SETL Newsletter No. 76 May 1973

Semantic-Definition Matters

J. Schwartz
and G. Jennings

Table of Contents

1. Introduction

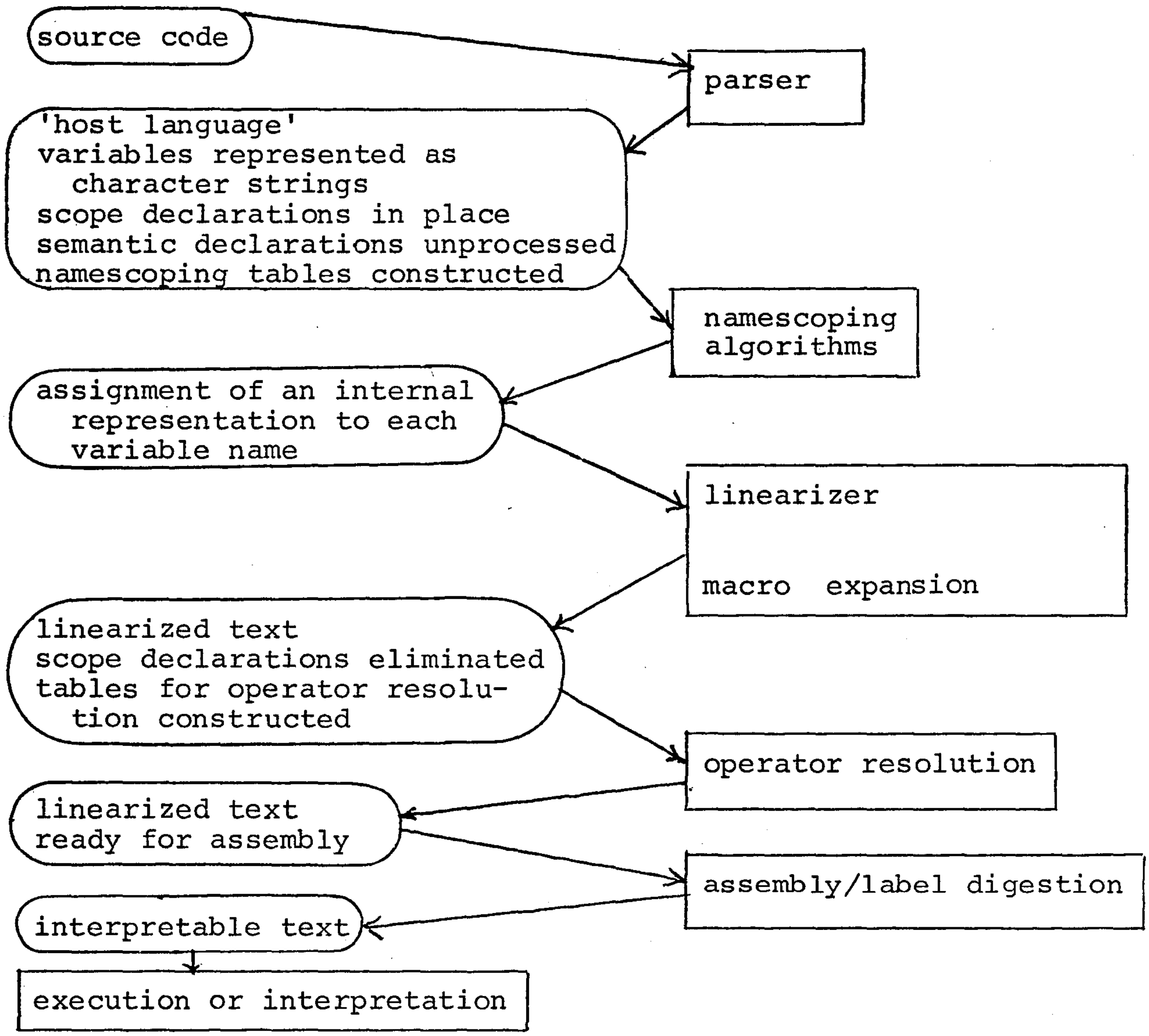
In the present newsletter, a preliminary attempt at systematic semantic definition of SETL will be made. This will be done by describing, in a deliberately abstract way, a hypothetical 'back end' for the SETL compiler. The 'back end' will consist of a set of routines concerned with name scoping, compilation of abstract recursively structured syntactic trees into interpretable serial structures, digestion of labels, and finally with the interpretation of an ultimate code form. The total package will include almost all the routines necessary to go from a simplified 'host language' form of SETL to interpretable text.

We use the phrase 'host language' to distinguish between 'host' and 'user' languages. A *host language* is a language providing a full set of semantic facilities, but with a syntax deliberately kept simple. Such languages are not intended for direct use, but rather as a basis and target for language extension. By keeping the syntax simple and modular, one confines the mass of irregularities which an attempted extension must digest. In designing a *user language*, on the other hand, one incorporates a fairly elaborate collection of syntactic facilities, hoping that these will be directly useful in a wide range of applications. SETL as currently specified is a user language. In the present newsletter, an attempt will be made to describe both an underlying host language and the manner in which this host language can support SETL (as one of several possible syntactic front ends). Note in this connection that we may eventually decide to make the host language an explicitly reachable part of our SETL implementation.

The present newsletter has significant points of contact with Hank Warren's Newsletter 53. However, the semantic notions there represented by code sequences in LITTLE and linkage conventions spelled out in LITTLE are put more abstractly in the present newsletter.

One item in the presently specified SETL, to wit the name scoping rule, is generalized in the present newsletter. We arrive at a name-scoping system which rests on the same semantic base as before, but which is considerably more general.

To make clear the overall structure of the translation process which the present newsletter describes we give a diagram summarizing its main stages.



In the present newsletter, algorithms will be given for the linearization, name resolution, operator resolution, and assembly processes appearing above. No specific parse will be described, as we wish to concentrate on semantic matters. We regard parsing as a separable user-variable part of the overall compilation process.

2. Namescoping Conventions

a. Syntax and Semantics

We shall now begin to outline a family of namescoping mechanisms, which it is hoped are sufficiently general and powerful to be convenient in the development of very large systems of programs. Of course, only experience not presently available can testify to the success (or failure) of the scheme proposed. It is hoped also that the scheme proposed will support user languages with a useful variety of user-level namescoping conventions.

We regard a namescoping system as a set of conventions which assign a unique 'resolved name' x to each 'source name' y appearing in a mass of text. The particular x to be assigned to each occurrence of y depends on the location of x within a nested, tree-like family of *scopes*.

The purpose of a namescoping system is to balance the pressures toward global use and local use of names. Unrestricted global use of names is unacceptable, since it creates a situation of 'name crowding' in which names once used become, in effect, reserved words for other program sections. Hard-to-diagnose 'name overlap' bugs tend to abound in such situations. 'Globalisation' of any subcategory of names can recreate this problem; for example, in large families of subroutines it may become difficult to avoid conflicts between subroutine names. In sufficiently large program packages, it will be desirable to give even major scope names a degree of protection.

On the other hand, a system in which names tend very strongly to be local unless explicitly declared global can tend to force one to incorporate large amounts of repetitive declaratory boilerplate into almost every protected bottom level namescope or subroutine. In a language like SETL, which aims at the compressed and natural statement of algorithms, this burden is particularly irritating.

What we therefore require is a system capable of dividing a potentially very large collection of programs into a rationally organised system of 'sublibraries', between which coherent cross-referencing is possible in a manner not requiring clumsy or elaborate locutions.

Certain important characteristics of the name resolution algorithm to be proposed are noted in the following remarks.

a. We deliberately break the conventionally very close connection between subroutine boundaries and name scopes. Name scopes enclosing several subroutines are allowed; at the same time, a single subroutine may contain several independent name scopes. A subroutine is also a namescope.

b. We regard scope boundaries as logical 'brackets' possessing a certain power to protect names within them from identification with names of the same spelling located outside. For flexibility, distinct numbered levels of bracketing are provided. We stipulate that within a scope, two variables with different names are different unless an explicit declaration is made.

c. We provide mechanisms for identifying variables which appear in the same scope and have different names, or appear in different scopes. The mechanisms for identification act recursively. Two methods are provided for the identification of variables appearing in different scopes. An explicit *alias* statement is provided to identify variables which appear in the same scope.

d. Variables can be identified by explicit remote references via the *include* statement or by being made *global* within a scope *s*, in which case they are transmitted to scopes included within *s*.

We shall prepare for a formal account of the semantic effects of our name-scoping scheme by describing a few points relating to its syntax. We begin by generalizing the notion of token. A simple token is an item recognized as integral by the lexical scanner for SETL; this may be either a special symbol, constant, simple name, underlined name, etc. A compound token or qualified token is a sequence of simple tokens connected by occurrences of the 'underbar' symbol. Thus

xl

is a simple token, while

xl_scopel_chapter3

is a qualified token. Similarly,

+ and maxop

are simple tokens;

+_scopel_chapter3

and

maxop_scopel_chapter3

are compound tokens. The successive simple tokens making up a compound token are its parts. The lexical type of a compound token is the lexical type of its first part. With the possible exception of its first part, every part of a qualified token must be a simple name.

We desire to represent a compound token by as few parts as uniquely determine it. For example *x1* and *x1_scope1* denote the same variable because *x1* is an initial part of the longer token. Similarly *x1_scope1* and *x1_scope1_item_chapter3* also designate the same variable as *x1*. In such a context the token *x1_scope2* is not allowed to appear for then *x1* would be synonymous with *x1_scope2* and *x1_scope1*.But the tokens *x1_scope1* and *x1_scope2* are different by virtue of having different names. We will provide an explicit declaration to stipulate that two (compound) tokens denote the same variable.
We demand that if t_1,t_2,t_3 appear in the same namescope

then t_1 may be the initial part of two compound tokens t_2 and t_3 only if t_2 is an initial part of t_3 or t_3 is an initial part of t_2.

The text with which we deal consists of a linear sequence of tokens, grouped into a nested family of namescopes (which for brevity we may refer to simply as scopes). A scope is opened by a header line having the form

(1) scope <(optional) level indicator> <scopename>;

for example

scope 3 main_part_of_optimizer;

Here, <scopename> designates a simple or compound name, which names the scope. The optional <level indicator>, if it occurs, has simply the form

<integer> or - <integer> .

The nonoccurrence of a level indicator is logically equivalent to the occurrence of a level indicator with a value of zero. A scope opened by the header line (1) is closed by the occurrence of a matching trailer line

(2) end <scopename>;

for example

end main_part_of_optimizer;

all the text included between (1) and the next following matching line (2) constitutes the body of the scope headed by (1). A line (2) matching each line (1) is required; the absence of a matching trailer constitutes a scoping error.

A subroutine definition

definef subrname;

is also a scope opener. This scope is named *subrname* and is closed by the end statement for the subroutine. To allow a level indicator to be associated with the subroutine an (optional) integer may separate definef and *subrname* :

definef 3 subrname; .

Several other forms of scoping error will be described in the following paragraphs. A text is acceptable to the namescope processor if it contains no scoping errors.

The text comprising a scope *ns* falls naturally into several portions:

(a) imbedded subscopes;

(b) scope-associated declaratory text (to be described in more detail shortly);

(c) other text, which we call the proper text of the namescope *ns*, which includes the executable statements, if any.

The beginning of a scope *ms* imbedded within *ns* is marked by the occurrence of a header line of the form (1); if such a header line occurs in *ns*, we require that a matching trailer line (2) be present in the body of *ns* (condition of well formed nesting). In such a case, we call *ms* a subscope of *ns*. We say that *ms* is directly imbedded within *ns* if *ms* is a subscope of *ns*, but is not a subscope of any proper subscope of *ns*. We call *ns* the parent scope of *ms*, and call *ms* an immediate descendant of *ns*. If two scopes have the same parent scope, they are said to be siblings of each other.

We require that a scope has a name different from the name of its parent and the names of its siblings. This allows us to refer to each scope in a unique manner by using a sufficiently long name string formed by concatenating the scope's immediate name with the name of its parent, its parent's parent, and so forth. Thus, for example, in a sufficiently large program library the following configuration of scopes might occur:

```
scope linear_programming;
     scope optimizer;
     x = ...
     end optimizer;
     ...                                        (3)
end linear_programming;
scope fortran_compiler;
     scope optimizer;
     ...
     end optimizer;
     ...
end fortran_compiler;
```

In the discussion which follows we shall, in order to refer unambiguously to one of the two different scopes called *optimizer* use hyperqualified names of the form 'optimizer.fortran_compiler' and 'optimizer. linear_programming'.

Similarly, two distinct variables named x, both occurring within these scopes, will be distinguished by using the hyperqualified names 'x.optimizer. fortran_compiler' and 'x.optimizer. linear_programming' . Note also that we will only insist on as deep a level of qualification as is required to guarantee uniqueness of reference; for example, we allow the same two variables x to be referenced as 'x.optimizer.fortran' and 'x.optimizer.linear' respectively. We insist on using '.' to separate scope names.

Note that hyperqualified names (punctuated by dots) belong exclusively to the 'metatheory' of namescoping. The user of our namescoping system will use qualified tokens (with underbars) exclusively. The following pages will define the manner in which 'names' (with underbars) correspond to 'items' (with dots).

Within the total mass of proper text (cf. (c) above) associated with a namescope *ns*, various tokens will occur. These are said to be *direct local tokens*. For the purposes of the following discussion, it will be convenient to designate each such occurrence of a token t by a symbol showing explicitly the nest of scopes in which t appears. For definiteness, we will write this symbol as

t. ns1. ns2. ns3. ... .nsk

where $ns_1,\ldots,ns_k$ is the nest of scopes containing t, ns_1 being the smallest such scope; ns_2 , the parent of ns_1; ns_3 , the parent of ns_2 ; etc. ns_k is an 'outermost' scope, i.e. a scope possessing no parent.

