BY GARY F. SIMONS

DATA ABSTRACTION

A BASIC IMPLEMENTATION FOR PROBLEM SOLVING

Editor's note: For more information and background reading on data abstraction, see the article by Niklaus Wirth *in the August issue,* "History *and* Goals *of* Modula-2."

PROGRAMMERS, AND INDEED problem solvers in general, have two basic strategies for attacking new problems: decomposition and abstraction. These strategies offer two different ways of solving a complex problem by simplifying it in some way. Decomposition is the strategy embodied by the time-honored Machiavellian dictum to "divide and conquer"—solving large problems by dividing them into simpler, smaller ones that can be solved independently. Abstraction is the strategy of ignoring certain details about the original problem so as to transform it into a simpler and more general one.

For example, consider the problem of computing the sum of the squares of two numbers, 3 and 4 (that is, compute $3^2 + 4^2$). We first simplify the problem by decomposing it into a sequence of three simpler problems: (1) compute the square of 3, (2) compute the square of 4, and (3) add the two results together. We assume that the final step of addition is sufficiently fundamental that we need not consider it further. However, the first two subproblems can be restated in more simple terms. "Compute the square of 3" means the same as "multiply 3 by 3," and "compute the square of 4" means the same as "multiply 4 by 4."

We may now apply the principle of abstraction to simplify the problem further. We see that there is something essentially the same about computing the square of 3 and computing the square of 4. By abstracting away the particular details of the 3 versus the 4, both subproblems can be solved by a single more general solution, namely, that of computing the square of *n*, where *n* can represent any number.

Gary F. Simons (7500 W. Camp Wisdom Rd., Dallas, TX 75236) is an International Consultant for Linguistics and Academic Computing with the Summer Institute of Linguistics.

INFORMATION HIDING AND ABSTRACT DATA TYPES

The essential design methodology for data abstraction is known as *information hiding*. The approach was first proposed by D. L. Parnas in 1972 (see reference 4). He proposed that the behavior of software modules be specified completely in terms of their external effects. Such a module hides a secret, namely, the representation of the data object that the module manages. To the outside user, the module provides a set of access functions that are used to create, alter, or observe instances of the abstract data object. There is no way for anyone or anything but the implementation of the module itself to access the objects, other than through those access functions.

The type of module that Parnas first described has come to be known as an abstract data type. It is abstract because the details of the concrete representation of the data type are unknown to the user. It has also been called an *encapsulated* data type, since the details of implementation are locked away from the user inside a capsule. The functions that access an abstract data type are now commonly referred to as its *operations*.

An abstract data type, therefore, presents itself to the user not as a data structure, but as a collection of procedural abstractions. These are the operations that allow one to create, observe, or alter objects of the abstract type. The task of implementing an abstract data type then consists of determining a concrete representation for objects of the type and writing the procedural abstractions that operate on objects thus represented.

THE INFORMAL SPECIFICATION OF ABSTRACT DATA TYPES

Two methods for specifying the behavior of abstract data types have emerged. The first is an informal approach that uses prose statements to describe the effect of each of the operations of the data type. The second is a formal approach that uses algebraic statements that are precise and unambiguous. Both approaches are described in turn.

Barbara Liskov gives a complete example of the informal method in her paper "Modular Program Construction Using Abstraction." (See reference 3.) In Liskov's informal approach, the specification of a data abstraction has three parts: a header that names the data abstraction and its operations; a brief description of the data abstraction as a whole; and a specification for each of the operations. Each of the operations is a procedural abstraction. The specification of a procedural abstraction may have four parts: a header; a modifies line; a requires line; and an effect line. The header defines how the procedure interfaces with the outside world, that is, its name, the order and types of its inputs and outputs, and the error conditions it signals. The modifies line defines which of the inputs may be modified by the procedure. The requires line defines any assumptions that are made about the calling environment. The effect line describes what the operation is intended to do. Figure 1 gives some sample specifications for procedural abstractions based on Liskov's approach.

We may now illustrate the specification of a data abstraction. Figure 2 gives a sample specification of a data type called intset (set of positive integers). Note the three parts of the specification: the header that names the data type and lists its operations; a brief description of the data type as a whole; and the specifications for each of the operations.

THE FORMAL SPECIFICATION OF DATA ABSTRACTIONS

The formal approach to specifying data abstractions defines an abstract

(continued on page 414)

```
divide (x,y:real) returns real signals divide-by-zero
  effect if y = O then signals divide-by-zero
    else returns x/y

sort (x:array[int])
  modifies x
  effect sorts the elements of x in ascending order

search (x:array[int], y:int) returns int signals no-match
  requires x is sorted in ascending order
  effect returns i such that x [i] = y;
    signals no-match if no such i is found
```

Figure 1: *Sample specifications of procedural abstractions (after* Liskov*).*

intset **is** create, insert, remove, isempty, ismember

Intsets are sets of positive integers; the maximum size of an intset is $2^{16} - 1$ members. Intsets are either created empty (using *create*) or made from other intsets with a new member added (using *insert*) or removed (using *remove*). *Isempty* tests whether an intset has any members; *ismember* tests whether a given integer is a member of an intset.

```
create ( ) returns intset
  effect returns an empty intset

insert (s:intset, x:int) returns intset signals no-room
  effect if the size of s union {x} is less than or
           equal to the maximum size,
        then returns s union {x},
        else signals no-room

remove (s:intset, x:int) returns intset
  effect returns s minus x

isempty (s:intset) returns Boolean
  effect returns true if s has no members

ismember (s:intset, x:int) returns Boolean
  effect returns true if x is a member of s
```

