SETL Newsletter #39 May 31, 1971

More detailed suggestions J. T. Schwartz

concerning 'data strategy' elaborations for SETL

This note will contain remarks concerning

i. SETL implementation and 31,

ii. The possibility (raised in newsletter 30, pp. 31-32, /pp.5-7) of securing increased efficiency by adding a system of optional elaborations to the language. These elaborations, if present, should increase running speed considerably and lead also to significant data compressions.

I. Main outlines of the ground to be covered. Implementation of 'collections'.

SETL allows finite sets to be entirely general and by this will often attain real advantages, both in expressive power and in that it will allow many problems concerning data structure to be postponed. Nevertheless, a survey of SETL algorithms shows that sets are generally used in quite stereotyped ways. The use-forms most frequently appearing seem to me to be as follows:

A. Sets used as maps. That is, sets f always referenced by addressing with a fixed number of indices, either in the sinister form

f(a,b,...) =

or in the dexter form

= ...f(a,b,...)...

B. Sets actually used as collections of elements; i.e., either as iteration controllers, 'workpiles', etc.

Let us, in order to focus our thoughts, examine uses of type B in more detail. Nine basic operations are supported: x ϵ a, a with x, a less x, ∂a, #a, random a, a eq b, a gt b, $\forall$x ϵ a (the iterator). We may also mention the compound operations a u b, a int b, a-b, etc., which can of course be defined in terms of the above operations, but

which can deserve special treatment for purposes of efficiency. Yet another very significant issue, set copying, enters at the implementation level but is hidden from the direct view of the SETL users. We may also mention the operation a lesf x, whose discussion however belongs rather under subheading A than under the subheading B with which we are now concerned.

We may consider a number of plausible techniques for the internal representation of sets.

a. Representation by linked linear lists

b. Representation by 'piling', i.e., by accumulation of elements within a delimited range.

c. Representation by trees, in which elements are located by a hash-assisted tree search, possibly with auxiliary bit-tables entering, so that several approximately equal sets may share a single tree .

d. Representation by some purer and possibly more efficient hash scheme.

e. Representation by bit-vectors, i.e., deliberately as subsets of some larger set.

The following table describes the general manner in which each of these treatments might affect the efficiency of the basic operations which must be supported.

Table estimating number of nominal cycles required to perform basic set operations, as function of number n of set elements

Technique / Operation	list	range	tree	hash	bit vector	footnotes
↛a	1	1, assuming range dense	1	1	1[a]	[a] assuming packing dense enough so that few words are identically zero.
#a	1 or n[b]	1 or n[b]	1 or n[b]	1 or n[b]	1 or n[b]	[b] depending on whether current count is kept
∀x ∈ a	n	n, assuming range dense	n[c]	n[c]	n	[c] assuming auxiliary list of members is kept.
x ∈ a	n	n	log n	<log n, possibly 1	1[d]	[d] assuming element directly represented by its bit position
a with x	1 or n[e]	1 or n[e]	log n	<log n, possibly 1	1[d]	[e] depending on whether equality with existing element must be checked
a less x	n	n	log n	<log n, possibly 1	1[d]	
a eq b	n log n[f] or n^2	nlogn[f] or n^2	1 (if not equal)	1 (if not equal)	n/P^g	[f] most efficient method involves sorting operation
a ge b	nlogn[f] or n^2	nlogn[f] or n^2	#b or n/P^g	1, n, or n/P^g	n/P^g	[g] where P is the average number of 1-bits per word
random a	n	1	log n?	n	n/P^g	
miscellaneous			copy $=n/W^h$		copy $=n/W^h$	[h] where W = number of bits per word, in bit or bit-assisted schemes
additional comments		good data density			not always available	[i] adaptable to fast search on first tuple component

Note also that in a relatively static situation, ranges might be sorted, or might be addressed by a specially contrived hash which allows elements present to be located in a specially small number of cycles.

These are the basic facts concerning the performance of various of the implementation methods which might be considered. Next let us consider the typical ways in which sets (when playing the 'collection' rather than the 'map' role) tend to be used. These seem to me to be as follows.

i. As a set with entries and deletions also used as a map. This is rare, but does occur; for example, in the nodal span parsing algorithm.

ii. As a set with frequent entries and specific deletions, but never used as a map (also rare).

iii. As a set being built up (possibly by the SETL set-former). Frequent entries which must be checked for identity with existing elements, but no deletions.

iv. As a static set, used for membership testing only. (Relatively rare.)

v. As a 'workpile'. Frequent entries, and deletions of 'unspecific' elements; i.e. using x from a rather than a less x. It may in some cases be certain that no duplicate elements can occur, and it may for other reasons be unnecessary to check for duplicates.

vi. As one of a family of sets frequently combined by particular higher level operations, e.g. union and intersection.

In classifying the usage-mode of sets, it is also quite important to try to take copying frequency into account. Copying will normally be necessary only when sets are changed subsequent to a point at which they are made members of other sets; this is fortunately a relatively infrequent case. It is vital, however, that the optimizing SETL compiler not be fooled into performing unnecessary copying operations.

Concerning the various uage modes described above we may make the following observations.

α. If a set is used in either of the 'general' modes i or ii, the tentative SETL hashed-tree approach ('tentative SETL mode') might perform as efficiently as the normal lower-level programmer's invention. The same remark applies to usage mode iii. In all three cases, the use of subsidiary bit vectors can bring advantages. This last remark applies to usage mode vi. Most of advantage of bit-vector schemes can probably be gained by the introduction of a method which explicitly forces a common reference-index scheme upon the members of several sets.

β. Usage iv can be accommodated either in the tentative SETL mode or by using a sorted range, possibly assisted by a specially contrived hash.

γ. The common usage v can be accommodated in any one of our proposed implementations; it is, however, most efficiently realized by the use of ranges. (Though most commonly programmed at a low level by using lists.)

We conclude that if SETL provides

1. Sets of tentative SETL mode;
2. a method of forcing a common reference-index scheme upon the members of several sets;
3. ranges, i.e. 'vectors' which may be entered by indexing, but which may be grown, i.e., are not of fixed or preallocated size;

and if these three devices are available at convenient user option, it should be possible to obtain good efficiency when using sets as collections.

Later certain additional variant storage forms aimed at efficiency in particular situations will be suggested.

Note here that lists implemented in the normal chained manner become unnecessary if ranges are available.

Next note that ranges are a very convenient and efficient mode in which to implement various of the other basic SETL objects and operations.

a. Variable length bit strings, variable length character strings. Ranges give dense packing, and easy indexed access to particular bits.

b. Multi-precision integers. Multiple precision arithmetic is implementable by quite fast loop.

c. Tuples. An n-tuple can be implemented by a range storing n elements. This gives fast indexing and last-element insertion. In fact, it assimilates the notion of a tuple to that of a squence, and pushes, in accordance with suggestions that have been made all along, to a different treatment at the SETL level, in which tuples would be systematically identified with sequences. In such a treatment

i. <a,b,c,...d>

would be a notation for the sequence whose terms are a,...,d;

ii. hd tuple would be an abbreviation for tuple(1) (as now)

iii. tℓ tuple would be <b,c,...d> ,i.e., would abbreviate tuple(2:#tuple) (as now)

iv. The construction tuple = <x,tuple> , which presently adds a new initial element to a tuple, would have a different meaning and its present usage would be abandoned. Instead, the form

tuple(#tuple+1)=x; equivalently newlast tuple=x;

would be used, as is presently the practice for sequences. The present multiple assignments

<a,b,c> = tuple

and

<a,b,c,-> = tuple

could however retain their syntactic form.

All of these possibilities depend upon our ability to implement "ranges" efficiently, i.e., to accommodate an indefinite number of 'arrays' able at least to grow at their upper boundaries; and I now turn to sketch a method which allows this to be done.

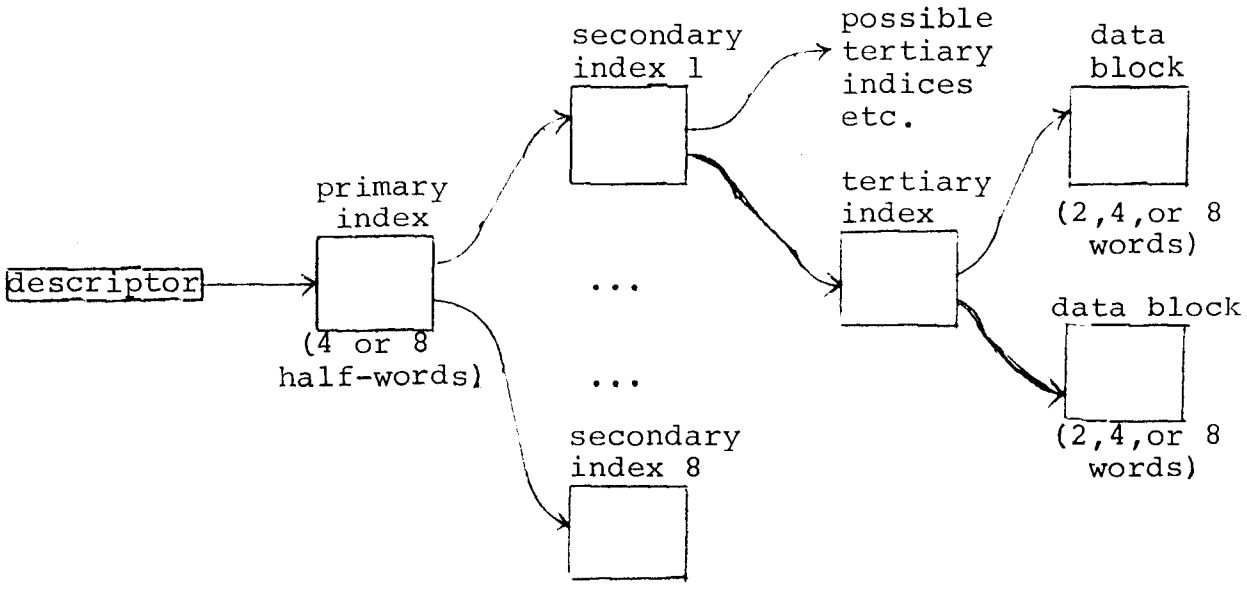
II. Implementation of 'Ranges.'

A plausible scheme for the implementation of ranges is as follows.

1. Words will be allocated in blocks of 1, 2, 4 and 8 by an allocator; the 'buddy' system described by Knuth can be used for this.

