# The Elements of Mesa Style

by James H. Morris, Jr.

June 1, 1976

An essay and a few examples are presented to illustrate some of the novel features of Mesa. Specifically, the compile-time checking facilities that deal with types and the inter-module connections are brought to bear on some typical systems programming problems.

XEROX

PALO ALTO RESEARCH CENTER
3333 Coyote Hill Rd/Palo Alto/California 94304

This document is for internal Xerox use only.

## Foreword

The essay and examples that follow are the product of my effort to learn the Mesa programming language and relate it to oft-discussed but little understood ideas like modularity, reliability, and structure. There are some suggestions about how Mesa can be used to produce reliable software. This is far from the final word on Mesa programming; a few years hence we shall all know a good deal more about programming in Mesa.

I have concentrated on some of the more interesting new features of Mesa and tried to use them in solving some perennial systems programming problems. I have tended to emphasize less obvious features of Mesa that may otherwise escape your attention. Because of the emphasis this manual is neither a primer for Mesa nor an essay on general good style in programming. I recommend the Mesa manual for the former and a few books for the latter: **The Elements of Programming Style**, by Kernighan and Plauger (McGraw-Hill), **Structured Programming** by Dahl, Dijkstra and Hoare (Academic Press), or **Systematic Programming** by Wirth (Prentice-Hall).

The discussion and examples herein are based upon the system that runs today.

Several Mesa experts have given me a large amount of help in my efforts to learn Mesa and write this manual, most notably Chuck Geschke, Rich Johnsson, Butler Lampson, Jim Mitchell, Ed Satterthwaite, and John Wick.

## Static Checking as a Programming Tool

*There isn't any debugger in Peoria.* If you are writing a module of code which is going to go into a system which a Xerox customer in Peoria is going to use, you must face up to the fact that software, just like hardware, gets "shipped" and can't be fixed easily after that. The customer will not be impressed with an interactive debugger -- his way of fixing bugs is to replace the system with another manufacturer's.

Mesa differs from other languages commonly used for systems programming in that its compiler has a rather elaborate part called the *type checker.* This checker is a tool, like a debugger, which one uses to eliminate programming errors. It is unlike the debugger, however, in that it is applied to the static program, and is not used at run time. When the type checker catches an error of yours you should not grumble, because it means one less error to plague you during the testing phase. In fact, you should be ecstatic, because it may have caught an error which would have turned up in Peoria.

The type and range declaration facilities should be looked upon like the T-square, triangles, and compass a draftsman uses. Once we learn how to use them they can be used to get the details right, once and for all. It would be ridiculous for a draftsman to suggest that one of these tools was interferring with his work; being able to draw straight lines free-hand is not what he prides himself on. It would be equally ridiculous for the purveyor of a new drafting tool to suggest that it will obsolete all the draftsman's skills.

### *Articulating Data Types*

How do you use the Mesa type checker? What kinds of programming errors can it prevent? To answer these questions we must first understand what it checks. Basically, it checks that a certain partitioning of the value space into distinct types is respected. It performs the same kind of checks that different arrangements of prongs on electrical connectors do. Every time values are passed from one place to another by assignment, procedure call, etc. the checker insists that the sender and receiver of the value agree about its type.

The partitioning into types is initially set up by Mesa. As a minimum requirement any two values requiring different amounts of storage must have a different type, but there are further distinctions. For example, Mesa has decided that INTEGER and BOOLEAN values are different and that a POINTER TO INTEGER is different from an INTEGER. It takes a while to learn how to describe all one's favorite data structures using Mesa's type language, but his efforts are rewarded by the absence of any bit counting errors.

The real fun comes when the programmer adds further refinements to the partition by using

the type constructor RECORD. Every occurrence of a RECORD constructor generates a new type distinct from all others.

If you made the declarations

```
Alist : TYPE = POINTER TO RECORD [hd: INTEGER, tl: Alist];
Blist : TYPE = POINTER TO RECORD [hd: INTEGER, tl: Blist];
x: Alist;
y: Blist;
```

you would be prohibiting yourself from mixing up Alists and Blists; e.g. the assignment

```
x←y
```

would be illegal.

Why would a sensible programmer do such a thing when the only effect will be to cause the type checker to complain more often? He would do it if there is a real, intentional difference between Alists and Blists and he is worried enough about getting them mixed up that he wants the compiler to check it.

For example, suppose Alists are expected to contain only even numbers while Blists may contain any numbers at all. The type system is not versatile enough to express this difference, but it is still of use in expressing the fact that there are two kinds of lists. If we want to prove Alists always contain even numbers we can break the proof into two stages:

(1) Find all the assignments of the form x.hd ← e where x has type Alist.
(2) Prove that e is even.

The first part is greatly aided by the type checker. The second part must be done by hand, but we have more mental energy left for this possibly difficult proof.

The difference between Alists and Blists may be virtually non-existent from a mathematical point of view. For example, Alists might contain identification numbers of people with top-secret security clearances while Blists contain the identification numbers of known communists. Even though the difference between these lists is rather subtle for a computer, a programmer is well advised to keep them separate.

If we keep these two types articulated we will run into certain problems. We might like to write procedures for concatenating, comparing, and otherwise fiddling with lists in ways which neither depend upon nor effect their "Aness" or "Bness". In fact we may even want to change the type of a list occasionally. It is often better to use the type loopholes rather than to make the types identical. For example, suppose the quintessential property of Alists

is that they contain only even numbers. Then the following procedures might be declared.

```
SumBlist: PROCEDURE [x: Blist] RETURNS [sum: INTEGER] =
    BEGIN t: Blist;
    sum ← 0;
    FOR t ← x, t.tl UNTIL t=NIL DO sum←sum+t.hd ENDLOOP;
    RETURN;
    END;

SumList: PROCEDURE [UNSPECIFIED] RETURNS [INTEGER] = SumBlist;

ConcatBlists: PROCEDURE [x,y: Blist] =
    BEGIN t: Blist ← x;
    IF x=NIL THEN ERROR;
    UNTIL t.tl = NIL DO t←t.tl ENDLOOP;
    t.tl ← y
    END;

ConcatAlists: PROCEDURE [Alist, Alist] = COERCE[ConcatBlists];

ConvertList: PROCEDURE [x: Blist] RETURNS [Alist] =
    BEGIN t: Blist;
    FOR t ← x, t.tl UNTIL t=NIL DO IF t.hd MOD 2 = 1 RETURN [NIL] ENDLOOP;
    RETURN [COERCE[x]]
    END;
```

SumList can be applied to Alists, Blists, or any other one word type. I would like to prevent the third possibility but I can't see any way. The situation is safer for ConcatAlists. Even though it is defined using COERCE, we can see that all is well since it demands that each of its parameters be Alists, and the concatenation of two Alists is still an Alist. Similarly, ConvertList is benign, as long as having even elements is the only qualification demanded of Alists.

A record declaration can be used for the sole purpose of inventing a distinct type as in

```
Prime: TYPE = RECORD [INTEGER];
x: INTEGER; y, z: Prime;
```

No extra space is taken by these records, but the effect on the type checker is rather interesting. We can convert Primes to INTEGERS without saying anything (because single component records are automatically converted to their components if necessary), but we must say "Prime" when going in the other direction. For example,

```
z ← Prime[7]; y ← z; x ← y; y ← Prime[x];
```

It seems reasonable that we cannot omit the Prime from those two places; they signal the places where the programmer is vouching for the primeness of the number.

The question of articulation comes up for variant records. Consider the following declaration for the ever-popular set of arithmetic expressions:

```
exp: TYPE = POINTER TO ex;

ex: TYPE = RECORD[SELECT etag:* FROM
                constant => [val: INTEGER],
                identifier => [id: identifier],
                negation => [neg: exp],
                sum,difference,product,quotient
                                => [left,right: exp],
                ENDCASE
                ];
```

The following alternative definition for ex minimizes variants by merging all the binary operators into a single sub-type.

```
ex: TYPE = POINTER TO RECORD[SELECT etag:* FROM
                constant => [val: INTEGER],
                identifier => [id: identifier],
                negation => [neg: exp],
                binaryexp => [op: {plus,minus,times,divide},
                                left,right: exp],
                ENDCASE
                ];
```

The second definition is less articulated than the first. The advantage of the second is that we can create expressions whose operator is not manifest and even change the operator, as in the following.

```
x ,y : exp;
x ← Alloc[SIZE[ex]];
x↑ ← binaryexp[variableop, y, x];
x.op ← minus;
```

In all of these situations one must weigh flexibility against the likelihood and cost of a mix-up.

## Interfaces and modularity

I have occasionally heard that modularity is a concept like motherhood; but nothing could be further from the truth. There are fairly well-defined requirements for achieving motherhood, but there is no clear criteria for what it takes to be a module. The general idea seems to be that a module is something that performs some conceptually simple task in a way its clients needn't be concerned with. Carried to its extreme it means that one can replace a module with a functionally equivalent one and no one will know the difference. All this points to the fact that a large part of module design must be devoted to designing the interface between the module and its clients.

When designing an interface one should try to decide which of three increasingly difficult situations obtains.

1. One-to-One. Even though we have decided to make an interface there will only be one program on either side of it. These programs may change through time, of course, but their identity will not be in doubt. Example 1 illustrates this situation.

2. Many-to-one. We are implementing a module which will serve many clients. There is only going to be one, possibly evolving, implementation; but it will be used by code with many purposes. Examples abound, device drivers, directory systems, etc. Examples 2 and 3 fall in this category.

3. Many-to-many. In this case the interface is the only thing left to design because we contemplate many implementations serving many clients. The latter parts of Example 2 illustrate this case.

### *Programming defensively*

In the more difficult situations it is often instructive to play the following game: Sit down with a single module in front of you and try to say something about its behavior that does not depend upon any other code with which it communicates. In fact, it helps to assume that all the other code in the system was written by Murphy, the discoverer and foremost exemplar of Murphy's law ("If something can go wrong it will"). Naturally, you will not be able to say all the things that you hope are true because the over-all performance of the module will depend on other components of the system. However, there may be some things you can say, like "this table is always sorted" which you can prove without looking outside the module. The starting point for such reasoning is the fact that we can restrict access to the information that a module depends upon.

Mesa offers a simple tool for putting some teeth into the notion of modularity, namely the ability to restrict the scope of text over which a name can be used. Generally speaking, a name coined by a module cannot be used by other modules unless the coining module declares it PUBLIC. This control can be applied to all kinds of names, including type names, procedure names, and the names of fields in records.

Suppose I were really uptight about the integrity of Alists from the previous discussion. I wanted to be absolutely certain that they only contained even numbers. I can isolate all the code that can affect Alists by writing the following module

```
AlistModule: PROGRAM =
Alist: PUBLIC TYPE = POINTER TO RECORD [hd: PRIVATE INTEGER, tl: Alist];

Hd: PUBLIC PROCEDURE [a: Alist] RETURNS [INTEGER] =
      BEGIN RETURN[a.hd] END;

SetHd: PUBLIC PROCEDURE [x: INTEGER, a: Alist] =
      BEGIN
      IF x MOD 2 # 0 THEN ERROR ELSE a.hd ← x;
      RETURN;
      END;
END.
```

Now we can be certain no Alist will ever have an odd number in it (except for initialization problems), and we do not need to look at any other part of the program to be sure. This is because the other parts of the program, even though they can declare Alists, cannot access the hd component directly but must use the procedures. (If I could change the PRIVATE to READ-ONLY the procedure Hd could be dispensed with. I understand that READ-ONLY is being added to the language.)

A rather surprising, if not obviously useful, feature of private type names is that, by leaving a type name private one can prevent a client from storing a class of values even though he might handle them briefly. Consider the following module

```
Silly: PROGRAM =

BEGIN
Secret: TYPE = RECORD[ id: INTEGER, mess: STRING];
Receiver: PUBLIC PROCEDURE[m: Secret] =
      BEGIN
      WriteDecimal[id];
```

```
        WriteString[mess]
        RETURN;
        END;

Sender1: PUBLIC PROCEDURE RETURNS[Secret] =
        BEGIN RETURN[Secret[1,"Hello"]] END;

Sender2: PUBLIC PROCEDURE RETURNS[Secret] =
        BEGIN RETURN[Secret[2,"Good-bye"]] END;

END.
```

Now a client is limited to saying Receiver[Sender1[]] or Receiver[Sender2[]] but little else because he can never declare anything to be a Secret.

*Confusion at the interfaces*

Even if your module works, its clients may not know how to use it properly. Suppose I wish to write a square root routine. We can distinguish three levels of misunderstanding about its performance, based upon how quickly they can be cleared up.

A typo: Sqrt can be applied only to real parameters. It will not work on strings.

A bug: Sqrt cannot be applied to negative numbers.

An unpleasant discovery: Sqrt(x)*Sqrt(x) is not always x.

It should be our goal to push the various misunderstandings that can occur as far down in this hierarchy as possible, in the direction of earlier detection. Notice that only the third requires the implementor's presence to explain. Mimimizing the errors that occur at this level saves wear and tear on implementor/client relations.

*Signals considered harmful*

Like any new and powerful language feature Mesa's signal mechanism, especially the UNWIND option, should be approached with caution. Because it is in the language one cannot always be certain that a procedure call returns, even if he is not using signals himself. Every call on an extra-module procedure must be regarded as an exit from your module, and you must clean things up before calling the procedure or include a catch phrase to clean things up in the event that a signal occurs. It is hard to take this stricture seriously because it is really a hassle, especially considering the fact that the use of signals is fairly

rare and their actual exercise even rarer. Because signals are rare there is hardly any reinforcement for following the strict signal policy; i.e. you will hardly ever hear anyone say, "I'm really glad I put that catch phrase in there; otherwise my program would never work." The point is that the program *will* work quite well for a long time without these precautions. The bug will not be found until long after the system is running in Peoria.

Here is a programming error I made which recently came to light (a year after its commission!). The basic idea will be familiar to all: I made a modification to modules A and B so that B passed A a piece of free space which A returned after a call on module C, which can generate a signal. Since I failed to put a catch phrase on that call of C, the expected return to free space was often missed, causing a gradual loss of free storage.

The discussion of DictionaryO presents another example.

Ironically, discouraging the use of signals has the opposite of the desired effect. The rarer signals are, the less chance of catching signal-related bugs during testing. This line of reasoning suggests that every module should generate an unwind signal now and then just to keep everyone on their toes! Bah!!

Maybe someone will write a checker which runs over a system of modules and warns us of all the procedure calls which may not return because of signals and don't have associated catch phrases. It seems likely that this checker will cry wolf a lot of the time.

It should be noted that Mesa is far superior to most languages in this area. In principle, by using enough catch phrases, one can keep control from getting away. The non-local transfers allowed by most Algol languages preclude such control. It has been suggested that systems programming is like mountaineering: One should not always react to surprises by jumping, it could make things worse.

The problem of handling exceptional conditions is a thorny one and Mesa has provided one of the more reasonable tools. This has not made the problem disappear, however.

## How to breach the type system

Here is a summary of all the ways I know of to breach the Mesa type system. If a program uses none of them, there should never be any anomalous, implementation-dependent behavior. However, it is occasionally necessary to subvert the system. Here are some suggestions about the relative dangers of the various ways of doing it. In general, the more obvious and transient the breach the less dangerous it is.

1. UNSPECIFIED is a type that matches any other one-word data type. It is inherited from PL/1. I try to avoid its use in favor of other constructs because it turns off too much type checking. To store 0 into an arbitrary memory location, 421B say, one could write

```
pi: POINTER TO INTEGER;
u: UNSPECIFIED;
u←421B; pi←u; pi↑ ← 0
```

2. COERCE is a compile time function which allows one to convert any one-word type into any other. Thus one can shorten the above to

```
pi: POINTER TO INTEGER = COERCE[421B];
pi↑ ← 0
```

It is better to use COERCE than declaring a variable UNSPECIFIED because it turns off checking at just one place in the program rather than every place the variable appears. If you cannot think of any reasonable type to describe the variable, it is a strange variable indeed. The use of COERCE also conveys much more information to the reader. It says, "I am now going to start treating this integer as a pointer. OK?" The reader is then expected to decide from the context whether that is a reasonable thing to do.

3. MEMORY is an array of UNSPECIFIEDS which happens to be the entire main memory of the machine. When using it one is expected to perspire a little rather than laugh fiendishly. Just say

```
MEMORY[421B] ← 0
```

4. Arithmetic on pointers is allowed. If x is a POINTER TO Bletch, so is x+1. If y is also a POINTER TO Bletch, x-y is allowed and is an INTEGER.

```
p: POINTER TO INTEGER = NIL;   -- NIL = 177777B = -1
(p+422B)↑ ← 0
```

See ArrayStore2 for an example of where pointer arithmetic seems to be justified.

5. It is possible to change a variant record from one variant to another at a time when

someone is depending upon its not changing.

```
R: TYPE = RECORD[SELECT typetag:* FROM
                    int => [a: INTEGER],
                    pint => [b: POINTER TO INTEGER],
                    ENDCASE
                ];
i: INTEGER ← 5;
r: R ← R[pint[@i]];
Ambush: PROCEDURE = BEGIN r ← R[int[421B]] END;
WITH r SELECT FROM pint => BEGIN Ambush[]; b↑ ← 0; ENDCASE;
```

This problem does not occur in practice very often because people don't change the variants of records very much.

6. Variant records can be declared with the COMPUTED attribute, meaning that the variant tag is computed by a programmer-supplied function. This is a useful facility, and not considered too dangerous, especially if one makes the function computing the tag explicit.

```
TypeTag: TYPE = {int,pint};
R: TYPE = RECORD[SELECT COMPUTED TypeTag FROM
                    int => [a: INTEGER],
                    pint => [b: POINTER TO INTEGER],
                    ENDCASE
                ];
r: R = R[int[421b]];
WITH r SELECT pint FROM pint => b↑ ← 0; ENDCASE;
```

See ArrayStore4 for an example of this feature used properly.

7. A variable can be declared to be a specific variant of a record type. Then, one can manage to falsify that declaration by assigning through a pointer to that variable which does not insist on the right variant. Starting with the type declarations immediately above one can say

```
pi: pint R;
sneakpath: POINTER TO R = @pi;
sneakpath↑ ← R[int[421B]];
pi.pint↑ ← 0
```

As in 5. this problem only arises if you are in the habit of changing the variants of records after they have been initialized.

8. For completeness I mention that out-of-bounds array subscripts are not checked against. Thus one can say

```
A: ARRAY [0..10] OF INTEGER;
A[NIL-@A[0]+422B] ← 0
```

9. Again for completeness, note that variables are not initialized when they are declared. Thus the following segment *might* clear the display. (On the Alto, 421B is the address of a chain of control blocks for the display. A zero in 421B clears the display.)

```
p1: PROCEDURE = BEGIN x: INTEGER; x ← 421B; RETURN END;
p2: PROCEDURE = BEGIN y: POINTER TO INTEGER; y↑ ← 0; RETURN END;
p1[]; p2[]
```

This will only work if the Mesa run-time system uses the stack frame released by p1 for the invocation of p2. Who cares? No one is supposed to make a virtue of this vice.

