SETL Newsletter # 133B. J.T. Schwartz
January 29, 1975

General Comments on High Level Dictions, and Specific Suggestions Concerning 'Converge' Iterators and Some Related Dictions.

1. Introduction.

The 'level' of a language, i.e., its degree of abstractness, is essentially defined by the manual optimisations and routine program transformations which the semantic structure of the language enables one to avoid making. For example, by writing programs in FORTRAN one avoids all necessity either to allocate registers to data or to linearise multidimensional arrays; by writing programs in SETL one is able to bypass many questions of data structure choice. It then becomes the business of a compiler to impose the bypassed transformations and optimisations. To the extent that the compiler succeeds in doing this, the step to a higher language level is without cost; to the extent that it uses efficiency-inferior substitutes for the better sequences which a hand programmer might invent, the use of a high level language imposes space and routine penalties.

At the present stage of development of optimisation methods, certain optimisations can be handled automatically, while others cannot. Among those which cannot, some are 'almost mechanisable', and most appropriately regarded as routine transformations which can be applied without undue intellectual effort by the educated programmer after study of a very high level program text; others are deep and mathematical, i.e., are real 'inventions'. Transformations of the first sort are typified by the iterator inversion optimisations introduced by Jay Earley, cf. Newsletter 138;

inventions belonging to the second category are typified by the heapsort algorithm, which we are hardly in a position to regard as a routinely optimised variant of any trivial, direct, sorting procedure.

To attain significant insight into the process of programming one will wish to see as many as possible of the devices used in programming as routine program transformations; programs will thereby be seen to involve relatively few unique 'inventions'. As part of this process, one will have to learn to write programs in their *'urforms'*, i.e., as they may be supposed to exist before the application of any routine, even if manual, optimisation (but nevertheless after the crystallisation of the program out of a still more primitive underlying 'rubble'.) To do so is not easy, since to apply routine optimisations is so much a part of the programmer's stock-in-trade, so fixed a habit, that a special intellectual effort is required to desist from it or even to see clearly that one can desist. Nevertheless, by disciplining oneself to supress routine optimisation i.e., by learning to use a language, and especially a high level language, in as high a style as possible, one can hope to create highly succinct, readable, and mathematical program versions, from which secondary, more efficient versions (still in a high level language such as SETL) can be seen to arise by a process of transcription having a formal flavor. And from this secondary high-level program version a further process of manual transcription to program versions in still lower language level can begin.

If this general approach to programming is adopted, the trained programmer's knowledge will among other things comprise:

i. Algorithms, i.e., various basic algorithmic inventions (e.g. heapsort, fast Fourier transform, fast polynomial factorisation methods, parsing methods, etc.). This knowledge is function-oriented and much of it has a mathematical flavor.

ii. Optimising transformations. (e.g. 'iterator inversion', techniques for deferring or eliminating computations, reducing iterations to recursions, recursions to stack manipulation, condensing and encoding tables, etc.) These transformations, some of which are discussed more explicitly below, have a pragmatic flavor.

iii. Formal principles of data structure choice.

iv. Tricks, perhaps even machine-dependent tricks, for inner-loop optimisation.

The optimising transformations noted under (ii) above may eventually come to lie within range of a fully automatic optimiser. However, even before this becomes possible, we may hope to develop semi-automatic (possibly interactive) systems capable of accepting 'high style' codes and transformation directives as input, and of producing 'low style' codes (perhaps in the same language) as output.

2. A catalog of routine but non-automatic optimisations.

At the periphery of any attempt to formalise the process of optimisation one will collect optimising transformations too complex to be worth performing automatically, but still essentially routine. At the ALGOL or FORTRAN level, for example, the following recognised optimisations fall into the 'routine but not automatic' category:

1. Unswitching. Convert a loop containing a loop-independent forward branch ('bypass') into two separate loops, one containing the bypassed code, the other omitting it; and enter one or the other loop, depending on the result of an appropriate test, made before loop entrance.

2. Loop Amalgamation (or 'Jamming'). If two sucessive loops

```
              DØ  1   N = A, B
                      block1
        1 ...
              DØ  2   N = A, B
                      block2
        2 ...
```

manipulate sufficiently disjoint data, amalgamate them into a single loop

```
              DØ   1  N = A, B
                      block1
                      block2
        1 ...
```

thereby saving loop-associated bookeeping and possibly attaining other benefits besides

3. Loop Unrolling. Change critical inner loops of the form

```
              DØ   1   N = A, B
                       block
     1 . . .
```

to loops

```
              DØ   1   N = A, B, K
                       block_1
                       block_2
                       . . .
                       block_k
     1 . . .
```

where each pass through the latter loop increases the loop index by k, and where $block_1,\ldots,block_k$ are obtained from the *block* of the former loop by substituting N + 0, N + 1,...,N + k - 1 respectively for N. This can cut down significantly on the number of loop-bookeeping operations executed.

