The Emerald Programming Language[1]

REPORT

Norman C. Hutchinson, Rajendra K. Raj,
Andrew P. Black, Henry M. Levy, and Eric Jul[2]

Department of Computer Science
University of Washington
Seattle, WA 98195

Technical Report 87-10-07
October 1987
(Revised September 1997)

Previously also released as:

DIKU Report No. 87/22, Department of Computer Science, University of Copenhagen, Denmark.
T.R. 87/29, Department of Computer Science, University of Arizona, Tucson, AZ 85721.

[1]This work was supported in part by the National Science Foundation under Grants No. MCS-8004111, DCR-8420945, and CCR-8700106, by Københavns Universitet (the University of Copenhagen), Denmark under Grant J.nr. 574-2,2, by Digital Equipment Corporation External Research Grants, by an IBM Graduate Fellowship, and a dissertation award from Microsoft Corporation.

[2]Authors' current addresses: Norman Hutchinson, Department of Computer Science, University of Arizona, Tucson, AZ 85721. Andrew Black, Digital Equipment Corporation, 550 King Street, Littleton, MA 01460. Eric Jul, DIKU, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark.

Abstract:

The programming language Emerald was designed and developed to demonstrate that the object-based style of programming can be incorporated both elegantly and efficiently in the distributed programming environment. At the same time, Emerald is an '80s programming language providing excellent features for abstraction and polymorphism. Primarily a language for distributed environments, Emerald includes features for dealing with the location of objects, and extends exception handling mechanisms for recovering from partial failures of distributed systems. This report presents an overview of Emerald and a concise description of its language constructs.

Contents

1 Introduction

The justification for Emerald, as for any new programming language, is that existing languages have proved to be unsatisfactory for the applications at hand. The primary goal of Emerald [Black 86, Black 87, Jul 88b] is to simplify distributed programming through language support while providing acceptable performance and flexibility both in local and distributed environments. Emerald also demonstrates that the object-based model of programming can be incorporated both elegantly and efficiently in distributed systems.

For pragmatic reasons, it is often advantageous to base new languages on the better features of existing languages. For efficiency and other considerations, language design for distributed systems is strongly influenced by the underlying distributed operating system. Consequently, Emerald draws heavily upon the experience gained from Smalltalk [Goldberg 83], the Argus Language and System [Liskov 84] and, in particular, the Eden system [Almes 85, Black 85a] and the Eden Programming Language (EPL) [Black 85b].

Featuring an object-oriented style of programming, Emerald presents a unified semantic view of objects appropriate for private, local, data-only objects as well as shared, remote, concurrently-executing objects. The nature of objects in Emerald is similar to that in Smalltalk [Goldberg 83], i.e., all data items are objects with a uniform semantic model for operations on them, but Emerald does not have any notion of *class*. Emerald was explicitly designed to support data abstraction: all typing of objects is at an abstract level and does not depend on the implementation chosen. Abstract typing aids in the dynamic construction of distributed programs by allowing any object in a large, possibly distributed, program to be replaced by any other type-consistent object. Type consistency or *conformity* is an important aspect of Emerald, and is discussed below. Another advantage of treating types as first class objects is that it makes polymorphism inherent in Emerald.

Recognizing *location* as an important attribute of an object in distributed programs, Emerald gives the programmer access to the location of objects through primitives that permit the inspection and selection of location. Alternatively, when desired, the location details may be left to the reasonably-chosen system-defaults. However, this recognition of the importance of location for distributed programming has its drawbacks, viz., the semantics of Emerald are complicated both because location is apparent and because systems may be partially unavailable.

The semantics of a programming language are difficult to describe precisely without resorting to formal semantics; perhaps a future report may describe the semantics of Emerald formally. In this report we first present an overview of the language features, and then describe the various language constructs informally. People who plan to write and execute Emerald programs will find the example in Appendix C useful; they should also keep the Emerald System User's Guide [Jul 88c] for more details about using the Emerald compiler and kernel. The Emerald approach to programming is discussed in [Raj 88], where several examples of Emerald programs may be found.

The rest of this introductory section contains an overview of the Emerald language, adapted in part from Norman Hutchinson's Ph.D. dissertation [Hutchinson 87b], and other Emerald Project reports.

1.1 Emerald Objects

Emerald extends the utility of a single object model to distributed systems. The object is used as the sole abstraction mechanism for the notions of data, procedure, and process. All entities in Emerald are objects spanning the spectrum from small entities (e.g., Booleans and integers) to large entities (e.g., directories, compilers, and entire file systems). An object exists as long as it can be named, or equivalently, can be referenced to by an identifier.

Each Emerald object consists of:

- A *name*, which uniquely identifies the object within the network.
- A *representation*, which, except in the case of a primitive object, consists of references to other objects.
- A set of *operations*, which define the functions and procedures that the object can execute. Exported operations may be invoked by other objects; other operations are private to the object.
- An optional *process*, which is started after the object is initialized, and executes in parallel with invocations of the object's operations. An object without a process is passive and executes only as a result of invocations while an object with a process has an active existence and executes independently of other objects.

Each object has two other attributes. An object has a *location* that specifies the node on which that object is currently located. Emerald objects may be defined to be *immutable*, i.e. these objects do not change over time. Immutability helps by simplifying sharing in a distributed system by permitting such objects to be freely copied from node to node. Immutability is a logical assertion on the part of the programmer rather than a physical property; the system does not attempt to check it.

Concurrency exists both between objects and within an object. Within the network many objects can execute concurrently. Within a single object, several operation invocations can be in progress simultaneously, and these execute in parallel with the object's internal process. To control access to variables shared by different operations, the shared variables and the operations manipulating them can be defined within a monitor. Processes synchronize through system-defined *condition* objects. An object's process executes outside of the monitor, but can invoke monitored operations when it needs access to the shared state.

Each object has an optional *initially* section; this is a parameterless operation that executes exactly once when the object is created, and is used to initialize the object state.

When the initially operation is done, the object's process is started, and it is ready to accept invocations.

1.2 Invocation

The only mechanism for communication in Emerald is the *invocation.* An Emerald object may invoke some operation defined in another object, passing arguments to the invocation and receiving results. Assuming that *target* is an object reference, the phrase:

target.operationName[*argument1, argument2*]

means execute the operation named *operationName* on the object currently referenced by *target*, passing *argument1* and *argument2* as arguments. Invocations are synchronous; the process performing the invocation is suspended until the operation is completed or until the run-time system determines that the operation cannot be completed. All arguments and results of invocations are passed by object reference, i.e., the invoker and invokee share references to the argument.

1.3 Abstract types

Central to Emerald are the concepts of *abstract type* and *type conformity.* Since all types in Emerald, by definition, are abstract types, this report will generally omit the adjective *abstract* and simply use the word *type*[1]. All identifiers in Emerald are typed, and the programmer must declare the type of the objects that an identifier may name. A type defines a collection of *operation signatures*, i.e., operation names and the types of their arguments and results. A type is represented by an Emerald object that specifies such a list of signatures. For example, if the variable *MyMailbox* is declared as:

var *MyMailbox : AbstractMailbox*

then any object that is assigned to MyMailbox must implement (at least) the operations defined by AbstractMailbox.

The type of the object being assigned must *conform* to the type of the identifier. Conformity is the basis of type checking in Emerald. Informally, a type S conforms to a type T (written $S \mathrel{\circ\!\!>} T$) if:

1. S provides at least the operations of T (S may have more operations).
2. For each operation in T, the corresponding operation in S has the same number of arguments and results.

[1]We may later make references to *concrete* types; these may informally be regarded as machine representations of object implementations and are not of primary concern to the Emerald programmer. To reduce confusion, we may occasionally qualify types as being abstract.

3. The types of the results of S's operations conform to the types of the results of T's operations.

4. The types of the arguments of T's operations conform to the types of the arguments of S's operations (i.e., arguments must conform in the opposite direction).

It requires little effort to establish that the conformity relation between types is both reflexive and transitive. Additionally, note that conformity is anti-symmetric: A $\circ\!>$ B does not imply that B $\circ\!>$ A; in fact, if A $\circ\!>$ B and B $\circ\!>$ A, then A and B are identical types. Types therefore form a partial order with conformity as the ordering relation.

The relationship between types and object implementations is many-to-one in both directions. A single object may conform to many types, and a type may be implemented by many different objects. Figure 1 illustrates these relationships. In the figure, A above B means A $\circ\!>$ B.

The object *DiskFile* implements the type *InputOutputFile*, the types *InputFile* and *OutputFile* (which require only a subset of the *InputOutputFile* operations), and also the type **Any** (which requires no operations at all). The type *InputOutputFile* illustrates that a type may have several implementations, perhaps tuned to different usage patterns. Temporary files may be implemented in primary memory (using *InCoreFile* objects) to provide fast access while giving up permanence in the face of crashes. On the other hand, permanent files implemented using *DiskFile* would continue to exist across crashes.

Since Emerald objects may conform to more than one type, it may be appropriate to change one's *view* of a particular object at run-time. This change may either be a *widening*, i.e., the number of operations viewed as being supported by the object is increased, or a *narrowing*, i.e., the number of available operations is reduced. Widening corresponds to a move up in the type partial order and narrowing to a move down the type hierarchy. Narrowing requires no run-time check of its validity, since any object conforming to some type in the partial order also conforms to all types that it is greater than (with respect to $\circ\!>$) . Widening on the other hand requires that the system check that the given object in fact does support the operations required by the new type. To prevent type misuse and to enhance security, Emerald also provides a *restrict* capability that prevents unrestrained widening of an object's type (see also Section 4.7).

An example of where such view changes are required is in the implementation of a directory system. For example, consider the type *Directory* defined as follows:

```
const Directory == type Directory
    operation Add[name : String, thing : Any]
    operation Lookup[name : String] → [thing : Any]
    operation Delete[name : String]
end Directory
```

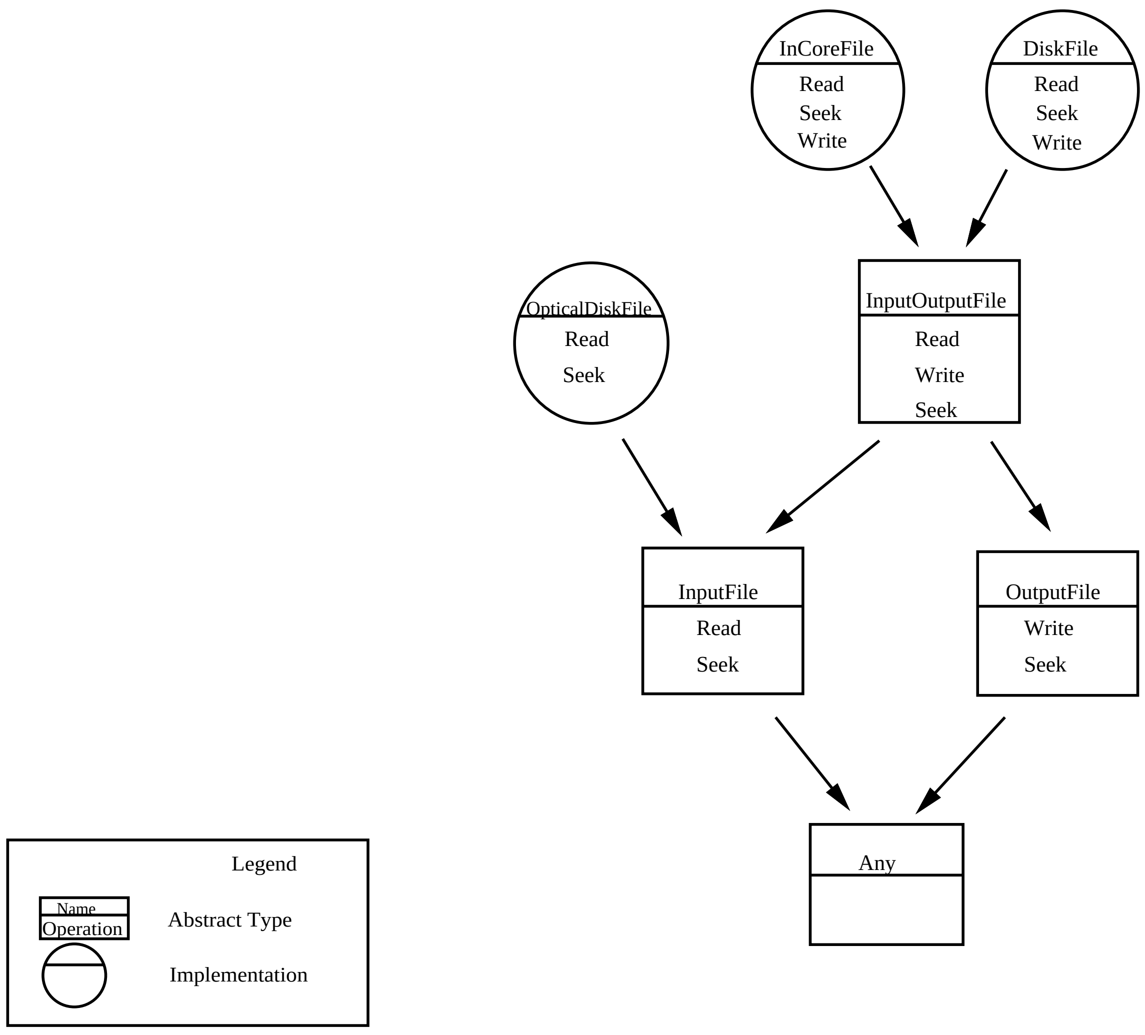

Figure 1: Example types and object implementations

with additional variables declared as

var *f : InputOutputFile*
var *g : OutputFile*

and f is currently naming a file object. If this file, f, is to inserted into a directory d, the invocation:

d.Add[*"myfile"*, *f*]

may be used. Since the second argument to *Add* on directories has type **Any**, its type *InputOutputFile* is narrowed to **Any**. When the same object is retrieved from the directory d, the assignment:

$f \leftarrow$ *d.Lookup*[*"myfile"*]

will fail because the type of the result of *Lookup* is **Any**, and **Any** does not conform to *InputOutputFile*, the type of *f*. Therefore, the preceding statement is not type-correct and is rejected by the compiler. On the other hand, since it is known that the object returned by executing *Lookup* on *d* with the argument *"myfile"* is in fact an *InputOutputFile*, an explicit change of view:

$f \leftarrow$ **view** *d.Lookup*[*"myfile"*] **as** *InputOutputFile*

may be used. This widening can be established as correct only at run-time. To place a limit on the widening of an object, an explicit restriction:

$g \leftarrow$ **restrict** *f* **to** *OutputFile*

may be placed. Note that the restriction is on the reference *g*, and not on the actual object.

1.4 Object creation

In most object-based systems, new objects are created by an operation on a *class* (as in Smalltalk) or a *type* object (as in Hydra). This class object defines the structure and behavior of all its *instances*. In addition, the class object responds to *new* invocations to create new instances.

In contrast, an Emerald object is created by executing an *object constructor* (cf. Section 8.1). An object constructor is an Emerald expression that defines the representation, the operations, and the process of an object. For example, suppose the Emerald program in Figure 2 is executed; it results in the creation of a single object. If we wished to create more *oneEntryDirectories* we would embed the object constructor of Figure 2 in a context in which it could be repeatedly executed, such as the body of a loop or operation. This is illustrated in Figure 3. Execution of this example creates the single object specified by the outermost object constructor. That object exports an operation called *Empty*; invoking the *Empty* operation executes the inner object constructor, creating a new object that conforms to the type *Directory*. The code generated when compiling an object constructor is called the *concrete type* of the objects created by execution of the constructor and serves to define the structure of these objects as well as provide the implementation for the operations defined on them.

The goal of supporting the uniform object model for all objects (local or distributed, small or large) may be achieved by a compiler using different implementation styles for objects by deducing the usage pattern and size of each object. The important thing to note is that these details are kept hidden from the programmer, who sees only the uniform object model.

```
const myDirectory : Directory == object oneEntryDirectory
    export Add, Lookup, Delete
    monitor
        var name : String ← nil
        var An : Any ← nil
        operation Add[n : String, o : Any]
            name ← n
            An ← o
        end Add
        function Lookup[n : String] → [o : Any]
            if n = name then
                o ← An
            else
                o ← nil
            end if
        end Lookup
        operation Delete[n : String]
            if n = name then
                name ← nil
                An ← nil
            end if
        end Delete
    end monitor
end oneEntryDirectory
```

Figure 2: A oneEntryDirectory object

1.5 Distribution

Emerald is designed for the construction of distributed applications, using objects as the units of processing and distribution. A programming language for distributed systems must support two broad classes of applications, viz., applications that are genuinely distributed, e.g., replicated nameservers, and centralized applications in a distributed environment, e.g., compilers.

Emerald helps in the construction of both classes: those that are born to distribution, as well as those that have had distribution thrust upon them. For the former, Emerald permits object migration through primitives to control the placement and movement of objects. For the latter, Emerald provides primitives to manipulate and invoke objects in a location-independent manner.

```
const myDirectoryCreator == immutable object oneEntryDirectoryCreator
    export Empty
    operation Empty → [result : Directory]
        result ← object oneEntryDirectory
            export Add, Lookup, Delete
            monitor
                var name : String ← nil
                var An : Any ← nil
                operation Store[n : String, o : Any]
                    name ← n
                    An ← o
                end Store
                function Lookup[n : String] → [o : Any]
                    if n = name then
                        o ← An
                    else
                        o ← nil
                    end if
                end Lookup
                operation Delete[n : String]
                    if n = name then
                        name ← nil
                        An ← nil
                    end if
                end Delete
            end monitor
        end oneEntryDirectory
    end Empty
end oneEntryDirectoryCreator
```

Figure 3: A oneEntryDirectory creator

2 Notation and Vocabulary

This report uses a slight variation of the commonly-used *Extended Backus Naur Form* (EBNF) to express the syntax of Emerald. Terminal symbols in Emerald (i.e. symbols in its vocabulary) are shown either as strings enclosed in quotes (e.g., "," represents a comma) or in bold font (for reserved words such as **loop**); when there is no possibility of confusion, the quotes around terminal symbols are dropped for enhancing readability. Non-terminal symbols are denoted by italicized English words that intuitively illustrate the meaning of the syntactic constructs. In EBNF, alternatives are indicated by "|":

$$A \mid B$$

means choosing either A or B; optional elements are shown using square brackets [ ]:

[A]

means either zero or one A; and (possibly empty) sequences by braces { }:

{ A }

means zero or more repetitions of A.

2.1 Identifiers

An Emerald identifier is a non-empty sequence of letters, digits and the underscore character "_", beginning with a letter or the underscore character. Identifiers are case-insensitive and significant up to 64 characters in length. Identifiers are used as keywords, constant names, variable names, operation names (cf. Section 2.5), parameter names, or local names of types.

2.2 Literals

Literal objects in Emerald are divided into the following categories:

Numeric

The syntax of numeric literals is

numericLiteral	::=	*"0x"* { *hexdigit* }
	|	*"0"* { *octdigit* }
	|	*digit* { *digit* } *"."* { *digit* }
	|	*digit* { *digit* }
digit	::=	*0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9*
octdigit	::=	*0* | *1* | *2* | *3* | *4* | *5* | *6* | *7*
hexdigit	::=	*digit* | *a* | *b* | *c* | *d* | *e* | *f*

Numeric literals without a decimal point (".") denote objects of the predefined type *Integer*; those with decimal points denote objects of the predefined type *Real*. Literals beginning with "0x" are interpreted in hexidecimal;n literals beginning with "0" are interpreted in octal, For example, 12, 014, and 0xc are *Integer* literals representing the decimal number 12, and 2.1 and 215.45 are *Real* literals.

Booleans

booleanLiteral ::= **true** | **false**

The Boolean literals are the reserved words **true** and **false**.

Nil

nilLiteral ::= **nil**

The reserved word **nil** refers to the distinguished nil object.

Characters

A character literal is a character written within single quotes;

characterLiteral	::=	*' character '*
character	::=	*AnyCharacterExceptDoubleQuoteOrBackSlash*
	|	*"\\" anyCharacterExceptUpArrow*
	|	*"\\" "^" anyCharacter*
	|	*"\\" oneTwoOrThreeOctalDigits*

A character literal is a single character written within single quotes. The character \ permits the introduction of escape sequences for the entry of special characters. \\ generates a single \ character, \" generates an embedded ", and \^C where C is any character generates a control character in an implementation-defined manner.[2] Standard escape sequences as in C (\n, \t, etc.) are also permitted. In addition, one, two, or three octal digits following a \ can be used to represent characters by giving their numerical (octal) equivalent.

Examples of characters are 'A', 'r', '\^C', '\\', '\^?', '\^J', '\n' and '\012'; the last three examples equivalently denote the newline character.

Strings

Strings are sequences of characters enclosed in double-quotes, and are permitted to extend over lines.

stringLiteral ::= *""" { character } """*

Examples of strings are "Emerald City", "The \"Evergreen\" State", and "".

[2]In ASCII implementations, \^C generates the ascii character formed by turning off the upper 2 bits in the character code for C. Thus, \^J is the newline character, and \^@ is the null character. The exception is the delete character, (octal 177) which is generated by the sequence \^?.

Vectors

$$\textit{vectorLiteral} \quad ::= \quad \text{“\{”} \ \textit{expression} \ \{ \ \text{“,”} \ \textit{expression} \ \} \ [\ \textit{: typeExpression} \ ] \ \text{“\}”}$$

A vector literal is a sequence of expressions enclosed in curly braces, representing immutable (read-only) vectors. The type of the expression is *ImmutableVector.of*[*t*], where t is either:

- the type expression (if present), otherwise
- the syntactic type of the elements, if they are all the same, otherwise
- *Any*

Examples of literal vectors are {1, 3, 5} (with type *ImmutableVector.of*[*Integer*]), {1, 3, 5 : *Any*} with type *ImmutableVector.of*[*Any*]), and {1, 'a', **true**} (with type *ImmutableVector.of*[*Any*]).

Types

Emerald supports the **record** and **enumeration** forms; these are discussed in Section 7.3. These types, along with typeobject constructors and object constructors (see Sections 7.1 and 8.1), are type literals in Emerald.

Built-in objects such as **self** and **nil**, as discussed in the next section, are also regarded as literals.

2.3 Identifiers

An Emerald identifier is a non-empty sequence of letters, digits and the underscore character "_", beginning with a letter or the underscore character. Identifiers are case-insensitive and significant up to 64 characters in length. Identifiers are used as reserved words, constant names, variable names, operation names (cf. Section 2.5), parameter names, and local names of objects.

2.4 Reserved Identifiers

Reserved identifiers are identifiers that have been reserved for special use and may not be used otherwise as identifiers. Following established convention, this report indicates reserved identifiers in bold font, e.g., **and**.

Reserved identifiers are further subdivided into keywords and literals.

Literals

The reserved literal identifiers are:

false	**nil**	**self**	**true**

Keywords

Emerald keywords are used to delimit language constructs; for example, the keywords **loop** and **end loop** are used to enclose a loop body.

The reserved keywords are:

abstracttype	**all**	**and**	**any**
array	**as**	**assert**	**at**
attached	**awaiting**	**begin**	**boolean**
by	**character**	**checkpoint**	**class**
condition	**confirm**	**const**	**else**
elseif	**end**	**enumeration**	**exit**
export	**failure**	**false**	**field**
fix	**from**	**function**	**if**
immutable	**import**	**initially**	**integer**
isfixed	**locate**	**loop**	**monitor**
move	**nameof**	**nil**	**node**
none	**object**	**on**	**op**
operation	**or**	**ownname**	**owntype**
primitive	**private**	**process**	**real**
record	**recovery**	**refix**	**restrict**
return	**returnandfail**	**self**	**signal**
signature	**string**	**then**	**time**
to	**true**	**type**	**unavailable**
unfix	**union**	**var**	**vector**
attachedvector	**view**	**virtual**	**visit**
wait	**when**	**while**	**where**

Operation names

The following identifiers are reserved as names of built-in operations.

ownType	**ownName**

These operations are supported by the system for every object and may be invoked on any object regardless of type. They may neither be used by any object for operation names nor may they be required by any type. These operations are discussed in depth in Section 7.

Built-in object names

The following reserved words are constants whose values are built-in objects. The operations and usage of these built-in objects are described in Appendix B.

AbstractType	**Any**	**Array**	**Boolean**
Character	**Condition**	**Integer**	**Node**
None	**Real**	**Signature**	**String**
Time	**Vector**		

Literals

As mentioned in Section 2.2, the following identifiers are used as literals:

false	**nil**	**self**	**true**

The keywords **true** and **false** have been discussed in the previous section. The literal **nil** denotes the unique undefined object, and will be discussed later. The literal **self** identifies the object it is contained in; this comes in handy for an object to make operation calls on itself (cf. Section 6).

2.5 Operators

```
operatorCharacter ::=   !   |   #   |   &   |   *
                    |   +   |   -   |   /   |   <
                    |   =   |   >   |   ?   |   @
                    |   ^   |   |   |   ~
operator          ::=   operatorCharacter { operatorCharacter }
```

An operator is a non-empty sequence of operator characters. Operators may be used both as punctuation and as operation names.

