D. Gries
Editor

Protection in Programming Languages

James H. Morris Jr.
University of California

Linguistic mechanisms which can be used to protect one subprogram from another's malfunctioning are described. Function-producing functions and various type-tagging schemes are considered. An attempt is made to distinguish between access limitation and authentication.

Key Words and Phrases: protection, types, environments, trademarks, seals, access keys, access control, authentication, secrecy

CR Categories: 4.2, 4.3

Author's address: Department of Computer Science, University of California, Berkeley, CA 94720.

1. Introduction

The ability of a programming language or system to protect and isolate program modules is vital. As a set of coexisting and perhaps cooperating programs grows, the proper functioning of any particular program should not be endangered.

This paper explores a few protection mechanisms as they exist or might exist in a programming language. In doing so we hope to illuminate certain devices found in operating systems (e.g. capabilities and access control) as well as programming languages (e.g. function-producing functions, programmer-defined data types).

Who is to be protected from what? Although the need for protection is usually inspired by situations involving hostility (e.g. competing government agencies sharing a computer system), experienced programmers will attest that hostility is not a necessary precondition for catastrophic interference between programs. An undebugged program, coexisting with other programs, might as well be regarded as having been written by a malicious enemy—even if all the programs have the same author! The protection problem to be dealt with here is: how does one allow programs to cooperate and communicate and yet minimize the confusion attendant upon a program's malfunctioning? There are other kinds of protection problems of a less microscopic nature (e.g. proprietary programs) that will not be dealt with here.

We offer the following as a desideratum for a programming system: A programmer should be able to prove that his programs have various properties and do not malfunction, solely on the basis of what he can see from his private bailiwick. To make "his bailiwick" more precise we simply equate it with a single textual region of a larger program. While we realize that logistical problems prevent a large system of programs from existing as one single program, we accept this artificiality for the sake of precision. Thus we are only interested in those properties of a subprogram which are invariant over all the possible program texts into which the subprogram might be inserted. For example, the distinction (in ALGOL 60) between an *own* variable and one declared in the outermost block is vital given this criterion; a programmer can depend upon the privacy of his access to the former, but not the latter.

2. Procedures as Objects

The procedure or subroutine has always been a convenient device for one programmer to make his work available to others. The essence of its power is that the user of a procedure need not be concerned with the details of its operation, only the effect. If we make this ignorance mandatory by arranging that users of procedures cannot discover anything about them except by executing them, we have the basis for a very flexible protection mechanism.

Execute-only procedures are, of course, the norm; both FORTRAN and ALGOL 60 enforce that restriction. However, in order to exploit this device to its fullest extent it is useful to make procedures full-fledged objects in the sense that they can be passed as parameters or values of other procedures and be assigned to variables. Then, to make a particular object accessible to someone in a restricted way (e.g. read only), we make the object local to a procedure and pass the procedure to them in its stead. The first language to allow procedures to be passed about was LISP, with its FUNARG lists [6]. Later languages, ISWIM [5], PAL [2], and GEDANKEN [7], also have such procedures and have the execute-only restriction (LISP does not). Our example programs will be written in GEDANKEN as its definition is probably the most readily accessible to the reader. An appendix offers a brief description of GEDANKEN in terms of ALGOL 60.

Example 1. The following GEDANKEN expression yields a procedure which will return as value the number of times it has been called. The reference *Count* is inaccessible to users of the procedure.

```
[Count is Ref(0);
 λ( ){Count := Count + 1;
   Val(Count)}] □
```

3. Local Objects

We wish to make precise the notion of an object being *local* to a particular part of a program; i.e. its being inaccessible to other parts. For example the address of a variable in a FORTRAN program is local to a subroutine if it does not appear in any COMMON declaration or as an actual parameter to a subroutine call. In general we define it as follows: An object is local to a textual region *R* if the only expressions that will ever have it as a value are part of *R*.

For example, the expression Ref(0) in Example 1 is local to the whole expression. Obviously, an object cannot be local unless it is created by a unique object generator like *Ref*, or is defined in terms of such objects. For example, neither the number 3 nor the function (λXX^*3) can be local to any region.

The advantage of an object's being local is that one can enumerate all the situations in which it plays a role by looking at a single (hopefully small) region of the program.