2. Each range will have a one-word range descriptor. This will give the range size (up to 128K or so maximum size), and if the range is of less than 8 words in size, will contain information defining the block size used and the defined-undefined state of various range elements. For larger ranges, this latter information will be contained in the primary range index, to which the range descriptor will point; in this case, the range descriptor will contain an integer field describing the number of levels of indexing used. The range descriptor may also contain a small amount of additional information as needed.

3. The structure of a large or fully expanded rnage will be as depicted in the following diagram(which shows a structure involving 3 levels of indexing.)



4. In structures like that shown above, the element R(I) whose index is I is located by the following procedure.

a. Calculate J = $I/8^{lev}$, I = $I//8^{lev}+1$ and (// denotes residue), where lev is the number of levels of indexing used. J then locates an entry in the primary index; which points to a secondary index. Repeat this division and chaining process through the 2^{nd}, etc. level indices, using I' in place of I at the 2^{nd} level, etc. until at the end of all indices the desired data item is reached.

5. We also can allow for the possibility that the range (think of it for the moment as a SETL sequence) is sparsely populated, i.e., that many of the elements R(I) are undefined. In this case, we may use compression as an additional technique, as follows. Associated (in an index) with the pointer to each data block will be a set of 8 bits; '0' bits indicate undefined values; '1' bits indicate defined values. The true location of the item expected in position j is the number of '1' bits found in positions 1 through j among those 8 bits. In this treatment, execution of a new definition R(I) = val may lead to the shifting of up to 3 items. (A slightly better 'double layer' compression might be worth exploring.) When more than 3 of 8 items are defined, compression can be abandoned.

If all the items in the data block which an index entry would reference become undefined, an 'undefined' flag in the index entry itself will be set, and the data block omitted. This flag may then be transmitted back to lower index levels, allowing, for very sparsely populated ranges, the elision of entire index blocks, etc.

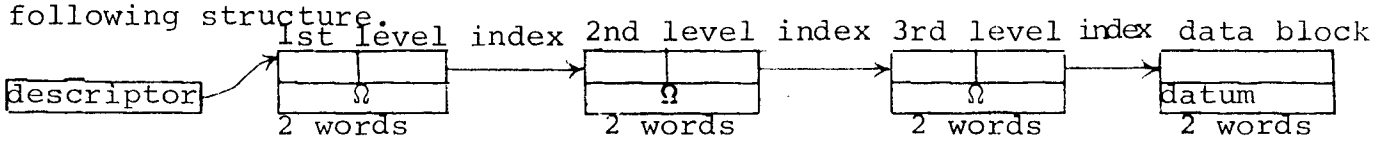
6. Space allocation and disallocation can work as follows. A reasonably current value for the maximum index I_{max} for which R(I) is defined will be kept in the range descriptor. When a new value R(I) is defined, an attempt will be made

access its storage location. At this time, any missing indices belonging to the access path will be allocated, and insufficiently large indices will be enlarged (from 4 to 8 half-words); this may involve the movement of index entries. When first created, an index block or data block will consist of two words; indices may be enlarged to 4 words, and data blocks to 8 words. Enlargement of a block will of course normally involve movement of data.

When previously defined values R(I) are made undefined by executing R(I) = Ω, it may only be necessary to change a flag in an index. On the other hand, this 'disallocation' will cause an examination to be made of the number of items in the same data block which are still defined. If this fails from 4 to 3, compression can begin and half of an 8-word data block can be returned to the allocator. If it falls to 1, half of a 4-word block will be returned. As appropriate, index blocks themselves may be compressed.

The scheme outlined is most typically wasteful of space when a randomly distributed 1/8'th of the elements of a large range are defined. This requires a full set of index blocks. Approximately 3 words are then used for every data item stored. Still sparser occupation of a range should not affect this ratio much. Of course, denser occuptation, in the manner normally expected in connection with ranges, leads to a density ratio close to 1.

An additional example: suppose that for a range, consisting initially of elements none of which are defined, the first instruction executed is R(3000) = 1. This creates the following structure.



Subsequent retrieval of the value of R(3000) requires 4 levels of indirection, and probabily some 20-30 cycles. Of course, for smaller ranges, this will be reduced.

Note also that proper optimization of storage/ retrieval loops involving successive indices I, especially with some associative hardware assist, might allow this scheme to perform almost as efficiently as standard indexing. Special operations which decoded the compression fields, etc. would in themselves serve to increase by a factor of approximately 5 the access efficiency attained by this scheme.

To summarize, the attributes to be associated with a 1-dimensional range are probably:

a) sparse (causes compression to be switched on)

b) item size (probably 1 for bitstrings, 6 for character strings, 30 for compound SETL objects, 30 for integers, 60 for reals).

It is now time to observe that the same techniques may be used to implement multi-dimensioned ranges with a fair degree of efficiency.

Multi-dimensioned ranges. A multi-dimensioned range is addressed by two or more indices, e.g. R(I,J). Each of the indices is an integer, neither has an upper bound. We handle this situation as if it were R(I)(J), i.e., I is used in the manner already described to locate a range descriptor, and J is then used as an index in this subrange. Here the typical effects on storage efficiency of 'sparseness' are as follows: if for most I at least one value of J is defined, then approximately 4 (or in the general case of a d-dimensional range approximately d+3) words will be used per true data item stored. On the other hand, if I is independently sparse, i.e., for most I no value R(I,J) is defined, then approximately 6 words (and in the

general case approximately 3*d words) will be used per true data item stored. Thus sparse multidimensioned ranges behave (for access and storage) in a manner not too different from sparse singly dimensioned ranges, e.g. a sparsely defined '60×60' array performs comparably to a sparsely defined range of I_{max}=3600.

Note that in the presence of additional information, there exist two evident ways in which access to a range may be made more efficient. If a range has attained its full size, it can be allocated, i.e., a contiguous, block of space sufficiently large to contain all its elements can be obtained from a central space allocator, all the entries in the range moved to this block, and the indices otherwise needed to support the range dropped. If the first index of a two dimensional range R(I,J) has a known variation, the range can be dimensioned, i.e., reduced to a one-dimensional range by the normal address transformation I' = I*dim+J. This makes the indexing chains needed to access a given range slightly shorter. The same remark evidently applies to 3, 4, etc. dimensional ranges.

Interesting mixtures of the 'range' technique suggested here and the paging -- segmentation techniques used to handle secondary storage probably exist. This deserves investigation.

III. Typical uses of sets as maps.

Next we turn to survey the typical usages of sets used as maps, i.e. sets always referenced, in dexter or sinister fashion, by addressing with a fixed number of indices. These seem to me to be as follows.

i. As stacks, growing at one end only, and always referenced at this end.

ii. As sequences, generally growing at one end, but in which any element is liable to be referenced using its (integer) index.

iii. As sequences of fixed size, or at any rate of a size not varying for many cycles of a SETL program. It is sequences

of this sort which correspond most closely to the arrays used in programming languages than SETL.

iv. As maps assigning attributes to the members of a fixed or growing set a. The 'symbol table' typically used within compilers and assemblers has this character. In this situation, several maps f,g, etc. may be 'associated', in the sense that values f(x), g(x) will normally be defined for all or most of the elements x ∈ a. Maps of this kind may be single-valued or multi-valued. The attributes, once defined, may be fixed, or may vary rapidly, as in the case of a map whose value is a 'count' associated in some way with a particular object.

v. As maps serving to define the structure of a data object, as for example 'next' pointers in lists or 'descendant' pointers in trees. This usage resembles usage iv, except that in this case the 'attributes' defined by the maps are the elements of sets appearing explicitly in a SETL algorithm, rather than being integers or other atoms having some essentially 'external' significance. In many cases of this sort, additional maps may serve to associate other attributes with the individual 'nodes' or 'links' of a compound data item.

vi. As maps f(p,j) of two indices, the second an integer, which serve to define families of sequences or of stacks depending on a parameter p. In cases of this sort, one may occasionally wish to refer to the set f{p} as a totality, perhaps only to calculate the number of elements which it contains.

vii. As maps serving to record, and to allow the rapid retrieval of, certain associations between pairs or triples of elements.

viii. As static maps, serving to assign fixed attributes to each of a fixed collection of elements.

ix. As maps serving to record some relationship, possibly boolean, between pairs or triples of elements, etc.; or as boolean-valued maps recording some true-false property of the members of a collection of elements.

x. As maps whose values are sets, which serve to give fast access to sets of particular importance in an algorithm.

xi. As maps depending on several parameters each of whose parameters has a relatively fixed domain, in which case a treatment like those customarily accorded to 'multiply dimensioned arrays' in programming languages of lower level may be appropriate.

xii. As maps depending on several parameters which together define a moderately or highly irregular domain, in which case efficient storage an access may require hashing techniques like those normally provided by SETL.

This list covers many of the most significant cases likely to be encountered. The SETL elaboration language to be described below aims, by adding optional statements to an algorithm, to make it possible for an optimizing compiler to produce a code considerably more efficient (both in regard to running speed and in regard to data space required) than might otherwise be possible.

The elaboration language will incorporate various devices, but one of these is of such central importance as to deserve special mention. Very often a program written in a language of lower level than SETL takes a key step toward efficiency by systematically referencing the objects with which it is concerned not directly but in terms of reference numbers associated with the objects in one or another manner. Thus, to take a typical example, a string S may be referred to, not by the characters of which it consists, but by its serial number j within some enumerated collection of strings. This basic device not only helps in the compression of data but, even more significantly, allows direct 'indexing' operations to replace more complex 'hashed access' operations systematically. For example, in the case considered above, the values of a function of S might be stored in an array accessed efficiently by using the associated index j. Knowing and systematically exploiting these 'representation conventions', a programmer working in a language of lower level can develop an efficient code. The elaboration language to be described below aims to make it possible to set down an explicit description of this normally implicit representation strategy, and in this way to attain a degree of efficiency that normally would require the use of a programming language of lower level than SETL.

IV. Tentative form for an elaboration language. Sets and ranges.

Guided by the above remarks, we shall now propose syntactic and semantic forms which can serve to define the elaboration language which we desire. Later in this newsletter, the use of this language for the elaboration of certain algorithms taken from the SETL notes will be illustrated.

