An Emerald Primer

Norman C. Hutchinson

Technical Report 96-??
September 1996

Department of Computer Science
University of British Columbia
Vancouver BC, Canada V6T 1Z4

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 a distributed programming environment. At the same time, Emerald is a modern programming language providing excellent features for abstractions and polymorphism. Primarily a language for distributed environments, Emerald includes features for dealing with the location of objects, and extends exception handling mechanisms to recovering from partial failures of distributed systems. This primer provides a gentle introduction to the language.

1 Introduction

The primary goal of Emerald [BHJL86, BHJ+87, JLHB88] 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. Emerald draws heavily upon the experience gained from Smalltalk [GR83], the Argus Language and System [Lis84] and, in particular, the Eden system [ABLN85, Bla85a] and the Eden Programming Language (EPL) [Bla85b].

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 [GR83], i.e., all data items are objects with a uniform semantic model for operations on them. 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; the semantics of Emerald are complicated both because location is apparent and because systems may be partially unavailable.

This document is intended as a gentle introduction to the Emerald programming language and system. We first present an overview of the language features, and through a few examples document their use. The Emerald approach to programming is discussed in [RTL+88], where several examples of Emerald programs may be found. The definitive description of the Emerald programming language is found in the language report [HRB+87]

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. In Emerald, everything from small entities (e.g., Booleans and integers) to large entities (e.g., directories, compilers, and entire file systems) are represented using objects. An object exists as long as it can be named or referenced.

Each Emerald object consists of:

- A *name* or *identity*, which uniquely identifies the object within the network. This name is not normally visible to the programmer.
- 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, objects can be declared to be monitored. Processes synchronize through system-defined *condition* objects.

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. Objects may also have a *recovery* section which is used to re-initialize the object after the node it was executing on fails and subsequently recovers.

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 objects.

1.3 Abstract types

Central to Emerald are the concepts of *type*[1] and *type conformity*. All identifiers in Emerald are typed, and except for constant identifiers, 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 : Mailbox*

then any object that is assigned to MyMailbox must implement (at least) the operations defined by the type Mailbox, which may be defined as:

```
const MailBox ← typeobject MailBox
    operation deliver[MailMessage]
    function list → [Sequence.of[MailMessage]]
    operation remove[MailMessage]
end MailBox
```

On every assignment, whether explicitly specified in an assignment statement, or implicitly specified through the arguments and results of invocations, 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 $S \circ\!\!> T$) 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*.