Naturally, there can be no complete set of rules for proving that an object is local since we can easily construct a subprogram with a subexpression whose value is local iff some other, arbitrary, computation halts. However, consider the following general approach:

Consider a region *R* and some expression in *R* known to produce a unique object every time it is evaluated (e.g. *Ref*(0) is such an expression). There are three ways in which a value *V* of this expression could "escape" from *R*: It can be passed as a parameter to a procedure not defined within *R*, it can be returned as a result of a procedure called from outside *R*, or it can be assigned to a reference which is not local to *R*. Therefore, to prove that *V* is local to *R* we must check each procedure call, procedure return, and assignment which might involve *V* and show that it does not escape.

4. Memory Protection

If a memory reference is local to *R* then only statements in *R* can assign to it or take its contents. However, any imaginable kind of restricted access (e.g. read-only, write-only, increment-only) can be allowed programs outside *R* by providing them with an appropriate procedure defined inside *R*.

Example 2. Suppose a programmer wishes to provide some table-maintaining operations to the general community. He writes the program shown in Figure 1 and publishes the following set of instructions to users:

To acquire a set of table-maintaining procedures, call the (publicly available) procedure *Createtable* with the maximum number of table entries you wish to allow. It will return a three-tuple of procedures

Item, *Index*, *Save*

Item(*i*) will return the *i*th smallest item in the table. *Index*(*x*) will return the index of *x* in the table (i.e. *Item*(*Index*(*x*)) = *x*) if it is there, otherwise 0. *Save*(*x*) will put *x* into the table if it is not already there. Initially the table is empty. So much for the external specification.

Inspection of the program reveals that the references created for *Table* and *Count*, as well as the procedures *Locate* and *Move* are local to the procedure body of *Createtable*. This implies that every assignment to the references is performed by statements inside the region and every call of the procedures is also from the region.

Fig. 1. Table maintaining routines.

```
Creatable is λN
[Table is Vector(1,N,λI Ref(0));    //The table
Count is Ref(0);                    //Number of entries
Locate is λX[J is Ref(Count);       //Returns index of X if present,
    Loop: if J=0 then 0 else        //Otherwise negative of next
                                    smallest
        if Table(J)<X then −J else
        if Table(J)=X then J else
        {J := J−1; go to Loop}];
Move is λ(X,I)[if Count=N then go to Error* else
                                                  //Inserts X after
    {J is Ref(Count);                             //Position I
    Mloop: if J=I then Table(J+1) := X else
           (Table(J+1) := Table(J);
           J := J−1; go to Mloop);
           Count := Count+1}];
Item is λI* if 1≤I≤Count then Val(Table(I))*
      else go to Error*;
Index is λX[*I is Locate(X)
    if I>0 then I else 0*];
Save is λX[*I is Locate(X);
    if I>0 then NIL
      else Move(X,−I)*];
Item, Index, Save*]
```

Fig. 2. Interval manipulating routines.

```
[Createint is λ(X,Y) if X≤Y then X,Y else Y,X;
Min is λZ Z(1);
Max is λZ Z(2);
Sum is λ(X,Y) (X(1) + Y(1)),(X(2) + Y(2));
Createint, Min, Max, Sum]
```

Consider all the places marked with an asterisk (*) in Figure 1. At each we make the assertion

Table(1), . . . ,Table(Count)

is in ascending order. Then we may prove, using whatever degree of rigor deemed necessary, that if the assertion is true at each relevant entrance to this textual region (i.e. the entries to *Item*, *Index*, and *Save*), then it must be true at each exit (i.e. the exits of *Item*, *Index*, *Save*, and *Createtable* and the *Error* exits). Now, because *Table* and *Count*—the only objects the assertion deals with—are local, we may infer that the assertion really is true upon each entry to *Item*, *Index*, and *Save*. □

FORTRAN local variables and ALGOL *own* variables offer limited versions of memory protection. Indeed, the programs in Figure 1 could be written as a FORTRAN subroutine with multiple entry points if it weren't for the fact that we may call *Createtable* many times.

5. Type Protection

One of the fundamental ways in which one programmer provides services to another is to implement a representation of a new kind of object. For example, a system programmer may represent files with disk-tracks and tape records, or a compiler writer may represent complex numbers with pairs of reals. Generally speaking, the first programmer writes some subroutines which create, alter, and otherwise deal with the new kind of objects. These new objects are represented by some older kinds.

Suppose we wish to provide a new data type, *intervals*, to the general public. Among the primitive routines to be provided are *Createint*(x,y) which creates an interval from x and y, *Max* and *Min* which yield the appropriate endpoints, and *Sum* which adds two intervals to produce a new one. We might implement them as shown in Figure 2.

