SETL Newsletter # 171

R. Dewar
J. Schwartz
E. Schonberg
April 29, 1976

More on Basings

1. Introduction.

This newsletter aims to clarify some of the questions surrounding our based represention scheme, and to prepare for its implementation. The scheme to be outlined will be one which is highly conservative and only modestly automatic; this should limit the complexity of the implementation problems which must be faced. Experience with the outcome of this first scheme should clarify many of the problems which a more highly automatic scheme must face.

Among the general principles which shape our design approach, we note the following:
There must exist some way of using the basing scheme efficiently (since otherwise it serves no purpose.) Moreover, attained efficiency must be predictable, i.e., the efficiency of a program must not depend on subtle program details which guide an optimizer's implicit decisions. In particular, we will always insist that our compiler annotate a program in a way which indicates all points at which it has inserted potentially expensive conversion or copy operations.

Some of the technical decisions which determine the scheme to be outlined are as follows:

(a) We shall treat all declared basings as static, strict mode determinations.

(b) A set of default rules determining the basing of x op y from the basings of x and y will be used.

(c) Not all basing declarations will be accepted by the compiler; i.e., certain semantic restrictions to be stated below will apply, and basings violating these restrictions will generate (fatal) diagnostics.

(d) Assignments x = y and f(x) = y may convert the value y being assigned from one basing to another, depending on the basing which has been declared for x (and/or f). The same applies to assignments made using the is operator.

(e) The basing of external variables will have to be declared in all detail. The binder will check to ensure that these declarations are consistent throughout all the modules of a program being bound together.

Basing must be supplied for procedure and function parameters.

(f) We will use a combination of static and dymanic techniques which together ensure that the based and unbased versions of a SETL program can never give different results if both run to completion. However, a program with declared basings can fail at run time where the same program with basings deleted would not.

(g) To achieve efficient use of basings and conformity with all restrictions, some modest degree of program restructuring may be required. However, a program with acceptable basings will always run giving the same result if the basings are switched off.

(h) Pure SETL, i.e., SETL with no basings will operate with a 'safe' subset of the larger library of implementation-level data structures provided to support the full based system. A program with declared basings will use a larger variety of implementation-level structures than its unbased variant. Any SETL operation can be applied to based objects, and if it does not fail, the operation will yield a result having the same abstract value as if the objects were unbased.

Failure will never result if the restrictions stated below are observed. The user of based SETL, familiar with these restrictions, may anticipate basing failures at certain program points, and to avoid such failure, may want to convert a particular object x to its standard, safe, unbased SETL form xx. For this purpose, we provide an elementary mode unbased. Any value assigned to a variable of this mode is converted to unbased form.

(i) A variable of declared mode is constrained to have its declared basing; an undeclared variable can have a basing which varies dynamically. Assignments to undeclared variables will in principle never force a basing conversion, whereas an assignment to a variable of declared mode will force such a conversion if a basing discrepancy is detected (statically or dynamically). All points at which such conversions are inserted will be annotated appropriately by the compiler. However, in the case of variables, such as compiler temporaries and local procedure variables not passed between subprocedures, for which the compiler might be able to deduce a basing strictly equivalent to standard SETL form and demonstrably more efficient, we allow compiler-chosen variations of basing as a possibility.

Whenever an undeclared x is assigned to a declared y, the mode of x will be checked dynamically, and its conformity with the mode of y will be established. If this check fails, a coercion will be applied to force x into the form declared for y. Even if no coercion is necessary, we will check to verify that none of the bases implicit in x has undergone any 'dmod'. (see below).

The conversion of a general variable x to y ∈ s basing will always be done by hashing.

(j) Sharing of a base will not be permitted, so that a base s will be copied on every assignment t = s, f(x) = s, etc., which would otherwise have to set the shared bit of s. The fact that assignments to a base and assignments from a base are treated differently from other assignments means that we will always need to know statically an object is a base.

(k) Note that a base sb can be erased or diminished when no longer needed, without necessarily making objects which are weakly (but not strongly) based on sb unusable. (see below) For example, suppose that we have made the following declaration:

```
declare   sb:base set,
          fb:smap (∈ sb ) int,
          s: base set (∈ sb),
          f: smap (∈ s) int;
```

Then s and f remain usable even after an assignment sb = nl; whereas fb becomes unusable.

2. Restrictions

Basing secures efficiency by using pointer mechanisms internally; criteria which restrict these uses of pointers to force compatibility with the value approach of SETL will therefore be required.

