XEROX

BUSINESS SYSTEMS

*Systems Development Division*

November 21, 1977

| | |
|---|---|
| To: | SD |
| From: | Peter Bishop / SD at Palo Alto |
| Subject: | Dealing with objects in Mesa. |
| Copies to: | Mesa, Humus Interest |

An important design decision that every OIS programmer must make is how much generality and modularity each program and each subsystem should have. Generality and modularity allow a specific piece of software to be used without modification for a long time. The simple approach to modularity is to use separate procedures for pieces of software that should be independent. If this is the only precaution that is taken, however, then it may still be necessary for the procedures that use a given procedure either directly or indirectly to be rewritten or recompiled when the procedure is modified, even if the purpose of the procedure has remained the same and the amount of information given to it by other procedures has remained the same. The most common kind of problem with this approach arises with the data that is passed between procedures. Usually the exact representation for a piece of data is determined by estimating the requirements and the frequency of the various operations on the data and then designing a representation on which all the operations can be performed and on which the most frequent operations can be performed quickly. This means that if a procedure is rewritten, the representation of the data used by the procedure may also change. Furthermore, two different procedures may very well use two different representations for the same kind of data. Any procedure that is written to be used by both of these procedures should be able to deal with both representations.

Extreme independence between two bodies of software can be achieved if the data communicated between the pieces of software are created as *objects* and used as *objects*. In order for a procedure to be able to identify which object it is dealing with, the procedure has an *object handle* for the object. An object handle is basically a pointer to the object, but it also identifies the *type* of the object, i.e. the way in which the object is represented. The most important property of an object handle is that the size of the object handle is the same regardless of what type of object it refers to. Thus the object handle may be copied by software that is not dependent on the representation of the object itself. A major purpose of objects is to allow two pieces of software, e.g. procedures A and B, that actually interact in some way, i.e. procedure A calls procedure B, to be as independent of each other as possible. The interaction occurs when procedure B wishes to perform an operation on data, e.g. object X, that was created by procedure A. To isolate A from B, procedure B must invoke an *operation* on X that comes very close to performing the entire manipulation on X that is desired by B. If such an operation is not available, then two or more operations on X may be performed, but only by risking the amount of actual independence between A and B. As far as procedure B is concerned, this is a high-level operation on X whose exact implementation details are of no concern to B. Usually such operations can be described to people by describing the object that X is supposed to represent and then describing the operation as a meaningful operation on the abstract object that X represents. Thus procedure A must merely be able to pass an object to B on which a set of operations are defined that B considers to be high level operations. The details of the implementation are left to A, however, because A defines the data type of X. Whenever a new data type is defined, the programmer that creates the data type must specify what operations are defined on objects of that type and must provide the code that performs all of those operations.

## Objects in Mesa

There is no doubt that Mesa does not encourage the use of runtime type determination. This does not mean that it is impossible, or even unusually inefficient, to create and use objects in Mesa, however. Mesa is designed with the philosophy that there is a great deal of code on a system that in fact uses only one representation for its data and that this code will run significantly faster if does not have to continually check the data type of its data. Mesa therefore suggests that even if source code is written so that it is independent of the representation of its data, when the code is compiled it should be made dependent upon the representation of its data. If a change is made later, the code can be recompiled to make use of the changed representation. There is a good deal of merit to this argument, but it should not be seen as an absolute position that objects are *never* needed at runtime. The problem becomes, then, how objects fit into the Mesa philosophy, rather than dwelling upon the limitations of the Mesa philosophy.

The basic assumption of the Mesa philosophy is that although object-oriented programming can achieve a great deal of independence between different pieces of software, it is often possible to skimp on the implementation of the object-oriented approach in order to achieve greater speed of execution. Often, the only penalty for this skimping on the runtime implementation of objects is the need to recompile some extra pieces of software when a change is made. It must be left to the programmer to decide in each case, however, whether this extra compilation is acceptable or not, especially since there are some cases where the ability for an object handle to point to different types of objects at different times is essential. Even if the decision is made to increase the low-level runtime overhead in order to achieve certain degrees of runtime generality, however, there are still varying degrees of generality that can be achieved with varying degrees of low-level overhead. The difference in generality and in overhead is achieved by using different formats of object handle and different conventions for using object handles. Thus we see that the Humus design of object handle achieves a great deal of generality, but involves what many people consider to be too much low-level overhead and indeed it is not clear that Mesa can be made to follow the conventions for the use of Humus object handles properly. On the other hand, it is possible to define a kind of object handle in Mesa that allows a range of different representations of objects to be used without causing the overhead used by Humus object handles. The Mesa philosophy that the compiler should sacrifice runtime generality for runtime speed, especially if source-code generality is not sacrificed, leads to the conclusion that many different kinds of object handles should exist.

It becomes even more clear that it will be necessary to have several different formats of object handle in D0 Mesa because of the nature of the address space on the D0. There are MDS pointers which are extremely efficient but cannot point outside of a single MDS. There are 24 and 32 bit pointers that point within virtual memory but do not point within files. There are FileIDs that point at files. There can be innumerable kinds of pointers to objects within files, since it is necessary for software to interpret such addresses regardless of what representation is chosen.

Once the decision is made that there must be several different types of object handles, then the Mesa philosophy becomes a sufficient reason to insist that programmers be able to create their own kinds of object handles rather than using one of 20 flavors of object handle that have been designed by system programmers.

When a program A uses a piece of data X, but wishes to be independent of the representation of X, then there are two pieces of software that should be independent: program A that invokes several *operations* on X, and the *definitions* of those operations that are procedures that were written by the programmer that defined the *type* of X. If a system has only one format of object handle, then it is acceptable for either or both of these pieces of software to be dependent upon the format of object handle that is being used, but if there are several different possible formats of object handle, then a change in the type of object handle should not require program A or much of the definitions of the operations on X to be rewritten (it may be necessary for program A to be recompiled, however).

All of the constraints listed above can be achieved in Mesa 4.0 without causing any runtime overhead to separate the choice of object handle format from both the users of the objects and the implementors of specific data types. The rest of this memo presents such a style of using objects in Mesa and explores the possibilities of the proposed style. There are three programmers that must be considered: the user of an object, the definer of a data type, and the definer of a format of object handle. In order for these three programmers to be separated from each other, two definitions files are necessary (see Figure 1).

## 1. The User of an Object

The user of an object merely has to invoke operations on an object and declare variables that use the specific format of object handle. The user of an object will have a definitions file that provides a declaration for the type of the object handle, e.g. WidgetHandle. The user of the object can then declare variables to be of type WidgetHandle and then obtain such an object handle from an appropriate piece of software. The declaration of WidgetHandle should specify that all of the fields are PRIVATE so that the user of the object will not access any of the fields in the object handle explicitly. This same definitions file will contain declarations for the generic procedures that perform the legal operations on objects with that type of object handle. These procedures are generic because they operate on any type of Widget. By convention, the first argument to a procedure that performs an operation on an object is the object handle of the object on which the operation is to be performed. This is merely a notational convenience so people will be able to tell quickly which object is being operated on. The user of an object handle invokes an operation on an object by calling the generic procedure that implements that operation:

```
val ← Append [widget, anotherwidget];
```

This method of invoking operations on objects in Mesa has several advantages:

1) it can be used regardless of the representation of the object handle without causing runtime overhead over any other method of invoking the operation, and

2) it allows Mesa type-checking to protect the user against mistakenly invoking the wrong procedure on the set of arguments that have been supplied.

## 2. The Definer of a Data Type

The programmer that creates a data type must provide a declaration of the type of the object, must implement the operations on this particular type of object, and must provide procedures that create objects of this type. The definer of the data type must provide a definitions file that makes these interfaces available to client code that is dependent upon the type of object being manipulated. Note that any client code that creates an object must specify the representation of object to be created. The definer of the data type defines a procedure for each operation on the object (e.g. Append) whose calling sequence is the same as the generic procedure except that instead of a general object handle as the first argument (WidgetHandle), the first argument can only be of a specific type (HelpedWidgetHandle). It is the responsibility of the generic procedure to invoke this specific procedure on objects of the appropriate type. Note that in the case of binary operations, such as Append, only one of the arguments is guaranteed to be of a specific type by the generic procedure.