Now the question arises: how is a user of these primitives to regard the objects that *Createint* produces—as intervals or as common, ordinary pairs, which they obviously are? Most people would agree that he should do the former and would regard as a type violation anything assuming the latter. Specifically, there are three kinds of mishap which we might regard as type violations—i.e. use of an object in a way not intended by its constructor.

1. *Alteration.* An object that involves references may be changed without use of the primitive functions provided for the purpose.
2. *Discovery.* The properties of an object might be explored without using the primitive functions; e.g. we may select a component of a pair representing an interval.
3. *Impersonation.* An object, not intended to represent anything in particular, may be presented to a primitive function expecting an object representing something

quite specific; e.g. we might call *Max* with a pair not constructed by *Createint* or *Sum.*

There are two parties involved in such mishaps: the programmer of the primitive procedures and the user (misuser, actually) of them. Let us examine the impact upon them.

If an alteration or impersonation occurs, the primitive functions might malfunction in a drastic way which would embarrass their programmer, e.g. cause a mysterious run-time error or never halt. For example, if *Max* were called with something that was not even a pair, errors would result. While good programming practice might dictate that a programmer make no assumptions about input parameters and always perform checks, doing so can be difficult. Suppose, for example, that we chose to represent sets with sorted vectors. If the programmer were required to check each set-representing parameter for well-formedness (i.e. being sorted), there would be little point in using a fast searching method such as binary search to locate an item. For to check that a vector is sorted, we must examine each element and therefore might as well do the locating at that time.

From the programmer's point of view, discovery presents no direct threat. If in using programs other than his own, he simply looks at the objects being used as representing something, the proper functioning of his programs is not impaired. There are more subtle considerations, but we won't deal with them here.

Now let us consider the user of the primitive functions. First, if his violation of the type rules is genuinely a mistake, he wishes to be informed of it as soon as possible. If he really thinks of the objects he deals with as intervals, there is no reason why he would wish a type violation to occur. However, if he is aware of the representation activity, he may wish to violate the type restrictions for efficiency's sake. Also, a programmer may discover entirely unplanned for, yet quite useful, applications of the primitive operations; e.g. using *Sum* to add pairs representing complex numbers. While recognizing that such activity is not entirely reprehensible, let us now consider devices for preventing it. These devices are basically generalizations of the type tags familiar to any implementor of a language with dynamic data types. Here they are being made available to the general user. Our tagging operations are more general (and expensive) in two respects. First, they are "cascadable": a tagged item may be retagged. This feature reflects our belief that representation can be a multilevel affair. Second, they are completely dynamic in the sense that a running program could generate an arbitrarily large number of different type tags. The practical virtues of this generality are not entirely clear.

[1] The use of a function-producing-function here is not essential; *Createseal* could return a pair *lock*$_i$, *key*$_i$ where *lock*$_i$ and *key*$_i$ are objects of special types usable by the generally available functions *Seal* and *Unseal,* respectively.

6. Seals

Define a universally available function of no arguments, *Createseal.* Each time it is called, *Createseal* yields a pair

Seal$_i$, *Unseal*$_i$

where i is a serial number which changes every time *Createseal* is called. *Seal*$_i(x)$ yields a new object x' such that any operation applied to x' except *Unseal*$_i$ results in an error (or some other abnormal condition). *Unseal*$_i(x')$ yields x, if x' was produced by *Seal*$_i$, and results in error otherwise. The object x' may. of course, be passed as a parameter or value of a function, but its properties cannot be examined except by *Unseal*$_i$.[1]

The interval manipulating functions can now be revised as shown in Figure 3.

The programmer of these functions may depend upon the following:

1. The value of *Unseal*(E) had to have been an argument of *Seal* at a previous time. Therefore, we have the inductive rule: to prove $\rho(\mathit{Unseal}(E))$ for all E prove that $(\forall E)[\rho(\mathit{Unseal}(E))]$ implies $\rho(F)$ for every F to which *Seal* is applied. (Proving this is easier if *Seal* is local to a small region.)
2. If *Unseal* is local, applying *Seal* to an object x does not endanger its being local (even if *Seal*(x) is not local).

