# The Eden Programming Language

**Andrew P. Black**

Technical Report 85-09-01

Department of Computer Science, FR-35,
University of Washington,
Seattle, WA 98115
U. S. A.

*September 1985*
*(Revised, December 1985)*

# The Eden Programming Language

Andrew P. Black
University of Washington

*The Eden Programming Language adds significantly to the usability of the Eden distributed system by providing direct support for two of the fundamental concepts of Eden – invocations and capabilities. Invocations, which take the place of inter-object messages, are packaged to look like procedure calls; EPL provides full stub generation for the invoker and invokee, rather like a remote procedure call system. Capabilities, which are used to address other objects, are treated just like any other other opaque data type, despite the fact that they are protected by the Eden kernel; the user does not manipulate "c-list indices". This paper describes the history that lead us to design EPL, its goals and facilities, its implementation on top of Concurrent Euclid, and some of its shortcomings.*

## 1. Introduction

The Eden project is an attempt to combine the advantages of a time-sharing system with those of a distributed collection of personal workstations. One of our goals is to provide the high degree of integration typically found in a time-sharing system, thus making it easy to share both information and computing power. At the same time we have aimed to retain the principal advantages of distributed workstations: insulation from the behaviour of other users and a high bandwidth between the processor and the display.

These goals are typically considered to be "system" requirements, and Eden was originally conceived as an operating system project. However, the Eden Programming Language (EPL) has played a significant rôle in the creation of a programming environment that satisfies them. We now believe that this is inevitable: the model of computation that Eden presents to the application programmer must be supported at the programming language level as well as at the operating system level. As of the autumn 1985, over three hundred thousand lines of EPL code have been written for Eden applications.

This work was supported in part by the National Science Foundation under Grant Number MCS-8004111. Computing Equipment was provided in part under a cooperative research agreement with Digital Equipment Corporation.

Despite this realisation, Eden has remained an operating system project. This did not happen by accident. When we entered the arena of language design, it was with trepidation, and against the advise of our self-appointed review external review committee. We were well aware that language design can become a consuming passion. So much effort can be spent on first getting the language "right", and then implementing it, that there is no time or energy left to build and use the system that it was supposed to support. We therefore restricted our ambitions: a year after our review committee had cautioned us to avoid language design, we were able to report to them that EPL had been designed and implemented, and had already proved valuable in the writing of applications [2].

This paper attempts to explain *why* we believe the rôle of the Eden Programming Language to be so important. It also describes the goals of the Eden Programming Language, the facilities that it provides, and the way in which its translation (compilation) is implemented. Section 2 sets the stage by outlining the basic architectural concepts of Eden; the reader requiring more detail should see "The Eden System: A Technical Review" [3]. Section 3 discusses an early prototype of Eden that was programmed in Pascal; the problems with this approach illustrate the need for programming language support. Section 4 describes EPL itself, and Section 5 the implementation of the EPL translator. Section 6 touches on some of the language design issues that we did not address with EPL. Section 7 concludes by discussing the attributes of our environment that made EPL and its preprocessor-based implementation successful.

## 2. Basic Eden Concepts

The Eden distributed architecture is based on active objects. Eden objects (sometimes called "Ejects") are the only durable entities within the Eden System; they subsume the concepts of program, file and abstract data type. In order to perform some task, an object may need to communicate with other objects; this is accomplished by a mechanism called *invocation*. The model of computation presented by Eden is thus very different from that of most conventional operating systems. For example, the Eden kernel does not implement a filing system. Instead it provides a general mechanism (checkpoint) whereby any object may make its data permanent; using this mechanism, durable sequences of bytes and durable sequences of records can both be implemented by an application programmer.

Each object within the Eden system contains a number of active processes. Invocation is synchronous with respect to the invoking process, but other processes within the invoking object may continue to execute. Invocation is location independent: an object can be invoked without the invoker needing to know its physical location. This is important because objects can move from machine to machine; the Eden kernel itself takes care of locating an object when it is invoked.

Objects address each other by means of *Capabilities*. Capabilities are not addresses; instead, a unique identifier is associated with each object when it is created. This identifier, together with a set of access rights, comprises a Capability for the object. Capabilities may be passed in invocations, but only the system kernel can create them. Possession of a capability for a server object (with appropriate rights) is the only qualification needed of a client before it may invoke that server.

Since Eden objects are the only repositories of data, they must be permanent; for example, they must be able to survive crashes of the system. Permanence is achieved by use of the *checkpoint* mechanism, which allows selected parts of an object's internal state to be written onto stable storage. The data on stable storage is known as the *passive representation* of the object. If the active form of a checkpointed object ceases to exist, either because of a crash or because of explicit deactivation, the passive representation will still contain all of the essential data from that object. If the crashed object is subsequently invoked, a new active form will be created by the Eden kernel; the active form will re-initialise itself by reading the passive representation. The transitions between the various forms of an object are shown in Figure 2.1.

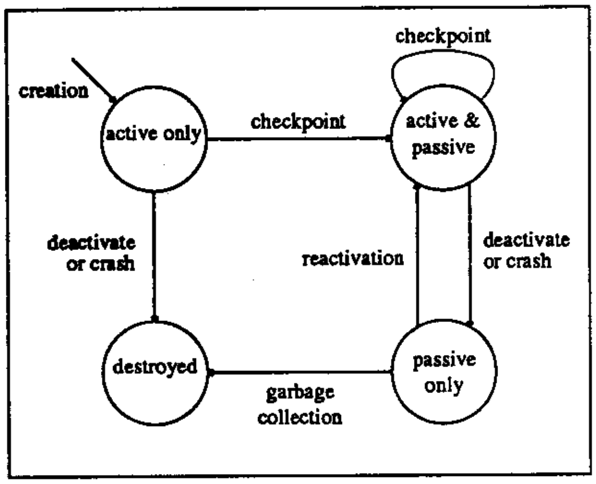

Figure 2.1: The Lifecycle of an Eden Object

It is important that checkpoint be atomic; a checkpoint either succeeds (in which case the data concerned has been written to stable storage) or fails (in which case the object concerned is notified). There is no possibility of a checkpoint succeeding partially. However, checkpoint is the only atomic action provided by Eden. In particular, invocation is not an atomic action. For example, it is quite possible for a client to make a request of a server, for the server to complete the requested operation, and for the server to crash before telling the client that it had done so. In this way Eden differs from the ARGUS system [16] and the CLOUDS [1] system. There is therefore no need for the Eden Programming Language to provide transaction primitives.

The Eden view was that different kinds of transaction could be built by the applications programmer out of checkpoint and invocation. For example, the Eden calendar system [10] and a replicated resource database [18] both include special purpose transaction systems. Because of this decision, the Eden kernel is simplified, and programmers can exploit the semantics of the application to minimise, for example, the number of phases in a transaction commit. However, while the checkpoint primitive is universal (in the sense that more sophisticated reliability mechanisms can be built from it) and conceptually simple, it is far from ideal in practice. Some of its disadvantages are consequences of our prototype, while others are more fundamental. This topic is discussed further in reference [7].

## 3. History and Evolution

As originally designed, the Eden System [15] was intended for implementation on the Intel iAPX 432 processor. One of the advantages of the 432 is that the architecture supports programmer-defined protected object and address spaces. Eden's initial design was thus freed from many of the restrictions that would have accompanied a design for a more conventional processor. However, because of the non-availability of 432 hardware and software, our first "throw-away prototype" of Eden (constructed in the Autumn of 1981) was built on a single VAX processor running the VMS Operating System. This prototype was called "Newark" and had the following characteristics.

Each of the processes internal to an object was implemented as a VAX process, and process creation was achieved by VMS kernel call. Each object accessed several address spaces; there was one address space per process and, in addition, a distinguished representation address space which was intended to contain all of the data that should be checkpointed to stable storage. One advantage of this arrangement was that the checkpoint primitive could be implemented entirely in the kernel; it involved a simple copy of the whole of the representation address space onto the disk.

A serious disadvantage of using VMS processes was the cost of communication between them. For example, if more than one process might update a shared variable in the representation address space, it was necessary to protect that variable by a semaphore. The only form of communication available in VMS was message passing. While it was possible to construct a semaphore out of these messages, the end result, in the words of those involved in the Newark experiment, was that one used a millisecond of processor time to protect a microsecond operation [17].

There was another practical disadvantage of requiring that all of the variables that were to be checkpointed be in the same address space. The Newark/Pascal compiler did not provide a flexible way of stating in which address space a variable should be placed. Instead it imposed the convention that the representation address space contained all global variables (i.e. those declared in the outermost block of the Pascal program) and nothing else. In particular, heap variables allocated with the Pascal *new* primitive were never in the representation address space. Thus, when programmers would normally use heap allocation, they were forced to implement their own storage allocators that subdivided a large array in the global data space. They were thus forced to write and maintain unnecessary code -- code that was hard to maintain because it was forced to breach the type system.