Figure 2: *Sample specifications of data abstraction (after* Liskov*).*

(continued from page 131)

type in terms of an algebra. In the general theory of algebras, an algebra is a pair <A;F>, where *A* is a nonempty set and *F* is a family of operations on *A*. For instance, the familiar algebra of grade school mathematics is defined by the set of real numbers and the operations of addition, subtraction, and so on. Since an abstract data type consists of a set of objects that carry the type and the operations on those objects, it is easy to see how abstract data types lend themselves to definition in terms of an algebra.

The meaning (or the effect) of the operations is defined as a set of formal axioms that state the relationships among the operations. The reduction of the operations' meanings to a set of axioms makes it possible to reason formally about the correctness of a design before it is implemented. This is one of the productive ways of using this approach in program design.

The algebraic approach to specifying abstract data types is rigorously defined by John Guttag and J. J. Horning (see reference 1) and consists of two parts: a syntactic specification and a semantic specification. The syntactic specification defines how the type interfaces with the outside world; it defines the name of the type, the names of all its operations, and the types of the domains (inputs) and ranges (outputs) of the operations. Figure 3 illustrates the syntactic specification for the abstract type intset.

The operations on any data type fall into two classes: generator operations and inquiry operations. The generators are those operations that produce an object of the type of interest (for example, intset). The inquiry operations focus on objects of interest but produce a result that is of a different type (for example, Boolean). The blank line in the example of figure 3 separates the two classes of operators. Within the set of generators there is a subset called basic generators that are sufficient to generate any object of the type of interest. The basic generators, create and insert, are marked in figure 3 with a preceding asterisk.

```
Abstract data type: Intset

Operations:

* create:                    -->  intset
* insert:    intset x int    -->  intset
  remove:    intset x int    -->  intset

  isempty:   intset          -->  Boolean
  ismember:  intset x int    -->  Boolean
```

Figure 3: *Syntactic specification for abstract type* intset.

The semantic specification of the operations consists of a set of axioms that define the meaning of the operations by stating their relationships to one another. The axioms are presented as equations in which the left-hand side specifies an expression to be defined and the right-hand side gives its meaning. For the basic generators, no definitions are written; they are assumed as given. Thus we first write axioms that define the meaning of the nonbasic generators (for example, remove); the right-hand sides of these equations must eventually be reduced to expressions involving only basic generators. Then

(continued)

we provide axioms that define the meaning of applying each inquiry operation to each of the basic generators. Note that since the meaning of any nonbasic generator can be expressed in terms of basic generators alone, there is no need to write axioms defining the application of inquiry operations to nonbasic generators. Figure 4 gives a sample semantic specification for the type intset. The equal sign used in the axioms is to be read as "means the same thing as."

Note that there are three sets of axioms, one for each nonbasic generator and inquiry operation that must be defined. Each is defined in terms of two axioms, one for each of the two basic generators of the type of interest.

The definition of isempty is easy to understand. If an intset, s, were generated by create, then the set has no members yet, and the expression empty(s) means the same thing as T, or true. If the intset was generated by insert, then the set must have members, and empty means the same thing as F, or false.

Remove and ismember use a basic generator of type Boolean named equal. It tests two integers for equality. In conjunction with this Boolean generator, they use the if-then-else operation that is defined by the following two axioms:

if T then *a* else *b* = *a*
if F then *a* else *b* = *b*

That is, "if T then *a* else *b*" means the same thing as (or, can be reduced to) *a*, and "if F then *a* else *b*" means simply *b*. In other words, if the Boolean condition reduces to true then the whole expression reduces to the *then* clause, otherwise to the *else* clause.

Both remove and ismember are defined recursively, that is, in terms of themselves. For instance, the second axiom for ismember says that if the integer *i* is not equal to the first item in the intset generated by "insert (i',s)", then the expression "ismember(i, insert(i',s))" means the same thing as "ismember(i,s)". We then apply the

(continued)

same axioms to this simplified expression to reduce it ultimately to T or F. The recursive application of the axioms is easily illustrated. For instance, imagine an intset of three members: {1,2,3}. In the language of our axioms, this set would be represented by the expression:

insert(1,insert(2,insert(3,create)))

We now want to test if 4 is a member of the set. This test is equivalent to the expression:

ismember(4,insert(1,insert(2,insert(3,create))))

To discover what this expression means, we simplify it by applying the axioms repeatedly until it can be reduced no further. Figure 5a illustrates this process; we see that the expression ultimately reduces to F, or false. Similarly, we can test if 2 is a member of the set. This is done in figure 5b where we see that the result is T, or true.