## 3. The Definer of a Format of Object Handle

The definer of a format of object handle must provide the definitions file that is used by the user of the object. The generic procedures must also be provided, but these procedures need only call the appropriate procedures created by the definers of specific data types. If the generic procedures are simple enough, they can be Inline procedures that Mesa expands to inline code instead of performing the procedure call to the generic procedure. It is this possibility of using inline procedures for generic procedures that allows us to consider this proposal as a serious alternative to the Humus approach of using objects in Mesa.

## Harsh Realities

The above description of how we would like to organize the software that deals with objects ignores a few harsh realities of Mesa. The Mesa supplied in Appendices B-D show a variety of techniques that deal with these harsh realities. Current Mesa seems to prevent the declaration of a skeleton variant record in one definitions file (e.g. Widget in Figure 1) that is expanded to include more variants in a different definitions file (e.g. HelpedWidget). If we insist on this division (as Humus does), it seems that it is necessary to use LOOPHOLES in many places that effectively robs the Mesa of much of its type-checking. The necessary LOOPHOLES can be seen in the use of TableHandles in Appendix B. Humus makes these LOOPHOLES less obvious by using some procedures that only do a LOOPHOLE. Such procedures would even be efficient in Mesa 4.0 since they would be Inline procedures.

## Variant Records

If we would prefer to have LOOPHOLE-free Mesa, however, then we can make use of variant records. The main difficulty with this approach is that there must be a single definitions file that contains the declarations of all of the variants in the variant record. This means that if a new data type is added, a new variant must be added and thus all of the software that uses this kind of object handle must be recompiled. In some cases, such a limitation on runtime generality is acceptable. For one thing, it is probable that whenever a new software release is made, most or all of the software will have been recompiled for the release. Secondly, there are many times when a programmer has a good idea of the range of data types that will be necessary, so declarations can be made for all of the data types before the software for each data type is written. As we will see later, this kind of pre-planning is even easier if prototyping is done in Smalltalk first.

If the object handle designer decides to use variant records, he has a choice of where the variant record is placed. Either the object handle or the object itself could be declared to be a variant record. In Appendix B, we see that ElementHandles are variant records. In Appendix C, however, Tables rather than TableHandles are variant records. The purpose of an object handle is to provide a small, mobile handle for the object. Thus programs copy object handles without knowing what type of object is referred to. It is often desirable to make the object handle be simply a POINTER TO the object, which is a variant record. If, however, it is inconvenient to provide type information in the object, then it may be better to make the object handle be a variant record that always contains a POINTER TO the object. The type of the variant specifies the type of object that is pointed at.

## Ordinary Mesa is a Special Case

The most interesting aspect of the approach to objects suggested here is that ordinary high quality Mesa falls out of the style in the special case where the definer of the object handle decides that there is only one possible representation for the object. In this case, the Specific Type Definitions File (see Figure 1) is used as the General Definitions File as well. In the example in Figure 1, however, the type of object handle would probably be called "WidgetHandle" rather than "HelpedWidgetHandle". Although the single definitions file appears to be the Specific Type Definitions File, this is mainly because it contains the declaration of MakeWidget. If, at a later time, the decision to have only one kind of Widget is changed, then this single definitions file becomes the General Definitions File and a new definitions file is created for HelpedWidgets. The client software must be recompiled when this change is made since the declaration of WidgetHandle would be changed. The code that implements HelpedWidgets will be modified to use the new type "HelpedWidgetHandle" to point to widgets that are known to be helpedwidgets. In addition, there will be other modifications that must be made to this code to make use of the fact that either the widgethandles or the widgets will have type information stored in them.

**Converting Smalltalk to Mesa**

There has been considerable interest in SD in using Smalltalk as a tool for constructing prototypes. Since Smalltalk is so easy to modify, this would allow many different user interface possibilities to be prototyped rapidly. At some point, the Smalltalk code would be converted into Mesa. Some people have been concerned that Smalltalk programmers would write programs that depended too much on the sophisticated features of Smalltalk to be converted into efficient Mesa. Fortunately, the essential object-oriented programming encouraged by Smalltalk can be mapped into acceptable Mesa. There are only two fundamental assumptions made in Mesa that need not be accepted by the Smalltalk programmer:

1) once an object has been allocated in virtual memory, it is not necessary to move the object in memory in order to fight fragmentation of storage, and

2) it is acceptable to have many different kinds of object handles since there is no software that is in fact used by all parts of the system and thus needs a single kind of object handle that can refer to any object in the system.

It is possible that these assumptions could cause some difficulty when converting Smalltalk programs to Mesa, but this would imply that these assumptions are invalid for the applications we are interested in writing software for. If we take the point of view that these assumptions are in fact valid for our applications, then the Smalltalk code that prototypes our applications should not run into these problems.

Thus the Mesa programmer should start from the position that storage compaction is unnecessary. This is an important point of disagreement between myself and the Humus approach, in which they created a storage compactor. I disagree with the Humus approach on this point because significant changes need to be made to Mesa in order to allow a storage compactor utility that can be invoked at any time. If storage compaction is needed, it can run at a time when no temporary variables are pointing into the middle of the area of storage that is being compacted. Even a compactor of this form should only be written when it becomes clear that it is necessary. If this approach to compaction is not adequate, it is a major indictment of Mesa and the Algol-68, PL/1 family of languages to which Mesa belongs. It is interesting to note that none of the Smalltalks before Fastalk had compaction of high speed memory, and this did not seem to be a significant problem. Paged Smalltalk did not have compaction, either, but again, there were not significant problems with fragmentation of storage. Paged Smalltalk suffered much more from the lack of a compactor that would place related objects together, rather than just a compactor that eliminated fragmentation of storage. Note that Humus also has different kinds of object handles and so it, also, is dependent upon the second assumption being valid.

One of the advantages of prototyping in Smalltalk is that it is very easy to write very general Smalltalk code. As a result, it is very easy to make changes to Smalltalk code. In particular, it is seldom necessary to make a change to Smalltalk code in order to provide generality in the code. When Smalltalk code is converted to Mesa code, however, it is necessary for the programmer to determine how much of the runtime generality is, in fact, used and to write Mesa code that has only as much runtime generality as is needed. Appendix A shows the Smalltalk sort routine and appendices B-D show three different implementations of the sort routine in Mesa. The first piece of generality that exists in Smalltalk but was eliminated from all three Mesa implementations is reference counting. It is usually possible to eliminate reference counting from Smalltalk code. In the few cases where it is not possible, however, reference counts can be supplied by the Mesa programmer and can be incremented and decremented explicitly. The frequency of incrementing and decrementing the reference counts can be drastically reduced from their frequency in Smalltalk when they are done explicitly unless the reference counting feature of Smalltalk is being abused.

The second piece of generality that was eliminated from the Mesa implementations of the sort routine was the possibility of using other representations of numbers besides integers, such as floating point, to index the table that is being sorted. Since this sort routine looks like it would work well on very large tables, however, I allowed the possibility that there might be more than 65000 elements in the table, so the indices into the table to be sorted are double precision.

The first implementation of the sort routine in Mesa (see Appendix B) uses two different types of object handles: TableHandles for the table to be sorted and ElementHandles for the elements of the table. Two different formats of object handle are used in these two cases. TableHandles use the basic Humus approach of making a table be essentially a variant record in which the tag points to an array of procedures. This must be done without help from Mesa, however, so it does not use Mesa variant records and it does use LOOPHOLES in several embarassing places. ElementHandles, on the other hand, use Mesa variant records and so do not use any LOOPHOLES. Note that both approaches require initialization code that constructs the arrays of procedures that allow the generic operations to invoke the proper procedures. Also note that the code in *testsort* accesses directly into the table because *testsort* actually creates a specific kind of table and so knows what its representation is.

I was disturbed by the large number of kinds of object handles in Appendix B, so I restructured the operations on Tables so that none would return a reference to an element of the table. I replaced the Index and Swap operations on tables and the Compare operation on elements by an IfCompareSwap operation on tables. Appendix C contains the Mesa that resulted from this modification to the structure of the sort program. This example points out that Smalltalk programmers should make an attempt to structure their messages so that a minimum number of different kinds of object handles that require runtime generality will be needed. Another change that was made from Appendix B to Appendix C was to use Mesa variant records for Tables. No LOOPHOLES appear in the code of Appendix C.

Finally, Appendix D was created to show what the Mesa would look like if we decided that no runtime generality was needed for the sort routine. Only one type of table can be sorted by the code in Appendix D. Since this type of table must fit in virtual memory, I also eliminated the generality of having double precision indices into the table.