Newark did not provide adequate support for Eden's main primitive – Invocation. Each object typically exports a number of operations that are characteristic of its function. For example, the Directory object exports operations called *Insert* and *Lookup*. The MailBox object exports operations *Deliver*, *Pickup* and *Remove*. Naturally, the parameters to each of these operations are different.

The Newark kernel provided a single invocation primitive. Its parameters included the Capability for the target object (the object to be invoked), a timeout, and the name of the operation to be performed. These parameter were of fixed types. However, the actual parameters of the invoked operation had types which depended on the operation itself. Moreover, since new of object types were created frequently, it was impossible to make allowance in the specification of the kernel invocation primitive for all possible invocation parameter lists. The heading of the kernel invoke operation therefore defined the invocation parameters to be of a type called *Unspec_data*. It was the responsibility of the application programmer to define *Unspec_data* appropriately before he included the file that defined the heading of the kernel invocation primitive. What does "appropriately" mean? The typical definition of *Unspec_data* was as a large variant record, with one variant for each invocation that could be made from the object being coded. Each variant contained the parameters for the appropriate operation. The result of this was a lot of unreadable and unmodifiable code. Maintenance was a problem because the definition of *Unspec_data* collected together in one place information about all the possible invocations that the object might make. Changing one of those invocations, or adding a new one, meant changing code segments that were physically remote and seemingly unrelated. And of course, there was a complete lack of type checking. The result of this was that the conceptually simple and elegant invocation construct was exceeding hard, cumbersome and dangerous to use in an actual program.

## 4. The Eden Programming Language: Its Goals and Facilities

With the experience of the Newark prototype behind us, it was clear that for the next prototype of Eden some programming language support would be necessary. It was not clear that it was reasonable to expect all Eden programming to be confined to one language. However, once we had drawn up a list of features that we though would simplify programming in Eden, it became clear that implementing these features on top of many base languages would represent an excessive amount of work. We therefore decided to concentrate our efforts, at least initially, on a single language, so that we could gain experience of distributed programming. Thus was the Eden Programming Language born.

In the event, we have not implemented similar support for other languages. The most important feature of EPL, concurrency, cannot readily be added to an arbitrary existing language. Programmers who need the facilities of some other language usually write subroutines in that language, and then call them from an EPL program.

### 4.1. Design Objectives for EPL

Our experience with Newark/Pascal made us realise that support for parallelism within EPL was a prerequisite: system-level support for concurrency was just too expensive. Another requirement for a language that was to be used for a system of the 1980's was that it provided at least those facilities that software engineers considered essential in the 1970's – modularity and data abstraction.

The other features that seemed to be essential to EPL were support for capabilities, checkpointing, and invocation. Since capabilities in Eden play the rôle of object pointers, they provide a system-level analogue to references. How should these references br typed? We did not wish to prevent programmers from using objects in unforeseen ways; neither did we wish to stifle evolutionary growth. For example, the dynamic typing of Smalltalk enables a client program which previously accessed an object of type $T_1$ to access a object of a new type $T_2$ instead, provided that $T_2$ supports all the operations of $T_1$ that the client uses. A strong typing system of the kind found in Pascal and its derivatives would prevent this.

Another clear requirement for the Eden Programming Language was a sensible invocation syntax. A simple way of writing both parameters and results to an invocation was clearly needed. Also, a simple way of receiving an invocation and allocating it to one of the processes of the invoked object was required.

One area in which Newark was very successful was the implementation of checkpoint. However, Newark's simplicity was obtained at the expense of requiring that all of the variables that were to be checkpointed be declared together. System level facilities like address-space boundaries were clearly too "heavy" to be used for the definition of checkpoint data. We therefore required that EPL provide a flexible way of defining which variables should be checkpointed. One consequence of this requirement was that the implementation of the checkpoint operation itself might be more complicated than in Newark.

### 4.2. EPL and Concurrent Euclid

A major decision was that we should not attempt to design and implement a complete programming language. It is very easy for system builders to underestimate the task involved in designing, specifying and implementing a programming language. The goal for the EPL sub-project was not to solve all the unsolved problems of language design. Rather, it was to provide some practical support for the programmers of the Eden system. We therefore decided to base EPL on an existing language, aiming to choose one that provided at least some of the features we wanted.

Another requirement on the language was that an implementation should be available on VAX computers under the UNIX[1] operating system, as this combination had been chosen as the vehicle for our second prototype. After considering a number of possible languages, we eventually settled on Concurrent Euclid, which had recently been implemented by Holt and his colleagues at the University of Toronto [11] [13]. Concurrent Euclid (CE) can be considered to be a Pascal superset providing modularity and parallelism, and also some features for systems programming such as explicit type conversion and the ability to locate variables at absolute addresses. It is more static than the Euclid language [14] in that there are no module types. As a consequence, each module in the running program corresponds to a piece of text in the source code.

## 4.3. Invocation Support in EPL

Invocations, like procedure calls, should be typed checked; it is unacceptable to leave type errors undetected at the very place where they are most likely to occur – the boundaries between modules (objects). Invocations must be sent and received; invocation support is therefore required for both the invoker and the invokee. We will discuss these three topic in turn.

### *4.3.1. Type Checking of Invocations*

Treating capabilities as untyped variables seemed to be the simplest way of achieving the flexibility of being able to invoke an arbitrary object. EPL capabilities would be like PL/I pointers; a given language variable would be typed as a capability, but the language would not distinguish between capabilities for Mail Boxes and capabilities for Distribution Lists. At the same time we wanted to avoid the possibility (that exists in PL/I) of misinterpreting the type of some datum accessed via a capability (pointer). Without compile-time enforcement, this implied that there would be a run-time check on each invocation.

There are two arguments in favour of compile-time rather than run-time type checking. The first is one of run-time efficiency; this we ignored because the run-time efficiency of the prototype was not considered to be very important, and because an invocation would in any case be a fairly expensive operation, to which a run-time check would add only a small overhead. The second argument is that compile-time checks prevent an erroneous program from being run at all, whereas a run-time check only signals an error when the incorrect code is executed – which may be months or years after the program is installed. This argument is more persuasive, although it applies less in a prototyping environment (where the implementor is at hand after the program is installed) than for a production program. We chose run-time checking primarily because it enabled us to postpone the research involved in developing a sufficiently flexible compile-time typing scheme, and get on with the primary task of providing a service for the rest of the Eden Project.

To avoid the danger of interpreting a value of one type as though it was of another, we decided that all the values sent in an invocation would be tagged as to their type. The Eden Standard Code for Information Interchange (ESCII) was developed to standardise this tagging. Although the data structure used to represent an ESCII has evolved, the basic idea is unchanged. Each argument to an invocation is preceded by a tag giving its type, and if it is an array also its number of dimensions and their sizes.

### *4.3.2. Sending Invocations*

Our initial thought in choosing a syntax for invocation was to add some new syntax to the language. We considered a syntax like that of 3R [4], which has the advantage of explicitly distinguishing parameters from results. For example, an invocation of the *Lookup* procedure of an object *d* might appear as

*capa, status := d.Lookup["Farnsworth"]*

Using the 3R syntax, the compiler would deduce the types of the formal parameters from the types of the variables and values presented as invocation arguments and results, and would use this information to build an appropriate ESCII for the invocation.

Another possibility was to use CE procedure call syntax for invocations. This would require a declaration for each invocation stating the type of each parameter and result. Invocations of the same name on different types of objects would be accommodated in the same way as procedures that exist within different modules. In other words, we would try to unify (at least syntactically) the concept of an operation within a module with that of an invocation on an object. This scheme also had the advantage that it would

[1] UNIX is a trademark of AT&T Bell Laboratories.

not be necessary to enlarge the programming language in any way. This second course of action is in fact the one we took. Because of this it was possible to implement almost all of EPL without changing the Concurrent Euclid compiler. Every invocation appears as, and indeed is, an ordinary procedure call. The procedure that is called is known as a stub procedure and is automatically generated by a piece of software called the "Stubber". The action of the stub procedure is to take the parameters of the various types that are supplied with a particular invocation, to package then up into an ESCII, and to call the synchronous invocation primitive. When this returns, the stub procedure unpackages the results from another ESCII and returns to the calling program, in the same way that a normal procedure would return. Figure 4.1 shows the whole of the stub procedure for the *Lookup* invocation.

Unlike Pascal, Concurrent Euclid allows procedures to take arrays of arbitrary size as parameters. In a normal procedure call, it is of course the caller who determines the sizes of both argument and result array parameters. When an invocation takes place, the stub code in the invoked object must reconstruct the original parameter list. This means that it must allocate storage not only for the array arguments, but also for the array results. It is thus necessary for the sizes of the result array parameters to be passed with the invocation request message. For simplicity, we in fact check the types of all result parameters at the same time as we check the arguments, i.e. when the invocation request message is received by the invokee. To make this possible, there is space in an invocation request ESCII for the types and sizes of the result parameters.