These illustrations demonstrate the potential of the method for giving us a rigorous way to reason about the meaning and ultimately the correctness of program designs. The sequences of derivations in figure 5 are actually proofs that "ismember(4, {1,2,3})" means false and that "ismember(2,{1,2,3})" means true. In

(continued)

```
Axioms:
For all i, i' of type Int, and all s of type Intset:

remove(i,create) = create
remove(i,insert(i',s)) = if equal (i,i') then s else insert(i',remove(i,s))

isempty(create) = T
isempty(insert(i,s)) = F

ismember(i,create) = F
ismember(i,insert(i',s)) = if equal(i,i') then T else ismember(i,s)
```

Figure 4: *Semantic specification for abstract type* intset.

```
(a) Is 4 a member of the set {1,2,3}?

ismember(4,insert(1,insert(2,insert(3,create)))) =
if equal(4,1) then T else ismember(4,insert(2,insert(3,create))) =
if F then T else ismember(4,insert(2,insert(3,create))) =
ismember(4,insert(2,insert(3,create))) =
if equal(4,2) then T else ismember(4,insert(3,create)) =
if F then T else ismember(4,insert(3,create)) =
ismember(4,insert(3,create)) =
if equal(4,3) then T else ismember(4,create) =
if F then T else ismember(4,create) =
ismember(4,create) =
F

(b) Is 2 a member of the set {1,2,3}?

ismember(2,insert(1,insert(2,insert(3,create)))) =
if equal(2,1) then T else ismember(2,insert(2,insert(3,create))) =
if F then T else ismember(2,insert(2,insert(3,create))) =
ismember(2,insert(2,insert(3,create))) =
if equal(2,2) then T else ismember(2,insert(3,create)) =
if T then T else ismember(2,insert(3,create)) =
T
```

Figure 5: *Reducing expressions by axiomatic substitution.*

a formal proof, we would also need to list with each reduction a reference to the axiom used.

These examples should make it clear that the axioms are expressed rigorously enough that a computer could help us in the tedious work of reducing expressions as we reason and test our designs. In fact, the axioms could be implemented directly on a substitution (or reduction) machine as a way of testing the design. Cristoph Hoffman and Michael O'Donnell describe just such a system in their article, "Programming with Equations" (see reference 2). Certainly the specification of data abstractions, whether by formal or informal means, is a powerful new tool for software designers.

Implementing Abstract Data Types in BASIC

We now turn our attention to the practical application of data abstraction by implementing two versions of the intset abstraction in BASIC. BASIC is by no means an ideal language for implementing data abstractions. However, in as much as it is the lingua franca of personal computing, there is perhaps no better way of demonstrating the principles of data abstraction to a general audience. Not only should this exercise illustrate the benefits of the data-abstraction technique, it should also demonstrate that with discipline, a programmer can produce good code in BASIC that is maintainable and portable.

BASIC lacks two things that data abstraction needs: parameterized procedure calls (for invoking the operations) and limited scoping of variables (to support information hiding). Neither shortcoming is insurmountable; both are solved by requiring a more careful use of variables. Unfortunately, this puts a greater responsibility on the programmer than would be required by more modern languages and consequently increases the opportunity for programming errors.

The variables used in a BASIC implementation of an abstract data type fall into three categories: variables used to pass parameters to, and return values from, the operations; variables used in the concrete representation of the abstract type; and variables used locally in the implementation of the operations. Unfortunately, all variables in BASIC have a global scope and so nothing can be used without reference to how it is used elsewhere in the program. The first category of variables defines the user's interface with the operations of the data type. These must be known and understood in order to use the data type. The second and third categories need not be understood by the user, but their names must be known.

(continued)

Circle 292 on inquiry card. →

he programmer must know what ariables are used in the implemen- ation of the data type, in order to void abusing the type by inadvert- ntly using one of these variables for different purpose elsewhere.

In a pure information-hiding en- ironment this would not be the case. Vhen a data abstraction truly hides he concrete representation of a type, hen the way one type is represented annot interfere with how another is. ut in BASIC this does not come auto- atically since all variables have a lobal scope and the representation f one type can therefore interfere vith another if the variables are not nique. The BASIC programmer must herefore be content with a weaker orm of data abstraction that hides he meaning and use of the data tructures that implement an abstract ype but cannot hide their names.

After an abstract data type has been specified, as discussed in the preceding sections, there are three steps in its implementation: (1) define the user interface, (2) define the concrete representation, and (3) implement the operations. Each of these steps will be considered in turn.

DEFINING THE USER INTERFACE

The user interface refers to what a person must know in order to invoke the operations of the data type. For each operation, the following items must be defined: the line number where the subroutine begins, the variables in which input parameters are passed to the operation, the variables in which values are returned, and the variables in which exception codes are signaled. All of this information can be conveyed in a one-line REM (remark) statement that serves as a header for the subroutine. (The version of BASIC used in the listing below allows ' as shorthand for REM.)

Figure 6 illustrates a possible definition of the user interface for the abstract type intset. Note, for instance, the header for the operation "insert." The address of the subroutine is identified as line 1200. The input parameters (a set identifier and an integer to insert into the set) are passed in the variables *S* and *X* respectively. The operation returns *s* as the set identifier of the resulting set. The operation may signal one of three exception codes; this is done with three variables that return a Boolean value. If one of these variables has a value of true when the subroutine returns, then that condition has occurred. Thus the insert operation may signal "not a valid set" in NS, "not a valid integer" in NI, or "no more room" in NR.