Token occurrences designated by the same hyperqualified symbol we regard *a priori* as referencing the same object. The central problem addressed by any namescoping scheme is to decide when two token occurrences not designated by the same symbol reference the same object. In the present namescoping scheme the following approach is taken. Symbols

t. ns1. ns2. ... .nsk [*]

will be called <u>items</u>. In a string of source text, each token, compound or simple, uniquely determines an item. We give rules to determine when two items represent the same variable. Within a namescope, *ns1*, the token t is sufficient to identify the item

t. ns1. ns2. ... nsk

We say that the local <u>alias</u> of the item *t.ns1.ns2. ... .nsk* is *t*. Two items, *t1.ns1.ns2. ... .nsk* and *t2.ns1.ns2. ... .nsk* which appear in the same namescopes with aliases *t1* and *t2* are identical if *t1* is an initial part of the compound token *t2*, or if *t2* is an initial part of *t1*. Items appearing in the same namescopes with alias *first_part* and *first_part_of_x* are identical because *first_part* is an initial part of *first_part_of_x*. An item with alias *first_part_of_y* would also be equal to *first_part*. We demand that the occurrence of these three tokens in a name-scope be an error. This condition is determined by noting that *first-part* is no longer an unambiguous first part of a larger compound token.

The declaratory text associated with scopes allows items whose aliases appear in different scopes to be identified. Two principal declaratory forms, an <u>include</u> declaration and a <u>global</u> declaration, are provided.

In preparation for a discussion of the semantics of <u>include</u> statements, we discuss their syntax. An <u>include</u> statement has the form

<u>include</u> <list>, <list>, ... <list>;

or, if only one <list> occurs, the simpler form

<u>include</u> <list>;

The syntax of <list> is as follows:

<list> = <aliased name> | <aliased name> (-<token>,...,<token>)
| <aliased name> (<list>,...,<list>) | <aliased name>*

<aliased name> = <token> | <token> [<token>] .

The following example will illustrate the inductive referral capability of the <u>include</u> statement.

<u>include</u> optimizer (routs3 (output (x1)));

We assume that the declaration appears in a namescope *ns* in which a scope item i_1 with alias *optimizer* is known. Within i_1 , a scope item i_2 under the name *routs3* is known. Similarly, within i_2 an item i_3 with alias *output* is available and is a scope item. Finally within i_3 an item i_4 is known with alias *x1*. The item i_4 is identified with the item whose alias in *ns* is x1_output_routs3_optimizer.

We now consider an example which uses more of the power of the include declaration.

```
include optimizer (routs1*, routs2 (-flowtrace),
                    routs3(input*,output));
include output(x1,x2);
```

Suppose that these include statements occur within a scope *ns*.

Suppose also that
the name *optimizer* is the alias of a scope item known in *ns*. An item known in *optimizer* as *routs1* is identified with the item known in *ns* under the alias *routs1_optimizer*. We use the alias of an item without specifying the namescope when no ambiguity can arise. In addition all items known in *routs1* are identified with items in *ns*. If *x* is the alias of an item in *routs1*, its alias in *ns*, is *x_routs1_optimizer*. All of the items known in *routs2* less the item known therein as *flowtrace* are identified with items in *ns*. *Input* denotes a scope item available in *routs3*. All of the items known in *input* including the scope item itself are propagated into *ns*. If *y* is the alias of an item in *input* its alias in *ns* is *y_input_routs3_optimizer*. Then, an item with alias *output_routs3_optimizer* is included. This last item will be identified with that whose alias appears in the second include statement as *output*. The identity of *x1* and *x2*, i_1 and i_2 respectively, can now be determined. i_1 is aliased in *ns* as *x1_output* and i_2, as *x2_output* as a result of this declaration.

The reader can see that the effect of these two statements is the same as the more complicated single statement:

```
include optimizer(routs1, routs2(-flowtrace),
                  routs3(input*,output(x1,x2)));
```

The identity of the item aliased as *x1* in *output* when calculated from the single expression is

x1_output_routs3_optimizer

whereas the alias produced from the two expressions is *x1_output*. Our conventions for identifying compound tokens imply that these are the same items.

An include statement may cause the identification of i_1 known in ns_1 with an item known in ns_2 under the alias $alias_2$. It is possible that $alias_2$, a token of which it is the initial part, or a token which is the initial part of $alias_2$ does not appear as a direct local token of ns_2. We allow this to facilitate the recursive application of our identification conventions. An item known in ns_2 as $alias_2$ is added to the set of variables known in ns_2. (See the discussion below of the algorithms which perform this name resolution process.)

It is possible to make an item i_1 available within *ns* under the alias a_b and another item i_2 under a_b_c and still another item i_3 under a_b_d. The rules imply that i_1 is identical to i_2 and that i_1 is identical to i_3. By transitivity of equality this should make i_2 equal to i_3 . The aliases under which i_2 and i_2 are known do not imply their uniqueness. This is an error. just as if a_b_c, a_b, and a_b_d were aliases of direct local tokens.

The reader is cautioned that it is possible for an item i_1 which is a direct local token of nsl and an item i_2 , which is a direct local token of ns_2 to be identified by including each in ns_3 with the same alias.

The above example does not illustrate the name-aliasing feature available in the syntax (and semantics) of the *include* statement. The use of this feature is shown in the following example:

```
include graphops (transitivity_routines(connectedness[cr](flagl),
        strong_connectedness[ ] (flagl[scflag], flag2));
```

Suppose that this statement occurs within a namescope *ns*, and that the scope name *graphops* (more precisely, the scope item designated in *ns* with this alias) is available within *ns*. Then the include statement shown above makes available within *ns* items, the identities of which are determined as if the brackets ('[ ]') were not present. The contents of the brackets determine the alias under which each item is known in *ns*. The first item whose alias is *flag1* in the innermost scope is aliased in *ns* as

flag1_cr_transitivity_routines_graphops.

'cr' appears in the brackets following 'connectedness' and is substituted for 'connectedness' in the algorithm to calculate the alias which was explained above. The items aliased as *flag1* and *flag2* in the scope *strong_connectedness* are aliased in *ns* as

scflag_transitivity_routines_graphops

and

flag2_transitivity_routines_graphops.

The null string in the brackets is substituted for 'strong_connectedness'. Two underbars coalesce to one. As above, these compound tokens can be abbreviated in *ns* as *scflag* and *flag2* so long as no ambiguity results.

Names can be transmitted between scopes not only by include declarations but also by global declarations. The syntax of a global declaration is

```
<global declaration> =  global <token>,...,<token>;
                      | global <token>;
                      | global <signed integer><token>,...,<token>;
                      | global <signed integer> <token>;

<signed integer>     =  <integer> | - <integer>
```

Examples are:

```
global addroutine, x1, x2, addroutine_y;
global 3 optflag;
global -1 case_flag;
```

A name *nm* available in a given scope *ns* and declared global in that scope possesses a globality level, defined as follows: if the global declaration in which *nm* appears begins with a

<signed integer> *k*, the value of k determines the globality level of *nm*, if such a signed integer is absent from the global declaration in which *nm* appears, then the globality level of *nm* is (by default) equal to the level of the scope *ns*.

Suppose, for example, that the three global declarations shown above appear in the context

```
scope 2 library1;
    global addroutine,x1,x2,addroutine_y;
    global 3 optflag;
    global -2 case_flag;
      . . .
```

Then *addroutine*, *x1*, *x2*, and *addroutine_y* have globality level 2; *optflag* has globality level 3, and *case_flag* has globality level -2.

An item *nm* designated by a name available within a scope *ns* and having a given globality level *n* becomes available within every scope *ms* directly imbedded within *ns*, provided that n is greater than or equal to the specified level of the scope *ms*. Moreover, if *nm* 'penetrates' into *ms* (i.e., becomes available via globality within *ms*), it has default globality level *n* within *ms*, and will therefore become known within all imbedded subscopes of *ms*, provided that n is greater than or equal to the level of these subscopes. This global propagation of name availability will continue through a nest of imbedded scopes until either a scope of level exceeding *n* or a scope containing no subscopes is encountered. The item *nm* known within a namescope *ns* by the alias *x1* is known under the alias *x1* within all scopes *ms* to which it is propagated through global declarations.

The propagation rules just described are basic to our name-scoping scheme. As a convenience, however, we include an additional mechanism which allows a whole group of names to be given a common designation and thus to be transmitted collectively. Suppose, for example, that within a program library a set of routines having some common overall purpose is available. Then, by giving a group name to the routines of this set, and by making the group item available in some other scope, we make all the member items of the group available in that scope.

A group statement has the form

group <token>: <list>,... <list>;

or the simpler form

group <token>: <list>;

where <list> has the syntax explained above.

Suppose that the following group statement appears in a namescope *ns*, within which we take a scope name *graphops* to be known:

```
group graph_flags: graphops(transitivity_routines
    (connectedness[cr](flag1),strong_connectedness[  ]
        (flag1[scflag], flag2));
```

This statement has, in the first place, the same force as the include statement

```
include graphops(transitivity_routines(connectedness[cr](flag1),
    strong_connectedness[  ], (flag1[scflag], flag2));
```

Moreover, the items known within *ns* under the aliases

flag1_cr_transitivity_routines_graphops,

scflag_ cr_transitivity_routines_graphops,

flag2_ cr_transitivity_routines_graphops,

become members of the group *graph_flags*.

In that group, they have the same aliases; indeed, an item always has the same alias within a group as within the scope in which the group is constituted.

If the group *graph_flags* is subsequently made available within some other scope *ms*, perhaps under an alias *atk*, and if the method of propagation does not specify explicitly that only a portion of the group is to become available, then these same objects will become available within *ms*. Their aliases within *ms* will be (in the absence of explicit re-aliasing)

flag1_connectedness_transitivity_routines_graphops

scflag_cr_transitivity_routines_graphops

flag2_scr_transitivity_routines_graphops

The following examples demonstrate additional details of the inclusion rules.

Suppose that *ms* contains the statement

<u>include</u> graph_flags;

then the items designated above as flagl_ ..., scflag_... , and flag2_ ... all become available within *ms*. Next suppose that *ms* contains the statement

<u>include</u> graph_flags(-scflag);

then only the items designated by *flag1* and *flag2* become available in *ms*. Note that 'scflag' is the first part of the alias in *graph_flags* of only one item in the group. Hence, there is no ambiguity.

Third, suppose that *ms* contains the statement

include graph_flags(scflag[scf],flag2);

Then only the items designated briefly by *scflag* and *flag2* are identified with items in *ms*. The former of these has *scf_graph_flags* as its local alias within *ms*. Finally, suppose that *ms* contains no include statement involving the name *graph_flags*, but that nevertheless the item designated by the name *graph_flags* becomes available within *ms*, perhaps in view of the appearance of *graph_flags* in a global statement within some scope in which *ms* is embedded. Then the objects designed by

```
flag1_connectedness_transitivity_routines_graphops
scflag_cr_transitivity_routines_graphops
flag2_scr_transitivity_routines_graphops
```

become available within *ms*. They are identified with items already known in *ms* in the usual way.

The above remarks concerning the include, global, and group features provided in our name-scoping scheme should make the general use and action of these features reasonably plain. Additional details will be given below; the conventions which apply in logically marginal cases can be deduced from an examination of the name-scope routines themselves, for which SETL code is given later in the present newsletter.

To identify items available within the same scope we provide the alias statement with syntax

alias *var1, var2, var3; var4, var5*;

The tokens *var1, var2,* and *var3* are the aliases of items i_1, i_2 and i_3 which are identified by virtue of this declaration. Moreover *var4* and *var5* designate items which are identified.

We regard every compound token occurring within a total mass of namescoped text as a synonym for the item which is its true designation. Note in particular that such a token will have precisely those special lexical or syntactic properties (such as the property of being a macro-name or a syntactically significant keyword) which its true designation has. We remark in this connection that if a token is a macro name at one point in a namescope *ns*, it is a macro name at every point in *ns*. This convention allows macro definitions to be placed anywhere within the namescope (or namescopes) in which they are to be applied. In addition <u>from</u> declarations and <u>kind</u> declarations (see below) may be included in macros and propagated by our namescoping conventions. <u>Include</u>, <u>global</u>, <u>group</u> and <u>alias</u> statements may not be included in macros.

The namescoping conventions described above are quite general in nature. They can be applied not only to SETL but also to other languages. The point we shall now make refers more specifically to SETL. A SETL text consists of a collection of subroutine and function bodies. All function and subroutine calls in SETL are recursive. If a routine is called before returns from all previous invocations have been executed, then all variables *local* to that routine must be stacked prior to entry. Side effects are propagated through variables which are not stacked. Items known in more than one subroutine (each of which is a namescope) will always be global and will be stacked upon entering a routine only if they are declared to be local to that routine. The syntax of the <u>local</u> declaration is provided

<u>local</u> $\text{routname}_1(\text{varname}_1,\text{varname}_2,\ldots)$,

$\text{routname}_2(\text{varname}_{k+1},\text{varname}_{k+2},\ldots)$, ...;

Here, $routname_1$, $routname_2$, ... are tokens, possibly compound, whose true designations *i* must be subroutines or functions.

Moreover, *varname*$_1$,*varname*$_2$,*etc.* are tokens, possibly compound, which must designate variables. No declaration is provided to prevent an item from being stacked because including an item trivially in a subroutine, even in one with no executable statements, prevents stacking.

We will give below algorithms for performing the identifications implied by group, include, and global statements. Subsequent to their execution, classes of equivalent items will have been formed. To prevent unintentional identifications, we impose the constraint that two or more items which represent direct local tokens of the same namescope may not be identified by group, include, or global statements. These identifications must be made by *alias* statements. Subsequent to the determination of the classes of equivalent items, each item is assigned an internal representation of the form $\langle m,n\rangle$ where m is the number of the subroutine (function) to which it is local and n is the number of the variable in that routine. A dummy routine *outrout* is created to which all variables designating subroutines and functions are assigned as are all variables known in more than one subroutine but which are not declared to be local to any routine. Each of the remaining variables is then local to exactly one routine. The k arguments of the routine, if any, are assigned the indices 1,2,...,k in the order of their appearance in the calling sequence. The remaining local variables are assigned indices from k+1.

b. Algorithms

We outline the strategy for making the identifications of items implied by global and include declarations. Consider the following lines of namescoped source text.

```
scope nsl;
  include ns2[a](ns3[b](c, d*, e[newname]));
  global globalvar;
  c = nℓ;
end nsl;
```

An initial pass of the source text will recognize the items

globalvar, c_b_a, d_b_a, newname_b_a

as the direct local tokens of *ns1*. The last three variables are recognized by a scan of the include statement. The variable denoted by *c_b_a* is the same variable as that denoted by *c* which appears in the one line of executable text in *ns1* (see above for a discussion of the rules for the identification of items designated by compound tokens). We assume further that the scope *ns1* appears in a nest of scopes *ns2, ns3, ...,* and *nsk* where *nsk* has no parent scope. We represent the scope *ns1* internally for the purposes of the algorithms which follow as the tuple

<ns1, ns2, ..., nsk> .