Because the default invocation syntax is a procedure call syntax, there needs to be some way of resolving the conflict that would arise if, for example, two different objects both chose to export an operation called *Lookup* but with different parameter lists. The module construct of Concurrent Euclid serves this purpose very

```
procedure Lookup(
  Target_EjectP : Capability,
  LookupPathP : String,
  var ActualCapaP : Capability,
  var EstatusP : EdenStatus)=
  imports(var Dispatcher,
        var Esciis,
        var StatusDefs,
        var StrDefs,
        ArrLength)

begin
  var Values : Escii
  var Results : Escii
  var ValSize : Integer
  var ResultReader : EsciiReader
  var ValueWriter : EsciiWriter
  var es : EsciiStatus
  var elemcount : Integer
  var OpName : String := StrDefs.Str('Lookup')
  const OpSpaceInEscii := Esciis.SpaceForType +
    Esciis.IntegerSize + 6

  ValSize := Esciis.SpaceForNothing +
      OpSpaceInEscii +
      Esciis.SpaceForVariable(
        Esciis.StringToken, LookupPathP) +
      Esciis.SpaceForType
  Values := Esciis.ESCreate(ValSize)
  Esciis.ESOpenWrite(Values, ValueWriter)
  Esciis.ESWrite(ValueWriter,
    Esciis.StringToken, OpName, es)
  assert (es = Esciis.ESCIS_Success)
  Esciis.ESWrite(ValueWriter,
    Esciis.StringToken, LookupPathP, es)
  assert (es = Esciis.ESCIS_Success)
  Esciis.ESWriteType(ValueWriter,
    Esciis.CapabilityToken, es)
  assert (es = Esciis.ESCIS_Success)
  Esciis.ESCloseWrite(ValueWriter)

  Dispatcher.SynchInvoke(
    Target_EjectP, Values, Results, EstatusP)

  if StatusDefs.InvocationWasSuccessful(EstatusP) then
    Esciis.ESOpenRead(Results, ResultReader)
    Esciis.ESRead(ResultReader,
      Esciis.EdenStatusToken, EstatusP, es)
    if StatusDefs.InvocationWasSuccessful(EstatusP) then
      Esciis.ESRead(ResultReader,
        Esciis.CapabilityToken, ActualCapaP, es)

    end if
    Esciis.ESCloseRead(ResultReader)
    Esciis.ESDiscard(Values)
    Esciis.ESDiscard(Results)
  end if
end Lookup
```

Figure 4.1: The stub procedure for *Lookup*

well. When the Directory object is compiled, all of the stub procedures needed to invoke it are placed in a module called *Directory*. So, when a client invokes *Lookup* on a Directory, it actually calls the *Directory.Lookup* procedure. This is shown in Figure 4.2. Similarly, to look something up in an Environment, the *Environment.Lookup* procedure is called. Observe once again that capabilities themselves are not typed; if *CurrentDirectory* is a capability for a Directory, there is no compile-time check that prevents the invocation

*Environment.Lookup(CurrentDirectory, "TerminalHandler", ...)*

from being issued. However, if the parameters of the two *Lookup* invocations are different, then a *EPLWrongParameters* status will be returned at run time, and no invocation procedure will have been called.

If the two *Lookup* invocations take the same parameters, the invocation will go ahead; the name of the module is not significant. Indeed, the bodies of the two stub procedures will be identical; we make no assumption that the invoked object is actually of the same type as the module. This is because various different objects may in fact support the same operations, and we do not wish the invoker to know which particular concrete type he is invoking. As an example, there is a set of invocations, including *Transfer* and *Close* that are concerned with byte-stream input and output. The *Transfer* invocation is actually declared in a module called *EIO*, but it is understood by numerous objects. Examples are the *ByteStore* object, which stores a sequential file, and the Eden Terminal Handler, which implements virtual screens on a terminal; both of these objects can be invoked with the *EIO.Transfer* invocation.

It is also possible to use an object oriented syntax for invocation. Returning to the previous example, the *Lookup* invocation can be invoked on *CurrentDirectory* by writing

*CurrentDirectory.Lookup("TerminalHandler", ...)*

The advantage of this second syntax is not just that it avoids an extra word, but that it brings out the object-oriented nature of the system. The syntax *CurrentDirectory.Lookup* much more clearly reflects the fact that the *Lookup* operation is possessed by *CurrentDirectory* than does the type-oriented notation *Directory.Lookup*, which implies that a type *Directory* possesses the *Lookup* operation.

The astute reader will notice a problem. When the EPL compilation system processes the invocation *CurrentDirectory.Lookup*, it must generate a call to a stub procedure; but it is not obvious whether it should call *Environment.Lookup* or *Directory.Lookup*. We resolve this ambiguity by permitting the object oriented invocation syntax only when the target

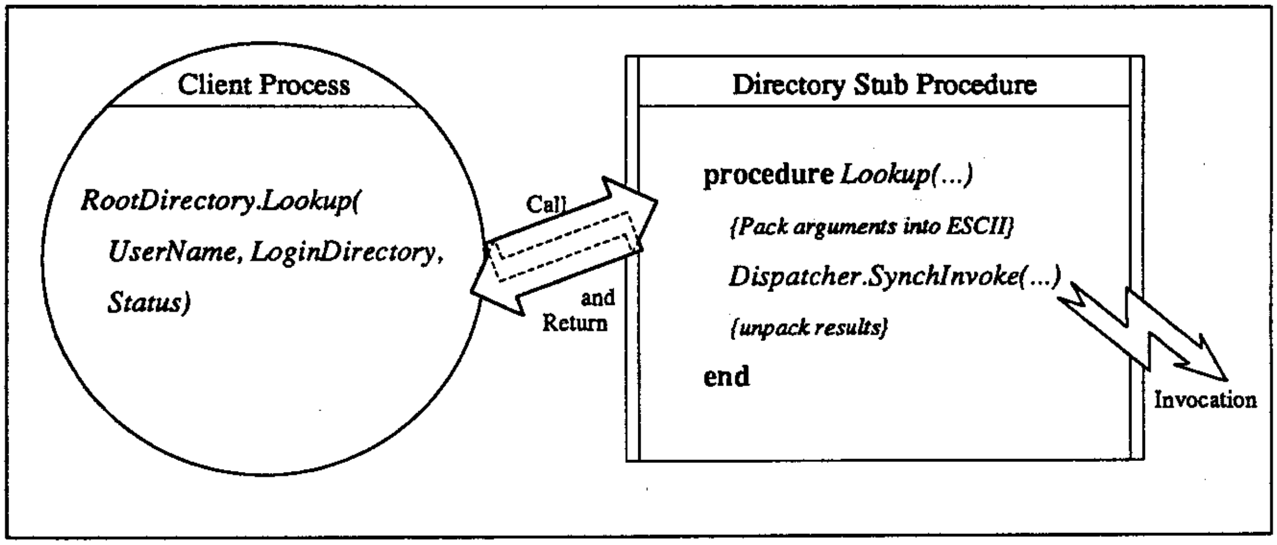

Figure 4.2: Sending a *Lookup* invocation

```
invocation procedure Lookup (
  Callersrights : EdenRights,
  LookupPath : String,
  var ActualCapa : Capability,
  var Estatus : EdenStatus) =

  { Possible values of Lookup's Return Codes:
   EFSS_Success
   EFSF_BadPathName
   EFSF_NoSuchPath }

  imports (var StrDefs, Extractor, var Directory,
       var StatusDefs, LookupEntry, var Kernel)
begin
  var Pathname : String
  var NewPath : String
  var SearchName : String
  var SubDirect : Boolean
  var DirectoryCapa : Capability for Directory
  var MyCapa : Capability for Directory
  var RetCapa : Capability
  var SubEstat : EdenStatus

  Kernel.CapaMakeNull (ActualCapa)
  PathName := LookupPath
  Kernel.CapaForMe (DirectoryCapa)
  Kernel.CapaForMe (MyCapa)

  loop
    { Peel off new SearchName from PathName }
    Extractor (PathName, SearchName, NewPath, SubDirect)

    { Lookup the capability of SearchName in DirectoryCapa }
    if not Kernel.CapaEqual (MyCapa, DirectoryCapa) then
      DirectoryCapa.LookupEntry (
        SearchName, RetCapa, SubEstat)
      if not StatusDefs.InvocationWasSuccessful (SubEstat) then
        Estatus := EFSF_BadPathName
        exit
      end if
    else
      LookupEntry (
        EdenRights (), SearchName, RetCapa, SubEstat)
    end if

    if SubEstat = EFSF_NoSuchName then
      Estatus := EFSF_NoSuchPath
      exit
    end if

    Estatus := EFSS_Success

    if not SubDirect then
      { Pathname does not refer to subdirectories }
      ActualCapa := RetCapa
      exit
    end if

    { The capability returned denotes a subdirectory to be
     descended. Now go back and search that directory for
     the capability of the subsequent SearchName. }
    DirectoryCapa := RetCapa
    PathName := NewPath
  end loop
end Lookup
```

Figure 4.3: The body of the *Lookup* invocation procedure

capability has been declared to be **for** some object type. In the context of the declaration

var *CurrentDirectory: Capability* for *Directory*

a call to *Directory.Lookup* will be generated.