Reserved Operators

Two categories of operators are reserved in Emerald, i.e., they may not be used to define new operation names. These categories are the predefined operators and reserved punctuation operators. The predefined operators[3] are:

[3]The symbols `<-`, `->`, and `*>` may be permitted to represent $\leftarrow$, $\rightarrow$, and $\circ\!>$ respectively because these special characters are not commonly available on standard keyboards.

*>	(the conformity operator)
==	(the object definition and identity operator)
!==	(destinction operator)

and the reserved punctuation operators are:

←	(the object naming operator)
→	(the function-returns operator)

These operators are described in Section 4.

2.6 Separators

Separators are sequences consisting of only spaces, tabs, and newlines; they are used to separate consecutive language tokens. Consecutive identifiers, operators and/or numeric literals must be separated by at least one separator.

2.7 Comments

Comments in Emerald are line-oriented. A comment starts on any line with the comment delimiter, %, and terminates at the end of the same line. The comment delimiter is ignored within string and character literals. A comment is lexically equivalent to a separator, and the substitution of a separator for a comment should not affect the semantics of a program.

3 Declarations

Every identifier (other than the reserved words) used in Emerald must be declared. The reserved words are pervasively available throughout an Emerald program. This section presents simple forms of declarations that introduce identifiers as constants or variables used for naming objects. Types and objects are defined in Emerald as discussed in Sections 7 and 8 respectively.

There are two (general purpose) declarative forms: one for constants and one for variables.

declaration	::=	*variableDeclaration*
	|	*constantDeclaration*
variableDeclaration	::=	[**attached**] **var** *identifierList : type* [*initializer*]
constantDeclaration	::=	[**attached**] **const** *identifier* [*: type*] *initializer*
identifierList	::=	*identifier* { "," *identifier* }
initializer	::=	"←" *expression*

A constant declaration introduces an identifier that refers to a single object throughout its lifetime. While a constant identifier always refers to the same object; the object's state may change if it is mutable.

A variable identifier may have its value changed by assignment. When an initializer clause is present, each variable in the list is assigned the value of the given expression. When not explicitly initialized, variables initially name the object **nil**.

The optional **attached** permits the programmer to provide one-way attachment between objects; the relocation of an object additionally relocates all objects attached to it (cf. Section 5.6.4). For example, consider a *stack* object that contains the following declarations:

> **var** *myDir* : *Directory*
> **attached var** *aList* : *linkedList*

This code declares two variables, *myDir* and *aList*, which respectively name objects of abstract type *Directory* and *linkedList*. When *stack* is moved from one node to another, the object named by *myDir* remains behind, but the object named *at that time* by *aList* will be moved along with it. Semantically this makes no difference, and invocations to both objects should work correctly.

All initializers and constant values are evaluated in textual order prior to the execution of any other statements in the block.

3.1 Scope

The scoping of Emerald identifiers is for the most part very traditional. An identifier name is visible throughout the scope in which it is declared, not just textually after that declaration. The following constructs open new scopes for identifiers, and identifiers are imported implicitly into nested scopes where they are not redefined:

- if, elseif, and else clauses (cf. Section 5.2)
- loop statement bodies (cf. Section 5.3.1)
- blocks, unavailable, and failure handlers (cf. Section 5.7)
- operation definitions and signatures (cf. Section 6.1)
- type constructors (cf. Section 7.1)
- object constructor, monitor, process, initially-block and recovery-block definitions (cf. Section 8.1)

The reader is advised to refer to the sections cited above for understanding the corresponding Emerald concept before reading the rest of this section.

Since object constructors and type constructors create new objects that are independent of their enclosing referencing environment (closures), identifiers imported into these constructs are specially treated. When the type or object constructor is executed, all imported identifiers are made constant. Throughout the lifetime of the created type or object, these identifiers will have the values that they had when the object constructor was executed. Consider the following example:

```
for i : Integer ← 0 while i < 10 by i ←i + 1
    o ← object trivial
        export operation getI → [r : Integer]
            r ← i
        end getI
    end trivial
end for
```

This loop creates ten identical objects, except that the value of the identifier i is different for each object. Once the first object (whose $i = 0$) is created, changes to the loop control variable i are not visible to it, as the i that it sees was made constant when that object was created.

4 Expressions

Expressions are Emerald constructs that provide rules for denoting objects

4.1 Literals and Identifier Expressions

```
expression  ::=  literal
              |  constantIdentifier
              |  variableIdentifier
```

A literal expression evaluates to the named object. A constant identifier names the object it was initialized to while a variable identifier names the object most recently bound to it.

4.2 Operator Expressions

Before examining Emerald expressions that involve operators, we define the precedence of the operators used. In Table 1, the operators are ordered by increasing precedence. Operators of the same precedence level are evaluated from left to right.

Precedence Level	Operator	Operation
1	**view-as**	Widen view of object's operations
	restrict-to	Restrict view of object's operations
2	|	Logical or
	or	Logical conditional (short-circuit) or
3	&	Logical and
	and	Logical conditional (short-circuit) and
4	!	Logical negation
5	==, !==	Object identity and distinction
	*>	Type conformity
	=, ! =, <, <=, >=, >	Relational operators
6	+,−	Additive operators
7	*, /	Multiplicative operators
	#	Modulus
	User-defined	
8	−, ˜	Arithmetic negation
	isfixed	Checks if object is fixed at node
	locate	Finds a possible location of the operand
	awaiting	Processes waiting on condition
	nameof	Name of an object
	typeof	Type of an object

Table 1: Precedence of Emerald Operators

4.3 Relational Operators

$$\begin{array}{lcl} \textit{expression} & ::= & \textit{expression}\ \textit{relop}\ \textit{expression} \mid \ldots \\ \textit{relop} & ::= & \text{“ =”} \mid \text{“! =”} \mid \text{“ <”} \mid \\ & & \text{“ <=”} \mid \text{“ >=”} \mid \text{“ >”} \mid \\ & & \text{“ ==”} \mid \text{“! ==”} \mid \text{“*>”} \end{array}$$

The relational operators: =, ! =, <, <=, >, >= respectively compare their operand objects for equality inequality, less than, less than or equal, greater than, and greater than or equal in the usual way. If the comparison holds, the relational expression evaluates to **true**, otherwise it evaluates to **false**. The operator = is also user-definable and may be used to define any operation including, but not limited to equality comparison. The operators == and ! == are predefined and reserved (not user-definable). == evaluates to **true** if the two

4.4 Arithmetic Operators

expression	::=	*expression arithop expression* | ...
arithop	::=	"+" | "−" | "∗" | "/" | "#"

The operators +, −, ∗, / and # represent the addition, subtraction, multiplication, division and modulus operators respectively. The first four of these take two operands of builtin types **integer** or **real**; the last takes operands of type **integer**. The types **integer** and **real** are not inter-convertible; mixed number expressions are not valid and will cause failures[4].

4.5 Boolean Operators

expression	::=	"˜" *expression* |
		expression binbooleanop expression | ...
binbooleanop	::=	"|" | "&" | **or** | **and**

The unary ˜ operator performs the logical negation of its **Boolean** operand. In general, the different binary operators take operands of type **Boolean** and evaluate to **true** or **false**. The operators | and & are the standard logical *or* and *and*. The operator **and** is a *conditional and* and evaluates as follows: if the left operand evaluates to **false**, the result is **false**; otherwise, the result is the value of the right expression. The operator **or** is a *conditional or* and evaluates as follows: if the left operand evaluates to **true**, the result is **true**; otherwise, the result is the value of the right expression.

4.6 Location-related Operators

expression	::=	**locate** *expression*
	|	**isfixed** *expression*

Unary operators, **isfixed** and **locate**, operate on object expressions and evaluate as follows:

isfixed evaluates to **true** if the argument object is currently fixed at a site, otherwise **false**.

locate evaluates to an object (of type **node**) that gives a location of the operand object during the execution of this expression. Applying this to **nil** causes a failure, and to an unavailable object leads to an unavailable exception. This is explained in detail in Section 9.

[4]Note that the **change-view** expression will not permit the conversion of objects of type **integer** to **real** and vice versa because they do not conform to one another; however, the functions, *asReal* and *asInteger*, may be used to perform any necessary conversion.

4.7 Other Expressions

$$
\begin{array}{lcll}
\textit{expression} & ::= & \textbf{awaiting} \;\; \textit{expression} & \\
 & | & \textbf{view} \;\; \textit{expression} \;\; \textbf{as} \;\; \textit{typeDenotation} & \\
 & | & \textbf{restrict} \;\; \textit{expression} \;\; \textbf{to} \;\; \textit{typeDenotation} & \\
 & | & expression \, \$ \, fieldIdentifier & \\
 & | & expression \, . \, enumIdentifier & \\
 & | & objectConstructor & \\
 & | & functionInvocation & \ldots \\
 & | & \textbf{nameof} \textit{expression} & \\
 & | & \textbf{typeof} \textit{expression} &
\end{array}
$$

The **view** expression permits an object to be regarded as being of a different type, subject to the restriction that no object expression be viewed as a type it does not conform to. In other words, this expression permits the user to widen the type of an object. The **restrict** expression permits an object expression to have a restricted view of its operations. This means that further widening of the reference to the object will not be permitted beyond the restricted type. The initial type of the object expression must conform to that of the restricted type.

An expression of the form, *complex$realPart*, is used to access the stated field of the record, *complex*. For convenience, Emerald permits this sugared syntax to be used for invocations to records. An expression of the form, *spectrum.indigo*, is used for an enumeration. Both these expressions are discussed in detail in Section 7.3.

The **awaiting** operator takes as its operand an expression of type *Condition* and returns **true** if at least one process is suspended on the operand condition, and **false** otherwise (cf. Section 8).

nameof, and **typeof** return the the name (a String) or the type (a Signature) of any object (including **nil**). See the appendix for a description of these builtin types.

4.8 Other Operators

All other operators are translated into object invocations. Each occurrence of a unary operator is translated into an invocation of the operand with the invocation name being the name of the operator and with no arguments. Each occurrence of a binary operators is translated into an invocation of the left operand with the invocation name being the name of the operator and with a single argument which is the right operand. For example:

$!e$	is translated as	$e.!$
$a + b$	is translated as	$a.{+}[b]$

The description of function invocation expressions and object constructor expressions is deferred to Sections 6 and 8 respectively.

5 Statements

Statements are used to perform all computation in Emerald. This section describes the various statements, starting with the assignment statement and the control structures. Statements for assertion, invocation and dealing with location-dependence are discussed next. This is followed by the discussion of specific statements for handling object unavailability and failures. Concurrency features are sketched out, postponing a detailed discussion to Section 8. Finally, statements used to checkpoint, to make primitive system calls, and to propagate failures are described.

5.1 Assignment statement

```
assignment  ::=  variableIdentifierList "←" expressionList
              |  [ variableIdentifierList "←" ] procedureInvocation
```

In the first case, the expression list is evaluated to yield a number of objects. In the latter case, the procedure invocation is performed, resulting in a number of objects (possibly 0). In both cases, the resulting objects are positionally bound to the variables on the left side of the assignment operator. The number of variables on the left side and the number of resulting objects on the right must be equal and must positionally conform in type.

5.2 Selection

```
ifStatement  ::=  if expression then
                          declarationsAndStatements
                  { elseif expression then
                          declarationsAndStatements }
                  [ else
                          declarationsAndStatements ]
                  end if
```

The expressions following the **if** and optional **elseif** keywords (which must be of type **Boolean**) are evaluated in textual order until one evaluates to **true** or none evaluate to **true**. In the former case, the statements following the next **then** keyword are executed, and in the latter case, the statements following the **else** keyword (when present) are executed.

5.3 Iteration

5.3.1 Loop statement

```
loopStatement  ::=  loop
                        declarationsAndStatements
                    end loop
```

The statements bracketed by **loop** and **end loop** are executed repeatedly until an exit statement at the same level of nesting is executed.

5.3.2 Exit statement

exitStatement ::= **exit** [**when** *expression*]

This statement terminates the execution of the textually inner-most enclosing loop; this statement is invalid if there is no such loop. The simple **exit** provides an unconditional exit from the loop; the optional **when** clause permits a conditional exit if the evaluated expression, which must be of type **Boolean**, evaluates to **true**. **exit when** *expression* is exactly equivalent to **if** *expression* **then exit end if**, but is somewhat easier to read and type.

5.3.3 For statement

Emerald has two forms of the for statement. These are conveniences whose semantics are defined in terms of their translations as given below.

```
forStatement  ::=  for ( initial : condition : step )
                       declarationsAndStatements
                   end for
```

This is equivalent to:

```
begin
    initial
    loop
        exit when !condition
        begin
            declarationsAndStatements
        end
        step
    end loop
end
```

```
forStatement  ::=  for identifier : typeExpression initialization while condition by step
                           declarationsAndStatements
                   end for
```

This is equivalent to

```
begin
   var identifier : typeExpression initialization
   loop
      exit when !condition
      begin
         declarationsAndStatements
      end
      step
   end loop
end
```

5.4 Assertions

assertStatement ::= **assert** *expression*

The expression, whose type must be **Boolean**, is evaluated. If the result is **false**, a failure occurs (as explained in Section 9.2). If the result is **true**, the statement has no further effect.