10. The DESCRIPTOR construct allows the following fiddle.

```
A: DESCRIPTOR FOR ARRAY OF INTEGER;
B: DESCRIPTOR FOR ARRAY OF POINTER TO INTEGER;
A[0] ← 421;
B ← DESCRIPTOR[BASE[A],LENGTH[A]];
B[0]↑ ← 0;
```

Here are two equally silly views one can take about breaches of the type system:

**Super-hardnose:** One breach invalidates everything, since we cannot prove that code won't be overwritten, etc..

**Flower-child:** I and all my fellow programmers are reasonable people who will do whatever is right.

Here are some less silly suggestions.

Understand when you are committing a breach and make it clear to the reader. Specifically, watch out for breaches 5, 7, 8, and 9.

*Confine the effect of a type breach to one module.* Try to prove that, assuming all the other modules don't commit a breach, nothing untoward will happen because of yours. In

particular, using UNSPECIFIED to declare public procedure entries seems dangerous since the module using them may be entirely ignorant of the breach.

## A guide to the examples

This edition contains three long examples. I am fully aware that they are not real, "blood 'n guts" systems programs. They have been kept simple so as to illustrate various ideas more clearly.

The programs appear after each example in the order discussed.

*Program layout and Fontology*

In an effort to make programs easier to read I have chosen the following conventions:

(1) A clean font, Helvetica, is used as the basic font. I have tried to avoid using the identifier j which looks too much like i.

(2) A smaller font is used for keywords. Otherwise Mesa programs tend to look like a blizzard of BEGINS, ENDS, and PROCEDURES. Most of the keywords are punctuation and don't deserve to attract so much attention. The general rule is that any word that the Mesa compiler knows about is in a smaller Helvetica font.

(3) Identifiers defined outside this manual, like `WriteChar` which is part of the Mesa library, appear in Gacha. A general rule is to write both the name of an insert file in the DIRECTORY section and all the identifiers that come from it in the same font. It is probably not a good idea to have a different font for every insert file; the reader will contract a case of font-fatigue, observed in people who habitually read ransom notes.

(4) The DIRECTORY and DEFINITIONS section of each program is pushed over to the right. They are not usually the first thing one wants to read.

(5) Boldface is used for defining occurrences of procedure names and comments that delineate major sections..

(6) Italics are used for comments that are remarks.

(7) The indentation methods suggested in the Mesa manual are used.

(8) Declarations usually appear in alphabetical order.

## Example 1. KWIC

The problem is to write a program to produce Key Word In Context listings. This is the same problem discussed by Parnas in "On the criteria to be used in decomposing systems into modules," in the *Comm. ACM 15*,12 (Dec. 1972). The main point of his paper is vitally important: the task should be decomposed according to representation of objects rather than the sequence of events. In this specific example the trick is to construct a module LineStorage which appears to be storing many more lines of text than it actually is. This illusion is accomplished by requiring anyone outside the module to use procedure calls to get at the characters in each line. The program consists of three modules, KWIC the master controller, LineStorage, and Utilities which contains Sort, a general purpose sort routine.

The program doesn't produce the nicest possible index. Given the (randomly chosen) input lines:

```
types are not sets
protection in programming languages
```

the program is supposed to produce the output

```
are not sets types
in programming languages protection
languages protection in programming
not sets types are
programming languages protection in
protection in programming languages
sets types are not
types are not sets
```

Go read the program now, come back, and ponder the following profundities:

*Procedures as parameters are useful for making general interfaces.*

Consider Sort. Since we want to be able to use it to sort all kinds of things, with many kinds of ordering relations we want to be quite noncommittal about the things it is actually sorting. Since Sort's commerce with its subject array can be reduced to two operations -- comparing and swapping -- we can get by with supplying two procedures which perform on an array which Sort never sees! Surprisingly there is no need to breach the type system, since the actual values from the hidden array never even make an appearance inside Sort.

Consider LineStorage. We know that it's going to get its input from a file, but it's nice to relieve it from the responsibility for finding out which one. It's also nice to give it input from the keyboard during debugging. Once again, we pass it two procedures, getc and endofc, which are all that it needs to read in a stream of characters. Since this is a rather common event we might even want to institutionalize such a pair of procedures as a record type:

```
InputStream: TYPE = RECORD
[getc: PROCEDURE RETURNS [CHARACTER], endofc: PROCEDURE RETURNS [BOOLEAN] ];
```

This idea is carried to wretched excess in some systems.

The reason procedure parameters are so useful is that they allow one to switch the locus of control back and forth between two modules in a fairly arbitrary way. Thus one can divide the responsibilty between two modules without worrying about the actual time sequence of the activity.

*Make the client pay for the space*

I had problems deciding how to allocate space for LineStreams. I wanted KWIC to be ignorant of how LineStorage was representing the LineStreams, so I thought that LineStorage would have to allocate the space for them. On the other hand, I certainly didn't want to have a general-purpose LineStream allocator since I knew at most two streams would ever be open at one time. The rather clumsy solution I chose was to have two distinct pairs of OpenLine and GetLineC routines, each with their own storage in the module. The method used here was suggested by Butler Lampson and is much nicer: KWIC allocates the storage; but, because of the PRIVATE attribute in the declaration of LineStream, it still can't see the representation. Thus, for very little hassle, LineStorage is more general since it can process any number of streams.

*Support your local type-checker.*

The stickiest part of LineStorage is the fact that Text is not a homogeneous array of characters; each CR gets replaced by an index of the array pointing back to the beginning of the line. I suppose I could have re-designed the scheme for representing all the rotations of a line, but I am rather fond of this one since it is so space economical.

The declaration of union is a circumlocution forced upon us by Mesa's insistence that type variation can occur only within records. The actual code is not too ugly, however. We must say

```
Text[i] ← union[char[c]]
```

where ALGOL-68 would let us get away with

```
Text[i] ← c
```

but you can't have everything.

The worst hassle, represented by the procedure AssertChar, is unavoidable in any language. Throughout most of the initialization phase we know that all the elements in Text are characters. The type checker is not very clever, however, so we have to surround every use of a Text element with a call on AssertChar.

I was tempted to declare Text UNSPECIFIED, but decided to go along with the type checker. If I had not, the GetLineC procedures would have been pretty messy, involving some bit-extraction by hand or some other barbarism.

*The goto is alive and well in* Mesa.

It lives under a variety of assumed names, one of which is SELECT. Observe the UNTIL loop which reads in the Text array. The four identifiers, Initial, LineEnded, InWord, and WordEnded are thinly disguised labels and the assignments like state ← InWord are delayed-action goto's. I arrived at this method of doing things after getting entirely confused trying to get a loop-with-loop arrangement. The difficulty is that one ends up testing for the end of the stream all over the place. This code was improved by B. Lampson. I originally SELECTed on the state first and then the character, a clumsy arrangement.

*Three loops are better than one.*

A casual inspection will reveal that the three loops in the initialization phase can be merged into one. Unless one is really pressed for computing time or code space it seems better to leave them separate since it makes the program easier to read. It is easier to read because it is broken into three simply described activities. The sentence "Mary cried after the ball that John threw hit her," is obviously more sophisticated than "John threw the ball. The ball hit Mary. Mary cried." It is harder to read, however.

LineStorageA shows the initialization code for LineStorage with its three loops merged. It took me about 15 minutes to derive from the original version and it would probably take you at least 15 minutes longer to read it to find a bug, especially if you hadn't seen the

original. It may run a little faster, but there are a lot of microseconds in 15 minutes. The program is also shorter, and we were able to eliminate all the AssertChar nonsense.

This idea is not very popular. Everyone seems to think that one should strive for short, elegant programs. It is one's natural inclination to merge the loops, even while writing them for the first time. One wishes to avoid the work and overhead associated with setting up a loop, it seems.

*Declarations can be decorative.*

The use of constant and interval declarations is purely for the benefit of the reader. As far as the type checker is concerned all intervals are integers. For example, the types TextIndex and LineIndex in LineStorage could be replaced by INTEGER. Doing so would be in bad taste, however. First of all, there may come a day when the compiler will check ranges. Second, the allocation of space in records takes advantage of the smaller ranges. In any case, the additional types are quite helpful to the reader. For example, declaring the parameter, i, of OpenLine to be a LineIndex and firstC to be a TextIndex would make it very easy for a reader to spot the error of saying "firstC ← i", even though the type checker couldn't.

The use of decorative declarations can be overdone, however. For example, declaring a constant WordSize = 16 when you know it is never going to change has always struck me as cruelly misleading. It seems better to write things like

```
i ← 16; -- word size
```

to convey the message. The following scenario should be familiar: The reader encounters WordSize somewhere and doesn't even know it's a constant so he has to go looking all over the place to see who is setting it. Failing to find any assignments to it he finally locates its constant definition on the third page of a definitions file. Not only has he wasted a lot of time, he may also get the impression that all he needs to do is change the definition of WordSize to make the code run on another machine! Hah!

*Arrays always start at 0.*

Mesa has chosen that convention for array descriptors and strings; so we had best stick to it whether we like it or not. In the privacy of your own module, you can start arrays at 1, -6, or anything else; but if an interface conflict ever occurs the person who assumed 0-origin is, by definition, right.

You might think that the compiler would settle such conflicts, but it doesn't. Even if range checking were added to the compiler there would be cases it wouldn't handle. Consider the

Sort procedure. Here the array in question is virtual since it is accessed through the procedures LessThan and Swap. No type checker would notice if the Sort module assumed that this virtual array started at 1. There might be an out of bounds array reference sometime, but who knows whether it would be caught? Even though it would be very nice to assume 1-origin indexing (because the tree-encoding hack requires it) I made the Sort module bend to the 0-origin convention.

Certain other conventions are suggested from the 0-origin one, and I have attempted to illustrate them from the rather stylized declarations of the arrays in LineStorage. Specifically,

a) Communicate the size of an array by giving the number of elements, N, rather than the highest possible subscript, N-1. Thus 0, not -1, means the array is empty. The use of the interval notation in FOR loops helps one avoid writing lots of -1's.

b) Declare a new array, A, with the following packet of declarations

```
A: ARRAY Aindex OF Foo;
Aindex: TYPE = [0..mxA);
mxA: INTEGER = 1000;
```

It has been suggested that one waste a word at the end of arrays when he can afford it, by changing the ")" to a "]". The idea is to make things look safer to a putative bounds checker when it encounters

```
i: Aindex;
UNTIL i=mxA DO Process[A[i]]; i←i+1 ENDLOOP;
```

This idea seems unappealing. During testing one would like an index overflow to clobbler someone else so it was brought to the tester's attention.

c) When the array is being filled up the running index should point at the next cell to receive a value. Thus it is initialized to 0 and denotes the number of elements present. The general idea is captured by the following

```
nA: Aindex ← 0;
PutA: PROCEDURE [x:Foo]=
      BEGIN IF nA>=mxA THEN ERROR;
      A[nA] ← x;
      nA ← nA + 1;
      END;
```

d) The null index should be -1. When you want to return an index which says "I

couldn't locate the item," return -1, since 0 would mean "I found the item in A[0]."

All of this can be summarized by saying that intervals should normally be described by the position of their first element and the position one after their last element.

*Anything goes between two consenting modules*

The interface between KWIC and LineStorage is not particularly general because KWIC accesses the array Line directly, rather than using a procedure call. I think this is alright since it seems clear that LineStorage is never going to be used for any other purpose than servicing KWIC or some revision of it. Contrast this with the extreme generality of the interface to Sort. We know that the Sort module is going to be used by many other modules, so we took some pains to generalize.

It takes a lot of thought to make a general, easy to use interface. It also takes the user of such an interface some work to specialize it to his needs. Don't waste your energy when you know there is only going to be one client for the module.

```
                                                DIRECTORY
                                                   IoDefs: FROM "IoDefs",
                                                   UtilitiesDefs: FROM "UtilitiesDefs",
                                                   LineStorage: FROM "LineStorage";

                                                DEFINITIONS FROM IoDefs, UtilitiesDefs, LineStorage;
KWIC: PROGRAM =
BEGIN

-- Storage
c: CHARACTER;
i: INTEGER;
LSM: POINTER TO FRAME[LineStorage];
N: INTEGER;
nxtC: CHARACTER ← SP;
st: LineStream;

-- Procedures
getc: PROCEDURE RETURNS [c: CHARACTER] =
        BEGIN c←nxtC; nxtC←ReadChar[]; WriteChar[nxtC]; RETURN END;

endofc: PROCEDURE RETURNS [BOOLEAN] =
        BEGIN RETURN [nxtC = 33B ] END; -- 33B=ESC

Swap : PROCEDURE [i,k: INTEGER ] =
        BEGIN OPEN LSM; t:INTEGER;
        t←Line[i]; Line[i]←Line[k]; Line[k]←t;
        END;

LessThan: PROCEDURE [i,k: INTEGER ] RETURNS [BOOLEAN] =
        BEGIN OPEN LSM; t1,t2:CHARACTER;
        s1, s2: LineStream;
        OpenLine[@s1, i]; OpenLine[@s2, k];
        DO
                t1 ← GetLineC[@s1]; t2 ← GetLineC[@s2];
                IF t1=t2  THEN BEGIN IF t1=CR THEN RETURN [FALSE] END
                ELSE RETURN [t1<t2]
        ENDLOOP;
        END;

-- Code

c←getc[]; -- just to get started
LSM ← NEW LineStorage [getc,endofc];  --  create space
BIND LSM; -- link externals
START LSM; -- execute initialization

Sort[LSM.nLine,LessThan,Swap];

FOR i IN [0..LSM.nLine) DO
        LSM.OpenLine[@st, i];
        c ← SP;
        UNTIL c=CR DO c←LSM.GetLineC[@st]; WriteChar[c]; ENDLOOP;
        ENDLOOP;
END.
```

```
                                DIRECTORY  IoDefs: FROM "IoDefs";
                                DEFINITIONS FROM  IoDefs;

LineStorage: PROGRAM [getc: PROCEDURE RETURNS [CHARACTER], endofc: PROCEDURE RETURNS [BOOLEAN] ] =
      -- stores lines for KWIC.  parameterized on input stream
BEGIN
-- Storage, Constants & Types
c: CHARACTER ;
fcl TextIndex;
i: TextIndex;
Line: PUBLIC ARRAY LineIndex OF TextIndex; -- holds line pointers
LineIndex: TYPE = [0..mxLine);
LineStream: PUBLIC TYPE = -- used by OpenLine, GetLineC
   PRIVATE RECORD [firstC, nxtC: TextIndex, end: BOOLEAN ];
mxLine: INTEGER = 500;
mxText: INTEGER = 1000;
nLine: PUBLIC LineIndex ← 0;
nText: TextIndex ← 0;
Text : ARRAY TextIndex OF union; -- holds input text
TextIndex: TYPE =[0..mxText);
union : TYPE = RECORD [SELECT tag: * FROM
                        ptr => [ ptr: TextIndex],
                        char => [ char: CHARACTER],
                        ENDCASE
                        ];

-- We contrive to represent all possible rotations of a title by storing the text just once by the following
method:
-- Text will consist of the input lines each terminated by a pointer back to its first character. The elements of
the array Line point to the first characters of the words in each line. Thus, a particular pseudo line is gotten
by starting at Line[i] and reading characters until the back pointer is reached, following the back pointer and
continuing until Line[i] is reached again.

-- Procedures
AssertChar : PROCEDURE [ u:union] RETURNS [CHARACTER] =
      -- This procedure is used solely to keep the type checker happy.  We know that the union item
      must be a char, so the ERROR will never happen.
      BEGIN
      WITH u SELECT FROM
            char => RETURN [char];
            ptr => ERROR;
            ENDCASE
      END;

GetLineC: PUBLIC PROCEDURE [s: POINTER TO LineStream ] RETURNS [c:CHARACTER] =
      -- gets character from pseudo line
      BEGIN OPEN s;
      IF end THEN RETURN [CR];
      WITH Text[nxtC] SELECT FROM
            ptr => BEGIN c ← SP; nxtC ← ptr END;
            char => BEGIN c ← char; nxtC ← nxtC +1 END;
            ENDCASE;
      end ← nxtC = firstC;
      RETURN;
      END;

OpenLine: PUBLIC PROCEDURE [s: POINTER TO LineStream, i: LineIndex] =
      -- opens pseudo line
      BEGIN OPEN s:
      firstC←Line[i]; end ← FALSE; nxtC ← Line[i] ; RETURN
      END;

PutLine:  PROCEDURE [i:TextIndex] =
      BEGIN IF nLine=mxLine THEN ERROR;
      Line[nLine] ← i;
      nLine ← nLine+1;
```

```
        END;

PutText: PROCEDURE [x:union] =
        BEGIN IF nText=mxText THEN ERROR;
        Text[nText] ← x;
        nText ← nText+1;
        END;

-- Initialization code

-- Read in Text, discarding extra lines and blanks
        state : {Initial,LineEnded,InWord,WordEnded} ← Initial;

        -- The subarray Text[0..nText) consists of the characters read so far, except each sequence of SP's
        is replaced by a single SP and any sequence of SP's and CR's is replaced by a CR. The variable
        state tells what kind of input sequence we're in. Only when a character other than SP or CR is read
        do we store an SP or CR in Text.

        UNTIL endofc[] DO
                c ← getc[] ;
                SELECT c FROM
                        CR => IF state#Initial THEN state ← LineEnded;
                        SP => IF state=InWord THEN state ← WordEnded;
                        ENDCASE =>
                                BEGIN
                                SELECT state FROM
                                        LineEnded => PutText[union[char[CR]]];
                                        WordEnded => PutText[union[char[SP]]];
                                        ENDCASE;
                                PutText[union[char[c]]];
                                state ← InWord;
                                END;
                ENDLOOP;

        PutText[union[char[CR]]];

-- Text[nText-1], is the CR of the last line read.
-- Every item in Text is a char, not a ptr.

-- Fill in the line table with pointers to word beginnings
        Line[0] ← 0;
        nLine←1; --  next word
        FOR i IN [0..nText-2] DO
                IF AssertChar[Text[i]] = SP OR AssertChar[Text[i]]=CR
                  THEN PutLine[i+1] ;
                ENDLOOP;

-- Replace all the CR's with back pointers
        fcl ← 0; -- First character of current line
        FOR i IN [0..nText) DO
                IF AssertChar[Text[i]] = CR THEN
                        BEGIN
                        Text[i] ← union[ptr[fcl]];
                        fcl ← i+1;
                        END;
                ENDLOOP;

END.
```

```
DIRECTORY SystemDefs: FROM "SystemDefs",
InLineDefs: FROM "InLineDefs",
UtilitiesDefs: FROM "UtilitiesDefs";
DEFINITIONS FROM SystemDefs, InLineDefs, UtilitiesDefs;

Utilities: PROGRAM  IMPLEMENTING UtilitiesDefs =

PUBLIC BEGIN

CompareString: PROCEDURE [x, y: STRING] RETURNS [CompareAnswer] =
        BEGIN
        lowerCase: PROCEDURE [c:CHARACTER] RETURNS [CHARACTER] =
                BEGIN RETURN
                [ IF c IN ['A..'Z] THEN c-'A+'a ELSE c ]
                END;
        i:INTEGER ← 0; -- current character
        DO
                IF i=x.length AND i=y.length THEN RETURN [equal];
                IF i=x.length THEN RETURN [prefix];
                IF i=y.length THEN RETURN [extension];
                IF lowerCase[x[i]] < lowerCase[y[i]] THEN RETURN [less];
                IF lowerCase[x[i]] > lowerCase[y[i]] THEN RETURN [greater];
                i ← i +1
        ENDLOOP;
        END;

CopyString: PROCEDURE [s:STRING] RETURNS [new: STRING] =
        BEGIN i:WORD;
        new ← AllocateHeapString[s.length];
        new.length ← s.length;
        FOR i IN [0..s.length) DO new[i] ← s[i] ENDLOOP;
        RETURN;
        END;


Sort: PROCEDURE
                [N:INTEGER,
                 LessThan : PROCEDURE [INTEGER,INTEGER] RETURNS [BOOLEAN],
                 Swap:  PROCEDURE [INTEGER,INTEGER]
                ] =
                -- This is just your basic TreeSort, except that it never actually touches the array in
                question but uses the procedures LessThan and Swap.  Furthermore, by Mesa's convention,
                Sort is obliged to work on 0-origin arrays. This is coped with by subtracting (an italic) 1
                from each actual parameter of LessThan and Swap.
        BEGIN
        siftUp: PROCEDURE [low,high: INTEGER] =
                BEGIN k,son: INTEGER;
                k←low;
                DO
                        IF 2*k>high THEN EXIT;
                        IF 2*k+1>high OR LessThan[2*k+1-1,2*k-1] THEN son←2*k ELSE son←2*k+1;
                        IF LessThan[son-1,k-1] THEN EXIT;
                        Swap[son-1,k-1];
                        k←son;
                ENDLOOP;
                RETURN
                END;

        i :INTEGER;
        FOR i DECREASING IN [1..N/2] DO siftUp[i,N] ENDLOOP;
        FOR i DECREASING IN [1..N) DO
                Swap[1-1,i+1-1];
                siftUp[1,i];
        ENDLOOP;
        RETURN
        END;
```

```
gt: PROCEDURE [x,y: UNSPECIFIED ] RETURNS [BOOLEAN] =
        BEGIN RETURN[USC[x,y] > 0] END;  -- USC is an UnSigned Compare primitive

ge: PROCEDURE [x,y: UNSPECIFIED ] RETURNS [BOOLEAN] =
        BEGIN RETURN[USC[x,y] >= 0 ] END;

lt: PROCEDURE [x,y: UNSPECIFIED ] RETURNS [BOOLEAN] =
        BEGIN RETURN[USC[x,y] < 0] END;

le: PROCEDURE [x,y: UNSPECIFIED ] RETURNS [BOOLEAN] =
        BEGIN RETURN[USC[x,y] <= 0] END;


END.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

UtilitiesDefs: DEFINITIONS =

BEGIN
CompareAnswer: TYPE = {less, prefix, equal, extension, greater};
CompareString: PROCEDURE [STRING, STRING] RETURNS [CompareAnswer];
CopyString: PROCEDURE [STRING] RETURNS [STRING];
LowerCase: PROCEDURE [CHARACTER] RETURNS [CHARACTER];
Sort : PUBLIC PROCEDURE
                [N: INTEGER,
                 LessThan : PROCEDURE [INTEGER,INTEGER] RETURNS [BOOLEAN],
                 Swap:  PROCEDURE [INTEGER,INTEGER]
                ];
gt: PROCEDURE [UNSPECIFIED, UNSPECIFIED] RETURNS [BOOLEAN];
ge: PROCEDURE [UNSPECIFIED, UNSPECIFIED] RETURNS [BOOLEAN];
lt: PROCEDURE [UNSPECIFIED, UNSPECIFIED] RETURNS [BOOLEAN];
le: PROCEDURE [UNSPECIFIED, UNSPECIFIED] RETURNS [BOOLEAN];
END.
```

```
                                DIRECTORY  IoDefs: FROM "IoDefs";
                                DEFINITIONS FROM  IoDefs;

LineStorageA: PROGRAM [getc: PROCEDURE RETURNS [CHARACTER], endofc: PROCEDURE RETURNS [BOOLEAN] ] =
      -- simplied intitialization

BEGIN
-- Storage, Constants & Types, same as LineStorage

-- Procedures, same as LineStorage, omitting AssertChar


-- Initialization code

-- Read in Text, discarding extra lines and blanks

      state : {Initial,LineEnded,InWord,WordEnded} ← Initial;
      fcl ← 0;
      -- The subarray Text[0..nText) consists of the characters read so far, execpt each sequence of SP's
      is replaced by a single SP and any sequence of SP's and CR's is replaced by a CR.  The variable
      state tells what kind of input sequence we're in.  Only when a character other than SP or CR is read
      do we store an SP or CR in Text.  The first character of the current line is Text[fcl].

      UNTIL endofc[] DO
            c ← getc[] ;
            SELECT C FROM
                  CR => IF state#Initial THEN state ← LineEnded;
                  SP => IF state=InWord THEN state ← WordEnded;
                  ENDCASE =>
                        BEGIN
                        SELECT state FROM
                              Initial => PutLine[nText];
                              LineEnded => BEGIN PutText[union[ptr[fcl]]];
                                          fcl ← nText;
                                          PutLine[nText];
                                          END;
                              WordEnded => BEGIN PutText[union[char[SP]]];
                                          PutLine[nText];
                                          END;
                              ENDCASE;
                        PutText[union[char[c]]];
                        state ← InWord;
                        END;
            ENDLOOP;

      PutText[union[ptr[fcl]]];

END.
```