The reader should note that Capabilities remain untyped despite the *Capability* **for** syntax. In the scope of the declarations

```
const Servers: Capability for Directory :=
                              Eject "system/servers"
var Env: Capability for Environment
var C: Capability
```

it is possible to assign *Servers* (or any other capability value) to both *Env* and *C*, and to invoke any operation on any capability using the unabbreviated syntax.

## *4.3.3. Receiving Invocations*

EPL provides the programmer with assistance in receiving invocations as well as in sending them. For each invocation that an object is willing to accept, the programmer declares an invocation procedure. These look exactly like ordinary procedures except that their declarations are prefixed by the word **invocation**. The first parameter of an invocation procedure must be of type *EdenRights* and the last must be of type *EdenStatus*. Moreover, all the arguments (input parameters) must precede any of the results (output parameters, indicated by the keyword var). Invocations do not have in/out parameters. The body of the *Lookup* invocation procedure of the directory object is shown in Figure 4.3.

In addition to declaring the appropriate invocation procedures, the programmer of a server also needs to arrange that the incoming invocations are actually received. A library module called the Dispatcher undertakes all the low level negotiations with the kernel that are required for invocation message dispatch and the enqueuing of incoming invocations. The programmer receives invocations from the Dispatcher by use of the routine *Dispatcher.ReceiveOperations*; this takes as an argument a set of invocation names and returns as result a handle to a particular invocation. Having received an invocation in this way, the programmer can call the appropriate invocation

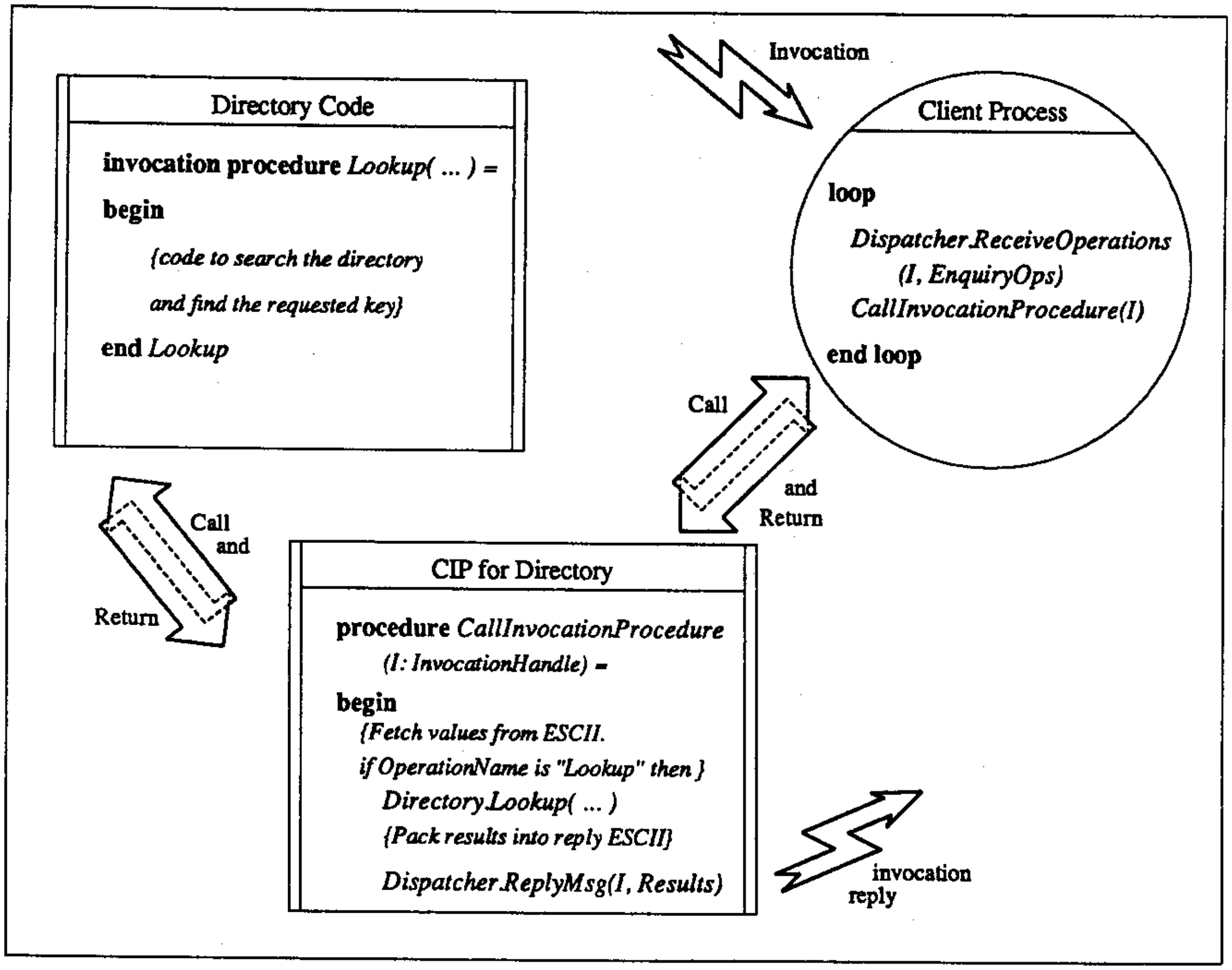
Invocation
Directory Code
invocation procedure Lookup( ... ) =
begin
{code to search the directory
and find the requested key}
end Lookup
Client Process
loop
Dispatcher.ReceiveOperations
(I, EnquiryOps)
CallInvocationProcedure(I)
end loop
Call
and
Return
Call
and
Return
CIP for Directory
procedure CallInvocationProcedure
(I: InvocationHandle) =
begin
{Fetch values from ESCII.
if OperationName is "Lookup" then}
Directory.Lookup( ... )
{Pack results into reply ESCII}
Dispatcher.ReplyMsg(I, Results)
invocation
reply

Figure 4.4: Receiving a *Lookup* invocation

procedure by means of an automatically generated piece of code called *CallInvocationProcedure*. This is shown in Figure 4.4. If the operation name in the invocation is not one for which he has declared an invocation procedure, the effect of *CallInvocationProcedure* will be to return the invocation to the invoker with an *EPLF_UndefinedOperationName* status. If the operation name in the invocation matches that of a declared invocation procedure but the parameters are of the wrong types, then CallInvocationProcedure returns the status *EPLF_WrongParameters*. If name and parameters match then the appropriate invocation procedure is called, and from inside it the programmer is free to take whatever actions are appropriate. Returning from the invocation procedure causes the invocation reply message to be sent.

## 4.4. Capability Protection

Another design goal of EPL was to provide capabilities as first-class objects. Within an EPL program, a capability is represented by the actual ten-byte quantity used to name objects within the Eden kernel. It is not represented by an index into a c-list [19]. This lets programmers access the rights field of a capability, compare two capabilities to see if they name the same object, restrict the rights in a capability, and assign capabilities in a very natural way. Both capability variables and constants may be declared. Capability constants are identifiers that name a new form of expression called a capability denotation. Capability denotations are similar in intent to integer denotations: they are resolved into capabilities when the object is compiled. It is thus possible to keep within the code of an object capabilities for arbitrary other objects. This means that we do not need to leave capabilities for sensitive objects in the directory system; rather the piece of software that needs these capabilities can have them "compiled in".

Protection of capabilities against forgery and theft is achieved in what we believe to be a novel way. For each object, the Eden kernel keeps a table of capabilities owned by that object. When an object is first created, this table contains the object's own capability and any capability constants that formed part of its type. (For example, the mail system send interface type contains a constant capability for the mailbox directory. This type forms the code for instances of the send interface, and so each instance also contains that capability.) Whenever an object attempts to make a kernel call, such as the call that send an invocation message, the kernel checks that the target capability, and any other capabilities contained in the message, are present in the sender's capability table. If any of the capabilities are not present in the table, then they have not been obtained legitimately, and the kernel call will fail with an appropriate status. In the case of an invocation, the inverse operation takes place at the destination node. Just before the invocation message is passed to the target object, the target's kernel adds a copy of all capabilities in the message to the target's capability table. Thus, when the target attempts to use these capabilities, they will be considered to be legitimate. Similar checks and table manipulations take place with an invocation reply. The actual process is a little more complicated than described here, since the kernel table contains just one entry for each *uid*; that entry contains the union of all the rights legitimately obtained by the object in question. The capability table is written to stable storage as part of the checkpoint operation, and re-initialised as part of a restore.