In this particular case $\rho(X)$ is "X is a pair of numbers (X_1, X_2) and $X_1 \leq X_2$." By assuming it is true of P' and Q' in *Sum*, we can easily prove it is true of *Seal*'s argument, $((P'(1))+(Q'(1))),((P'(2))+(Q'(2)))$. Furthermore, ρ is true of the expression **if** $X \leq Y$ **then** X,Y **else** Y,X in *Createint*, so we have established by (1) that ρ is true of all objects produced by *Unseal.*

Furthermore, by (2), any user of the primitive functions will not be able to examine the objects except through the functions provided.

The *Seal* mechanism provides two separable functions of protection: authentication and limited access. The first is embodied in (1); the second in (2).

7. Trademarks

If we are not concerned with limiting access to a class of objects, we could make the *Unseal* operation publicly available. The same effect can be achieved in a more convenient way by use of a new primitive function, *Createtrademark.* It is called with no arguments and returns a pair

Mark$_i$, i

where i is an ever-changing serial number as before. $Mark_i(x)$ yields an object x' that behaves exactly like x except that it bears trademark 1. *Trademarklist*(x') is a publicly available operation which yields a list of the trademarks borne by x'. (By lists we mean nested pairs.) $Mark_i$ can be used in place of $Seal_i$ and "*if* $i \in$ *Trademarklist*(x) **then** x **else go to** *Error*", in place of *Unseal*(x). A suitably rephrased version of the induction rule in (1) still applies.

If the reader doubts the utility of this device, stripped of any access limiting power, we urge him to consider how trademarks and other authenticating marks are used in the commercial/bureaucratic sphere. A mark from the Underwriters' Laboratories attests that a particular device may be used without risk, a fact that a user might discover only by extensive experience. A letter saying, "You have been drafted," is much more likely to be believed if it bears the mark of a draft board. In neither case is limited access a consideration. The fact that marks can be forged in the real world is of no concern here.

The access-limiting ability of the *Seal* mechanism is already present in the local variable conventions of GEDANKEN. We can use trademarks and function-producing functions to simulate *Seals.* Figure 4 displays the programs. When *Createseal* is called it creates two trademarks and then the functions *Seal* and *Unseal* which use them. A sealed object is represented by a trademarked function which will divulge the original object only when called with a password.

It is clear that this simulation provides the authentication of the original *Seal* primitives. For, *Unseal* transfers to *Error* unless its argument bears trademark 1; and an object can bear trademark 1 only if it was produced by *Seal,* because *Mark*1 is local.

To show that access to sealed objects is limited to holders of the *Unseal* function we argue as follows: From the argument in the preceding paragraph we know that any object bearing trademark 1 is a function which arose from the λ-expression in *Seal.* Therefore every Y applied to *Password* in *Unseal* is an incarnation of that λ-expression. Hence P is the only other expression that can ever have the same value as *Password.* (Applying *Trademarklist,* a primitive, to P does not endanger this claim.) Therefore *Password* is local. Now suppose $x' = Seal(x)$. By inspection of *Seal,* the only way x can be recovered from x' is to apply x' to something marked by trademark 2. Since *Mark*2 is local we see that *Password* is the only such object, and it is also local. Therefore, the only way to get x' applied to an object bearing trademark 2 is to apply *Unseal* to x'.

Since seals can be simulated by trademarks without undue loss of efficiency we are inclined to discard seals as a basic semantic unit, invoking Occam's razor, orthogonality, or some other folk principle.

There is another reason for preferring trademarks to seals, however; they offer greater flexibility. Suppose we have some class of objects (e.g. ordered pairs of num-

Fig. 3. Protected interval manipulating routines.

```
[Seal, Unseal is Createseal ( );
Createint is λ(X,Y) Seal (if X≤Y then X,Y
                                  else Y,X);
Min is λP(Unseal(P))(1);
Max is λP(Unseal(P))(2);
Sum is λ(P,Q) [P' is Unseal(P);
              Q' is Unseal(Q);
              Seal ((P'(1)+Q'(1)),(P'(2)+Q'(2)))];
Createint, Min, Max, Sum]
```

Fig. 4. Simulation of Seals with trademarks.

```
Createseal is λ( )
  [Mark1, I1 is Createtrademark( );
  Mark2, I2 is Createtrademark( );
  Password is Mark2(NIL);
  Seal is λX
    Mark1(λP if I2∈Trademarklist(P)
                then X
                else go to Error);
Unseal is λY
    if I1∈Trademarklist(Y)
      then Y(Password)
      else go to Error;
Seal,Unseal]
```

