex.
    tokenType = $:
        ifTrue: [ self readBlockTemporaries ].
    stmts <- self readStatementList.
    tempNames <- saveTemps.
    tokenType = $]
        ifTrue: [ self nextLex.
            ↑ BlockNode new statements: stmts
                temporaryLocation: saveTemps size ]
        ifFalse: [ self error: 'unterminated block']

readBlockTemporaries
    [ tokenType = $: ]
        whileTrue: [ self currentChar isAlphabetic
            ifFalse: [ self error: 'ill formed block argument'].
            self nextLex.
            self tokenIsName
                ifTrue: [ self addTempName: token asSymbol ]
                ifFalse: [ self error: 'invalid block argument list '].
            self nextLex ].
    tokenType = $|
        ifTrue: [ self nextLex ]
        ifFalse: [ self error: 'invalid block argument list ']
```

Once a receiver has been identified, the routine `readExpression` (page 88) goes on to parse a sequence of messages. These are processed by the method `readCascade:`. A cascade consists of a series of one or more messages being sent to the same receiver, seperated by colons. A message, in turn, can be a keyword message, a sequence of binary messages, or a sequence of unary messages. These are each recognized by a specialized routine.

```
readCascade: base   | node list |
    node <- self keywordContinuation: base.
    tokenType = $;
        ifTrue: [ node <- CascadeNode new head: node.
            list <- List new.
            [ tokenType = $; ]
                whileTrue: [ self nextLex.
```

```
                    list add:
                        (self keywordContinuation: nil ) ].
            node list: list ].
    ↑ node

keywordContinuation: base  | receiver name args |
    receiver <- self binaryContinuation: base.
    self tokenIsKeyword
        ifFalse: [ ↑ receiver ].
    name <- ''.
    args <- List new.
    [ self tokenIsKeyword ]
        whileTrue: [ name <- name + token. self nextLex.
            args add:
                (self binaryContinuation: self readTerm) ].
    ↑ MessageNode new receiver: receiver name: name asSymbol arguments: args

binaryContinuation: base | receiver name |
    receiver <- self unaryContinuation: base.
    [ self tokenIsBinary]
        whileTrue: [ name <- token asSymbol. self nextLex.
            receiver <- MessageNode new
                receiver: receiver name: name arguments:
                    (List with:
                        (self unaryContinuation: self readTerm)) ].
    ↑ receiver

unaryContinuation: base | receiver |
    receiver <- base.
    [ self tokenIsName ]
        whileTrue: [ receiver <- MessageNode new
                receiver: receiver name: token asSymbol
                    arguments: (List new).
                self nextLex ].
    ↑ receiver
```

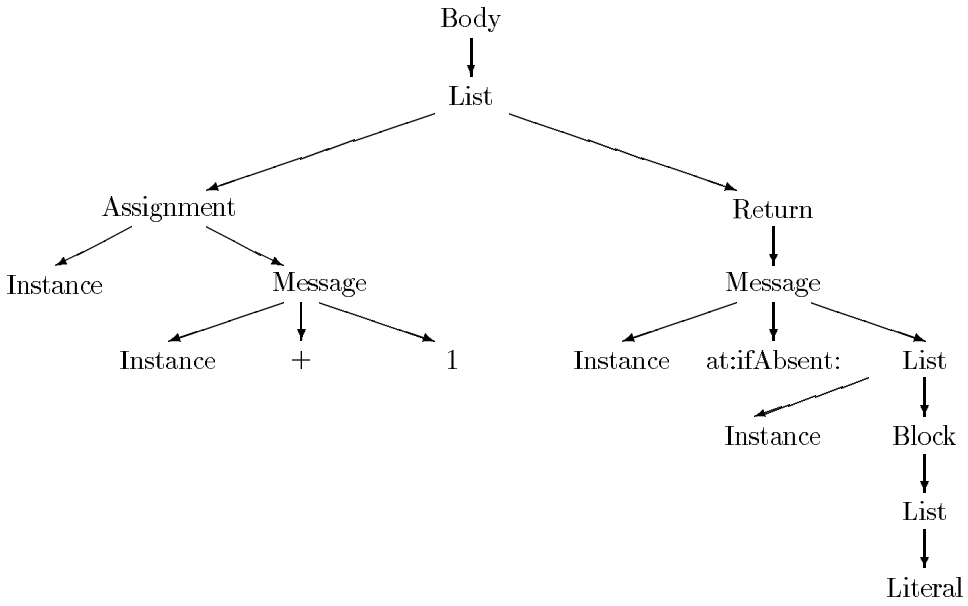
If parsing is successful the result is the program represented by a tree of parser nodes. The next step is then to translate this into the bytecode format.

9.4 Bytecode Generation

If parsing is successful then a method will have been sucessfully transformed into a parse tree representation. The method `nextChar`, for example, which has the following textual description:

```
nextChar
    index <- index + 1.
    ↑ text at: index ifAbsent: [ $  ]
```

is represented by the following tree:



The next step is to convert the parse tree into bytecode format. Bytecodes can be thought of as machine instructions for an imaginary "smalltalk machine". The bytecodes used by the Little Smalltalk system are shown in Figure 9.2. Notice there are 16 operations.[2] These are encoded in the high-order four bits of a byte value. The low order 4 bits are used as an argument value. In the case that the argument value is larger than 16 an extended form is used (opcode 0), which permits the argument value to be as large as 255. No arguments are allowed to be larger than this value.

During the compilation process the actual bytecodes are held by the encoder. We will defer for the moment a discussion of the implementation of the encoder. Three encoder

[2] Actually, only 15 operation codes are used. Opcode 14 is unassigned.

Number	Name	Operation
1	Push Instance	push instance variable on to stack
2	Push Argument	push argument value on to stack
3	Push Temporary	push temporary variable on to stack
4	Push Literal	push literal constant on to stack
5	Push Constant	push a special constant on to stack
6	Assign Instance	assign top of stack to instance var
7	Assign Temporary	assign top of stack to temporary var
8	Mark Arguments	mark top of stack as arguments
9	Send Message	send a message using arguments on stack
10	Send Unary	send an optimized unary message
11	Send Binary	send an optimized binary message
12	Push Block	push a block value on to stack
13	Do Primitive	execute a primitive instruction
15	Do Special	do a variety of special operations
0	Extended	execute an extended operation

Figure 9.2: The Little Smalltalk Bytecode Instruction

methods are used during the generation of bytecodes. These are `genHigh:low:`, which takes and argument the high and low order portions of a bytecode instruction, `genCode:`, which takes as argument a full byte value, and `genLiteral:`, which adds a literal value to the literal table and returns the offset in that table.

Transformation of parse tree nodes into bytecodes is accomplished by sending the message `compile:block:` to each parse tree node. The first argument is the encoder value, while the second is a boolean that is true if the expressions being compiled are statements from inside a block. We therefore will proceed to describe how each of the eleven different forms of parse tree nodes respond to this message.

The various forms of identifiers simply generate a simple instruction. The offset of the temporary variable in the run-time temporary array has already been computed. The following, for example, is the code for temporary nodes. The code for instance variables is similar.

```
compile: encoder block: inBlock
    encoder genHigh: 3 low: position - 1
```

The code for argument nodes is only slightly more complex. The offset zero is used to indicate the pseudo-variable "super", although the actual offset is the same as the for the pseudo-variable "self".

```
compile: encoder block: inBlock
    position = 0
        ifTrue: [ encoder genHigh: 2 low: 0 ]
        ifFalse: [ encoder genHigh: 2 low: position - 1 ]
```

Assignment nodes generate code for the expression portion, then pass the `assign` message to the receiver to generate the actual assignment operation. Both temporary nodes and instance nodes (the only two types of values that can be assigned) generate instructions for this message.