The effect of these rules is that, within its own address space, an object is free to assign, delete, and indeed fabricate capabilities as it sees fit. Some fabrication is legitimate; for example, an object may restrict the set of rights in a capability. Other capability fabrication is illegal; for example, constructing a capability out of an arbitrary node number and timestamp, or expanding the set of rights beyond that which the object legitimately possesses. However, if an object indulges in these illegal acts, it is fooling no object but itself. Its kernel will prevent it from ever using an illegally constructed capability.

Note that in Eden capabilities are owned by objects, not by procedures. Eden has no equivalent of Hydra's Local Name Space [19]. Eden is less secure than Hydra in that an object may retain a copy of any capability passed to it for an indefinite time. However, the view of capabilities and rights seen by the Eden programmer is far simpler than that presented by Hydra.

## 4.5. Strings in EPL

One early addition that we made to EPL was the provision of a comprehensive strings package. Like Pascal, Concurrent Euclid provides only fixed length character arrays and not true strings. Although a strings package might seem to be a trivial piece of software, a number of practical problems arise when writing a strings package for production use.

It is all too easy to provide functionality at the expense of performance, with the result that strings are too expensive to use routinely, and programmers revert to character arrays. One of our goals in designing the Eden package was to make simple uses of strings cheap. In particular, unnecessary copying was to be avoided. We achieved this by using pointers to existing character arrays, rather than copies, whenever possible. A string is represented by a *<length, address>* pair; taking a substring of an existing string is done by simply constructing a pointer to the middle of that string. Construction of a new string by concatenating substrings of existing strings involves copying the characters only once.

One problem with adding strings in a package rather than having them built into the language is that there are no denotations for strings. Concurrent Euclid does provide denotations for character arrays, which take the form

*'This is a character array'*

The strings package provides a conversion function *Str* to turn such arrays into strings:

var *s: string := StrDefs.Str('a character array')*

In the case that the argument of *Str* is a compile time constant, the compiler will allocate static storage for it; it is therefore safe for *Str* to return a pointer to this storage. In the case that the argument is a character array variable local to an inner procedure or block, the storage that it occupies will be deallocated at block exit; furthermore, the value of the array variable may be changed by assignment. In these cases *Str* ought to copy the characters into freshly allocated storage. Rather than provide the programmer with two versions of *Str*, with the attendant danger of calling the wrong one, we took advantage of the run-time storage layout of UNIX programs on VAXes and Suns. If the address of *Str*'s argument is very large, it is assumed to be on the stack, and is copied; otherwise it is not copied.

Of course, it is rather cumbersome to have to call *StrDefs.Str* every time one needs a string constant. The preprocessor that performs the first phase of the EPL translation process therefore accepts string denotations surrounded by *double* quotes, and turns them into the appropriate call on a string creation function. So in EPL one can write

var *s: string := "a string"*

(In fact, the call inserted by the preprocessor is to a special string-creation function, *ZZZStrZZZ*, that never copies its argument.)

All of the above assumes that strings can be returned by functions. Unfortunately, Concurrent Euclid imposes the restriction that the result of a function call must fit in one machine word, i.e., 32 bits on our machines. We divided this into 12 bits for the length field and 20 bits for the address, thus limiting strings to a length of 4095 bytes, and programs to 1 Mbyte of string space. In practice, the latter limit has caused some unpleasant surprises.

So far nothing has been said about the deallocation of strings. This is the weakest part of the package. In achieving our goal of minimising copying, we have created a situation in which it is very hard for a programmer to know who "owns" a given string, and whether it is safe to deallocate it. We took the attitude that most programmers ought to assume the presence of garbage collector that "did the right thing"; however, the fact that we have not implemented such a garbage collector† means that a *FreeString* operation is provided for those programmers who need it. There are also explicit rules about string ownership. For example, string arguments to invocations are the property of the invocation procedure, not the caller. This is convenient in, for example, the Directory object *AddEntry* invocation procedure, where the new string can be linked into the directory data structure without the necessity of making a copy. It is less convenient in the *LookupEntry* procedure, when the name provided as the argument to *LookupEntry* must be explicitly freed.

In a similar way, string results of an invocation procedure are deemed to be owned by the procedure itself. Most of the time this rule is appropriate, but it does lead to complications in a few cases. Consider the *Read* invocation procedure *Read* in the *Bytestore* type. Bytestores represent a sequential file of bytes as a sequence of fixed-size blocks. When a *Read* invocation returns a string that is entirely within one block, the invocation procedure does not need to make a copy of the bytes: it merely creates a string that references a part of one of the blocks. In this case the string must not be freed, and the *Read* procedure benefits from the knowledge that no attempt to do so will be made. However, when the caller of *Read* requests a string that crosses a block boundary, a new string must be created by concatenation: this string belongs to the *Read* invocation procedure, and should be freed *just after* it returns and the result parameters have been marshalled into the reply ESCII. Unfortunately, this marshalling occurs within *CallInvocationProcedure*, a piece of automatically generated code that has no knowledge of which string results should be freed. After *CallInvocationProcedure* returns, the string is garbage, in the sense that there is no variable that names it, and thus there is no way of freeing it.

Our solution to this problem is less than elegant. We have written a module called *StringSing* that serves as a "death-row" for strings. Using *StringSing.MarkForDeath*, the invocation procedure can mark a string for future deallocation. After the *CallInvocationProcedure* call, the marked strings belonging to the current CE process can be deallocated by calling *StringSing.Execute*. This is clumsy, but does solve the problem of string result deallocation for that relatively small number of invocation procedures where it is a problem. The alternative to *StringSing* would have been to change the string ownership rules so that results were deemed to be owned by the caller, not by the

† Although Concurrent Euclid is strongly typed, except for explicit loopholes, the strings package itself would cause problems for most garbage collectors: they will (rightly) object to pointers to the middle of character arrays.

invocation procedure. Unfortunately, this would have meant making copies of results in the common case where the invocation returns information from the object's data structures.

## 5. Implementation

There are five main parts to the implementation of EPL: the EPL Preprocessor, two Concurrent Euclid program generators, the dispatcher module, the Concurrent Euclid compiler, and various library modules. Each of these components will be discussed in turn. Figure 5.1 shows how the various components of the translation system fit together.

### 5.1. The EPL Preprocessor

The EPL preprocessor understands the syntax of EPL and deals especially with invocation procedures, capability for declarations, capability denotations, and string denotations. It produces two outputs: a Concurrent Euclid program containing calls to library procedures and automatically generated procedures, and a binary file called *interface.code*. This file contains a description of the object's invocation interface, that is, it lists the name of each invocation procedure and the parameters that the procedure takes. A utility program converts *interface.code* into a human-readable form that is used for documentation (and as an external definitions file for the compiler).

### 5.2. Program Generators

Two program generators, the stubber and the cipper, emit Concurrent Euclid source code that is subsequently compiled. The stubber generates stubs that are used by other objects that invoke the object that is being compiled. The cipper generates a *CallInvocationProcedure* that is tailored to the needs of the object in question, and which is eventually linked with it. The information needed to generate the stubs and *CallInvocationProcedure* is obtained from the *interface.code* file. The stub procedure used to call *Lookup* was shown in Figure 4.1; the cip procedure used by the directory object is shown in Figure 5.2.

### 5.3. The Dispatcher

The dispatcher is a library module that is used to simplify invocation. The dispatcher uses the asynchronous invocation primitive provided by the Eden kernel to implement an entry called *SyncInvoke* that provides synchronous invocation for EPL processes. It also implements the *ReceiveOperations* call (and some other *Receive...* calls). The stub procedures enter the dispatcher by calling *SyncInvoke*; the dispatcher in turn calls the kernel *AsyncInvoke* primitive and enqueues the calling process on a condition variable until the arrival of the invocation reply message. In this way many invocations made by separate EPL processes can be in progress at once. When an invocation arrives at an object, the object is notified by a UNIX signal (software interrupt). A daemon process inside the dispatcher waits for the arrival of this signal and then receives the invocation message. The message will either be the reply to a previously transmitted invocation message or a new incoming invocation request. In the first case, the calling process will be suspended on a condition variable. The action of the dispatcher daemon is simply to find the right condition variable and signal it, and to make arrangements for the calling process to access the data in the reply message.

If the message is an incoming invocation request, the dispatcher looks at the name of the operation in that invocation. It then searches a data structure to determine if there is a process waiting for an invocation with that name. If there is such a process, the dispatcher provides the process with the invocation message and signals the appropriate condition variable. If there is no such process, the invocation is queued until a process calls the appropriate *Receive* entry.