Several types of pointers can cause trouble:

(a) If x is based as an element of s, then x will point to one of the element blocks in the representation of s; map and attribute values f(x) will be held in this block. If s is changed (e.g., by an assignment s = expn), or replaced by a copy of itself, or stacked or unstacked by a recursive call, these pointers will not be changed; then if a subsequent assignment f(y) = .. changes f(x), this change may not be properly recorded. The same thing can happen if x is deleted from s and then subsequently re-inserted (creating a new element block, but leaving x pointing to the old element block).

(b) The same difficulties are connected with sets s_1 declared to be based as a set of elements of s, since in this case each of the elements of s_1 will contain a pointer to an element block of s. Moreover, if some based map g is declared to have range values based as elements of s, then the same is true for the elements of g.

(c) If g is a map, and is declared to be a map based on the domain s and is represented by a remote vector, then (assuming that serial numbers of elements of s can be re-used) we have the same difficulties. The same is true in connection with attribute maps, sets represented by remote bitvectors, sets of remote maps like g, etc.

The difficulties raised by case (c) are more severe than those inherent in cases (a) and (b), since evaluation of g(x) can involve hashing of x to *locate* an element block of its base s. If an element x of the domain of g has been deleted from s, then hashed access to g(x) is bound to yield the value Ω, which can be incorrect.

Moreover, we have no easy way of detecting that this value is incorrect.

The path that we choose through this set of problems will reflect an underlying view of the fundamental semantic implication of a basing declaration. What do we wish to imply when we declare x:∈ s, f:smap(∈s) int? At least three significantly different interpretations, each stricter than its predecessor, are possible:

(a) We may imply only that the value of x is a member of s, and that all values of f can be located via information stored with the elements of s, whenever x and/or f are *actually used*.

(b) We may imply that these same relationships hold *at every moment during program execution* at which x and f have values different from Ω.

(c) We may imply that x and f are dead whenever s is changed or diminished.

Alternative (a) is attractive *a priori*, but does not seem to be implementable. Assuming that x and f are declared as above, consider the sequence

x = ∋s; s less y;...; s with y; z = f(x).

If y = x, the operation s less y will mark the element pointed to by x as deleted; the subsequent s with y will re-insert the element x into s, but, having no way of telling that f(x) should still have some prior value, will set up a new s-list element in which f(y) is shown as having the value Ω. If we try to treat the subsequent operation f(x) as legal, e.g., by rehashing when we find that the list element pointed to by x has been marked deleted, we will get the incorrect value Ω for f(x). It is not clear how this difficulty can be avoided.

In alternative (b), the operation f(x) will in principle be illegal if s has been diminished in such a way as to omit an element x that belongs to the domain of f.

However, we have no way of enforcing this rule statically, nor does there exist any plausible combination of static and dynamic checks which enforces it. Even if elements of s are dynamically flagged as deleted when they are removed from s, we still have no good way of dealing with cases like

x = ∋s; s less (x + 1);... s with (x +1); z = f(x+1);

in which the element of s from which the desired value of f is being retrieved is located by hashing.

These difficulties drive us to adopt something close to a variant of (c). The restrictions that (c) implies can be checked statically; however, we shall prefer to take a roughly equivalent dynamic approach as fundamental, and to regard the competing static approach as an optimization. For describing the details of our approach, it is useful to introduce the following terms. A set, map or other object is called *s-dependent* if it is based in such a way as to contain pointers to elements of s, or if it is realized either as a remote mapping or attribute mapping based on s or by an object containing such a mapping. An object f declared to have one of the basings

map (∈s) *rmode* or smap (∈s) *rmode* or subset(s),

or to be based in such a way as to contain any object with one of these basings is said to be *strongly s-dependent*. An object which has an s-dependent basing but which is not strongly s-dependent is said to be *weakly s-dependent*. We call any operation which modifies s and may diminish it a *dmod* (diminishing modification) of s. Any assignment to s other than the assignments s = s with x and s = s + t are potentially dmods. Drastic assignments, e.g., s = t, will be treated as if they were written s = s - (s-t) + (t-s) i.e., elements in s will be retained as long as they are not logically removed by the assignment s = t.

A use of a weakly s-dependent object is legal if it makes no use of any pointer to an element of s which was deleted from s after the operation which created the pointer. A use of a strongly s-dependent object x is legal only if s has not undergone any dmod since x was last used.