4. Various forms of call optimisation.

The text of routines called from within critical inner loops, and also routines called only once, can be inserted 'in line', and optimised in the context of their points of call. This will allow constants to be propagated into the routine body, test outcomes to be calculated at compile time and useless code eliminated, redundant computations removed, etc.

5. Transformation of recursions to stack manipulation.

A hand programmer, knowing the subset of internal variables of a recursive routine whose values will be required after a recursive call, can stack only these; moreover, his specially tailored stacking procedures can be considerably more efficient than the general procedures used to support a generalised recursive call facility.

As one of a large number of occasionally useful program improvements we mention

6. Transformation of an immediate pre-exit recursive self-call to a 'change' of input parameters and restart;

The code structure

```
        procedure recursive (parama, paramb);
enter: ...
        call recursive (apa, apb);
        return;
        . . .
```

can be transformed to

```
    procedure  recursive (parama, paramb);
enter: ...
    savea = parama; saveb = paramb;
    parama = apa; paramb = apb;
    go to enter;
    . . .
```

At or above the SETL level of language we find the following routine but not easily mechanisable optimisations.

7. Set theoretic strength reduction. (J. Earley's 'iterator inversion'; 'formal differentiation'.) This transformation, of common occurence, keeps the current value of a frequently used expression available, and, at each program point at which one of the parameters of the expression changes, inserts operations updating the value of the expression. Updating may be very much faster than recalculation since the new value required may not differ much from the available prior value.

8. Transformation of tree iterations into recursions.

If some process P must be applied to all the nodes of a tree, and if the order in which the tree nodes are processed is irrelevant, then the tree may be walked recursively and P applied to its nodes as they are encountered. This same remark applies to any situation in which a necessary order of node processing is compatible or can be made compatible with, some treewalk order, and to a wide variety of tree related calculations. The recursive routines typically used in such situations can be considered to arise by application of this organising idea to an underlying, less specifically arranged, algorithm. Generally speaking, any relevant aspect of the mathematical structure of a compound data object can be used to guide and optimise the order of processing of its constituent subparts. For example, strings may be processed in right-to-left order, cycle-free graphs in an ancestor-descendant-ancestor order, etc.

9. Computation deferal, replacement of compound data objects by 'generator' coroutines which generate their individual parts.

In some cases, a compound object may be seen to be generated at one point in a code only in order that it may be iterated over later in the same code. If this the case, we may, instead of generating the object, simply provide a generator routine which will supply its sucessive parts as they are subsequently required. The generator routine and its internal data can then be regarded as a kind of symbolic form of the object, which would otherwise have to be enumerated *in extenso*. As a typical example of this frequently occuring optimisation we may note the existence of 'on the fly' parser/code-generator routines which generate the nodes of a parse tree implicitly and use them immediately for object code generation, without ever finding it necessary to build up a full representation of the tree itself. A parser/code-generator of this sort can be regarded as an optimised version of a two stage compiler which first generates a tree and then walks it to produce object code.

Note that routine program transformations of lower-level (1-6) above are also applicable at the SETL level and at higher linguistic levels.

The program transformations defined in the preceeding pages are optimisations in the strict sense that they transform one code into another having exactly the same function. It is worth considering, as akin to these, a wider class of transformations which do not precisely preserve, but instead extend program function, in ways however that are essentially routine.