### 5.4. The Concurrent Euclid Compiler

The fourth main component of the EPL implementation is a slightly modified version of the University of Toronto Concurrent Euclid compiler. The changes that we have made fall into three classes. The most major set of changes is independent of Eden and EPL. It is the addition of symbol table directives compatible with the UNIX debugger, *dbx*. We have also modified dbx itself so that it deals reasonably with CE processes.

A second category of compiler changes represents minor extensions to and modifications of the Concurrent Euclid language itself. I give three examples. Our compiler supports sets whose base type has a cardinality of 32, whereas the Toronto compiler supports sets with a base type cardinality of not more than 16. A larger set

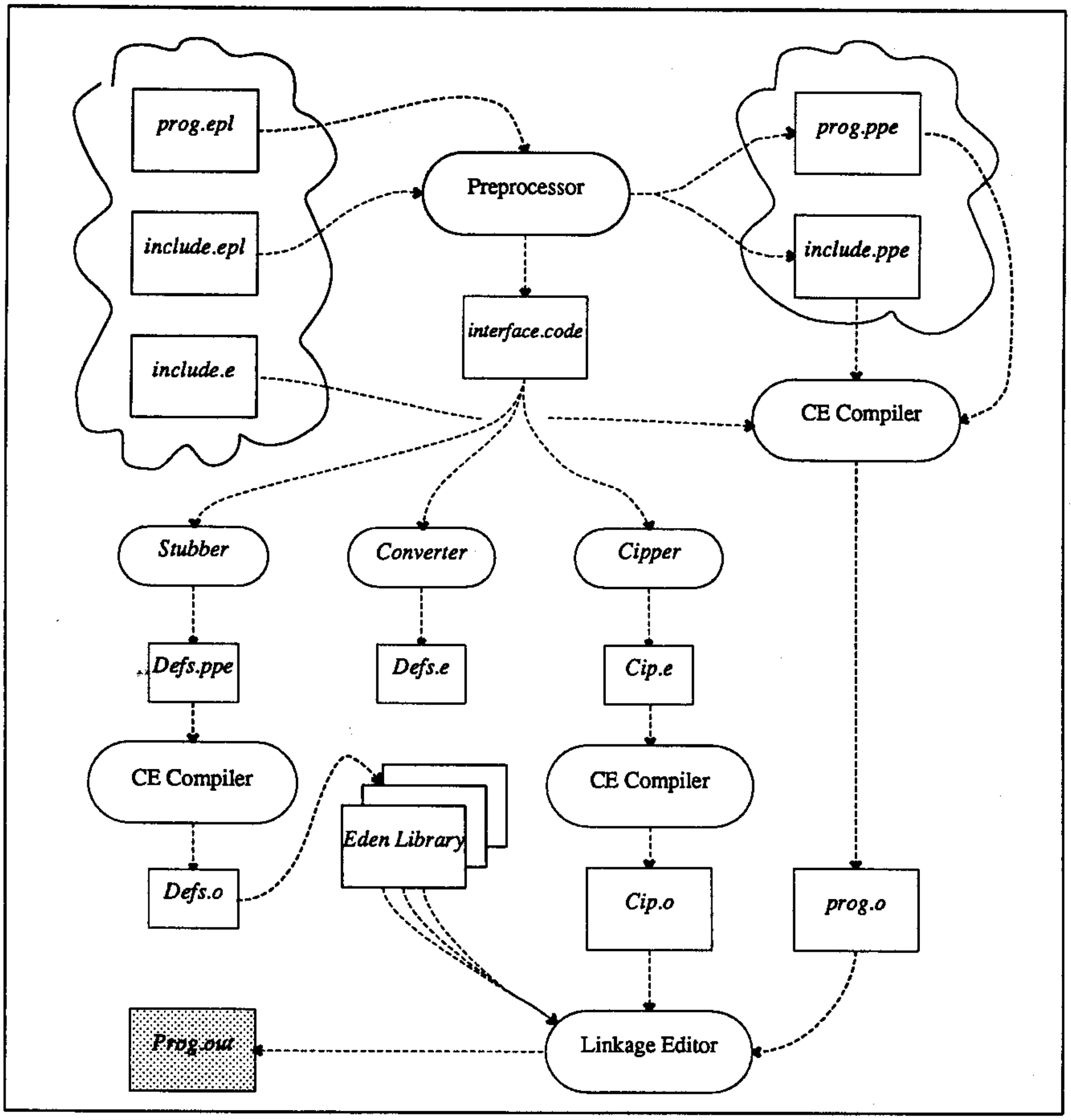

Figure 5.1: The internals of the EPL translation system.

Boxes are used to represent files, and ovals to represent programs; the dashed lines show data flow. The cloud in the upper left corner represents the EPL source program. The shaded box labelled *Prog.out* represents the final output, ready to become the code for a new Eden type. Note that *Defs.o* and *Defs.e* are not used in the compilation of this object; they will be used in the translation of another object that invokes the object whose translation is illustrated.

of constant expressions are considered manifest by our compiler than by the Toronto compiler. When declaring an external procedure, it is possible with our compiler to give an external (linkage editor) name for that procedure, whereas the Toronto compiler assumes that the external name is the same as the internal name. This facility is useful when interfacing with the C language code used to implement the Eden kernel library. In the absence of an explicit external name, one is constructed from the module name

```
procedure CallInvocationProcedure(var I : KernelInvkHandle) =
  imports(var Dispatcher, var Esciis, var StatusDefs,
        var StrDefs, malloc, dealloc, var Directory)
begin
  var Values : Escii
  var Results : Escii
  var ResSize : Integer
  var ValueReader : EsciiReader
  var ResultWriter : EsciiWriter
  var OpName : String
  var InvokersRights : EdenRights
  var EPLStatus : EdenStatus
  var es : EsciiStatus

  EPLStatus := StatusDefs.EPLS_Success
  Dispatcher.InspectValueParams(I, Values)
  Dispatcher.InspectCallersRights(I, InvokersRights)
  Esciis.ESOpenRead(Values, ValueReader)
  OpName := StrDefs.NullString
  Esciis.ESRead(ValueReader,
    Esciis.StringToken, OpName, es)
  if StrDefs.Equal(OpName, "AddEntry") then
    . . .
  elsif StrDefs.Equal(OpName, "CheckpointSelf") then
    . . .
  elsif . . .

  elsif StrDefs.Equal(OpName, "Lookup") then
  begin
    var LookupPathP : String
    var ActualCapaP : Capability
    loop {once}
        Esciis.ESRead(ValueReader,
          Esciis.StringToken, LookupPathP, es)
      exit when es not = Esciis.ESCIS_Success
        Esciis.ESReadType(ValueReader,
          Esciis.CapabilityToken, es)
      exit when es not = Esciis.ESCIS_Success

        es := Esciis.ESCIS_Success
      exit
    end loop {once}
    if es = Esciis.ESCIS_Success then
      Directory.Lookup(InvokersRights,
          LookupPathP, ActualCapaP, EPLStatus)
      ResSize := Esciis.SpaceForNothing +
        Esciis.SpaceForType + Esciis.EdenStatusSize +
        Esciis.SpaceForType + Esciis.CapabilitySize
      Results := Esciis.ESCreate(ResSize)
      Esciis.ESOpenWrite(Results, ResultWriter)
      Esciis.ESWrite(ResultWriter,
        Esciis.EdenStatusToken, EPLStatus, es)
      assert (es = Esciis.ESCIS_Success)
      Esciis.ESWrite(ResultWriter,
        Esciis.CapabilityToken, ActualCapaP, es)
      assert (es = Esciis.ESCIS_Success)
      Esciis.ESCloseWrite(ResultWriter)
    else
      ResSize := Esciis.SpaceForNothing
        + Esciis.SpaceForType + Esciis.EdenStatusSize
      Results := Esciis.ESCreate(ResSize)
      Esciis.ESOpenWrite(Results, ResultWriter)
      EPLStatus := StatusDefs.EPLF_WrongParameters
      Esciis.ESWrite(ResultWriter, Esciis.EdenStatusToken,
        EPLStatus, es)
      assert (es = Esciis.ESCIS_Success)
      Esciis.ESCloseWrite(ResultWriter)
    end if

  end
  elsif . . .
    . . .
  else
    ResSize := Esciis.SpaceForNothing
      + Esciis.SpaceForType + Esciis.EdenStatusSize
    Results := Esciis.ESCreate(ResSize)
    Esciis.ESOpenWrite(Results, ResultWriter)
    EPLStatus := StatusDefs.EPLF_UndefinedOperationName
    Esciis.ESWrite(ResultWriter, Esciis.EdenStatusToken,
      EPLStatus, es)
    assert (es = Esciis.ESCIS_Success)
    Esciis.ESCloseWrite(ResultWriter)
  end if
  Dispatcher.ReplyMsg(I, Results, EPLStatus)
  Dispatcher.DeallocateIncoming(I, EPLStatus)
  Esciis.ESDiscard(Values)
  Esciis.ESDiscard(Results)
end CallInvocationProcedure
```

Figure 5.2: The *CallInvocationProcedure* for the *Directory* type

and the internal name. This enables two procedures with the same name declared in different modules to be separately compiled.