This tuple uniquely identifies this namescope. The four variables designated above are items which are initially known in *ns1*. The token *globalvar* is said to be the *local name* or *alias* of the item *<globalvar, ns1, ns2,...,nsk>* in the scope *<ns1,ns2,...,nsk>*. Similarly, each of *c_b_a, d_b_a* and *newname_b_a* is the *local name* (*alias*) of an item in the scope *<ns1,ns2,...,nsk>*. These four items are said to be initially known in the scope <*ns1,ns2,...,nsk*>. In addition to these four items, the parent scope, the scope itself, the sibling scopes, and all immediate descendant scopes of the scope *ns1* are known in <*ns1,ns2,...,nsk*>. For example, the parent scope <*ns2,ns3,...,nsk*> is known in <*ns1,ns2,...,nsk*>, under the *local name ns2*.

All direct local tokens and scopes adjacent to a scope are given names on a formal basis during an initial pass of the source text. We do not give the code for this process in this newsletter. The *local name* of an item in the scope in which it appears is the first component of the tuple which is its name as an item. Moreover, for an item which is not a scope, the (SETL) tail of the tuple is the name of the scope in which the alias is a direct local token.

In the <u>include</u> statement of the current example, the string d^* implies that all items known in the scope *d* are to be included in *ns*. Suppose that *ghj* is the local name of an item which appears in the scope *d*. That item is, by virtue of this <u>include</u> statement, to be identified with an item known in *ns1* with *local name ghj_d_b_a*. There is no item which is known initially in <*ns1,ns2,...,nsk*> with this local name. The algorithm we give will create an item known in *ns1* with the local name *ghj_d_b_a* upon determination of such an impasse. This latter item will then be identified with the item with *local name ghj* in the scope *d*.

This inclusion, vacuously, of items into *ns1* facilitates recursive application of the <u>include</u> and <u>global</u> declarations.

We assume that the initial pass of the source text creates the set *knownby* which contains pairs of the form *<scopeitem, varitem>* where *varitem* is an item known in *scopeitem*. The creation of the item with alias *ghj_d_b_a* results in the pair *<<ns1,ns2,...,nsk>,<ghj_d_b_a, ns1,...,nsk>>* being introduced into *knownby*.

We now describe the mechanisms for retaining the information that items known initially in different scopes have been identified. As 'identity' is an equivalence relation, i.e. a = b and b = c implies a = c, it suffices to provide a vehicle for determining a canonical representative of the class to which an item belongs. The set *ident* evaluated at i_0 is the canonical representative of i_0. Also *equivset{rep}* is the set of items equivalent to the canonical representative *rep*. *Equivset* is the relation inverse to *ident*. We retain each set for economy of execution. We manipulate these sets through two routines *ultdesig* and *getequivitem*. In this way we avoid initializing each set to {<x,x>, x an item known in source text}.

Group items are distinguished by being members of the set *isgroup*. If *gpitem* is a group item and *i* is an item which is a member of *gpitem*, then *<gpitem,i>* is an element of *knownby*.

An _include_ statement as it appears in a line of source text implies the identification of one or more pairs of items. The method of determination of the identity of the elements of each item in each pair should be clear from the discussion above of the semantics of the _include_ statement. We now give an example of a complication with which algorithms for processing _include_ statements must cope.

Consider the nested scopes:

```
scope ns1;
  scope x;
    w = ...;
  end x;
end ns1;
scope ns2;

scope z;
  include ns2(y(w));                          (4)
    w = ...;
end z;
  include ns1(x[y]);                          (5)
end ns2;
```

By virtue of (4), the item <w_y_ns2,z,ns2> is to be identified with another item, known as *w* in a scope with alias *y* in the scope *ns2*. The item <y,ns2,...> is not known to be a scope item until the _include_ statement (5) which identifies it with the scope <x,ns1> is processed. This shows that _include_ statements must be considered in an order which need not be the order in which they appear in the source text.

In addition to the sets which we have discussed above, we will require a coded representation of the include statements. We assume that the first pass of the source text associates a set of tuples, called *include{ns}*, with every namescope *ns*. Each tuple in *include{ns}* is of the form

$$\langle ans_1, ans_2, ans_k, aliastring, \begin{matrix}\text{'all'}\\ \text{'allbut'}\\ \text{'only'}\end{matrix}, \{\ldots\}\rangle$$

ans_j is the alias in ans_{j-1} of a scope item. One of the phrases *'all'*, *'allbut'*, and *'only'* appears. This phrase is named *keywd*. The set which is the last entry of the tuple is not present if *keywd* is 'all'. The scope item *ns* is the namescope in which the include statement from which *inctuple* was derived originally appeared. This component which we generically call *inctuple* causes the identification of one or more pairs of items. One member of each pair is known in *ns* which we call *targetscope*. The other is known in the scope *source*. The scope *source* is identified in the following manner.

Starting with *source = targetscope*, the item $\tilde{i}_1$ whose local name is ans_1 is identified. The canonical representative, i_1, of the equivalence class to which $\tilde{i}_1$ belongs is determined. If i_1 is a scope item, then *source* is set equal to i_1 and the second component of *inctuple* is considered in the same way. This process is repeated until all the items designated by the components preceding *aliastring* are identified or until the item i_k is not a scope item. In the former case, identifications between items in the scope *targetscope* and items in the scope *source* are made. We will make further remarks on this process below. In the latter case, *inctuple* corresponds either to a namescoping error or the processing of additional *inctuples* must uncover a new scope item in *source* with alias ans_k. When this occurs, further decoding of *inctuple* is attempted. A partially condensed form of *inctuple* which reflects the successful part of the decoding is saved and tagged with *source* to facilitate the continuation of the decoding process. If *keywd* of *inctuple* is '*all*' or '*allbut*', then subsequent introduction of items into *source* which are not known at the time of decoding of *inctuple* may require reprocessing *inctuple* so as to propagate the newly discovered items into *targetscope*. A condensed form of *inctuple* is retained in the set *decoded* for this purpose.

The members of the set which is the last component of *inctuple* determine the pairs of items to be identified. If *keywd* is '*all*' then each item known in *source* is identified with an item in *targetscope*. If *i* is an item with alias *a* known in *source*, then the item known in *targetscope* with alias *a_aliastring* is identified with *i*. Note that *aliastring* may be a compound token.

We have now explained the function of each component of *inctuple*. We give an example to indicate how include statements are reduced to *inctuples*. Suppose that the two include statements

```
include optimizer(routs1,routs2(-flowtrace),
            routs3(input*, output(x1,x2)));
include graphops(transitivity_routines(connectedness[cr](flag1),
   strong_connectedness[scr](flag1[scflag], flag2));
```

occur within a scope *x*. Then *includes(x)* will contain (at least) the following set.

```
{<'optimizer','optimizer','only',{<'routs1'>,<'routs2'>,<'routs3'>}> ,
 <'optimizer','routs2','routs2_optimizer','allbut',{'flowtrace'}>
 <'optimizer',routs3','routs3_optimizer', 'all' >
 <'optimizer','routs3','output',
            'output_routs3_optimizer','only',{<'x1'>,<'x2'>}>

 <'graphops','transitivity_routine','connectedness',
     'cr_transitivity_routine_graphops,'only',{'flag1'}>,
 <'graphops','transitivity_routines','strong_connectedness',
  'scr_transitivity_routines_graphops','only',
   {<'flag1','scflag'>,<'flag2'>}>}
```

Suppose that *item1* and *item2* are canonical representatives and are to be identified. As scope items, group items, and macros are definite semantic constructs, only one of *item1* and *item2* may be either a scope, a group item, or a macro. The contrary case is a namescoping error. We suppose,without loss, that *item1* is a scope item, a group item, or a macro, then, we set *item1* to be the representative of the equivalence class of *item2*. There is further action if *item1* is a scope item or a group item. If *item1* is a scope item, then every item equivalent to *item2* is a newly uncovered scope item. We then attempt to decode partially decoded *inctuples* whose decoding terminated in the scope in which each of these items is known. In the case that *item1* is a group item, all elements of the equivalence class of *item2* become group items. The members of the group *item1* must then be identified with items known in the scopes in which the members of the equivalence class of *item2* are known. *Ident(item2)* is then set to *item1*. *Equivset(item1)* is augmented to include *equivset(item2)* and then pairs corresponding to the latter set are deleted from *equivset*.

The algorithm for processing global declarations is a straightforward implementation of the semantics of global statements. The reader should be able to comprehend the code for this part of the process. We now summarize the functions of the various sets and routines required in the process.

iscope - set of all scope items

isgroup - set of all group items
ismacro - set of all macro items
knownby - <x,y> is in *knownby*, if and only if x is a scope item, y is an item known in the scope y (note x is known in itself) or, x is a group item and y is a member of x

dscopes{scope} - set of all immediate descendant scopes of *scope*

level(scope) - globality level of scope (if none was specified 0 is returned)

globlev(var,scope)- the globality level of *var* in *scope*. If no explicit declaration was made, this number is set to *level(scope)*

includes - elements are tuples which result from include declarations, see above for details

decode - subroutine which determines *source* scope referred to by a member of *includes* *decoded* - set contains condensed information about successfully decoded elements of *includes* whose *keywd* is 'all' or 'allbut'

propinclude - subroutine which performs the identifications implied by decoded members of *includes*

ident(item) - canonical representative if not Ω of the class of equivalent items to which *item* belongs, otherwise representative of *item* is *item*.

ultdesign(item) - coded routines which calculates the canonical representative of *item*

equivset - <*rep*,x> is in *equivset* if and only if *rep* is the canonical representative of the class of items to which x and *rep* belong. Pairs of the form <*rep*,*rep*> are omitted.

getequivitem(rep) - coded function which calculates the set of items equivalent to *rep*.

equate(item1,item2) - subroutine which makes change in *ident* and *equiv* so that *item1* and *item2* are identified

locname <u>designatesin</u> *scope* - coded function which determines the item known in *scope* the first part of whose (compound) local name is *locname* - if none exists an item is created, and all decoded inctuples with *keywd* 'all' or 'allbut' and with *source* equal to *scope* are reprocessed.

With these remarks, the reader should find comprehensible the following code:

```
/* first process all global statements */
work = copy(globlev);
(while work ne nℓ)
   globitem from work; <scope,item,lvlitem> = globitem;
   /* propagate item to all immediate descendants of scope
     to which item  penetrates */
   (∀desc ∈ dscope{scope})
        if lvlitem ge level(desc)
           then /* item penetrates desc */
              equate(item,(hd item)designatesin desc is newitem);
              newlevel = if globlev(newitem,desc) is lvlnewitem ne Ω
                 then lvlitem max lvlnewitem  else lvlitem;
              <desc, newitem, newlevel> in work;
        end if;
   end ∀desc;
end while;
/* all global statements processed  proceed to include statements*/
undecoded = nℓ;  decoded = nℓ;
(∀stmt ∈ includes)
newhome = source = hd stmt;
decode(<source, newhome, stmt(2:)>);
end ∀stmt;
```

```
/* if fall out of loop with undecoded statements,
     then issue diagnostics */
if undecoded ne nℓ
 then print 'following include statements not interpretable',
      {x(z), x ∈ undecoded};
end if;
/* check if have identified two different items known
     originally in the same scope */
(∀ scope ∈ iscope)
  if #(knownby{scope} is varscope) ne #ultdesig[varscope] then
     print 'have identified via global and include declarations
     two or more items initially known in', scope, 'list of
     all variables initially known in this scope together
     with the canonical representative of the equivalence
     class follows', {<x, ultdesig(x)>, x ∈ varscope};
  end if;
end ∀ scope;
/* finished global and include declarations - code to
     process alias declarations  belongs here - it is omitted */
```