## Example 2. Dictionaries

Let us design a module to associate integer values with names -- a dictionary or symbol table. In doing so let us try to make it as general and as impervious to its environment as possible. In other words, let us assume that it will be used for many years in many different contexts and that we won't be around to answer questions, fix bugs, or add enhancements. The idea is that, even though we are designing a piece of software, we want it to have the social characteristics of a hardware device: it performs reliably in any reasonable environment and 99% of its users never look inside the cabinet. Thus we want to protect the module from the programming mistakes of its clients and provide a simple, yet complete interface. I shall also illustrate how to provide mechanisms for alternating between different implementations and adjusting the type of things a dictionary stores.

*Choosing a primitive set*

The basic abstraction we have in mind is a memory that has strings for addresses. Thus we have the primitives

```
Fetch: PROCEDURE [STRING] RETURNS [INTEGER]
Store: PROCEDURE [STRING,INTEGER]
```

Fetch returns the last value stored for a string or -1 if there is none. I considered having Fetch generate a signal when there was no value stored, but decided that signals were a little drastic and that many applications can get by without using -1 as a legitimate value.

Are these primitives enough? The answer to this question depends in complicated ways upon how we are going to use the module and who is asking for the enhancement, but here is a completely general argument that says these are not enough: Suppose one wants to write a program to save a dictionary on the disk or send it over some telephone lines, and later restore it. He cannot save it because there is no way for him to tell when he has fetched all the previously stored values. If dictionaries could be arbitrary partial functions, including ones with infinite domains, there might be intrinsic problems about saving them, but we know that a dictionary can contain only a finite amount of information. This problem could, in principle, be solved if a primitive to count the number of non -1 values was provided: one could then write a program which enumerated and tested all possible strings and stopped after the right number had been found. That is not very pleasant; let us have another primitive

```
Generate: PROCEDURE[proc: PROCEDURE[STRING, INTEGER] RETURNS [INTEGER]]
```

which applies proc to each of the dictionary's non -1 entries (in alphabetical order), and resets the value to whatever proc returns.

I spent an embarrassingly long time deciding what Generate should do. At first the plan was to have Generate just apply proc to the strings, letting proc use Fetch and Store to fiddle with the dictionary. This raised nasty conceptual problems: If proc deletes an entry (by storing -1) should that entry be generated? If the entry has already been generated, there is no question, but what if it is alphabetically later than the entry with which proc is being called? If proc adds an item, should the item be generated? There seem to be two extreme answers, with variations in between: (1) Let the bits fall where they may; i.e. just implement Generate, and see what happens. I probably would have done this if I were in a hurry. (2) Make a copy to generate from so that changes to the dictionary do not effect the set of strings generated. This seemed rather expensive considering how often it will matter. It seems that whatever one chooses there is going to be confusion. Finally, I decided to prohibit all changes to the dictionary during a generation except to the entry being generated, because in all the examples I could think of that was the only thing I ever wanted to change anyway.

Incidentally, were it not for Mesa's compile time checking of parameter types I would never choose this kind of interface between Generate and proc. It would be too dangerous, because the writer of proc might forget to return the same value he receives in those cases he didn't care about. Then random values would get stored into the dictionary. Fortunately, Mesa checks that proc has the right type and that every return from it gives some integer value. This will serve to remind the forgetful. (A really nice designer would provide an alternate version of Generate for the common case in which no alteration occurs.)

These primitives are complete in the weak sense that we can read and write the abstract state of a dictionary with them. This ability is shown by the following code to copy the contents of D1 into D2.

```
Forget2: PROCEDURE [s:STRING, x: INTEGER] RETURNS [INTEGER] =
    BEGIN RETURN [-1] END;

StoreIn2: PROCEDURE [s:STRING, x: INTEGER] RETURNS [INTEGER] =
    BEGIN D2.Store[s,x]; RETURN [x] END;

D2.Generate[Forget2];
D1.Generate[StoreIn2];
```

(Notice that Fetch is not needed anymore, except for efficiency.)

Now we know that any *reasonable* operation on the state of a dictionary can be

programmed using these primitives by the following brute force method:

(1) Read out the abstract state.
(2) Fiddle with it any way you like.
(3) Write the abstract state back.

Example of reasonable operations are: Count the entries, merge two dictionaries, and reset a dictionary to empty. These are easily programmed in terms of the primitives.

Unreasonable operations are: tell which entry was stored last, create a duplicate entry, and count available space for new entries. These are unreasonable because they have no meaning in terms of the abstract state which, in this case, is a function from strings to integers. Adding any of these operations would change the nature of the abstract state space.

The abstract state that a module implements is a rather slippery thing. In this case it is easy to see, but in more realistic examples it is not. Here is a general rule: The concrete state is the state of the storage inside the module when it is not running, i.e. the storage in its global frame. Abstract states are represented by concrete states. Two concrete states represent the same abstract state if there is no way to distinguish them from outside the module.

Here is a typical discussion about abstract dictionary states:

> *Client:* I want a primitive to tell me the last thing I stored.
> *Programmer:* That's not part of the abstract state.
> *C:* Yes it is.
> *P:* O.K., wise guy, write some code which behaves differently depending upon which of "Store["a",1]; Store["b",2]" or "Store["b",2]; Store["a",1]" it follows.
> *C:* That's exactly my point: I can't.
> *P:* That's exactly *my* point: The abstract state that exists after those two sequences is the same, so the information you want is not part of the abstract state. Providing the primitive you want involves redesigning the abstraction, and that requires a meeting of the board.

*The basic design* -- Dictionary0

Our strategy is to keep all the entries in an array Pair, ordered alphabetically. We use the system's free space package to get space for both the arrays and the strings that go into them.

The representation mapping describes how the abstract state, a function, is represented by the concrete state which is comprised of Pair, nPair, Psize, and indisposed. The second two variables are irrelevant to the representation mapping.

The module invariant describes things about the concrete state that we hope are always true when the module is entered. As we change things it may become false, but we will make it true before we exit the module again. It is a good idea to make the invariant more precise by writing it as a procedure, Check, which causes an error if the invariant is not true. This is valuable, not only for debugging but also because it forces one to be more precise in formulating the invariant. Unfortunately, part (c) of the invariant is hard to capture in the check procedure because it asserts something about other modules and other instances of this module. The check that the keys are in strictly increasing order guarantees that no sharing occurs in this instance, but there seems to be no way to check elsewhere.

The heart of this module is the procedure Lookup which performs a binary search to find an entry with a given string. If it fails to find it, it returns the index of where it should go if we want to insert it.

Fetch and Forget are straight-forward. Notice that we can return the string to free space because the module invariant assures us that no one else points to it.

Store is complicated because we must occasionally expand the array Pair. Notice that we copy the string to assure that the module invariant (c) remains true.

Generate sets indisposed to TRUE upon entry, FALSE upon exit. This prevents re-entry to the module via Store or Generate. This policy eliminates many problems, some of which were discussed above.

What should we do if an entry is invoked when the module is indisposed? It is clearly a bug, but the answer depends upon whether the programmer who commited the bug is sitting in front of the screen. We shall temporize by signalling DictionaryIndisposed and continuing as best we can. During testing this signal can invoke a debugger. When the system containing this module is in service all one can do is log some information and try to keep going. For example, Store just returns if it is running in Peoria.

Speaking of signals, what if proc causes a premature termination of Generate by an UNWIND signal? If the UNWIND causes the end of the world we don't care, but if we ever want to use this module again we had better reset indisposed, thus the ENABLE clause.

Notice that Generate does not let proc see the real string, on the off chance a perverse proc might alter it.

Given all this hassle why not make Pair and nPair public and forget about the Generate procedure? My main reason for not doing this is my fear that the client will someday alter the array and cause the module to fail in some mysterious way. For example, if the array becomes unordered the binary search in Lookup can fail. This worry could be overcome if

Mesa allowed us to grant read-only access to a module's storage. There are still reasons for hiding the storage from even readers, however. Suppose after many months of the module's use we change the strategy of keeping the array ordered, and decide to use linear searches. How do we know that there are not programs out there which depend upon the ordering? We don't, and we had better not change anything. By keeping the storage entirely private we might be able to repeal, in a small way, a cardinal rule of programming: If it works don't change it!

The program DictionaryClient0 exercises this module in various ways. The procedure Murphy reflects some of the problems alluded to above.

My original hope was to conclude the discussion of this module with an informal proof that the module maintains its invariant; i.e. that the procedure Check can never cause an error. Unfortunately, there are so many ways that the invariant can fail to be true, that I gave up. Here is how I started:

The general idea is to assume that the invariant is true upon every entry (except the first) and prove that it is true upon every exit. Assuming that all the storage which the invariant depends upon is private to the module, this proves that the invariant is, indeed, true upon every entrance to the module. Thus the first step is to enumerate all the entries and exits; there are more than you think.

Entries:

1. The beginning of the module, where indisposed is initialized, when DictionaryClient0 said NEW[Dictionary0].
2. At the Initialization comment, when DictionaryClient0 said START Dict.
3. The beginnings of Fetch, Generate, and Store.
4. The return from proc, in Generate.
5. The catch phrase in Generate.
6. The returns from FreeHeapString, AppendString, CompareString, AllocateHeapNode, FreeHeapNode, and CopyString.
7. The returns from signals of DictionaryIndisposed.

Exits

1. At the initialization comment, after entry 1.
2. At the very end, after entry 2.
3. The returns from Fetch, Generate, and Store.
4. The call of proc in Generate.
5. Leaving the catch phrase in Generate.
6. The calls on FreeHeapString, AppendString, CompareString,

AllocateHeapNode, FreeHeapNode, and CopyString.
7. Signals of DictionaryIndisposed.

Right from the beginning there is trouble: What happens if someone calls Store before performing the START operation (not to mention Bind!)? It might be in bad taste, but there is nothing to prevent it. In Dictionary1 I shall expand the role of indisposed to detect this problem.

I would like to leave out categories 6 and 7 by arguing that those things can be regarded as atomic actions, and that every call is followed by a return at the same spot. However, because of a possible UNWIND there is no guarantee of this. For example, suppose the call of CopyString in Store results in an UNWIND. Then the assignments to Pair[place] and nPair will not happen and the effect will be to duplicate one entry while deleting another. Thus there will be two pointers to one string, and another will be lost. I don't know whether CopyString can cause an UNWIND or not. If it or any of the procedures it calls performs any signal whatsoever, the catcher of that signal, who could be a caller of Store, can cause an UNWIND to happen. Even if I inspect the code for CopyString and all its subordinates and discover none of them perform signals today there is no guarantee that it will be true tomorrow. If I were being really careful I would put a catch phrase in Store.

It should be emphasized that these difficulties are minor compared to those one finds in most programming languages. Most languages offer no satisfactory way to keep a module's storage private so that even contemplating such a proof is impossible.

*Creating multiple dictionaries* -- Dictionary1

Every time we say NEW[Dictionary0] we get a new dictionary, but if we are really going to exploit this feature we should change the module a little.

First, we may want to be able to get rid of a dictionary. We need an entry Finalize which returns the storage we allocated from the heap. While we are at it we might as well provide an initialization procedure so that a module can be reset easily. The role of indisposed is expanded to cover the case when an instance cannot be entered because it has not been initialized.

As a further enhancement we add a procedure, Extend, which adds one dictionary to another. This operation could be accomplished from outside the module by

```
StoreSink: PROCEDURE[s:STRING,x:INTEGER] RETURNS[INTEGER]
      BEGIN sink.Store[s,x]; RETURN[x] END;
source.Generate[StoreSink]
```

For reasons of efficiency, however, we shall include a primitive in the module. Specifically

```
sink.Extend[source]
```

will accomplish the same thing, but faster.

Extend is programmed to grab the entries directly out of source's array, which it refers to as source.Pair, and perform a merge with its own array. I was pleasantly surprised to discover that I could do this. I thought, because the storage of the module was private, that one instance of a module could not access the storage of another, but that is not the case. Apparently the protection of storage is based upon the entirely static question of where the code lives. The code in Dictionary1 can access the storage of any instance of the Dictionary1 module if it has its frame pointer.

Notice that I was careful to copy the strings coming out of source so as to preserve the non-sharing property.

DictionaryClient1 shows how this module can be used.

*Multiple dictionaries via private records* -- Dictionary2

There is another method to provide multiple dictionaries that does not involve multiple instances of modules: write a module that works on records which contain just the information needed to specify an instance, namely Pair, nPair, Psize, and indisposed. The various procedures will have to be changed to take pointers to the records as parameters. By using OPEN we avoid mentioning the dictionary parameter repeatedly, except in Extend where I chose to use the variable sink. Is this version of Extend easier to read?

DictionaryClient2 shows how this module is used.

Even though the storage resides in the client's domain he cannot access its contents because of the PRIVATE attribute in Dictionary's declaration. Thus the security of the module is not lessened very much by using records rather than instances.

This method will use less space per dictionary. Every time we create an instance of Dictionary1 a certain amount of invisible space is used: about ten words of frame overhead plus one word for each external procedure referenced (6, by my count). Since there is only one instance of Dictionary2 space is saved. In this application, the amount of space taken by a dictionary's data is large enough that one will not notice the extra 16 words, but it is probably better to use this method for smaller objects.

*Multiple implementations* -- Distributor

Since we have designed such a simple interface for dictionaries it is easy to imagine someone else writing a replacement. (Is that why we hardly ever make simple interfaces in real life?) Suppose it is called Dictionary1a. It is simple to switch over: just change the "Dictionary1" to "Dictionary1a" in the DIRECTORY section of a client who wants to use the new version.

Suppose we want to be able to use both versions simultaneously, and in most contexts be ignorant as well as indifferent about which one we have. This may sound rather strange, but there are real life examples of this sort of thing, namely streams. Whether a particular output stream is implemented by storing characters on a file, displaying them, or simply discarding them is usually of no concern to the programmer. To achieve this variability of implementation I used a record of procedure values, Dictionary, to play the role of a frame, as shown in Distributor. The client, DistributorClient, calls InitializeDictionary with a number indicating which implementation it wants. InitializeDictionary creates an instance of the appropriate kind of module and snatches its procedure entries away, putting them into the record. I was rather surprised that this worked; it was conceivable that Mesa wouldn't support the creation of free floating procedure values in this way.