(Note: The expression $x \in L$ is true iff x is on list L).

bers) which are used to represent various other classes (e.g. intervals, points in the plane), in a variety of ways (e.g. cartesian versus polar coordinates [3]). These objects may in turn represent others (e.g. a point in the plane might represent a complex number or a vertex in a geometrical figure). We can invent trademarks to attest to these various properties and attach them to objects in any order we like. For example, a complex number represented in the cartesian system might bear four marks: pair, point-in-plane, cartesian, complex. Programs may examine these marks or not as they please; e.g. a routine that prints pairs is unconcerned with all but the pair tag. If seals were used in this kind of situation great complications would result. Every routine that dealt with pairs would have to be equipped with a battery of *Unseal* operations which it would use to unlock all the nested seals; to make matters worse it would have to have a way of determining which *Unseal* was called for at each stage.

Naturally, when we wish to limit access to a class of objects as well as authenticate them, the full facilities of the *Seal* mechanism are called for. Only in situations involving no need for suppressing information is the trademark best.

Fig. 5. Simulation of trademarks with access control.

```
[Oblist is Ref(NIL);  //List of all Marked Objects, really nested
                      //triples.
Serialnumber is Ref(0);
Findob is λX[E is Ref(Oblist);
  Loop: if E=NIL then NIL else   //E(1) is unique name
        if E(1)=X then E(2) else  //E(2) is (TMlist,Object)
        {E:=E(3); go to Loop}]; //E(3) is next item
Trademarklist is λX
  [T is Findob(X);
    if T=NIL then NIL else T(1)];
Get is λX
  [T is Findob(X);
    if T=NIL then X else T(2)];
Createtrademark is λ( )
  [I is Val(Serialnumber);
    Mark is λX //Always add to Oblist
  [T is (I,Trademarklist(X)), Get(X);
    Oblist := Ref(NIL), T, Val(Oblist);
    Oblist(1)];
  Serialnumber .= Serialnumber + 1;
  Mark, I];
Createtrademark, Trademarklist, Get]
```

8. Access Keys

Now let us examine the implications of making the *Seal* operation public and leaving *Unseal* private; e.g. *Seal*(i,x) places seal number i on x and *Createseal* produces the pair

Unseal$_i$, i.

If one puts such a seal on an object and leaves it in a public place, he is guaranteed that only the restricted group of programs holding *Unseal*$_i$ can access it. Those programs, however, cannot depend upon an object thus sealed to have any particular properties since anyone can seal an object with any number. This mechanism is similar to the access key described in [4]; each *Unseal*$_i$ plays the role of an access key.

Access keys are symmetric to trademarks in the sense that they allow many unprivileged senders to give information to one privileged receiver, the holder of the *Unseal* operation. With trademarks the sender of information has the privilege.

These new *Createseal* and *Seal* operations can also be simulated using the trademark primitives.

9. Authentication vs. Access Control

There seems to be a chicken-egg relation between these two notions that make them hard to disentangle. Access to the authentication operation, e.g. *Mark*, must be controlled if it is to be useful. On the other hand, gaining access to things sometimes requires that a program provide authenticated evidence of its access rights.

It seems clear that access control is a sine qua non of security; if every program has access to every object and operation there is no protection. The case is not so clear for authentication. Given the access control mechanisms already present in GEDANKEN, i.e. local references, we can simulate the trademark system if we require the user to use a function, *Get*, when accessing a marked object. (This is a minor inconvenience.) The idea is to hold all objects submitted to *Mark* in a protected list (*Oblist*), paired with their trademark lists. Each such pair is given a unique name which is used to retrieve the object or its trademark list. GEDANKEN's *Ref* operation is an admirable generator of unique names. The scheme works because access to the list is limited. Figure 5 shows an expression which evaluates to produce the functions *Createtrademark*, *Trademarklist*, and *Get*. The idea behind this program should be familiar to those who understand how *C*-lists [1] are used; basically, we are using memory protection (rather clumsily) to achieve type bit protection.

This simulation of trademarks is unsatisfactory in two respects. First, one can say that we really haven't started without type protection since we use references to simulate marked objects and they are presumably marked in some way. To avoid this charge we could have used integers instead, keeping a counter to generate new ones. However, this is not entirely adequate since it would then be impossible for a trademarked object to be local to any region. The second shortcoming is that it is wildly inefficient: a list must be searched just to access an object, and marked objects will never be garbage collected even after all references outside the protected region have disappeared. At present we see no way of overcoming these problems.