To enforce these rules, we will proceed as follows. Whenever a dmod of a base s is performed, the elements deleted from s will be flagged as having been deleted. Uses of pointers to elements of s will always test this delete flag, and execution will be terminated with a dynamic basing violation diagnostic if a delete condition is detected during pointer use. With each base s, we shall associate an auxiliary cell c; s will always be accessed and referenced indirectly through c. Whenever a dmod is applied to s, a 'dmod bit' will be set in the existing c, and a new c will be allocated and assigned to s. Operations using s which assume that no dmod of s has occured will check this dmod bit, and will fail if it has been set.

During garbage collection, all pointers to deleted set elements will be changed to point to some standard deleted element, allowing the deleted elements to be reclaimed. The same treatment will be applied to 'bad' auxiliary cells c.

When the operation s with x is performed, a quantity xx logically equivalent to x but having ∈s basing will always be calculated. This quantity can be stored, and implicitly kept available as long as x is not changed and is not subject to any dmod. The availability of xx will sometimes allow a later hashing operation to be avoided. A typical example is the sequence

```
s with x;  t with x;
```

where we suppose that t has been declared to have set (∈s) basing.

The machine code sequence for the retrieval operation f(x) in the case that f has local smap(∈s)int and x has x∈s basing is approximately

```
load      blockref _ x(1)
load      blockref _ x(f)
test      deleteflag ≠ 0
ifsogoto error.
```

On the 6600 this would be 8 cycles, as compared to 2 for a FORTRAN indexed load (without range checking). If flow analysis shows that there can have been no dmod of s since the definition of x or since its last preceeding use, then redundant check elimination can reduce the number of machine cycles to 4.

If f has remote smap(∈s)int basing, the code sequence for the retrieval f(x) is approximately

```
load      blockref_ x(1)
load      blockref_ f
test      dmodflag  ≠ 0 or deleteflag ≠ 0
ifsogoto error
load      vectref_f(index_x).
```

On the 6600 this would be 11 cycles, reducible by redundant check elimination to 5. There is therefore real hope of bringing based SETL to perform with something like 10% of the efficiency of FORTRAN, even without special hardware assistance.

3. Procedure Parameters and Calls.

In the unbased SETL semantic environment, items are handled on a pure 'value' basis, which means that no operation applied to a variable y can have any effect on a different value x. In the based situation this is not quite true, since for example if x is based on y then an assignment y = nl can make x unusable (though if x remains usable, an assignment to y can have no other effect). This is a mild kind of 'pointer' effect, and reflects the fact that pointers are used more intensively in the presence of basings than in their absence. Effects of this kind will be particularly irritating within subprocedures, since an assignment to a variable having no obvious connection with a procedure parameter might make a parameter unusable. A more serious procedure-parameter related problem arises in connection with the fact that we do not wish to have to perform large numbers of dynamic tests in connection with basings (e.g., to determine how a given quantity is based), since this could eat up the efficiency advantages made available by the basing scheme. Thus, whenever two or more quantities come together in a subprocedure, e.g., when we evaluate f(x) or make an assignment f(y) = x, we shall want to be sure that the relationship between the basings of all of them is known statically, even though this forces somewhat of a reduction of language level. Since we do not wish to accept declarations on faith, systematic static methods for checking the validity of parameter basings declared in subprocedures are necessary.

To make this possible, we apply the following restrictions to procedure calls. All variables accessed within a procedure, including procedure parameters and global variables, can have declared basings. The general rule concerning parameter-argument basing declaration consistency is that no conversion of argument form may be implied by a procedure or function call.

Thus an argument value with any basing can become the value of a read-only procedure parameter for which no mode is declared; but if a basing declaration is given for the parameter, or if the parameter is read-write, essentially the same declaration (or none) must be given for each argument passed to it.

If the manner in which a parameter is declared to be based within a procedure is not implied by the basing information available at point of call, then a basing violation exists, and the program will be rejected. Often the parameters to be passed to a subroutine will share some common base, and it can be important to declare this fact within the subroutine. However, since in general the base of a based SETL object can be located through the object, it is not necessary to pass all these bases to the subroutine explicitly. If a base sb of a subroutine parameter is not passed to the subroutine explicitly, we call sb a *virtual base* within the subroutine. Mode declarations for base sets must include the keyword base. For declaration of virtual bases, we provide the keyword virtual. The following procedure illustrates the use of this facility:

```
define      inout(s =, x, y);
declare     sb:virtual base,
            s: base set (∈sb), x:∈ sb, y:∈ s;
s with x;   s less y;
return;
end inout;
```