The use of this intermediary seems required. This is because a frame pointer variable can be tied to only one implementation, whereas we can stuff any procedures we like into a record.

This arrangement seems to have nice "need-to-know" properties. The implementations, Dictionary1 and Dictionary1a, need not know they are being distributed. The distributor can be entirely ignorant of how the various implementations work and of how they are used. The client can distinguish between implementations only if he remembers what numbers he passed InitializeDictionary or if the implementations behave differently. The only improvement I could ask is that there be some way to guarantee that the Distributor is the only module that fills in Dictionary records. As things stand, anyone who can access a record's fields can change them as well.

This module would be even simpler if it were not for Extend. First, in order to pass one dictionary to another we must carry along the frame pointer in each record. Second, it would be wrong to pass an instance of Dictionary1 a frame pointer for an instance of Dictionary1a, since the former is not competent to fiddle with the latter's data structures. Therefore we program Extend in Distributor as a two-parameter procedure which does the right thing. What if Extend couldn't be programmed in terms of the other primitives? The completeness property discussed above guarantees that, if our backs are to the wall, we can always translate one kind of dictionary into the other kind in order to perform a binary operation on dictionaries with different implementations.

I tried to solve this problem using the implementation style of Dictionary2. The approach I followed was to declare a variant record type which was either a Dictionary2-type record or a Dictionary2a-type record. Then Fetch, Generate, and Store as well as Extend had to be written inside Distributor, and each had to branch on the variant. It was a mess.

*Multiple types* -- Dictionary3, IntegerShell and StringShell

Dictionaries that can store only integers are not very useful. In any application we would have to keep an array around to hold the real values and use the integers as indices.

A better alternative might be to give our client programmer the sources and let him edit in whatever type he likes.

An intermediate solution is to say that our module will handle any one-word type the client likes, since it seems clear we are not depending upon the numerical nature of the items we are storing. Then he can store pointers to any kind of things he likes.

The quickest way to effect this solution is to change all the INTEGER's in Dictionary1 to

UNSPECIFIED. It is not the preferred way, however, because it also turns off the the client's type checker in all those places where he invokes our module. He thinks that he is always storing Bletches in his dictionary and hopes he is getting nothing but Bletches, and might like the type checker to remind him if he tries to store or fetch something else.

We must breach the type system to solve this problem, but there is a better way to do it. The suggested method involves two independent steps.

First we change Dictionary1 into Dictionary3 by replacing INTEGER with the rather strange one-word type

```
Thing: TYPE = RECORD[a: [0..400B), b: [0..400B)]
```

which is at the other end of the type semi-lattice from UNSPECIFIED; i.e. it matches nothing but itself rather than everything. The purpose is to get the compiler to prove that we really don't depend upon the things in the array being integers. As it turns out the compiler complains about the -1 that Fetch returns, so we have to amend Initialize to take a parameter, nullValue, to play the role of -1 . After this small change, the compiler accepts Dictionary3, proving that it does not depend upon the nature of Things.

The property we're trying to establish is that if, from the day of its birth, an instance of Dictionary3 is fed nothing but Bletches then the instance will emit only Bletches. Since Thing is a unique type, private to the module we can be sure that it is not getting Things from anywhere but the intakes explicitly labelled with Thing, namely the parameters of Initialize and Store and the returned value of proc in Generate.

The second step is to make a shell module like IntegerShell whose instances sit between Dictionary3 and its clients. Its only role is to instantiate the type of the instance. When the client, ShellClient creates an instance of IntegerShell the effect is to create a new instance of both IntegerShell and Dictionary3 and plug the procedures from the latter into the former. (Again I was surprised that this was allowed and worked.) By inspection of IntegerShell we see that the only access to the intakes of this newly created instance are through procedure variables that demand integers in all the places where Dictionary3 expects Things. Therefore, we are justified in assigning integer types to all the outputs of this instance of Dictionary3.

As usual Extend requires special treatment. We must declare the Extend in IntegerShell to take only frames instantiated from IntegerShell. This seems like the only way: If we said Extend could take frames instantiated from Dictionary3 then someone could extend an integer dictionary with a string dictionary. One the other hand, passing an instance of IntegerShell to the Extend entry of an instance of Dictionary3 would have unpleasant consequences: Extend would reach for the array Pair and get random bits.

Because I use "=" initializations of the procedure names in IntegerShell, no one can alter them. Contrast this with the situation in Distributor where we stuffed the procedures into a record.

The method we use to do the plugging in IntegerShell is very dangerous because it does not guard against really gross discrepancies between the procedures, such as in how many parameters they expect. If someone re-compiles Dictionary3, changing the type of Fetch in a drastic way, we will never be warned, even when we recompile IntegerShell. I'm told that this kind of error will make the Mesa run-time system very unhappy. All we want the type checker to ignore is the apparent discrepancy between Thing and INTEGER. The method demonstrated in StringShell was suggested by Ed Satterthwaite and is much preferred. The procedure Gedanken will not compile properly if our assumptions about Dictionary3 are violated. This check is still not entirely fool-proof, e.g. df.Fetch might return a BOOLEAN and we wouldn't be warned, but it is still pretty good.