(continued)

Note also that figure 6 defines four new operations on type intset: display, kill, intersect, and union. Display prints the contents of a set on the screen. Kill deletes an existing set. Intersect produces a new set that is the intersection of two existing sets. Union produces a new set that is the combination of two existing sets. These operations provide a fuller environment for testing our implementations of the intset abstraction.

```
1000 'abstract data type: INTSET
1100 'create( ) returns(s) signals(os)
1200 'insert(s,x) returns(s) signals(ns,ni,nr)
1300 'remove(s,x) returns(s) signals(ns,ni)
1400 'isempty(s) returns(b) signals (ns)
1500 'ismember(s,x) returns(b) signals(ns,ni)
1600 'display(s) signals(ns)
1700 'kill(s) signals(ns)
1800 'intersect(s1,s2) returns(s) signals(ns,os)
1900 'union(s1,s2) returns(s) signals (ns,os,nr)

Key to exception codes:

ni = not a valid integer
nr = no more room
ns = not a valid set
os = out of sets
```

Figure 6: *User interface for abstract data type* intset.

Using the Abstract Data Type

With the design of the user interface in hand, we know enough to write a program that uses the type. Figure 7 gives the overall design for a program to test an implementation of intset. (It overlooks the details of what to do when an exception is signaled.) The test program repeatedly takes a one-letter command and two numerical arguments, executes the named operation, and displays the resulting set.

See listing 1 for the test program. As an example of how to use an operation, consider the use of insert in lines 410 and 420. The operation is invoked by GOSUB 1200, but before calling the subroutine we must pass the parameters. The interface requires

(continued)

that the set identifier be passed in S and the integer to insert in X, thus the code: "S = P1: X = P2". In line 420, the statement GOSUB 1600 calls display. This too requires a set identifier in S but since insert returns with the current set in S, there is no need for an assignment statement to pass the parameter. As long as S was not an invalid set identifier, that is, as long as NS = False, we display the resulting contents of set S. It is assumed that all exception conditions will generate an error message when they are discovered by intset. If NI or NR occurs, a message will be given, but we still want to display the current status of S. The use of the other operations follows the same principles.

```
PROCEDURE test intset abstraction:
    Initialize abstract data type Intset
    REPEAT until commanded to quit,
        Get a command and two parameters (p1 and p2)
        CASE of command,
            c: display(create( ))
            d: display(p1)
            e: isempty(p1)
            i: display(insert(p1,p2))
            k: kill(p1)
            m: ismember(p1,p2)
            q: quit program
            r: display(remove(p1,p2))
            u: display(union(p1,p2))
            x: display(intersect(p1,p2))
```

Figure 7: *Design for program to test an implementation of* intset.

It is important to note that we were able to write a program using intset before we decided how the type is to be represented or implemented. This is the power of information hiding and data abstraction at work. As long as we stick to the interface definition in figure 6 when we implement the data type, the test program will work.

There is, however, one slight complication brought on by the global scoping of all variables in BASIC. Our test program is guaranteed to work only if we avoid variable conflicts. The test program happens to use three variables of its own that are not part of the intset interface. These are C$, P1, and P2. The test program is

(continued)

guaranteed, therefore, only if the implementation of intset stays away from these three variable names. Conversely, if we begin with an already implemented data type and want to write a program that uses it, that program may not use any variable as a free variable that is used in the implementation of the data type.

DEFINING THE CONCRETE REPRESENTATION

The second step in implementing an abstract data type is to define the concrete representation. A straightforward representation for integer sets is to store them in a matrix where each row represents a set and the columns hold individual set elements. A value of −1 means that the matrix cell is empty; a positive integer is a set element. A value of −1 in column 0 means that the whole row is unused.

Note that in figure 6, the first operation is not coded until line 1100. The lines between 1000 and 1100 are reserved for comments that describe the concrete representation, followed by an initialization subroutine that sets up the storage space for the data type as required by the method of representation. This subroutine is the first thing called by the test program of listing 1.