## Conclusion

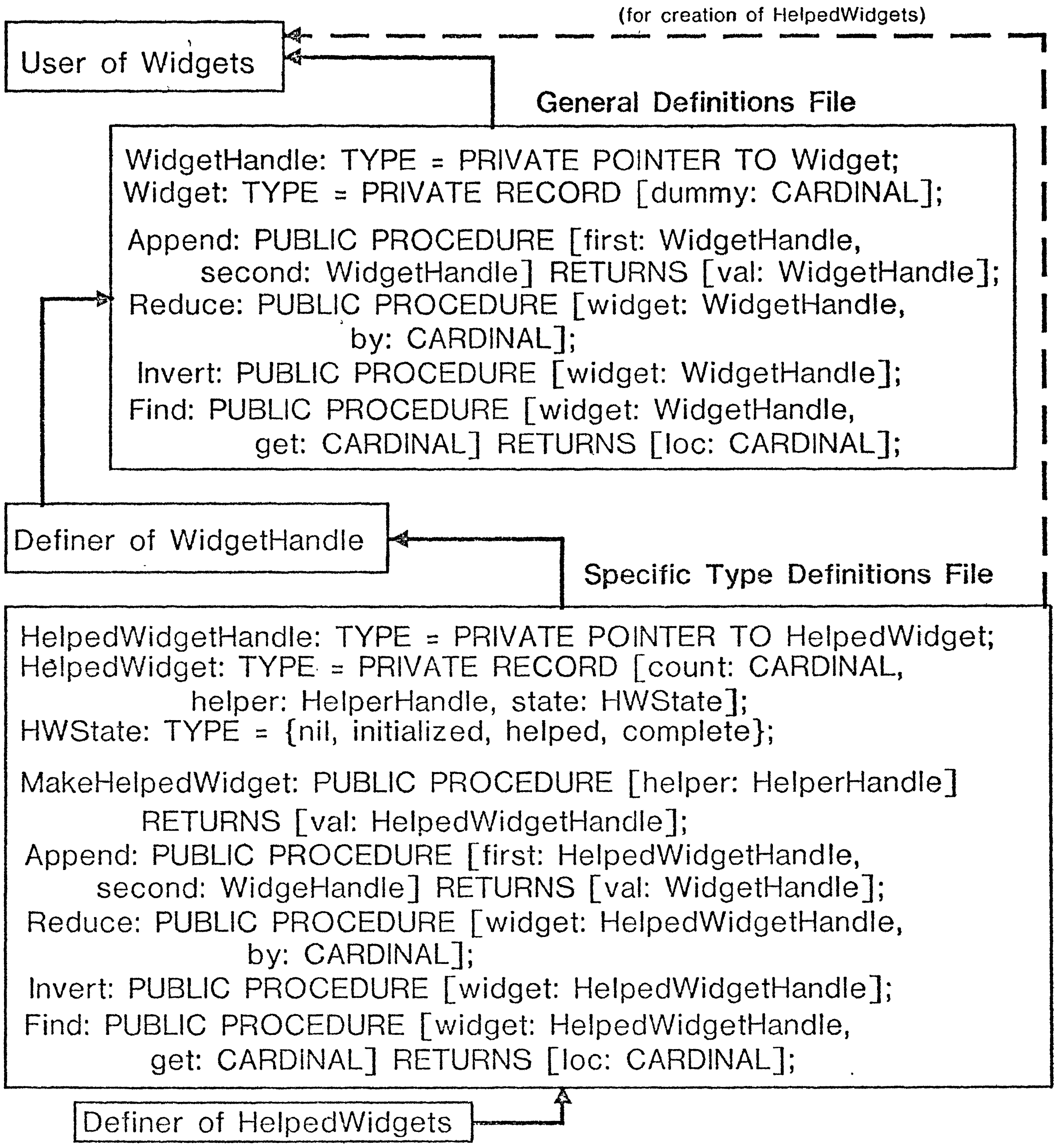
The appendices do not represent recommendations for how the sort routine should be implemented in Janus. The appendices are merely intended to show several different ways of implementing Smalltalk programs in Mesa in the rare cases when a Smalltalk program actually makes use of the possibility of operating on several different representations of the same object.

This memo presents a set of ideas that I think need to be thought about and discussed. Topics that will be covered in later memos are:

1) whether files should be objects,

2) how to translate Smalltalk subclasses into Mesa, and

3) how Mesa could be improved so that the considerations in this memo listed under "harsh realities" would not exist.



```
WidgetHandle: TYPE = PRIVATE POINTER TO Widget;
Widget: TYPE = PRIVATE RECORD [dummy: CARDINAL];

Append: PUBLIC PROCEDURE [first: WidgetHandle,
      second: WidgetHandle] RETURNS [val: WidgetHandle];
Reduce: PUBLIC PROCEDURE [widget: WidgetHandle,
                by: CARDINAL];
Invert: PUBLIC PROCEDURE [widget: WidgetHandle];
Find: PUBLIC PROCEDURE [widget: WidgetHandle,
           get: CARDINAL] RETURNS [loc: CARDINAL];
```

```
HelpedWidgetHandle: TYPE = PRIVATE POINTER TO HelpedWidget;
HelpedWidget: TYPE = PRIVATE RECORD [count: CARDINAL,
              helper: HelperHandle, state: HWState];
HWState: TYPE = {nil, initialized, helped, complete};

MakeHelpedWidget: PUBLIC PROCEDURE [helper: HelperHandle]
          RETURNS [val: HelpedWidgetHandle];
Append: PUBLIC PROCEDURE [first: HelpedWidgetHandle,
      second: WidgeHandle] RETURNS [val: WidgetHandle];
Reduce: PUBLIC PROCEDURE [widget: HelpedWidgetHandle,
                by: CARDINAL];
Invert: PUBLIC PROCEDURE [widget: HelpedWidgetHandle];
Find: PUBLIC PROCEDURE [widget: HelpedWidgetHandle,
          get: CARDINAL] RETURNS [loc: CARDINAL];
```

Figure 1

"basic data structures"
"Version 3.5 6 Sept 77"

# Appendix A

"Array"

```
Class new title: 'Array';
   fields: '';
   asFollows!
```

*Array is an abstract class in the sense that it has no state, and instantiation is consequently not meaningful. However it defines the default message set inherited by its subclasses, notably String, Vector, and UniqueString. Notice that subscripting is not done here, except to handle the exceptional cases such as subscripting by other types as in a◦(1 to: 3).*

## Reading and Writing

```
◦ x
   [⇑x subscripts: self]
◦ x ← val
   [⇑x subscripts: self ← val]
= arg | x
   [self length ≠ arg length⇒ [⇑false]
   for⦂ x to: self length do⦂
      [(self◦x) = (arg◦x)⇒ [] ⇑false]
   ⇑true]
all ← val | i
   [for⦂ i to: self length do⦂
      [self◦i ← val]]
last
   [⇑self◦self length]
last ← val
   [⇑self◦self length ← val]
length  [user notify: 'message not understood.']
```

## Copying and Altering

```
+ arg [⇑self concat: arg]
concat: arg | x
   [x ← self species new: self length + arg length.
   self copyto: x.
   arg copyto: (Stream new of: x from: self length+1 to: x length).
   ⇑x]
copy
   [⇑self copyto: (self species new: self length)]
copy: a to: b [⇑(self◦(a to: b)) copy]
copyto: t | i s me
   [s ← t asStream.
   me ← self asStream.
   for⦂ i to: self length do⦂      "general code wont stop at false"
      [s next ← me next]
   ⇑t]
delete: obj | s each
   [s ← (self species new: self length) asStream.
   for⦂ each from: self do⦂
      [obj=each⇒[] s next← each]
   ⇑ s contents]
grow
   [⇑self copyto: (self species new: (4 max: self length*2))]
growby: n
   [⇑self copyto: (self species new: self length+n)]
replace: a to: b by: s | x xs
   [x ← self species new: self length+s length -(1+b-a).
   xs ← x asStream.
   self◦(1 to: a-1) copyto: xs.
   s copyto: xs.
   self◦(b+1 to: self length) copyto: xs.
   ⇑x]
swap: i with: j | t
   [t ← self◦i. self◦i ← self◦j. self◦j ← t]
```