Notice that none of the modules involved has interfaces with UNSPECIFIED types. All the dirty doings are confined to the shell modules. This seems like a good policy; things are confused enough at module interfaces. It seems to be an unfortunate historical accident that type fudging is usually done between modules where the responsibility for it is unclear.

```
                    DIRECTORY SystemDefs: FROM "SystemDefs",
                              StringDefs: FROM "StringDefs",
                              UtilitiesDefs: FROM "UtilitiesDefs";
                    DEFINITIONS FROM UtilitiesDefs, SystemDefs, StringDefs;
DictionaryO: PROGRAM = -- The basic design
BEGIN
-- Types
Pindex: TYPE = WORD;
R: TYPE = RECORD [ key: STRING, value: INTEGER ];

-- Signals
DictionaryIndisposed: PUBLIC SIGNAL = CODE;
UNWIND: EXTERNAL SIGNAL;

-- Storage
indisposed: BOOLEAN ← FALSE;
nPair: Pindex ← 0;
Pair: DESCRIPTOR FOR ARRAY OF R;
Psize: INTEGER ← 10; -- just initial value, see Store

-- The module invariant:
        -- (a) 0<=nPair<=Psize
        -- (b) Pair[0..nPair) is alphabetically sorted by its keys.
        -- (c) Each string in Pair is not shared with anyone else.
-- The representation: This module represents a function from STRINGS to INTEGERS. The non -1 values are given
by the elements of Pair[0..nPair). The function is changed by Store and Generate. The function is sensed by
Fetch and Generate.

-- Procedures

Check: PUBLIC PROCEDURE =
        BEGIN i:WORD;
        IF nPair NOT IN [0..Psize] THEN ERROR;
        FOR i IN [0..nPair-1) DO
                SELECT CompareString[Pair[i].key, Pair[i+1].key] FROM
                        IN [equal..greater] => ERROR;
                        ENDCASE;
                ENDLOOP;
        FOR i IN [0..nPair) DO IF Pair[i].value = -1 THEN ERROR ENDLOOP;
        END;

Fetch: PUBLIC PROCEDURE [s: STRING ] RETURNS [INTEGER] =
        -- returns the value of the function at s
        BEGIN i:Pindex; t: BOOLEAN;
        [t, i] ← LookUp[s];
        IF t THEN RETURN [ Pair[i].value ] ELSE RETURN [-1];
        END;

Forget: PROCEDURE [ i:Pindex ] =
        -- removes the entry
        BEGIN j: Pindex;
        FreeHeapString[Pair[i].key];
        FOR j IN [i..nPair-1) DO Pair[j] ← Pair[j+1] ENDLOOP;
        nPair ← nPair -1;
        END;

Generate: PUBLIC PROCEDURE [proc:PROCEDURE[STRING, INTEGER] RETURNS [INTEGER]] =
        -- applies proc to each element, in alphabetical order, resetting the item entry.
        BEGIN
        i:Pindex;
        temp: STRING = [256];
        IF indisposed THEN BEGIN SIGNAL DictionaryIndisposed; RETURN END;
        indisposed ← TRUE;
        BEGIN ENABLE UNWIND => indisposed ← FALSE;
                i ← 0;
                WHILE i < nPair DO
```

```
                    temp.length ← 0; AppendString[temp,Pair[i].key];
                    Pair[i].value ← proc[temp, Pair[i].value];
                    IF Pair[i].value=-1 THEN Forget[i] -- decrements nPair
                    ELSE i ← i+1;
                    ENDLOOP;
    END;
    Indisposed ← FALSE;
    END;

LookUp: PROCEDURE [s: STRING] RETURNS [BOOLEAN, Pindex] =
    BEGIN i:INTEGER; lower,upper,m: Pindex;
    lower←0; upper←nPair-1;
     -- Pair[0..lower).key are all less than s
     -- Pair(upper..nPair).key are all greater than s.
     -- lower <= upper+1
     -- The interval [lower..upper] decreases with each iteration.
    UNTIL lower=upper+1 DO
        m←(lower+upper)/2; -- lower<=m<=upper
        SELECT CompareString[s, Pair[m].key] FROM
            equal => RETURN [ TRUE, m];
            less, prefix* => upper ← m-1;
            extension, greater => lower ← m+1;
            ENDCASE;
        ENDLOOP;
    -- Thus s is not in the interval Pair[0..nPair).
    RETURN [FALSE, lower]; -- lower is the first element greater than s
    END;

Store: PUBLIC PROCEDURE [ s: STRING, x: INTEGER ] =
    BEGIN place, j: Pindex; t: BOOLEAN;
    -- makes the function at s have value x
    -- remove entry if x is -1
    newPair: DESCRIPTOR FOR ARRAY OF R;
    IF indisposed THEN BEGIN SIGNAL DictionaryIndisposed; RETURN END;
    [t, place] ← LookUp[s];
    IF t THEN BEGIN IF x = -1 THEN Forget[place] ELSE Pair[place].value ← x END
    ELSE IF x#-1 THEN
        BEGIN
        IF nPair=Psize THEN
            BEGIN
            Psize ← IF Psize<1000 THEN 2*Psize ELSE Psize+1000;
            newPair ←
            DESCRIPTOR [ AllocateHeapNode[SIZE[R]*Psize], Psize];
            FOR j IN [0..nPair) DO newPair[j] ← Pair[j] ENDLOOP;
            FreeHeapNode[BASE[Pair]];
            Pair ← newPair;
            END;
        FOR j DECREASING IN [place..nPair) DO Pair[j+1] ← Pair[j] ENDLOOP;
        Pair[place]← R[CopyString[s], x];
        nPair ← nPair + 1;
        END;
    END;

-- Initialization
Pair ← DESCRIPTOR [ AllocateHeapNode[SIZE[R]*Psize], Psize];
    -- we have to do this here because AllocateHeapNode is unbound during the initialization above.
END.
```

```
                                DIRECTORY IoDefs: FROM "IoDefs",
                                        StringDefs: FROM "StringDefs",
                                        UtilitiesDefs: FROM "UtilitiesDefs",
                                        Dictionary0: FROM "Dictionary0";
                                DEFINITIONS FROM IoDefs,StringDefs,
                                        UtilitiesDefs

DictionaryClient0: PROGRAM  = --  This module drives the Dictionary
BEGIN
-- Storage & Constants & Signals
Dict: POINTER TO FRAME[Dictionary0];
Numbers: ARRAY [0..10] OF STRING =
["zero", "one", "two", "three", "four", "five", "six", "seven", "eight", "nine", "ten" ];
i: INTEGER;
UNWIND: EXTERNAL SIGNAL;

-- Procedures
Murphy: PROCEDURE [s: STRING, x: INTEGER] RETURNS [INTEGER] =
        --  what an unpleasant client could do
        BEGIN OPEN Dict;
        WriteString[s];
        WriteChar[SP];
        IF EqualString[s, "ten"] THEN SIGNAL UNWIND;
        IF Fetch[s] MOD 2 = 0 THEN RETURN [ -1];
        s[0] ← 'z; -- mess up string
        RETURN [x]
        END;
PrintPair: PROCEDURE [s: STRING, x: INTEGER] RETURNS [INTEGER] =
        BEGIN OPEN Dict;
        WriteString[s];
        WriteString[" = "];
        WriteDecimal[x];
        WriteChar[SP];
        RETURN [x] --  keep going
        END;
-- Code

 -- create and bind the dictionary module
Dict ← NEW Dictionary0;-- create space
BIND Dict; -- bind its externals (like AllocateHeapNode)
START Dict; -- intialize it
BEGIN OPEN Dict;

FOR i IN [0..10] DO Store[Numbers[i],i] ENDLOOP;

Generate[PrintPair]; WriteChar[CR];
        -- should print "eight =  8 five =  5 four =  4 nine =  9 one =  1 seven =  7 six =  6 ten = 10 three
        = 3 two =  2 zero = 0 "

Store["one", -1]; Store["two", -1]; Store["three", -1]; Store["four", -1]; Store["five", -1];

Generate[PrintPair]; WriteChar[CR];
-- should print "eight =  8 nine =  9 seven =  7 six =  6 ten = 10 zero = 0 "

BEGIN
Generate[Murphy ! UNWIND => GOTO Stop];
EXITS
Stop => WriteChar[CR] -- emerges here after Murphy["ten",10]
END;
        -- should print "eight =  8 nine =  9 seven =  7 six =  6 ten = 10"

Generate[PrintPair]; WriteChar[CR];
-- should print "nine = 9 seven = 7 ten = 10 zero = 0"
END
END.
```

```
                                DIRECTORY SystemDefs: FROM "SystemDefs",
                                          StringDefs: FROM "StringDefs",
                                          UtilitiesDefs: FROM "UtilitiesDefs";
                                DEFINITIONS FROM UtilitiesDefs, SystemDefs, StringDefs;
Dictionary1: PROGRAM = --A slight alteration of Dictionary0 to support multiple instances
BEGIN
-- Types (same as Dictionary0)
Pindex: TYPE = WORD;
R: TYPE = RECORD [ key: STRING, value: INTEGER ];

-- Signals (same as Dictionary0)
DictionaryIndisposed: PUBLIC SIGNAL = CODE;
UNWIND: EXTERNAL SIGNAL;

-- Storage (same as Dictionary0)
indisposed: BOOLEAN ← TRUE;
nPair: Pindex;
Pair: DESCRIPTOR FOR ARRAY OF R;
Psize: INTEGER;

-- The module invariant: if ~indisposed
        -- (a) 0<=nPair<=Psize
        -- (b) Pair[0..nPair) is alphabetically sorted by its key components.
        -- (c) Each string in Pair is not shared with anyone else.
-- The representation: An instance of this module represents a function from STRINGS to INTEGERS. The non -1
values are given by the elements of Pair[0..nPair). The function is changed by Store and Generate. The
function is sensed by Fetch and Generate.

-- Procedures

Check: PUBLIC PROCEDURE =
        BEGIN i:WORD;
        IF indisposed THEN RETURN;   -- added
        IF nPair NOT IN [0..Psize] THEN ERROR;
        FOR i IN [0..nPair-1) DO
                SELECT CompareString[Pair[i].key, Pair[i+1].key] FROM
                        IN [equal..greater] => ERROR;
                        ENDCASE;
                ENDLOOP;
        FOR i IN [0..nPair) DO IF Pair[i].value = -1 THEN ERROR ENDLOOP;
        END;

Extend: PUBLIC PROCEDURE [source: POINTER TO FRAME [Dictionary1] ] =
        BEGIN
        nPairBound: Pindex = nPair + source.nPair;
        newPsize: Pindex = nPairBound + nPairBound/4;
        newPair: DESCRIPTOR FOR ARRAY OF R;
        i, si, ni: Pindex ← 0;
        IF indisposed OR source.indisposed THEN BEGIN SIGNAL DictionaryIndisposed; RETURN END;
        newPair ← DESCRIPTOR[ AllocateHeapNode[SIZE[R]*newPsize], newPsize];
        DO -- Invariant: Pair[0..i) and source.Pair[0..si) have been merged into newPair[0..ni)
                IF i=nPair THEN GOTO FlushSource;
                IF si=source.nPair THEN GOTO FlushSink;
                SELECT CompareString[Pair[i].key, source.Pair[si].key ] FROM
                        equal =>
                                BEGIN -- extension takes precedence
                                newPair[ni] ← R[Pair[i].key, source.Pair[si].value]; -- reuse name
                                si ← si+1;
                                i ← i+1;
                                END;
                        less, prefix =>
                                BEGIN
                                newPair[ni] ← Pair[i];
                                i ← i+1;
                                END;
                        extension, greater =>
```

```
                                BEGIN
                                newPair[ni] ← R[CopyString[source.Pair[si].key], source.Pair[si].value];
                                si ← si+1;
                                END;
                        ENDCASE;
                        ni ← ni+1;
        REPEAT
                FlushSink =>
                        FOR i IN [i..nPair) DO
                                newPair[ni] ← Pair[i];
                                ni ← ni + 1;
                                ENDLOOP;
                FlushSource =>
                        FOR si IN [si..source.nPair) DO
                                newPair[ni] ← R[CopyString[source.Pair[si].key], source.Pair[si].value];
                                ni ← ni + 1;
                                ENDLOOP;
        ENDLOOP;
        FreeHeapNode[BASE[Pair]];
        Pair ← newPair;
        nPair ← ni;
        Psize ← newPsize;
        END;

Fetch: PUBLIC PROCEDURE [s: STRING ] RETURNS [INTEGER] =
        -- returns the value of the function at s
        BEGIN i:Pindex; t: BOOLEAN;
        IF indisposed THEN BEGIN SIGNAL DictionaryIndisposed; RETURN [-1] END; --added
        [t, i] ← LookUp[s];
        IF t THEN RETURN [ Pair[i].value ] ELSE RETURN [-1];
        END;

Finalize: PUBLIC PROCEDURE =
        -- prepare to die
        BEGIN i: Pindex;
        IF indisposed THEN BEGIN SIGNAL DictionaryIndisposed; RETURN END;
        FOR i IN [0..nPair) DO FreeHeapString[Pair[i].key] ENDLOOP;
        FreeHeapNode[BASE[Pair]];
        indisposed ← TRUE;
        RETURN;
        END;

-- Forget  same as Dictionary0
-- Generate same as  Dictionary0

Initialize: PUBLIC PROCEDURE =
        BEGIN
        IF ~indisposed THEN Finalize[];
        Psize ← 10;
        Pair ← DESCRIPTOR [ AllocateHeapNode[SIZE[R]*Psize], Psize];
        nPair ← 0;
        indisposed ← FALSE;
        RETURN;
        END;

-- LookUp same as  Dictionary0
-- Store same as  Dictionary0

-- Initialization
Initialize[];
END.
```

```
                                DIRECTORY IoDefs: FROM "IoDefs",
                                    UtilitiesDefs: FROM "UtilitiesDefs",
                                    Dictionary1: FROM "Dictionary1";
                                DEFINITIONS FROM IoDefs, UtilitiesDefs;

DictionaryClient1: PROGRAM =
--    This module drives multiple instances of Dictionary1
BEGIN
-- Storage & Constants

Dict1, Dict2, Dict3: POINTER TO FRAME[Dictionary1];
i : INTEGER;
Numbers: ARRAY [0..10] OF STRING =
["zero", "one", "two", "three", "four", "five", "six", "seven", "eight", "nine", "ten" ];

-- Procedures

PrintPair: PROCEDURE [s: STRING, x: INTEGER] RETURNS [INTEGER] =
        BEGIN
        WriteString[s];
        WriteString[" = "];
        WriteDecimal[x];
        WriteChar[SP];
        RETURN [x] --  keep going
        END;

-- Code

Dict1 ← NEW Dictionary1; BIND Dict1; START Dict1;
Dict2 ← NEW Dictionary1; BIND Dict2; START Dict2;
Dict3 ← NEW Dictionary1; BIND Dict3; START Dict3;


FOR i IN [0..7] DO Dict1.Store[Numbers[i],i] ENDLOOP;
FOR i IN [4..10] DO Dict2.Store[Numbers[i],-i] ENDLOOP;

Dict3.Extend[Dict1]; Dict3.Extend[Dict2];

Dict1.Generate[PrintPair]; WriteChar[CR];
        -- Should display "five = 5 four = 4 one = 1 seven = 7 six = 6 three = 3 two = 2 zero = 0"
Dict2.Generate[PrintPair]; WriteChar[CR];
        -- Should display "eight = -8 five = -5 four = -4 nine = -9 seven = -7 six = -6 ten = -10"
Dict3.Generate[PrintPair]; WriteChar[CR];
        -- Should display "eight = -8 five = -5 four = -4 nine = -9 one = 1 seven = -7 six = -6 ten = -10
        three = 3 two = 2 zero = 0"


Dict3.Store["three", -1]; Dict3.Store["four", -1]; Dict3.Store["five", -1];

Dict1.Generate[PrintPair]; WriteChar[CR];
        -- Should display "five = 5 four = 4 one = 1 seven = 7 six = 6 three = 3 two = 2 zero = 0"

Dict1.Finalize[];

Dict2.Generate[PrintPair]; WriteChar[CR];
        -- Should display "eight = -8 five = -5 four = -4 nine = -9 seven = -7 six = -6 ten = -10"

Dict2.Finalize[];

Dict3.Generate[PrintPair]; WriteChar[CR];
        -- Should display "eight = -8 nine = -9 one = 1 seven = -7 six = -6 ten = -10 two = 2 zero = 0"

Dict3.Finalize[];

END.
```

```
                                DIRECTORY SystemDefs: FROM "SystemDefs",
                                          StringDefs: FROM "StringDefs",
                                          UtilitiesDefs: FROM "UtilitiesDefs";
                                DEFINITIONS FROM UtilitiesDefs, SystemDefs, StringDefs;
Dictionary2: PROGRAM = -- Achieving multiple dictionaries through private records
BEGIN
-- Types
Dictionary: PUBLIC TYPE = PRIVATE RECORD[
        indisposed: BOOLEAN, -- wish I could initialize this to TRUE
        nPair: Pindex,
        Pair: DESCRIPTOR FOR ARRAY OF R,
        Psize: INTEGER
        ];
Pindex: TYPE = WORD;
R: TYPE = RECORD [ key: STRING, value: INTEGER ];

-- Signals
DictionaryIndisposed: PUBLIC SIGNAL = CODE;
UNWIND: EXTERNAL SIGNAL;

-- The module invariant: if a given record, d, is not indisposed
        -- (a) 0<=d.nPair<=d.Psize
        -- (b) d.Pair[0..d.nPair) is alphabetically sorted by its key components.
        -- (c) Each string in d.Pair is not shared with anyone else.
-- The representation: A Dictionary record represents a function from STRINGS to INTEGERS. The non -1 values are
given by the elements of d.Pair[0..d.nPair). The function is changed by Store and Generate. The function is
sensed by Fetch and Generate.

-- Procedures

Check: PUBLIC PROCEDURE [d: Dictionary] =
        BEGIN OPEN d; i:WORD;
        -- same as Dictionary1
        END;

Extend: PUBLIC PROCEDURE [sink, source: POINTER TO Dictionary] =
        -- adds all the entries of source to sink.
        BEGIN
        nPairBound: Pindex = sink.nPair + source.nPair;
        newPsize: Pindex = nPairBound + nPairBound/4;
        newPair: DESCRIPTOR FOR ARRAY OF R;
        isink, isource, inew: Pindex ← 0;
        IF source.indisposed OR sink.indisposed THEN BEGIN SIGNAL DictionaryIndisposed; RETURN END;
        newPair ← DESCRIPTOR[ AllocateHeapNode[SIZE[R]•newPsize], newPsize];
        DO -- Invariant: sink.Pair[0..isink) and source.Pair[0..isource) have been merged into newPair[0..inew)
                IF isink=sink.nPair THEN GOTO FlushSource;
                IF isource=source.nPair THEN GOTO FlushSink;
                SELECT CompareString[sink.Pair[isink].key, source.Pair[isource].key ] FROM
                        equal =>
                                BEGIN -- extension takes precedence
                                newPair[inew] ← R[sink.Pair[isink].key, source.Pair[isource].value];
                                        -- reuse name
                                isource ← isource+1;
                                isink ← isink+1;
                                END;
                        less,prefix =>
                                BEGIN
                                newPair[inew] ← sink.Pair[isink];
                                isink ← isink+1;
                                END;
                        greater, extension =>
                                BEGIN
                                newPair[inew]
                                  ← R[CopyString[source.Pair[isource].key], source.Pair[isource].value];
                                isource ← isource+1;
                                END;
```

```
                    ENDCASE;
                    inew ← inew+1;
        REPEAT
                FlushSink =>
                    FOR isink IN [isink..sink.nPair) DO
                        newPair[inew] ← sink.Pair[isink];
                        inew ← inew + 1;
                        ENDLOOP;
                FlushSource =>
                    FOR isource IN [isource..source.nPair) DO
                        newPair[inew]
                          ← R[CopyString[source.Pair[isource].key], source.Pair[isource].value];
                        inew ← inew + 1;
                        ENDLOOP;
        ENDLOOP;
        FreeHeapNode[BASE[sink.Pair]];
        sink↑ ← Dictionary[indisposed: FALSE, Pair: newPair, nPair: inew, Psize: newPsize];
        END;

Fetch: PUBLIC PROCEDURE [d: POINTER TO Dictionary, s: STRING ] RETURNS [INTEGER] =
        BEGIN OPEN d;
        -- same as Dictionary1
        END;

Finalize: PUBLIC PROCEDURE[d: POINTER TO Dictionary] =
        BEGIN OPEN d;
        -- same as Dictionary1
        END;

Forget: PROCEDURE [d: POINTER TO Dictionary, i:Pindex ] =
        BEGIN OPEN d;
        -- same as Dictionary1
        END;

Generate: PUBLIC PROCEDURE [d: POINTER TO Dictionary, proc:PROCEDURE[STRING, INTEGER] RETURNS [INTEGER]] =
        BEGIN OPEN d;
        -- same as Dictionary1
        END;

Initialize: PUBLIC PROCEDURE[d: POINTER TO Dictionary] =
        BEGIN OPEN d;
        -- same as Dictionary1
        END;

LookUp: PROCEDURE [d: POINTER TO Dictionary, s: STRING] RETURNS [BOOLEAN, Pindex] =
        BEGIN OPEN d;
        -- same as Dictionary1
        END;

Store: PUBLIC PROCEDURE [d: POINTER TO Dictionary, s: STRING, x: INTEGER ] =
        BEGIN OPEN d;
        -- same as Dictionary1
        END;
END.
```

```
                                        DIRECTORY IoDefs: FROM "IoDefs",
                                                SystemDefs: FROM "SystemDefs",
                                                UtilitiesDefs: FROM "UtilitiesDefs",
                                                Dictionary2: FROM "Dictionary2";
                                        DEFINITIONS FROM IoDefs, SystemDefs,
                                                UtilitiesDefs,Dictionary2;
DictionaryClient2: PROGRAM =
--    This module drives Dictionary2, creating multiple Dictionaries
BEGIN
-- Storage & Constants
dm: POINTER TO FRAME[Dictionary2];
d1,d2: Dictionary; -- create space
Dict1, Dict2, Dict3: POINTER TO Dictionary;
i : INTEGER;
Numbers: ARRAY [0..10] OF STRING =
["zero", "one", "two", "three", "four", "five", "six", "seven", "eight", "nine", "ten" ];

-- Procedures
PrintPair: PROCEDURE [s: STRING, x: INTEGER] RETURNS [INTEGER] =
        BEGIN
        WriteString[s];
        WriteString[" = "];
        WriteDecimal[x];
        WriteChar[SP];
        RETURN [x] --  keep going
        END;

-- Code
dm ← NEW Dictionary2; BIND dm; START dm;

BEGIN OPEN dm;
Dict1 ← @d1; Initialize[Dict1]; -- intialize pointers
Dict2 ← @d2; Initialize[Dict2];
Dict3 ← AllocateHeapNode[SIZE[Dictionary]]; Initialize[Dict3];  -- just for variety

FOR i IN [0..7] DO Store[Dict1, Numbers[i],i] ENDLOOP;
FOR i IN [4..10] DO Store[Dict2, Numbers[i],-i] ENDLOOP;

Extend[Dict3, Dict1]; Extend[Dict3, Dict2];

Generate[Dict1, PrintPair]; WriteChar[CR];
        -- Should display "five = 5 four = 4 one = 1 seven = 7 six = 6 three = 3 two = 2 zero = 0"
Generate[Dict2, PrintPair]; WriteChar[CR];
        -- Should display "eight = -8 five = -5 four = -4 nine = -9 seven = -7 six = -6 ten = -10"
Generate[Dict3, PrintPair]; WriteChar[CR];
        -- Should display "eight = -8 five = -5 four = -4 nine = -9 one = 1 seven = -7 six = -6 ten = -10
        three = 3 two = 2 zero = 0"


Store[Dict3, "three", -1]; Store[Dict3, "four", -1]; Store[Dict3, "five", -1];

Generate[Dict1, PrintPair]; WriteChar[CR];
        -- Should display "five = 5 four = 4 one = 1 seven = 7 six = 6 three = 3 two = 2 zero = 0"

Finalize[Dict1];

Generate[Dict2, PrintPair]; WriteChar[CR];
        -- Should display "eight = -8 five = -5 four = -4 nine = -9 seven = -7 six = -6 ten = -10"

Finalize[Dict2];

Generate[Dict3, PrintPair]; WriteChar[CR];
        -- Should display "eight = -8 nine = -9 one = 1 seven = -7 six = -6 ten = -10 two = 2 zero = 0"

Finalize[Dict3]; FreeHeapNode[Dict3];
END
END.
```

```
                                        DIRECTORY SystemDefs: FROM "SystemDefs",
                                                  StringDefs: FROM "StringDefs",
                                                  UtilitiesDefs: FROM "UtilitiesDefs";
                                        DEFINITIONS FROM UtilitiesDefs, SystemDefs, StringDefs;
Dictionary3: PROGRAM = -- Illustrates how to prove you're not depending upon the type of Thing
BEGIN
-- Types
Pindex: TYPE = WORD;
R: TYPE = RECORD [ key: STRING, value: Thing ];
Thing: TYPE = RECORD[a: [0..400B), b: [0..400B)];
  -- unique one-word type, matches only self, cannot come from outside this module

-- Signals same as Dictionary1
DictionaryIndisposed: PUBLIC SIGNAL = CODE;
UNWIND: EXTERNAL SIGNAL;

-- Storage
indisposed: BOOLEAN ← TRUE;
nPair: Pindex;
nullValue: Thing;
Pair: DESCRIPTOR FOR ARRAY OF R;
Psize: INTEGER; -- just initial value, see Store

-- The module invariant: if ~indisposed
        -- (a) 0<=nPair<=Psize
        -- (b) Pair[0..nPair) is alphabetically sorted by its key components.
        -- (c) Each string in Pair is not shared with anyone else.
-- The representation: An instance of this module represents a function from STRINGS to Thing. The
non-nullValue values are given by the elements of Pair[0..nPair). The function is changed by Store. The
function is sensed by Fetch and Generate.

-- Procedures

Check: PUBLIC PROCEDURE =
        BEGIN i:WORD;
        IF indisposed THEN RETURN;
        IF nPair NOT IN [0..Psize] THEN ERROR;
        FOR i IN [0..nPair-1) DO
                SELECT CompareString[Pair[i].key, Pair[i+1].key] FROM
                        IN [equal..greater] => ERROR;
                        ENDCASE;
                ENDLOOP;
        FOR i IN [0..nPair) DO
                IF Pair[i].value = nullValue THEN ERROR -- nullValue rather than -1
                ENDLOOP;
        END;

Extend: PUBLIC PROCEDURE [ d: POINTER TO FRAME [Dictionary3] ] =
        BEGIN -- same as Dictionary1
        END;

Fetch: PUBLIC PROCEDURE [s: STRING ] RETURNS [Thing] =
        -- returns the value of the function at s
        BEGIN i:Pindex; t: BOOLEAN;
        IF indisposed THEN BEGIN SIGNAL DictionaryIndisposed; RETURN [nullValue]END;
        [t, i] ← LookUp[s];
        IF t THEN RETURN [ Pair[i].value ] ELSE RETURN [nullValue];
        END;

 -- Finalize: PUBLIC PROCEDURE = same as Dictionary1

 -- Forget: PROCEDURE [ i:Pindex ] = same as Dictionary1

Generate: PUBLIC PROCEDURE [proc:PROCEDURE[STRING, Thing] RETURNS [Thing]] =
        -- applies proc to each element , in alphabetical order, resetting the item entry.
        BEGIN
```

```
        i:Pindex;
        temp: STRING = [256];
        IF indisposed THEN BEGIN SIGNAL DictionaryIndisposed; RETURN END;
        indisposed ← TRUE;
        BEGIN ENABLE UNWIND => indisposed ← FALSE;
                i ← 0;
                WHILE i < nPair DO
                        temp.length ← 0; AppendString[temp,Pair[i].key];
                        Pair[i].value ← proc[temp, Pair[i].value];
                        IF Pair[i].value=nullValue THEN Forget[i] -- decrements nPair
                        ELSE i ← i+1;
                        ENDLOOP;
        END;
        indisposed ← FALSE;
        RETURN;
        END;

Initialize: PUBLIC PROCEDURE[nv: Thing] =
        BEGIN
        IF ~indisposed THEN Finalize[];
        nullValue ← nv;
        Psize ← 10;
        Pair ← DESCRIPTOR [ AllocateHeapNode[SIZE[R]*Psize], Psize];
        nPair ← 0;
        indisposed ← FALSE;
        RETURN;
        END;

-- LookUp: PROCEDURE [s: STRING] RETURNS [BOOLEAN, Pindex] = same as Dictionary1

Store: PUBLIC PROCEDURE [ s: STRING, x: Thing ] =
        BEGIN place, j: Pindex; t: BOOLEAN;
        -- makes the function defined at s with value x if x is not nullValue
        -- remove entry if x is nullValue
        newPair: DESCRIPTOR FOR ARRAY OF R;
        IF indisposed THEN BEGIN SIGNAL DictionaryIndisposed; RETURN END;
        [t, place] ← LookUp[s];
        IF t THEN BEGIN IF x = nullValue THEN Forget[place] ELSE Pair[place].value ← x END
        ELSE IF x#nullValue THEN
                BEGIN
                IF nPair=Psize THEN
                        BEGIN
                        Psize ← IF Psize<1000 THEN 2*Psize ELSE Psize+1000;
                        newPair ←
                        DESCRIPTOR [ AllocateHeapNode[SIZE[R]*Psize], Psize];
                        FOR j IN [0..nPair) DO newPair[j] ← Pair[j] ENDLOOP;
                        FreeHeapNode[BASE[Pair]];
                        Pair ← newPair;
                        END;
                FOR j DECREASING IN [place..nPair) DO Pair[j+1] ← Pair[j] ENDLOOP;
                Pair[place]← R[CopyString[s], x];
                nPair ← nPair + 1;
                END;
        RETURN;
        END;

END.
```

```
                                                DIRECTORY Dictionary3: FROM "Dictionary3",
                                                UtilitiesDefs: FROM "UtilitiesDefs";
                                                DEFINITIONS FROM UtilitiesDefs;

IntegerShell: PROGRAM[nullValue: INTEGER] =
BEGIN
df: POINTER TO FRAME [Dictionary3] = NEW Dictionary3;

Extend: PUBLIC PROCEDURE [ d: POINTER TO FRAME [IntegerShell] ] =
        BEGIN
        df.Extend[d.df]
        END;
Fetch: PUBLIC PROCEDURE [STRING ] RETURNS [INTEGER] = COERCE[df.Fetch];
Finalize: PUBLIC PROCEDURE = COERCE[df.Finalize];
Generate: PUBLIC PROCEDURE[PROCEDURE[STRING, INTEGER] RETURNS [INTEGER]] = COERCE[df.Generate];
Initialize: PUBLIC PROCEDURE[INTEGER] = COERCE[df.Initialize];
Store: PUBLIC PROCEDURE [STRING,INTEGER] = COERCE[df.Store];

BIND df; START df;

-- This is a dangerous way to plug in the procedures! See  StringShell for a safer way.

Initialize[nullValue]

END.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

                                                DIRECTORY Dictionary3: FROM "Dictionary3",
                                                UtilitiesDefs: FROM "UtilitiesDefs";
                                                DEFINITIONS FROM UtilitiesDefs;

StringShell: PROGRAM [nullValue: STRING] =
BEGIN
df: POINTER TO FRAME [Dictionary3] = NEW Dictionary3;

Extend: PUBLIC PROCEDURE [ d: POINTER TO FRAME [StringShell] ] =
        BEGIN
        df.Extend[d.df]
        END;
Fetch: PUBLIC PROCEDURE [STRING ] RETURNS [STRING] = COERCE[df.Fetch];
Finalize: PUBLIC PROCEDURE = COERCE[df.Finalize];
Generate: PUBLIC PROCEDURE[PROCEDURE[STRING, STRING] RETURNS [STRING]] = COERCE[df.Generate];
Initialize: PUBLIC PROCEDURE[STRING] = COERCE[df.Initialize];
Store: PUBLIC PROCEDURE [STRING,STRING] = COERCE[df.Store];

Gedanken: PROCEDURE =
        -- This is a compile-time experiment to see if Dictionary3 is providing the right sort of procedures.
        This procedure is never executed!
        BEGIN
        uFetch: PUBLIC PROCEDURE [STRING ] RETURNS [UNSPECIFIED] = df.Fetch;
        uFinalize: PUBLIC PROCEDURE = df.Finalize;
        uGenerate: PUBLIC PROCEDURE[PROCEDURE[STRING, UNSPECIFIED] RETURNS [UNSPECIFIED]] = df.Generate;
        uInitialize: PUBLIC PROCEDURE[UNSPECIFIED] = df.Initialize;
        uStore: PUBLIC PROCEDURE [STRING,UNSPECIFIED] = df.Store;
        END;


BIND df; START df;

Initialize[nullValue]

END.
```

```
                                DIRECTORY IoDefs: FROM "IoDefs",
                                        UtilitiesDefs: FROM "UtilitiesDefs",
                                        IntegerShell: FROM "IntegerShell",
                                        StringShell: FROM "StringShell";
                                DEFINITIONS FROM IoDefs, UtilitiesDefs;

ShellClient: PROGRAM =
--    This module drives Dictionary3 through IntegerShell and StringShell
BEGIN
-- Storage & Constants

IntDict:POINTER TO FRAME[IntegerShell];
StrDict1, StrDict2:POINTER TO FRAME[StringShell];

i : INTEGER;
Numbers: ARRAY [0..10] OF STRING =
["zero", "one", "two", "three", "four", "five", "six", "seven", "eight", "nine", "ten" ];
Zahlen: ARRAY [0..10] OF STRING =
["null", "eins", "zwei", "drei", "vier", "funf", "sechs", "sieben", "acht", "neun", "zehn" ];

-- Procedures

IntPrint: PROCEDURE [s: STRING, x: INTEGER] RETURNS [INTEGER] =
        BEGIN
        WriteString[s]; WriteString[" = "]; WriteDecimal[x]; WriteChar[SP];
        RETURN [x]
        END;

Reverse: PROCEDURE [s: STRING, x: STRING] RETURNS [STRING] =
        BEGIN
        StrDict1.Store[x.CopyString[s]];  -- forgetting to copy was a nasty bug!
        RETURN [x]
        END;

StrPrint: PROCEDURE [s: STRING, x: STRING] RETURNS [STRING] =
        BEGIN
        WriteString[s]; WriteString[" = "]; WriteString[x]; WriteChar[SP];
        RETURN [x]
        END;
-- Code

IntDict ← NEW IntegerShell[-1]; BIND IntDict; START IntDict;
StrDict1 ← NEW StringShell[NIL]; BIND StrDict1; START StrDict1;
StrDict2 ← NEW StringShell[NIL]; BIND StrDict2; START StrDict2;

FOR i IN [0..10] DO IntDict.Store[Numbers[i],i] ENDLOOP;
FOR i IN [0..10] DO StrDict1.Store[Numbers[i],Zahlen[i]] ENDLOOP;

StrDict2.Extend[StrDict1];  StrDict2.Generate[Reverse];

IntDict.Generate[IntPrint]; WriteChar[CR];
        -- Should display " eight = 8 five = 5 four = 4 nine = 9 one = 1 seven = 7 six = 6 ten = 10 three = 3
        two = 2 zero = 0"
StrDict1.Generate[StrPrint]; WriteChar[CR];
        -- Should display "acht = eight drei = three eight = acht eins = one five = funf four = vier funf = five
        neun = nine nine = neun null = zero one = eins sechs =  six seven = sieben sieben = seven six =
        sechs ten = zehn three = drei two = zwei vier = four zehn = ten zero = null zwei = two"
StrDict2.Generate[StrPrint]; WriteChar[CR];
        -- Should display "eight = acht five = funf four = vier nine = neun one = eins seven = sieben six =
        sechs ten = zehn three = drei two = zwei zero = null"
StrDict1.Store["vier", NIL]; StrDict1.Store["four", NIL];
StrDict1.Generate[StrPrint]; WriteChar[CR]; StrDict1.Finalize[];
        -- Should display "acht = eight drei = three eight = acht eins = one five = funf funf = five neun = nine
        nine = neun null = zero one = eins sechs =  six seven = sieben sieben = seven six = sechs ten =
        zehn three = drei two = zwei zehn = ten zero = null zwei = two"
END;
END.
```