We may regard the 'unextended' version of a program to be extended as an *urform* from which the extended version arises routinely. Among transformations of this class we note the following:

10. Insertion of diagnostics and data-acceptability tests; relaxation of assumptions concerning input data.

In a logically minimal version of a program which handles input data, one will probably want to assume that the data conforms to some convenient external specification; such an assumption may of course be overoptimistic. One corrects it, and comes to a sounder program version, by a process, often routine, which inserts tests for data acceptability at suitable points along data input paths. These tests can correct or reject erroreous data, and may emit notifications when data is rejected; by the time they release data to the rest of the system, it can have been certified as (partially!) correct. Insertions of this type cause a program to grow incrementally; a similar process of incremental program growth is to be expected whenever logical assumptions concerning input data are relaxed, and when in consequence processing of this data must cope with the new possibilities. In all such cases, it will sometimes be possible to insert, along the relevant input paths, code which handles these new possibilities. If the 'old' code will never 'see' any of the new cases which are being handled, this incremental approach is fully successful. Generally, so fully isolated a treatment of a significantly expanded set of allowed cases will not be possible, and the code which handles new cases may need to use sections of old code; so that to accomodate the new code in a rational manner old code may have to be restructured, online blocks moved around or converted to subprocedures, etc. Note that many of these same transformations will have to be applied when and if the presence of a bug signals the occurence of internal data not conforming to assumption.

11. We note, to conclude this section, that *equations* of ordinary mathematical form may be considered to constitute a programming language of very high level. Equations specify their solution, but not how to find it; however, if the way in which equations of a given class are to be solved can be deduced from the form of the equations themselves, we can regard passage from the equations to the routine which solves them as a transformation from higher to lower program language, in principle like the other transformations which have been consider in the preceeding pages.

3. Benefits associated with the formal use of 'high style' program variants.

One only extracts a programming style's full potential benefit when one succeeds in codifying the style as a language subject to automatic processing. Nevertheless, even if this crowning step is not taken, benefit can still be derived from the deliberate use of 'high style', i.e., deliberately abstract and unoptimised, program variants. A high style algorithm will suppress some of the optimising complications which the same algorithm, written in the same language in a 'lower' style, would embody. The introduction of these optimisations then constitutes a separate step of composition. By breaking the process of algorithm composition into two subparts, and by approaching its second step in a manner emphasising its routine aspect, the programmer will attain a significantly better final result than if he approaches the whole design of an algorithm at once.

It is also worth noting that, when a program is to be proved correct, it is bound to be best to approach it via a variant of maximally high level, to prove this variant correct first, and then to prove that the optimising transformations subsequently applied to it preserve correctness. Note that a relatively small standard set of optimising transformations is likely to be used repeatedly, so that the proof that these transformations preserve correctness will be a standard 'Lemma'. Moreover, compared to an equivalent low level variant, a high level program variant will be significantly less cluttered with subsidiary detail of the sort that makes difficulties for and lengthens a correctness proof.

By using a given language L in a deliberately 'high' style, we prepare ourselves for the formal definition of a still higher semantic level L' of language. The program transformations that are informally seen as routine improvements when L is used in high style become potentially automatic optimisations when L' is explicitly formalised. Note that it is only after we have formally defined L' that the problem of optimal translation of L'-programs into L-program is truly opened; i.e., it is only this step of formal definition that allows us to see certain fundamental issues concerning L-level programs in their truest light.

We note finally that the development of programming languages to progressively higher levels will eventually close the gap which presently separates the 'bottom-up' approach of the formal language designer from the 'top-down' approach inherent in various current studies of 'automatic programming'. As this gap narrows, programming language design should be able to contribute significant ideas to, and absorb ideas from, the natural-language/artificial-intelligence oriented 'automatic programming' work.

4. A few suggestions made in conformity with the preceeding generalisations.

While not having any great improvement in language level to recommend at the present moment, I shall suggest a few syntactic conventions intended to make 'converge' iterators of the sort introduced in Newsletter 133 and 133A easier to use. For the reasons adduced in Newsletter 135A, iterators of this sort may be expected to occur frequently. I will also suggest a few small SETL extensions which address deficiencies of the language and which may be found particularly convenient for the 'high style' use of SETL suggested in the preceeding pages. After these extensions are outlined, a few sample algorithms, written in the envisaged style, will be given.

Revising the syntax (but not the semantics) suggested in Newsletter 133A, we shall write simple converge iterators as

```
(1)                 ( ∀ )
                        block
                    end  ∀;
```

Note that, as before, the *block* is executed till its execution fails to modify any variable. The frequently occuring case of the general form

```
                    ( ∀ )
                        (∀x∈s | C(x))
                                block
                        end ∀x;
                    end ∀;
```

will be abbreviated (cf. Newsletter 133A) as

```
                    (∀∀x∈s | C(x))
                            block
                    end ∀∀;
```

If the *block* in (1) consists of a single assignment statement *x = expn*, we shall abbreviate (1) simply as

```
(2)                 x = : expn;
```

A converge iterator of the form (2) will often be preceeded by an assignment initialising x. Accordingly, we write

(3) $\quad$ x = :: $expn_1$ <u>op</u> $expn_2$;

as an abbreviation for

x = $expn_1$;
x = : x <u>op</u> $expn_2$;

For succinctness, we allow the 'iterating assignments' (2) and (3) to be used as expressions also, their value being that of x when iteration ceases.