The value returned by a function can be declared; for this purpose, we provide the keyword returns; the mode declared for this pseudo-variable within a function is the mode of the value returned by the function. An example of its use is

```
define    sizeof(f,x);
declare   sb: virtual base,
           f: smap(∈sb) set, x: ∈ sb,
          returns: int;
return  #f(x);
end sizeof;
```

The parameter and return-value mode declaration used with procedure variables and external procedures is syntactically the same as the body of a declare statement appearing within the procedure, except that non-base names are omitted and virtual bases preceed the parenthesis surrounding the other parameter declarations while the declaration of the type of value returned follows the parameter list. Thus corresponding to the two preceeding examples, we might have

```
declare   inout: proc sb virtual (s: set (∈sb) =, ∈sb, ∈s),
          sizeof: fnct  sb virtual(f: smap(∈sb) int, ∈sb) int;
```

Since assignments to and from bases must be treated differently from other assignments, we shall insist that a base be declared as such whenever it is passed to a subprocedure. An example is

```
define     assignbase(s,t =);
declare    s:base;
t = s;
return;
end assignbase;
```

Procedure entries and exits cause stacking and unstacking of variables v, and both of these actions are in effect assignments to v. (When v is stacked, it gets the value Ω unless it is an argument; when unstacked, it is restored to some prior value.) In order to ensure that the use of bases is semantically transparent, we need to be sure that when a base sb is stacked no element based on it is live, and that when sb is unstacked all elements based on it are simultaneously returned to values which they had just before sb was stacked. Accordingly, we impose the following rule: a variable v can only be based on a stacked sb if either v and sb are stacked by the same procedure p, or if v is local to some other procedure p' and is stacked by p'.

Note that in the second of these cases we can be sure that (on stacking) v will be re-initialized (e.g., to Ω) before it is accessed, and also that each unstacking of sb corresponds to one or more unstackings of v.

In regard to parameters vp of p, we insist that they be based either on other parameters or on bases sb not stacked by p. The same rule applies to the implicit returns parameter of a function. Note that we allow procedure parameters to be declared global; this is essential if a base passed to one of the procedures of a group is to be made available to the others (e.g, for basing of local variables) without either repeated redundant parameter passing or expensive copying.

A base cannot be assigned without copying, since there will generally exist multiple references to its elements. For this reason we will not allow sharing of bases; for example, a base s will always be copied at each assignment t = s which would otherwise set its shared bit. More generally, we shall use techniques which (except in connection with the use of basepak, see below) prevent the invalid use of multiple references to a base.

It is not clear that this can be done easily by purely static means. A simplified 'difficult' case is

```
        define    recrout(s);
        declare   s base,
                  os base static;
...;    recrout(os);  $ recursive self-call
        . . .
        x = s;  os less something; y = s;
        . . .
        end recrout;
```

After the indicated recursive call of *recrout*, there exist two references, *s* and *os*, to the same base object (which for the sake of efficiency we do not wish to copy.) Thus x and y might get different values.

To suppress this difficulty, we impose a number of restrictive rules, which are enforced using a partly static, partly dynamic technique. Our rules, which make possible an efficient, essentially pointer-oriented treatment of bases without violating the SETL semantic rules, are as follows:

i. A base s can be passed multiply to a function or subprocedure provided that it is passed as a 'read only' parameter. If it is passed as a parameter which can be both read and written, then it can be passed only as the value of this parameter, and not in any other parameter position.

ii. Let sb be a local or global base variable available for writing, and suppose that it is passed to a procedure p as a read-write argument, where it becomes known as sbb. Then sb becomes unavailable for reading or writing until return from the outermost such p, though of course sbb can be both written and read. If sb is passed as a read only argument, it remains available for reading, but will become unavailable for writing.

In all cases, the parameter sbb can be declared global.

Note that an operation which accesses an object x based on sb is not considered to read sb in the sense of the preceeding paragraph; we shall call uses of sb which are required to support a reference to x *virtual* use of sb. Of course, a reference to x which makes virtual use of sb can fail if sb has been reduced since x became a member of sb.

Rule (*ii*) is enforced by incorporating two bits, a read-permit bit and a write permit-bit, into the 'value specifier' through which each variable declared to be a base is referenced (see 'SETLX data structures'). When sb is passed for reading, and assuming that before passage sb is available for writing, we drop the read and write bits in the value specifier of sb, and reallocate a new value specifier, which continues the old values of these bits and which is passed.