## Example 3. Compacting storage allocators

Let us design a series of space allocators that rearrange the storage occasionally to make room for a new array. This exercise is interesting for a number of reasons:

a) It taxes the Mesa type system severely. We must deal with an array containing variable length, heterogeneous objects. Furthermore, the clients of the allocator wish to use it for arrays of differing types.

b) As a programming exercise it can involve tricky pointer manipulations. We would like to use the type system to help us detect programming errors such as the ubiquitous address/contents confusion.

c) There are some nasty kinds of bugs associated with the use of such packages which the language might help us discourage. First, as with all free space allocators, someone might use some space after he has apparently relinquished it. Second, peculiar to compacting allocators, he can squirrel away a pointer to storage that the compacter might move.

We shall present a series of allocators, culminating with a version of Larry Tesler's Rack allocator which tortures the Mesa type system much as its namesakes tortured heretics!

*A simple allocator* -- ArrayStore0

First let us consider a naive, Algol style solution. We shall assume, unrealistically, that the client wants arrays of integers only. The module ArrayStore0 maintains a storage area, Storage, and a table of indices, Table. These arrays are private to the module and no pointer to them is ever created (much less passed outside). Thus, ignoring acts of God, all access to the arrays is through the procedures Fetch, Store, and Length. This means that the long comment is true every time the module is entered and the Check procedure never signals error, independently of what happens outside.

What happens if a client uses an array after relinquishing it? If he is lucky an error will occur because the Table entry for the array has been set to -1. If he is unlucky a subsequent allocation has used that entry for a new array and his program will charge on without an immediate error. In either case, ArrayStore1 keeps functioning happily. Thus from a "module-centric" point of view we have solved the relinquishment problem, but in the more global sense we have not.

On the other hand the pointer-squirreling violation has been made impossible. Since Fetch,

Store, and Length all use Table, which is adjusted to compensate for compactions, we know that the occurrence of a compaction will not be noticed by anyone outside the module.

ArrayClient0 shows how this module might be used.

This solution has little to recommend it other than its simplicity and lack of type violations. In fact, living within the type system has decreased the reliability of the code! Because we declared everything to be an INTEGER or, equivalently, a subrange of the integers, ArrayStore0 can make many mistakes about which the type checker will never complain. For example, we can change any "+2" (or "+ovh") to a "+1", or vice-versa. We can change any occurrence of "Table[p]" to "Storage[p]" or even "p", and there will be no complaint because they all have the same type. Another problem is that the client can pass any integer he likes as a TableIndex, rather than ones he has previously received from AllocArray. It seems that declaring everything to be INTEGER is nearly as bad as declaring everything UNSPECIFIED.

*Provide different types of arrays* -- ArrayStore1

Let us change the program to provide arrays of Things, an arbitrary one-word type of the sort described in Example 2. We must now face up to the fact that some of those INTEGERS in Storage were really of a different type. The long comment explains the situation. The only way to force things entirely into the type system would be to define Storage as follows:

```
Storage: ARRAY StorageIndex OF union;
union: TYPE = RECORD [SELECT tag:* FROM
            backPointer => [ backPointer: LineIndex ],
            length => [ length: INTEGER ],
            element => [ element: Thing ],
            ENDCASE ]
```

Then we would be branching on the type of the elements of Storage all the time, even when we knew which kind we had. It would be horrible. It seems better to use a type loophole and be careful. Specifically, we use the function COERCE to convert Things to INTEGERS on their way in and back on their way out of the module. The revised program is ArrayStore1. Aside from the addition of declarations for Thing and COERCE, the only changes are to Fetch and Store.

I feel it is better to use the COERCE function than the more obvious method of declaring the elements of Storage to be UNSPECIFIED. If I did, I would get much less checking for my money; e.g. I could say really crazy things like

Storage[i] ← Fetch

and no one would complain. Using COERCE pinpoints the places where the type checker would have complained, and draws the reader's attention to them.

Another way to breach the type system would be to define Thing to be UNSPECIFIED. Declaring Thing to be UNSPECIFIED would not decrease the quality of checking inside ArrayStore1. It would decrease it in all the clients of ArrayStore1. This would be much more pernicious, in my opinion. For example, a client could say

```
x:INTEGER; y: POINTER TO STRING;
a: INTEGER = AllocArray[10];
...
Store[a,1,x]; y←Fetch[a,1]
```

to perform a sneaky type conversion. If he really wanted to do that he could have used COERCE himself. While it may be all right to turn off the type checking in the module you are writing, it is not very nice to turn it off in everyone else's without even telling them!

ArrayClient1 shows how to make type specific procedures in the same way that StringShell from Example 2 does it. I have declared three distinct sets of procedures to work on the three distinct types, IntArray, StrArray, and MixedArray. The rationale behind this particular way of breaching the type system is the following: the part of the program that follows the "no breaches below here" comment is usually long and complicated, so we wish to apply the full force of the type checker to it. Specifically, we would like to have a mechanical check that we are not mixing up INTEGERS and STRINGS as we fiddle with arrays of them. Using lots of loopholes in that code is likely to make things even more confusing and error-prone. Therefore we commit the breach once and for all when linking to the procedures, satisfy ourselves that things will work properly, and swear off loopholes for the rest of the program.

Notice that one of the array types, MixedArray, has elements of UNSPECIFIED type. This is perfectly fine with me, in contrast to declaring Thing UNSPECIFIED, since the client knows he is doing it.

How do we satisfy ourselves that things will work properly? Consider FetchInt. We have claimed that it is a "PROCEDURE [IntArray, INTEGER] RETURNS [INTEGER]" while m.Fetch from ArrayStore1 is a "PROCEDURE [INTEGER, INTEGER] RETURNS [Thing]" How can we be sure that FetchInt always returns an INTEGER, when it appears that it *never* returns one? First, we have to believe that ArrayStore1 is implemented properly; i.e. that it keeps the various arrays separate and doesn't mix up their values. Second, we must assume that after this initial

flurry of loopholes we are not going to violate the type system again. Then we reason as follows: IntArray is a type distinct from StrArray and MixedArray. FetchInt can only be applied to IntArrays. IntArrays can get elements put into them only by StoreInt, and StoreInt accepts only INTEGER. (Of course, if we use FetchInt to access an element we have not stored into yet we shall get what we deserve.) The assignment of NIL to the frame pointer m is just an extra precaution to assure that no later part of the program can call m.Store directly.

It seems too bad that we need type-specific procedures for finding the length and for freeing an array, but the alternative of defining procedures which take UNSPECIFIED parameters seems a little risky.

Notice that ArrayClient1 could work just as well with ArrayStore0 because it makes no mention of the type Thing. The only reason to give the client ArrayStore1 is to emphasize that it will work on arbitrary one word types.

Another improvement is that, after the type breaches are over, it is impossible for ArrayClient1 to pass anything except a TableIndex to Fetch, Store, Length, or Free.

*Pointerize the indices* -- ArrayStore2:

Now we shall re-write the program changing all the indices into absolute pointers. This has a number of effects, most of them beneficial:

1) Access to the elements is a mite faster (and the code a bit shorter), since the base of the array needn't be added to the pointer.

2) We can more easily use a record declaration to describe the layout of an array in Storage. Thus Storage[Table[p]+1] becomes p↑.length (which really means p↑↑.length) a much less error-prone expression.

3) Many potential address/contents errors will now be checked for. The situation we had before where p, Table[p], and Storage[p] were all of the same type has changed: p, p↑, and p↑↑ all have different types, ArrayPtr, PR, and R, respectively.

4) We must perform arithmetic on pointers as in

```
source ← source + n + ovh
```

You might think, as I did, that this is in bad taste; but it is not much worse than array indexing without bounds checking. (The phrase MACHINE DEPENDENT increases our confidence that the arithmetic makes sense.) In fact, the type checker is rather

scrupulous about preserving the type information: If source is a PR, so is source+n. In this program I have chosen the convention that a PR is any PW that points to the beginning of a record in Storage, or where a record might reasonably begin. It is necessary to check by eye that statements such as the above that assign to variables of type PR preserve this property. The loop

```
FOR k IN [0..source.length+ovh) DO (sink+k)↑ ← (source+k)↑ ENDLOOP;
```

is a little worrisome because the type checker thinks that sink+k is of type PR while, by our informal definition, it is not. We could reduce such worry if we were willing to write the equivalent statements:

```
sink.backp ← source.backp;
sink.length ← source.length;
FOR k IN [0..sink.length) DO sink.a[k] ← source.a[k] ENDLOOP;
```

Formally speaking I could have eliminated the type PW by replacing it with PR everywhere. Doing so would have eliminated some hassle, but I feel it is good to be reminded that not every word in Storage is the start of a record.

Notice that we must use home-made procedures, gt and ge (defined in Utilities), to compare pointers. If Mesa provided them it would probably insist that the pointers be of the same type, in which case we would have to convert some PR's to PW's. Here we have been a little lazy, letting gt and ge take UNSPECIFIED values.

Despite this rather massive perturbation, the program ArrayClient1 can happily use this new module without changing anything save its directory section. *Viva* modularity!

However, now that we are passing absolute pointers to the client he can by-pass the Fetch, Store, and Length procedures and access the arrays directly as shown in ArrayClient2. There is a clear gain in speed to compensate for the loss in safety. It is now the client's responsibility to see that subscripts are in bounds, etc. Thus ArrayStore2 and the truth of its invariant are susceptible to programming errors on the part of the client.

However, somewhat surprisingly, one of the bugs we are most concerned about -- squirreling away a raw pointer -- cannot happen as long as the client does not commit any further breaches of the type system. The trick is in the way we declared IntArray -- all in one mouthful. That makes it impossible for anyone to declare a variable to hold a raw pointer. This is because (as mentioned before) every occurrence of the type constructor RECORD generates a new type, distinct from all other types. Therefore, even if we should declare

```
rawPointer: POINTER TO MACHINE DEPENDENT
```

```
RECORD [NoNo: IntArray, length: INTEGER, a: ARRAY[0..0] OF UNSPECIFIED ];
```

we could not perform the assignment "rawpointer ← IA↑" because IA↑ has a different type even though it looks the same. If one cannot declare the type of IA↑ it is rather difficult to hang onto it for very long. In fact, we believe that it is impossible for any type-checked program to hold such a pointer across a procedure call, but this requires a rather detailed study of the compiler.

Now, you ask, what gives us the right to violate the type system in one place and expect it to be obeyed elsewhere? The answer is that we don't really expect it to be obeyed elsewhere all the time, but we do expect the programmer to proceed with caution when disobeying it. Recall that our original problem was simply one of forgetfulness -- forgetting that certain pointers become invalid if a compaction occurs. Hopefully, if one goes to the trouble of saying

```
rawPointr ← COERCE[ IA↑ ]
```

he will also take the trouble to worry about compactions.

*A clever compacter* -- ArrayStore3:

As a first real step towards the Rack program let us eliminate the permanent back-pointer fields. The trick is to reverse the pointers that go from Table to Storage by temporarily putting the length part of the record in the Table entry. I am not very pleased with the encoding method used -- using negative indices as back-pointers -- but at least it is entirely confined to the Compact procedure.

A suitable client for this revision, and the ones that follow, can be derived from ArrayClient2 by removing the NoNo field from all the record declarations.

*Make a free space list* -- ArrayStore4:

To avoid the need to search for a free entry in Table, we chain them all together on a list. The contents of a Table entry can now be either a pointer into Storage or, if it is free, a pointer into Table. I chose to cope with this by inventing a variant record type, Finger (Tesler's name), and using Mesa's computed variant feature (which is a sort of controlled loophole). If I had used a genuine variant record, with a bit to discriminate, the entries in Table would have grown to two words. The procedure FingerType is used to discriminate between the two types by detecting which array a Finger points into. It is recommended that one put the procedure next to the record declaration and always use it to compute the

variant in SELECT statements. Notice that there is one place, in AllocArray, where we must test the variant of DeadFingerList in order to keep the type checker happy even though we know that it must be dead, barring acts of God. I am tempted to replace the FingerType[DeadFingerList↑] by dead here. In an earlier version of this program, I tried declaring DeadFingerList and the next variant of Finger to have type dead Finger since it happened to be true. This made things messier because I had to use COERCE in AllocArray and FreeArray when fiddling with DeadFingerList.

Chaining all the free entries together (instead of leaving them NIL) makes the effect of relinquishment errors rather devastating. Is the increase in allocation speed worth it?

*Grow* Storage *and* Table *in the same array,* Rack -- ArrayStore5

As a final step, we merge the two arrays so that there is only one ceiling to bump against. This complicates things a little more. We use a slightly different representation during compaction, and describe the temporary state of a deformed record by a type definition in Compact.