We now give code for the auxiliary routines.

```
define decode(inctuple);
 <source, newhome,stmt>=inctuple;
keyloc = if stmt(#stmt) eq 'all'
     then #stmt else #stmt-1;
keywd = stmt(keyloc);
/* begin decoding of stmt */
```

```
    (1 ≤ ∀j < keyloc-1)
       itemdesignated = ultdesig(hd stmt(j) designatesin source);
       if itemdesignated n ∈ iscope
            then /* decoding failed */
            <source,newhome,stmt(j:)> in undecoded;
            return;
       end if;
    end ∀j;
    /* decoding successful */
    source = itemdesignated;
    finalpart = stmt(keyloc-1:);
    /* decoding successful - condensed form of inctuple in decoded
       if keywd ne 'only' */
    newstmt = <source,newhome> + finalpart;
    if keywd ne 'only'  then newstmt in decoded;;


    propinclude(newstmt, knownby{source});
    end decode;

    define propinclude(stmt,knownbysource);
    <tgtscope, source, aliastring, keywd, -> = stmt;
    set = keywd eq 'all' then nℓ else stmt(5);
    separator = if aliastring ne nulc then '-' else nulc;
    if keywd eq 'only'
     then (∀x ∈ set)
          atgtitem= x(#x) + separator + aliastring;
```

```
    equate(x(1) designatesin source, atgtitem designatesin tgtscope);
    end ∀x;
else      /* keywd is 'all' or 'allbut' */
    excluded = [set] designatesin source;

    (∀item ∈ (knownbysource - excluded))
    atgitem = hd item + separator + aliastring;
    equate(atgtitem designatesin tgtscope, item);
    end ∀item;
    end if keywd;
    end propinclude;
```

```
define equate(item1,item2)
/* makes additions to equivset                    so that
     item1 and item2 are identified                 */

ultitem1  =      ultdesig(item1 );
ultitem2 =       ultdesig(item2 );
if ultitem1 eq ultitem2 then return;; /* else */
if ultitem1 ε (isgroup + iscope + ismacro) and
  (ultitem2 ε (isgroup + iscope + ismacro)) is twospecial
then print 'attempt to identify', item1, 'and', item2,
     'each is either a scope or group item'; return;;
/* else */
if twospecial then <ultitem1, ultitem2> = <ultitem2,ultitem1>;;
   itemsequiv = getequivitems(ultitem2);
if ultitem1 ∈ iscope
   then  /* have uncovered new scope items - all items
     equivalent to ultitem2 are scope items */

   (∀ item ∈  itemsequiv)
      homescope = tℓ item;
      /* decode all inctuples which failed in homescope */
      (∀ minctuple ∈ undecoded {homescope})
      decode(<homescope> + minctuple);
      end ∀minctuple;
   end ∀item;
end if ultitem1;  /* else */
```

```
if ultiteml ∈ isgroup
   then /* all items equivalent to ultiteml are group items */

   (∀ item ∈ itemsequiv)
        homescope = tℓ item;
        propinclude(<ultiteml, homescope, nulc, 'all'>);
   end ∀item;
end if ultiteml;

/* identify ultiteml and ultitem2 */

equivset = equivset + {<ultiteml,x>, x∈  itemsequiv};
equivset = equivset lesf ultitem2;
(∀x ∈   itemsequiv)
   ident(x) = ultiteml
end ∀x;

return;
end equate;

definef ultdesig(item);
initially ident = nℓ;;
 return if ident(item) is ult eq Ω then item else ult;
end ultdesig;

definef getequivitems(item);
initially equivitems = nℓ;;
return equivset{item} + {item};
end getequivitems;
```

```
definef locname designatesin scope;
/* compound token is of the form str1_str2_str3 */
candidates = {x, x ε knownby{scope}|match(x,locname)};
if #candidates gt 1
 then print 'more than one item known in', scope,
     'with local name', locname, return Ω;
end if;
/* else */
if candidates eq nℓ
 then /* create an item with local name equal locname */
(<locname> + scope) is newitem in candidates;
<scope, newitem> in knownby;
/* reprocess decoded inctuple with keywd 'all' or 'allbut'
     and equal to scope  so as to propagate newitem  */
(∀inctuple ε decoded {scope})
 propinclude(<scope> + inctuple, {newitem});
end ∀inctuple;
end if candidates;
return ∍candidates;
end designatesin;


definef match(name1,name2);
it1 = copy(name1); it2 = copy(name2);
if(#it1 ge #it2)
 then <it2,it1> = <it1,it2>;;
return (it2(1:#item2) eq it1) and (it2(#item1 +1) eq '-'
     or #item1 eq #item2);
end match;
```

c. Internal representations of variables

We now turn to assigning an internal representation to each variable. The processing of *include* and *global* declarations creates the sets *equivset* and *ident*. Items known in two different scopes have been identified and will be assigned to a dummy subroutine *outrout* in the absence of an *owns* declaration. All subroutines will be considered to be owned by this routine. The identifications implied by *alias* statements are then made and the remaining variables are assigned a representation in the form <m,n>, i.e. the n^{th} variable of the m^{th} subroutine. The identification of two or more arguments of a subroutine is a namescoping error as is identifying an argument to a variable known in another scope. Arguments to subroutines are dummy variables which should not also be global variables. If there are *narg* arguments in the n^{th} subroutine, these variables are assigned internal representations

<n,1>, <n,2>, ..., <n,*nrarg*>

The remaining variables local to (owned by) this subroutine are assigned representations

<n,*nrarg* +1>, <n, *nrarg* +2>, ...

We have adopted the convention that subroutine headers are de-facto scope openers. The associated end statement also terminates the range of the scope. Executable code must appear in a scope which is also a subroutine or is contained in a subroutine. Macros and namescoping declarations are the only statements allowed in namescopes which properly include subroutines.

The identification as a result of *include* and *global* declaration of two scopes is not allowed. *A fortiori*, two different subroutines may not be identified.
The sets we require are created during an initial pass of the source code. We ignore the problem of their creation and list the structures required together with a brief explanation of their structure.

ultdesig(item) - if *item* is equivalent to an item in another scope then is equal canonical representative of *item*, otherwise Ω.

equivset(item) - the set of atoms, other than *item*, equivalent to *item*

subroutines - subset of *iscope* consisting of all subroutines.

arguments(subr) - tuple which contains the arguments of subroutine *subr*.

aliastmt(ns) - set containing members of form $\{var_1, var_2, \ldots, var_k\}$. $Var_1, var_2, \ldots, var_k$ are aliases in *ns* of the same variable

outrout - name of dummy subroutine to which all global variables (not declared to be owned by a routine) and all subroutines are local.

ownstmt - set containing pairs *<item,subroutine>* which result from *item* being declared to be owned by *subroutine*.

internrep - elements are of the form *<item,pairint>* where *pairint* is a pair <m,n>, the internal representation of *alias* in *ns*.

The auxiliary routines include a coded function

localvar(subr) - set containing all items known in *subr* or a descendant scope which are neither scope items, group items, global variables owned by *outrout* or another routine or arguments of *subr*.

We require from the earlier processes *dscopes*.

dscopes{ns} - set of scopes which are immediate descendants of *ns*.

homescope(item) - scope in which item is initially known.

homesubr - is a coded function which calculates the subroutine in which an item appeared.

We now give the algorithms for the assignment of internal representatives to each item.

```
/* first the subroutines */
nrvar = 1; internrep = nℓ;  outrout = 0;
(∀subr ∈ subroutines doing nrvar = nrvar+1;)
internrep(<subr>)      = <outrout,nrvar>;
end ∀subr;
```

```
/* calculate all global variables not declared
     to be owned by a subroutine */

(∀item∈{x, x∈hd[tℓ[ultdesig]] is globalitem|notown(x)} is globalitem)

nrvar= nrvar + 1;
setid(item,<outrout,nrvar>);

end ∀item;

/* process all alias declarations - make further

     identifications */

(∀ns ∈ iscope, ∀astmt ∈ aliastmt{ns})

     x from astmt;  ownstmt(x) = ns;

     (∀var ∈ astmt)

          equate(var,x);

     end ∀var;

end ∀astmt;  end ∀ns;

/* assign internal representatives to all

     remaining variables */

(∀subr ∈ subroutines doing varnr = 1;)

subrno = internrep(<subr>)(2);

(1 ≤ ∀i ≤ #arguments(subr) doing varnr = varnr+1;)

     eqitems = getequivitems(arguments(i));

if(homesubr[eqitems] ne {subr} or internrep[eqitems] ne nℓ)

     then print dec  i-th argument of , subr, 'has been

       identified with an item known in another subroutine

     or to another argument of this subroutine';

     continue ∀i;

end if;

setid(arguments(i),<subrno,varnr>);

end ∀i;
```

```
/* assign representation to remaining local variables */
(∀item ∈ localvar(subr) doing nrvar = nrvar+1;)
setid(item,<subrno,varnr>);
end ∀item;
end ∀subr;
```

We now code the auxiliary routines.

```
definef notown(item);
/* determine if item or a member of its equivalence class
     has been declared to be owned by a subroutine
     (not outrout) */
return ownstmt[getequivitems(item)] eq nℓ;
end return;

define  setid(item,id);
(∀x ∈ getequivitems(ultdesig(item)) is eqitem)
     internrep(<homescope(x),x>) = id;
end ∀x;
getequivitems(ultdesig(item)) = Ω;
ultdesig[eqitem] = nℓ;
end setid;

definef localvar(subr);
/* return set of canonical representatives of variables known
     in subr- i.e. items not scopes, groups, macros
     or arguments - not owned by other subroutines */
variables = nℓ;  scopes = {subr};
```

```
(while scopes ne nℓ doing scopes = descopes[scopes];)
newvar = knownby[scopes]
newvar = newvar = (iscope + isgroup + ismacro);
newvar less {x, x∈newvar | internrep<homescope(x),x> ne Ω);
/* delete variables  global to another routine */
newvar  less {x, x∈newvar   |not (ownstmt[getequivitems(x)]
                                                       }
                                                  le {subr}));
return newvar;
end localvar;
```

We have made no assignment of internal names for macros. A macro will be recognized during linearization of tree like source text (see below) when no internal representation is defined for what appears to be a subroutine. Macro expansion will occur at that point.

3. Programmer definable object types

We now discuss a system of programmer definable object types which allows operators to be applied to objects in a type dependent manner. This enhances the expressive power and extensibility of the language in a very useful way. The main features of the scheme are that it is static and declaratory. Types are assigned to variables by declaration. Type information is used not at run time which might necessitate a great deal of dynamic type checking, but to control the compilation process.

To make plain the overall nature and intended use of the proposed scheme we shall first set it forth in a particular syntactic realization.

The notion basic to our scheme is that of an object type or kind. Such a kind is merely a token (simple or compound), which, because of the manner in which it appears in one of the declarations to be described below, can be recognized as denoting, or being the name of, an object kind. This convention allows the programmer to introduce any number of differently named kinds of objects. As various object kinds are introduced, the variable names appearing in a SETL program will be declared to be of these kinds. The declared kinds of the variables appearing in an expression will then be used to control the manner in which the expression is compiled. The kind declaration has the syntactic form

<u>kind</u> $\text{kindname}_1(\text{varname}_1, \text{varname}_2, \ldots)$, $\text{kindname}_2(\text{varname}_n, \ldots), \ldots;$

Here, $\mathit{kindname}_1$, $\mathit{kindname}_2$, etc. are tokens which, by virtue of their appearance in the declaration shown above, are the names of variable kinds (briefly: kind names);

while *varname*$_1$,...,*varname*$_k$, ... are variable-designating tokens. At most one kind can be declared for a variable name. If no kind is declared, the variable is taken to be of default kind. This is the kind which is named in a declaration

default kindname;

At most one such declaration can appear in a given name scope *ns*. If no such declaration appears *ns* will be assigned the same default kind as its parent scope. If *ns* has no parent, *setlstdtype* is the default.

The declared kinds of variables are used to control the manner in which expressions, subroutine calls and iterators of the '∀' type are compiled. The manner in which this is done is clear from the form of declaration which specifies the manner in which binary infix operators are compiled. This has the form

from <kindname$_1$> <operator-symbol> <kindname$_2$> get <kindname$_3$> using <routinename>;

Examples are

from aplobj + aplobj get aplobj using aplplus;

and

from matrix * vector get vector using matvectprod;

The significance of the from declaration is as follows: whenever two objects x_1 and x_2, always of known kind, are to be combined by an infix operator op, reference is made to the full collection of from declarations available in the given namescope. If one declaration is applicable, i.e., if the operator-symbol occurring in the declaration matches op, and the object kinds occurring in the declaration match the known kinds of x_1 and x_2 respectively, then the result of the operation is taken to have the kind specified by the third *kindname* appearing in the from declaration. Moreover, the operation is compiled as a call to the (two-argument) function appearing in the from declaration.

Consider, for example, the code (at the 'basic interpreter' level see below) which would be compiled from the statement

```
x = (a max b) * c + d;
```

```
call(sysmax,a,b,t1);
call(sysprod,t1,c,t2);
call(syssum,t2,d,x);
```

where we assume *sysmax*, *sysprod*, and *syssum* to denote the standard 'library' procedures which correspond to the ordinary SETL operations max, *, and + respectively. If the declarations

```
default matrix;
kind vector(c,d);
from matrix max matrix get matrix using matmax;
from matrix * vector get vector using matvectprod;
from vector + vector get vector using vectsum;
```

are active within the context in which the statement appears, the expression seen on the right-hand side of the assignment statement displayed above would be compiled as follows.

```
call(matmax, a, b, t1)
call(matvectprod, t1, c, t2);
call(vectsum, t2, d, x);
```

We allow the more general form

```
from kindname1 kindop kindname get ...
```

where kindop is a kind name which designates a class of operators.

The above remarks should make plain the general force of the kind, from and default declarations. We now go on to describe useful variants of these statements, and also certain other related declarations needed to give a system of 'object types' adequate flexibility. Note first that we will in some cases wish to use the standard SETL operations to combine objects of particular kinds, but will nevertheless wish to know the kind of object which results. For this purpose, we provide a variant of the from statement, having the abbreviated form

<u>from</u> <kindname$_1$><operator-symbol><kindname$_2$>

get <kindname$_3$>;

An example of this construction might be

<u>from</u> stringset + stringset get stringset;

this would be useful in a situation in which we wish to distinguish sets of kind *stringset* from other sets, even though the ordinary SETL union operation is used to form the sum of two variables of kind *stringset*.

If a token *t* naming a variable appears in one of our declarations where a kind name is expected, it is understood that the token name is also a kind name, and that the variable is of the kind having this name. Thus, for example, if *mainset* occurs as a variable name in some program together with the declaration

<u>from</u> vector $\in$ mainset get bool using specialtest;

it is understood that *mainset* is also a kind name, and that the variable *mainset* is of kind *mainset*.

We must of course deal not only with infix binary functions of two variables, but with functions of several variables, and even in a few cases with functions of an indefinite number of variables. Here, our declaratory conventions are as follows. We write

<u>from</u> <kindname$_0$>(<kindname$_1$>,...,<kindname$_k$>) get

<kindname> using <routinename>;

and

<u>from</u> <kindname$_0$>{<kindname$_1$>,...,<kindname$_k$>} get

<kindname> using <routinename>;

and

<u>from</u> <kindname$_0$>[<kindname$_1$>,...,<kindname$_k$>] get

<kindname> using <routinename>;

These forms allow us to create kind-dependent usages of any of the three basic application forms provided in the SETL syntax.

These forms, as just described, presume a fixed number of arguments. Similar declaration forms, whose details will be apparent to the reader, must also be available for use in connection with prefixed monadic operators. 'Short' declaration forms, in which the 'using <routinename>' part of the declaration is dropped, are also allowed, and have the significance already explained.