```
compile: encoder block: inBlock
    expression compile: encoder block: inBlock.
    target assign: encoder
```

Primitive nodes generate code for each argument, then produce a primitive instruction. The low order bits of the primitive instruction are the number of arguments, while the next byte is the primitive number.

```
compile: encoder block: inBlock
    arguments reverseDo: [ :a | a compile: encoder block: inBlock ].
    encoder genHigh: 13 low: arguments size.
    encoder genCode: number
```

The constants true, false, nil and the integers less than 10 occur with such frequency that they are recognized as special instructions. Otherwise a literal node simply uses the encoder to place a value into the literal pool and generates an instruction to push the given value.

```
compile: encoder block: inBlock
    value == nil ifTrue: [ ↑ encoder genHigh: 5 low: 10 ].
    value == true ifTrue: [ ↑ encoder genHigh: 5 low: 11 ].
    value == false ifTrue: [ ↑ encoder genHigh: 5 low: 12 ].
    (value class == SmallInt and: [ value < 10 ])
        ifTrue: [ ↑ encoder genHigh: 5 low: value ].
    encoder genHigh: 4 low: (encoder genLiteral: value)
```

A block node implements two different compile methods. As we will see in a moment, often blocks will be expanded in-line, thereby avoiding the overhead of block creation. In these cases the code simply executes the sequence of statements, popping the value remaining on the stack after each statement. The final pop is "erased" after it is generated by the method `backUp`, leaving the value of the last statement on the top of stack.

```
compileInLine: encoder block: inBlock
    statements reverseDo:
        [ :stmt | stmt compile: encoder block: inBlock.
            encoder genHigh: 15 low: 5 " pop top " ].
    encoder backUp
```

When not optimized, a block node generates a push block instruction. This instruction takes two arguments. The first argument is the starting location for block arguments in run-time temporary array. The second argument is a bytecode offset, indicating the location after the block code. Following the push block instruction the code for the statements within the block are placed in-line. A final "return top of stack" instruction completes the block code. Since the offset of the code which will follow the block code is not known until after it is generated, the encoder goes back and "patches" the push block instruction with the proper offset.

```
compile: encoder block: inBlock | patchLocation |
    encoder genHigh: 12 low: temporaryLocation.
    patchLocation <- encoder genCode: 0.
    self compileInLine: encoder block: true.
    encoder genHigh: 15 low: 2. " return top of stack "
    encoder patch: patchLocation
```

A return statement generates a return operation if it is found outside of a block, and a block return operation if inside a block. Both are implemented as special instructions.

```
compile: encoder block: inBlock
    expression compile: encoder block: inBlock.
    inBlock
        ifTrue: [ encoder genHigh: 15 low: 3 " block return " ]
        ifFalse: [ encoder genHigh: 15 low: 2 " stack return " ]
```

A body node holds a list of statements. Code is generated for each statement, followed by a special instruction to pop the resulting value from the stack. Finally a "return self" instruction is generated, which will be executed if the body does not include an explicit return statement.

```
compile: encoder block: inBlock
    statements reverseDo:
        [ :stmt | stmt compile: encoder block: inBlock.
            encoder genHigh: 15 low: 5 " pop "].
    encoder genHigh: 15 low: 1 " return self "
```

A cascade is similar. The code for the receiver is generated. For each cascaded expression an instruction is generated prior to the expression to duplicate the receiver (on the top of stack). The code for the cascaded expression is then produced. Finally an instruction is generated to pop the resulting value from the stack.

```
compile: encoder block: inBlock
    head compile: encoder block: inBlock.
    list reverseDo: [ :stmt |
        encoder genHigh: 15 low: 4. " duplicate "
        stmt compile: encoder block: inBlock.
        encoder genHigh: 15 low: 5 "pop from stack " ]
```

Compiling code for the message send node is made complicated by the large number of special cases that must be handled:

- Cascades, which are messages without any receiver
- Messages sent to "super"
- The messages `isNil` and `notNil`, which are handled as special unary instructions (opcode 10).
- The messages +, < and <=, which are handled as special binary instructions (opcode 11).
- The messages `ifTrue:`, `ifFalse:`, `ifTrue:ifFalse:`, `and:`, `or:`, and `whileTrue:`, which are expanded in-line into contol flow instructions rather than being processed as messages.

The following top-level routine shows how all of these cases are merged into a single framework.