5.5 Invocations

procedureInvocation	::=	*expression* "." *operationName* [*argumentList*]
operationName	::=	*identifier* | *operator*
argumentList	::=	"[" *argument* { "," *argument* } "]"
argument	::=	[**move**] [**visit**] *expression*

An invocation statement specifies the target object, the operation to be invoked, and the required arguments. When an invocation returns results, they are assigned to variables using an assignment (see Section 5.1). The keywords **move** and **visit** suggest that the expression be physically moved to the same node as the invoked object; **visit** further suggests that the expression be moved back after the invocation is performed. Invocations are discussed in Section 6 and these parameter passing modes in Section 6.2.

5.6 Location-related Statements

Mobility is an important feature of Emerald([Jul 88a, Jul 88b]) and is supported via several language constructs. The statements that permit the programmer to specify and change the location of the argument objects are discussed below.

5.6.1 The Fix statement

fixStatement ::= **fix** $expression_1$ **at** $expression_2$

The object named by $expression_1$ is moved to the location of the object named by $expression_2$, and forced to remain there until explicitly **unfixed**; the *unfix* and *refix* statements described below permit the movement of previously fixed objects. Fixing objects at object **nil**, and attempts to move or fix previously fixed objects result in failures (cf. Section 9.2).

5.6.2 The Unfix statement

unfixStatement ::= **unfix** *expression*

The object denoted by the *expression* is made free to move. It is not an error to unfix an object not currently fixed at any location.

5.6.3 The Refix statement

refixStatement ::= **refix** $expression_1$ **at** $expression_2$

This statement unfixes the object named by $expression_1$ and fixes it at some (presumably different) node; the **refix** is performed atomically.

5.6.4 The Move statement

moveStatement ::= **move** $expression_1$ **to** $expression_2$

The object denoted by $expression_1$ is moved to the current location of the object denoted by $expression_2$. The statement fails if the object denoted by $expression_1$ is fixed.

The **move** primitive is actually a hint, i.e., the implementation is not required to perform the move suggested. On the other hand, the primitives **fix** and **refix** have stronger semantics, and when they succeed, the object must stay at the specified destination until explicitly unfixed.

5.7 Compound statement

```
compoundStatement   ::=  begin
                                 blockBody
                         end

blockBody           ::=  declarationsAndStatements
                         [unavailableHandler]
                         [failureHandler]

unavailableHandler  ::=  when identifier [ " :" typeDenotation ] unavailable
                                 declarationsAndStatements
                         end unavailable

failureHandler      ::=  on failure
                                 declarationsAndStatements
                         end failure
```

The compound statement permits several statements to be grouped together as one composite statement. In addition, it permits suitable recovery code to be attached in the form of handlers dealing with object unavailability and failures (cf. Sections 9.1 and 9.2).

5.8 Concurrency

Concurrency features are described in detail in Section 8 and are briefly outlined here. Each object may have an optional process associated with it; this process is created after the termination of the object's **initially** section and it executes until it reaches the end of its block. Each object may also have a possibly empty monitored section in which mutual exclusion is guaranteed. Objects of system-implemented type **Condition** with Hoare monitor-condition semantics are available here.

The **wait** and **signal** statements (available only within monitors) permit process synchronization as described below. Note that a condition object used in a **wait** or **signal** statement (or an **awaiting** expression) must be used only inside the monitor of the object by which it was created.

5.8.1 Wait statement

```
waitStatement  ::=  wait expression
```

The **wait** statement must be executed inside a monitored section, and the *expression* must evaluate to a **condition** object. The process executing the **wait** is suspended on the condition object, and the monitor lock is passed on to the next process waiting to enter the monitor; if no process is waiting to enter, the monitor lock is released.

5.8.2 Signal statement

signalStatement ::= **signal** *expression*

The *expression* must evaluate to a **condition** object. If the condition object has one or more processes suspended on it, one of these processes will be resumed, the monitor lock will be passed to it, and the signalling process is placed at the head of the monitor entry queue. Finally, if the condition object does not have any processes suspended on it, the signal statement has no effect.

5.9 The Checkpoint statement

checkpointStatement	::=	**checkpoint** [**AT** *destination*]
destination	::=	*expression* | **ALL**

The checkpoint statement permits an object to store its state on permanent storage. On node failure and subsequent recovery, the object uses this stored state and continues from that state, first performing any programmer-specified recovery action. It is only allowed within the monitored section of an object.

5.10 The Return Statement

returnStatement ::= **return**

This statement is used to terminate the execution of an operation and return to the invoking object. It may also be used to prematurely terminate an initially, process, or recovery section.

5.11 The ReturnAndFail statement

returnAndFailStatement ::= **returnandfail**

The "return and fail" statement is analagous to the return statement, but in addition, it permits the invoked object to report a failure to the invoking object. The return happens first, so the state of the invoked object is not affected by the failure.

5.12 The Primitive Statement

primitiveStatement	::=	**primitive** *primitiveImplementation*
		"[" *identifier* { , *identifier* } "]" ← "[" *identifier* { , *identifier* }
primitiveImplementation	::=	*identifier* | *integerLiteral*

This statement is used to implement lower-level calls to the underlying operating system and to implement certain operations on builtin-types.

The list of identifiers on the right of the assignment operator provide the arguments for the primitive, which the list of identifiers on the left of the assignment get the results of the primitive.

There is no check that the implementation of the primitive actually expects the number of arguments or returns the number of results that the arg/resultvars lists mention. You are assumed to be careful using these things.

6 Operations

Emerald objects communicate with one another only through the invocation of operations. This section describes the definition and invocation of operations.

6.1 Defining operations

operationSignature	::=	*operationKind* *operationName* [*parameterList*]
		["→" *parameterList*] { *clause* }
operation	::=	[**private**|**export**] *operationSignature*
		blockBody
		end *operationName*
operationKind	::=	**op** | **operation** | **function**
parameterList	::=	"[" *parameter* { "," *parameter* } "]"
parameter	::=	*parameterKey* [*identifier :*] *type*
parameterKey	::=	[**move**] [**attached**]
	|	**attached move**
Clause	::=	**where**
		{ *identifier* *whereOperator* *typeDefinition* }
		end where
whereOperator	::=	"←" | "*>" | "=="

Emerald provides two kinds of operations: procedural and functional. Procedural operations are heralded by the keyword **operation**, while the keyword **function** indicates a functional operation. In declaring a functional operation, the programmer asserts that the operation is side-effect free, i.e., the abstract state of the system is not modified by the execution of the operation[5]. Note that the burden is on the programmer; the Emerald system may perform optimizations on function invocations that are incorrect if the operation has side effects.

Private operations may only be declared within the monitor, and called from within the monitor. They do not acquire the monitor lock (since the caller must already hold the lock). They are intended for auxiliary operations shared by multiple monitored operations.

The operation signature (cf. Section 7) includes the operation name and the number, names and abstract types of the arguments and results. Its **where** clause serves two purposes: if the operator is $\leftarrow$ or $==$, the identifier becomes a new constant whose value is the given type; if the operator is $\circ\!>$, type constraints for the given formal parameters are imposed. This clause becomes useful for the implementation of polymorphic types (cf. Section 8.3).

6.2 Parameter Passing

The Emerald language uses call-by-object-reference semantics for all invocations, local or remote. Because Emerald objects are mobile, it may be possible to optimize by avoiding many remote references by moving argument objects to the site of a remote invocation. Emerald provides mechanisms for the Emerald programmer to explicitly move objects. This is through parameter passing modes called *call-by-move* and *call-by-visit*. In both modes, at actual invocation time, the argument object is relocated to the destination site. Following the call the argument object may either return to the source of the call or remain at the destination site; the former mode takes place in *call-by-visit* and the latter in *call-by-move*. Neither mode affects the location-independent semantics of the invoked operation.

6.3 Making Invocations

An invocation of an operation that returns exactly one result may be used as an expression. Any operation may be invoked in an assignment statement.

Executing an operation invocation involves:

- evaluating the invocation target expression,
- evaluating the argument objects and then positionally assigning them to the formal parameters of the operation,
- executing the body of the operation in the context of the target object of the invocation, and
- returning the final values of any output parameters of the invocation.

[5]Note that Emerald does not rule out the possibility of the operation having concrete side-effects (sometimes termed *beneficial* or *benevolent* side-effects).

7 Types

An abstract type is defined as a collection of operation signatures, where each operation signature includes the operation name, and the names and types of its arguments and results. Abstract types, being objects, are first-class citizens in Emerald[6]. Each type object exports a function without arguments called *getSignature* that returns an object of the predefined **Signature** type. In other words, any object that conforms to the following type:

```
immutable type aType
   function getSignature → [Signature]
end aType
```

is a type. Note that each object with type *signature* has a getSignature operation that returns the target object, thus Signatures are Types.

7.1 Type Constructors

Types are created using type constructors. A type constructor has the following structure:

```
typeConstructor  ::=  [ immutable ] type typeIdentifier
                          { operationSignature }
                      end typeIdentifier
```

Operation signatures have been defined in Section 6.1, however in type constructors, the identifiers in parameter declarations may be omitted. An immutable type implies that its objects are abstractly immutable, i.e. its objects cannot change value over time. For example, the predefined type **Integer** is immutable because its objects represent integer values which cannot change; for instance, the integer 3 cannot be changed to the integer 4.

```
type Directory
   operation Add[name : String, thing : Any]
   operation Lookup[name : String] → [thing : Any]
   operation Delete[name : String]
end Directory
```

This constructor is executable, and when executed causes the creation of an immutable object conforming to **AbstractType**. The execution of the *getSignature* operation on the resulting object returns itself: a type requiring the three operations *Add*, *Lookup*, and *Delete*.

[6]However, Emerald requires all expressions in abstract type positions in variable declarations, constant declarations, and view expressions to be manifest; this restriction makes Emerald statically-typed.

7.2 Conformity

Some types in the system are exceptions to the standard conforms and matching rules. For ensuring correctness, types such as **Boolean**, **Condition**, **Node**, **Signature** and **Time** must be implemented only by the system. For performance enhancement, the types **Character**, **Integer**, **Real**, and **String** are also restricted to be implemented only by the system.

7.3 Other types

Emerald defines four syntactic abbreviations for commonly occurring constructions.

7.3.1 Classes

While Emerald does not have a notion of *class*, we do recognize that it is often convenient to do class-based programming. Therefore the Emerald compiler implements a syntactic extension called a class, and supports a form of inheritance (by macro expansion) for classes.

class	::=	**class** *identifier* ["(" *baseClass* ")"] [*parameterList*]
		{ *classoperation* }
		{ *declaration* }
		[*monitor*|*initially*]
		{ *operation* }
		[*process*]
		end *identifier*
baseClass	::=	*identifier*
classoperation	::=	**class** *operation*

Classes are expanded syntactially into two nested object constructors. The outer object is immutable and has operations getSignature and create in addition to the class operations defined by the programmer. The parameter list is the parameter list to the create operation, and instance constants with those names will be declared for each instance. The rest of the components of the class construct go to defining the body of the inner object constructor.

Inheritance is syntactic. Each component in the base class that is not redefined in the subclass will be inherited into the subclass. Because any component can be redefined, there is no guarantee that the type of a subclass will conform to the type of its superclass.

An example might help. Suppose we write the following declaration:

```
const Complex ← immutable class Complex[r : Real, i : Real]
    class export operation fromReal[r : Real] → [e : Complex]
        e ← self.create[a, 0.0]
    end fromReal
    export function +[other : Complex] → [e : Complex]
        e ← Complex.create[other.getReal + r, other.getImag + i]
    end +
    export function getReal → [e : Real]
        e ← r
    end getReal
    export function getImag → [e : Real]
        e ← i
    end getImag
end Complex
```

This is rearranged into the following:

```
const Complex ← immutable object Complex
    const ComplexType ← immutable typeobject ComplexType
        function getReal → [ComplexType]
        function getImag → [ComplexType]
        function +[ComplexType] → [ComplexType]
    end ComplexType
    export function getSignature → [r : Signature]
        r ← ComplexType
    end getSignature
    export operation fromReal[a : Real] → [e : Complex]
        e ← self.create[a, 0.0]
    end fromReal
    export operation create[r : Real, i : Real] → [e : Complex]
        e ← immutable object aComplex
            export function +[other : Complex] → [e : Complex]
                e ← Complex.create[other.getReal + r, other.getImag + i]
            end +
            export function getReal → [e : Real]
                e ← r
            end getReal
            export function getImag → [e : Real]
                e ← i
            end getImag
        end aComplex
    end create
end Complex
```

7.3.2 Enumerations

```
enum ::= enumeration identifier
                 enumIdentifier { "," enumIdentifier }
         end identifier
```

Enumerations provide a creation operation for each element of the type, operations *first* and *last* that return the first and last elements of the enumeration, respectively, and an operation *create* that takes an integer argument and returns that element of the enumeration with that ordinal (0 base). In addition, elements of the enumeration support <, <=, =, ! =, >=, >, *succ*, *pred*, *ord* and *asString* operations. The *asString* operation returns the name of the element as a *String*. All elements of enumerations are immutable.