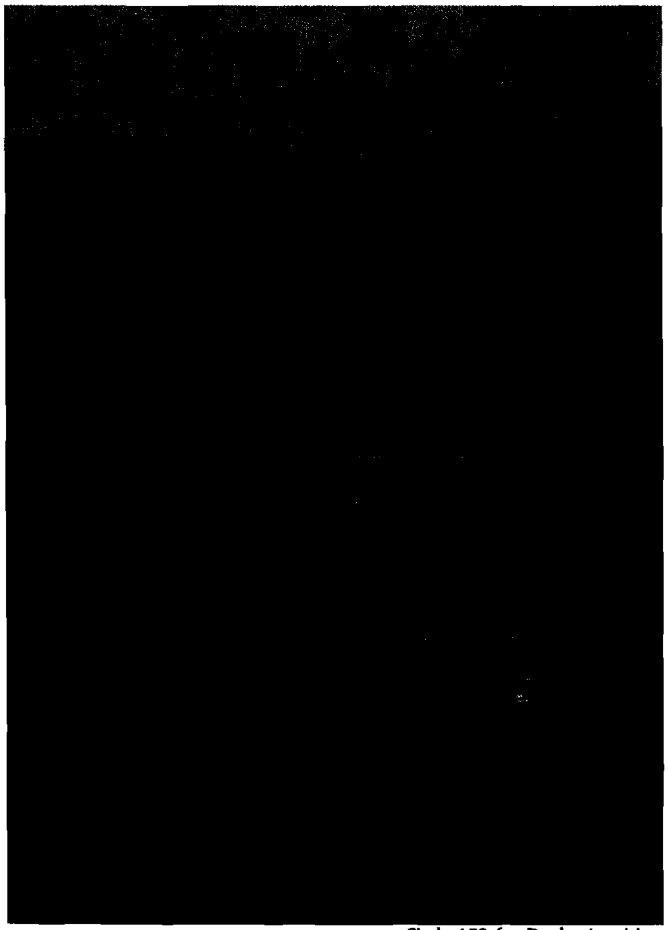
Listing 2 gives a full implementation of intset with an underlying matrix representation of 11 sets (in rows 0 to 10) with 10 elements each (in columns 1 to 10). Note lines 1000 to 1040, which are comments describing the method of representation, and lines 1050 to 1080, which define an initialization routine that sets up the storage space for intsets.

IMPLEMENTING THE OPERATIONS

Now we are ready for the third step, implementing the operations. The headers defined for the user interface (see figure 6) serve as the first lines for the subroutines that implement each of the operations. Given the representation of the data type and the variables specified in the header for parameter, result, and exception code passing, the implementation of the operations falls into place. See listing 2 for the complete implementation of the operations. Note that the implementation makes use of two private subroutines (at lines 2000 and 2100) for checking the validity of parameters s and x. These do not appear in the list of operations of the data type (figure 6) because they are meant to be used only from within the module, not by outside users.

A new complication presents itself when implementing the operations. That is the problem of local variables. One must ensure that the extra vari-

(continued)



Listing 1: *Program for testing the* intset *abstraction.*

```
100 GOSUB 1000 'initialize INTSET storage
110 T=-1: F=0 'initialize TRUE and FALSE
200 INPUT C$,P1,P2: IF C$<"a" THEN C$=CHR$(ASC(C$)+32)
250 IF C$="c" THEN 260 ELSE 300
260    GOSUB 1100 'create( )
270    IF NOT OS THEN GOSUB 1600 'display(s)
280    GOTO 200
300 IF C$="d" THEN 310 ELSE 350
310    S=P1: GOSUB 1600 'display(p1)
320    GOTO 200
350 IF C$="e" THEN 360 ELSE 400
360    S=P1: GOSUB 1400 'is-empty(p1)
370    IF NOT NS THEN IF B THEN PRINT "True" ELSE PRINT "False"
380    GOTO 200
400 IF C$="i" THEN 410 ELSE 450
410    S=P1: X=P2: GOSUB 1200 'insert(p1,p2)
420    IF NOT NS THEN GOSUB 1600 'display(s)
430    GOTO 200
450 IF C$="k" THEN 460 ELSE 500
460    S=P1: GOSUB 1700 'kill(p1)
470    IF NOT NS THEN PRINT "Set";S;"deleted"
480    GOTO 200
500 IF C$="m" THEN 510 ELSE 550
510    S=P1: X=P2: GOSUB 1500 'is-member(p1,p2)
520    IF NOT NS AND NOT NI THEN IF B THEN PRINT"True" ELSE PRINT"False"
530    GOTO 200
550 IF C$="q" THEN 560 ELSE 600
560    STOP 'quit program
600 IF C$="r" THEN 610 ELSE 650
610    S=P1: X=P2: GOSUB 1300 'remove(p1,p2)
620    IF NOT NS THEN GOSUB 1600 'display(s)
630    GOTO 200
650 IF C$="u" THEN 660 ELSE 700
660    S1=P1: S2=P2: GOSUB 1900 'union(p1,p2)
670    IF NOT NS AND NOT OS THEN GOSUB 1600 'display(s)
680    GOTO 200
700 IF C$="x" THEN 710 ELSE 750
710    S1=P1: S2=P2: GOSUB 1800 'intersect(p1,p2)
720    IF NOT NS AND NOT OS THEN GOSUB 1600 'display(s)
730    GOTO 200
750 PRINT "Unrecognized command": GOTO 200
```

Listing 2: *Matrix implementation of* intset *abstraction.*

```
1000 'Abstract data type: INTSET
1005 '
1010 'Representation:
1015    'Each row of matrix S(10,10) stores a set. There is thus
1020    'a maximum of 11 sets (numbered 0 to 10). If the 0 element
1025    'of a row is -1, then that set has not been created.
1030    'That leaves columns 1 to 10 for set elements. A cell with
1035    'value -1 is empty; otherwise the cell contains an element
1040    'of the set
1050 'Initialize the INTSET storage:
1060    DIM S(10,10)
1070    FOR I=0 TO 10: FOR J=0 TO 10: S(I,J)=-1: NEXT J,I
1080    RETURN
1095 '
1100 'create( ) returns(s) signals(os) local(i)
1110 'OS=F
```