The elaboration language will serve to define:

A. The manner in which certain maps are stored within ranges.

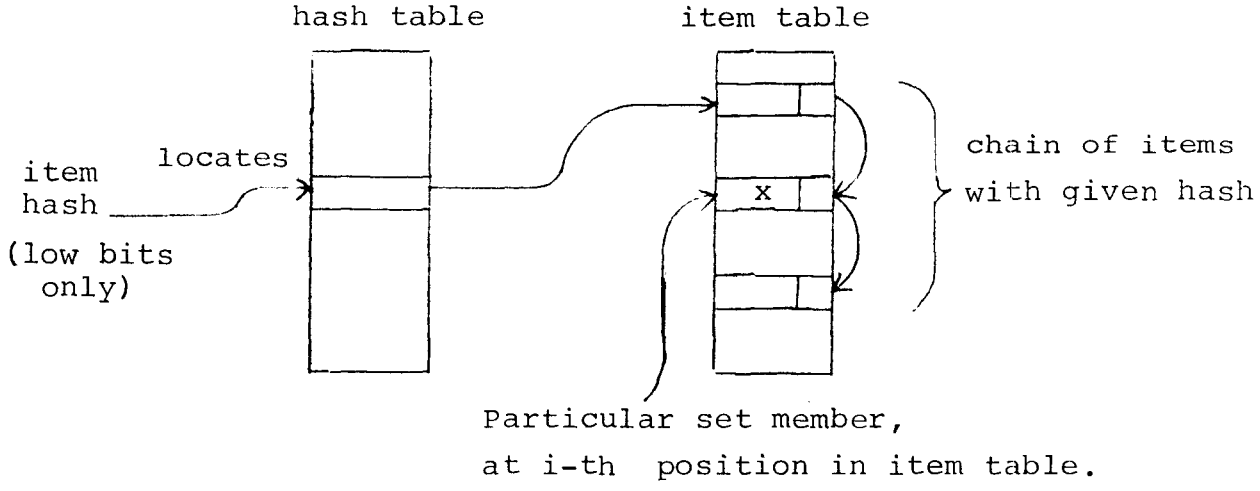
B. The manner in which ranges are addressed.

C. The manner in which certain sets are stored within ranges.

D. The manner in which reference is made to the elements of sets.

E. Various subsidiary operations and items of information, to be described below.

We begin with item D. We take it that, as elements are added to a set, the SETL implementation will assign them implicit or explicit 'reference numbers', in an essentially ascending order, as members of this set. To make this point clearer, suppose, for example, that sets are implemented using hashtables which point to entries in indexed ranges (i.e. ranges of the type considered in section II), and that these ranges contain chained lists of items, as in the following diagram.



The hash table and the item table shown in the above figure together represent a set a; the 'reference number' (as a member of a) of the item marked x in the figure, that is, of the ith item

in the item table, is i. Note that in this case the serial number i gives fast access to the item x, even if the item hash is not known initially.

Even if the eventual implementation of sets in SETL is somewhat different, we assume that for the elements of any set declared as a pile (see below for additional details) reference numbers having the principal properties noted above will be available. If x is a member of the set a, the quantity x aselt a refers to this reference number. A serial number of this kind, rather like a pointer in programming languages of lower level, retains its validity as long as x remains an element of the set a, and as long as the value of a is not reassigned by any operations more extensive than a = a with x, a = a less x. More radical reassignments might leave 'dangling reference numbers'; basically, we regard the avoidance of this situation as the programmer's responsibility, if he chooses to elaborate SETL.

The following syntactic discussion will begin to put the intended semantics of reference numbers into sharper focus. A set occurring in a SETL program may either be declared or undeclared. (In unelaborated SETL, all sets will be undeclared.) An undeclared set will be treated in a standard fashion; declared sets will generally be treated in a manner which is logically equivalent to this standard treatment, but which attains higher efficiency. A set a is declared if its name occurs in a set declaration (or in a range, see below for this possibility).

The first part of such a declaration, the so-called opener of the declaration, has one of the forms

(1) pile a;

(2) pile nodup a;

(3) hash a;

(4) set a;

The intended semantics of these set declaration openers are as follows. The declaration (2), i.e., pile nodup a, states that a is a set into which no two identical elements will ever be placed, or, at any rate, one in which the presence of duplicate members may be ignored. This declaration therefore suppresses the SETL identity-with-existing-element check which must normally be made whenever a new member is introduced into a set. Sets a declared in this way can be maintained in stack-like form (without any hash-table) particularly advantageous for the implementation of the ∀x ∈ a iterator and the x from a sequence. On the other hand, the operation a = a less x may have an inefficient implementation.

If a set a is declared in the pile nodup mode (2), then each element x of a retains a fixed reference number from the time it is introduced into a to the time that it is removed from a. The same is true if a is declared in the simpler pile mode (1). Note that this requirement may interfere somewhat with our ability to reclaim space by repacking the 'item table' shown in the figure above.

As has been noted, the declaration pile a serves to guarantee that elements of a will retain fixed reference numbers as long as a is not modified by any operation more radical than a = a with z or a = a less z. On the other hand, the addition of elements to a set declared in this mode will still involve an (efficient) check for identity with some existing element.

Note then that both the pile nodup and the pile declarations serve to guard the validity of element reference numbers, i.e., to ensure that two separate uses of x aselt a yield the same integer if x has not been changed and a has not been radically changed between these two uses.

The declaration hash a has much the force of pile a, except that if a is declared in this mode any reference x aselt a will add x to a if it is not already a member of a. Sets declared in this way will generally be used simply to systematize reference to elements of other sets, and to define the manner in which ranges (see below) are addressed.

The opener in set a merely states that a is to be treated in the standard SETL mode; the reason why such an opener may sometimes be required will shortly become clear.

Following upon the opener (1-4) of a set declaration, there will follow a content describer. This will have one of the following forms:

(5) obj;

(6) int, or more specifically int k;

(7) string, " string k;

(8) bit , " bit k;

(9) elt b , " elt b:k;

(10) subset b;

(11) tupl(...), where within parentheses there occurs a list of content describers, separated by commas;

(12) set cd , where cd is itself some content describer.

The semantic intent of this set of syntactic forms is as follows.

i. The describer obj signifies that the set a contains general items, represented in the standard SETL manner. This requires a field long enough to hold a standard-form SETL object reference.

ii. The describer a int signifies that the members of a are integers. For the representation of these integers, a field long enough to hold a standard SETL object-reference is required. However, if the integer value is small enough for its binary representation to be stored directly in such a field, this value may be given directly.

The describer a int k signifies tha the members of a are integers n confined to the range $|n| \leq 2^{k-1}$. A field of k bits suffices to store these values, which, in the case of sets declared as pile nodup, may lead to storage economies. An attempt to introduce an out-of-range element into a is an error.

iii. The describers string and string k : mutis mutandis from the preceding case, except that in the case of strings whose values are availble directly rather than indirectly one character will be reserved to indicate the length of the directly stored string. Similarly for the describers bit and bit k.

iv. The describer elt b signifes that the elements of a are also members of the set b, and that at the implementation level a is represented in a manner actually giving its elements in terms of their reference numbers as members of b. This serial number will be given directly, in a field large enough to contain the maximum plausible reference number. The describer elt b:k has a similar significance, but also certifies that the total number of elements added to b since its inception is certainly less than 2^k. Reference numbers to elements of b may therefore be stored in a field at most k bits long, which can lead to storage economies if a has been declared as pile nodup.

v. The describer subset b signifies that the elements of a are subsets of a set b, and these subsets are represented by bit-vectors, the bit-position corresponding to any element of such a subset being determined by its serial number as an element of b.

vi. The describer tupl(cdseq), where cdseq is a sequence of content describers separated by commas, asserts that the elements of a set are n-tuples of fixed length, and that the components of these n-tuples are of the type described by the successive

members of the describer sequence cdseq. Fields of appropriate length, coded in the manner determined by cdseq according to the conventions set forth in the last few paragraphs, are then reserved for the representation of these components.

vii. The describer set cd, where cd is itself some content describer, indicates that the elements of a set a are themselves sets; cd describes the nature of the elements of these sets.

After the opener and the content describer of a set there may follow an optional iteration describer. This can be of one of the two following forms:

(13) iter asobj;

(14) iter aselt;

These describers control an aspect of the SETL iterator (∀x ∈ a) which is important for efficiency. If a set is declared with the iteration describer (14), then in the implementation of an iterator of the form (∀x ∈ a) the variable x will run iteratively over the reference numbers of the elements of a (i.e., their reference numbers as elements of a), rather than over the corresponding sequences of standard SETL object references. If no iteration describer is present, then in an iterator of the form (∀x ∈ a) the variable x will run iteratively over the elements of a, represented however not as standard SETL object references but in whatever way the content describer used in a's declaration causes them to be represented. Thus, for example, if a is declared as

pile a; elt b;

then in an iteration (∀x ∈ a) the values successively assigned variable x will be the reference numbers which members of a have as elements of b. On the other hand, suppose that a is declared as

pile a; elt b; iter aselt; .

Then, even though the representation of a (at the SETL implementation level) will describe the elements of a using their reference

numbers as elements of b, the iterator (∀x ∈ a) will cause x to assume successive values all of which are a-reference numbers. If it subsequently becomes necessary to reduce x to a standard SETL object reference, two successive layers of indirection will have to be resolved.

The iteration describer (13), i.e., iter asobj, forces the iterator (∀x ∈ a) to set x successively to a sequence of standard form SETL object describers, irrespective of the manner in which a stores its elements, i.e., independently of the content-describer occurring in the declaration of a.

Having said enough for the moment concerning item D of the list given at the beginning of the current section we turn to item B, i.e., to describe the manner in which logical ranges can be employed to represent sets used as maps in a SETL algorithm, and the manner in which these ranges can be addressed.

A logical range is defined by a range declaration.

1. The first part of this, which is the addressing declaration for the range, opens with a statement of the form

(15) range name(x_2,x_2,...,x_n);

where name is the range name, and where x_1,...,x_n are the parameters using which the range will be addressed. We call these its addressing parameters, and call the statement (15) the declaration opener.

2. Next there follow a set of parameter description units, separated by commas. These have one of the following forms

(16) i. x_i int, x_i blank index, x_i bit index, x_i string index;

ii. x_i aselt a, or more generally $\langle x_{i_1},\ldots,x_{i_k}\rangle$ aselt a ,

iii. x_i hash a, " $\langle x_{i_1},\ldots,x_{i_k}\rangle$ hash a ,

iv. x_i hash , " $\langle x_{i_1},\ldots,x_{i_k}\rangle$ hash .

Here a is a SETL name; the value associated with this name will be some set. Every parameter of the range must be mentioned in precisely one parameter description unit.

The semantic intent of the syntactic forms shown above is as follows. The SETL range declared in (15) will be implemented as a singly- or multiply-dimensioned indexed range R of the kind described in section II. The number of dimensions of R will (for a reason that should soon be clear) equal the number of parameter description units (16) that follow the opener (15). Each access to R is of course indexed by integers; the parameter description units serve to describe the manner in which the indices supplied to the logical range name are converted into integers to be transmitted to the indexed range R. More specifically

i. If x_i is described as an int, it will be an integer to be transmitted directly to R.