## Sorting and Searching

```
find: x | i
   [for⦂ i to: self length do⦂
```

```
        [self◦i=x⇒ [⇑i]].
    ⇑0]
 findnon: x | i
    [for⦂ i to: self length do⦂
        [self◦i≠x⇒ [⇑i]].
    ⇑0]
has: x
    [⇑0≠(self find: x)]
insertNonDescending: x     | a c lo mid hi  "self is assumed to be sorted"
    [
    c ← (a ← self species new: (hi ← self length + (lo ← 1))) asStream.
    while⦂ lo < hi do⦂ [self◦(mid←lo+hi/2) > x⇒ [hi ← mid] lo ← mid+1].
    self◦(1 to: hi-1) copyto: c. c next ← x. self◦(hi to: self length) copyto: c.
    ⇑a
    ]
insertSorted: x     | a c lo mid hi     "self is assumed to be sorted"
    [c ← (a ← self species new: (hi ← self length + (lo ← 1))) asStream.
    while⦂ lo < hi do⦂     "binary search"
        [self◦(mid←lo+hi/2) > x⇒[hi ← mid]  lo ← mid+1].
    self◦(1 to: hi-1) copyto: c. c next ← x. self◦(hi to: self length) copyto: c.
    ⇑a]
permutationToSort
    ["Return a Vector, permutation, such that self◦permutation is sorted nondescending.  Do not
alter self."
    ⇑((self◦((1 to: self length) copy)) sort: 1 to: self length) map.]
promote: t | n
    [n ← self find: t. n=0⇒ []
    self◦(n to: 2 by: ¯1) ← self◦(n-1 to: 1 by: ¯1).
    self◦1 ← t]
reverse
    [⇑Substring new data: self map: (self length to: 1 by: ¯1)]
sort
    ["Permute my elements so they are sorted nondescending.  Note: if I am a substring, only my
map will be permuted.  In certain situations, this may not be what you expect."
    self sort: 1 to: self length.]
sort: i to: j | di dij dj tt ij k l n
    ["Sort elements i through j of self to be nondescending."

    "The prefix d means the data at."
    (n←j+1-i)≤1⇒ ["Nothing to sort."]
    "Sort di,dj."
    di ← self◦i. dj ← self◦j.
    [di>dj⇒ [self swap: i with: j. tt←di. di←dj. dj←tt]].
    n=2⇒ ["They are the only two elements."]
    ij ← (i+j) lshift: ¯1. "ij is the midpoint of i and j."
    "Sort di,dij,dj.  Make dij be their median."
    dij ← self◦ij.
    [di>dij⇒ [self swap: i with: ij. dij←di] dj<dij⇒ [self swap: j with: ij. dij←dj]].
    n=3⇒ ["They are the only three elements."]
    "Find k>i and l<j such that dk,dij,dl are in reverse order.  Swap k and l.  Repeat this procedure
until j and k pass each other."
    k ← i. l ← j.
    while⦂
        [
        while⦂ self◦(l←l-1) > dij do⦂ [].
        while⦂ self◦(k←k+1) < dij do⦂ [].
        k≤l
        ]
    do⦂
        [self swap: k with: l].
    "Now l<k (either 1 or 2 less), and di through dl are all less than dk through dj.  Sort those two
segments."
    self sort: i to: l.
    self sort: k to: j.]
```

## Mapping

```
asStream
    [⇑Stream new of: self]
subscripts: x          "subarrays"
    [⇑Substring new data: x map: self]
```