The final version seems rather overburdened with type declarations and COERCES. In practice, given the small size of the program, I think a more liberal use of UNSPECIFIED would be in order. Nevertheless, this program is of interest since it shows how to deal with an extremely fluid type situation. Such extreme violations of the type system should never occur in practice.

```
ArrayStore0: PROGRAM =

BEGIN
Storage: ARRAY StorageIndex OF INTEGER;
                        -- Storage will hold a mixture of things, see the description below.
StorageIndex: TYPE = [0..StorageSize);
StorageSize : INTEGER = 2000;
nStorage: StorageIndex ←0;-- next available space in Storage

Table: ARRAY TableIndex OF StorageIndex;
TableIndex: TYPE = [0..TableSize);
TableSize: INTEGER = 500;

ovh: INTEGER = 2; -- overhead per array represented, backpointer and length

-- The sub-array Storage[0..nStorage) consists of m sequences each with the form <bp,n,e0,...,e(n-1)>, where
n(>=0) varies from sequence to sequence. If bp is not -1 then Table[bp] is the index of the element of
Storage containing bp. The array represented by the sequence is <e0,...e(n-1)>. Conversely, if Table[i] is not
-1, it is the index of the first element (bp) of one of these sequences. See Check for a precise statement of
how things should be.

-- Procedures

AllocArray: PUBLIC PROCEDURE [n:INTEGER] RETURNS [new: TableIndex] =
        BEGIN i:INTEGER;
        IF n<0 THEN ERROR;

        -- find some space
        IF n+ovh > StorageSize-nStorage THEN
                BEGIN
                Compact[];
                IF n+ovh > StorageSize-nStorage THEN ERROR;
                END;

        -- Find a table entry
        FOR new IN TableIndex DO
                IF Table[new]=-1 THEN EXIT
                REPEAT
                FINISHED => ERROR
        ENDLOOP;
        Table[new] ← nStorage; -- put indirect pointer in table

        -- initialize the array storage
        Storage[nStorage] ← new; -- the back pointer
        Storage[nStorage+1] ← n; -- the length
        FOR i IN [nStorage+ovh..nStorage+ovh+n) DO Storage[i]←0; ENDLOOP;
                                -- zero his array for him

        nStorage←nStorage+n+ovh; -- move available pointer

        RETURN;
        END;

Check: PUBLIC PROCEDURE =
        BEGIN
        i, length: WORD;
        InUse: ARRAY TableIndex OF BOOLEAN;
        FOR i IN TableIndex DO InUse[i] ← FALSE ENDLOOP;

        -- check layout of Storage and backpointers
        IF nStorage NOT IN [0..StorageSize] THEN ERROR;
        FOR i ← 0, i+Storage[i+1]+ovh UNTIL i>=nStorage DO
                IF Storage[i+1] < 0 THEN ERROR;
                IF Storage[i] # -1 THEN -- array is alive
                        BEGIN
                        IF Table[Storage[i]] # i THEN ERROR;
                        InUse[Storage[i]] ← TRUE;
```

```
                END;
            ENDLOOP;
        IF i#nStorage THEN ERROR;

        -- Are any pointers wrong
        FOR i IN TableIndex DO
            IF ~InUse[i] AND Table[i]#-1 THEN ERROR;
            ENDLOOP;
        END;

Compact: PROCEDURE =
        BEGIN
        source,sink: StorageIndex;
        length, k: INTEGER;
        sink ← 0;
        FOR source ← 0, source+length+ovh UNTIL source>=nStorage DO
            length ← Storage[source+1]; -- copy to avoid clobber below
            IF Storage[source]#-1 THEN -- sequence is in use
                BEGIN
                Table[Storage[source]] ← sink; -- adjust pointer
                FOR k IN [0..length+ovh) DO Storage[sink+k]←Storage[source+k]; ENDLOOP;
                sink ← sink+length+ovh;
                END;
            ENDLOOP;
        nStorage←sink;
        RETURN
        END;

Fetch: PUBLIC PROCEDURE [p:TableIndex,i:INTEGER] RETURNS [INTEGER] =
        BEGIN IF Table[p]=-1 OR i NOT IN [0..Storage[Table[p]+1]) THEN ERROR;
        RETURN [Storage[Table[p]+i+ovh]]
        END;

FreeArray: PUBLIC PROCEDURE [p:TableIndex] =
        BEGIN IF Table[p]=-1 THEN ERROR;
        Storage[Table[p]] ← -1;
        Table[p] ← -1;
        RETURN;
        END;

Length: PUBLIC PROCEDURE [p:TableIndex ] RETURNS [INTEGER] =
        BEGIN IF Table[p]=-1 THEN ERROR;
        RETURN [Storage[Table[p]+1]]
        END;

Store: PUBLIC PROCEDURE [p:TableIndex, i:INTEGER, v:INTEGER ] =
        BEGIN IF Table[p]=-1 OR i NOT IN [0..Storage[Table[p]+1]) THEN ERROR;
        Storage[Table[p]+i+ovh] ←v;
        RETURN.
        END;


-- Initialization

i:TableIndex;
FOR i IN [0..TableSize) DO Table[i] ← -1 ENDLOOP;

END.
```

```
DIRECTORY ArrayStore0: FROM "ArrayStore0";

ArrayClient0: PROGRAM =

BEGIN

x,y : INTEGER;

i: INTEGER;

m: POINTER TO FRAME[ArrayStore0] = NEW ArrayStore0;
START m;
BEGIN OPEN m;

x ← AllocArray[10];
y ← AllocArray[20];

FOR i IN [0..Length[x]) DO Store[x,i,2*i] ENDLOOP;
FOR i IN [0..Length[y]) DO Store[y,i,Fetch[x,i/2]] ENDLOOP;

FreeArray[x];

END;

END.
```

```
ArrayStore1: PROGRAM =
  -- generalize to hand arbitrary one-word types
BEGIN
Thing: TYPE = RECORD [a: [0..400B), b: [0..400B)];
  -- unique one-word type, matches only self, cannot come from outside this module


Storage: ARRAY StorageIndex OF INTEGER;
                        -- Storage will hold a mixture of things, see the description below.
StorageIndex: TYPE = [0..StorageSize);
StorageSize : INTEGER = 2000;
nStorage: StorageIndex ←0;-- next availble space in Storage

Table: ARRAY TableIndex OF StorageIndex;
TableIndex: TYPE = [0..TableSize);
TableSize: INTEGER = 500;

ovh: INTEGER = 2;   -- overhead for represented array, backpointer and length


-- The sub-array Storage[0..nStorage) consists of m sequences each with the form <bp,n,e0,...,e(n-1)>, where
n(>=0) varies from sequence to sequence. If bp is not -1 then Table[bp] is the index of the element of
Storage containing bp. The array represented by the sequence is <e0,...e(n-1)>. Conversely, if Table[i] is not
-1, it is the index of the first element (bp) of one of these sequences. See Check for a precise statement of
how things should be.

-- Procedures

-- AllocArray same as in ArrayStore0

-- Check same as in ArrayStore0

-- Compact same as in ArrayStore0

Fetch: PUBLIC PROCEDURE [p:TableIndex,i:INTEGER] RETURNS [Thing] =
        BEGIN IF Table[p]=-1 OR i<0 OR i>=Storage[Table[p]+1] THEN ERROR;
        RETURN [COERCE[Storage[Table[p]+i+ovh]]]
          -- Breach: convert INTEGER to Thing
        END;

-- FreeArray same as in ArrayStore0

-- Length same as in ArrayStore0

Store: PUBLIC PROCEDURE [p:TableIndex, i:INTEGER, v:Thing ] =
        BEGIN IF Table[p]=-1 OR i<0 OR i>=Storage[Table[p]+1] THEN ERROR;
        Storage[Table[p]+i+ovh] ←COERCE[v];
          -- Breach: convert Thing to INTEGER
        RETURN;
        END;

-- Initialization

i:TableIndex;
FOR i IN [0..TableSize) DO Table[i] ← -1 ENDLOOP;

END.
```

```
DIRECTORY UtilitiesDefs: FROM "UtilitiesDefs",
        ArrayStore: FROM "ArrayStore1";
DEFINITIONS FROM UtilitiesDefs;

ArrayClient1: PROGRAM =

BEGIN

m: POINTER TO FRAME[ArrayStore] = NEW ArrayStore;

Gedanken: PROCEDURE =
    -- This procedure will fail to compile if ArrayStore does't have the right sort of procedures.  It is not
    meant to run
    BEGIN
    uFetch:PROCEDURE [UNSPECIFIED,INTEGER] RETURNS [UNSPECIFIED] = m.Fetch;
    uStore: PROCEDURE [UNSPECIFIED, INTEGER, UNSPECIFIED] = m.Store;
    uLength: PROCEDURE [UNSPECIFIED] RETURNS [INTEGER] = m.Length;
    uAllocArray: PROCEDURE [INTEGER] RETURNS [UNSPECIFIED] = m.AllocArray;
    uFreeArray: PROCEDURE [UNSPECIFIED] = m.FreeArray;
    END;
  -- Thus the procedures all have roughly the expected shape and the COERCES below are ok.

-- Integer array primitives
IntArray: TYPE = RECORD[a: [0..400B), b: [0..400B)];
FetchInt: PROCEDURE [IntArray,INTEGER] RETURNS [INTEGER] = COERCE[m.Fetch];
StoreInt: PROCEDURE [IntArray, INTEGER, INTEGER] = COERCE[m.Store];
LengthInt: PROCEDURE [IntArray] RETURNS [INTEGER] = COERCE[m.Length];
AllocIntArray: PROCEDURE [INTEGER] RETURNS [IntArray] = COERCE[m.AllocArray];
FreeIntArray: PROCEDURE [IntArray] = COERCE[m.FreeArray];

-- String array primitives
StrArray: TYPE = RECORD[a: [0..400B), b: [0..400B)];
FetchStr: PROCEDURE [StrArray,INTEGER] RETURNS [STRING] = COERCE[m.Fetch];
StoreStr: PROCEDURE [StrArray,INTEGER,STRING] = COERCE[m.Store];
LengthStr: PROCEDURE [StrArray ] RETURNS [INTEGER] = COERCE[m.Length];
AllocStrArray: PROCEDURE [INTEGER] RETURNS [StrArray] = COERCE[m.AllocArray];
FreeStrArray: PROCEDURE [StrArray] = COERCE[m.FreeArray];

-- Mixed array primitives
MixedArray: TYPE = RECORD[a: [0..400B), b: [0..400B)];
FetchMixed: PROCEDURE [MixedArray,INTEGER] RETURNS [UNSPECIFIED] = COERCE[m.Fetch];
StoreMixed: PROCEDURE [MixedArray,INTEGER,UNSPECIFIED] = COERCE[m.Store];
LengthMixed: PROCEDURE [MixedArray] RETURNS [INTEGER] = COERCE[m.Length];
AllocMixedArray: PROCEDURE [INTEGER] RETURNS [MixedArray] = COERCE[m.AllocArray];
FreeMixedArray: PROCEDURE [MixedArray] = COERCE[m.FreeArray];

IA: IntArray; SA: StrArray; MA: MixedArray;
i: INTEGER;

BIND m; START m; m ← NIL; --  cuts off any other kind of access to this instance of ArrayStore

-- no type breaches below here

IA ← AllocIntArray[100]; SA ← AllocStrArray[10]; MA ← AllocMixedArray[50];

FOR i IN [0..LengthInt[IA]) DO StoreInt[IA,i,i/3] ENDLOOP;
StoreStr[SA,0,"zero"]; StoreStr[SA,1,"one"]; StoreStr[SA,2,"two"]; StoreStr[SA,3,"surprise"]; StoreStr[SA,4,"four"];
FOR i IN [0..LengthMixed[MA]) DO
        IF i MOD 7 > 4
         THEN StoreMixed[MA, i, FetchStr[SA,i MOD 5]]
         ELSE StoreMixed[MA, i, FetchInt[IA,i]]
        ENDLOOP;
FreeIntArray[IA];
END.
```

```
DIRECTORY UtilitiesDefs: FROM "UtilitiesDefs";
DEFINITIONS FROM UtilitiesDefs;

ArrayStore2: PROGRAM =
   -- change the indices of ArrayStore1 into pointers
BEGIN

-- Types
ArrayPtr: TYPE = POINTER TO PR;
PR: TYPE = POINTER TO R;
PW: TYPE = POINTER TO W;
R: TYPE = MACHINE DEPENDENT
  RECORD [ backp: ArrayPtr, length: INTEGER, a: ARRAY [0..0] OF Thing ];
        -- We expect each field to take 1 machine word
TableIndex: TYPE = [0..TableSize);
Thing: TYPE = RECORD [a: [0..400B), b: [0..400B)];
W: TYPE = RECORD [a: [0..400B), b: [0..400B)];  -- just a machine word, not equal to Thing

-- Storage & Constants
Storage: ARRAY [0..StorageSize) OF W;
StorageSize : INTEGER = 2000;
beginStorage: PR = COERCE[@Storage[0]];
                        -- Breach: The informal requirement for being a PR is that one be a PW which
                        points to the beginning of an R as defined below in the invariant.
nStorage: PR ← beginStorage:      -- next availble space to lay an R
endStorage: PW = @Storage[StorageSize];

Table: ARRAY TableIndex OF PR;
TableSize: INTEGER = 500;
beginTable: ArrayPtr = @Table[0];
endTable: ArrayPtr = @Table[TableSize];

ovh: INTEGER = 2;   -- overhead for represented array, backpointer and length

-- The storage area [beginStorage..nStorage) consists of m Rs.  each with the form
<backp,length,e0,...,e(length-1)>. where length(>=0) varies from sequence to sequence. If backp is not NIL then
backp is an address in Table and backp↑ is the address of backp itself. The array represented by the record
is <e0,...e(n-1)>. Conversely, if Table[i] is not NIL, it is the address of one of these records.
-- See Check for a precise statement of how things should be.

-- Procedures

AllocArray: PUBLIC PROCEDURE [n:INTEGER] RETURNS [new: ArrayPtr] =
        BEGIN i:TableIndex;
        IF n<0 OR n>77777B-ovh THEN ERROR;
        -- find some space
        IF n+ovh > endStorage-COERCE[nStorage,PW] THEN
                -- Breach: demote nStorage to a PW.
                BEGIN
                Compact[];
                IF n+ovh > endStorage-COERCE[nStorage,PW] THEN ERROR;
                END;
        -- Find a table entry
        FOR i IN TableIndex DO
                IF Table[i]=NIL THEN GOTO found
                REPEAT
                found => new ← @Table[i];
                FINISHED => ERROR
                ENDLOOP;
        new↑ ← nStorage;
        -- initialize the array storage
        new↑.backp ← new;
        new↑.length ← n;
        nStorage←nStorage+n+ovh;
                -- Breach: move available pointer.  nStorage is still a good PR because we move it just the
```

```
            right distance past the new R.
        RETURN
        END;

Check: PUBLIC PROCEDURE =
        BEGIN
        i, length: WORD;p: PR;
        InUse: ARRAY TableIndex OF BOOLEAN;
        FOR i IN TableIndex DO InUse[i] ← FALSE ENDLOOP;

        -- check layout of Storage and backpointers
        IF gt[beginStorage, nStorage] OR gt[nStorage, endStorage] THEN ERROR;
        FOR p ← beginStorage, p+p.length+ovh UNTIL ge[p, nStorage] DO
                -- Breach: adding to p
            IF p.length<0 THEN ERROR;
            IF p.backp # NIL THEN -- array is alive
                    BEGIN
                    IF p.backp↑ # p THEN ERROR;
                    InUse[p.backp-@Table[0]] ← TRUE;
                    END;
            ENDLOOP;

        IF p#nStorage THEN ERROR;

        -- Are any pointers wrong
        FOR i IN TableIndex DO
            IF ~InUse[i] AND Table[i]#NIL THEN ERROR;
            ENDLOOP;
        END;

Compact: PROCEDURE =
        BEGIN
        source,sink: PR;
        length, k: INTEGER;
        sink ← beginStorage;
        FOR source ← beginStorage, source+length+ovh UNTIL ge[source, nStorage] DO
                    -- Breach: source remains a good PR
            length ← source.length; -- copy to avoid ambush below
            IF source.backp#NIL THEN -- record is in use
                    BEGIN
                    source.backp↑ ← sink;-- adjust pointer
                    FOR k IN [0..length+ovh) DO (sink+k)↑ ← (source+k)↑ ENDLOOP;--Breach
                    sink ← sink+length+ovh;  -- Breach: sink remains a good PR
                    END;
            ENDLOOP;

        nStorage ←sink;
        RETURN
        END;

Fetch: PUBLIC PROCEDURE [p:ArrayPtr,i:INTEGER] RETURNS [Thing] =
        BEGIN IF p↑=NIL OR i NOT IN [0..p↑.length) THEN ERROR;
        RETURN [p↑.a[i]]
        END;

FreeArray: PUBLIC PROCEDURE [p:ArrayPtr] =
        BEGIN IF p↑=NIL THEN ERROR; -- array already free
        p↑.backp ← NIL;
        p↑ ← NIL;
        RETURN;
        END;

Length: PUBLIC PROCEDURE [p:ArrayPtr ] RETURNS [INTEGER] =
        BEGIN IF p↑=NIL THEN ERROR;
        RETURN [ p↑.length ]
        END;
```

```
Store: PUBLIC PROCEDURE [p:ArrayPtr, i:INTEGER, v:Thing ] =
        BEGIN IF p↑=NIL OR i NOT IN [0..p↑.length) THEN ERROR;
        p↑.a[i] ← v;
        RETURN;
        END;

-- Initialization

i: TableIndex;
FOR i IN TableIndex DO Table[i] ← NIL ENDLOOP;

END.
```

```
                                DIRECTORY UtilitiesDefs: FROM "UtilitiesDefs",
                                        ArrayStore: FROM "ArrayStore2";
                                DEFINITIONS FROM UtilitiesDefs;
ArrayClient2: PROGRAM =

BEGIN

m: POINTER TO FRAME[ArrayStore] = NEW ArrayStore;

Gedanken: PROCEDURE =
        -- This procedure will fail to compile if ArrayStore does't have the right sort of procedures.  It is not
        meant to run
        BEGIN
        uAllocArray: PROCEDURE [INTEGER] RETURNS [UNSPECIFIED] = m.AllocArray;
        uFreeArray: PROCEDURE [UNSPECIFIED] = m.FreeArray;
        END;

  -- Thus the procedures all have roughly the expected shape and the COERCES below are ok.

-- Integer array primitives
IntArray: TYPE =
  POINTER TO POINTER TO MACHINE DEPENDENT RECORD[ NoNo: IntArray, length: INTEGER, a: ARRAY [ 0..0 ] OF INTEGER];
AllocIntArray: PROCEDURE [INTEGER] RETURNS [IntArray] = COERCE[m.AllocArray];
FreeIntArray: PROCEDURE [IntArray] = COERCE[m.FreeArray];


-- String array primitives
StrArray: TYPE =
  POINTER TO POINTER TO MACHINE DEPENDENT RECORD[ NoNo: StrArray, length: INTEGER, a: ARRAY [ 0..0 ] OF STRING];
AllocStrArray: PROCEDURE [INTEGER] RETURNS [StrArray] = COERCE[m.AllocArray];
FreeStrArray: PROCEDURE [StrArray] = COERCE[m.FreeArray];

-- Mixed array primitives
MixedArray: TYPE =
POINTER TO POINTER TO MACHINE DEPENDENT RECORD[NoNo: MixedArray, length: INTEGER, a: ARRAY [ 0..0 ] OF UNSPECIFIED];
AllocMixedArray: PROCEDURE [INTEGER] RETURNS [MixedArray] = COERCE[m.AllocArray];
FreeMixedArray: PROCEDURE [MixedArray] = COERCE[m.FreeArray];

IA: IntArray;
SA: StrArray;
MA: MixedArray;
i: INTEGER;

BIND m; START m; m ← NIL; --  cuts off any other kind of access to this instance of ArrayStore

-- no type breaches below here

IA ← AllocIntArray[100];
SA ← AllocStrArray[10];
MA ← AllocMixedArray[50];

FOR i IN [0..IA↑.length) DO IA↑.a[i] ← i/3 ENDLOOP;

SA↑.a[0] ← "zero"; SA↑.a[1] ← "one"; SA↑.a[2] ← "two"; SA↑.a[3] ← "surprise"; SA↑.a[4] ← "four";

FOR i IN [0..MA↑.length) DO
        IF i MOD 7 > 4
         THEN MA↑.a[i] ← SA↑.a[i MOD 5]
         ELSE MA↑.a[i] ← IA↑.a[i]
        ENDLOOP;
FreeIntArray[IA];
END.
```

```
                                        DIRECTORY UtilitiesDefs: FROM "UtilitiesDefs";
                                        DEFINITIONS FROM UtilitiesDefs;

ArrayStore3: PROGRAM =
  -- eliminate the backp field by making the compactor more clever.
BEGIN

-- Types
ArrayPtr: TYPE = POINTER TO PR;
PW: TYPE = POINTER TO W;
PR: TYPE = POINTER TO R;
R: TYPE = MACHINE DEPENDENT
  RECORD [ length: INTEGER, a: ARRAY [0..0] OF Thing ];
      -- We expect each field to take 1 machine word
TableIndex: TYPE = [0..TableSize);
Thing: TYPE = RECORD [a: [0..400B), b: [0..400B)];
W: TYPE = RECORD [a: [0..400B), b: [0..400B)];  -- just a machine word, not equal to Thing

-- Storage & Constants
Storage: ARRAY [0..StorageSize) OF W;
StorageSize : INTEGER = 2000;
beginStorage: PR = COERCE[@Storage[0]];
                    -- Breach: The informal requirement for being a PR is that one be a PW which
                    points to the beginning of an R as defined below in the invariant.
nStorage: PR ← beginStorage;        -- next available space to lay an R
endStorage: PW = @Storage[StorageSize];

Table: ARRAY TableIndex OF PR;
TableSize: INTEGER = 500;
beginTable: ArrayPtr = @Table[0];
endTable: ArrayPtr = @Table[TableSize];

ovh: INTEGER = 1;   -- overhead for represented array length

-- The storage area [beginStorage..nStorage) consists of m Rs, each with the form <length,e0,...,e(length-1)>,
where length(>=0) varies from sequence to sequence. The array represented by the record is
<e0,...e(length-1)>. If Table[i] is not NIL, it is the address of one of these records. See Check.

-- Procedures

AllocArray: PUBLIC PROCEDURE [n:INTEGER] RETURNS [new: ArrayPtr] =
        BEGIN i:TableIndex;
        IF n<0 OR n>77777B-ovh THEN ERROR;
        -- find some space
        IF n+ovh > endStorage-COERCE[nStorage,PW] THEN
                -- Breach: demote nStorage to a PW.
                BEGIN
                Compact[];
                IF n+ovh > endStorage-COERCE[nStorage,PW]THEN ERROR;
                END;