If x_i is described as a blank index, it will be an integer whose 'least significant' part is to be transmitted directly to R. The 'most significant' part will be an additional integer serving to identify a blank atom uniquely. Indices of the form here envisaged will be produced by calls on the elaborated newat function; see below.

If x_i is described as a bit index, it will be a bit-string, and the integer bin x_i is then to be transmitted to R.

If x_i is described as a string index, it will be a character string, and the integer bin hol x_i (cf. newsletter 34, p. 1) is then to be transmitted to R.

ii. If the declaration x_i aselt a occurs, then x_i is an element of a represented either directly or indirectly, and quite possibly by the reference number which x_i has as a member of a. In this case, this reference number, i.e., the integer x_i aselt a, is to be transmitted to R.

If the declaration $\langle x_{i_1},\ldots,x_{i_k}\rangle$ aselt a occurs then the indicated k-tuple, formed from the group of parameters $x_{i_1},\ldots,x_{i_k}$, is a member of the set designated by the name a. In this case, the serial number $\langle x_{i_1},\ldots,x_{i_k}\rangle$ aselt a is to be transmitted to R; this reference number will of course stand for an entire group of parameters of the logical range name.

iii. If the declaration x_i hash a occurs, then a names a set not used in an unelaborated SETL program, but which is to be used to supply an index to be transmitted to R. The element x_i will then be inserted into the set a, and the serial number x_i aselt a is supplied to R. The declaration $\langle x_{i_1},\ldots,x_{i_k}\rangle$ hash a has a similar meaning, which the reader will readily supply.

Several separate indices x_i may occur in addressing declarations x_i hash a with fixed a, either in one or in several separate ranges; likewise, several separate groups of indices may occur. The set designated by a name a used in this way will continually increase, unless all the ranges in which a is so used are simultaneously drop'ed (see below), in which case a will be reset to nl.

iv. The simplified forms x_i hash and $\langle x_{i_1},\ldots,x_{i_k}\rangle$ hash are respectively equivalent to x_i hash aa and $\langle x_{i_1},\ldots,x_{i_k}\rangle$ hash aa, where aa is some unique generated name.

3. After the opener and the parameter description units forming the first part of a range declaration follows a group of storage description units. These are introduced by the token

stores

and have the form

(17) f contentdescriber ,

where contentdescriber is a content describer of one of the forms described earlier, and where f is the name of some set used as a mapping within a SETL program.

A set whose name occurs in a storage description unit (17) of some logical range r is said to be stored within r; the precise manner in which its values are represented within r is determined by the contentdescriber of (17), in the manner explained in our earlier discussion of the semantic significance of content describers.

Note that a given set name f may not be mentioned in more than one storage description unit, and not in two storage description units belonging to different ranges.

4. After the opener, parameter description units, and storage description units of a range declaration follow an optional group of sharing assertions. These are introduced by the token

share ,

followed by a list of sharing groups, enclosed in parentheses and separated by commas. Each sharing group has either the form

(18) (f,g,...)

or

(19) (f,g,...)k .

Here f,g,... are set-names mentioned in a storage description unit of the range declaration. A given set-name may not be contained in more than one sharing group.

The occurrence of a sharing group (19) has the following significance. Not more than k of the functions f,g are likely to be defined for any given set of logical parameter-values. A field large enough to store k entries of the type appropriate to the declared value-types of each of these functions, allowing a 'typical average' field size if the field-sizes specified (in the preceding storage description units) for these various functions differ. In addition, a number of bits are reserved for flags which indicate which of the values f,g, etc. are defined and which are undefined. If more function values become defined than will fit into the total field, one passes automatically to a variant storage technique, in which one or more overflow areas belonging to the indexed range R are used.

The special variant (18) is roughly equivalent to

(f,g,...)1 ;

it reserves a field equal in size to the maximum size of the fields required to store f,g, etc.

If a range is to be used to store the values of any mapping $f(x_1,...,x_n)$ which might be multi-valued, its opener should have the special form

multi range name$(x_1,...,x_n)$;

This indicates to the SETL compiler that an implementation efficient for cases of this kind is required.

If the address-transformations to be used with a range are not such as to guarantee that the numerical indices into which addresses will be transformed fall into a densely populated interval, the keyword

sparse

should be prefixed to the range opener. This will guarantee that the indexed range R within which the logical range name is stored employs compression, which, in the manner described in section II, will attain acceptable data packing even in the sparse case.

V. The elaboration language, continued. Conversions, dimensioning, range operations.

The SETL elaboration language, as described so far, allows sets and their elements to be referenced in various ways. E.g., an object can be represented by a standard SETL object reference, by a number which specifies one such object reference among all the elements of a particular set, etc. We intend in this connection that conversions (between the various possible representations of a single object) should be postponed as long as possible; this principle can be of fundamental importance in attaining high efficiency. Thus, for example, if a is a set declared as

pile nodup a; string; iter aselt;

while f and g are functions stored in the manner described by

(20) range fg(x); x aselt a; stores f int, g obj;

then in the code sequence

(21) (∀x ∈ a) if f(x) gt 3 then ...

the iterator will set x equal to the reference numbers of successive elements of a; since these reference numbers are precisely the indices which serve to locate the various values f(x) required within the code sequence (21), a suitable optimizer can convert (21) into efficient code which uses 'direct indexing', even though from the abstract SETL point of view x remains a character string of unrestricted length.

If, on the other hand, f and g were stored in the manner described by the declaration

(22) range fg(x); x hash; stores f int, g obj;

then in the code sequence (21) the evaluation of f(x) would require a 'conversion'. Specifically:

i. The reference number of x as a member of a initially available when f(x) is to be evaluated, must first be converted into a standard SETL object-describer;

ii. This object-describer will then be sought in the 'hash-table' implicitly specified in the declaration (22); if not already present in this table, it will be inserted.

iii. The serial position of x's object-describer in this hash table will then serve as the index locating the actual value of f(x).

Effective use of the elaboration language requires the design of a set of declarations which minimize the number of conversions and other 'overhead' operations appearing in the compiled form of SETL code.

The elaboration language contains various statements designed to aid the programmer's in controlling the mode in which objects occurring in a SETL code are referenced and stored. The expression

(23) x aselt a

converts an object into its reference number as an element of a. If x is not a member of a, (23) is erroneous unless a has been declared to be a hash; in this case. (23) adds x as a new member of a.

The expression

(24) x asobj

converts a reference number into the standard SETL object-describer which this reference number represents. It must be used in those cases in which an object described by a reference number will 'outlive' the set used to assign it a reference number.

These same keywords can be attached to iterators. By writing

(25) (∀x ∈ a asobj)

we cause the successive values of x to be standard SETL describers corresponding to the successive members of a, irrespective of the manner in which a may have been declared. Similarly, by writing

(26) (∀x ∈ a aselt b) ,

we cause the successive values of x to be the reference numbers which the elements of a have as elements of a (possibly different) set b. The diction

(27) (∀x ∈ a ashash b)

has a similar significance, which the reader will readily supply.

A related convention applies to the SETL set-former construction. The unmodified set-former

(28) $\{x \in a \mid C(x)\}$,

together with its more elaborate variants, creates a set in standard SETL form. On the other hand, if b is a declared set, then

(29) $\{x \in a \mid C(x)\}$ inr b

builds the same set in the special, and generally more efficient, form declared for b; when the set-forming opeation concludes, b will have been made equal to the set (28). A similar convention applies if b has been declared as a set of n+1-tuples used as a mapping and stored within a logical range name. In this case, it is required that the set (28) consist of n+1-tuples of corresponding form; the evaluation of (29) then makes b equal to the set (28). Of course, b is maintained in its declared form. Any attempt to place an element x whose form is logically incompatible with a given range into a set stored within the range is an error.

Remarks similar to those just made apply to various other set-creating operations. For example, if a has been declared to be stored within a range the statement

(30) read a;

will convert a directly from its external form to the special, efficiently represented form declared for a . The evaluation of

(31) $\{x \in a \mid C(x)\}$ assub b ,

or of any such more complex variant of (31) as

(32) $\{e(x,y),\ x \in a,\ y \in aa(x) \mid C(x,y)\}$ assub b ,

builds a new set, representing it as a bit-vector, in which the bit position of an element is its reference number as an element of b. Here b should have been declared as pile, hash, or possibly pile nodup. The set formed in (31) or (32) must be a subset of b, unless b has been declared as hash, in which case elements

present in the set (31) but not initially present in b will be added to b as necessary. A similar remark applies to such an instruction as

(33) read a assub b;

whose semantics the reader should be able to supply without difficulty.

Assignments

(34) a = expn;

operate as follows in the presence of SETL elaborations. If the set name a is undeclared, i.e., if no declaration requiring a to be stored within a pile, heap, or range has been made, then after (34) has been executed the value of a will be stored in whatever manner is dictated by the right-hand side of (34). This accords with our general desire that conversions should be postponed as long as possible. Thus, for example, if no special storage mode has been declared for a, the assignment

(35) a = b assub c;

will cause a to reference a bitvector, whose 1-bits correspond of course to those elements of c which belong to its subset b. Similarly, the assignment

(36) a = f(x);

would in the presence of a declaration of the form

(37) range fg(x); x int; stores f,g aselt b;

cause the value of a to be a reference number corresponding to some certain element of the set b; provided, that is, that no special storage mode has been declared for a.

If, on the other hand, a special storage mode is declared for a, then the assignment (34) implies a conversion of the value expn into whatever form this storage mode requires.

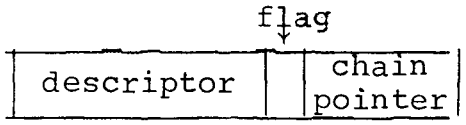
Of course, if the value expn already has the correct form, conversion is unnecessary and the operation (34) is maximally simple. On the other hand, if the value expn is such as to make its conversion to the declared storage mode of a impossible, an error results.

The same rules carry over, in appropriate form, to indexed assignments, read operations, and so forth.