```
compile: encoder block: inBlock
    receiver isNil
        ifTrue: [ ↑ self cascade: encoder block: inBlock ].
    ((receiver isBlock and: [ self argumentsAreBlock ])
        and: [name = #whileTrue: or: [ name = #whileFalse ] ] )
        ifTrue: [ ↑ self optimizeWhile: encoder block: inBlock ].
    receiver compile: encoder block: inBlock.
    receiver isSuper
        ifTrue: [ ↑ self cascade: encoder block: inBlock ].
    name = #isNil ifTrue: [ ↑ encoder genHigh: 10 low: 0 ].
    name = #notNil ifTrue: [ ↑ encoder genHigh: 10 low: 1 ].
```

```
    self argumentsAreBlock ifTrue: [
        name = #ifTrue: ifTrue: [ ↑ self compile: encoder
                test: 8 constant: 10 block: inBlock ].
        name = #ifFalse: ifTrue: [ ↑ self compile: encoder
                test: 7 constant: 10 block: inBlock ].
        name = #and: ifTrue: [ ↑ self compile: encoder
                test: 8 constant: 12 block: inBlock ].
        name = #or: ifTrue: [ ↑ self compile: encoder
                test: 7 constant: 11 block: inBlock ].
        name = #ifTrue:ifFalse:
            ifTrue: [ ↑ self optimizeIf: encoder block: inBlock ].
        ].
    self evaluateArguments: encoder block: inBlock.
    name = #< ifTrue: [ ↑ encoder genHigh: 11 low: 0].
    name = #<= ifTrue: [ ↑ encoder genHigh: 11 low: 1].
    name = #+ ifTrue: [ ↑ encoder genHigh: 11 low: 2].
    self sendMessage: encoder block: inBlock
```

The message `cascade:block:` is simply a wrapper around a pair of methods which will evaluate the arguments and send a message. These two methods are also used by the compilation routine just described. The method `evaluateArguments` sends the message `pushArgs` to the encoder, which is keeping track of the maximum size the stack will need to be in order to accomodate the method. The corresponding method `popArgs` is invoked in the method `sendMessage`.

```
cascade: encoder block: inBlock
    self evaluateArguments: encoder block: inBlock.
    self sendMessage: encoder block: inBlock

evaluateArguments: encoder block: inBlock
    encoder pushArgs: 1 + arguments size.
    arguments reverseDo: [ :arg |
        arg compile: encoder block: inBlock ].

sendMessage: encoder block: inBlock
        " mark arguments, then send message "
    encoder genHigh: 8 low: arguments size.
    encoder genHigh: 9 low: (encoder genLiteral: name).
    encoder popArgs: arguments size.

argumentsAreBlock
    arguments do: [ :arg | arg isBlock ifFalse: [ ↑ false ]].
```

↑ `true`

Messages that implement control flow, such as while statements and if statements, can be made greatly more efficient by subsituting control flow bytecode instructions for message sending. The following pseudo-code, for example, describes an operational definition of a while statement. Note that the statement must still *look* like a message, and thus a final `nil` value is left sitting on the stack.

labelOne:
evaluate receiver in-line
branch if false to labelTwo
evaluate argument in-line
pop result from stack
branch back to labelOne
labelTwo: *push nil on stack*

In the case of a `whileFalse:` loop the branch if false instruction is changed to a branch if true. The function that emits this code is as follows. The `patch:` routine is the same one that was used in the code generation for the block node.