To be concrete, consider the declaration:

const *colors* ← *enumeration colors red, blue, green* **end** *colors*

The identifier *colors* will have type:

```
immutable typeobject ColorCreatorType
    function getSignature → [Signature]
    operation create[Integer] → [ColorType]
    operation first → [ColorType]
    operation last → [ColorType]
    operation red → [ColorType]
    operation green → [ColorType]
    operation blue → [ColorType]
end ColorCreatorType
```

Each element of the enumeration has type:

```
immutable typeobject ColorType
    function <[ColorType] → [Boolean]
    function <=[ColorType] → [Boolean]
    function =[ColorType] → [Boolean]
    function ! =[ColorType] → [Boolean]
    function >=[ColorType] → [Boolean]
    function >[ColorType] → [Boolean]
    function succ → [ColorType]
    function pred → [ColorType]
    function ord → [Integer]
    function asString → [String]
end ColorType
```

7.3.3 Fields

```
field ::= [ attached ] field identifier : type [ initializer ]
        | [ attached ] const field identifier : type initializer
```

It is often convenient to declare an externally accessible data element of an object. A field declaration does exactly this. Field declarations can only occur with the declaration part of an object constructor. Each field declaration expands to a variable declaration and two operation definitions (first case) or to a constant declaration and one operation definition (second case). The expansion of:

```
    attached field a : b ← c
is
    attached var a : b ← c
    export operation setA[x : b]
        a ← x
    end setA
    export function getA → [x : b]
        x ← a
    end getA
```

Where the identifier x is chosen to not conflict with any other identifier. Constant fields expand to constant declarations and only the *getA* operation.

7.3.4 Records

```
record ::= [ immutable ] record identifier
                            field { field }
           end identifier
field  ::= [ attached ] var fieldIdentifier : type
```

A record declaration:

```
    immutable record aRecord
        var a : b
        var c : d
    end aRecord
expands to a class:
    immutable class aRecord[xa : b, xc : d]
        field a : b ← xa
        field c : d ← xc
    end aRecord
```

Emerald supports syntactic sugar to facilitate accessing the fields of record-like objects. The syntactic forms are:

```
fieldSelection ::= expression "$" identifier
subscript      ::= expression "[" expression { "," expression } "]"
```

In an expression context these are translated as follows:

a\$*b*	is translated as	*a.getB*
a[*b, c, d*]	is translated as	*a.getElement*[*b, c, d*]

Predefined Type	Type Description
Type	To be used as types in declarations.
Any	Has no operations.
Array	A polymorphic, flexible array.
BitChunk	Arbitrary bit-level operations.
Boolean	Logical values with literals **true** and **false**.
Character	Individual characters with operations such as <, >, =, ord, etc.
Condition	Condition variables satisfying Hoare monitor semantics.
ImmutableVector	Read-only vector.
InStream	Input streams.
Integer	Signed integers.
Node	Objects representing machines.
NodeList	Immutable vectors of node descriptions.
NodeListElement	Immutable node descriptions.
None	The type of **nil**.
OutStream	Output streams.
Real	Approximations of real numbers.
Signature	Primitive abstract type
String	Character strings.
Time	Times and dates
Vector	Fixed sized polymorphic vectors.
ConcreteType	Executable code.

Table 2: Built-in Types

In an assignment context, these are translated as follows:

a$b ← c	is translated as	a.setB[c]
a[b, c, d] ← e	is translated as	a.setElement[b, c, d, e]

7.4 Predefined types

Emerald implements a number of pre-defined objects; these objects are outlined in Table 2 and specified in greater detail in Appendix B.

7.5 Multiple Implementations

The Emerald type system permits objects with different implementations to have the same type (cf. Section 1.1). An implicit many-to-one relationship exists between types and objects in both directions, i.e., a number of objects may all conform to the same abstract type, and a single object may conform to a number of abstract types. This is rearranged into the following:

Each object identifier, i.e., an identifier that may be bound to an object, is typed abstractly; its abstract type determines which operations may be performed on the object it names. For example, the declaration

var *d : Directory*

declares that any object named by the identifier *d* within this scope will be of abstract type *Directory*. There is, however, no restriction on the objects named by *d* other than that they must implement the *Directory* abstraction. Furthermore, no matter what operations an object named by *d* implements, only operations valid for directories may be performed on the object that *d* names. Since any given object may implement more than one abstract type, Emerald provides a mechanism for altering the abstract type used to view the object (cf. Section 4.7).

8 Objects

Most so-called object-oriented languages such as Smalltalk, C++, Eiffel, and the Eden Programming Language (EPL) have a concept of *class*, which may be regarded as an object defining the behavior of a number of objects, i.e., its instances. Emerald takes a different, more elegant approach to the creation of objects. It provides a single general purpose way of constructing objects: the object constructor.

8.1 Object Constructors

An object constructor defines the complete representation and operations of a single object as well as its active behavior. Objects are created when an object constructor is executed. In other words, object constructors are expressions. The form of a constructor shown below demonstrates its generality, i.e., all Emerald objects may be defined using this feature.

```
objectConstructor  ::=  [ immutable ] object identifier
                                 { declaration }
                                 [ monitor|initially ]
                                 { operation }
                                 [ process ]
                        end identifier

monitor            ::=  monitor
                                  { declaration }
                                  { operation }
                                  [ initially ]
                                  [ recovery ]
                        end monitor

process            ::=  process
                                 blockBody
                        end process

initially          ::=  initially
                                 blockBody
                        end initially

recovery           ::=  recovery
                                 blockBody
                        endrecovery
```

Each object in Emerald owes its existence to either an implicit or explicit execution of an object constructor. The object constructor provides the necessary information about the object's implementation, i.e.,

- Representation declarations for data and processes that are contained in instances of the type.
- A collection of operation signatures, where each operation signature includes the name of the operation, and the number and types of the parameters of the operation.
- A collection of operation bodies, i.e., the implementation for the previously specified operations.

Figures 2 and 3 illustrate the usage of object constructors to create new objects.

Further discussion of object constructors, and their usefulness in (distributed) programming may be found in [Hutchinson 87a].

8.2 Objects as Types

Type constructors have already been described as one method for constructing abstract types in Emerald (cf. Section 7). Since typing in Emerald is based entirely on the signatures of operations, any object which conforms to the type

```
immutable type AbstractType
    function getSignature → [Signature]
end AbstractType
```

is a type. Thus objects which serve other useful purposes can also be used as types. Object creators in particular can take advantage of this to allow a single object to serve as both a type and a creator.

8.3 Polymorphism Example

To demonstrate the polymorphism present in Emerald, a polymorphic *Set* object is presented in Figure 4. The *Set* implemented has an operation *of* that takes a type as an argument and returns an object that can be used as the abstract type of, as well as a creator of sets of things conforming to the original argument to the operation *of*. The elements put in this set are immutable and must implement an = operation that returns a **Boolean** object. With this *Set* definition, we can define a set of integers as:

const *setOfInteger* == *Set.of* [**Integer**]

Hutchinson [Hutchinson 87a] provides more details about polymorphism in Emerald.

9 Location and Reliability

Emerald was developed primarily to facilitate the construction of distributed application programs. To be resilient to machine crashes, these programs should be capable of detecting and recovering from such crashes. They should also be able to control the location of component objects so that the available nodes in the system are optimally exploited. This section discusses the Emerald location-related constructs that help in the development of fault-tolerant software.

There are two Emerald concepts that concern location. These correspond to two desires that motivate application programmers to deal with location. As stated previously, invocation in Emerald is location independent. This means that the location of an object need not be determined in order to invoke it. There are however two considerations that we expect to motivate application programmers to concern themselves with location: performance and reliability/availability.

```
const Set == immutable object Set
  export of

  function of[eType : AbstractType] → [result : NewSetType]
    where
      eType ⊳
        immutable type eType
          function =[eType] → [Boolean]
        end eType
      NewSetType ==
        immutable type NewSetType
          operation new → [NewSet]
          operation singleton[eType] → [NewSet]
          operation create[Vector.of[eType]] → [NewSet]
        end NewSetType
      NewSet ==
        immutable type NewSet
          function contains[eType] → [Boolean]
          function with[eType] → [NewSet]
          function without[eType] → [NewSet]
          function choose → [eType]
          function +[NewSet] → [NewSet]
          function ∗[NewSet] → [NewSet]
          function −[NewSet] → [NewSet]
          function cardinality → [Integer]
        end NewSet
    end where

    result ←
      object NewSetType
        export create

        operation create[v : Vector.of[eType]]→ [result : NewSet]
          result ←
            object NewSet
              export contains, with, without, choose, +, ∗, − , cardinality

              const repType == Vector.of[eType]
              var rep : repType

              % The implementation of these operations and functions.
          end NewSet
        end create
      end NewSetType
  end of
end Set
```

Figure 4: A Polymorphic Set Object

Performance

Since remote invocation will necessarily be at least an order of magnitude more expensive than local invocation, the placement of Emerald objects may seriously affect their performance. In order to provide the programmer with control over the placement of objects the move statement (see Section 5.6.4) is provided. In addition, the call-by-move implementation strategy for arguments to invocations (see Section 6.2) allows further optimizations.

Reliability and Availability

Since an object may be moved at arbitrary times by any other object with a reference to it, a more permanent binding between objects and locations is often required. In particular, in order to implement an available replicated service, it is necessary to place the replicas on differing machines and not allow them to move. This allows the programmer to guarantee that a single machine failure will not cause more than one of his replicas to become unavailable.

In order to provide for this requirement, the fix and unfix statements (see Sections 5.6.1 and 5.6.2) may be used. An object, once fixed at a particular location, may not be moved from there. Any attempt to do so will fail (see subsection 9.2).

9.1 Unavailable objects

Due to machine crashes or communication network failures, objects may be temporarily or permanently unavailable. Emerald provides unavailable handlers to allow programmers to detect such situations, and attempt recovery.

```
unavailableHandler ::=
        when [ identifier [ ":" typeDenotation ] ] unavailable
            declarationsAndStatements
        end unavailable
```

An object is regarded as being **unavailable** when it cannot be located at any available node following suitable system action [Jul 88b]. The unavailable-handler specifies the action to be taken on object unavailability.

9.2 Failures

Failures can result from a number of causes; these include attempting to invoke a **nil** reference, assertion failures, divide-by-zero and subscript-range errors.

```
failureHandler ::=
        on failure
            declarationsAndStatements
        end failure
```

After a failure is detected, the following action is taken.

1. The appropriate failure handler to execute is found. This handler is the handler attached to the smallest block containing the statement. An omitted handler is equivalent to the following:

   ```
   on failure
       assert false
   end failure
   ```

   This handler causes the block to fail, and the algorithm is restarted in the enclosing block.

2. If the block body of a monitored operation, the initially section, or the recovery section of an object fails, then the object is said to have failed. Any subsequent invocation attempted on the object will fail, and any invocations that have started but not yet completed also fail.

3. A failure in the block body of an operation is propagated by causing the corresponding invocation statement to fail.

4. A failure in an initially section implies that the object creation has failed; this is propagated by causing the statement containing the object constructor expression to fail.

5. A failure in the block body of a recovery section cannot be propagated because its execution did not result from an invocation. So the recovery fails and the object remains unavailable.

6. A failure in a process block body cannot be propagated, but the object itself does not fail.

7. When an object fails, no attempt is made to immediately track down and fail all processes (including the one contained in the object) that have threads of control that have passed through the object. When these threads of control return to the body of any operation inside the object, they will then fail.