As a first example, note that these conventions enanle us to write a quite succinct transitive closure routine:
definef tranc (f, s); return x = :: s + f[x]; end tranc;

Especially in using condensed dictions such as (2) or (3), but also in SETL programs more generally, the syntactic overhead associated with a subfunction definition and call may be bigger than the function body itself. With this in mind, we introduce an abbreviated function-call style. Functions are called in this abbreviated style simply by writing their names, with no parameters; (in effect, parameters are transmitted globally.) The function body of an 'abbreviated form' function is introduced by the keyword where, which must be followed immediately by a token identical with the name *fname* of the function being called. This may either be a label, or (for brevity) an assignment target. The abbreviated function body is terminated by

end where;

All variables occuring in the body of an abbreviated function are global to the ordinary function or subroutine containing its body; and the function itself has this same scope. The value returned is the value of the variable *fname* at the moment of return, which is the first moment when either a 'return' statement or an 'end where'is encountered.

The following example illustrates these conventions, (and also assumes, for convenience, that operators sending a SETL map into its domain and range respectively have been defined; 'range' is also assume to apply to a tuple, and give the set of its components.)

It is a program which solves the combinatorial 'matching' problem described on pp. 122-125 of OP II. I.e., a map with disjoint domain and range are given, and the algorithm extracts a maximal 1-1 submapping of the given map. This procedure used is adapted from the *maxflow* algorithm of OP II, p. 123.

```
definef maxmatch(map);
<source, sink> = <newat, newat>;
graph = map + {<source, x    >, x ∈ domain(map)}
              + {<x,    sink>, x ∈ range(map)};
graph = : graph - (path is p) + {<x(2), x(1)>, x ∈ p};
where path = :: nl + {<pred(z), z>, z ∈(domain(path) + {sink}) |
                                                   pred(z) ne Ω};
    where pred = :: nl + {<x(2),x(1)>, x ∈ graph | sink not ∈ domain(pred)
        and x(1) ∈ domain(pred) + {source} and x(2) not ∈ domain(pred)};
end where; end where;
return map-graph;
end maxmatch;
```

We shall convert this 'high SETL' program to an equivalent 'low SETL' form, simply in order to illustrate the transcription process involved. Studying the preceeding code, we note that *path* is used only to support an iteration, so that its explicit formation can be suppressed. Moreover, the map *pred* and its domain can be formed differentially; and the two successive iterations used in the initial formation of *graph* can be amalgamated. The inner program loop is that which forms *pred*. These observations lead us to the following 'low SETL' code:

```
definef maxmatch(map);
<source, sink> = <newat, newat>;
graph = nl;
(∀x∈ map)
```

```
     x in graph; <source, x(1)> in graph; <x(2), sink> in graph;
end ∀;
(while true)  /* loop till return is made */
     /* build up pred */
     reached = {source}; new = {source}; pred = nℓ;
     (while new ne nℓ)
          point from new;
          newest = graph {point} - reached;
          (∀ np ∈ newest)
               pred(np) = point;
               np in new;
               np in reached;
          end ∀np;
     end while;  /* now pred is built up */
     if sink not ∈ reached then
          return map-graph;
     else  /* replace path edges by their reverses */
          point = sink;
          (while pred(point) is predp ne Ω)
               <predp, point>  out graph;
               <point, predp>  in  graph;
          end while;
     end if;
end while;
end maxmatch;
```

Two other SETL extensions are worth suggesting:

a. The very restrictive way in which Ω is presently treated in SETL has the advantage of exposing program faults rapidly at run time, but in some cases it forces longish circumlocutions to be used where the programmer would find it more convenient to use an Ω or illegal operation as a termination signal of some kind. For use in such cases, the following device is suggested. If *expn* is an 'elementary' SETL expression, i.e., an expression involving only primitive SETL operations without embedded function calls, the expression

(1) $$\text{expn } \underline{\text{ort}} \text{ expn}_2$$

has the following semantics: if evaluation of *expn* leads either to an Ω result or to a run-time error, (1) has the same value as $expn_2$; otherwise it has the same value as *expn*.