The last category of changes to the compiler are those required directly by the EPL implementation. There is only one change in this category that comes to mind: it concerns the error messages emitted by the compiler. The output of the EPL preprocessor is legal Concurrent Euclid code. The EPL preprocessor takes pains to preserve the original line numbering in its output. Thus, if there is a syntax error on line 17 of the file *Prog.ppe* containing the Concurrent Euclid generated by the preprocessor, there is also a syntax error on line 17 of the file *Prog.epl* that contained the source code written by the programmer. The compiler, of course, produces an error message *Prog.ppe, line 17: syntax error.* The programmer would rather not know that *Prog.ppe* exists. There is certainly no point in his editing it: he needs to change *Prog.epl.* We therefore modified the compiler so that it accepts input files with the suffix *.ppe* (as well as with the suffix *.e*) and also so that errors found in a *.ppe* file are reported as if they had been found in the corresponding *.epl* file. The effect of these changes is that if the compiler is run from the emacs "compile-it" routine [9], the "next-error" key takes the cursor to the correct line of the file in which the error should be corrected. Programmers never need to look at the generated Concurrent Euclid code.

## 5.5. Library Modules

The final part of the EPL implementation consists of various library modules, which are linked into objects as required. The most important of these is the dispatcher, which has already been described. Other services available from the library include the strings package, a package that provides conventional read and write access to streams managed through the Eden Asymmetric Transput Protocol [5], a package providing timer services, and a package providing routines for writing data items into passive representations.

## 6. Issues Not Addressed by EPL - Some Speculations

As has been previously stated, the goal of EPL was to provide a serviceable programming language with a reasonable implementation in a short amount of time using limited manpower. In this, we were largely successful. The EPL implementation and the Eden kernel were available more or less simultaneously after about nine months of design and construction. In order to achieve this deadline, we deliberately avoided language design "issues", that is, areas that involved research. In fact, while we were developing the design for EPL, we kept two lists: one contained things that we understood how to include in EPL, while the described research issues that we would like to investigate but which could be deferred. If a topic became an issue of language design, we assiduously avoided it and moved it into the research list. Here is a selection from our research list, and some thoughts about their fate.

One of the most obvious deficiencies of the current invocation scheme in EPL is the fact that the parameters of an invocation must be of a small, fixed set of types. There is no difficulty in enlarging this set; this has happened once or twice already in the evolution of EPL. But there is a conceptual difficulty in allowing user-defined types. The problem is that of type equivalence; we must answer the question "When are two types the same?" in a distributed context.

Within an EPL program, types are equivalent only if they are identical. In other words, two types *a* and *b* are always considered to be distinct, even if *a* and *b* are in turn defined by identical text. Such an equivalence rule is quite convenient, but it is hard to see how it can be extended to a universe of more than one program. Falling back on structural type equivalence is one option, and an option that could be easily implemented by extending ESCIIs so that type constructors and field names could be described, in addition to primitive types. But of course, as language designers have widely recognised, structural equivalence is not what the programmer wants most of the time. Structural identity does not support information hiding, and has little to do with logical identity. What is needed is a system-wide way of defining types, as opposed to the program-wide way provided by EPL.

The problem with the EPL type equivalence rules is that the type names used to determine equivalence are limited in scope to a single program. For use in inter-object invocation, we need to find a way to enlarge that scope to cover the whole Eden system. In other words, we need a system-wide name space for types. The Eden system itself provides such a name space: the space of capabilities. One can imagine creating a "type object" for each type. Two different programs that wish to access the same type can do so if and only if they reference the same type object. Because capabilities are unique and cannot be forged, the type objects would provide a set of type identifiers guaranteed to be unique across the system. If two programmers refer to a type by the same capability, they must intend the very same type.

Another place where the system-wide name space of capabilities could be used is in the naming of invocation procedures. At present, invocations are named by strings. An object making a *Lookup* invocation constructs a message that actually contains the string *"Lookup"*. The object receiving it searches for this string. Because of the checks on parameter types, it might be more correct to say that the name of the invocation is *lookup (string)* returns *(capability, status)*. But in any case, a procedure heading and a procedure body that look as if they match will in fact match. This scheme has the advantage of simplicity of implementation. If the invoker and the invokee can agree on the text of a procedure heading, everything will work out at run-time. There is no necessity for any kind of system-wide database for mapping strings into unique identifiers. We wished to avoid any need for such a database in the initial version of EPL because the implementation consisted of a collection of UNIX programs running on a number of separate machines, and there was no convenient place to keep a system-wide database in that environment. However, now that Eden is available, we do have a system-wide filing system accessible from all of the separate machines. Into this filing system one could easily put the a mapping from strings and sets of parameters into capabilities. This would make the implementation of *CallInvocationProcedure*, which performs a case switch on the operation name and checks the type information in the ESCII, very much simpler and more efficient. (We have not implemented this change because it would be incompatible with all of our existing applications, and because the speed of invocation dispatch is not an issue in practice.)

Another area of research that we have not investigated in Eden is object typing. Whereas each object in the system does have a unique piece of source code associated with it which defines the invocations it accepts, we do not have any formalised notion of type conformity. In other words, there is no formal way of saying that object A behaves in the same way as object B, or that A is an extended version of B that exports some additional invocations. There are various informal ways of saying these things, of course. One example was mentioned above: a number of different objects support the asymmetric transport protocol. As another example, the Eden Mail System makes uses of an abstract type called a *MailSink* to which one can deliver mail. It is *abstract* because there is no piece of code that implements *MailSinks*. Instead, both mailboxes and distribution lists obey the invocation protocol of a MailSink. As a final example, the filing system contains a concrete type called a *ByteStore*, which implements a basic sequential file. For the purposes of the Eden compilation system, we needed to enhance this type so that it could store type code, which consists of a sequence of capabilities as well as a sequence of bytes. The enhanced type, called *TypeStore*, behaves in exactly the same way as a *ByteStore* with respect to all the *ByteStore* invocations, but in addition, has invocations to accept and emit a stream of capabilities.

What has begun to emerge from our use of objects in Eden is a directed graph of type containment, or conformity, where some types provide all the operations of others, and perhaps some operations of their own in addition. So far this graph structure has not been formalised. This has not been a problem because the number of object types (between seventy and one hundred) and the number of programmers (less than twenty) have both been small. But in the long term we would like to formalise this graph within a distributed programming system. The reader should note that the notion of a directed graph of types described here is similar to the multiple inheritance hierarchy implemented for Smalltalk–80 by Borning and Ingalls [8]. The main difference is that the Smalltalk graph relates *implementations*, whereas the Eden graph relates *semantics*. In Smalltalk, because the operations provided by a "Supertype" can be overridden by a "Subtype", there is no necessity for a subtype to provide all the operations of its supertype, or that operations should have the same argument lists. In Eden, there is no necessity for an Eden type like *TypeStore* to inherit the implementation of *ByteStore*, and in fact it does not do so.

There are two advantages to formalising such a type system. One is the documentation aid it provides to programmers about to write a new object. The other is that the formalisation makes it possible to perform object type checking. As was mentioned above, there is no object type checking at present: the capability **for** syntax, whereby one may state that a given capability is **for** a given type, is regarded as an assertion on the part of the programmer. When the variable thus declared is assigned a value, it is possible in principle to check that object in question understands the appropriate set of invocations with the appropriate parameter types. If this were done, there would be no need for the checks that are currently performed at invocation- time. However, the reader should observe that the declaration of a variable as *capability* **for** *ByteStore* does *not* imply that the variable should refer only to objects of concrete type *ByteStore*. In fact, many legal invocations would fail such a check, because many objects that *behave* as a *ByteStore* (i.e., repond appropriately to *Read* invocations, etc.) have other concrete types. What one needs to check is the *abstract* type of the object.

The issue of abstract typing is discussed further in reference [7]. Rather than attempting to resolve it in EPL, we have started designing a new object-based language, called Emerald [6], that adopts abstract typing as one of its fundamental concepts.

## 7. Conclusion

EPL is a very modest programming language. Nevertheless, it has achieved the goals we set for it. We did not attempt to push back the frontiers of programming language research; we tried instead to produce a state-of-the-art language that enhanced the programmability of Eden. In this sense EPL has been more successful than we ever hoped. Our first application, the Eden Demonstration Mail System [2] was programmed by two students who had no familiarity with the Eden System in a period of four weeks. This was at a time when the Eden Kernel itself was unstable, and the students had other course work commitments.