```
optimizeWhile: encoder block: inBlock | start save |
    start <- encoder currentLocation.
    receiver compileInLine: encoder block: inBlock.
    name = #whileTrue:    " branch if false/true "
        ifTrue: [ encoder genHigh: 15 low: 8 ]
        ifFalse: [ encoder genHigh: 15 low: 7 ].
    save <- encoder genCode: 0.
    arguments first compileInLine: encoder block: inBlock.
    encoder genHigh: 15 low: 5. " pop from stack "
    encoder genHigh: 15 low: 6. " branch "
    encoder genCode: start. " branch target "
    encoder patch: save.
    encoder genHigh: 5 low: 10  " push nil "
```

By changing the values of the test and branch instruction and the final constant pushed on the stack, the `ifTrue:`, `ifFalse:`, `and:` and `or:` instructions can all be described by the following pattern:

branch if test to labelOne
evaluate argument statements
branch to labelTwo

labelOne: *push constant on stack*
labelTwo:

The procedure to emit this code is the following. Note that both the branch instruction opcode and the final constant value are passed as argument to the procedure.

```
compile: encoder test: t constant: c block: inBlock | save ssave |
    encoder genHigh: 15 low: t.  " branch test "
    save <- encoder genCode: 0.
    arguments first compileInLine: encoder block: inBlock.
    encoder genHigh: 15 low: 6.  " branch "
    ssave <- encoder genCode: 0.
    encoder patch: save.
    encoder genHigh: 5 low: c.  " push constant "
    encoder patch: ssave
```

The full `ifTrue:ifFalse:` method simply replaces the evaluation of the constant with the body of the false block.

```
optimizeIf: encoder block: inBlock | save ssave |
    encoder genHigh: 15 low: 7.  " branch if true test "
    save <- encoder genCode: 0.
    arguments first compileInLine: encoder block: inBlock.
    arguments removeFirst.
    encoder genHigh: 15 low: 6.  " branch "
    ssave <- encoder genCode: 0.
    encoder patch: save.
    arguments first compileInLine: encoder block: inBlock.
    encoder patch: ssave
```

Chapter 10

Classes and Metaclasses

Part III

Example Programs

Chapter 11

The Eight Queens Puzzle

Chapter 12

A Simple Simulation

Part IV

The Smalltalk Virtual Machine

Chapter 13

Overview

Chapter 14

Memory Management

Chapter 15

The Bytecode Interpreter

Part V

Appendices

Appendix A

Differences between Little Smalltalk and Smalltalk-80

A.1 Cascades

A.2 Primitive Operations

A.3 The User Interface

A.4 Symbols

In Little Smalltalk symbols are subclasses of class `Magnatude`, and understand the ordering messages $<$ and $=$. This fact is used by the creation routine for symbols, which maintains all symbols in a large binary tree. In Smalltalk-80 symbols only understand the identity testing message $==$.

A.5 The Collection Hierarchy

The collection hierarchy in Little Smalltalk is considerably smaller than the Smalltalk-80 system. In addition, many of the classes that provide similar features have slightly different names (`List` for `LinkedList`, for example).

- The list class in Little Smalltalk is called `List`, not `LinkedList`, as in Smalltalk-80.
- The abstract classes `LookupKey`, `SequenceableCollection`, `ArrayedCollection`, and `OrderedCollection` are not provided.

- The data structures `RunArray`, `Text`, `SortedCollection`, `Bag`, `MappedCollection`, `Set` and `IdentityDictionary` are not provided.
- Ordered classes in Little Smalltalk are represented by the class `Tree`, which has no counterpart in Smalltalk-80. The Smalltalk-80 class `OrderedCollection` is, as we have already noted, not supported.
- The Little Smalltalk system uses trees to implement the class `Dictionary`, whereas the Smalltalk-80 system uses hash tables. The advantage of trees is that they are easier to implement. The disadvantage is that they require that all key values must be orderable. The disadvantage of hash tables, on the other hand, is that they require all elements know how to compute a hash value. (This hash value must remain constant even when garbage collection occurs. This rules out the use of, for example, machine addresses as hash values; as garbage collection will force objects into new locations in memory.)

A.6 Strings

- The + is used to as the catenation operator for strings in Little Smalltalk, in Smalltalk-80 the comma is the catenation operator.