(continued)

```
1120 FOR I=0 To 10 'find first unused set
1130    IF S(I,0)=-1 THEN S(I,0)=0: S=I: GOTO 1150
1140 NEXT I: OS=T: PRINT "Out of sets"
1150 RETURN 1195 '
1195 '
1200 'insert(s,x) returns(s) signals(ns,ni,nr) local(i,j)
1210 GOSUB 2000: IF NS THEN RETURN
1220 GOSUB 2100: IF NI THEN RETURN
1230 NR=F: J=0 'J holds first available cell
1240 FOR I=10 TO 1 STEP -1
1250 IF S(S,I)=X THEN RETURN 'no duplicates in a set
1260 IF S(S,I)=-1 THEN J=I
1270 NEXT I
1280 IF J=0 THEN NR=T: PRINT "No more room in set";S: ELSE S(S,J)=X
1290 RETURN
1295 '
1300 'remove(s,x) returns(s) signals(ns) local(i)
1310 GOSUB 2000: IF NS THEN RETURN
1320 GOSUB 2100: IF NI THEN RETURN
1330 FOR I=1 TO 10
1340    IF S(S,I)=X THEN S(S,I)=-1
1350 NEXT I
1360 RETURN
1395 '
1400 'is-empty(s) returns(b) signals(ns) local(i)
1410 GOSUB 2000: IF NS THEN RETURN
1420 B=T
1430 FOR I=1 TO 10
1440    IF S(S,I)>-1 THEN B=F 'test for a used cell
1450 NEXT I
1460 RETURN
1495 '
1500 'is-member(s,x) returns(b) signals(ns) local(i)
1510 GOSUB 2000: IF NS THEN RETURN
1520 GOSUB 2100: IF NI THEN RETURN
1530 B=F
1540 FOR I=1 TO 10
1550    IF S(S,I)=X THEN B=T
1560 NEXT I
1570 RETURN
1595 '
1600 'display(s) signals(ns) local(i)
1610 GOSUB 2000: IF NS THEN RETURN
1620 PRINT "Set";S;": {";
1630 FOR I=1 TO 10: IF S(S,I)<>-1 THEN PRINT S(S,I);
1640 NEXT I: PRINT "}"
1650 RETURN
1695 '
1700 'kill(s) signals(ns) local(i)
1710 GOSUB 2000: IF NS THEN RETURN
1720 FOR I=0 TO 10
1730    S(S,I)=-1 'every cell becomes unused
1740 NEXT I
1750 RETURN
1795 '
1800 'intersect(s1,s2) returns(s) signals(ns,os) local(i,j,k)
1810 S=S1: GOSUB 2000: IF NS THEN RETURN
1815 S=S2: GOSUB 2000: IF NS THEN RETURN
1820 GOSUB 1100: S3=S: IF OS THEN RETURN 'create new set for result
1830 FOR K=1 TO 10
1840    IF S(S1,K)=-1 THEN 1860 'for each member of first set,
1850    X=S(S1,K): S=S2: GOSUB 1500 'if it is a member of second set,
1855    IF B THEN S=S3: GOSUB 1200 'then insert into result
1860 NEXT K
```


(continued)

```
1870 S=S3: RETURN
1895 '
1900 'union(s1,s2) returns(s) signals(ns,os,nr) local (i,j,k)
1910 S=S1: GOSUB 2000: IF NS THEN RETURN
1915 S=S2: GOSUB 2000: IF NS THEN RETURN
1920 GOSUB 1100: S3=S: IF OS THEN RETURN 'create new set for result
1930 FOR I=1 TO 10
1935    S(S3,I)=S(S1,I) 'copy first set into result
1940 NEXT I
1950 FOR K=1 TO 10
1960    IF S(S2,K)=-1 THEN 1980 'for each member of second set,
1970    S=S3: X=S(S2,K): GOSUB 1200 'insert it into result
1975    IF NR THEN 1990
1980 NEXT K
1990 RETURN
1995 '
2000 'is-valid-set(s) return(ns)
2010 IF S<0 OR S>10 THEN NS=T: PRINT S;"not a valid set number": RETURN
2020 IF S(S,0)=-1 THEN NS=T: PRINT "Set";S;"not created yet": RETURN
2030 NS=F: RETURN
2095 '
2100 'is-not-integer(x) returns(ni)
2110 IF X<0 OR X<>INT(X) THEN NI=T:PRINT X;"not a valid integer";RETURN
2120 NI=F:RETURN
```

ables used (that is, those not part of the concrete representation or the user interface) do not conflict with those used elsewhere, whether in another abstraction, the calling procedure, or in other operations of the same data type. For this reason it is advisable to add one more item of information to the header for each operation, namely, a list of the additional local variables that it uses. Note that this is done in listing 2.

The local variables are ones that the operations use temporarily. Whatever value they may have had previously is destroyed. When a subroutine returns, their value is undefined and they are free to be used again. Keeping track of local variables becomes tricky (and imperative) when one operation calls another. The implementation of intersect (line 1800 and following) is a case in point. One

(continued)

Data abstraction allows you to modify a program simply by changing a module's implementation.

would be tempted to use I as the local indexing variable as is done in all other operations. However, intersect calls insert (line 1200 and following), which already uses I and J as local variables; we see this by looking at the *local* statement in the header of insert. This alerts us to the fact that if we use I in intersect we are in for trouble; every call to insert would destroy its value. Thus, we use a new variable, K. Note also that the local variables for intersect are given as I, J, and K, even though I and J do not appear in the code for the intersect operation. This is because a procedure always inherits the local variables of any procedure it calls.