A Syntax of Emerald

This appendix summarizes the syntax of Emerald discussed in this report. It should be borne in mind that Emerald is an active research language and is constantly being modified. The syntax given in this appendix is the YACC-grammer for Emerald

compilation	::=	*constantDeclarationS*
constantDeclarationS	::=	*empty*
	|	*constantDeclarationS environmentImport*
	|	*constantDeclarationS environmentExport*
	|	*constantDeclarationS constantDeclaration*
empty	::=	
environmentImport	::=	**import** *symbolDefinitionS* **from** *environmentPathName*
environmentExport	::=	**export** *symbolReferenceS* **to** *environmentPathName*
environmentPathName	::=	**string**
typeClauseOpt	::=	*empty*
	|	*typeClause*
typeClause	::=	" :" *typeDefinition*
typeDefinition	::=	*invocation*
builtinType	::=	**abstracttype**
	|	**any**
	|	**array**
	|	**boolean**
	|	**character**
	|	**condition**
	|	**integer**
	|	**node**
	|	**none**
	|	**signature**
	|	**real**
	|	**string**
	|	**time**
	|	**vector**
	|	**attachedvector**
optVariable	::=	**var**
	|	*empty*
abstractType	::=	**type** *optVariable symbolDefinition operationSignatureS* **end** *symbolReference*
record	::=	**record** *symbolDefinition recordFieldS* **end** *symbolReference*

recordFieldS	::=	*recordField*
	|	*recordFieldS recordField*
recordField	::=	**var** *symbolDefinitionS typeClause*
	|	**attached var** *symbolDefinitionS typeClause*
union	::=	**union** *symbolDefinition recordFieldS* **end** *symbolReference*
enumeration	::=	**enumeration** *symbolDefinition symbolDefinitionS*
		end *symbolReference*
export	::=	*empty*
	|	**export** *operationNameReferenceS*
operationNameReferenceS	::=	*operationNameReference*
	|	*operationNameReferenceS* "," *operationNameReference*
symbolDefinitionS	::=	*symbolDefinition*
	|	*symbolDefinitionS* "," *symbolDefinition*
symbolReferenceS	::=	*symbolReference*
	|	*symbolReferenceS* "," *symbolReference*
operationSignatureS	::=	*empty*
	|	*operationSignatureS operationSignature*
operationSignature	::=	*operationKind operationNameDefinition parameterClause*
		returnClause whereClause
operationKind	::=	**operation**
	|	**op**
	|	**function**
parameterClause	::=	*empty*
	|	"[" "]"
	|	"[" *parameterS* "]"
parameterS	::=	*parameter*
	|	*parameterS* "," *parameter*
parameterFirstExpression	::=	*expression*
	|	**attached** *expression*
	|	**move** *expression*
	|	**attached move** *expression*
	|	**move attached** *expression*
parameter	::=	*parameterFirstExpression*
	|	*parameterFirstExpression* " :" *expression*
returnClause	::=	*empty*
	|	" $\rightarrow$" "[" "]"
	|	" $\rightarrow$" "[" *parameterS* "]"
whereClause	::=	*empty*
	|	**where** *whereWidgitS* **end where**
whereWidgitS	::=	*whereWidgit*
	|	*whereWidgitS whereWidgit*
whereWidgit	::=	*symbolDefinition whereOperator expression*

whereOperator	::=	“ ==”
	|	“ ←”
	|	“ ∘>”
object	::=	**object** *symbolDefinition export declarationS monitoredPart operationDefinitionS processDefinition* **end** *symbolReference*
creators	::=	*empty*
	|	*creators* **type** *operationDefinition*
obase	::=	*empty*
	|	“(” *symbolReference* “)”
virtuals	::=	*empty*
	|	**virtual** *operationSignatureS* **end virtual**
class	::=	**class** *symbolDefinition obase parameterClause exportcreatorsdeclarationS monitoredPart operationDefinitionS processDefinition virtuals* **end** *symbolReference*
declarationS	::=	*empty*
	|	*declarationS declaration*
attached	::=	**attached**
	|	*empty*
declaration	::=	*attached declarationprime*
declarationprime	::=	*constantDeclaration*
	|	*variableDeclaration*
	|	*fieldDeclaration*
	|	*error*
constantDefOp	::=	“ ==”
	|	“ ←”
fieldDeclaration	::=	**field** *symbolDefinition typeClause initializerOpt*
	|	**const field** *symbolDefinition typeClause initializerOpt*
constantDeclaration	::=	**const** *symbolDefinition typeClauseOpt constantDefOp expression*
initializerOpt	::=	*empty*
	|	“ ←” *expression*
initializer	::=	“ ←” *expression*
variableDeclaration	::=	**var** *symbolDefinitionS typeClause initializerOpt*
monitoredPart	::=	*empty*
	|	**monitor** *declarationS operationDefinitionS initiallyDefinition recoveryDefinition* **end monitor**
	|	**initially** *blockBody* **end initially**
operationDefinitionS	::=	*empty*
	|	*operationDefinitionS operationDefinition*
private	::=	**private**
	|	*empty*

operationDefinition	::=	*private oexport operationSignature blockBody* **end** *operationNameReference*
blockBody	::=	*declarationsAndStatements unavailableHandler failureHandler*
initiallyDefinition	::=	*empty*
	|	**initially** *blockBody* **end initially**
recoveryDefinition	::=	*empty*
	|	**recovery** *blockBody* **end recovery**
processDefinition	::=	*empty*
	|	**process** *blockBody* **end process**
declarationsAndStatements	::=	*declarationS statementS*
statementS	::=	*empty*
	|	*statementS statement*
statement	::=	*ifStatement*
	|	*loopStatement*
	|	*forStatement*
	|	*exitStatement*
	|	*assignmentOrInvocationStatement*
	|	*assertStatement*
	|	*fixStatement*
	|	*refixStatement*
	|	*unfixStatement*
	|	*moveStatement*
	|	*compoundStatement*
	|	*primitiveStatement*
	|	*waitStatement*
	|	*signalStatement*
	|	*checkpointStatement*
	|	*returnStatement*
	|	*returnAndFailStatement*
	|	*error*
optDeclaration	::=	*empty*
	|	*symbolDefinition typeClause*
unavailableHandler	::=	*empty*
	|	**when** *optDeclaration* **unavailable** *blockBody* **end unavailable**
failureHandler	::=	*empty*
	|	**on failure** *blockBody* **end failure**
ifClauseS	::=	*ifClause*
	|	*ifClauseS* **elseif** *ifClause*
ifClause	::=	*expression* **then** *declarationsAndStatements*
elseClause	::=	*empty*
	|	**else** *declarationsAndStatements*
ifStatement	::=	**if** *ifClauseS elseClause* **end if**

forStatement	::=	**for** “(” *assignmentOrInvocationStatement* “ :”
		expression “ :” *assignmentOrInvocationStatement* “)”
		declarationsAndStatements
		end for
	|	**for** *symbolDefinition typeClause initializer*
		while *expression* **by** *assignmentOrInvocationStatement*
		declarationsAndStatements
		end for
loopStatement	::=	**loop** *declarationsAndStatements* **end loop**
exitStatement	::=	**exit** *whenClause*
whenClause	::=	*empty*
	|	**when** *expression*
assignmentOrInvocation−Statement	::=	*expressionS*
	|	*expressionS assignmentOp expressionS*
assignmentOp	::=	“ ←”
	|	“ :=”
assertStatement	::=	**assert** *expression*
fixStatement	::=	**fix** *expression* **at** *expression*
refixStatement	::=	**refix** *expression* **at** *expression*
unfixStatement	::=	**unfix** *expression*
moveStatement	::=	**move** *expression* **to** *expression*
compoundStatement	::=	**begin** *blockBody* **end**
checkpoint	::=	**confirm**
	|	**confirm checkpoint**
	|	**checkpoint**
checkpointStatement	::=	*checkpoint*
	|	*checkpoint* **to** *expression*
	|	*checkpoint* **at** *expression*
	|	*checkpoint* **at all**
returnStatement	::=	**return**
returnAndFailStatement	::=	**returnandfail**
primitiveImplementation	::=	**integer**
	|	**string**
primitiveStatement	::=	**primitive** *primitiveImplementation* “[” *symbolReferenceSopt* ‘
		“ ←” “[” *symbolReferenceSopt* “]”
symbolReferenceSopt	::=	*empty*
	|	*symbolReferenceS*
waitStatement	::=	**wait** *expression*
signalStatement	::=	**signal** *expression*
expressionS	::=	*expression*
	|	*expressionS* “,” *expression*

expression	::=	*expressionZero*
	|	*expression* “|” *expression*
	|	*expression* **or** *expression*
	|	*expression* “&” *expression*
	|	*expression* **and** *expression*
	|	“!” *expression*
	|	*expression* “ ==” *expression*
	|	*expression* “! ==” *expression*
	|	*expression* “ =” *expression*
	|	*expression* “! =” *expression*
	|	*expression* “ >” *expression*
	|	*expression* “ <” *expression*
	|	*expression* “ >=” *expression*
	|	*expression* “ <=” *expression*
	|	*expression* “∘>” *expression*
	|	**view** *expression* **as** *expression*
	|	**restrict** *expression* **to** *expression*
	|	*expression* “+” *expression*
	|	*expression* “−” *expression*
	|	*expression* “∗” *expression*
	|	*expression* “/” *expression*
	|	*expression* “#” *expression*
	|	*expression* **operator** *expression*
	|	“ ” *expression*
	|	**isfixed** *expression*
	|	**locate** *expression*
	|	**awaiting** *expression*
expressionZero	::=	*invocation*
	|	*expressionZero* “$” **identifier**
	|	*expressionZero* “(” *argumentS* “)”
invocation	::=	*primary*
	|	*identifierOperationNameReference* “[” “]”
	|	*identifierOperationNameReference* “[” *argumentS* “]”
	|	*expressionZero* “.” *operationNameReference argumentClause*
primary	::=	*literal*
	|	*symbolReference*
	|	“(” *expression* “)”
operationNameDefinition	::=	*operationName*
	|	*operatorName*
	|	*definableOperatorName*
operatorName	::=	**operator**
operationName	::=	**identifier**

```
definableOperatorName              ::=   “|”
                                      |  “&”
                                      |  “ =”
                                      |  “! =”
                                      |  “ >”
                                      |  “ <”
                                      |  “ >=”
                                      |  “ <=”
                                      |  “ ”
                                      |  “!”
                                      |  “+”
                                      |  “−”
                                      |  “*”
                                      |  “/”
                                      |  “#”
nondefinableOperatorName           ::=   “ ==”
                                      |  “! ==”
                                      |  “◦>”
operationNameReference             ::=   operatorName
                                      |  definableOperatorName
                                      |  nondefinableOperatorName
                                      |  operationName
                                      |  nondefinableOperationName
identifierOperationNameReference   ::=   operationName
                                      |  nondefinableOperationName
nondefinableOperationName          ::=   owntype
                                      |  ownname
argumentClause                     ::=   empty
                                      |  “[” “]”
                                      |  “[” argumentS “]”
argumentS                          ::=   argument
                                      |  argumentS “,” argument
argument                           ::=   expression
                                      |  move expression
                                      |  visit expression
literal                            ::=   string
                                      |  character
                                      |  integer
                                      |  real
                                      |  true
                                      |  false
                                      |  self
                                      |  nil
                                      |  builtinType
                                      |  typeLiteral
                                      |  vectorLiteral
```

vectorLiteral	::=	"{" *expressionSOpt* *typeClauseOpt* "}"
expressionSOpt	::=	*empty*
	|	*expressionS*
typeLiteral	::=	*typeRest*
	|	**immutable** *typeRest*
typeRest	::=	*abstractType*
	|	*object*
	|	*record*
	|	*union*
	|	*enumeration*
	|	*class*
symbolReference	::=	**identifier**
symbolDefinition	::=	**identifier**

B Built-in Objects

This appendix defines the built-in objects.