Our base level interpreter conventions (see below) allow polyargument primitives (though not nonprimitive calls involving an indefinite number of variables). Moreover, SETL provides the 'tuple-forming' polyargument primitive

$$<x_1, x_2, \ldots, x_k>$$

which can be used to reduce most other polyargument situations to situations in which only a fixed number of arguments will occur. We make it possible to use the present declaratory scheme in poly-argument situations by providing the <u>from</u> declaration in the generalized form

(1) <u>from</u> <kindname$_0$>(<kindname$_1$>,...,<kindname$_k$> -) get

<kindname> using <routinename>;

The semantics of this declaration are as follows. If an item having the syntactic form

(2) $$i_0(i_1,\ldots,i_k,i_{k+1},\ldots,i_n)$$

appears in an expression, and if i_j is of the kind designated by *kindname*$_j$ for j = 1,...,k, then the declaration shown above is <u>relevant</u>. In this case, the items $i_{k+1},\ldots,i_n$ appearing in (2) are classified as 'extra arguments', and a call of the form

(3) call(routinename, $i_1,\ldots,i_k$, $<i_{k+1},\ldots,i_n>$, t>

is generated at the basic interpreter level, t being a 'compiler temporary' storing the result of the function call (2), and $<i_{k+1},\ldots,i_n>$ being the n-k tuple formed from the values of the extra arguments.

A declaration like (1) is also provided in the forms

from <kindname$_0$>{<kindname$_1$>,...,<kindname$_k$>-} get
<kindname> using <routinename>;

from <kindname$_0$>[<kindname$_1$>,...,<kindname$_k$>-] get
<kindname> using <routinename>;

and also

from <<kindname$_1$>,...,<kindname$_k$>-> get
<kindname> using <routinename>;

from {<kindname$_1$>,...,<kindname$_k$>-} get
<kindname> using <routinename>;

from [<kindname$_1$>,...,<kindname$_k$>-] get
<kindname> using <routinename>;

These last three declaration forms allow the various kinds of 'brackets' provided in SETL to be used in a manner depending on the kinds of objects which appear within them.

Note that declarations of the type we are now describing can appear within macros, which can be carried from one namescope to another using the mechanisms of token transmission.

This allows a whole group of declarations to be invoked by including a single token at an appropriate point in a text, thereby allowing one in effect to 'name' standard systems of conventions which are to apply during the compilation of particular code passages. Namescopes can be used as boundaries at which the system of conventions change.

The family of declarations introduced above is valuable not only for the extension of language syntax but also in debugging. If, in compiling an expression, one comes upon an operation applied to variables of kinds for which no <u>from</u> statement has been supplied, a diagnostic message can be issued. Thus our declarations serve to attach a network of compile-time consistency checks to a SETL text. To ensure that this network is free of loopholes, we shall insist not only that the kinds of objects appearing in function calls be validated (by providing appropriate from statements) but also that the kinds of objects appearing in subroutine calls be validated. To allow this, we provide an additional declaration of the form

<u>allow</u> <kindname$_0$>(<kindname$_1$>,...,<kindname$_k$>);

which validates a subroutine call with the obvious pattern of kinds of arguments.

We take it that the conditional expression appearing in an *if* or *while* statement must always have a value of the standard SETL type <u>bool</u>; given this, and given the conventions concerning variable kinds in sinister calls which are explained below, we can check any program systematically for consistency of the kinds of objects which appear in it. Note however that by making a statement

<u>default</u> setℓstdtype;

active in a context in which no explicit declarations concerning variable kinds appear, we disable this consistency check mechanism reducing its action simply to a verification of syntactic wellformedness.

'Iteration over all subparts' is a concept potentially applicable to, and useful in connection with, compound objects of all sorts. To allow this notion to be applied to objects in a kind-dependent way, we introduce a declaration which specifies three basic routines, one to set up the first subpart 'address' of a compound object, the second to advance this 'address' from one subpart to the next (returning Ω if advance is impossible), the third to calculate the actual subpart corresponding to a given subpart address. More specifically, call these three routines *first*, *next*, and *actelt* respectively. Then we take the iteration

$$(\forall x \in a)\text{<body> end } \forall;$$

to expand as

```
xaddr = first(a);
(while xaddr ne Ω doing xaddr = next(xaddr,a);)
     x = actelt(xaddr); <body> end while;
```

A declaration appropriate for this purpose must specify the three routines *first*, *next*, and *actelt*, and must also describe the kind of subpart which a compound object has. For this purpose, we propose the following syntax:

<u>forit</u> <kindname_0> ∈ <kindname_1> use

<first routine name>, <next routine name>,

<actelt routine name>;

In this declaration, <kindname_0> is the name of a compound object type, and <kindname_1> names the kind of parts which an object of this type has. The 'first routine name', 'next routine name', and 'actelt routine' have the significance already explained.

Occasionally, though probably not often, one will wish to use subroutines or functions which can return a value of one of several kinds; more generally, variables whose values are of a kind not precisely known may appear in a program. We propose to handle this situation as follows. A kind name designating whatever ambiguity of kind exists for a given variable will be invented, and a variable whose kind is syntactically ambiguous will be declared to be of this kind. For example, one might find oneself writing

<u>kind</u> tree_or_graph (x);

Ultimately, and probably quite swiftly, a variable ambiguous in kind will be tested, and its kind determined as a necessary preliminary to further processing. Normally this will imply conditional transfer to one of several points; transfer to a particular point will mean that an initially existing ambiguity of kind has been resolved in a particular way; at each such point, code appropriate to the processing of the formerly ambiguous variable, now of known kind, will be found. To handle all this within our system of declarations, we propose the following scheme. Several separate names, all designating the same variable, will be invented. Each name will be declared to be a particular kind. More precisely, one such variable name will be declared to have the 'ambiguous' kind alluded to above, while the other variables named will be declared to have the various separate kinds whose confounding creates this ambiguity. Then all the variable names which have been used will be declared to be aliases for each other, i.e., to refer to the same object. In this way, our changing state of knowledge concerning an object is reflected syntactically by the varying names we give it. The form proposed above for the necessary declaration is

$$\underline{\text{alias}}\ \langle\text{token}_0\rangle\ \langle\text{token}_1\rangle,\ldots,\langle\text{token}_k\rangle\ ;$$

Often (and especially in situations like the one which has just been described) most of the operations performed on objects of two different kinds will be identical (i.e., will be performed by the same basic-interpreter-level subroutine or function) even though a few particular operations should be performed by different kind-dependent routines. The notion we propose as basic to the treatment of the situation which then arises is that of the <u>reversion</u> of kinds. A particular variable kind k_1 is said to revert to another kind k_2 if, in a significant family of cases, operations may be performed for an object of kind k_1 by using the routines already supplied to handle these same operations for objects of kind k_2. We declare the kinds to which an object of given kind may in this sense 'revert' by writing

<u>revert</u> <kindname$_1$>(<kindname$_2$>,...,<kindname$_j$>),
<kindname$_{j+1}$>(<kindname$_{j+2}$>,...,<kindname$_{j+m}$>), ...;

Let $k_1, k_2, \ldots$ be the kinds named by <kindname$_0$>,<kindname$_1$>,... respectively. The preceding declaration states that an object of kind k_1 may revert to any one of the kinds $k_2, \ldots, k_j$; that an object of kind k_{j+1} may revert to any one of the kinds $k_{j+2}, \ldots, k_{j+m}$, etc. This declared information is used in the following ways. Suppose that an object i whose values have been declared to have kind k_1 appears in an operation. For the sake of illustration, we assume this operation to be monadic, and to have the form

<u>op</u> i,

<u>op</u> being some particular operation symbol.
Suppose now that no <u>from</u> statement describing the mode of application of the operator <u>op</u> to an object of kind k_0 has been provided. Then, in compiling, an attempt will be made to find a <u>from</u> statement defining the way in which <u>op</u> applies to an object of one of the kinds $k_2, k_3, \ldots k_j$. If one and only one such statement is found, this will be taken to define the manner in which <u>op</u> is to be applied to an object of kind k_1. Replacement of the kind k_1 by one of the kinds $k_2, k_3, \ldots k_j$ we call <u>reversion</u>. If more than one <u>from</u> statement defining the manner of application of <u>op</u> to an object of kind k_i, $i \geq 2$, is found, an ambiguous situation exists, and an appropriate diagnostic will be issued.

If no such statement exists, then an attempt will be made (recursively) to apply the process of reversion to each of the kinds $k_2, k_3, \ldots k_j$. That is, one uses any declarations

<u>revert</u> ..., $k_i(k_{i1}, k_{i2}, \ldots, k_{im_i})$, ...;

which have been made, and searches for a <u>from</u> statement defining the manner in which <u>op</u> is to be applied to an object of kind k_{im}. If this process is continued as far as possible, one of three situations will result. It may be that no chain of reversions leads from k_1 to a kind k for which there exists a declared manner of application of the operator op. In this case, we take it that the application of op to an object of kind k_1 is undefined, and issue an appropriate diagnostic. Suppose, on the other hand, that some chain of reversions leads from k_1 to a kind k for which the manner of application of op has been declared. In this case, we collect all triples consisting of such k, of the length n of the chain of reversions leading from k_1 to k, and of the routine to be used in applying op to an object of type k. If there exists precisely one among these triples for which the length n takes on its minimum value, we use this to define the application of op to an object of type k. If, on the other hand, there exists more than one among these triples for which n takes on its minimum value, an ambiguous situation exists, and we issue an appropriate diagnostic.

The reversion procedure just described for the case of operators with a single parameter will be used in suitably generalized form for operations of any number of parameters. Suppose that some certain operation, which we shall designate by the symbol ϕ, is to be applied to a collection of parameters of kinds $k^{(1)},\ldots,k^{(m)}$. If there exists a from statement declaring the manner in which this application is to be made, we proceed in the specified manner. Suppose on the other hand that no such declaration has been made. In this case, we consider all tuples of the form

$\langle rk^{(1)}, k^{(2)},\ldots,k^{(m)}\rangle$, $\langle k^{(1)},rk^{(2)},\ldots,k^{m}\rangle \ldots \langle k^{(1)},k^{(2)},\ldots rk^{m}\rangle$

where $rk^{(i)}$ is a kind to which k(i) may revert. Each of these tuples is a reversion of length one. If there exists exactly one tuple of length one for which there is a from declaration for op, then this from declaration specifies the semantics of the appliation of the operator op to a collection of parameters of kind $k^{(1)},k^{(2)},\ldots,k^{(m)}$ respectively. If there is more than one specification, then this is an ambiguity error and an appropriate diagnostic is issued.

If there is none, each of the tuples of length one is again reverted using the same prescription and a search is made for an applicable from declaration. This process continues until an applicable from declaration is found or until no further reversions are possible. If a from declaration is not found, we consider this application of op to be ambiguous and issue an appropriate diagnostic.

We now come to describe a last declaration in the present 'object-kind' related group. This declaration allows sinister calls to be used in a kind-dependent way. It has the form

forℓ <kindname$_0$>(<kindname$_1$>,...,<kindname$_m$>)=<kindname$_{m+1}$>
use <routinename>;

Let $k_0,\ldots,k_{m+1}$ be the kinds designated by the tokens appearing as kindnames in the above declaration. The declaration applies in cases in which a sinister call of the form

$$t_0(t_1,\ldots,t_m) = t_{m+1};$$

is encountered, and in which $t_0,\ldots,t_{m+1}$ are respectively of kinds $k_0,\ldots,k_{m+1}$. It applies also to a wider range of situations under the reversion rules just explained, which the reader will readily adapt to the present slightly different situation. In situations in which the declaration applies, a (sinister) call to the procedure named by the <routinename> occurring in the declaration is generated; the normal rules apply to compound arguments appearing in this sinister call.

Closely related declarations, having the somewhat different syntactic forms

forℓ <kindname$_0$>{<kindname$_1$>,...,<kindname$_m$>} = <kindname$_{m+1}$>
use <routinename>;

forℓ <kindname$_0$>[<kindname$_1$>,...,<kindname$_m$>] = <kindname$_{m+1}$>
use <routinename>;

forℓ <kindname$_1$> op <kindname$_2$> = <kindname$_3$> use <routinename>;

etc. are also provided. The reader will readily deduce the import of these declarations. For use with sinister forms admitting an indefinite number of arguments, we provide the related declaratory forms

forℓ <kindname$_0$> (<kindname$_1$>,...,<kindname$_m$>-)
 = <kindname$_{m+1}$> use <routinename>;

forℓ <kindname$_0$>{<kindname$_1$>,...,<kindname$_m$>-}
 = <kindname$_{m+1}$> use <routinename>;

etc. The above-described conventions concerning 'extra parameters' apply here. We illustrate the resulting semantics with an example. Suppose that the declaration

```
forℓ branchlist(tree-) = tree use treelist;
```

is active in a context in which the sinister call

```
(1)     branchlist(t1,t2,...,tn) = newtree;
```

also occurs, and that *t1* and *newtree* have been declared to be of kind *tree*. Then the code represented most directly by the sinister call

```
treelist(t1,<t2,...,tn>) = newtree;
```

will be compiled in place of (1). Note that this same code may also be written as

```
x = <t2,...,tn>;
treelist(t,x) = newtree;
<t2,...,tn> = x;
```

where x is a compiler-generated temporary variable.

In some cases one will wish to associate some particular action with a simple assignment operation appearing in a source text as

```
(2)                a = b;
```

provided of course that a and b are of specified kinds k_1 and k_2. This can of course be done using the following particular case of the general <u>forℓ</u> declaration:

<u>forℓ</u> <kindname$_1$> = <kindname$_2$> use <routinename>;

If no such declaration is provided, while a and b are of the same kind, then the standard SETL assignment procedure will automatically be used.

In some cases, we will wish to treat a value which would ordinarily be of one kind as if it were of another kind. This will be the case especially for constants occurring in SETL programs, for complex structures built up out of constants during one or another 'initialization' process, and for structures read in from external media. To allow for this, we introduce the binary 'syntactic operator' <u>as</u>. The expression

x <u>as</u> k

is identical, as a SETL object, with x, but is treated (during compilation) as an object of kind k.

We give code in SETL to resolve the semantics of operators in section 6 after we discuss the compilation process and the form of interpretable text.