# Appendix B

```
-- File: testsort.mesa Last edited: November 4, 1977  5:49 PM  By: Peter Bishop

DIRECTORY
    IODefs: FROM "iodefs",
    SystemDefs: FROM "systemdefs",
    StringDefs: FROM "stringdefs",
    MyTableDefs: FROM "mytabledefs",
    SortDefs: FROM "sortdefs";

testsort:PROGRAM
        IMPORTS SortDefs, IODefs, MyTableDefs, SystemDefs, StringDefs =
BEGIN OPEN IODefs, SystemDefs, StringDefs, SortDefs;

--This program merely tests the sort routine. It allows the user to type in a table to be sorted,
   sorts the table, and prints out and frees the storage for the sorted table.

size, i: CARDINAL;
p: POINTER TO MyTableDefs.Entry;
name: STRING = "                                   ";
tab: MyTableDefs.MyTableHandle;

--testsort is the control program for the configuration, so initialize the sort routine.
START SortProg;
START MyTableDefs.TableProg;

    DO  --infinite loop containing a STOP allows the program to be restarted.
    WriteString ["Enter size of table: "];
    size ← ReadDecimal [];
    WriteLine [""];
    tab ← MyTableDefs.MakeMyTable [size];
    FOR i IN [1..size] DO  --fill in the entries in the table from the keyboard.
        WriteString ["name: "];
        ReadID [name];
        p ← @tab.t[i];
        p.name ← AllocateHeapString[name.length];
        p.name.length ← 0;
        AppendString[p.name, name];
        WriteString [" value: "];
        p.value ← ReadDecimal [];
        WriteLine [""];
        ENDLOOP;

    WriteLine ["Sorting"];
    Sort[LOOPHOLE[tab]];  --use the sort routine.
        --A LOOPHOLE is needed at this point because tab is a specific kind of table, but Sort
        takes a general kind of table and because Mesa is not aware of the relationship between
        these two types.

    FOR i IN [1..size] DO  --type out the sorted table and free the storage.
        p ← @tab.t[i];
        WriteString[p.name];
        FreeHeapString[p.name];
        WriteString[" "];
        WriteDecimal[p.value];
        WriteLine[""];
        ENDLOOP;
    FreeHeapNode[tab];
    STOP;
    ENDLOOP;
END.
```

```
-- File: SortDefs.mesa Last edited: November 10, 1977  11:34 AM  By: Peter Bishop

DIRECTORY
   DoubleDefs: FROM "doubledefs";

SortDefs: DEFINITIONS =
BEGIN OPEN DoubleDefs;

   SortProg: PROGRAM;

   TableHandle: TYPE = POINTER TO Table;
   Sort: PROCEDURE [table: TableHandle];
   Index: PROCEDURE [table: TableHandle, subscript: LongCARDINAL]
            RETURNS [element: ElementHandle];
   Swap: PROCEDURE [table: TableHandle, first: LongCARDINAL,
            second: LongCARDINAL];
   Range: PROCEDURE [table: TableHandle]
            RETURNS [low: LongCARDINAL, high: LongCARDINAL];
   SubscriptOutOfBounds: ERROR;

   ElementHandle: TYPE = PRIVATE RECORD [
      var: SELECT type: ElementType FROM
         string => [element: STRING],
         integer => [element: INTEGER],
         ENDCASE];
   ElementType: TYPE = {string, integer};
   SetElementTypeArray: PROCEDURE [
      type: ElementType,
      compproc: PROCEDURE [first: ElementHandle, second: ElementHandle]
         RETURNS [comp: Comparison]];
   Compare: PROCEDURE [first: ElementHandle, second: ElementHandle]
      RETURNS [comp: Comparison];

   Table: PRIVATE TYPE = RECORD [
      type: POINTER TO TableProcedures,
      tptr: POINTER TO UNSPECIFIED];
   TableProcedures: PRIVATE TYPE = RECORD [
      Index: PROCEDURE [ TableHandle, LongCARDINAL]
         RETURNS [ElementHandle],
      Swap: PROCEDURE [ TableHandle, LongCARDINAL, LongCARDINAL],
      Range: PROCEDURE [TableHandle]
         RETURNS [LongCARDINAL, LongCARDINAL]];
END.
```

```
-- File: Sort.mesa Last edited: November 4, 1977  2:32 PM  By: Peter Bishop

DIRECTORY
    DoubleDefs: FROM "doubledefs",
    SortDefs: FROM "sortdefs";

SortProg: PROGRAM IMPORTS DoubleDefs
    EXPORTS SortDefs
    SHARES SortDefs =
BEGIN OPEN DoubleDefs, SortDefs;

Long1: LongCARDINAL = [highbits:0, lowbits:1];
Long2: LongCARDINAL = [highbits:0, lowbits:2];
Long3: LongCARDINAL = [highbits:0, lowbits:3];

Sort: PUBLIC PROCEDURE [table: TableHandle ] =
    BEGIN
    x,y: LongCARDINAL;
    [x, y] ← Range [table];
    SortRange[table, x, y];
    END;

SortRange: PROCEDURE [
    table: TableHandle, i: LongCARDINAL, j: LongCARDINAL] =
    BEGIN
    n, ij, l, k: LongCARDINAL;
    di, dj, dij, tt: ElementHandle;

    --Sort elements i through j of table to be nondescending.
    --The prefix d means the data at.
    --The basic technique is to choose three elements from table
    --at "random" (first, last, middle), sort these three elements,
    --choose the middle element and then divide table into two parts,
    --the part all of whose elements are less than this middle value,
    --and the other part having only elements larger than the middle value.
    --SortRange is then called recursively on these two portions of the table.

    n ← DSub [DInc [j], i];
    IF n.highbits=0 AND n.lowbits<=1 THEN RETURN;  --Nothing to sort.
    di ← Index [table, i];
    dj ← Index [table, j];
        --Sort di, dj.
    IF Compare [di, dj] = greater THEN
        BEGIN
        Swap [table, i, j];
        tt ← di;
        di ← dj;
        dj ← tt;
        END;
    IF n = Long2 THEN RETURN;  --There are only two elements.

    [ij, ] ← DDivide [DAdd [ i, j], Long2];  --ij is the midpoint of i and j.
    dij ← Index [table, ij];
    --Sort di, dij, dj so they are in nondescending order.
    --After this, we will only need to have dij valid - di or dj may be invalid.
    IF Compare [di, dij] = greater THEN
        BEGIN  --Now we know that dij<di<=dj, so swap di and dij.
        Swap [table, i, ij];
        dij ← di;  --we are not swapping di and dij properly
                       --because di need not remain valid.
        END
    ELSE IF Compare [dj, dij] = less THEN
        BEGIN  --Now we know that di<=dj<dij, so swap dj and dij.
        Swap [table, j, ij];
        dij ← dj;  --we are not swapping dj and dij properly
                       --because dj need not remain valid.
        END;
    IF n = Long3 THEN RETURN;  --di, dij, and dj are the only three elements.
    k ← i;
    l ← j;
    --Find k>i and l<j such that dk, dij, dl are in reverse order. Swap k and l.
    --Repeat until k and l pass each other, at which point k and l are approximately equal and
       specify the dividing line in the table between the elements that are less than or equal to
       dij and those that are greater than or equal to dij. SortRange is then called recursively on
       these two portions of the table.
        DO
        WHILE Compare [Index [table, (l←DSub [l, Long1])], dij] >= equal
            DO ENDLOOP;  --Find l such that dl<dij.