MAINTENANCE AND PORTABILITY

From the perspective of the program code that is outside an abstract data type, we have already seen that an advantage of programming with data abstractions is that you can write programs that use them without knowing how they are implemented on the inside. Now we take the perspective of the program code inside the abstract data type and see that an advantage of programming with data abstractions is that you can modify the implementation without affecting the outside programs that use it. This is a boon for maintaining a program and porting it to other systems.

For instance, suppose we decided that limiting our sets to a maximum of 10 elements (as does the implementation of listing 2) is too restricting. We decide we want to modify our program to allow for sets of up to 20 elements. Because the information about how the intset abstraction is represented is hidden inside the intset module, any programs that use the abstraction (in this case the test program of listing 1) are not affected. The only changes to make include redefining the matrix dimensions in the initialization procedure (lines 1050–1080) and changing the upper bound of the FOR statements in all the subroutines for the operations. We soon discover that this latter change gets rather tedious and that we would have been better off in the first place to make the maximum size of a set a variable in the concrete representation of intset, and then to use that variable in the FOR loops for all the subroutines. Then changing the maximum size of sets would mean chang-

(continued)

ing only one assignment statement in the concrete representation of the data type. In any case, all program modifications are confined to the intset module.

Unfilled sets waste space, and increasing the size of the matrix to accommodate potentially larger sets has the disadvantage of wasting even more space. One solution is to represent an intset as a linked list of elements so that the set never uses more memory than it needs. With such a scheme, the maximum length of a set depends on the total length of all other sets, rather than a fixed limit for each set.

Normally such a change would require a complete rewrite. When data abstraction is used, it means replacing only the implementation of the intset module; the programs that use intset change in no way. As an example, listing 3 gives a completely new implementation of the intset abstraction using a linked-list representation. This implementation adheres

(continued)

Listing 3: *Linked list implementation of* intset *abstraction.*

```
1000 'Abstract data type: INTSET
1005 '
1010 'Representation:
1015      ' ST(10) is the "Set Table"
1020           'ST(I) specifies the address in S of the first element of set I
1025           'ST(I) = -1 means set I not yet created
1030      'S(100,1) stores the set elements
1035           'Column 0 stores a set element
1040           'Column 1 is link which specifies address in S of next element
1045           'Column 0 value of -1 marks end of list (and of set)
1050      'A points to next available cell of S
1055           'Initially all cells of S are linked together on available list
1060 'Initialize the INTSET storage:
1065      DIM ST(10): FOR I = 0 TO 10: ST(I) = -1: NEXT I
1070      DIM S(100,1): FOR I = 0 TO 99: S(I,1) = I + 1: NEXT I 'link all cells
1075      S(100,0) = -1: A = 0 'put entire list of cells on available list
1080      RETURN
1095 '
1100 'create( ) returns(s) signals (os) local(i,n)
1110 OS = F
1120 FOR I = 0 TO 10
1130      IF ST(I) = -1 THEN 1150 'find first unused set
1140 NEXT I: OS = T: PRINT "Out of sets": RETURN 'fail with OS if none
1150 GOSUB 2200: IF NR THEN OS = T: RETURN 'or if no more room
1160 ST(I) = N: S(N,0) = -1: S = I: RETURN 'initialize set
1195 '
1200 'insert(s,x) returns(s) signals(ns,ni,nr) local(i,n)
1210 GOSUB 2000: IF NS THEN RETURN
1220 GOSUB 2100: IF NI THEN RETURN
1230 GOSUB 1500: IF B THEN RETURN 'no duplicates
1240 GOSUB 2200: IF NR THEN RETURN
1250 S(N,1) = ST(S): ST(S) = N S(N,0) = X 'insert the element
1260 RETURN
1295 '
1300 'remove(s,x) returns(s) signals(ns,ni) local(i,j)
1310 GOSUB 2000: IF NS THEN RETURN
1320 GOSUB 2100: IF NI THEN RETURN
1330 I = ST(S): J = 0 'J will point to cell preceeding one to remove
1340 IF S(I,0) = -1 THEN RETURN
1350      IF S(I,0) < > X THEN J = I: I = S(I,1): GOTO 1340 'find element to remove
1360 IF J = 0 THEN ST(S) = S(I,1) ELSE S(J,1) = S(I,1) 'remove it
1370 S(I,1) = A: A = I 'return removed cell to available list
1380 RETURN
1395 '
1400 'is-empty(s) returns(b) signals(ns)
```