4. Base-level interpreter.

We now sketch a base-level interpreter which is capable of sustaining SETL (essentially this defines the SETL calling conventions). The data structure required by the interpreter is

text - tuple of subroutines - A subroutine may be either compound or primitive. If it is compound, the entry in *text* is a vector of interpretable instructions. If the subroutine is primitive, i.e., an operation conforming to the implementation level requirements of the SETL system, but written in some acceptable lower-level language, the corresponding entry in *text* contains linkage information. Linkage to routines whether compound or primitive is 'by value with deferred argument return' as described in NL 53. The values of all subroutine arguments become part of the environment of the called routine, and are manipulated there just as any other values. After return, all arguments in the calling routine are set to the values which they had in the called routine immediately before return. Except in the case of a primitive which is flagged as 'polyargument', interpretation of a call operation verifies correspondence between the number of arguments appearing in the call and the number of arguments appearing in the called routine. This corresponds to a compiled style in which routines of a variable number of arguments are not really possible. Of course, SETL will allow any number of values to be transmitted to a subroutine; it is only necessary to pre-group these values into a vector.

The "instructions" in a compound routine belong to one of the classes *subroutine call, sinister call, subroutine return, conditional transfer, unconditional transfer,* or *stop.*
The arguments designate variables.

Each variable is "local" to exactly one subroutine and is represented as a pair <*subrno,varno*>. If the variable is local to the subroutine being interpreted a single integer *varno* represents it. Constants are represented as <const,*val*> whose fixed value is *val*. The arguments of a subroutine, $arg_1, arg_2, \ldots, arg_n$ are the 1st, 2nd,...,nth variables local to that routine.

We now turn to the "instructions" that the interpreter processes. The principal vehicle is subroutine call

$$<\mathrm{call}, arg_0, arg_1, \ldots, arg_n>$$

where arg_0 is either a constant whose value is a subroutine or is a variable whose value is a subroutine. For example,

<call,<const,assign>,left,right>

where *left* and *right* are integers denoting variables local to the currently executing subroutine. To expedite determination of the called subroutine, we will *assemble* this code to

<callc,assign,left,right>

The opcode *callc* implies that the second component is a constant designating a subroutine. Sinister calls are formed as

$$<\mathrm{lcall}, \mathrm{var}_0, \mathrm{arg}_1, \ldots, \mathrm{arg}_n>$$

or

$$<\mathrm{lcallc}, \mathrm{subrno}, \mathrm{arg}_1, \ldots, \mathrm{arg}_n>$$

Subroutine names will be variables local to a trivial global routine *outrout*. See the namescoping algorithms above.

Additional interpreter "instructions" include

<go, label>

<ifgo,var,label>

and

<ifnotgo,var,label>

The last form is provided to reduce the number of labels generated in the "compilation" of *if-then-else* expressions in the host language (see below). *Label* is either a variable whose value is a label or is a constant, in which case the forms

<goc,*stmtnr*>

<ifgoc,var,*stmtnr*>

<ifnotgoc,var,*stmtnr*>

where *stmtnr* is an integer, are produced by the assembler. A transfer may not pass from one subroutine to another. The last interpreter "instructions" *stop* and *subroutine return* have no arguments.

To permit access to variables local to another routine, the most recent environment block of a subroutine *rout* is retained as a tuple *curenv(rout)*. If *rout* is the currently executing subroutine, its environment block is *nowenv*. Another call statement causes *curenv(rout)* to be stacked. *Curenv(rout)* is set to *nowenv* before control passes to the called routine.

We now identify the variables used in the interpreter routine.

nowrout	is the currently executing routine
nowenv	environment block of *nowrout*
argno(rout)	the number of arguments of a routine If *argno(rout)* is Ω, *rout* is a polyargument primitive.
prim(rout)	a flag distinguishing between programmed routines and primitives.
primcall(rout)	subroutine which effects the operation associated with a routine *rout*
invoc(rout)	an array counting the number of prior invocations of *rout*

curenv(rout) the environment block of *rout* at the time of its last invocation

envstack tuple used as stack during the recursive calling process

dexit is an assumed routine which supplies useful diagnostic information in case of normal or error exit.

We give two source language macros and then the code for the base-level interpreter.

```
macro  vararg(x) = if atom x then nowenv(x)
         else if x(1) eq const then x(2)
           else curenv(x(1))(x(2)) endm;

macro refarg(x)= if atom x then nowenv(x)
           else curenv(x(1))(x(2)) endm;

/* base level interpreter
invoc(rout) curenv(·), lc, nowenv must be initialized */
nextop: lc = lc+1;
getop:  opitem = nowrout(lc);
        go to <call,callc,lcall,lcallc,retn,go,goc,ifgo,ifgoc,
           ifnotgo,ifnotgoc,stop>(opitem(1));
/* entries for subroutine invocation follow - subname is an integer
                                              designating a routine*/
lcall:   subname = refarg(opitem(2));
         sflag = t;  /* marks sinister call */
         go to link;
lcallc:  subname = opitem(2);
         sflag = t;  go to link;
callc:   subname = opitem(2); sflag = f; go to link;
call:    subname = refarg(opitem(2));sflag = f;
link:
/* check argument number */
if not sflag  and  argno(subname) is argnr ne (#opitem)-2

         then dexit(2);
endif;
/*  stack  curenv(nowrout) */
```

```
if invoc(nowrout) gt 1  then
    envstack(#envstack+1) = curenv(nowrout);
end  if;
curenv(nowrout) = nowenv;
newenv = curenv(subname);
/* pass arguments of subroutine call to newenv */
( 3 ≤ ∀j ≤ #opitem)
    newenv(j) = valarg(opitem(j));
end ∀j;
/* put return information into newenv required to return */
newenv(1) = nowrout;
newenv(2) = lc;
nowenv = newinv;
nowrout = text(curenv(outrout,subname));
if prim(nowrout) then primcall(nowrout); go to retn;;



/* else compound subroutine */
lc = 1;
go to getop;
/* end call operation */
return:  invoc(nowrout) = invoc(nowrout)-1;
         argnr=argno(nowrout); /* number of arguments*/
         nowrout = nowenv(1);
         lc = nowenv(2);

/* restore arguments    */
         retenv = nowenv;
         nowenv = curenv(nowrout);
         if invoc(nowrout) ne 0
             then curenv(nowrout) = envstack(#envstack);
                  envstack(#envstack) = Ω;
         end if;
         (3 ≤ ∀j≤argnr+2)|hd opitem(j) ne const)
```

```
          valarg(j) = retenv(j);
          end ∀j;
          go to nextop;
go:       dest = valarg(opitem(2));
gothere: if (1 le dest and dest le #nowrout)
               then go to getop;
               else print 'illegal transfer operation; dexit(3);
          end if;
goc:      dest = opitem(2); go to gothere;
ifgo:     dest = valarg(opitem(3));
goif:     if valarg(opitem(2))
               then go to gothere; else go to nextop;
          end if;
ifgoc:    dest = opitem(2); go to goif ;
ifnotgo: dest = valarg(opitem(3));
goifnot: if valarg(opitem(2))  then go to nextop;;
          go to gothere;

ifnotgoc:dest  = opitem(3); go to goifnot;
stop:     dexit(4);
/* end interpreter */
```

5. Tree to Linear Text Compiler

Next we 'skip back' one step toward the host language level, and present a tree-to-linear-text 'compiler' close to that which might be used in the SETL system. This routine accepts as input 'abstract syntactic text', i.e., prediagnosed and name-resolved tree structures. It uses what are basically a tree-walk, temporary variable generation and label generation processes to produce linearized text almost identical with that required by the base-level interpreter. The labels are generated in a form somewhat different from that required by the base level interpreter. To process this text into directly interpretable form, an intermediate step of abstract 'assembly' is required, the code for which will be given below. This 'assembly' process may generate 'repeated label' and 'missing label' diagnostics.

In the tree-structured text, variables will be represented as <var,*characterstring*>. The name scoping algorithms are exercised prior to compilation and result in the assignment to each token of a pair of integers <*subnro, varno*>. During compilation each *characterstring* is replaced by an internal representation for the variable it represents. If the subroutine being compiled owns the variable, then a single integer is included in the developing text, otherwise the pair is included. As scope openers and terminators are in place, the compilation process is aware of the namescope in which the source text which generated the tree-structured text appears. In SETL, the indications for macro expansion are syntactically indistinguishable from those for subroutine calls and will be compiled into a subroutine invocation. The "compiler" attempts to find the number of the subroutine associated with a subroutine invocation. If no assignment has been made, a macro expanding routine is invoked. We do not give the details of the macro expansion process. The reader should note that a macro may contain code or declarations (see above) associated with the determination of the semantics of the SETL operations but may <u>not</u> contain statements which affect name resolution, i.e. <u>include</u>, <u>global</u>, <u>group</u>, <u>alias</u>, <u>own</u>, ...

We now give an example of the compilation process.

'source': a = f(a+b);

/* we assume that a is owned by the routine in which this statement occurs; b is owned by another routine, and f is global */

the input to the 'abstract compiler' described below:

<assign,<var,a>,<fcall,<var,f>,<fcall,<const,+>,<var,a>,<var,b>>>

/* n_a is the integer corresponding to the 'locally owned' resolved name *a*;

m_b the number of the subroutine owning *b*. n_f and n_b are the integers corresponding to the resolved names *f* and *b* respectively*/

as input to the base level interpreger

$<$callc,+,n_a,$<m_b,n_b>$,$n_t$$>$,

$<$call,$<0,n_f>$,n_t , $n_a$$>$

/* n_t is an integer corresponding to a "generated temporary" variable */

$<$callc, assign, n_a,$n_t$$>$

Note that since we treat labels as being global to the full body of a subroutine, and since we allow transfers between any two points in such a semantic scope, <u>compilation</u> rather than direct interpretation of abstract syntax trees is indicated. Indeed, direct interpretation of syntax trees would make recursive stacking-unstacking actions necessary at many points within a subroutine, and this would require the association with every jump of expensive stack-checking and -correction actions. We adopt a more highly compiled approach and associate stacking actions with subroutine call and return exclusively, and restrict the maximal scope of transfers to lie within a single subroutine.

The 'abstract syntactic' text accepted by the 'compiler' described in the present section supports certain important compound linguistic constructs very directly. It provides in an abstract representation the following forms:

i. expressions with subexpressions
ii. code blocks
iii. code blocks within expressions
iv. iterative if - then - else - if and if - then - else - if -else forms
v. A 'while' statement
vi. go to, call, return, and return (expression) statements.
vii. A 'subroutine' header statement, which designates an attached code section as a subroutine body, and which gives both the serial number of a particular subroutine and the number of arguments which it possesses. Note that subroutines are assigned serial numbers by the name-scoping procedures. These same procedures standardize the representation of 'globally' or 'externally' referenced variables.
viii. 'Primitive subroutine' statements, in dexter and sinister form, which designate an attached constant as the (primitive, hence unanalyzed) calling information for a primitive. Each sinister primitive p is associated with a dexter primitive (with one fewer argument) of which p is the 'associated sinister form'.

We now discuss the syntax and semantics of the host language forms.

<u>constant</u>	<const,value> where value is a suitable encoding of the constant
<u>variable</u>	<var,index> where index is a single integer, i, which refers to the ith variable of the current subroutine or is a pair <m,n> designating the nth variable of the mth subroutine

<u>function invocation</u> <fcall,fname,expl,exp2,...,expn>
where fname,expl,exp2,...,expn are expressions

<u>if expression</u> <ife,cond$_1$,exp$_1$,cond$_2$,exp$_2$,...,cond$_n$,exp$_n$,exp$_{n+1}$>
corresponds to the SETL code

if cond$_1$
  then exp$_1$
  else if cond$_2$
    then exp$_2$
    else if cond$_3$

        else if cond$_n$
          then exp$_n$
          else exp$_{n+1}$ ... ;

The value returned is an expression.

<u>block</u> <<u>block</u>,stmt$_1$,stmt$_2$,...,stmt$_n$>

where stmt$_1$,stmt$_2$,...,stmt$_n$ are statements to be compiled separately and executed consecutively.

<u>while</u> <<u>while</u>, cond, dostat, block, contlab, outlab>;
The compiled code is a sequence equivalent to the SETL code.

start: if cond
    then block; contlab; dostat; go to start;
  outlab;

cond is an expression whose compile time value is <u>t</u> or <u>f</u>
block is a tuple of the form above and
contlab, and *outlab* are pregenerated labels.

<u>go to</u>: <<u>goto</u>,exp>

exp is an expression whose compile time value is a label.

<u>shortif</u>: <<u>ifs</u>,cond, block>
cond is an expression, whose compile time value is <u>t</u> or <u>f</u>.
If the value is <u>t</u> the code in *block* is executed.

Now we explain the long if statement.

longif: <<u>ifℓ</u>, $cond_1$, $block_1$, $cond_2$, $block_2$, ..., $cond_n$, $block_n$>.
Each of $cond_1$, $cond_2$,..., $cond_n$ is an expression.
The code generated is equivalent to the SETL sequence

```
if cond1
   then block1;
      if cond2
         then block2;
            if cond3
               then ...
            end if;
      end if;
end if;
```

The kth block is executed if and only if each of $cond_k$, $cond_{k-1}$, ..., $cond_1$ is <u>t</u>. If $cond_k$ is false, transfer is made beyond the range of the first if.

There is a scope declaration <<u>scope</u>,*characterstring*>. No code is generated by this "statement". However, *characterstring* is made available to the coded function *internalrep* which replaces external representations like <<u>var</u>,*charstring*> with an internal representation n or $<m,n>$. Also a declaration <<u>endscope</u>> is provided. Occurrence of this declaration closes the current scope. (See below.)

We now give the format for subroutine headers.

subroutine header: <subrout, subrnumber, # of arguments>

primitive subroutine header: <primsub. subrnumber, #of arguments,calling info>

sinister primitive subroutine header:

<lprimsub,subr number, # arguments, calling info, number of dexter form >;

Subr number is the number of the subroutine. In the primitive formats, *calling info* is an integer which represents the linkage information which is decoded by callinf. We omit code for this routine. The appearance of a subroutine header marks the termination of the compilation of the preceding subroutine and causes *currtext*, which contains the compiled code, to be entered in the comprehensive vector *text*. *Currtext* is initialized to blanksubr which is a skeletal form into which the identifying number, number of arguments, the primitive flag, and the sinister form of a dexter routine are inserted. The routine add attaches additional statements to the vector *currtext* during compilation.