B.1 AbstractType

The AbstractType and Signature objects cooperate to define the concept of type. Signature describes the primitive result of a type constuctor and AbstractType allows other objects to implement it. The Signature object is described later in this appendix.

```
const AbstractType == immutable object AbstractType
    export getSignature
    const AbstractTypeSignature == immutable type AbstractTypeSignature
        function getSignature → [Signature]
    end AbstractTypeSignature

    function getSignature → [result : Signature]
        result ← AbstractTypeSignature
    end getSignature
end AbstractType
```

B.2 Any

Any is the type that requires no operations; every Emerald object has type Any.

```
const Any == immutable object Any
    export getSignature

    const AnyType == type AnyType
        % no operations
    end AnyType

    function getSignature → [result : Signature]
        result ← AnyType
    end getSignature
end Any
```

B.3 Array

Arrays implement expandable indexable storage. The of operation on Array takes an AbstractType, and returns an array creator. As arrays can expand and shrink, common data types such as Stacks and Queues can be implemented using Arrays: Stacks use addUpper and removeUpper, while Queues use AddUpper and RemoveLower.

```
const Array ==
   immutable object Array
      export of
      function of[ElementType : AbstractType] → [result : NAT]
         where
            NA == type NA
               function getElement [Integer] → [ElementType]
               operation setElement [Integer, ElementType]
               function upperBound → [Integer]
               function lowerBound → [Integer]
               function getSlice [Integer, Integer] → [NA]
               operation setSlice [Integer, Integer, NA]
               operation slideTo [Integer]
               operation addUpper [ElementType]
               operation removeUpper → [ElementType]
               operation addLower [ElementType]
               operation removeLower → [ElementType]
               function empty → [Boolean]
               operation catenate [a : NA] → [r : NA]
            end NA

            NAT == immutable type NAT
               function getSignature → [Signature]
               operation empty → [NA]
               operation literal [Vector.of[ElementType]] → [NA]
               operation create [Integer] → [NA]
            end NAT
         end where
```

```
        result ← immutable object aNAT
            export getSignature, empty, create, literal
            function getSignature → [result : Signature]
                result ← NA
            end getSignature
            operation create [length : Integer] → [result : NA]
                result ←
                    object aNA
                        export
                            getElement, setElement, upb, lwb, getSlice, setSlice,
                            slideTo, addUpper, removeUpper, addLower,
                            removeLower, empty, catenate
                        ⋮
                    end aNA
            end create
            operation empty → [result : NA]
                ⋮
            end empty
            operation literal [v : Vector.of[ElementType]] → [result : NA]
                ⋮
            end literal
        end aNAT
    end of
end Array
```

B.4 Boolean

In addition to the operations on Booleans listed here, Booleans are involved in the evaluation of the conditional and (**and**) and conditional or (**or**) expressions. These conditional operations cannot be described in terms of operations on Booleans, since that would imply evaluation of the arguments to the operations, which is exactly what the conditional expressions wish to avoid.

```
const Boolean == immutable object Boolean
   export getSignature, makeTrue, makeFalse
   const BooleanType == immutable type BooleanType
      function   >    [Boolean] → [Boolean]
      function   >=   [Boolean] → [Boolean]
      function   <    [Boolean] → [Boolean]
      function   <=   [Boolean] → [Boolean]
      function   =    [Boolean] → [Boolean]
      function   ! =  [Boolean] → [Boolean]
      function   &    [Boolean] → [Boolean]
      function   |    [Boolean] → [Boolean]
      function   !    → [Boolean]
      function asString → [String]
   end BooleanType

   function getSignature → [result : Signature]
      result ← BooleanType
   end getSignature

   function makeTrue → [result : Boolean]
      result ← immutable object aTrueBoolean
         export >, >=, <, <=, =, ! =, &, |, !
         ⋮
      end aTrueBoolean
   end create

   function makeFalse → [result : Boolean]
      result ← immutable object aFalseBoolean
         export >, >=, <, <=, =, ! =, &, |, !
         ⋮
      end aFalseBoolean
   end makeFalse
end Boolean
```

B.5 Character

```
const Character == immutable object Character
    export getSignature, create
    const CharacterType == immutable type CharacterType
        function ord → [Integer]
        function  >   [Character] → [Boolean]
        function  >=  [Character] → [Boolean]
        function  <   [Character] → [Boolean]
        function  <=  [Character] → [Boolean]
        function  =   [Character] → [Boolean]
        function  !=  [Character] → [Boolean]
        function asString → [String]
    end CharacterType

    function getSignature → [result : Signature]
        result ← CharacterType
    end getSignature

    function create [Integer] → [result : Character]
        result ← immutable object aCharacter
            export ord, >, >=, <, <=, =, !=, asString
            ⋮
        end aCharacter
    end create
end Character
```

B.6 Condition

The formulation of Hoare condition variables in terms of objects is not very satisfying. The condition object may only be used within the monitor within which it is used the first time, regardless of where the condition was created. Although this relationship is impossible to describe using object semantics, Emerald conditions do work properly.

```
const Condition == immutable object Condition
    export create, getSignature

    const ConditionType == type ConditionType
    end ConditionType

    function getSignature → [result : Signature]
        result ← ConditionType
    end getSignature
    operation create → [result : Condition]
      result ← object aCondition
      end aCondition
    end create
  end Condition
```

Objects whose type is Condition have no operations. Wait, signal, and awaiting are language primitives, not operations on conditions. This might be worth changing.

B.7 InStream

InStream objects provide the ability to read files. The InStream object is immutable and has the following interface:

```
  const InStreamType == type InStreamType
    operation getChar → [Character]
    operation unGetChar [Character]
    operation getString → [String]
    function eos → [Boolean]
    operation close
  end InStreamType
```

B.8 Integer

Conversion between integers and reals is accomplished by the asReal operation on Integers, and the asInteger operation on Reals. Viewing an integer as a real does not work (since the type Real can only be implemented by Reals), nor does it do conversions, since view changes the type of an expression without changing its representation, while value conversion must change the representation as well as the type.

```
const Integer == immutable object Integer
    export getSignature, create
    const IntegerType == immutable type IntegerType
        function   +    [Integer] → [Integer]
        function   −    [Integer] → [Integer]
        function   ∗    [Integer] → [Integer]
        function   /    [Integer] → [Integer]
        function   #    [Integer] → [Integer]
        function   >    [Integer] → [Boolean]
        function   >=   [Integer] → [Boolean]
        function   <    [Integer] → [Boolean]
        function   <=   [Integer] → [Boolean]
        function   =    [Integer] → [Boolean]
        function   ! =  [Integer] → [Boolean]
        function   ˜    [Integer] → [Integer]
        function asString → [String]
        function asReal → [Real]
    end IntegerType

    function getSignature → [result : Signature]
        result ← IntegerType
    end getSignature

    function create [rep : String] → [result : Integer]
        result ← immutable object anInteger
            export +, −, ∗, /, #, >, >=, <, <=, =, ! =, ˜, asString, asReal
            ⋮

        end anInteger
    end create
end Integer
```

B.9 Node, NodeList, NodeListElement

A number of miscellaneous operations are implemented by Nodes. The nodeEventHandler entries allow appropriate operations to be invoked when the node detects changes in the network topology. The operations that query network topology use the auxiliary types NodeList and NodeListElement which are described below. The object Node is immutable and has the following interface:

```
    const Node == immutable object Node
        export getSignature, create, getStdIn, getStdout
        const NodeType == type NodeType
            operation getActiveNodes → [NodeList]
            operation getAllNodes → [NodeList]
            operation getNodeInformation → [NodeListElement]
            operation getTimeOfDay → [Time]
            operation delay [Time]
            operation waitUntil [Time]
            operation getLoadAverage → [Real]
            operation setNodeEventHandler [HandlerType]
            operation removeNodeEventHandler [HandlerType]
            operation getStdin → [InStream]
            operation getStdout → [OutStream]
            function getLNN → [Integer]
        function getName → [String]
    end NodeType
    function getSignature → [result : Signature]
        result ← NodeType
    end getSignature
    function create → [result : Node]
        result ← immutable object aNode
            export
                getActiveNodes, getAllNodes, getNodeInformation,
                getTimeOfDay, delay, waitUntil, getLoadAverage, setNodeEventHandler,
                removeNodeEventHandler, getStdin, getStdout, getLNN
            ⋮
        end aNode
    end create
    operation getStdin → [result : InStream]
        ⋮
    end getStdin
    operation getStdout → [result : OutStream]
        ⋮
    end getStdout
end Node
```

```
const NodeListElement == immutable type NodeListElement
    function getTheNode → [Node]
    function getUp → [Boolean]
    function getIncarnationTime → [Time]
    function getLNN → [Integer]
end NodeListElement

const NodeList == ImmutableVector.of[NodeListElement]

const HandlerType == type HandlerType
    operation nodeUp [Node, Time]
    operation nodeDown [Node, Time]
end HandlerType
```

B.10 None

None is the type that supports all operations, and is therefore implemented only by the **nil** object. It is defined to complete the lattice structure of Emerald types; None represents the top element of the type lattice.

B.11 OutStream

The OutStream object represents the output stream of a given Node (explained in this appendix) object.
Objects with type OutStream have the following interface:

```
const OutStreamType == type OutStreamType
    operation putChar [Character]
    operation putInt [n : Integer, width : Integer]
    operation putReal [Real]
    operation putString [String]
    operation flush
    operation close
end OutStreamType
```

B.12 Real

The Real type is implemented as a 32-bit floating-point number.

```
const Real == immutable object Real
    export getSignature, create
    const RealType == immutable type RealType
        function    +         [Real] → [Real]
        function    −         [Real] → [Real]
        function    ∗         [Real] → [Real]
        function    /         [Real] → [Real]
        function    >         [Real] → [Boolean]
        function    >=        [Real] → [Boolean]
        function    <         [Real] → [Boolean]
        function    <=        [Real] → [Boolean]
        function    =         [Real] → [Boolean]
        function    ! =       [Real] → [Boolean]
        function    ˜                → [Real]
        function    −                → [Real]
        function asString [Real] → [String]
        functionasInteger[Real] → [Integer]
    end RealType

    function getSignature → [result : Signature]
        result ← RealType
    end getSignature

    function create [rep : String] → [result : Real]
        result ← immutable object aReal
                                export +, −, ∗, /, >, >=, <, <=, =, ! =, ˜, asString, asInteger
                                ⋮

        end aReal
    end create
end Real
```

B.13 Signature

The Signature object, as explained in the earlier subsection on AbstractType, permits the complete definition of types in Emerald.

```
const Signature == immutable object Signature
    export getSignature, create
    const SignatureType == immutable type SignatureType
      function getSignature → [Signature]
    end SignatureType

    function getSignature → [result : Signature]
      result ← SignatureType
    end getSignature

    function create → [result : SignatureType]
      result ← immutable object aSignature
        export getSignature
        ⋮
      end aSignature
    end create
  end Signature
```

B.14 String

The || operation on strings returns a new string which is the catenation of the two argument strings. The getSlice operation returns a new string, a substring of the original starting at the given index (0 origin) and with the given length.

```
  const String == immutable object String
    export getSignature, create

    const StringType == immutable type StringType
      function getElement [Integer] → [Character]
      function getSlice [lb : Integer, length : Integer] → [String]
      function length → [Integer]
      function   ||   [String] → [String]
      function   >    [String] → [Boolean]
      function   >=   [String] → [Boolean]
      function   <    [String] → [Boolean]
      function   <=   [String] → [Boolean]
      function   =    [String] → [Boolean]
      function   !=   [String] → [Boolean]
    end StringType

    function getSignature → [result : Signature]
      result ← StringType
    end getSignature
    function create [v : Vector.of[Character]] → [result : String]
        result ← immutable object aString
            export getElement, getSlice, ||, >, >=, <, <=, =, ! =
            ⋮
        end aString
    end create
end String
```

B.15 Time

Times represent times and dates. They are stored as a number of seconds (since Jan 1, 1970) and a number of microseconds. They can be used as either dates or times, and the standard arithmetic operations are defined on them (where they make sense).

```
const Time == immutable object Time
    export getSignature, create

    const TimeType == immutable type TimeType
        function   +    [Time] → [Time]
        function   −    [Time] → [Time]
        function   ∗    [Integer] → [Time]
        function   /    [Integer] → [Time]
        function   >    [Time] → [Boolean]
        function   >=   [Time] → [Boolean]
        function   <    [Time] → [Boolean]
        function   <=   [Time] → [Boolean]
        function   =    [Time] → [Boolean]
        function   ! =  [Time] → [Boolean]
        function getSeconds → [Integer]
        function getMicroSeconds → [Integer]
        function asString → [String]
        function asDate → [String]
    end TimeType

    function getSignature → [result : Signature]
        result ← TimeType
    end getSignature
    function create [rep : Any] → [result : Time]
      result ← immutable object aTime
        export
          +, −, ∗, /, >, >=, <, <=, =, ! =,
          getSeconds, getMicroSeconds, asDate, asString
        ⋮
      end aTime
    end create
  end Time
```