We believe that Eden is easy to program because the conceptually simple idea of invocation is simple to use and understand in the programming language. The programming language design strategy of avoiding anything that might be described as an "issue" has clearly paid dividends here; for a programmer familiar with a Pascal-like language there are few new concepts to learn. Our conservative implementation strategy has also paid off, in that the time taken to build a robust and reliable programming language implementation was very small. The implementation is modular in that the preprocessor, stubber and cipper are all separate programs, and each one performs a well-defined task. The changes that we have decided to make over the course of the evolution of EPL have usually been confined to one of these modules.

Preprocessors do not have a very good reputation in the world of language implementation. One of the reasons for this is that the programmer is not presented with a single language but rather with two languages: the source and the target of the preprocessor. In particular, error messages from the compiler will typically refer to the source generated by the preprocessor rather than source written by the programmer. By attention to detail (such things as keeping the line numbers consistent between the input and output of the preprocessor, and making all error messages name the preprocessor source) we have managed to achieve a good user interface. Of course, this was possible only because the input and output languages of the pre-processor are very similar.

Another thing that has helped us is that from the very beginning we decided to use a full parser in the preprocessor, even though the number of constructs that the preprocessor was initially required to recognise was very small. This initial set consisted principally of capability denotations and invocation procedures, both of which were prefixed by special keywords. We resisted the temptation to simply scan for these keywords. Instead we took the first pass of the compiler (written in S/SL [12]) and used it to construct a full parser for EPL. This has meant that on subsequent occasions it was easy to add constructs to the language. For example, string denotations are now built into the preprocessor, whereas originally the programmer had to apply a library procedure to an array of characters in order to produce a string. Another advantage of having a full parser in the preprocessor is that it detects context-free syntax errors, and thus a programmer using EPL is warned of such errors before his program ever reaches the compiler. In addition, the error messages are in the same form as those he would receive from the compiler. The strategy of using a preprocessor and some code generators enabled us to minimise our construction costs because they we did not need to understand the internals of the compiler. The execution cost of the preprocessor, stubber and cipper are very small compared with the compiler itself, although it must be admitted that the whole EPL compilation system system is slow. However, this is mostly the fault of the CE compiler, which is implemented as four passes communicating via UNIX temporary files. We are currently experimenting with a compiler that uses a separate Eden object for each pass; the objects communicate via invocation, so we hope to obtain real parallelism between the phases of the compilation, as well as to avoid the overhead of the UNIX file system.

Of course, some deficiencies of EPL cannot reasonably be fixed by preprocessing. For example, standard Concurrent Euclid does not permit procedures to be passed as parameters to other procedures. There is no obvious reason for this restriction; CE does not permit procedure declarations to be nested within other procedures, and CE modules are static, so there is no need to maintain a static chain of activation records in order to enable procedures to resolve global references. This problem could not be solved by a preprocessor, but has been fixed fairly easily by modifying the CE compiler itself. Nevertheless, I feel that the decision to use a preprocessor (at least initially) was the right one in our prototyping environment. In a production version of Eden, a complete compiler for EPL would be a reasonable option.

Another place where the use of EPL has paid great dividends is in the provision of language level processes. Compared with UNIX processes, these processes are very cheap. CE processes are created at program initialisation time, and their number is therefore fixed. Although this lack of flexibility is somewhat annoying, it is not usually a serious constraint because the processes are cheap enough to start a large number when the object is initialised and to keep them in a pool for use as they are needed. (However, managing the pool is a chore; the natural place for the management code is in the run-time system. So, we eventually added to CE a facility for the creation of detached processes. Once again, the modifications to the compiler and the run-time system were rather small.

EPL processes and monitors take advantage of the locality of an object. One of the consequences of this is that the model of computation within an object – based on variables, processes and monitors – is very different from that which applies between Eden object, which communicate by message passing. One can argue that this difference in models reflects a difference in what the hardware is able to provide, but this is an argument for different implementations, not different semantics. In any case, it remains true that scaling-up of a small application into a large one is difficult. The scaling-up process might ideally involve taking some language level processes and monitors and turning them into objects. But doing this would change the syntax in non-trivial ways and would transform the semantics completely. It would be more pleasant if the object-oriented model of computation that Eden presents was visible within the Eden Programming Language. Providing this, of course, would mean designing our own language from scratch. A small group of researchers is currently engaged in just such a project [6]. Nevertheless, for the purposes of the Eden prototype, the language level processes and monitors provided by CE are easy to use, certainly when compared with the VMS kernel processes of Newark.

## 8. Acknowledgements

The Eden project and the EPL language sub-project have both benefited from the insight, experience and hard work of a large number of people. I can mention only a few. Alan Borning made valuable early contributions to the EPL philosophy. Guy Almes made many insightful suggestions. Barry McCord and Rajendra Raj implemented and maintained the Stubber and Cipper. Norman Hutchinson was always ready to fix the CE compiler, and then to break it again to retain compatibility with the C compiler. Richard Korry and Rajendra Raj have worked hard experimenting with a new version of the EPL translation system that takes advantage of the potential for parallelism in the Eden system. Finally, the Eden programming community has been free with their bouquets and brickbats (particularly the latter), and have thus contributed significantly to the evolutionary improvement of the Eden Programming Language.

## References

[1] Allchin, J. E. and McKendry, M. S. "Synchronization and Recovery of Actions". *Proc. 2nd Symp. Principles Distributed Computing*, August 1983, pp31-44.

[2] Almes, G. T., Black, A. P., Bunge, C. and Wiebe, D. "Edmas: A Locally Distributed Mail System". *Procs 7th Int'l Conf Softw. Eng.*, March 1984, pp56-66.

[3] Almes, G. T., Black, A. P., Lazowska, E. D. and Noe, J. D. "The Eden System: A Technical Review". *IEEE Trans. on Software Eng. SE-11, Nr 1* (January 1985), pp43-59.

[4] Black, A. P. "Report on the Programming Notation 3R". Technical Monograph PRG-17, Oxford University Computing Laboratory, Programming Research Group, Oxford, England, August 1980.

[5] Black, A. P. "An Asymmetric Stream Communication System". *Proc. 9th ACM Symp. on Operating System Prin.*, October 1983, pp4-10.

[6] Black, A. P., Hutchinson, N., Jul, E. and Levy, H. M. "Distribution and Abstract Types in Emerald". Technical Report 85-08-05, University of Washington, Computer Science Dept, August 1985.

[7] Black, A. P. "Supporting Distributed Applications: Experience with Eden". *Proc. 10th ACM Symp. on Operating System Prin.*, December 1985, pp181-193.

[8] Borning, A. H. and Ingalls, D. H. H. "Multiple Inheritance in Smalltalk–80". Tech. Rep. 82-06-02, University of Washington, Computer Science Dept, June 1982.

[9] Gosling, J. *Emacs – screen Editor.* UniPress Software, Inc, Highland Park, NJ 08904, August 1983.

[10] Holman, C. and Almes, G. T. "The Eden Shared Calendar System". Tech. Rep. 85-05-02, University of Washington, Computer Science Dept, May 1985.

[11] Holt, R. C. "A Short Introduction to Concurrent Euclid". *SIGPLAN Notices 17, Nr 5* (May 1982), pp60-79.

[12] Holt, R. C., Cordy, J. R. and Wortman, D. B. "An Introduction S/SL: Syntax/Semantics Language". *Trans. Prog. Lang and Systems 4, Nr 2* (April 1982).

[13] Holt, R. C. *Concurrent Euclid, The Unix System, and Tunis.* Addison-Wesley, 1983.

[14] Lampson, B., Horning, J., London, R., Mitchell, J. and Popek, G. "Report on the Programming Language Euclid". *SIGPLAN Notices Notices 12, Nr 1* (February 1977).

[15] Lazowska, E., Levy, H., Almes, G., Fischer, M., Fowler, R. and Vestal, S. "The Architecture of the Eden System". *Proc. 8th ACM Symp. on Operating System Prin.*, December 1981, pp148-159.

[16] Liskov, B. and Scheiffer, R. "Guardians and Actions: Linguistic Support for Robust, Distributed Programs". *Conf. Rec. 9th ACM Symp. on Prin. of Prog. Lang.*, 1982.

[17] Pu, C. and Jacobson, D. "An Evaluation of Newark". Eden Project Document, University of Washington, Computer Science Dept, July 1982.

[18] Pu, C., Noe, J. and Proudfoot, A. "Regeneration of Replicated Objects: A Technique for Increased Availability". Tech. Rep. 85-04-02, University of Washington, Computer Science Dept, April 1985.

[19] Wulf, W. A., Levin, R. and Harbison, S. P. *HYDRA/C.mmp: An Experimental Computer System.* McGraw-Hill, 1981.