We also provide a *return* statement which has no arguments.

return: <retn>

In addition, there is an *expression return* instruction which has one form within a routine which corresponds to statements

return f(x + g(y))

and another form within an expression codeblock, which supports the SETL construct

y = if bool then z else w;

In the former case, the expression is evaluated and the result is assigned to <var,nr> where *nr* is one plus the number of arguments in the currently executing subroutine.

The compilation of expression-representing code blocks involves a few details which may not be entirely familiar. Before compilation of the statements of such a block begins, an 'exit label' *ebout* is generated, and a global compiler variable *ebtemp* is set equal to the required result variable of the block. "Expression return" statements of the form

<return, expression> ,

which normally would be compiled as

```
 <callc, assign, output argument number,<expression>>,
      <return>
```

are compiled as

```
 <callc, assign, ebtemp,<expression>>,
      <goto, ebout>
```

Note that our conventions allow code blocks used as expressions to contain arbitrary statements, including quite general 'go to' statements, and also to contain embedded code blocks, leading to expressions which are very general.

The compilation of general assignment statements, which includes the expression of sinister calls, turns out to be surprisingly easy. First one compiles whatever code corresponds to the expression appearing on the right-hand side of the assignment statement. Once this is done, we have only to compile the special assignment case that would appear in source as

```
<sinister expression> = temp;
```

To do this, we first generate the expansion which corresponds to the source

```
temp = <sinister expression>;
```

Then we take the code which results, invert its order, transform every dexter call into a sinister call, and append the result of these successive transformations to the already generated code. Note, for example, that (writing in a suggestive rather than precise notation) an assignment like

```
f(g(a, h(b)), h(c)) = a + b;
```

compiles first into

```
call (sum,a,b,t);
f(g(a, h(b)), h(c)) = t;
```

then into

```
call(sum,a,b,t);
call(h,b,t1);
call(g,a,t1,t2);
call(h,c,t3);
call(f,t2,t3,t);
```

and finally into

```
call sum(a,b,t);            /* t = a + b */
call(h,b,t1);               /* t1 = h(b) */
call(g,a,t1,t2);            /* t2 = g(a,t1) */
call(h,c,t3);               /* t3 = h(c) */
ℓcall(f,t2,t3,t);           /* f(t2,t3) = t */
ℓcall(h,c,t3);              /* h(c) = t3 */
ℓcall(g,a,t1,t2);           /* g(a,t1) = t2 */
ℓcall(h,b,t1);              /* h(b) = t1 */
```

The following comments describe the principal routines of the algorithm given below.

compile(obj) - transforms the code block *obj* which is of tree form into a linear interpretable code sequence.

excomp(exp) - is a function which transforms the expression *exp* which is of tree form into a linear sequence. The value of this function is a variable, perhaps a generated temporary, to which the value of the expression is assigned at run time.

gentemp(t) - generates a temporary variable, t, or more precisely, a unique integer representing a temporary variable.

genlab(l) - generates a label, or more precisely, a unique integer, representing a label, from which an actual label item will be produced by the assembler described in the following section.

currtext - is the tuple to which code fragments of the current subroutine are added during the compilation process.

add *frag* - attaches *frag* to the partially compiled code sequence *currtext*.

macexpand - expands a macro and adds the result to *currtext*. The code is not given.

We first give the code for the function *excomp(exp)* which returns a temporary variable to which the value of *exp* is assigned upon the execution of the code generated.

We assume a function *internalrep( )* which converts the external representation of a variable into an internal representation. *Internalrep* uses as hidden variables the namescope in which the original source code appeared and the number of the subroutine currently being compiled. *Scope* and *endscope* declarations cause the current namescope to be changed.

```
definef excomp(exp);
/* inblock = f if compiling function return */
go to <ife,block,fcall,const,var>(exp(1));
const:  return exp;
var:    return internalrep(exp(2));
/* internal representation substituted for external form <var,charstring */
fcall:  add     (<call> + excomp[exp(2:)] + <gentemp(argtemp)>);
        return argtemp;
block:  /* case of code block within an expression */
        inblock = t; genlab(ebout);  gentemp(ebtemp);
        compile[ exp(2:)];
        inblock = f;
        add <herelabel,ebout>
        return ebtemp;
ife:    /* compilation of conditional expression */
        genlab(outlab); j = 2; gentemp(outvar);
        (while j lt #exp doing j = j+2;)
        genlab(nxtcond);
```

```
        add <ifnotgo, excomp(exp(j)),<const,<label, nxtcond>>>;
        add <callc, assign, outvar, excomp(exp(j+1))>;
        add <goc, <label,outlab>>;
        end while;
        add <callc, assign, outvar, excomp(exp(j-1))>;
        add <herelabel, outlab>;
        return outvar;

end excomp;
```

We now give code for the process of compiling a code block *obj*. We identify some of the important functions and parameters used below.

subno - integer designating the subroutine being compiled
nargthis - number of (explicit) arguments of subroutine being compiled
currprim - is t, if subroutine being compiled is primitive
argno(.) - function which extracts the number of arguments of the current subroutine
prim(.) - function which is t if argument is a primitive; f otherwise.
callinf(.) - decodes the calling information in a primitive subroutine header.

inblock - flag which marks the compilation of *return* exp rather than *x = exp*
assemble(.) - processes labels into form consistent with conventions of base-level interpreter.

```
/* the 'abstract compiler algorithm' */
define  compile(obj);
initially sublist = nult;  subno = Ω;
    currtext = nult;  inblock = f;;
go to <ifℓ, ifs, while, assign, goto, block, subrout,
    lprimsub, primsub, labhere, call, retn, eretn,
    scope, endscope> (obj(1));
subrout:  lprimsub:  primsub:
/* terminate the last subroutine */
if (subno ne Ω and not currprim)
    then /* save text of current subroutine */

      prim(subno) = f; text(subno) = assemble(currtext);
end if;
<-,subno,nargthis,-> = obj;
currtext = blanksubr;
argno(currtext) = nargthis;
if obj(1) eq subrout
    then prim(currtext) = f; currprim = f; return;
end if;
/* else primitive header */
prim(currtext) = t;
callinf(currtext) = pconvert(obj(4));
if obj(1) eq lprimsub
    then sinf(obj(5)) = subno;
end if;
sublist(subno) = currtext;  currprim = t;
return;
scope:   newscope(exp(2)); return;
         /* this routine changes the namescope              */
endscope: scopend;
        /* reverts current scope to its parent */
```

```
call:     if internrep(obj(2)) is subname eq Ω
            then add macexpand(obj(2:));
            else add (<call> + compile[obj(2:)]);
          end if;
          return;
retn:     add <retn>;  return;
eretn:
          /* one form within routine, another within code block */
          if inblock  then go to blockret;;


          add <callc, assign,<var, nargthis+1>, excomp(obj(2))>;
          return;

 blockret:     /* ebout and ebtemp are global */
          add <callc, assign, ebtemp, excomp(obj(2))>;
          add <goc,<label, ebout>>;
          return;
 goto:    add <go, excomp(obj(2))>;  return;
 assign:  /* compilation of assignment statement which has
          left and right side */
          rightside = excomp(obj(3));
          ldexter = #currtext;
           /* now compile lefthand side */
               leftside = excomp(obj(2));
           /* change last expression to lcall if obj(2) not atom */
           if (#currtext eq ldexter)
               then add <callc, assign, leftside, rightside>;
               else /* change last expression to lcall */
                  text(#text is npt)(1) = lcall;
                  /* sequence of lcalls in reverse order */
                  (npt > ∀j > ldexter)
                     x = text(j); x(1) = lcall; add x;
                  end ∀j;
   end if;
   return;
```

```
ifℓ: ifs: /* long and short if statement */
    genlab(outlab), j = 2;
    (while j lt #obj doing j = j+2;)
        add <ifnotgo, excomp(obj(j)), <const,<label,outlab>>>;
    end while;
    add <herelabel, outlab>;
    return;
block:   (2 ≤ ∀j ≤ #obj) compile(obj(j));; return;
while:  <-,cond,block,dostat,contlab,outlab> = obj;
    genlab(start); add <herelabel, start>;
add <ifnotgo, excomp(cond),<const,<label, outlab>>>;
compile(block);
add <herelabel,contlab>;
compile(dostat);
add <goc,<label,start>>;
add <herelabel, outlab>;
return;
```

For completeness we give

```
define  add x;
/* currtext is global */
currtext = currtext + <x>;
return;  end;
```

Also the function

```
definef internalrep(extrep)
cstring = extrep(2);
internrep = internalias (ns,cstring);
/* ns is the current namescope internalias produced by
    namescope algorithms  see below */
return if hd internrep eq subno
          then internrep(2) else internrep;
end;
```

6. Assembler

Labels are processed into the form accepted by the base-level interpreter by the function *assemble*. The algorithm requires two passes.

The first pass determines the location of all labels; the second adjusts all arguments of the form <label,dest> into a single integer. During the second pass entries of the form

<call,<const, subname>,...>

are adjusted to

<callc, subrno, ... >

where subrno is the index in *text* at which *subname* is stored. Similar transformations are made on entries of the form

<lcall,<const, ...>, ...> .

Opcodes *goto*, *ifgo*, and *ifnotgo* are adjusted to *gotoc*, *ifgoc*, and *ifnotgoc* if the labels are constants.

```
definef assemble(subr);
initially where = {<goto, go>,<ifgo, stmtifgo>,
    <ifnotgo, stmtifnot>,<call, stmtcall>,
    <lcall, stmtlcall>, <retn, nop>,<stop, nop>};;
macro findconst(loc,fn,newop,adjlabl)
    if n atom item(loc) and hd item eq const
         then item(loc) = fn(item(loc));
              item(1) = newop;
    endif;
         subr = subr + adjlabl(<item>);
         endm;
labels = nℓ; newsubr = nult;
lc = 1;
/* determination location of all labels */
(∀entry(j) ∈ subr)
if entry(1) eq herelabel
    then if labels(entry(2)) ne Ω
         then print 'second specification of label',dec entry(2);
         else <entry(2),lc> in labels;
       end if;
    else lc = lc+1; newsubr = newsubr + <entry>;
end if;
end ∀entry;
```

```
/* have determined location of all labels - adjust destinations
    of all <const,<label,lblnr>>   on second pass   */
/* second pass */


subr = nult;
(∀item(j) ∈ newsubr)
   go to where (item(1));
go: findconst(2,reallbl, goc,); return;
stmtifgo:      findconst(3,reallbl,ifgoc,); return;
stmtifnotgo:   findconst(3,reallbl,ifnotgoc,); return;
stmtcall:      findconst(2,realrout,callc,adjlabl); return;
stmtlcall:     findconst(2,realrout,lcallc,adjlabl); return;
nop:           subr = subr + <item>; return;
end assemble;
```

We now give the functions which perform the transformations.

```
definef reallbl(arg);
/* labels is global */
return if labels(arg(2)) is retarg eq Ω
         then print 'missing label', maklbl(arg);
              retn lastlbl;
         else retarg;
end reallbl;

definef realrout(rout);
return rout(2);
end realrout;
```

Finally, the function

```
definef adjlabl(item);
return (item(1:2) + [+:3≤j≤#item]<if hd item(j) eq label
         then reallbl(item(2))  else item(j);>);
end adjlabl;
```

7. Resolution of programmer specified semantics

In what follows, we shall be processing the output of a tree to a linear-text compiler similar to that defined by the last given algorithm. Our remarks are peculiar to SETL, although the substance of them can be modified to accommodate other languages.

We suppose that the abstract recursively structured syntactic tree discussed above has been linearized but that ambiguities in the names of the operators exist. The namescoping/name-propagation process has been carried out, so that every subroutine and every variable is put in correspondence with a pair <m,n>. We suppose that similar symbol transformations have been applied to the names appearing in our various declarations.

It is the objective of the following to resolve the significance of function and operator references starting with the form in which such references initially appear. We will outline the processes which ascribe a definite interpretation to the meaning of an operator or function in a line of code; for example, to the plus sign in

... A + B ...

This meaning depends on the kind types which have been specified for A and B in the namescopes in which this line appears and upon the "from" declarations applicable to these kindnames which are active within this namescope. We gather together the forms of semantic definition available to the user, as they have been specified above. First, the dexter forms, in which we include the *allow* statement.

monadicfn	1d from op <kindname > get <kn > using <routname>
dyadicfn	2d from <kindname1> op <kindname2> get <kn > using <routname>
fneval	3d from <kn0>(<kn1>,<kn2>,...,<knj>,-) get <kn> using <routname>
releval	4d from <kn0>{<kn1>,<kn2>,...,<knj>,-} get <kn> using <routname>
rangeval	5d from <kn0>[<kn1>,<kn2>,...,<knj>,-] get <kn> using <routname>
setform	6d from {<kn1>,<kn2>,...,<knj>,-} get <kn> using <routname>
brackform	7d from [<kn1>,<kn2>,...,<knj>,-] get <kn> using <routname>
tuplform	8d from <<kn1>,<kn2>,...,<knj>,-> get <kn> using <routname>
subcall	9d allow <kn0>(<kn1>,<kn2>,...,<knj>,->)using <routname>

The '-' in the forms 3d to 9d is optional and indicates that a variable number of additional arguments may be part of the argument list of the construction.

The possible existence of user supplied redefinition of the semantics of the normal SETL constructions restricts the amount of digestion of the source program which can occur prior to the analysis of these declarations. Let underscoring mark user defined infix operators. The constructions a $\underline{f}$ b, and f(a,b) are inherently different even though the symbols are the same. Different forms of *from* declaration apply to these two different constructions. Knowledge of the form of the construction must be preserved until the user supplied semantics are considered. Thus we give these types of construction generic names *dyadicfn* and *fneval*. The arguments to each of these generic functions is <f,a,b>. Note that the infix symbol $\underline{f}$ appearing in a $\underline{f}$ b is the first token of the argument list.

Construction 6d corresponds in the usual semantics to the process of set formation and similarly 8d corresponds to tuple formation. We assume the tree to linear-text compiler will designate these processes in a generic manner and compile the tokens which surround the punctuation into an argument list. We stipulate the designators *setform*, *brackform*, and *tuplform* for each of 6d, 7d and 8d respectively. Similarly, 3d mimics the syntax of functional evaluation. Given that f is a set and not a routine, the standard semantics of f(a,b,c) is the invocation of a routine in the RTL which returns d if there is only one tuple in f, considered as a set, with initial components a,b,c whose fourth component is d. Briefly,

g{a} is the set {<u>hd</u> x, x ∈ g}

and

h[a,b] is the set {elth(3), elth∈h, t_1∈a, t_2∈b|elth(1:2) <u>eq</u> <t1,t2>}.