        WHILE Compare [Index [table, (k ← DInc [k])], dij] <= equal
            DO ENDLOOP;  --Find k such that dij<dk.
        IF DCompare [k, l] = greater THEN EXIT; --Do loop while k<=l.
        Swap [table, k, l];  --k will never equal l because dl<dij<dk.
        ENDLOOP;
    --Now l<k and di through dl are less than or equal to dij while dk through dj are all greater
    than or equal to dij. l+1 may be less than k, but only if the elements between dl and dk are
    all equal to dij. Fortunately, all elements equal to dij will cluster at exactly this point in the
    table anyway in one of three ways:
        --1)l+1<k and the elements between are equal to dij.
        --2) elements equal to dij within di through dl are larger than any other elements in that
              range and so will be placed at l.
        --3) elements equal to dij within dk through dj are less than any other elements in that
              range and so will be placed at k.
    --Thus it is alright for elements equal to dij to be sprinkled all over the table at this point.

    SortRange [table, i, l];
    SortRange [table, k, j];

    RETURN;
    END;

SubscriptOutOfBounds: PUBLIC ERROR = CODE;

Index: PUBLIC PROCEDURE [table: TableHandle, subscript: LongCARDINAL]
    RETURNS [element: ElementHandle] =
    BEGIN
    element ← table.type.Index [table, subscript];
    END;

Range: PUBLIC PROCEDURE [table: TableHandle]
    RETURNS [low: LongCARDINAL, high: LongCARDINAL] =
    BEGIN
    [low, high] ← table.type.Range [table];
    END;

Swap: PUBLIC PROCEDURE [
    table: TableHandle, first: LongCARDINAL, second: LongCARDINAL] =
    BEGIN
    table.type.Swap[table, first, second];
    END;

Compare: PUBLIC PROCEDURE [first: ElementHandle, second: ElementHandle]
    RETURNS [comp: Comparison] =
    BEGIN
    comp ← Comparearray [first.type].proc [first, second];
    END;

--This next procedure is used to initialize the array that contains the specific Compare
        procedures for specific data types. This procedure is used mainly during initialization, but
        may be called at any time to define an undefined ElementType.
SetElementTypeArray: PUBLIC PROCEDURE [
    type:ElementType,
    compproc: PROCEDURE [first: ElementHandle, second: ElementHandle]
        RETURNS [comp: Comparison]] =
    BEGIN
    IF Comparearray[type].set THEN ERROR duplicateTypeDefinition[];
    Comparearray [type].proc ← compproc;
    Comparearray [type].set ← TRUE;
    END;

Comparearray: ARRAY ElementType OF RECORD [
    set: BOOLEAN,
    proc: PROCEDURE [ElementHandle, ElementHandle] RETURNS [Comparison]];

duplicateTypeDefinition: ERROR;

t: ElementType;

FOR t IN ElementType DO  --The Comparearray must be initialized to nil.
    Comparearray[t].set ← FALSE;
    ENDLOOP;

END.
```

```
-- File: mytabledefs.mesa Last edited: November 4, 1977  5:29 PM  By: Peter Bishop

DIRECTORY
    SortDefs: FROM "sortdefs";

MyTableDefs:DEFINITIONS
            SHARES SortDefs =
BEGIN OPEN SortDefs;
    MyTableHandle: TYPE = POINTER TO MyTable;
    MyTable: TYPE = RECORD[
        type: POINTER TO TableProcedures,
        size: CARDINAL,
        t: ARRAY [1..1) OF Entry];
    Entry: TYPE = RECORD[name: STRING, value: INTEGER];
    MyElement: TYPE = string ElementHandle;

    MakeMyTable: PROCEDURE [size: CARDINAL] RETURNS [val: MyTableHandle];
        --This procedure constructs tables of this specific type. Procedures of this form are used
          to create objects rather than defining "initialize" operations on objects.

    TableProg: PROGRAM;

END.
```

```
-- File: table.mesa Last edited: November 10, 1977  11:38 AM  By: Peter Bishop

DIRECTORY
    SortDefs: FROM "sortdefs",
    SystemDefs: FROM "systemdefs",
    MyTableDefs: FROM "mytabledefs",
    DoubleDefs: FROM "doubledefs";

TableProg: PROGRAM IMPORTS SortDefs, SystemDefs
            EXPORTS MyTableDefs
            SHARES SortDefs =
BEGIN OPEN SortDefs, DoubleDefs, MyTableDefs;

MyTableProcs: TableProcedures;

MakeMyTable: PUBLIC PROCEDURE [size: CARDINAL]
    RETURNS [val: MyTableHandle] =
    BEGIN
    val ← SystemDefs.AllocateHeapNode[size*SIZE[Entry] + SIZE[MyTable]];
    val.type ← @MyTableProcs;
    val.size ← size;
    END;

Index: PROCEDURE [table: MyTableHandle, subscript: LongCARDINAL]
    RETURNS [val: MyElement] =
    BEGIN
    p: POINTER TO Entry;
    i: CARDINAL;
    i← Shorten[table.size, subscript];
    p ← @table.t[i];
    val ← [string [p.name]];
    END;

Swap: PROCEDURE [table: MyTableHandle, first, second: LongCARDINAL] =
    BEGIN
    f,s: POINTER TO Entry;
    temp:Entry;

    f ← @table.t[Shorten[table.size, first]];
    s ← @table.t[Shorten[table.size, second]];

    temp ← f↑; --Swap first and second.
    f↑ ← s↑;
    s↑ ← temp;
    END;

Range: PROCEDURE [table: MyTableHandle]
            RETURNS [low, high: LongCARDINAL] =
    BEGIN
    low.highbits ← high.highbits ← 0;
    low.lowbits ← 1;
    high.lowbits ← table.size;
    END;

Shorten: PRIVATE PROCEDURE [size: CARDINAL, subscript: LongCARDINAL]
    RETURNS [i: CARDINAL] =
    BEGIN
    --This procedure checks subscripts to make sure that they are within the table and converts
    them to single precision.

    i ← subscript.lowbits;
    IF subscript.highbits # 0 OR i > size OR i = 0 THEN
        ERROR SubscriptOutOfBounds [];
    END;

Compare: PROCEDURE [first, second: ElementHandle]
    RETURNS [comp: Comparison] =
    BEGIN
    l, i: CARDINAL;
    f, s: string ElementHandle;
    WITH x: first SELECT FROM
        string => f ← x;
        ENDCASE => ERROR;
    WITH x: second SELECT FROM
        string => s ← x;
        ENDCASE => ERROR;
    l ← MIN[f.element.length, s.element.length];
    comp ← equal;
    FOR i IN [0..l) DO
        IF f.element[i] # s.element[i] THEN
            BEGIN
            comp ← (IF f.element[i] > s.element[i] THEN greater ELSE less);
            EXIT;
            END;
        ENDLOOP;
    IF comp = equal THEN
        IF f.element.length > l THEN comp ← greater
        ELSE IF s.element.length > l THEN comp ← less;
    END;

SetElementTypeArray[type: string, compproc: Compare];
    --This procedure call initializes the Comparearray that allows the generic Compare procedure
        to invoke this Compare procedure on this type of Element.

MyTableProcs.Index ← LOOPHOLE [Index];
MyTableProcs.Range ← LOOPHOLE [Range];
MyTableProcs.Swap ← LOOPHOLE [Swap];
    --These statements initialize the array of procedures that define the operations on tables. The
        generic operations invoke these operations because Tables contain a pointer to this array
        of procedures.
END.
```