        -- Find a table entry
        FOR i IN TableIndex DO
                IF Table[i]=NIL THEN GOTO found
                REPEAT
                found => new ← @Table[i];
                FINISHED => ERROR
                ENDLOOP;
        new↑ ← nStorage;
        new↑.length ← n;

        nStorage←nStorage+n+ovh;
                -- Breach: move available pointer. Note that nStorage is still a good PR because we move it
                just the right distance past the new R.
        RETURN
        END;
```

```
Check: PUBLIC PROCEDURE =
        BEGIN
        i, length: WORD;p: PR;
        FOR i IN TableIndex DO InUse[i] ← FALSE ENDLOOP;

        -- check layout of Storage
        IF gt[beginStorage, nStorage] OR gt[nStorage, endStorage] THEN ERROR;
        FOR p ← beginStorage, p+p.length+ovh UNTIL ge[p, nStorage] DO
                IF p.length<0 THEN ERROR;
                ENDLOOP;
        IF p#nStorage THEN ERROR;

        -- Are any pointers wrong?
        FOR i IN TableIndex DO
                -- This is not too efficient
                IF Table[i] # NIL THEN
                        FOR p ← beginStorage, p+p.length+ovh UNTIL p=nStorage DO
                                IF p=Table[i] THEN EXIT; -- entry is valid
                                REPEAT
                                FINISHED => ERROR
                                ENDLOOP;
                ENDLOOP;
        END;

Compact: PROCEDURE =
        BEGIN
        source,sink: PR;
        length,k: INTEGER;
        i: [0..TableSize];

        -- Reverse all the pointers. Replace the length of each live block with a back pointer (actually a
        negated index of Table) and hide the displaced length in the Table entry.

        FOR i IN TableIndex DO
                IF Table[i] # NIL THEN
                        BEGIN
                        length ← Table[i].length;
                        Table[i].length ← -i;
                        Table[i] ← COERCE[length];
                           -- Breach: replace pointer with length, temporarily
                        END;
                ENDLOOP;

        -- Move all the live records towards beginStorage, fixing up the reversals as we go. A record is live
        iff its length is negative because negative lengths are impossible normally.

        sink ← beginStorage;
        FOR source ← beginStorage, source+length+ovh UNTIL ge[source, nStorage] DO
                   -- Note: length is set inside loop!
                   -- source remains a good PR
                IF source.length<0 THEN     -- record is alive
                        BEGIN
                        length ← COERCE[Table[-source.length]];-- copy to avoid ambush below
                            -- Breach: undo previous
                        Table[-source.length] ← sink; -- new location
                        source.length ← length;
                        -- reversal is undone now
                        FOR k IN [0..length+ovh) DO (sink+k)↑ ← (source+k)↑ ENDLOOP;--Breach
                        sink ← sink + length+ovh;
                           -- sink remains a good PR
                        END
                ELSE length ← source.length
                ENDLOOP;

        nStorage ←sink;
        RETURN
        END;
```

```
FreeArray: PUBLIC PROCEDURE [p:ArrayPtr] =
        BEGIN IF p↑=NIL THEN ERROR; -- array already free
        p↑ ← NIL;
        RETURN;
        END;

-- Initialization

i: TableIndex;
FOR i IN TableIndex DO Table[i] ← NIL ENDLOOP;

END.
```

```
                                DIRECTORY UtilitiesDefs: FROM "UtilitiesDefs";
                                DEFINITIONS FROM UtilitiesDefs;

ArrayStore4: PROGRAM =   -- chain all the free entries together
BEGIN
-- Types
ArrayPtr: TYPE = PF;
Finger: TYPE = RECORD [SELECT COMPUTED FT FROM -- by FingerType
                        alive => [ ptr: POINTER TO R],
                        dead => [ next: POINTER TO Finger],
                        ENDCASE
                        ];
FingerType: PROCEDURE [f:Finger] RETURNS [FT] =
        -- test if it points into Storage
        BEGIN
        RETURN [ IF le[beginStorage,f] AND lt[f,endStorage] THEN alive ELSE dead ]
        END;
FT: TYPE = {alive, dead};
PR: TYPE = POINTER TO R;
PW: TYPE = POINTER TO W;
PF: TYPE = POINTER TO Finger;
R: TYPE = MACHINE DEPENDENT
  RECORD [ length: INTEGER, a: ARRAY [0..0] OF Thing ];
       -- We expect each field to take 1 machine word
TableIndex: TYPE =[0..TableSize);
Thing: TYPE = RECORD [a: [0..400B), b: [0..400B)];
W: TYPE = RECORD [a: [0..400B), b: [0..400B)];  -- just a machine word, not equal to Thing

-- Rs are kept in ...
Storage: ARRAY [0..StorageSize) OF W;
StorageSize : INTEGER = 2000;
beginStorage: PR = COERCE[@Storage[0]];
                        -- Breach: The informal requirement for being a PR is that one be a PW which
                        points to the beginning of an R as defined below in the invariant.
nStorage: PR ← beginStorage; -- next availble space to lay an R
endStorage: PW = @Storage[StorageSize];

Table: ARRAY TableIndex OF Finger;
TableSize: INTEGER = 500;
beginTable: PF = @Table[0];
endTable: PF = @Table[TableSize];

DeadFingerList: PF;
ovh: INTEGER = 1;   -- overhead for represented array, length

-- The storage area [beginStorage..nStorage) consists of m Rs, each with the form <length,e0,...,e(length-1)>,
where length(>=0) varies from sequence to sequence. The array represented by the record is
<e0,...e(length-1)>.
-- Table[i] is either the address of one of these records, in which case it is an alive Finger, or it is and
address in Table or NIL, in which case it is a dead Finger. DeadFingerList is a chain of all the dead Fingers. A
record in Storage is alive iff it is pointed to from Table.  See Check.

-- Procedures

AllocArray: PUBLIC PROCEDURE [n:INTEGER] RETURNS [new: ArrayPtr] =
        BEGIN
        IF n<0 OR n>77777B-ovh THEN ERROR;
        -- find some space
        IF n+ovh > endStorage-COERCE[nStorage,PW] THEN
                -- Breach: demote nStorage to a PW.
                BEGIN
                Compact[];
                IF n+ovh > endStorage-COERCE[nStorage,PW] THEN ERROR;
                END;
        -- Find a table entry
        IF DeadFingerList = NIL THEN ERROR;
```

```
        new ← DeadFingerList;
        DeadFingerList ← WITH DeadFingerList SELECT
                        FingerType[DeadFingerList↑] FROM
                        dead => next,
                        ENDCASE => NIL; -- not possible

        new↑ ← Finger[alive[nStorage]];

        -- initialize the array storage
        nStorage.length ← n;

        nStorage←nStorage+n+ovh;
                -- move available pointer. Note that nStorage is still a good PR because we move it just the
                right distance past the new R.
        RETURN
        END;

Check: PUBLIC PROCEDURE =
        BEGIN
        i, length: WORD;p: PR; f:PF;

        -- check layout of Storage
        IF gt[beginStorage, nStorage] OR gt[nStorage, endStorage] THEN ERROR;
        FOR p ← beginStorage, p+p.length+ovh UNTIL ge[p, nStorage] DO
                IF p.length<0 THEN ERROR;
                ENDLOOP;
        IF p#nStorage THEN ERROR;

        -- Is the DeadFinger list good? If circular we won't return
        f ← DeadFingerList;
        UNTIL f=NIL DO
                WITH f SELECT FingerType[f↑] FROM
                        alive => ERROR;
                        dead => f ← next;
                        ENDCASE;
                ENDLOOP;

        -- Check all the pointers in Table
        FOR i IN TableIndex DO
                -- This is not too efficient
                WITH tableEntry:Table[i] SELECT FingerType[Table[i]] FROM
                        alive =>
                                FOR p ← beginStorage, p+p.length+ovh UNTIL p=nStorage DO
                                        IF p=ptr THEN EXIT; -- entry is valid
                                        REPEAT
                                        FINISHED => ERROR
                                        ENDLOOP;
                        dead =>
                                BEGIN
                                f ← DeadFingerList;
                                UNTIL f=NIL DO
                                        WITH f SELECT FingerType[f↑] FROM
                                        dead => BEGIN
                                                IF f=tableEntry.next THEN EXIT;
                                                f ← next;
                                                END;
                                        ENDCASE; -- alive is impossible
                                        REPEAT
                                        FINISHED => ERROR; -- lost finger
                                        ENDLOOP;
                                END;
                        ENDCASE;

                ENDLOOP;
        END;

Compact: PROCEDURE =
```

```
        BEGIN
        source,sink: PR;
        length,k: INTEGER;
        i: TableIndex;

        -- Reverse all the pointers. Replace the length of each live block with a back pointer (actually a
        negated index of Table) and hide the displaced length in the Table entry.

        FOR i IN [0..TableSize) DO
                WITH Table[i] SELECT FingerType[Table[i]] FROM
                        alive =>
                                BEGIN
                                length ← ptr.length;
                                ptr.length ← -i;
                                Table[i] ← COERCE[length];
                                   -- Breach: replace pointer with length, temporarily
                                END;
                        ENDCASE;
                ENDLOOP;

        -- Move all the live records towards beginStorage, fixing up the reversals as we go. A record is live
        iff its length is negative because negative lengths are impossible normally.

        sink ← beginStorage;
        FOR source ← beginStorage, source+length+ovh UNTIL ge[source, nStorage] DO
                -- length is always set inside loop
                IF source.length<0 THEN   -- record is alive
                        BEGIN
                        length ← COERCE[Table[-source.length]];-- copy to avoid ambush below
                            -- Breach: undo previous breach
                        Table[-source.length] ← Finger[alive[sink]];
                                -- new location
                        FOR k IN [0..length+ovh) DO (sink+k)↑ ← (source+k)↑ ENDLOOP;--Breach
                        sink ← sink+length+ovh;
                           -- sink remains a good PR
                        END
                ELSE length ← source.length; -- just normal length of dead record
                ENDLOOP;
        nStorage ←sink;
        RETURN
        END;

FreeArray: PUBLIC PROCEDURE [p:ArrayPtr] =
        BEGIN
        WITH p SELECT FingerType[p↑] FROM
                alive => BEGIN
                        p↑ ← Finger[dead[DeadFingerList]];
                        DeadFingerList ← p;
                        END;
                dead => ERROR; -- this error is possible, dangling reference
                ENDCASE;
        RETURN;
        END;

-- Initialization

i: TableIndex;
FOR i IN [0..TableSize-1) DO
        Table[i] ← Finger[dead[@Table[i+1]]]
        ENDLOOP;
Table[TableSize-1] ← Finger[dead[NIL]];

END.
```

```
                                DIRECTORY UtilitiesDefs: FROM "UtilitiesDefs";
                                DEFINITIONS FROM UtilitiesDefs;

ArrayStore5: PROGRAM = -- No, no. Not the Rack!
 -- Put Table and Storage at opposite ends of same array
BEGIN
-- Types
ArrayPtr: TYPE = PF;
Finger: TYPE = RECORD [SELECT COMPUTED FT FROM -- by FingerType
                        alive => [ ptr: POINTER TO R],
                        dead => [ next: POINTER TO Finger],
                        ENDCASE
                        ];
FingerType: PROCEDURE [f:Finger] RETURNS [FT] =
        -- test if it points into [beginStorage..nStorage)
        BEGIN
        RETURN [ IF le[beginStorage,f] AND lt[f,nStorage] THEN alive ELSE dead ]
        END;
FT: TYPE = {alive, dead};
PR: TYPE = POINTER TO R;
PW: TYPE = POINTER TO W;
PF: TYPE = POINTER TO Finger;
R: TYPE = MACHINE DEPENDENT
  RECORD [ length: INTEGER, a: ARRAY [0..0] OF Thing ];
        -- We expect each field to take 1 machine word
Thing: TYPE = RECORD [a: [0..400B), b: [0..400B)];
W: TYPE = RECORD [a: [0..400B), b: [0..400B)]; -- just a machine word, not equal to Thing

-- Rs and Fingers are kept in ...
Rack: ARRAY [0..RackSize) OF W;
RackSize : INTEGER = 2500;
beginStorage: PR = COERCE[@Rack[0]];
                        -- Breach: The informal requirement for being a PR is that one be a PW which
                        points to the beginning of an R as defined below in the invariant.
nStorage: PR ← beginStorage; -- next availble space to lay an R
beginTable: PF = COERCE[@Rack[RackSize-1]];
                        -- Breach: end of rack contains Fingers
nTable: PF ← beginTable; -- next Finger to allocate

DeadFingerList: PF;

ovh: INTEGER = 1;   -- overhead for represented array, length

-- The storage area [beginStorage..nStorage) consists of m Rs, each with the form <length,e0,...,e(length-1)>,
where length(>=0) varies from sequence to sequence. The array represented by the record is
<e0,...e(length-1)>.
-- The storage area (nTable..beginTable] contains Fingers which are either the address of one of the above
records, in which case it is an alive Finger, or it is an address in (nTable..beginTable] or NIL, in which case it
is a dead Finger. DeadFingerList is a chain of all the dead Fingers. A record is alive iff it is pointed to from
this area. See Check.

-- Procedures

AllocArray: PUBLIC PROCEDURE [n:INTEGER] RETURNS [new: ArrayPtr] =
        BEGIN new:ArrayPtr;
        IF n<0 OR n>77777B-ovh THEN ERROR;
        -- find some space. The +1 on the left allows room for new finger as well as array. The expression
        on the right is the number of unused words in the Rack.
        IF n+ovh+1> COERCE[nTable,PW]-COERCE[nStorage,PW]+1 THEN
                -- Breach: demote pointers to same type
                BEGIN
                Compact[];
                IF n+ovh+1> COERCE[nTable,PW]-COERCE[nStorage,PW]+1 THEN ERROR;
```

```
        END;
    -- Find a table entry, there must be one
    IF DeadFingerList = NIL THEN
        BEGIN
        new ← nTable;
        nTable ← nTable - 1;
        END
    ELSE BEGIN
        new←DeadFingerList;
        DeadFingerList ← WITH DeadFingerList SELECT
                        FingerType[DeadFingerList↑] FROM
                        dead => next,
                        ENDCASE => NIL; -- not possible
        END;
    new↑ ← Finger[alive[nStorage]];
    -- initialize the array storage
    nStorage.length ← n;

    nStorage←nStorage+n+ovh;
        -- move available pointer. Note that nStorage is still a good PR because we move it just the
        right distance past the new R.
    RETURN
    END;

Check: PUBLIC PROCEDURE =
    BEGIN
    i, length: WORD;p: PR; f,t :PF;
    -- check layout of Rack
    IF gt[beginStorage, nStorage] OR gt[nStorage, nTable+1] OR gt[nTable, beginTable] THEN ERROR;
    FOR p ← beginStorage, p+p.length+ovh UNTIL ge[p, nStorage] DO
        IF p.length<0 THEN ERROR;
        ENDLOOP;
    IF p#nStorage THEN ERROR;
    -- Is the DeadFinger list good? If circular we won't return
    f ← DeadFingerList;
    UNTIL f=NIL DO
        WITH f SELECT FingerType[f↑] FROM
            alive => ERROR;
            dead => f ← next;
            ENDCASE;
        ENDLOOP;
    -- Check all the pointers in Table
    FOR t ← beginTable, t-1 UNTIL t=nTable DO
        -- This is not too efficient
        WITH tableEntry:t SELECT FingerType[t↑] FROM
            alive =>
                FOR p ← beginStorage, p+p.length+ovh UNTIL p=nStorage DO
                    IF p=ptr THEN EXIT; -- entry is valid
                    REPEAT
                    FINISHED => ERROR
                    ENDLOOP;
            dead =>
                BEGIN
                f ← DeadFingerList;
                UNTIL f=NIL DO
                    WITH f SELECT FingerType[f↑] FROM
                    dead => BEGIN
                        IF f=tableEntry.next THEN EXIT;
                        f ← next;
                        END;
                    ENDCASE; -- alive is impossible
                    REPEAT
                    FINISHED => ERROR; -- lost finger
                    ENDLOOP;
                END;
```

```
                ENDCASE;
        ENDLOOP;
    END;

Compact: PROCEDURE =
    BEGIN
    DeformedR: TYPE =
      RECORD [ backp: POINTER TO INTEGER, a: ARRAY [0..0] OF Thing];
    source,sink: PR ← beginStorage;
    length,k: INTEGER;
    pFing: POINTER TO Finger;
    pDR: POINTER TO DeformedR;
    -- Reverse all the pointers. Replace the length of each live block with a back pointer and hide the
    displaced length in the Table entry.
    FOR pFing ← beginTable, pFing-1 WHILE gt[pFing,nTable] DO
        WITH pFing SELECT FingerType[pFing↑] FROM
            alive =>
                BEGIN
                length ← ptr.length;
                pDR ← COERCE[ptr];
                pDR.backp ← COERCE[pFing];
                  -- Breach: deform ptr
                pDR.backp↑ ← length;
                END;
            ENDCASE;
        ENDLOOP;
    -- Move all the live records towards beginStorage, fixing up the reversals as we go. A record is live iff
    its length is greater than nTable. Because of the way Rack is laid out it would be impossible for a
    legitimate length to be greater than nTable; therefore such a length must be one of the pointers we
    stuck in before!
    sink ← beginStorage;
    FOR source ← beginStorage, source+length+ovh UNTIL ge[source, nStorage] DO
        IF gt[source.length,nTable] THEN     -- record is alive
            BEGIN
            pDR ← COERCE[source];
                -- Breach: source is a pointer to a DeformedR
            length ← pDR.backp↑;-- copy to avoid ambush below
            pDR.backp↑ ← COERCE[sink];
                 -- Breach: new location written over length
            source.length ← length;
            -- reversal is undone now
            FOR k IN [0..length+ovh) DO (sink+k)↑ ← (source+k)↑ ENDLOOP;--Breach
            sink ← sink + length+ovh;
              -- sink remains a good PR
            END
        ELSE length ← source.length; -- dead record
        ENDLOOP;
    nStorage ←sink;
    RETURN
    END;

FreeArray: PUBLIC PROCEDURE [p:ArrayPtr] = -- same as ArrayStore5
    BEGIN
    WITH p SELECT FingerType[p↑] FROM
        alive => BEGIN
            p↑ ← Finger[dead[DeadFingerList]];
            DeadFingerList ← p;
            END;
        dead => ERROR; -- this error is possible, dangling reference
        ENDCASE;
    RETURN;
    END;
END.
```