If a is a set used within a SETL algorithm, then the (executable) instruction

(38) pin a

converts a from its normal form (whether undeclared or declared) to a special form advantageous for use in connection with sets to which additions and deletions will be addressed only rarely. One possible form in which such a pinned set might be maintained is as follows. A block, equal in size to the number of elements of a, can be set aside. This block can contain entries of the form



Items are located in the pinned set using an index derived by hashing. The single 'flag' bit shown in the above diagram is set to 1 for items that are valid table entrances, and to zero for all other items. The 'chain pointer' field connects all items which have identical initial indices; all but the first of these entries have their flag bit set to zero. The 'descriptor' contains the item reference itself, maintained in whatever form may have been declared for the entire pinned set a of items. Individual items within a collection maintained in this form may be located quite rapidly. Note that the membership test x ∈ a and the iteration (∀x ∈ a) can be implemented with particular efficiency for pinned sets.

If enough items are added to a 'pinned' set a to cause it to overflow the area reserved for its representation, an error condition will arise. Thus, before adding new items to a pinned set, one should execute the instruction

(39) unpin a;

this returns a to its normal representation.

When a set a is pinned (or unpinned) the reference indices associated with its members all change. Thus pinning/unpinning of a is only possible at moments when no range within which particular parameters or stored values are referenced aselt a contain any entries. One way in which this restriction can be accommodated is as follows. A function f whose values are eventually to be indexed aselt a, where a is to be pinned, can be represented in the standard SETL form before a is pinned, (which means that no special storage mode should be declared for f). After a is pinned one may execute ff = f; followed by f = ff;. Here ff should be a mapping whose values are declared to be stored within a range indexed aselt a. As has been noted above, the first of these assignments will cause f to be converted from the standard SETL form to whatever special form is declared for ff; the second assignment will transmit this converted form back to f. Since we assume that no special storage form has been declared for f, this second assignment does not imply any special conversion. Note also that the restrictions encountered here relate to the fundamental fact that sets f used as mappings and stored within ranges cannot be exactly reconstructed in their entirety, since the special form in which they are maintained, which is contrived to support indexed assignment and retrieval operations correctly and efficiently, suppresses some of the ('reverse-direction') connections between indices and the elements which they index which would be necessary for this purpose. We also remark that, if the reconstruction of maps is associated with it, pinning can be an expensive operation, and one may therefore prefer to pin sets only when they will remain unchanged through a substantial sequence of SETL operations.)

If a logical range r is used to store the values of only one single map f, the instruction

f = nl;

will be efficiently implemented; essentially, it will cause the range r to revert to an initial 'null' condition. If, on the other hand, r is used to store the values of only several maps, execution of the above instruction may be quite expensive, as it may involve an examination of every item stored in r. In such cases, it may be preferable to execute the composite 'range nulling' operation

drop r;

This is logically equivalent to the sequence of instructions

f = nl; g = nl; ... etc.

where f,g,... are all the maps whose values are declared to be stored within r. However, the drop instruction always has an efficient implementation, and causes r to revert to its initial 'null' condition.

Note that the possibility of using the drop instruction in this way may influence the programmer's grouping of functions to be stored within a common range.

If a reasonable maximum size can be assigned on a priori grounds to a declared set or range, the set or range can be allocated, using the executable instruction

(40) a alloc n ,

where a is a declared set or range name, and where n is an integer. This operation reserves for the storage of a contiguous block large enough to contain n entries, and thus facilitates access to a by eliminating some of the overhead associated with the maintenance of variable-size objects. If subsequent to the execution of (40) enough entries to a are made to cause a to overflow its allocated area, a will automatically revert to the standard format used to implement entities of variable size.

A related operation which can be called dimensioning may be worth providing in connection with arrays. A plausible syntax for this operation might be as follows. Suppose that a range r is declared in some such form as

(41) $\underline{\text{range}}\ r(x_1,\ldots,x_n);\ \ldots\ .$

In this case, the instruction

(42) $\underline{\text{dim}}\ r(x_{i_1}:n_1,\ldots,x_{i_k}:n_k)\ ,$

in which $x_{i_1},\ldots,x_{i_k}$ are parameters appearing in (41) and $n_1,\ldots,n_k$ are integers, may be executed. The effect of this instruction can be explained as follows. Whenever a value stored in the range r is accessed, the parameters $x_1,\ldots,x_n$ are converted to integer indices $p_1,\ldots,p_n$, in the manner described earlier in the present newsletter. These integer indices are then used to access an element in an indexed range R of the type described in section II above. This accessing opeation may involve a sequence of steps, during which successive indices $p_1,\ldots,p_n$ are converted into machine addresses with the help of whatever auxiliary index tables associated with R may be required. The instruction (42) speeds up this addressing process by converting the subcollection $p_{i_1},\ldots,p_{i_k}$ of indices to a single 'linearized' index p, this index p being calculated by the customary formula

(43) $p = p_{i_1} + (p_{i_2}-1)*n_1 + (p_{i_3}-1)*n_1*n_2+\ldots+(p_{i_k}-1)*n_1*\ldots*n_{k-1}.$

(Clearly, the last integer n_k appearing in (42) is irrelevant; it may be omitted.) It is only appropriate to apply the index-transformation (43) if it is known a priori that the variation of the individual indices $p_{i_1},\ldots,p_{i_k}$ is confined to the intervals

(44) $1 \le p_{i_1} \le n_1\ ,\ \ldots\ ,\ 1 \le p_{i_k} \le n_k\ .$

It may also be remarked that the use of the transformation (43) will normally be associated with situations in which the intervals (44) are densely (rather than sparsely) populated with indices generated in addressing the range (41). Once a range (41) has been dimensioned using (42), the occurrence of an excessively large or nonpositive index p_j constitutes an error.

The form (42) of dimension statement can only be used if the addressing part of the basic declaration (41) contains no statements of the form

(45) $\quad <x_{j_1},\ldots,x_{j_n}>$ aselt a

or $\quad <x_{j_1},\ldots,x_{j_m}>$ hash , etc.

which cause a group of range parameters to be used collectively for the generation of an index. If such forms as (45) are used in the addressing part of (41), and if it is desired to use the numerical indices thereby generated in forming a linearized address p as in (43), the instruction (42) should have some such modified form as

(46) $\quad$ dim $r(x_{i_1}:n_1, <x_{j_1},\ldots,x_{j_m}>: n_2, \ldots)$.

The instruction

(47) $\quad$ undim r

causes the range r to revert to its initial 'undimensioned' form; this might be necessary if, for example, indices p_j not belonging to the stated intervals (44) might be encountered in a particular section of SETL code.

Both (42) and (47) will normally be expensive instructions to execute, and one will normally use them only when the dimensions established by executing (42) need not be changed until a substantial number of references to r have been made.

VI. The elaboration language, concluded. Miscellaneous conventions, remaining deficiencies.

The SETL elaboration language will incorporate certain conventions related to the blank atom generator function newat. If r is the name of a logical range of one integer parameter, i.e., a range declared in some such form as

(48) range r(x); x int; ...

then newat r denotes an integer (probably the smallest integer) for which every function-value f(x) stored in the logical range r is undefined. In effect, the function call newat r 'allocates' one entry E of the standard form declared for r, within r flags all the values stored in this entry as being undefined, and returns the location of E as its value. Indexed assignments of the form f(x) = expn may then assign values other than Ω to certain of the logical 'storage fields' of E.

An alternate elaboration of the newat function is provided for use in those cases in which one wishes to generate sequences of blank atoms which can serve initially to address a logical range r but which may 'outlive' drastic changes in r. This has the form

(49) newat ofseq a,

in which a itself is a blank atom. This form of blank atom generator works as follows. As blank atoms are generated, they are assigned identifying sequence numbers long enough to prevent inadvertent duplications from occurring. The form (49) causes this sequence number of have high bits calculated from the low bits of a, but to have low bits which range sequentially from 0 to some large integer. This low bits can then be used directly to index a range; which will be done if some parameter of the range is declared to be addressed a blank index.

I will not now attempt to suggest detailed semantic conventions concerning the automatic copying of sets declared as pile or hash, or concerning the conventions to be applied when such sets appear within routines used recursively. A study of these important questions will be undertaken later. Hopefully, the SETL optimizer will be able to discover enough of the cases in which the copying of an entire set is unnecessary so that a single fully automatic

copying system can be applied both to sets for which no special storage form is declared and to sets declared as pile, pile nodup, or hash. If this hope is not realized, the elaboration language may have to include some explicit 'copy' instruction applicable to declared sets. A similar remark applies to ranges and to maps whose values are declared to be stored within a range.

We assume here that any storage declaration made for a set or mapping a has the same scope as the name a itself. Thus the storage mode declared for a will apply consistently to a whenever it is referenced by a subroutine used in a larger SETL program. This rule is to be applied even when a is made available as a parameter to a subroutine or function.

Consideration of the examples to be presented in the following section reveals a number of shortcomings in the elaboration language as it has just been outlined; among these, the following deserve to be noted.

i. No form is available for specifying either that the elements of a set or that the elements stored in a particular 'field' within a range are tuples, of indeterminate length, but all of whose components are of a particular type. This may deprive an optimizer of information of which it could make good use.

ii. No 'union type' like that of ALGOL 68 is available. This means that if an item to be stored in a range exhibits any structural variability whatsoever, it must be declared as obj, which is an entirely general declaration, and which may deprive an optimizer of information of which it would make good use.

iii. The elaboration language as it stands provides no mechanism whereby the items of a doubly indexed range d(x,j), whose second index is an integer varying over an interval $1 \leq j \leq n(x)$, can be 'stacked' into a single linear range in a manner calculated to make indexing efficient. Note however that the 'pinning' of such a range might be implemented in a manner accomplishing this.

iv. In some cases a doubly indexed range d(x,j) and a singly indexed range f(x) will be associated, in the sense that the indices x are the same for both ranges, and that the second index j occurring in one of these ranges is an integer varying over an interval $1 \leq j \leq n(x)$. The elaboration language contains no mechanism for expressing this association, a fact which may have certain intransparencies of expression among its consequences, and which may also lead to certain inefficiencies in data packing and indexing.

VII. Use of the elaboration language: some illustrations.

We now illustrate the use of the elaboration mechanisms proposed in the preceding pages, applying them to various algorithms taken from the SETL notes. Our first example is the tree selection sort, given on page 112. We modify the unelaborated algorithm slightly so that it applies to the sorting of other objects than integers (as, for example, strings); indicate all I/O explicitly for the sake of completeness; and write the elaborated and unelaborated algorithms in parallel columns to make the elaborations stand out vividly. I have attempted to write a set of elaborations from which a fairly straightforward, optimizer could produce reasonably efficient code.