b. The conventions for tuple component naming described in O.P. II. page 94, are still too primitive and clumsy. The following revised conventions are suggested.

i. If e is not an integer constant, and in particular if e is an expression with a tuple, map, or string value, while n is an integer-valued expression, then e.n is synonymous with e(n).

ii. By writing

$$\textit{gname } \underline{\text{has}} \text{ cn}_1, \ldots, \text{cn}_k;$$

one defines $cn_1, \ldots, cn_k$ as macros for the integers $1, \ldots, k$ respectively. Of course, it is intended that these integers should be used to designate tuple components. The token *gname* is semantically insignificant, and has mnemonic value only; it serves to remind one of the objects whose components are being named. The related more general form

(2) $$\textit{gname } \underline{\text{has}} \text{ cn}_0 \ \underline{\text{and}} \ \text{cn}_1, \ldots, \text{cn}_k;$$

where cn_o is an integer, and which designates $cn_1,\ldots,cn_k$ as macros for $cn_o + 1,\ldots,cn_o + k$ respectively, might also be useful.

Once this macro facility is introduced, the following additional notational forms will probably be convenient:

(3) $$<: \; n_1 \quad e_1 \quad n_2 \quad e_2 \;..n_k \quad e_k>.$$

for the tuple whose n_j - th component is e_j, $1 \le j \le k$, and whose other components are Ω. Moreover,

$$a.\ n_1,\ n_2,\ldots n_k = \text{expn};$$

can usefully stand for the multiple assignment

$$<a.\ n_1,\ a.\ n_2,\ldots a.\ n_k> = \text{expn}.$$

iii. As an aside, we note a few notational possibilities suggested by (3). It would not be unreasonable to introduce 'named parameter' procedures and functions introduced by header lines of such forms as

(4) definef procname parlist_1 parname_2 parlist_2...parname_k parlist_k; e.g.

definef swiveled body distance x angle theta;

Here *procname* names the procedure, $\textit{parname}_2,\ldots,\textit{parname}_k$ names its parameter subgroups, and each $\textit{parlist}_j$ is a comma-separated list of parameters. Then, when *procname* is called, we allow the parameter subgroups to appear in any order, each preceeded by its name; but within a group arguments must be given in the order in which the corresponding parameters appear in the corresponding *parlist* of (4).

This allows calls to have the pleasing form exemplified by

```
swivel body2 distance x2 angle theta2;
```

which can equivalently be written

```
swivel body2 angle theta2 distance x2;
```

Omission of designated 'optional'parameter groups can be allowed.

Named parameter functions can be allowed also, but since functions nest some system of parenthesising delimiters is called for. One possibility is to call the function introduced by a declaration such as (4) by writing

$$(:\ \text{procname arglist}_1\ \text{parname}_2\ \text{arglist}_2 \ldots \text{parname}_k\ \text{arglist}_k),$$

where the named-argument groups can be optional. An example might be

```
(: swiveled body distance x angle y).
```

As a final example illustrating some of the conventions suggested above, we give 'high SETL' code for the Cocke-Allen program graph analysis process described on pp. 269-272 of O.P. II. Note that the code which follows combines the four routines *interval, intervals dg, dseq* of those pages.

```
definef dseq(graphnodes, graphcesor, graphead);
<nodes, cesor, head>= <graphnodes, graphcesor, graphead>;
intov = nl;  /* map each node into its interval */
dseq = :: <nodes> + if #(intervals is ints) eq # dseq(# dseq)
                                 then nult else ints;
where intervals = :: nl + {interval} ort nl);
     intov = intov + {<nd, int>, int ∈ intervals, nd(k) ∈ int}
     <nodes, head> = <intervals, intov(head)>;
     cesor = cesor + {<int,intov(cesnd)>, int ∈ intervals,
               cesnd ∈ cesor {range(int)} | intov(cesnd) [illegible]
```

```
        where interval = :: < ∃followers> ort nult +
             [+: nd ∈ (cesor[range(interval)]-
                                      cesor [nodes-range(interval)]}) <nd>
          where followers = {head} +
               (cesor [range [interval] is intnodes] - intnodes ort n
end where; end where; end where;
return <dseg, cesor, intov>;
end dseg;
```

Concerning the above, note that it is assumed, since the *followers* function is internal to the code for *interval*, that each occurence of the token *interval* within followers refers simply to the current value of the variable '*interval*', and does not cause a recursive call.