B.16 Vector

There are two flavors of vectors: mutable and immutable. Vector is the name of an object with a polymorphic operation of that creates mutable vectors. ImmutableVector is the name of the corresponding object that creates immutable vectors (see B.17. Vectors provide the most primitive mechanism for acquiring indexable storage. The builtin object Array is implemented entirely in Emerald, using the facilities offered by Vector. The object Vector is immutable and has the following interface:

```
  const Vector ==
    immutable object Vector
      export of
      function of [ElementType : AbstractType] → [result : NVT]
        where
          NV ==
            type NV
              function getElement [Integer] → [ElementType]
              operation setElement [Integer, ElementType]
              function upperbound → [Integer]
              function lowerbound → [Integer]
              function getSlice [Integer, Integer] → [NV]
            end NV
          NVT ==
            immutable type NVT
              operation create [Integer] → [NV]
              function getSignature → [Signature]
            end NVT
        end where

        result ←
          immutable object aNVT
            export create, getSignature
```

```
                function getSignature → [result : Signature]
                    result ← NV
                end getSignature
                operation create [length : Integer] → [result : NV]
                    result ←
                        object aNV
                            export getElement, setElement, upperbound, lowerbound, getSlice
                            ⋮
                        end aNV
                end create
            end aNVT
        end of
    end Vector


const ImmutableVector ==
    immutable object ImmutableVector
        export of
        function of [ElementType : AbstractType] → [result : NIVT]
            where
                NIV ==
                    type NIV
                        function getElement [Integer] → [ElementType]
                        function upperbound → [Integer]
                        function lowerbound → [Integer]
                        function getSlice [Integer, Integer] → [NIV]
                    end NIV
                NIVT ==
                    immutable type NIVT
                        operation create [Integer] → [NIV]
                        function getSignature → [Signature]
                    end NIVT
            end where
            result ←
                immutable object aNIVT
                    export create, getSignature
                    function getSignature → [result : Signature]
                        result ← NIV
                    end getSignature
            operation create [length : Integer] → [result : NIV]
              result ←
                object aNIV
                  export getElement, upperbound, lowerbound, getSlice
                  ⋮
                end aNIV
            end create
          end aNIVT
      end of
    end ImmutableVector
```

B.17 ImmutableVector

ImmutableVector is the name of an object that creates immutable vectors. Vector is the name of the corresponding object that creates mutable vectors (see B.16. The primary function of ImmutableVectors is to implement the Vector literals in the language, but they are also available for use whenever read-only vectors are necessary. The object ImmutableVector is immutable and has the following interface:

```
    function of[T : Type] → [NVT]
      forall T
```

The object resulting from *ImmutableVector.of*[*T*] is immutable and has the following interface (herein abbreviated NVT):

```
    function getSignature → [Signature]
    operation create[Integer] → [NV]
```

Objects with type *ImmutableVector.of*[*T*] have the following interface (herein abbreviated NV):

```
    function getElement[Integer] → [T]
    function upperbound → [Integer]
    function lowerbound → [Integer]
    operation catenate [NV] → [NV]
    function getSlice[lb : Integer, length: Integer] → [NV]
```

B.18 BitChunk

BitChunks allow the manipulation of arbitrarily sized sequences of bits. There are operations to set or retrieve collections of bits at arbitrary bit positions with lengths up to 32 bits. BitChunk is an immutable object with the following interface:

export function *getSignature* → [*Signature*]
export operation *create*[*n* : *Integer*] → [*Bitchunk*]
% Create a bitChunk large enough to hold n bytes of information.
% This really should be measured in bits, but that proves problematic at
% this point.

An object whose type is BitChunk has the following interface:

function *getSigned*[*off*: *Integer*, *len*: *Integer*] → [*Integer*]
% return the bits at offset off for length len as a signed Integer
% (treat the highest order bit as a sign bit)
function *getUnsigned*[*Integer*, *Integer*] → [*Integer*]
% return the bits at offset off for length len as an unsigned Integer
function *getElement*[*Integer*, *Integer*] → [*Integer*]
% equivalent to getUnsigned
operation *setSigned*[*off*:*Integer*, *len*:*Integer*, *val*:*Integer*]
% set the bits at offset off for length len to the low order bits of val.
operation *setUnsigned*[*Integer*, *Integer*, *Integer*]
% equivalent to setSigned
operation *setElement*[*Integer*, *Integer*, *Integer*]
% equivalent to setSigned
operation *ntoh*[*Integer*, *Integer*]
% Convert the bits at offset off with length len from network to host
% byte order. len must be either 16 or 32.

B.19 VectorOfChar

A VectorOfChar is merely a Vector.of[Character].

B.20 Unix

This has been thought about, but is not currently a builtin. The idea was that it would give you access to other Unix facilities not currently available to Emerald programs. These could include file system operations, command execution, etc.

B.21 RISC

The literal operation on String requires a readable indexed sequence of characters, so we invented RISC. It is just what you want. Note that Vector.of[Character], ImmutableVector.of[Character], and Array.of[Character] all conform to RISC.

The object RISC is immutable and has just one operation:

function *getSignature* → [*Signature*]

Objects whose type is RISC have the following interface:

function $lowerbound \rightarrow [Integer]$
function $upperbound \rightarrow [Integer]$
function $getElement[Integer] \rightarrow [Character]$

C Compiling Emerald Programs

This appendix presents a sample Emerald program, and shows how Emerald programs are compiled and executed. More details about practical programming in Emerald may be found in the Emerald System User's Guide [Jul 88c], and examples of the Emerald style of programming in [Raj 88].

The Emerald run-time system runs on top of Unix, with the Emerald compiler cross-compiling Emerald programs from Unix to the Emerald world. Source Emerald programs are created as Unix text-files, and the Emerald compiler translates the program into a set of executable code files for use by the Emerald run-time system. To permit the sharing object definitions between different compilations, a special mechanism is used to export objects being defined in one compilation to the Emerald environment, and import them back into another compilation. Each Emerald compilation is defined as follows:

```
compilation            ::=  environmentImports
                       ::=  environmentExports
                       ::=  constantDeclarations
environmentImports     ::=  [ "["environmentImport { "," environmentImport }"]" ]
environmentExports     ::=  [ "["environmentExport { "," environmentExport }"]" ]
environmentImport      ::=  import symbolDefinitions from environmentPathName   |
environmentExport      ::=  export symbolDefinitions to environmentPathName     |
environmentPathName    ::=  stringLiteral
symbolDefinitions      ::=  "[" symbolDefinition { "," symbolDefinition } "]"
symbolDefinition       ::=  identifier
```

The environment mechanism permits the sharing of object definitions between compilations. Object (actually, identifiers) exported from one compilation unit that are to be used in another must be exported to the environment in order to be visible to later compilation units. An Emerald compilation unit is defined as follows:

For example, if an object *Directory* is being defined in compilation, and needs to be used by other objects (being compiled later), a statement:

export *Directory* **to** *"MyStuff"*

is needed. In later compilations, *Directory* may be used by imported by using:

import *Directory* **from** *"MyStuff"*

These clauses must not be confused with the **export** statement within an object constructor that permits the given object to accept invocations to the exported operations. At present, only symbols representing completely manifest object definitions may be exported and imported from the environment path.

C.1 A Sample Program

This simple example involves an integer stack creator (*Stack*), which is compiled first, and a test object (*Tester*), which is compiled later and makes use of the *Stack*. The two programs are entered into separate Unix files.

The file `stack.m`

This Emerald program file defines the stack-creator (see Figure 5); once created, this object accepts *create* invocations and returns a new stack-like object (conforming to *Stacktype*) on each such invocation. See [Hutchinson 87a, Raj 91] for a better understanding of this program. Note that the name Stack is exported to the compilation environment using the export directive.

The file `tester.m`

This file (see Figure 6) contains a simple object that can be used to test the *Stack* object defined in Figure 5. This object invokes the stack-creator object *Stack* to create the new stack named *myStack*; the rest of the program is fairly straight-forward. The name *Stack* in unbound in this example, therefore this program must be compiled in a compilation environment that includes the name *Stack*. Also note the predefined identifier *stdin* and *stdout* which name the (already opened) standard input and output streams respectively.

C.2 Compiling the Program

The files are compiled using the Emerald compiler, `ec`, by executing the following Ultrix commands:

```
ec -C stack.m
ec -C tester.m
```

where the `-C` switch forces the compiler to stop after creating the objects and placing their object-ids in corresponding `.g` files.

When the first file is compiled, the *Stack* object definition is exported to the Emerald environment path “Junk”; you can set up suitable environments by using different names here for different objects. The *Stack* definition can now be used in any subsequent compilations on the same node. Note that recompilation of the file `stack.m` results in the name *Stack* naming a new object with a new object-id; this means that recompilation should used cautiously because exportation/importation of symbols introduces compilation dependencies.

```
% This file defines the Stack object, which implements the single operation "create"
% that returns an object conforming to type Stacktype.

export Stack to "Junk"

const Stack ==
  immutable object iStack
    export getSignature, create

    const StackType ==
      type iStackType
        operation Push[n: Integer]
        operation Pop → [n: Integer]
        function Empty → [result : Boolean]
      end iStackType

    function getSignature → [r : Signature]
      r ← StackType
    end getSignature

    operation create → [result : StackType]
      result ←
        object aStack
          export Push, Pop, Empty
          monitor
            var store: Array.of[Integer] ← Array.of[Integer].create[0]

            operation Push[n: Integer]
              store.addUpper[n]
            end Push

            operation Pop → [n: Integer]
              n ← store.removeUpper
            end Pop

            function Empty → [result : Boolean]
              result ← store.empty
            end Empty
          end monitor
        end aStack
    end create

end iStack
```

Figure 5: The file `stack.m`

C.3 Executing the Program

To start up an Emerald object on a given node, we have to ensure that the Emerald kernel is running on that node. We shall simply assume that the Emerald kernel is up and running

```
% This program first creates the stack named myStack by invoking Stack.
% It pushes 4 integers into myStack, and then pops and prints them.
import Stack from "Junk"
const Tester == object Tester
  process
    var i: Integer ← 0
    const myStack: Stack == Stack.create
    stdout.PutString["Playing looney tunes now!\^J"]
    for (i ← 0 : i < 4 : i ← i+1)
      stdout.PutString["Pushing " || i.asString || " on my Stack.\^J"]
      myStack.Push[i]
    end for
    stdout.PutString["Printing in Reverse Order.\^J"]
    loop
      var x: Integer
      exit when myStack.Empty
      x ← myStack.Pop
      stdout.PutString["Popped " || x.asString || " from my Stack.\^J"]
    end loop
    stdout.PutString["That's all for now, folks!\^J"]
    stdout.close
    stdin.close
  end process
end Tester
```

Figure 6: The file `tester.m`

(details about this are provided in [Jul 88c]). The Ultrix command

```
runec -i tester.g
```

starts up the object, and the `i` switch makes the `stdin/stdout` input and output streams of the object the same as that of the Ultrix shell. On executing this command, we get the following output:

```
Playing looney tunes now!
Pushing 0 on my Stack.
Pushing 1 on my Stack.
Pushing 2 on my Stack.
Pushing 3 on my Stack.
```

```
Printing in Reverse Order.
Popped 3 from my Stack.
Popped 2 from my Stack.
Popped 1 from my Stack.
Popped 0 from my Stack.
That's all for now, folks!
```

This has been a brief introduction to the Emerald compilation and run-time environment; more details about using the Emerald system can be found in [Jul 88c].

References

[Almes 85] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The Eden System: A Technical Review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59, January 1985.

[Black 85a] Andrew P. Black. Supporting Distributed Applications: Experience with Eden. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 181–93, December 1985.

[Black 85b] Andrew P. Black. *The Eden Programming Language.* Technical Report 85-09-01, Department of Computer Science, University of Washington, Seattle, September 1985.

[Black 86] Andrew P. Black, Norman C. Hutchinson, Eric Jul, and Henry M. Levy. Object Structure in the Emerald System. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 78–86, October 1986.

[Black 87] Andrew P. Black, Norman C. Hutchinson, Eric Jul, Henry M. Levy, and Larry Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, 13(1), January 1987.

[Goldberg 83] Adele Goldberg and David Robson. *Smalltalk-80: The Language And Its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[Hutchinson 87a] Norman C. Hutchinson. *Emerald, An Object-Based Language for Distributed Programming.* PhD thesis, Department of Computer Science, University of Washington, Seattle, WA 98195, January 1987.

[Hutchinson 87b] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming.* PhD thesis, TR 87-01-01, Department of Computer Science, University of Washington, Seattle, January 1987.

[Jul 88a] Eric Jul. *Object Mobility in a Distributed Object-Oriented System.* PhD thesis, TR 88-12-06, Department of Computer Science, University of Washington, Seattle, December 1988.

[Jul 88b] Eric Jul, Henry M. Levy, Norman C. Hutchinson, and Andrew P. Black. Fine-grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[Jul 88c] Eric Jul, Rajendra K. Raj, and Norman Hutchinson. The Emerald System: User's Guide. July 1988. Emerald Project Internal Memorandum (Revised November 1988).

[Liskov 84] Barbara Liskov. *Overview of the Argus Language and System.* Programming Methodology Group Memo 40, M.I.T. Laboratory for Computer Science, February 1984.

[Raj 88] Rajendra K. Raj, Ewan D. Tempero, Henry M. Levy, Norman C. Hutchinson, and Andrew P. Black. The Emerald Approach to Programming. Technical Report 88-11-01, Department of Computer Science, University of Washington, Seattle, November 1988. Revised February 1989.

[Raj 91] Rajendra K. Raj, Ewan D. Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: a general-purpose programming language. *Software—Practice and Experience*, 21(1):91–118, January 1991.