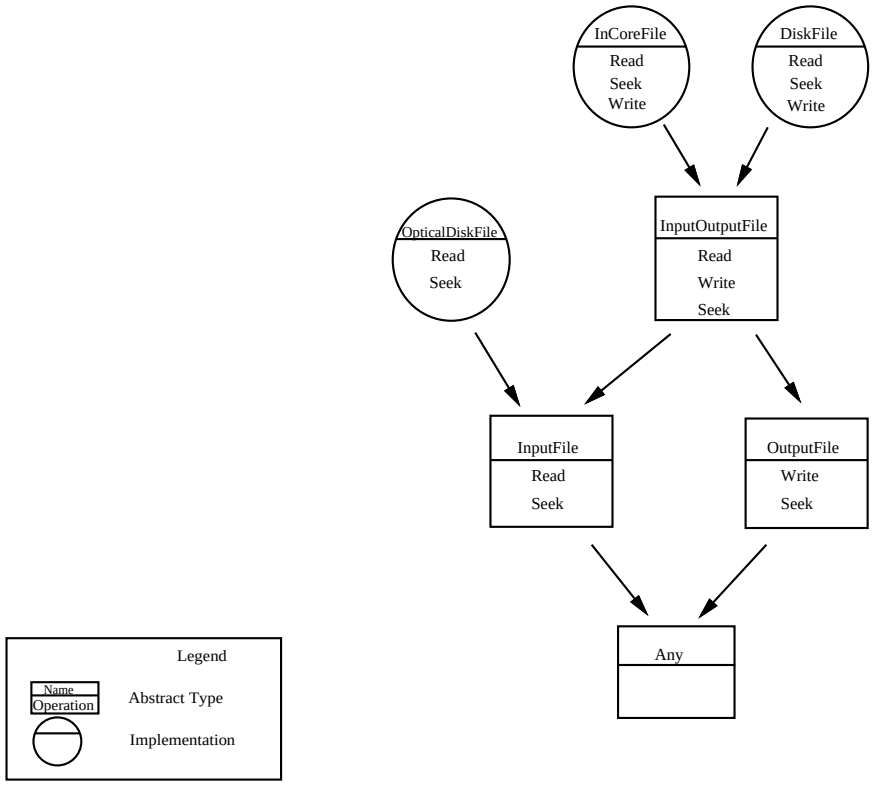


Figure 1: Example types and object implementations

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 type A appearing above a type B means that 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 after a crash.

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 visible operations in the object is increased, or a *narrowing*, i.e., the number of visible operations is reduced. Widening corresponds to a move up in the type partial order and narrowing to a move down. Narrowing is always correct and requires no run-time check, 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.

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 ← typeobject Directory
    operation Insert [name : String, thing : Any]
    function Lookup [name : String] → [thing : Any]
    operation Delete [name : String]
    function List  → [names : ImmutableVectorOfString]
end Directory
```

with additional variables declared as

```
var f : InputOutputFile
var g : OutputFile
```

If *f* currently names a file object, and the file is to be inserted into a directory *d*, the following invocation suffices:

```
d.Insert["myfile", f]
```

Since the second argument to *Insert* on directories has type **Any**, the type of *f*, *InputOutputFile*, is narrowed to **Any**. When the same object is retrieved from the directory *d*, the assignment:

```
f ← 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 ← view d.Lookup["myfile"] as InputOutputFile
```

may be used. This widening can be established as correct only at run-time.

1.4 Object creation

In most object-based systems, new objects are created by an operation on a *class* (as in Smalltalk or C++) 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*. An object constructor is an Emerald expression that defines the representation, the operations, and the process of an object. For example, the execution of the Emerald program in Figure 2 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 syntax { expressions } used in the implementation of the List operation create a sequence of values (technically, an immutable vector of values).

```
const myDirectory : Directory ← monitor object oneEntryDirectory
    var name : String ← ""
    var value : Any ← nil
    export operation Insert[n : String, o : Any]
        name ← n
        value ← o
    end Insert
    export function Lookup[n : String] → [o : Any]
        if n = name then
            o ← value
        else
            o ← nil
        end if
    end Lookup
    export operation Delete[n : String]
        if n = name then
            name ← ""
            value ← nil
        end if
    end Delete
    export function List → [r : ImmutableVectorOfString]
        if name = "" then
            r ← ImmutableVectorOfString.Literal[{ : String }]
        else
            r ← ImmutableVectorOfString.Literal[{ name }]
        end if
    end List
end oneEntryDirectory
```

Figure 2: A oneEntryDirectory object

1.5 Classes

While Emerald objects do not depend on classes, classes are supported because that style of programming is very convenient. A class in Emerald is transformed into a pair of nested object constructors in the manner hinted at by the previous two examples. More precisely, the declarations and operations of the class become attributes of the instances (that is, they appear in the inner object constructor), while the class object itself contains only the class operations and necessary boilerplate. The following example uses the sieve of Eratosthenes to find all of the prime numbers less than some limit:

```
const myDirectoryCreator ← immutable object oneEntryDirectoryCreator
   export operation Empty → [result : Directory]
      result ← monitor object oneEntryDirectory
         var name : String ← ""
         var value : Any ← nil
         export operation Insert[n : String, o : Any]
            name ← n
            value ← o
         end Insert
         export function Lookup[n : String] → [o : Any]
            if n = name then
               o ← value
            else
               o ← nil
            end if
         end Lookup
         export operation Delete[n : String]
            if n = name then
               name ← ""
               value ← nil
            end if
         end Delete
         export function List → [r : ImmutableVectorOfString]
            if name = "" then
               r ← { }
            else
               r ← { name }
            end if
         end List
      end oneEntryDirectory
   end Empty
end oneEntryDirectoryCreator
```

Figure 3: A oneEntryDirectory creator