Bit-dropping and reallocation are unnecessary if sb is local stacked, since in this case it will in any case be unavailable until return. If sb is passed for reading only, we drop the write bit and reallocate a new value specifier, which contains the old value of these bits. In both cases, after return, the newly allocated value specifier is substituted for the modified one. The read permit bit is checked on every use of sb (resp.assignment to sb) and the use (resp. assignment) fails if this bit is set.

iii. As we have noted, a parameter of a procedure can only be based on a static global variable or another parameter. However, a local variable based on s can be assigned as the value of an undeclared parameter y and then returned. To prevent misuse of values y generated in this way, we set the dmod bit associated with every base stacked by a procedure on returning from the procedure.

Read-write procedure parameters can be used when one wishes to use a subprocedure to build up a base together with an item based on it, and then to return both items together. That is, we allow read-write parameters (like other parameters) to be declared as bases (for local procedure variables and for other procedure parameters.) As usual, the arguments with which a procedure is called must have declarations which correspond to the declarations of its parameters.

4. Basepaks.

A deficiency in the basing scheme outlined till now is that bases cannot be assigned and remain bases, which in particular makes it impossible to use a function to generate both a base s and an object f based on s, and then to return a pair consisting of <s,f> for use elsewhere. Nor can such pairs be saved and restored or otherwise manipulated. The essential difficulty in trying to do this is that we generally wish at most one reference to a base to exist at any one time. To overcome the deficiency which rigorous imposition of this rule would create, we propose to introduce an additional semantic notion, that of a basepaks. A basepak is a tuple $t = \langle v_1,\ldots,v_k\rangle$ of values, some of which can be based on other variables of t (as well as on variables external to t); however, the formation and use of basepaks will be severely restricted. The rules which apply are as follows:

i. A basepak is formed by assigning a tuple

$$\langle v_1,\ldots,v_k\rangle,$$

as the value of a variable of type basepak. (see below) Basepaks are objects which can be sets members, tuple components, and which can be passed between subprocedures. Like blank atoms, basepaks can be compared for equality and inequality (but are generally not equal), and can also be compared (comparison yields orderly but implementation-defined results.) A basepak bp can be decomposed into its components only by a multiple assignment

(1) $$\langle u_1,\ldots,u_k\rangle = bp;$$

this is the only operation which examines the internals of a basepak.

ii. A basepak is declared using a structure of the form

basepak $\langle name_1: mode_1,\ldots,name_k: mode_k\rangle$

where the component names $name_j$ are optional. If a name $name_j$ is omitted, the following colon should be omitted also.

A sample basepak declaration is

declare bp:basepak <s:set (char), set (smap(∈s) ∈t)>;

In this example, the first component of the basepak is named s and serves as a base for the second component, which is a set of single-valued maps each having domain based on s and range based on some other set t. We shall call the set t an *exterior base* of the basepak bp, whereas s will be called an *interior base*.

iii. The basepak bp described in the preceeding paragraph can be formed by executing an operation

(2) bp = <s1, ss2>.

In such a case, we require that the declaration of bp correspond precisely to that of s1 and ss2; i.e., in our example s1 and ss2 would have to be declared as

declare s1: base set (char),
ss2: set (smap (∈s1) ∈t);

Execution of the operation (2) involves formation of new copies of s1 and ss2, which during the copy operation are made to contain corresponding pointers and which become components of the basepak vector that (2) creates. We generate new logical copies of s1 and s2 to ensure that at most one reference to a base can exist at any one time; actually, it is only the components of bp which serve as interior bases that need to be copied immediately, and even these need not be copied if the variables that reference them are dead immediately following the operation (2).

iv. If the basepak bp appearing in (1) has its 'shared' bit set, or if bp is not dead immediately after the execution of (1), then execution of (1) may cause yet another copy of bp to be generated. This technique allows basepaks to be 'unpacked' more than once.

v. If a basepak value *bpx* is assigned to a variable y, or included in a set sy, then *bpx* will be converted to standard SETL form unless the declaration of y (or sy) is such as to anticipate the appearance of an element of precisely the form of *bpx*. For example, if *bpx* is of the basepak mode *bpm*, then the operation sy with bpx will convert *bpx* to standard SETL form unless sy has been declared to be of mode set(bpm).

5. Public and External Variables.

The basing of all external variables must be declared, but the bases themselves may not be declared virtual. Basing information declared for all public and external variables, and of all variables on which public or external variables are based, will be saved in appropriate tabular format as part of the compiled form of a module passed to the binder. Then, when all the modules of a program are bound together, the consistency of these declarations will be checked, and a fatal diagnostic will be given if any inconsistency is detected.