(continued)

```
1410 GOSUB 2000: IF NS THEN RETURN
1420 IF S(ST(S),0)=-1 THEN B=T ELSE B=F
1430 RETURN
1495 '
1500 'is-member(s,x) returns(b) signals(ns,ni) local(i)
1510 GOSUB 2000: IF NS THEN RETURN
1520 GOSUB 2100: IF NI THEN RETURN
1530 I=ST(S)
1540 IF S(I,0)=-1 THEN B=F: RETURN 'end of set, not found
1550      IF S(I,0)<>X THEN I=S(I,1): GOTO 1540 'not yet, keep looking
1560 B=T: RETURN 'it is found
1595 '
1600 'display(s) signals(ns) local(i)
1610 GOSUB 2000: IF NS THEN RETURN
1620 PRINT "Set";S;" = {";
1630 I=ST(S)
1640 IF S(I,0)=-1 THEN 1660
1650      PRINT S(I,0);: I=S(I,1): GOTO 1640
1660 PRINT "}": RETURN
1695 '
1700 'kill(s) signals(ns) local(i)
1710 GOSUB 2000: IF NS THEN RETURN
1720 I=ST(S)
1730 IF S(I,0)<>-1 THEN I=S(I,1): GOTO 1730 'find end of set
1740 S(I,0)=0: S(I,1)=A: A=ST(S) 'return cells to available list
1750 ST(S)=-1: RETURN 'mark ST entry unused
1795 '
1800 'intersect(s1,s2) returns(s) signals(ns,os) local(i,j,n)
1810 S=S1: GOSUB 2000: IF NS THEN RETURN
1815 S=S2: GOSUB 2000: IF NS THEN RETURN
1820 GOSUB 1100: S3=S: IF OS THEN RETURN 'create a new set for result
1830 J=ST(S1)
1840 IF S(J,0)=-1 THEN 1870
1850    X=S(J,0): S=S2: GOSUB 1500 'if a member of S1 is in S2,
1855    IF B THEN S=S3: GOSUB 1200:IF NR THEN 1870 'then insert in result
1860    J=S(J,1): GOTO 1840
1870 S=S3: RETURN
1895 '
1900 'union(s1,s2) returns(s) signals(ns,os,nr) local(i,j,n)
1910 S=S1: GOSUB 2000: IF NS THEN RETURN
1915 S=S2: GOSUB 2000: IF NS THEN RETURN
1920 GOSUB 1100: S3=S: IF OS THEN RETURN 'create a new set for result
1930 J=ST(S1) 'insert every element of S1 into result
1940 IF S(J,0)=-1 THEN 1960
1945    X=S(J,0): S=S3: GOSUB 1200: IF NR THEN 1990
1950    J=S(J,1): GOTO 1940
1960 J=ST(S2) 'insert every element of S2 into result
1970 IF S(J,0)=-1 THEN 1990
1975    X=S(J,0): S=S3: GOSUB 1200: IF NR THEN 1990
1980    J=S(J,1): GOTO 1970
1990 S=S3: RETURN
1995 '
2000 'is-not-set(s) returns(ns)
2010 IF S<0 OR S>10 THEN NS=T: PRINT S;"not a valid set number": RETURN
2020 IF ST(S)=-1 THEN NS=T: PRINT "Set";S;"not created yet": RETURN
2030 NS=F: RETURN
2095 '
2100 'is-not-integer(x) returns(ni)
2110 IF X<0 OR X<>INT(X) THEN NI=T:PRINT X;"not a valid integer":RETURN
2120 NI=F: RETURN
2195 '
2200 'get-next-cell( ) returns(n) signals(nr)
2210 IF S(A,0)=-1 THEN PRINT "No more room": NR=T: RETURN
2220 N=A: A=S(A,1): NR=F: RETURN
```

to the interface of figure 6 and therefore is equivalent to the matrix implementation in its external effects. The test program works equally well, whether one combines it with the intset module of listing 2 or the module of listing 3.

Since nearly any two implementations of BASIC differ in some details, BASIC programs turn out to be of limited portability in actual practice. Writing software in terms of data abstractions is an excellent way to enhance a program's ultimate portability. Information hiding localizes the details of implementation that are likely to be changed, such as variable names, matrix dimensions, and input/output protocols.

Passing parameters with assignment statements and calling GOSUB in most lines of a program can get tedious. It may often seem justifiable on the grounds of efficiency to access a data structure directly without going through the operations. You should avoid such temptations at all costs: you'll pay the price when it comes time to debug, enhance, or port the program. ■

REFERENCES

1. Guttag, J., and J. J. Horning. "The Algebraic Specification of Abstract Data Types." *Acta Informatica*, volume 10, 1978, pages 27–52.
2. Hoffman, C. M., and M. J. O'Donnell. "Programming with Equations." *ACM Transactions on Programming Languages and Systems*, volume 4, 1982, pages 83–112.
3. Liskov, B. "Modular Program Construction Using Abstraction." *Abstract Software Specifications*. D. Bjørner, ed. *Lecture Notes in Computer Science*, number 86. Berlin: Springer-Verlag, 1980, pages 354–389.
4. Parnas, D. L. "A Technique for Software Module Specification with Examples." *Communications of the ACM*, volume 15, 1972, pages 330–336. Reprinted in *Communications of the ACM*, volume 26, number 1, 1983.

ACKNOWLEDGMENTS

The author is indebted to two of his colleagues of the Summer Institute of Linguistics: to Graeme Costin for many helpful comments on an earlier draft of the article, and to Dr. Joseph E. Grimes (also of Cornell University) for some insights on programming with data abstractions in BASIC.