PURE SETL ALGORITHM

```
read seq;

treesort(seq) ; print seq;
  exit;
definef treesort(seq);
/*first build the tree*/




ℓ=nℓ; r =nℓ;
v={<newat,seq(j)>,1≤∀j≤#seq};
trees = hd[v];

loop: newtrees=nℓ;

(while trees ne nℓ)
nd = newat;
ℓn from trees;
rn from trees;
nd in newtrees;
```

ELABORATED ALGORITHM

```
read seq;
range seq(x) ; x int; stores seq obj;
treesort(seq) ; print seq;
  exit;
definef treesort(seq);
/*first build the tree*/ pin seq;
range tnodes(x); x int;
 stores ℓ,r,par int, v obj;
drop tnodes;
ℓ=nℓ; r=nℓ;
v={<newat tnodes,seq(j)>,1≤∀j≤#seq};
trees = hd[v];
pile nodup trees ; int;
loop: newtrees=nℓ;
pile nodup newtrees; int;
(while trees ne nℓ)
nd = newat tnodes;
ℓn from trees;
rn from trees;
nd in newtrees;
```

```
ℓ(nd)=ℓn;v(nd)=v(ℓn);
iff (rn ne Ω)?
rnlit? quit,
setrt, setlf;
setlf: r(nd)=rn;
setrt: ℓ(nd)=rn;
r(nd)=ℓn;v(nd)=v(rn);
rnlit := smaller(v(rn),v(ℓn));
end iff;
/*'smaller' is a macro
 to be supplied. it specifies
 the ordering principle for
 the objects to be sorted*/
/* for integers it would be*/
block smaller(v1,v2);
 v1 ℓt v2; end smaller;
 end while trees;
if #newtrees gt 1
 then trees=newtrees;
go to loop;

/*then put in parent links*/
par=nℓ;

(∀x ∈ hd[ℓ])par(ℓ(x))=x;;



(∀x ∈ hd[r])par(r(x))=x;;
/* now tree is built.
begin main selection and
repair process */
top = ϶newtrees;
seq = nℓ;
```

```
ℓ(nd)=ℓn; v(nd)=v(ℓn);
iff(rn ne Ω)?
rnlit?  quit,
setrt, setlf;
setlf: r(nd)=r(n)
setrt:ℓ(nd)=rn;
r(nd)=ℓn; v(nd)=v(rn);
rnlit := smaller(v(rn),v(ℓn));
end iff;
/* same comment */

/*same comment */
block smaller (v1,v2);
v1 ℓt v2; end smaller;
 end while trees;
if #newtrees gt 1
 then trees=newtrees;
go to loop;
pin tnodes;
/* same comment */
/* this may be omitted, because of
previous drop of tnodes */
(∀y ∈ℓ) par(tℓ y)=hd y;;
/* which generates a loop covering
the whole range 'tnode'*/
(∀y ∈ r)par(tℓ y) = hd y;;
/* same comment */


top = ϶newtrees;
seq = nℓ;
```

```
(while ℓ(top) ne Ω) node=top;
(while ℓ(node)ne Ω)
  node = ℓ(node);;
seq(#seq+1)=v(node);
ℓ(par(node)) = Ω;
(while par(node)ne Ω)
node = par(node);
iff (r(node) eq Ω)?
(ℓ(node) eq Ω)? isnoleft?
takeleft,dropnode,
        takeright, compare;
takeleft:v(node)=v(ℓ(node));
dropnode:ℓ(par(node))=Ω;
takeright:v(node)=v(r(node));
ℓ(node)=r(node);
r(node)=Ω;
compare:
if smaller(v(r(node),
 v(ℓ(node)))
 then ℓ(node)=r(node);
r(node)=ℓ(node); end if;
v(node)=v(ℓ(node));
isnoleft:=ℓ(node) eq Ω;
end iff; end while par;
seq(#seq+1)=v(top);
  end while;
return;
  end treesort;
```

```
(while ℓ(top) ne Ω) node=top;
(while ℓ(node) ne Ω)
  node = ℓ(node);
seq(#seq+1)=v(node);
ℓ(par(node)) = Ω;
(while par(node) ne Ω)
node = par(node);
iff (r(node) eq Ω)?
(ℓ(node) eq Ω)? isnoleft?
takeleft, dropnode,
        takeright, compare;
takeleft:v(node)=v(ℓ(node));
dropnode:ℓ(par(node))=Ω;
takeright:v(node)=v(r(node));
ℓ(node)=r(node);
r(node)=Ω;
compare:
if smaller(v(r(node)),
   v(ℓ(node)))
 then ℓ(node)=r(node);
r(node)=ℓ(node); end if;
v(node)=v(ℓ(node));
isnoleft:=ℓ(node) eq Ω;
end iff; end while par;
seq(#seq+1)=v(top);
  end while;
return;
  end treesort;
```

It will be observed that relatively few elaborations need be made; hopefully, these will suffice to give good efficiency.

Next we show the elaboration of a SETL code which as it stands is more highly set-theoretic than the preceding algorithm: the set of procedures for flow-analysis and live-dead tracing given in the SETL notes. Again, we write in two parallel columns so that the unelaborated and elaborated algorithms may be compared.

PURE SETL ALGORITHM

```
/*hypothetical 'main
 optimizer program'*/
/* it is assumed that an initial
 'successor function', set of
 nodes, and 'uses' function
 is given, each value of this
 last function being a subset
 of some  comprehensive set of
 variables */
builduse(nodes,entry);
```

```
/* now we begin the series of
 subroutines which build up the
 principal processes used above*/
definef interval(x);
optimizer external cesor,
 nodes, npreds, followers;
int = nl;
followers = {x};
count = {<y,0>, y ϵ nodes};
/* except for x, 'count' counts
 the number of predecessors of
 a node which belong to the
 interval being constructed*/
count(x)=npreds(x);
(while {y ϵ followers |
 npreds(y) eq count(y)}
 is newin ne nl)
```

ELABORATED ALGORITHM

```
/* same comment */

/* same comment */

/* the set of variables will
 be called 'vars'
 below */
builduse(nodes,entry);
pile nodup nodes; elt allnodes;
range cesor(x); x aselt allnodes;
 stores cesor set (elt allnodes);
range npreds(x);x aselt allnodes;
 stores npred  int 12;
hash allnodes; obj;
/* same comment */

definef interval(x);
optimizer external cesor,
 nodes, npreds, followers, allnodes;
int = nl;
followers ={x} aselt allnodes;
count = {<y,0>, y ϵ nodes}
/* same comment */
range int(j); j int;
stores int int; range count(x);
x aselt allnodes; stores count int12;
count(x) = npreds(x);
(while {y ϵ followers |
 npreds(y) eq count(y)}
 is newin ne nl)
pile nodup newin; elt allnodes;
```

```
(∀z ∈ newin) int(#int+1)=z;
followers = followers less z;
(∀y ∈ cesor(z) | y ne x)
count(y) = count(y)+1;
y in followers;;
end ∀z;
return int;
end interval;
definef intervals(nodes,entry);
optimizer external cesor,
 followers, follow, intof;
ints = nℓ;


seen = {entry};
(while seen ne nℓ)
node from seen;
interval(node) is i in ints;
follow(i) = followers;



(∀b ∈ tℓ[i] /*tℓ[i] is
 the set of all nodes in i*/ )
intov(b) = i;;


seen = seen u followers;
end while;
return ints;
end intervals;
/* now the derived graph
 algorithm */
definef dg(nodes,entry);
optimizer external cesor,
 follow, intov, dent;
```

```
(∀z ∈ newin) int(#int+1)=z;
followers = followers less z;
(∀y ∈ cesor(z) | y ne x)
count(y)= count(y)+1;
y in followers;;
end ∀z;  pin int;
return int;
end interval;
definef intervals(nodes,entry);
optimizer external cesor,
 followers, follow, intof,allnodes;
ints = nℓ;
pile nodup ints; set elt allnodes;
pile nodup seen; elt allnodes;
seen = {entry};
(while seen ne nℓ)
node from seen;
interval(node) is i in ints;
follow(i) = followers;
range follow(x); x aselt allnodes;
 stores follow pile nodup(elt
  allnodes);
(∀bb ∈ i) b = tℓ bb;

intov(b) = i;
range intov(b); b aselt allnodes;
 stores intov elt allnodes;
seen = seen u followers;
end while;
return ints;
end intervals;
/* same comment */

definef dg(nodes,entry);
optimizer external cesor,
 follow, intov, dent, allnodes;
pin nodes;
```

```
npreds = nℓ;
(∀x ∈ nodes, y ∈ cesor(x))
 npreds(y)=
if npreds(y) is np eq Ω
then np+1
       else 1;
ints=intervals(nodes,entry);
dent= intov(entry);
(∀i ∈ ints)
cesor(i)=intov[follow(i)];;
return ints;
end dg;
/* now the algorithm giving
 the full derivation sequence*/
definef dseq(nodes,entry)
optimizer external dent;
seq={<1,nodes,entry>};



<n,e> = <nodes,entry>;
(while#(dg(n,e) is der)
  ℓt #n
doing<n,e>=<der,dent>;)
seq(#seq+1)=<der,dent>;;
end dseq;
/* now the inner-to-outer
 pass of the dead trace
 algorithm */
definef builda(nodes,entry);
optimizer external cesor,
 intv, uses, thru, seqd;
seqd = dseq(nodes,entry);
(1 < ∀k ≤ #seqd,
 intv ∈ hd seqd(k))
```

```
npreds = nℓ;
(∀x ∈ nodes, y ∈ cesor(x))
 npreds(y) =
if npreds(y) is np eq Ω
then np+1
       else 1;
ints=intervals(nodes,entry);
dent = intov(entry);
(∀i ∈ ints)
cesor(i) = intov[follow(i)];;
return ints;
end dg;
/* same comment */

definef dseq(nodes,entry);
optimizer external dent;
seq = {<1,nodes,entry>};
range seq(n); n int;
 stores tupl(pile elt allnodes,
 elt allnodes);
<n,e> = <nodes,entry>;
(while #(dg(n,e) is der)
  ℓt #n
doing<n,e>=<der,dent>;)
seq(#seq+1)=<der,dent>;;
end dseq;
/* same comment */


definef builda(nodes,entry);
optimizer external cesor, throo,
 intv, uses,thru,seqd,allnodes,vars;
seqd=dseq(nodes,entry);
(1 < ∀k ≤ #seqd,
 intv ∈ hd seqd(k))
```

```
naux = nℓ; taux = nℓ;
head = intv(1);
(#intv ≥ ∀n ≥ 1)
b = intv(n);
forward ={y ∈ cesor(b)
 |intov(b) eq intv and y ne head};

if k eq 2 then
 naux(b)=uses(b) u
(thru(b) *
 [u: y ∈ forward]naux(y)) ;;
(∀intx ∈ cesor(intv))
if intx(1) ∈ cesor(b) then
 taux(b,intx)=thru(b);
else taux(b,intx)=thru(b)*
[u: y ∈ forward] taux(y,intx);
end else; end ∀intx;

else /* k gt 2 */
naux(b) = uses(b) u
[u: y ∈ forward]
(thru(b,y) *naux(y));
(∀intx ∈ cesor(intv))
taux(b,intx) =
(if intx(1) ∈ cesor(b) then
 thru(b,intx(1)) else nℓ) u
[u: y ∈ forward]
(thru(b,y) * taux(y,intx));
end ∀intx; end else;
uses(b) = naux(b);
```

```
range naux(b); b aselt allnodes;
 stores naux subset vars; sparse
range taux(b,x); b,x aselt allnodes;
 stores taux subset vars;
naux = nℓ; taux = nℓ;
head = intv(1);
(#intv ≥ ∀n ≥ 1)
b = intv(n);
forward ={y ∈ cesor(b)
 | intov(b) eq intv and y ne head};
pile nodup forward; elt allnodes;
if k eq 2 then
 naux(b) = uses(b) u
(thru(b) *
[u: y ∈ forward] naux(y));;
(∀intx ∈ cesor(intv))
if intx(1) ∈ cesor(b) then
 taux(b,intx)=thru(b);
else taux(b,intx) = thru(b)*
[u: y ∈ forward] taux(y,intx);
end else; end ∀intx;

else /* k gt 2 */
naux(b) = uses(b) u
[u: y ∈ forward]
(throo(b,y) *naux(y));
(∀intx ∈ cesor(intv))
taux(b,intx) =
(if intx(1) ∈ cesor(b) then
 throo(b,intx(1)) else nℓ) u
[u: y ∈ forward]
(throo(b,y)*taux(y,intx));
end ∀intx; end else;
uses(b) = naux(b);
sparse range uses(b);
 b aselt allnodes;
stores uses subset vars;
```

```
end ∀n;
uses(intv)=naux(head);
(∀intx ∈ cesor(intv))
thru(intv,intx)=taux(head,intx);;
end ∀k; return;
end builda;
/* now the routine which
 completes the construction
 of 'uses' */
define builduse(nodes,entry);
optimizer external cesor,
 intøv,uses,thru,seqd;

builda(nodes,entry);
(#seqd ≥ ∀k ≥ 1,
  intv ∈ hd seqd(k))
(#intv ≥ ∀n ≥ 1)
b =intv(n);
backorexit={c ∈ cesor(b) |
 c n ∈ tℓ[intv] or c eq intv(1)};
uses(b) = uses(b) u
 if k eq 2 then
thru(b)*[u: c ∈ backorexit]
  uses(intov(c));
else
[u: c ∈ backorexit]
(thru(b,c)*uses(intov(c)));
end ∀n; end ∀k;
return;
end builduse;
```

```
sparse range throo(b,y);
 b aselt allnodes, y hash;
 stores throo subset vars;
range thru(b); b aselt allnodes;
 stores thru subset vars;
end ∀n;
uses(intv)=naux(head);
(∀intx ∈ cesor(intv))
throo(intv,intx)=taux(head,intx);;
end ∀k; return;
end builda;
/* same comment */


define builduse(nodes,entry);
optimizer external cesor,
 intov,uses,thru,seqd,
 throo,allnodes,vars;
build(nodes,entry);
(#seqd ≥ ∀k ≥ 1,
  intv ∈ hd seqd(k))
(#intv ≥ ∀n ≥ 1)
b = intv(n);
backorexit = {c ∈ cesor(b) |
  c n ∈ tℓ[intv] or c eq intv(1)};
uses(b) = uses(b) u
 if k eq 2 then
thru(b)*[u: c ∈ backorexit]
  uses(intov(c));
else
[u: c ∈ backorexit]
(throo(b,c)*uses(intov(c)));
end ∀n; end ∀k;
return;
end builduse;
```