# Appendix C

```
-- File: testsort2.mesa Last edited: November 8, 1977  11:26 AM  By: Peter Bishop

DIRECTORY
    IODefs: FROM "iodefs",
    SystemDefs: FROM "systemdefs",
    StringDefs: FROM "stringdefs",
    SortDefs: FROM "sortdefs2";

testsort2:PROGRAM
        IMPORTS SortDefs, IODefs, SystemDefs, StringDefs
        SHARES SortDefs =
BEGIN OPEN IODefs, SystemDefs, StringDefs, SortDefs;

size, i: CARDINAL;
p: POINTER TO StringIntEntry;
name: STRING = "                              ";
tab: POINTER TO stringInt Table;

START SortProg;
START StringIntTableProg;

    DO
    WriteString ["Enter size of table: "];
    size ← ReadDecimal [];
    WriteLine [""];
    tab ← MakeMyTable [size];
    FOR i IN [1..size] DO
        WriteString ["name: "];
        ReadID [name];
        p ← @tab.t[i];
        p.name ← AllocateHeapString[name.length];
        p.name.length ← 0;
        AppendString[p.name, name];
        WriteString [" value: "];
        p.value ← ReadDecimal [];
        WriteLine [""];
        ENDLOOP;

    --Note that in this implementation, LOOPHOLES are not needed either to  invoke the Sort
    procedure or to modify the fields of a table when the code is sure it is dealing with a
    specific type of object.

    WriteLine ["Sorting"];
    Sort[tab];

    FOR i IN [1..size] DO
        p ← @tab.t[i];
        WriteString[p.name];
        FreeHeapString[p.name];
        WriteString[" "];
        WriteDecimal[p.value];
        WriteLine[""];
        ENDLOOP;
    FreeHeapNode[tab];
    STOP;
    ENDLOOP;

END.
```

```
-- File: SortDefs2.mesa Last edited: November 8, 1977  10:58 AM  By: Peter Bishop

DIRECTORY
   DoubleDefs: FROM "doubledefs";

SortDefs: DEFINITIONS =
BEGIN OPEN DoubleDefs;

   SortProg: PROGRAM;

   --This sort procedure is written in terms of the IfCompareSwap operation on tables, which
   does not require the sort routine to have ElementHandles, so only one type of object handle
   is declared in this definitions file: TableHandle.

   TableHandle: TYPE = POINTER TO Table;
   Sort: PROCEDURE [table: TableHandle];
   --The IfCompareSwap procedure compares the first element in the table to the second
      element in the table and if the condition specified by CompareOp is satisfied, the elements
      are swapped. In any case the procedure returns the result of the actual comparison unless
      comp is all, in which case the comparison need not be performed at all and equal is
      returned.
   IfCompareSwap: PROCEDURE [
      table: TableHandle, first, second: LongCARDINAL, comp: CompareOp]
      RETURNS [val: Comparison];
   Range: PROCEDURE [table: TableHandle]
            RETURNS [low: LongCARDINAL, high: LongCARDINAL];
   SubscriptOutOfBounds: ERROR;

   --This procedure allows types to be initialized so the generic procedure can find the
   appropriate specific procedure to call for each type.
   SetTableTypeArray: PROCEDURE [
      type: TableType,
      procs: TableProcedures];

   TableType: TYPE = {stringInt};
   --Tables are declared to be a variant record this time so that LOOPHOLES will not be
      necessary. Although there is only one variant, that is only because this toy problem does
      not actually have more than one type of table. The whole point of this exercise is to allow
      several types of tables, so this implementation strategy would not be used unless there
      actually were several different types of tables.
   Table: PRIVATE TYPE = RECORD [var:
      SELECT type: TableType FROM
         stringInt => [
            size: CARDINAL,
            t: ARRAY [1..1) OF StringIntEntry],
         ENDCASE];
   StringIntEntry: TYPE = RECORD [name: STRING, value: INTEGER];
   TableProcedures: PRIVATE TYPE = RECORD [
      IfCompareSwap: PROCEDURE [
         TableHandle, LongCARDINAL, LongCARDINAL, CompareOp]
         RETURNS [Comparison],
      Range: PROCEDURE [TableHandle]
         RETURNS [LongCARDINAL, LongCARDINAL]];

   StringIntTableProg: PROGRAM;
   MakeMyTable: PROCEDURE [size: CARDINAL]
      RETURNS [val: POINTER TO stringInt Table];

   CompareOp: TYPE = [0..8);
```

```
    greaterthan: CompareOp = 4;
    equalto: CompareOp = 2;
    lessthan: CompareOp = 1;
    all: CompareOp = 7;
    none: CompareOp = 0;

END.
```

```
-- File: Sort2.mesa Last edited: November 10, 1977  10:13 AM  By: Peter Bishop

DIRECTORY
    DoubleDefs: FROM "doubledefs",
    SortDefs: FROM "sortdefs2";

SortProg: PROGRAM IMPORTS DoubleDefs
    EXPORTS SortDefs
    SHARES SortDefs =
BEGIN OPEN DoubleDefs, SortDefs;

--This sort procedure is written in terms of the IfCompareSwap operation on tables rather than
   the Index and Swap operations on tables and the Compare operation on elements.

Long1: LongCARDINAL = [highbits:0, lowbits:1];
Long2: LongCARDINAL = [highbits:0, lowbits:2];
Long3: LongCARDINAL = [highbits:0, lowbits:3];

Sort: PUBLIC PROCEDURE [table: TableHandle ] =
    BEGIN
    x,y: LongCARDINAL;
    [x, y] ← Range [table];
    SortRange[table, x, y];
    END;

SortRange: PROCEDURE [
    table: TableHandle, i: LongCARDINAL, j: LongCARDINAL] =
    BEGIN
    n, ij, l, k: LongCARDINAL;

--Sort elements i through j of table to be nondescending.

    --The prefix d means the data at a place in the table.

    --The basic technique is to choose three elements from table
    --at "random" (first, last, middle), sort these three elements,
    --choose the middle element and then divide table into two parts,
    --the part all of whose elements are less than this middle value,
    --and the other part having only elements larger than the middle value.
    --SortRange is then called recursively on these two portions of the table.

    n ← DSub [DInc [j], i];
    IF n.highbits=0 AND n.lowbits<=1 THEN RETURN;  --Nothing to sort.
        --Sort i, j.
    [] ← IfCompareSwap [table, i, j, greaterthan];
    IF n = Long2 THEN RETURN;  --There are only two elements.

    [ij, ] ← DDivide [DAdd [ i, j], Long2];  --ij is the midpoint of i and j.
    --Sort i, ij, j so they are in nondescending order.
        IF IfCompareSwap [table, i, ij, greaterthan] >= equal
            THEN NULL  --We now know that di<=dij<=dj, so they are sorted
                            -- do nothing.
            ELSE [] ← IfCompareSwap [table, j, ij, lessthan]; -- Now di<=dij<dj.
    IF n = Long3 THEN RETURN;  --di, dij, and dj are the only three elements.
    k ← i;
    l ← j;
    --Find k>i and l<j such that dk>dij>dl (i.e. are in reverse order). Swap k and l.
    --Repeat until k and l pass each other, at which point k and l are approximately equal and
```

```
        specify the dividing line in the table between the elements that are less than or equal to
        dij and those that are greater than or equal to dij. SortRange is then called recursively on
        these two portions of the table.
    --Note that elements that are equal to dij are not moved.
        DO
        WHILE IfCompareSwap [ table, l←DSub [l, Long1], ij, none] # less
                        --No swap was performed, only a comparison.
            DO ENDLOOP;  --Find l such that dl<dij.
        WHILE IfCompareSwap [table, (k ← DInc [k]), ij, none] # greater
            DO ENDLOOP;  --Find k such that dij<dk.
        IF DCompare [k, l] = greater THEN EXIT; --Do loop while k<=l.
        [] ← IfCompareSwap [table, k, l, all];
                    --Doesn't compare, just swaps.
                    --k will never equal l because dl<dij<dk.
        ENDLOOP;
    --Now l<k and di through dl are less than or equal to dij while dk through dj are all greater
    than or equal to dij. l+1 may be less than k, but only if the elements between dl and dk are
    all equal to dij. Fortunately, all elements equal to dij will cluster at exactly this point in the
    table anyway in one of three ways:
        --1)l+1<k and the elements between are equal to dij.
        --2) elements equal to dij within di through dl are larger than any other elements in that
             range and so will be placed at l.
        --3) elements equal to dij within dk through dj are less than any other elements in that
             range and so will be placed at k.
    --Thus it is alright for elements equal to dij to be sprinkled all over the table at this point.

    SortRange [table, i, l];
    SortRange [table, k, j];

    RETURN;
    END;

SubscriptOutOfBounds: PUBLIC ERROR = CODE;

Range: PUBLIC PROCEDURE [table: TableHandle]
        RETURNS [low: LongCARDINAL, high: LongCARDINAL] =
    BEGIN
    [low, high] ← TableArray[table.type].procs.Range [table];
    END;

IfCompareSwap: PUBLIC PROCEDURE [
    table: TableHandle, first, second: LongCARDINAL, comp: CompareOp]
    RETURNS [val: Comparison] =
    BEGIN
    val ← TableArray[table.type].procs.IfCompareSwap
        [table, first, second, comp];
    END;

SetTableTypeArray: PUBLIC PROCEDURE [
    type:TableType, procs: TableProcedures] =
    BEGIN
    IF TableArray[type].set THEN ERROR duplicateTypeDefinition [];
    TableArray [type].procs ← procs;
    TableArray [type].set ← TRUE;
    END;

TableArray: ARRAY TableType OF RECORD [
    set: BOOLEAN,
    procs: TableProcedures];

duplicateTypeDefinition: ERROR;

t: TableType;

--Initialize the TableArray to all types undefined. Each type can only be defined once.
FOR t IN TableType DO
    TableArray[t].set ← FALSE;
    ENDLOOP;

END.
```

```
-- File: stringinttableprog.mesa Last edited: November 10, 1977  10:05 AM  By: Peter Bishop

DIRECTORY
    SortDefs: FROM "sortdefs2",
    SystemDefs: FROM "systemdefs",
    InlineDefs: FROM "inlinedefs",
    DoubleDefs: FROM "doubledefs";

StringIntTableProg: PROGRAM IMPORTS SortDefs, SystemDefs
            EXPORTS SortDefs
            SHARES SortDefs =
BEGIN OPEN SortDefs, DoubleDefs;

--These are the type specific delcarations that were in mytabledefs in Appendix B.
MyTable:TYPE = stringInt Table;
MyTableHandle: TYPE = POINTER TO MyTable;

MakeMyTable: PUBLIC PROCEDURE [size: CARDINAL]
    RETURNS [val: MyTableHandle] =
    BEGIN
    val ← SystemDefs.AllocateHeapNode[
        size*SIZE[StringIntEntry] + SIZE[MyTable]];
    val↑ ← [stringInt [size, ]];
    END;

SIRange: PRIVATE PROCEDURE [table: TableHandle]
                RETURNS [low, high: LongCARDINAL] =
    BEGIN
    WITH t: table SELECT FROM
        stringInt => BEGIN
            low.highbits ← high.highbits ← 0;
            low.lowbits ← 1;
            high.lowbits ← t.size;
            END;
        ENDCASE => ERROR;
    END;

Shorten: PRIVATE PROCEDURE [size: CARDINAL, subscript: LongCARDINAL]
    RETURNS [i: CARDINAL] =
    BEGIN
    i ← subscript.lowbits;
    IF subscript.highbits # 0 OR i > size OR i = 0 THEN
        ERROR SubscriptOutOfBounds [];
    END;

comparray: ARRAY Comparison OF CompareOp =
    [lessthan, equalto, greaterthan];
    --this is needed because sets are not supported in Mesa.

SIIfCompareSwap: PRIVATE PROCEDURE [
    table: TableHandle, first, second: LongCARDINAL, comp: CompareOp]
    RETURNS [val: Comparison] =
    BEGIN

    --We see here one of the deficiencies of Mesa: this procedure takes a general TableHandle
       as the first argument even though the generic procedure will never call it with anything but
       MyTableHandle. Fortunately this is not a serious performance question, but is merely an
       annoyance.
    t: MyTableHandle;
    l, i: CARDINAL;
    f, s: POINTER TO StringIntEntry;
    temp: StringIntEntry;
    WITH tab: table SELECT FROM  --perform the redundant type check in a way that retains full
                                                         compiler type checking.
        stringInt => t ← @tab;
        ENDCASE => ERROR;
    f ← @t.t[Shorten[t.size, first]];
    s ← @t.t[Shorten[t.size, second]];
    val ← equal;
    IF comp # all THEN
        BEGIN
        l ← MIN[f.name.length, s.name.length];
        FOR i IN [0..l) DO
            IF f.name[i] # s.name[i] THEN
                BEGIN
                val ← (IF f.name[i] > s.name[i] THEN greater ELSE less);
                EXIT;
                END;
            ENDLOOP;
        IF val = equal THEN
            IF f.name.length > l THEN val ← greater
            ELSE IF s.name.length > l THEN val ← less;
        IF InlineDefs.BITAND[comp, comparray[val]] = 0 THEN RETURN;
                --the BITAND is needed because Mesa does not support sets.
                --would like to write: IF val NOT IN comp THEN RETURN;
        END;
    temp ← f↑;  --do the swap.
    f↑ ← s↑;
    s↑ ← temp;
    END;

--Initialize the procedure array for this type of table.
SetTableTypeArray[type: stringInt, procs: [SIIfCompareSwap, SIRange]];

END.
```