The compiler will have to take the attitude that any public global variable, and any global variable that could be modified in consequence of a call to any of the procedures of a module which can be reached by a call *from* an external procedure, might be modified by a call *to* any external procedure. For this reason, we shall prefer to restrict the external accessibility of the procedures of a module by insisting that a procedure resident in one module *ma* can only be accessed for use by another module *mb* if it has been declared public in *ma*.

6. Summary of Language Changes Proposed in Connection with Basings.

i. Additional elementary mode descriptors.

base - used to declare bases within subprocedure.
general - used to declare components of unknown basing within tuple, e.g.
<int, general, smap(s)int>
unbased - used to declare the values of a variable unbased.

ii. Basepak mode descriptor.

basepak $\langle name_1: mode_1, \ldots, name_k: mode_n \rangle$

where all the $name_j$ are variable names, and all the $mode_k$ are mode descriptors. The various $name_j$ can be used as bases in forming the $mode_j$ within the basepak. Any $name_j$ which is not used can be omitted with its following colon.

iii. Declaration of mode returned by function.

Uses the keyword returns in a declaration statement appearing within the function.

iv. Declaration of public functions of a module.

Not all the functions and procedures defined within a module, but only those to whose header line the keyword public has been affixed, are available for inclusion in other modules.

v. Virtual declaration.

A variable may be declared virtual. Variables thus declared cannot be read, except to form virtual components of basepaks (see section 4.) Virtual bases of procedure parameters cannot be modified.

Summary of restrictions:

vi. Rules for dmods.

A pointer p into a set s can only be used if the object at which it points has remained in s since p was established. An object f strongly based on s can only be used if s has not undergone any dmod since the last preceeding use or redefinition of f.

vii. Procedures and Parameters.

a. A base available globally cannot be passed as a parameter.
b. A base can be passed multiply to a function or subprocedure only if it is passed as a 'read only' parameter.
c. Procedure constants or variables y can only be assigned to a variable x if x is either undeclared or has the same declaration as y.
d. The actual arguments passed to a procedure p must have declarations which correspond precisely to the declaration of p's parameters, unless the parameters are undeclared.
e. A parameter cannot be strongly based on a local variable of a routine.
f. If a base is passed to a subprocedure it must be declared base if it might be a base even if it is not used as a base within the subprocedure. However, in this case the parameter value passed need not be a base on every call.
g. Read-only parameters of a procedure p may not be passed as read-write parameters to a subprocedure called by p.

viii. Miscellaneous.

a. Global variables can be based only on global variables.
b. The use of virtual variables is restricted in the manner described in vi, above.

7. Example of the Use of Basings.

We begin this section by writing out a based version of the Huffman-table generating routines of O.P.II, pp. 149 ff. as an example of the use of basings. We shall suppose these routines to be part of a library module called *hufrouts*.

```
module    hufrouts; $ huffman table generator, decoder routines
declare   allnodes: base set, $base set: all nodes of the huffman tree
          work: set(∈allnodes), $workpile during tree construction
          wfreq: smap(∈allnodes) real, $frequency function
          top:  ∈ allnodes,    $ top of decoding tree
          ℓ, r: smap(∈allnodes)∈allnodes, $descendant map in tree
          seq:  bool;          $auxiliary boolean string

      chars: base set(char) $ the set of characters; global parameter.
       code: smap (∈ chars) bool, $ maps characters to their codes

definef huftables(chars, freq) public; $huffman table generator routine
declare  freq: smap(∈ chars) real, $ frequency function for characters
         n, c1, c2: ∈ allnodes  $ various auxiliary nodes
        returns: smap(∈ chars) bool;

allnodes = chars; wfreq = freq;     $initialization
ℓ = nℓ; r = nℓ; work = allnodes;    $initialization
(while  # work gt 1)
          work less ((wfreq getmin work) is c1);
          work less((wfreq  getmin work) is c2);
          allnodes with (newat is nn); $use general variable, since
                                       $ nn is initially not in the
                                        $ base allnodes.
          ℓ(nn is n) = c1; r(n) = c2;
          wfreq(n) = wfreq(c1)+ wfreq(c2);
          work with n;
end while;
code = nℓ; seq = nulb;
walk (∋ work is top, code =);
return code;
end huftables;
```

```
definef wfreq getmin set;
declare keep, x: ∈ allnodes,  $ auxiliary node quantities
        least: real,          $ temporary minimum
        wfreq: smap (∈allnodes) real,
        set:   set (∈ allnodes),
        returns:    ∈ allnodes;
<keep, least> = < ∋ set is x,wfreq(x)>;
(∀ x ∈ set)
     if wfreq(x) lt least then <keep, least> = <x, freq(x)>;;
end ∀x;
return keep;
end getmin;
define walk(tap);           $ tree-walk routine
declare   tap: ∈ allnodes,


if  ℓ(tap) ne Ω  then
          seq || f; walk(ℓ(tap));
          seq || t; walk(r(tap));
else
          code(tap) = seq;
end  if;
seq = seq (1: # seq - 1);
end walk;
definef cseq (bitseq) public;     $ huffman decoding routine
declare      output: char,        $ decoded string
             node:∈ allnodes,     $ auxiliary node
             bitseq: bool,        $ input: a bit string
             b:  bool,            $ auxiliary: current bit
             returns: char;       $ initialization
output = nulc; node = top;
(∀b = bitseq(n))
          if ℓ(node) eq Ω then
               output || node;  node = top;
          else
```

```
        node = if b then r(node) else  ℓ(node);
        end if;
        end ∀b;
return output;
end cseq;
end hufrouts;
```