Next, we illustrate the application of the SETL elaboration language to the lexical analyzer, preparse, postparse set of programs described in the SETL notes. Here we give the elaborated algorithms only, in fact only those sections of the elaborated algorithms which differ from the basic algorithms given in the notes. This set of elaborations will serve to illustrate the manner in which, by elaborating a SETL algorithm, we solve some of the key problems which must be faced when the algorithm is to be realized in a programming language of lower level than SETL. We begin with the lexical scan algorithm.

```
definef nextoken;
initially setup(typex, tablex, rpak, cstring);
 range table(type,state); state aselt states; type aselt typeset;
stores table tupl(elt cases, obj);
range type(char); char string index; stores type aselt typeset;
range switchf(case); case aselt cases; stores switch obj;
pile typeset; string;
pile cases; string;
pile states; string;
n=1; <nxt,end,go,skip,cont,do>=;
switchx={<end,endc>,<go,goc>,<skip,loop>,<cont,contc>,<do,doc>};
/* build 'cases', and then 'switch' */
cases = hd[switchx]; pin cases; switchf=switchx; pin switch;
/* build 'typset' and 'states' */
typeset = (hd tl)[typex];  /*note that this involves the tuple
 conventions set forth in newsletter 42 */
states = states u (hd tl)[tablex]; pin states;
type = typex; pin type;
table = tablex; dim table(type: #typeset,state:) pin table;
/* note that our new tuple conventions require tablex to
 have a form differing slightly from that assumed in the
 setup routine given on pp. 129-131 of the notes */
end initially;
```

```
state = nxt aselt states;
nn = n-1; data = Ω; token = nulc;
loop: nn=n+1; action=table(type(cstring(nn),state);
switch: go to switchf(hd action);
goc: state = tℓ action;
/* may not attain maximum possible efficiency */
conc: token = token + cstring(nn);
/* efficiency might be improved by declaration that
 both arguments are strings */
 go to loop;
endc: n =nn; return of data ne Ω then <state,token,data>
   else <state,token>;
doc: <-,rout,action> = action; rpak(rout);
   go to if action eq Ω then loop else switch;
end nextoken;
/* next follows the elaborated form of the preparse routine*/
define preparse(treetop);
initially setup; var=;
/* set up collection of all preparse related 'kinds' of tokens*/
pile kinds; string; kinds=(hd tℓ)[symbkind] u(hd tℓ[typkind]
   with var;
pin kinds;
/* now initialize sumbkind and typkind by using auxiliary ranges*/
domsymkind = hd[symbkind]; pile domsymkind; string;
pin domsymkind;
range symbaux(x); x aselt domsymbkind; stores symbaux elt kinds;
symbaux = symbkind; symbkind = symbaux; pin symbkind;
domtkind = hd[typkind]; pile domtkind; string;
pin domtkind;
range typaux(x); x aselt domtkind; stores typaux elt kinds;
typaux=typkind; typkind=typaux; pin typkind;
/* now in much the same way we initialize the 'mask' and 'label'
   functions used below */
```

```
range maskaux(x); x aselt kinds; stores maskaux bit 15;
maskaux=mask; mask=maskaux; pin mask;
range labaux(x); x hash; stores labaux obj;
labaux=label; label-labaux; pin label;
/* next we initialize the left and right procedure arrays */
range raux(x); x aselt kinds; stores raux int 12;
range laux(x); x aselt kinds; stores laux int 12;
raux=rprec; rprec=raux; pin rprec;
laux=lprec; lprec=laux; pin lprec;
/* and in much the same way, initialize the 'gross' mapping */
range gaux(x); x aselt kinds; stores gaux aselt kinds;
gaux=gross; gross=gaux; pin gross;
end initially;
/* now we give those intiailization operations which are to be
performed each time that 'preparse' is called to produce a tree*/
statstak=nℓ; desc=nℓ; nodtype=nℓ; nodepile = nℓ;
range statstak(j); j int; stores statstak tupl(bit 15, elt
   kinds, obj);
range bakstak(j); j int; stores bakstak  tupl(elt kinds, obj);
statstak = nℓ; bakstak = {<1,er,nℓ>} ;
zero=00000o; state=zero; go to jumpin;
/*stack routines identical with notes page 173-not repeated-*/
/* begin of main process up to subroutine 'getkind' identical
   with notes page 173-not repeated-*/
/* auxiliary subroutine 'getkind'  to classify token */
preparse external symbkind, typkind, kinds;
<type,token,-> = tokdat;
return if symbkind(token) is x ne Ω then x
   else if typkind(type) is x ne Ω then x else 'var' aselt kinds;
end getkind;
newstate: state=lb+state(1:#state-1) or starts and mask(kind);
go to label(state and finish);
nnon: /* same as notes p. 173 */
```

```
/* note (cf. p. 173 for slightly different comment) that
   the elements on statstak have the form <Domolki-state,
   token kind, <token lexical type, token, token-associated
   data (if any)>>  */
/* part-finders for statstak */
definef knd stelt; return stelt(2); end knd;
definef tokof stelt; return stelt(3)(2); end tokof;
definef tdat stelt; return stelt(3); end tdat;
/* next follow notes p. 174 for 5 lines, till test which
   should read as follows: */
if gross(knd 2 elem statstak) eq ('var' aselt kinds)
   then condensenon; else new=newat aselt nodepile; condeseon;
hash nodepile; stores obj;
/* this auxiliary set will initially be null when 'preparse'
   starts to act*/
range nodtype(x); x aselt nodepile; stores nodtype string;
range desc(x,j); x aselt nodepile; j int; stores desc obj;
/* this set of declarations will not attain the full efficiency
   of lower-level code. the difficulty experienced here
   indicates that something like the ALGOL 68 'union' type
   ought to be included in the SETL elaboration language */
/* note also that the lower-level implementation that one
   would normally think of using for desc(x,j) might use
   chained lists. with initial access through the field
   storing 'nodtype'. The implementation that will result from the
   the elaborated SETL is somewhat inferior to this, but not
   drastically so. these differences deserve to be pondered, as
   it may suggest useful extensions to the elaboration
   language. a fast realization of the present algorithm
   might allocate space for two descendants of x in fields
   attached to the field storing nodtype(x); extensions to the
   elaboration language which allow one to call for a
   realization of this type might also be useful */
```

```
newcycle: tokdat = new;
newcyc: kind= 'var' aselt kinds;
/*  every occurrence of 'var' in the remainder of the preparse
    algorithm ought to be replaced by a reference to 'var'
    aselt kinds. the tokens which could appear as node types
    ought to be 'recoded' in much the same way using a
    'pinned hash', but this is harder to accomplish without
    substantial changes in the  SETL 'preparse' algorithm
    as it stands */
/* aside from the points described above, and from the fact
    that the switch 'go to  <er,")",erp , "(",er,per ,...' etc.>
    occurring on page 175 can be speeded up by dimensioning and
    pinning, the remainder of the preparse algorithm can stand
    as is */
end preparse;
/* now we discuss the postparse algorithm */
/* in a few cases, updates to the algorithm appearing in the
    SETL notes, pp. 199 ff, will be shown, in addition
    to elaborations */
definef postparse(syntype,node,nodenum);
parser external lockey,altset,secaltset, sesor, threshold,
 namekeyelt, namesynt, pretests, preacts, postests, postacts,
 rpak, reval, gathalt, minlast, fixed, oblig, literals, lexics;
/* the above sets and mappings are all 'fixed tables' for the
    postparse. they will be declared and 'pinned' for fast access
    in the manner shown below. cf. the table descriptions given
    on pp. 194-196 */
parser external mstak,desc,nodtype; preparse external nodepile;
/* 'desc' and 'nodtype' here are taken to refer to the
    preparse-produced tree structures which are input to the
    postparse routine. the corresponding functions for the
    internally stored 'tree-fragments' defining the structure
    of the syntactic alternatives will be given the names
    'idesc' and 'inodtype' in what follows, and will be treated
    somewhat differently to achieve efficiency */
```

```
initially setup; mstak = nℓ;
/* first obtain the collection of all syntactic types, and
   pin it */
allsynt = hd [keysymbol]; pile allsynt; string; pin allsynt;
/* next form the set of all alternatives, and all