```
const Sieve ← class Sieve
    var next : Sieve
    var myprime : Integer
    export operation filter[n : Integer] → [isPrime : Boolean]
        if myprime == nil then
            myprime ← n
            isPrime ← true
        elseif n # myprime = 0 then
            isPrime ← false
        else
            if next == nil then next ← Sieve.create end if
            isPrime ← next.filter[n]
        end if
    end filter
end Sieve

const driver ← object driver
    initially
        const s ← Sieve.create
        const limit ← 100
        for i : Integer ← 2 while i < limit by i ← i + 1
            if s.filter[i] then
                stdout.putint[i, 0]
                stdout.putchar['\n']
            end if
        end for
    end initially
end driver
```

Class objects serve two purposes: to create instances of the class and to serve as the type of instances of the class. In this simple example the create operation takes no parameters. The single operation filter on sieve objects accepts the first number that a sieve element sees, reports as composite any multiple of that number, and passes along any other number to its successor sieve element, which it creates only when necessary.

1.6 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: 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. Besides location independent invocation, the primary distribution related language primitives are the move and fix statements. Here is a simple example of the use of move:

```
const Kilroy ← object Kilroy
    process
        const home ← locate self
        var there : Node
        var startTime, diff : Time
        var all : NodeList
        var theElem :NodeListElement
        home$stdout.PutString["Starting on " || home$name || "\n"]
        all ← home.getActiveNodes
        home$stdout.PutString[(all.upperbound + 1).asString || " nodes active.\n"]
        startTime ← home.getTimeOfDay
        for i : Integer ← 1 while i <= all.upperbound by i ← i + 1
            there ← all[i]$theNode
            move Kilroy to there
            there$stdout.PutString["Kilroy was here\n"]
        end for
        move Kilroy to home
        diff ← home.getTimeOfDay - startTime
        home$stdout.PutString["Back home. Total time = " || diff.asString || "\n"]
    end process
end Kilroy
```

This example requires a few words of explanation. Emerald machines are represented by objects of type *Node* which export an operation *getActiveNodes* which returns a sequence of node descriptions, one for each currently executing Emerald machine. The first entry in this sequence is always the current machine. Kilroy moves himself to each of the nodes in the sequence, printing a short message on the "console" of each machine until he returns back home where he reports how long the tour took.

2 Compiling Emerald Programs

The implementation of Emerald that is available to you uses an interpreter to increase portability. The compiler is written in Emerald, and generates executable files containing byte codes which are then interpreted.

Before you can use Emerald you must perform the following steps:

1. In the UBC Computer Science department, you must the enable command to enable Emerald: enable emerald. In most other installations Emerald should be installed in /usr/local/bin, and nothing special should be necessary to enable it.

2. Create an empty file in your home directory called .emenv

3. Set the environment variable EMPLANE to a unique integer value (different from that used by every other Emerald user). If you run no more than one Emerald interpreter on each physical machine that you use, then the numeric value of your userid should work. This should be done in your .cshrc, .login, or .profile file. Csh users can probably get away with:

   setenv EMPLANE ‘id -u‘

   and sh users:

   EMPLANE=‘id -u‘ ; export EMPLANE

2.1 The compiler: ec

The compiler is named ec, and in it's most simple form is invoked as:

ec filename ...

This compiles the emerald program in the named file (whose name by convention ends with .m), and leaves the resulting executable in a file of the same name with the suffix changed to .x. The only often used argument is:

ec -l environment filename ...

which causes the compilation to be done in the named environment (see section 2.3). A very good thing to do is to create a default compilation environment by creating an empty file in your home directory named `.emenv`. A default compilation environment is searched for in the following way:

1. Use the value of the EMENV environment variable, if set, or

2. use ./.emenv if that file exists, or

3. use ~/.emenv if that file exists.

The default environment can be overridden by explicitly loading a new environment using -l.

2.2 The interpreter: emx

The emerald interpreter is named emx. It is typically invoked as:

emx filename.x ...

Emx supports a large number of flags, the most useful of which are:

-i

Invoke the integrated Emerald debugger when errors occur. Note: In the current version (1.00alpha), the debugger is rather useless.

-I

With -i, invoke the integrated Emerald debugger immediately upon program startup so that initial breakpoints can be set.

-Tcall

Generate a call trace giving the calls and returns of all operations and functions.

-v

Generate summary statistics on program termination.

-R[host[:port]]

Become part of a collection of cooperating Emerald machines by contacting the version of the interpreter running on the given host and listening on the given port (or at the default port if none is specified). If no host is given, then become the first of such a collection of machines.

2.3 The environment

An environment mechanism permits the sharing of object definitions between compilations. Objects (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:

compilation	::=	{ environmentExport | constantDeclaration }
environmentExport	::=	**export** identifier { "," identifier }

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

export *Directory*

is needed. Other source code that uses the identifier *Directory* must be compiled in the context of this export. Only immutable objects (like classes), type objects, and integers are currently handled properly by the environment mechanism. The -l argument to the compiler specifies the environment that it is to use.

2.4 A Sample Program

This section discusses a trivial sample program which is split across two source files. The first source file "stack.m", contains the definition of a stack:

```
% Define an Integer stack creation object
export Stack
const Stack ← immutable object Stack
    const StackType ←
        typeobject StackType
            operation Push[n: Integer]
            operation Pop → [n: Integer]
            function Empty → [result : Boolean]
        end StackType
    export function getSignature → [r : Signature]
        r ← StackType
    end getSignature
    export operation create → [result : StackType]
        result ← object aStack
            const store ← Array.of[Integer].empty
            export operation Push[n: Integer]
                store.addUpper[n]
            end Push
            export operation Pop → [n: Integer]
                n ← store.removeUpper
            end Pop
            export function Empty → [result : Boolean]
                result ← store.empty
            end Empty
        end aStack
    end create
end Stack
```

The second source "stacktest.m" file contains the main program.

```
% This program first creates a stack named myStack by invoking Stack.create.
% It pushes 4 integers into myStack, and then pops and prints them.
const Tester ← object Tester
    process
        const myStack: Stack ← Stack.create
        for i : Integer ← 0 while i < 4 by i ← i + 1
            stdout.PutString["Pushing " || i.asString || " on my stack.\n"]
            myStack.Push[i]
        end for
        stdout.PutString["Printing in reverse order.\n"]
        loop
            var x : Integer
            exit when myStack.Empty
            x ← myStack.Pop
            stdout.PutString["Popped " || x.asString || " from my stack.\n"]
        end loop
    end process
end Tester
```

2.5 Compiling the Program

Compiling the program is as simple as:

```
ec stack.m stacktest.m
```

or, if you want to be able to compile them one at a time, you must indicate an environment:

```
ec -l stackenv stack.m
ec -l stackenv stacktest.m
```

or, if you have already created a default environment, then you may use:

```
ec stack.m
ec stacktest.m
```

Note that the identity of an object is maintained across compilations, so if you change the stack, but have not changed its interface, it is sufficient to recompile just stack.m.

2.6 Executing the Program

The previous steps have generated two emerald executable files, stack.x and stacktest.x. These must be executed together:

```
emx stack.x stacktest.x
```

On executing this command, we get the following output:

```
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.
```

2.7 A Distributed Example

We'll use the kilroy example from before to demonstrate how to get a distributed Emerald program running. Write the code to a file named kilroy.m, compile it using ec kilroy.m, and then you are ready to run it.

Open a window on each of the machines that you are going to use. In the first one, on host1, use the following command to indicate that this version of the interpreter is to be the first of a collection of cooperating interpreters:

```
host1% emx -R
Emerald listening on port 16099
```

In all of the other windows except the last one, use the following command to cause the interpreter to contact the one on host1:

```
hostj% emx -Rhost1
Root directory should be on host1
Emerald listening on port 16099
```

In the last one, you have to give the executable file name (kilroy.x) as well as the distribution argument:

```
hostn% emx -Rhost1 kilroy.x
Root directory should be on host1
Emerald listening on port 16099
Starting on hostn.cs.ubc.ca
3 nodes active.
Back home.  Total time = 0:408319
```

And you should get a message in each of the other windows:

```
Kilroy was here
```

3 Debugging Emerald Programs

Note: In the current version (1.00alpha) the debugger is rather useless. This will be rectified very soon.

The Emerald debugger is built into the interpreter. The interpreter option -i enables the interactive debugger. If execution of the program causes a failure, control will be passed to the debugger. If the -i flag is not given, then a stack dump will be printed and the interpreter will exit taking all of it's interesting information with it.

If you wish to set breakpoints in your program before it executes, give the "-I" option to the interpreter. This will place you in the debugger before the program gets going, just as if a breakpoint were inserted before the first statement. To run the program, use the `continue` command.

The debugger commands are:

where
: Print out a minimal stack back trace, including only the source file names and line numbers of the operations.

dump
: Print out a maximal stack back trace, including all local variables and parameters at each level as well as the instance variables of the target object at each level.

processes
: Print a description of all of the processes that exist.

process *id*
: Switch to debugging the process whose id is given (get it using the processes command).

q

quit
: Exit emx.

c
cont

continue
: Continue the interpreter. If a failure caused entry to the debugger, something strange may happen.

p *expression*

print *expression*
: Print the value of a variable. Variables are searched for in the current activation record and in the currently invoked object, and if not found there, the debugger searches down the call stack.

  You can look inside of objects using the operator ".". If x is an identifier, and x has an instance variable named i, then you can print the object refered to by the instance variable i using "print x.i".

up

down
: Move up (toward older operations) or down (toward more recently called operations) the call stack.

info

look
: Print the current activation record and current object. Just like a one level "dump".

b *filename linenumber*

stop *filename linenumber*
: Set a breakpoint. If the file name is not given the "current" file is assumed. If the linenumber is not given the current line is assumed.

delete *filename linenumber*

Delete a breakpoint. If the file name is not given the "current" file is assumed. If the linenumber is not given the current line is assumed.

n

next

Set a breakpoint at the next source line and continue. This command is completely broken, as it does not deal with control flow that bypasses the next source line. Use this with care - or better yet, fix it.

s

step

Step to the next source line. This command single steps the interpreter until it moves to executing code from a different source line.

S

Step

Step to the next source line in the same source file. This command single steps the interpreter until it moves to executing code from a different source line in the same source file as it is currently in. In most cases, this allows invocations to be stepped across.

bu

Set a breakpoint just after the call to the current operation (break up).

finish

Set a breakpoint just after the call to the current operation (break up) and then continue. This corresponds exactly to a "bu" command followed by a "continue", but it is sufficiently common to deserve its own command.

References

[ABLN85] 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.

[BHJ+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.

[BHJL86] 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.

[Bla85a] 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.

[Bla85b] Andrew P. Black. The Eden Programming Language. Technical Report 85-09-01, Department of Computer Science, University of Washington, Seattle, September 1985.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language And Its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[HRB+87] Norman C. Hutchinson, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. The Emerald programming language report. Technical Report 87-29, Department of Computer Science, University of Arizona, October 1987.

[JLHB88] 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.

[Lis84] Barbara Liskov. Overview of the Argus Language and System. Programming Methodology Group Memo 40, M.I.T. Laboratory for Computer Science, February 1984.

[RTL+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.