The following declarations would be required at the point of use of the Huffman routines:

```
external (hufrouts)      $ to reference the hufrouts module
        huftables:fnct(cbase: base set(char), smap(∈cbase) real)
                                                smap (∈cbase) bool;
        cseq:fnct (bool) char;    $ decoder function.
declare chars: base set           $ these declarations are necessary
        freq:smap(∈chars) real,   $ to allow parameter passage
        code:smap(∈chars) bool                ,
        bitcode: bool;
```

In the presence of these declarations, the encoding and decoding processes appear simply as follows:

```
charcodes = huftables(chars, freq); $ set up for encoding & decoding
bitcode = [||: c = input(n)] charcodes(c);  $ encoding operation
...
decode = cseq(bitcode);             $ decoding operation.
```

Next we show a based variant of the maxflow procedure of O.P. II, pp. 123 ff. These appear as a library module called *maxflowrouts*.

```
module     maxflowrouts;
macro      r(e);  <e(2), e(1)>; endmacro r; $ edge reversal
macro      start(re); re(2)(if re(1) then 1 else 2); endmacro start;
                $ starting point of 'reversible' edge
macro      finish(re); re(2) (if re(1) then 2 else 1); endmacro finish;
declare    nodes: base set,   $ global parameter: graph nodes
           mode edge: <∈ nodes, ∈ nodes>,
           grb: base set (edge),  $ global parameter, set of edges
           mode redge: <bool, ∈ grb>, $ edge with 'reversal' flag
           grm:  map (∈ nodes) redge, $ only this is regenerated from
                                      $ parameters on call
           f,fcap: smap (∈ grb) real; $ fcap is global capacity paramet
definef    maxflow (nodes, grb, x,y,fcap)  public;
declare    x, y: nodes,   $ auxiliary nodes
           p: set (redge), $ set of edges constituting path
           e:  ∈ grb,      $ auxiliary edge
           er:  redge,     $ auxiliary, possibly reversed edges
           auxflowv:   real,  $ capacity added by path reversal flag
           tval:   bool;
           returns: smap (∈ grb) real;
grm := {<x(1), <true, x>>, x ∈ grb} + {<x(2), <false, x>>, x ∈grb};
(∀ x ∈ grb) f(x) = 0.;
(while  path(x,y) is p ne Ω)
           auxflowv = [min: er ∈ p] cap(er);
           (∀ <tval, e> ∈ p)
                f(e) = f(e) + if tval then auxflowv else - auxflowv;
           end ∀;
end while;
return f;
end maxflow;
definef cap(re): $ yields present capacity of possibly reversed edge
declare    re: redge,  $ parameter: a reversible edge.
           returns:  real;
return     if  re(1) then  fcap(re(2)) - f(re(2)) else f(re(2));
end cap;
```

```
definef    path(x,y);        $ transitive-closure path construction
declare    x,y,u,v,pt:  ∈ nodes,  $ auxiliary nodes
           re: redge,        $ auxiliary reversible edge
           new, newer, prior: set(∈ nodes),  $ auxiliary sets
           pth: set(redge), $ path to be returned
           set: subset(nodes),  $ nodes already examined
           pre: smap(∈ nodes) redge,  $ maps each node to edge
                                        leading into it.
           returns: set(redge);   $ path will be returned

new = {x}; set = new; next = nl;
(while new ne nl doing new = newer;)
           newer = nl;
           (∀ v ∈ new)
                (∀re∈grm{v}|finish(re) is u notin set and cap(re) gt 0)
                     pre (u) = re;
                     if u eq y then go to done;
                     set with u;
                     newer with u;
                end ∀re;
           end ∀v;
end while;
return Ω;       $ fallout means no path exists
done:    pth = nl; pt = y;
(while pre(pt) is re ne Ω doing pt = start(re);)
           pth with re;
end while;
return pth;
end path;
end maxflowrouts;
```