   alternative nodes */
allalts = [u: x ∈ (altset u secaltset)]x(3) u[u: x ∈ tℓ[lexalts]]x;
/* now initialize and pin a pile giving  all alternative nodes
   and their subnodes, as preparation for initializing the internal
   'idesc' and 'inodtype' functions */
pile altnodes; obj;
new = allalts; (while new ne nℓ) node from new;
if node n c altnodes then node in altnodes; new=new u idesc{node};;
end while new; pin altnodes;
/* now pin allalts */ pile allaux; elt altnodes;
allaux = allalts; allalts = allaux; pin allaux;
/* next initialize and pin various maps with allsynt as domain */
range nameaux(synt) ; synt aselt allsynt;
stores nameaux, keyaux string, prelaux obj, lexaux set elt allalts,
sesaux elt allsynt; sactaux, sucaux set int; share(lexaux,sesaux,
                                        sactaux, sucaux) ;
nameaux = namesynt; keyaux = keysymbol; namesynt=nameaux,
   keysymbol = keyaux;
sesaux = sesor; sesor=sesaux; lexaux = lexalts; lexalts = lexaux;
prelaux = prelacts; prelacts = prelaux;
sactaux = sucacts; sucacts = sactaux;
sucaux = suctests; suctests=sucaux; pin nameaux;
/* now initialize and pin a range giving all the principal
   information associated with an alternative */
range altaux(alt); alt aselt allalts; stores apretests, apreactions,
apostests, apostacts obj, aminlast int 3, aaltname string,
agathalt obj; share(apretests, apreactions, apostests,aminlast) ;
apretests = pretests; pretests=apretests;
apreactions = preactions; preactions = apreactions;
apostests = postests; postests = apostests;
```

```
apostacts = postacts; postacts = apostacts;
aminlast = minlast; minlast = aminlast;
aaltname = altname; altname = aaltname; pin altaux;
/* surely a better syntax is required for the above, through
   perhaps a suitable macro suffices */
oblig = oblig assub allnodes;
literals = literals assub allnodes;
lexics = lexics assub allnodes;
/* now we create and pin the pile of twigs */
pile twigs; elt altnodes; twigs={node ϵ altnodes|desc(node,1) eq Ω};
pin twigs;
/* and then initialize and pin two functions which are only
   defined on twigs */
range ltaux(x); x aselt twigs; stores lextyaux set string,
                                    fixaux int;
share(lextyaux,fixaux);
lextyaux = lextype; lextype = lextyaux;
fixaux = fixed; fixed = fixaux; pin ltaux;
/* now we initialize and pin the maps 'inodtype' and 'idesc',
   which define the basic tree structure of alternatives */
range auxnod(alt); alt aselt altnodes; stores auxnod obj;
range auxdesc(alt,j); alt aselt altnodes, j int;
stores auxdesc elt altnodes;
auxnod = inodtype; inodtype = auxnod;
auxdesc = idesc; idesc = auxdesc; pin auxnod; pin auxdesc;
end initially;
/* now we begin the postparse code proper */
pin auxnod;  pin auxdesc;
synt = syntype; tastak = nℓ; /* test and action stack */
range tastak(x); x int; stores tastak tupl(set int, set int);
[syntry:] /* perform preliminary actions */
(1 ≤ ∀n ≤ if prelacts(synt) is pa ne Ω then 0 else #pa)
   rpak(pa(n));;
```

```
iff                          nodeterm?
                arelexalts?                 islockey?
            lextrue+  maynext,    keypres?                 arealts?
            findalt,          aresecalts?  maynext,findalt,maynext,
                              findalt,  maynext;
nodeterm := nodtype(node) eq Ω;
arelexalts := lexalts(synt) is setalts ne Ω;
islockey : lextry=f; = lockey(synt,nodetype(node))
                                          is keyloc ne Ω;
arealts := altset(synt,nodtype(node)) is setalts ne Ω;
keypres := desc(node,keyloc) is keydesc ne Ω;
aresecalts : key = if nodtype(keydesc) is x ne Ω then x
   else hd tℓ keydesc; = secaltset(synt,key) is setalts ne Ω;
lextrue: lextry=t;
end iff nodeterm;
[maynext:] iff                issesor?
                        testoracts?   guessneeded?
                   stackacts+ syntry, goguess,  returnfalse,
                   syntry;
issesor := sesor(synt) is x ne Ω;
testoracts := suctests(synt) is tsts ne Ω or sucacts(synt) ne Ω;
guessneeded := nodnum ge 0;
returnfalse: return f;
stackacts: tsts = if tsts eq Ω then nℓ else tsts;
tastak(#tastak+1) = if(acts is sucacts(synt)) eq Ω
   then <tsts,nℓ>  else <tsts,acts>;
   synt = x;
end iff issesor;
[goguess:] guess(node,syntype,score);
/* algorithm is now identical with notes, p. 200, up to label
   'matched:' */
[matched:] rpak[preacts(alt)];
```

```
if nodnum ge 0 then mstak(#mstak+1) = <nodnum,namesynt(synt)>;
range mstak(j); j int; stores mstak tupl(int 12, string);
ok=t; typecount=nl; range typecount(x); x aselt allsynt;
   stores typecount int;
(∀part ∈ partseq) <subsynt,subnode,start> = part;
if start eq Ω then /* case of non-multiple node */
subnum = ifx                (nodnum lt 0)?
                    (nodnum),       isfixed?
                                (x),      istc?
                                       (tc+1),  (1);
isfixed := fixed(subnode) is x ne Ω;
istc := typecount(subsynt) is tc ne Ω; end ifx;
if nodnum gt 0 and x eq Ω then typecount(subsynt)=subnum;;
ok=ok and postparse(subsynt,subnode,subnum);
else /* case of multiple node */
(start ≤ ∀j ≤ #desc{node})
subnum = ifx              (nodnum lt 0)?
                (nodnum),    (typecount(subsynt) is tc ne Ω)?
                          (tc+1),                (1); end ifx;
if nodnum gt 0 then typecount(subsynt) = subnum;;
ok=ok and postparse(subsynt,desc(subnode,j),subnum);
end ∀j; end else;
/* if failure, simply return */
if n ok then return f;;
/* otherwise do all post-tests and post-actions */
(∀actmsg ∈ postests(alt))<act,msg> = actmsg; rpak(act);
if                      (ok eq f)?
              (nodnum ge 0)?          (nodnum ge 0)?
       printout+     returnfalse,   quit,      returntrue,
       returnfalse;
printout:  print prefix(synt,nodnum) + msg;
returnfalse:  return f;
returntrue:  return t;
end iff;
```

```
rpak[postacts(alt)]; /*also work off accumulation of tests
   and actions from predecessor types */
(∀tact ∈ tastak)<tests,acts> = tact;
(∀tmesg ∈ tests)<act,mesg> = tmesg; rpak(act);
iff                        (ok eq f)?
               (nodnum ge 0)?      quit,
         printout+   returnfalse;
         returnfalse;
returnfalse: return f;
end iff;
end ∀tmesg; if nodnum ge 0 then rpak[acts];;
 end ∀tact; return t;
/* this is end of main body of postparse routine */
/* the subroutine 'matches' which tests a region in a
   parse-tree against a pattern and pretests specified by
   a tree-grammar alternative is essentially thesame as notes,
   p. 202 */
/* the inner recursive  routine 'matcher' is somewhat different,
   and appears as follows */
definef matcher(alt,node);
matches external partseq;
postparse external desc, idesc, nodtype, inodtype, minlast,
   literals, lexics, oblig, lextype;
postparse external allsynt, allalts, altnodes, nodepile;
if idesc(alt,1) eq Ω then go to twig;
/* else not twig; type and subparts to be examined */
if nodtype(node) is nt eq Ω then return f;;
 if nt n ∈ inodtype(alt) then return f;;
desreq = #idesc{alt} is ndesca + if minlast(alt) is min ne Ω
   then min-1  else 0;
/* check on appropriate number of descendants */
if #desc{node}is  ndesc &t desreq or (min eq Ω and ndesc ne desreq)
                                   then return f;;
/* otherwise check parts individually */
```

```
if 1 ≤ ∃j ≤ ndesca - if min ne Ω then 1 else 0 |
   n matcher(idesc(alt,j), desc(node,j)) then return f;;
if min eq Ω then return t;;
/* otherwise last is multiple; check on whether
   obligatory or not */
if idesc(alt,ndesca) is descalt ∈ oblig then
   /* obligatory- check subparts first */
if ndesca ≤ ∃j ≤ ndesc | n postparse(inodtype(descalt),
   desc(node,j),-1)
   then return f;; end if idesc;
/* make additon to partseq */
partseq(#partseq+1) = <inodtype(alt),node>; return t;
range partseq(j); j int; stores partseq tupl(string,elt nodepile);
end matcher;
```

This is as far as we choose to extend our elaboration of the postparse algorithm. The sections already elaborated are in fact those most crucial to the efficiency of the postparse. The reader, basing himself on the material given above, should have little difficulty in supplying the few elaborations necessary for the remainder of the postparse algorithm (notes, pp. 202-207).