# Appendix D

```
-- File: testsort6.mesa Last edited: November 14, 1977  3:07 PM  By: Peter Bishop

DIRECTORY
    IODefs: FROM "iodefs",
    SystemDefs: FROM "systemdefs",
    StringDefs: FROM "stringdefs",
    SortDefs: FROM "sortdefs6";

testsort:PROGRAM
        IMPORTS SortDefs, IODefs, SystemDefs, StringDefs
        SHARES SortDefs =
BEGIN OPEN IODefs, SystemDefs, StringDefs, SortDefs;

--This implementation has only one type of table and uses only single precision subscripts for
   the table because the representation of the table cannot have more than 65000 entries
   anyway.

size, i: CARDINAL;
p: POINTER TO StringIntEntry;
name: STRING = "                              ";
tab: POINTER TO Table;

START SortProg;
START StringIntTableProg;

    DO
    WriteString ["Enter size of table: "];
    size ← ReadDecimal [];
    WriteLine [""];
    tab ← MakeMyTable [size];
    FOR i IN [1..size] DO
        WriteString ["name: "];
        ReadID [name];
        p ← @tab.t[i];
        p.name ← AllocateHeapString[name.length];
        p.name.length ← 0;
        AppendString[p.name, name];
        WriteString [" value: "];
        p.value ← ReadDecimal [];
        WriteLine [""];
        ENDLOOP;

    --Note that this implementation, like the one that used variant records for tables does not
       require any LOOPHOLES.
    WriteLine ["Sorting"];
    Sort[tab];

    FOR i IN [1..size] DO
        p ← @tab.t[i];
        WriteString[p.name];
        FreeHeapString[p.name];
        WriteString[" "];
        WriteDecimal[p.value];
        WriteLine[""];
        ENDLOOP;
    FreeHeapNode[tab];
    STOP;
    ENDLOOP;
END.
```

```
-- File: SortDefs6.mesa Last edited: November 14, 1977  1:45 PM  By: Peter Bishop

DIRECTORY
   DoubleDefs: FROM "doubledefs";

SortDefs: DEFINITIONS =
BEGIN OPEN DoubleDefs;

   --Note the similarity of this definitions file and sortdefs2.mesa in Appendix C. This definitions
     file does not contain SetTableTypeArray, but that is the major difference.

   SortProg: PROGRAM;

   TableHandle: TYPE = POINTER TO Table;
   Sort: PROCEDURE [table: TableHandle];
   IfCompareSwap: PROCEDURE [
      table: TableHandle, first, second: CARDINAL, comp: CompareOp]
      RETURNS [val: Comparison];
   Range: PROCEDURE [table: TableHandle]
            RETURNS [low: CARDINAL, high: CARDINAL];
   SubscriptOutOfBounds: ERROR;

   Table: PRIVATE TYPE = RECORD [
      size: CARDINAL, t: ARRAY [1..1) OF StringIntEntry];
   StringIntEntry: TYPE = RECORD [name: STRING, value: INTEGER];

   StringIntTableProg: PROGRAM;
   MakeMyTable: PROCEDURE [size: CARDINAL]
      RETURNS [val: POINTER TO Table];

   CompareOp: TYPE = [0..8);
   greaterthan: CompareOp = 4;
   equalto: CompareOp = 2;
   lessthan: CompareOp = 1;
   all: CompareOp = 7;
   none: CompareOp = 0;

END.
```

```
-- File: Sort6.mesa Last edited: November 14, 1977  2:08 PM  By: Peter Bishop

DIRECTORY
    DoubleDefs: FROM "doubledefs",
    SortDefs: FROM "sortdefs6";

SortProg: PROGRAM IMPORTS SortDefs
    EXPORTS SortDefs
    SHARES SortDefs =
BEGIN OPEN DoubleDefs, SortDefs;

--This file is rather different from sort2.mesa in Appendix C because it does not contain
   definitions of the generic procedures since generic procedures are only needed if there are
   many possible implementations of tables.

Sort: PUBLIC PROCEDURE [table: TableHandle ] =
    BEGIN
    x,y: CARDINAL;
    [x, y] ← Range [table];
    SortRange[table, x, y];
    END;

SortRange: PROCEDURE [
    table: TableHandle, i: CARDINAL, j: CARDINAL] =
    BEGIN
    n, ij, l, k: CARDINAL;

--Sort elements i through j of table to be nondescending.

    --The prefix d means the data at a place in the table.

    --The basic technique is to choose three elements from table
    --at "random" (first, last, middle), sort these three elements,
    --choose the middle element and then divide table into two parts,
    --the part all of whose elements are less than this middle value,
    --and the other part having only elements larger than the middle value.
    --SortRange is then called recursively on these two portions of the table.

    n ← j + 1 - i;
    IF n<=1 THEN RETURN;  --Nothing to sort.
        --Sort i, j.
    [] ← IfCompareSwap [table, i, j, greaterthan];
    IF n = 2 THEN RETURN;  --There are only two elements.

    ij ← (i + j)/2;  --ij is the midpoint of i and j.
    --Sort i, ij, j so they are in nondescending order.
        IF IfCompareSwap [table, i, ij, greaterthan] >= equal
            THEN NULL  --We now know that di<=dij<=dj, so they are sorted
                          -- do nothing.
            ELSE [] ← IfCompareSwap [table, j, ij, lessthan]; -- Now di<=dij<dj.
    IF n = 3 THEN RETURN;  --di, dij, and dj are the only three elements.
    k ← i;
    l ← j;
    --Find k>i and l<j such that dk>dij>dl (i.e. are in reverse order). Swap k and l.
    --Repeat until k and l pass each other, at which point k and l are approximately equal and
       specify the dividing line in the table between the elements that are less than or equal to
       dij and those that are greater than or equal to dij. SortRange is then called recursively on
       these two portions of the table.
    --Note that elements that are equal to dij are not moved.
       DO
       WHILE IfCompareSwap [ table, (l←l-1), ij, none] # less
                        --No swap was performed, only a comparison.
          DO ENDLOOP;  --Find l such that dl<dij.
       WHILE IfCompareSwap [table, (k ← k+1), ij, none] # greater
          DO ENDLOOP;  --Find k such that dij<dk.
       IF k > l THEN EXIT; --Do loop while k<=l.
       [] ← IfCompareSwap [table, k, l, all];
                   --Doesn't compare, just swaps.
                   --k will never equal l because dl<dij<dk.
       ENDLOOP;
    --Now l<k and di through dl are less than or equal to dij while dk through dj are all greater
    than or equal to dij. l+1 may be less than k, but only if the elements between dl and dk are
    all equal to dij. Fortunately, all elements equal to dij will cluster at exactly this point in the
    table anyway in one of three ways:
       --1)l+1<k and the elements between are equal to dij.
       --2) elements equal to dij within di through dl are larger than any other elements in that
             range and so will be placed at l.
       --3) elements equal to dij within dk through dj are less than any other elements in that
             range and so will be placed at k.
    --Thus it is alright for elements equal to dij to be sprinkled all over the table at this point.

    SortRange [table, i, l];
    SortRange [table, k, j];

    RETURN;
    END;

SubscriptOutOfBounds: PUBLIC ERROR = CODE;

END.
```

```
-- File: stringinttableprog6.mesa Last edited: November 14, 1977  2:17 PM  By: Peter Bishop

DIRECTORY
    SortDefs: FROM "sortdefs6",
    SystemDefs: FROM "systemdefs",
    InlineDefs: FROM "inlinedefs",
    DoubleDefs: FROM "doubledefs";

StringIntTableProg: PROGRAM IMPORTS SortDefs, SystemDefs
            EXPORTS SortDefs
            SHARES SortDefs =
BEGIN OPEN SortDefs, DoubleDefs;

--This file is similar to stringinttableprog.mesa in Appendix C except that use of MyTableHandle
    has been converted to use of TableHandle and there is no initialization code to notify generic
    procedures of the existence of this type of table since there are no generic procedures. The
    type specific procedures are being called directly now.

MakeMyTable: PUBLIC PROCEDURE [size: CARDINAL]
    RETURNS [val: TableHandle] =
    BEGIN
    val ← SystemDefs.AllocateHeapNode[
        size*SIZE[StringIntEntry] + SIZE[Table]];
    val↑ ← [size, ];
    END;

Range: PUBLIC PROCEDURE [table: TableHandle]
                RETURNS [low, high: CARDINAL] =
    BEGIN
    low ← 1;
    high ← table.size;
    END;

Shorten: PRIVATE PROCEDURE [size: CARDINAL, subscript: CARDINAL]
    RETURNS [i: CARDINAL] =
    BEGIN
    IF subscript > size OR subscript = 0 THEN
        ERROR SubscriptOutOfBounds [];
    i ← subscript;
    END;

comparray: ARRAY Comparison OF CompareOp =
    [lessthan, equalto, greaterthan];

IfCompareSwap: PUBLIC PROCEDURE [
    table: TableHandle, first, second: CARDINAL, comp: CompareOp]
    RETURNS [val: Comparison] =
    BEGIN
    t: TableHandle;
    l, i: CARDINAL;
    f, s: POINTER TO StringIntEntry;
    temp: StringIntEntry;
    t ← table;   -- this is the easiest way to change the general code.
    f ← @t.t[Shorten[t.size, first]];
    s ← @t.t[Shorten[t.size, second]];
    val ← equal;
    IF comp # all THEN
        BEGIN
        l ← MIN[f.name.length, s.name.length];
        FOR i IN [0..l) DO
            IF f.name[i] # s.name[i] THEN
                BEGIN
                val ← (IF f.name[i] > s.name[i] THEN greater ELSE less);
                EXIT;
                END;
            ENDLOOP;
        IF val = equal THEN
            IF f.name.length > l THEN val ← greater
            ELSE IF s.name.length > l THEN val ← less;
        IF InlineDefs.BITAND[comp, comparray[val]] = 0 THEN RETURN;
        END;
    temp ← f↑;
    f↑ ← s↑;
    s↑ ← temp;
    END;

END.
```