8. A Survey of Opportunities for Global Optimization.

a. Elimination of dmod testing, dmod test motion.

The use of a quantity f strongly based on a base s will in principle involve a test to make sure that s has not been subject to any dmod since the last use of f. However, this test can be elided if all paths to the use of f pass thru uses of other quantities strongly based on s after passing through the last preceeding dmod of s. This is essentially the redundancy condition for a hypothetical quantity wasd(s), which is killed by every dmod of s, and calculated by any use of a quantity strongly based on s. Moreover, a dmod test, like a calculation, can be moved outside a loop if a base used in the loop is not diminished within the loop.

If x is a pointer to an element of a base s, then the use of x will involve a test to verify that this element has not been deleted. But if every path to the use passes thru other uses of the same x not followed by any dmod of s, this test can be elided. Moreover, in this case, the index of x within s can be kept available (provided that no garbage collection can intervene between the calculation of this index and its use.)

b. Elimination of unnecessary conversion operations.

Whenever an object x is used in a way making it necessary to convert x from its current basing to some other basing ob, a logical copy xx of x, having ob as its basing, will be generated. Then if x is again required in a context requiring an ob-based representation, xx can be used in place of x. The logical equivalence of x and xx will persist along any path on which there occurs no dmod of a base appearing in the basing ob.

This remark enables replacement of x by xx to be handled by the ordinary techniques of redundant calculation elimination. Similar techniques can be used to move the conversion operation which generates xx out of a loop.

c. Choice of local vs. remote representation.

A map f represented locally can never be shared, but must be copied at every point at which its shared bit would be set. On the other hand, reference to a locally stored map will be somewhat faster than reference to a remotely stored map, since at least one level of indirection, and probably also an out-of-bounds check, can be avoided. Thus the choice of the particular representation, local or remote, to use for a given variable f (which we assume to be declared as map or smap) depends on the relative frequency of access operations and of operations which would want to set the shared bit of f. This is not something which can be discerned statically in all cases. In doubtful cases it is probably better to use the remote representation, since a single copy operation can be of very large cost, while the cost of a single remote access operation will remain bounded. However, if global analysis reveals that it is never necessary to set the shared bit of f, and that the value of f is never assigned to any variable f2 whose value might need to have its shared bit set, then local map representation of f is certainly better and should be chosen.

d. Generation of warning or fatal diagnostics.

In some cases situations likely to lead to program failure can be detected at compile time. When this is the case, either fatal disgnostics or warning diagnostics of an appropriate level of severity can be generated.

The cases which need to be caught are

i. Violation of a basing rule associated with procedure-parameter passing. (Fatal)

ii. Loader-detected incompatibility between declarations in different program modules. (Fatal)

iii. Objects which are obviously smaps declared simply as maps. (Warning)

iv. An object weakly based on s live at a reassignment of s. (Warning)

v. An object strongly based on s live at a dmod of s. (Severe warning)

In addition, the places at which the compiler has inserted conversions should be flagged.

e. Copy optimization of bases.

A base can never be shared. To avoid the formation of some unnecessary copies, the following techniques can be used. If a base is dead at the point at which it is assigned to another variable or incorporated into some more compound operation, then it need not be copied. Since it ceases to be a base when so assigned or incorporated, it then becomes subject to the normal copy rules. If a base is dead immediately after it is incorporated as an interior base of a basepak, then it can be incorporated without copying. If a basepak is dead immediately after it is unpacked, then its interior bases can be unpacked without copying. In applying this last rule, one may wish to treat the from operator and an iteration over a set of basepaks in some special way.

Note however, that a base can only be considered dead if no object based on it is live.

If all the objects based on s are known to be dead at a certain point, then assignments from s at this point can be handled normally, i.e., the shared-bit of s can be treated in the normal way. However, if this has been done, a new copy of s must be generated at any subsequent point at which an object based on s is formed.