We believe that authentication is a notion orthogonal to limitation of access. Our belief rests upon intuition and the fact that the former seems difficult to simulate with the latter.

Remarks

The ideas presented here are, in a sense, a repackaging of familiar notions from programming languages and operating systems: scope rules, capabilities, memory protection, and type bits. We have considered these devices in extremely general forms, and it is entirely possible that less general devices may be sufficient and economically more reasonable. Specifically, the function-producing functions, indefinitely cascadable type tags, and run-time creation of types entail substantial costs by current standards. Finding reasonable compromises between these facilities and ones efficiently implementable today is a subject of continuing research.

This work builds upon the ideas of many people, most notably, Lampson [4], Landin [5], Reynolds [7], and Dennis [1].

Acknowledgments. Earlier versions of this paper were given sympathetic readings by Butler W. Lampson and James N. Gray.

Received June 1971; revised January 1972

References

1. Dennis, J. B., and Van Horn, E. Programming semantics for multiprogrammed computations. *Comm. ACM 9*, 3 (Mar. 1966), 143–155.
2. Evans, A. PAL—A language designed for teaching programming linguistics. Proc. ACM 23rd Nat. Conf. 1968, Brandon Systems Press, Princeton, N.J., pp. 395–403.
3. Fischer, A.E., and Jorrand, P. BASEL: The base language for an extensible language facility. Rept. CA-6806-2811, Computer Associates, Wakefield, Mass., 1968.
4. Lampson, Butler W. Dynamic protection structures. Proc. 1969 FJCC. The Thompson Book Co. pp. 27–38.
5. Landin, P.J. The next 700 programming languages. *Comm. ACM 9*, 3 (Mar. 1966), 157–166.
6. McCarthy, J., et al. *LISP 1.5 Programmers' Manual*, M.I.T. Press, Cambridge, Mass., 1962.
7. Reynolds, J.C. GEDANKEN—A simple typeless language based on the principle of completeness and the reference concept. *Comm. ACM 13*, 5 (May 1970), 308–319.

Appendix. A Brief Comparison of Gendanken with Algol-60

The basic means of introducing new identifiers is the **is**-declaration which is used in three ways here:

1.	*Count* **is** *Ref*(0)	is like	**integer** *Count*; *Count* := 0
2.	*Locate* **is** $\lambda X[\cdots]$	is like	**integer procedure** *Locate*(*X*); **begin** ⋯ **end**
3.	*I* **is** *Locate*(*X*)	is like	**integer** *I*; *I* := *Locate*(*X*)

except subsequent assignments to *I* are forbidden. The types of identifiers are never declared. The declaration and calls of parameterless procedures employ the null parameter list, written "( )".

References (addresses) are dealt with more explicitly. *Ref* is a function which returns a new reference, containing its argument. If *r* is a reference, *Val*(*r*) is its contents, and $r := e$ changes its contents to *e*. A set of coercion rules (not given here) serve to insert applications of *VAL* so that, if *Count* is a reference,

Count+1	means	*Val*(*Count*)+1
$I \leq$ *Count*	means	$I \leq$ *Val*(*Count*)
Ref(*Count*)	means	*Ref*(*Val*(*Count*))
and *Count*(1)	means	(*Val*(*Count*))(1)

Vectors and *n*-tuples are subsumed under the more general notion of functions. The expressions

(2,4,6)

and (λX **if** $1 \leq X \leq 3$ **then** $2*X$
else go to *Error*)

and *Vector*(1,3,λX $2*X$)

all denote the same thing: a function *F* such that $F(1) = 2$, $F(2) = 4$, $F(3) = 6$. If *Createseal*( ) returns a pair, the declaration

Seal, *Unseal* **is** *Createseal*( )

serves to decompose it and bind the components to the two identifiers.

The general form for a (large) GEDANKEN expression is a sequence of (0 or more) **is**-declarations followed by (1 or more) expressions ($X := y$ is an expression).

$[d_1;\cdots;d_n;e_1;\cdots;e_m]$

The declarations and expressions are evaluated from left to right and the value of the whole expression is e_m. The (textual) scope of the definitions is limited by the brackets.

To make the programs more readable we write key words of GEDANKEN in boldface and capitalize the first letter of identifiers. Also variously-shaped brackets are used. Comments are written to the right of double slashes (//).