The semantics of these constructions are modified by declarations of the form 4d and 5d respectively. We give each of these constructions a generic designator *fneval*, *releval*, and *rangeval* in analogy to their usual semantics in SETL.

In all of these constructions, the clause "using <routname>" is optional. If omitted, the construction is interpreted as designating the usual SETL operation. The argument list which is compiled for each of these forms depends on the construction. For later reference, we include an example of the argument lists produced by the parser and preserved by the tree to linear-text compiler for each of the constructions 1d,2d,...,9d. The symbol t denotes the result in each case. We stipulate arg0,arg1,...,argj have kindtypes kn0,kn1,...,knj respectively.

1d <op, arg1, t>
2d <op, arg1, arg2, t>
3d <arg0, arg1, ..., argj, t>
4d <arg0, arg1, ..., argj, t>
5d <arg0, arg1, ..., argj, t>
6d <arg1, arg2, ..., argj, t>
7d <arg1, arg2, ..., argj, t>
8d <arg1, arg2, ..., argj, t>
9d <arg0, arg1, ..., argj, t>

In addition to the dexter forms, sinister forms are available. We give each form a generic designator.

```
ℓmonadicfn    1ℓ forℓ        op <kn1> = <kn > use <routname>
ℓdyadicfn     2ℓ forℓ <kn1> op <kn2> = <kn > use <routname>
ℓfneval       3ℓ forℓ <kn0>(<kn1>,<kn2>,...,<knj>,->
                                   = <kn> use <routname>
ℓreleval      4ℓ forℓ <kn0>{<kn1>,<kn2>,...,<knj>,-}
                                   = <kn> use <routname>
ℓpowfneval    5ℓ forℓ <kn0>[<kn1>,<kn2>,...,<knj>,- ]
                                   = <kn> use <routname>
ℓsetform      6ℓ forℓ      {<kn1>,<kn2>,...,<knj>,-}
                                   = <kn> use <routname>
ℓbrackform    7ℓ forℓ      [<kn1>,<kn2>,...,<knj>,-]
                                   = <kn> use <routname>
ℓtuplform     8ℓ forℓ      <<kn1>,<kn2>,...,<knj>,->
                                   = <kn> use <routname>
ℓassign       9ℓ forℓ       <kn1> = <kn2> use <routname>
```

Types 6ℓ, 7ℓ, and 8ℓ are novel. Their significance should be clear to the reader. In all of the above, the kindlists are terminated by an optional '-'. The presence of this symbol indicates a variable number of additional arguments.

These dexter forms impose constraints on the parse similar to those imposed by dexter forms. For reference, we give the argument list produced by the parser for each construction. *Argi* has kind *kni*.

```
1ℓ <op, arg1, arg2>
2ℓ <op, arg1, arg2, arg3>
3ℓ <arg0, arg1, ...,        argn>
4ℓ <arg0, arg1, ...,        argn>
5ℓ <arg0, arg1, ...,        argn>
6ℓ <arg1, arg2, ...,        argn>
7ℓ <arg1, arg2, ...,        argn>
8ℓ <arg1, arg2, ...,        argn>
9ℓ <arg1,arg2 >
```

We also display the syntax of the 'forit' declaration

```
forit <kindname>   ∈   <kindname1> use
     <firstroutname>,<nextroutname>,<actelroutname>;
```

Note that the source-language iteration

```
(∀x ∈ a) <body> end ∀;
```

is assumed to be expanded into linear code equivalent to

```
xaddr = first(a);
(while xaddr ne Ω doing xaddr = next(a,xaddr);)
     x = actelt(xaddr); <body> end while;
```

For the identity of *first*, *next*, and *actelt* to be determined using user-supplied *forit* declarations, these functions must be marked both in the abstract syntactic tree form of the program and in the linear text derived from it.

We choose *forit1*, *forit2*, and *forit3* as the respective designators of these functions. The argument list compiled by the tree to linear-text compiler must include x and a because the loop header (∀ x∈ a) determines the ultimate identity of the functions designated by *forit1*, *forit2*, and *forit3*. Avoiding redundancy where possible we specify the complete argument list for each of these designations as:

```
forit1:      <x,a>
forit2:      <x,a,xaddr>
forit3:      <x,a,xaddr>
```

We now consider the transformation process. A typical item in the linear text in which operator and functional references must be resolved is

<*desig,ns,arglist,result*>

where,

desig - designator of syntactic type of construction
ns - namescope in which original text appeared
arglist - tuple with local names of arguments
result - local name of result of dexter construction, absent in sinister forms.

According to the preceding remarks, *desig* takes on one of the following values

monadicfn, dyadicfn, fneval, releval, ..., subcall, lmonadic, ldyadicfn,..., lassign, forit1, forit2, and *forit3.*

We assume that the *for, forl,* and *forit* declarations have been processed by the tree to linear-text compiler into a set

using - {<desig,ns,kindlist,polyarg,resultkind,ultrout>}

where,

desig - item type designator, one of *monadicfn, ...*
ns - namescope in which kind declaration appeared
kindlist - tuple of kindnames of arguments
polyarg - <u>t</u> if variable number of additional arguments possible, <u>f</u> otherwise
resultkind- kind of the result in a dexter construction, Ω in sinister
ultrout - integer identifying ultimate routine, if specified, Ω if not

The set *using* must of course include items describing the standard SETL semantics. To provide for the case in which the user does not specify a kind for a variable, we assume that

default(ns) - function which returns the default kindtype for namescope *ns*

is available and defined for every namescope. If a default specification for a namescope is not explicitly made, the default option within that namescope is the specification made in the parent scope. (Cf. the detailed account of namescoping conventions given below.) We assume that *default* has been defined on each namescope by using this recursive construction, and that it is single valued. In the absence of a default declaration in an outermost namescope, '*setlstdtype*' is used. The kind of variables for which no kind declarations have been made is the default specification for the namescope. We also require a set *revertf* built out of the kind and reversion declarations of the form

revertf = {<*ns*, *arg*, *revertarg*>}

in which *arg* reverts to *revertarg* in *ns*.
Since we allow *from* declarations of the form

from p(scalar) get ...

where p is a variable name, kind declarations are in effect reversion stipulations of the first order. Moreover, declarations of the form x <u>as</u> k require an entry be made in *revertf*. Only *from* declarations appearing in the same namescope as a line of source text influence the semantics of a construction.

From text of the form just described, processes which we will now outline will produce directly interpretable code of the form

$<\text{opcode},\text{fn},\text{arg}_1,\text{arg}_2,\ldots,\text{arg}_k>$

where,

opcode - *fcall*, *call*, *lcall*, *goto*, *ifgoto*, or *stop*.

fn - < name designating a function or routine>

arg_i - name of the i-th argument in the form <m,n> - m-th variable of the n-th routine. arg_k is the result of a dexter construction.

We make no distinction in this section between *call* and *callc* or any of the other pairs of related opcodes discussed above.

We outline the conversion process for the line of code

g(c) = a + f(a) (*)

which we assume appears in a namescope *ns* which contains the following declarations; *integer* and *real* are kindnames.

```
from f(real) get real;
from real + real get real using floatadd;
forℓ g(real) = real use evalg;
kind real(a); integer(c);
revert integer(real);
```

A parser and a tree to linear-text compiler together convert the line of source code into the following linearized text

<*fneval*, *ns*, <f,a>,t1> (1)

<*dyadicfn*, *ns*, <'+',a,t1>,t2> (2)

<*lfneval*, *ns*, <g,c,t2,t3>> (3)

which corresponds to the expansion of (*) into

t1 = f(a)
t2 = a + t1
g(c) = t2

After the *from* declarations are processed *using* contains at least the following entries. The fourth entry $\underline{f}$ is the value of *polyarg*.

<*fneval*, *ns*, <f, real>, $\underline{f}$, *real*>

<*dyadicfn*, *ns*, <+, *real*, *real*>, $\underline{f}$, *real*, floatadd>

<*lfneval*, *ns*, <g, *real*, *real*>, $\underline{f}$, *real*, evalg>

The variable a is identified as having kind *real*; c, as having kind *integer*. There is one reversion declaration - a variable of kind *integer* is to be considered also as kind *real*.

The line (1) is interpreted by considering the set *using*{*fneval*,*ns*}. The argument list <f,a> must be matched to the kindlists of the elements in *using*{*fneval*,*ns*}. The result, tl, of a dexter function invocation does not influence the determination of an interpretation. There are no kindlists of the form <f,a>. We then consider pairs of the form <rf,a> and <f,ra> where *rf* is a kind to which f can be reverted and *ra* is a kind to which *a* can be reverted. *a* can be reverted to *real*.

There is a kindlist in *using*{*fneval*,*ns*} which matches <f,*real*>. The line (1) is then changed to

<callc,fneval,f>,a,tl>

The result tl must be assigned the kind *real*, as the clause "*get real*" appears in the original declaration, which reads:

from f(*real*) get *real*

An entry into the sets which govern the reversion of kinds, must be made because *tl* is a temporary generated by the parser and is unknown at the source code level. The tuple

<*dyadicfn*, *ns*, <'+',a,tl>,t2 >

is interpreted after the argument list <'+',a,tl> is reverted twice. There is no specification in *using* {*dyadicfn*,*ns*} in the form <'+', a tl > or as either <'+',a,real> or <'+',real,tl>. However, there is an entry in *using*{ *dyadicfn*,*ns*} which corresponds to <'+',real,real>. The result t2 is given the kind *real*. The code generated is

<callc, fneval, +, a,tl,t2>

Interpretation of the sinister form

<*lfneval*, *ns*,<g,c,t2>>

requires consideration of the kind of the right-hand side *t2*, unlike the dexter forms in which the *kind* of the left-hand side is of no concern. The argument list must be reverted successively to <g,*real*,*real*> before the conversion to interpretable text

<*lcall*, fneval, g, c, t2>

is made.

In general, the principal part of the production of interpretable text from linearized code is to determine the identity of the operator *fn* to be invoked in the case of function invocations or subroutine calls. The algorithm depends upon finding a chain of reversions of the tokens of *arglist*, such that the reverted tokens match the *kind-list* of one of the tuples of *using*{*locfn*,*ns*}. This corresponds to finding a *from* declaration in the relevant namescope. For example, consider the reversion of the argument list <a,b>. If no entry in *using*{*locfn*,*ns*} corresponds to <a,b> then one considers tuples of the form <a,rb> and <ra,b> where rb ∈ revert{b} and ra ∈ revert{a}. These are reversions of level one. If a unique match is found, the reversion process is complete. A nonunique match is an ambiguity error. If no match is found, each element in the set of level one reversions of <a,b> is again reverted and the process is repeated. The details of this process will be clear from the algorithm given below.

Note that it is also assumed in what follows that the name-scoping process creates a 'dummy' subroutine *outrout*, numbered 0, to which belong variables <0,1>, <0,2>, ... whose values are respectively initialized to the first, second, etc. primitives, followed by the first, second, ... subroutines compiled.

These remarks should enable the reader to comprehend the code which follows. First we define various sets which are needed in the code.

```
dexter      - {'monadicfn', 'dyadicfn', 'fneval', 'releval',
                'rangeval', 'setform', 'brackform', 'tuplform'};
sinister    - {'ℓ' + x, x ∈ dexter} + {'lassign'};
loop        - {'forit'+ dec n, 1 ≤ ∀n ≤ 3};
```

We now give the main routine.

```
definef prodcode(a);
/* main routine of semantic resolution process */
<desig,ns,arglist,result> = a;
keys = using{desig,ns};
/* typical component of keys is <kindlist,polyarg,resultkind,
   ultrout> */
willdo = nℓ;
argkind = {arglist};
(while argkind ne nℓ and #willdo eq 0 doing argkind
        = revert(ns,argkind);)
willdo = {t, t ∈ keys|(∃s ∈ argkind|match(s,t))};
end while;
if #willdo eq 0 then
   print 'unable to find semantic interpretation for', a;
   return nult;  end if;
if #willdo gt 1 then print 'semantic ambiguity for', a,
   'all the following constructs apply', willdo;  return nult;
end if;
/* else code item can be built */
<kdlist,polyarg, resultkind,ultrout> = ∋willdo;
/* mark kind of result in dexter construction */
if desig ∈ dexter then revertf(ns,result) = resultkind;
```

```
if desig ∈ {fneval, dyadicfn, monadicfn}
    then return <callc,fneval> +
        <if ultrout eq Ω then arglist(1) else ultrout>
         + arglist(2) + <result>;;
if desig ∈ forit
    then return <callc> +
        <if ultrout eq Ω then <const, desig> else ultrout>
         + absarglist(arglist,desig) + <result>;;
end if desig ∈ dexter;

/* else desig ∈ sinister */
return if desig ∈ {lfneval, lydadicfn, lmonadicfn} then
   <lcallc,lfneval> + <if ultrout eq Ω then arglist(1) else ultrout>
         + arglist(2:);
else <lcall, if ultrout eq Ω then <const, desig> else ultrout>
         + arglist;
end prodcode;

definef match(arglist,eltkeys);
/* dexter, sinister, loop are global */
<kindlist, polyarg> = eltkeys;
return kindlist eq arglist(1: #kindlist)
   and((#kindlist lt #arglist)imp polyarg);
/* imp is the boolean operator imples */
end match;


definef absarglist(arglist,desig);
return if desig ∈ {'forit1','forit2'} then arglist(2:)
   else if desig eq 'forit3' then arglist(3:)
        else arglist;
end absarglist;
```

In the code that follows ns <u>knows</u> y is true if y is a variable known in ns.

```
definef revert(ns,setarg);
newkd = [union: x ∈ setarg, 1 ≤ ∀j ≤ #x] revertf{ns,x(j)};
return if newkd ne nℓ then newkd
    else if (ns knows ∋ setarg) then default(ns)
        else nℓ;
end revert;
```