SETL NEWSLETTER NO. 83 November 1, 1972

USER EXPERIENCE AND HUMAN FACTORS IMPROVEMENT OF SETL

J. Schwartz, et al.

This newsletter collects user impressions of the present SETLB system, with a view toward reaching a consensus concerning 'human factors' or 'usability' issues which deserve future attention.

This first draft (by J.Schwartz) should be reviewed by the members of the SETL group, emendations made, and parts added as appropriate. It is intended merely to start the process of reaching an overall consensus.

1. OVERALL IMPRESSIONS: DEBUGGING.

Even in its present imperfect state, the SETLB system seems very good. It is possible to program complex processes with ease, and should, as we have hoped form the beginning, focus our attention on the larger issues of programming (algorithm invention and overall strategy) by overcoming a very significant part of the mass of programming irritation inevitable if other approaches are used. The removal of these difficulties will cause other issues, perhaps marginal for other methods, to assume a more central role in our considerations. If 80 percent of a problem is solved, the factors affecting the unsolved 20 percent become significant.

SETLB debugging is significantly easier than debugging at, say The FORTRAN or PL(1 level. Error - symptoms generally occur near causes, so that the fix to be made is often obvious. The 'HELP' debugging aid is quite effective, especially the store trace. The flow trace seems almost useless. The subroutine entry trace should probably allow another level, in which arguments are printed as the subroutine is entered. Tracing with HELP on is a quite effective method of early debugging. The store trace should print values of of variables read by a 'READ' statement, and assignments made by IS. FRØM., and IN. should also be detectable. Unexpectedly, the ASSERT feature does not seem too useful, perhaps because the language level of SETLB is already high, making ASSERT expressions as complicated as the statements to which they would relate.

Use of SETLB seems to decrease the bug-content of programs to a remarkable degree. The commonest bugs are

i. Trivial 'immediate crash' bugs caused by mispellings, non-initialisations, conflicting uses of variable names, name scope misunderstandings and misusages, etc.

ii. Logic bugs, i.e., fundamental errors in the algorithm being used

iii. System bugs, i.e., bugs related to discrepancies between the implemented SETLB and the more highly rational 'publication SETL'.

Comments on bugs of Category i :

a)Warnings for likely small errors, as 'NL' for 'NL.', 'PRINT' for 'PRINT.' should be given in the suspicious variable list.

b)The suspicious variables list is still only marginally useful, as it tends to include too many variables whose use is legitimate. The following improvements are possible:

ba) count each occurence of a token as label as two ordinary occurences, and each occurence of a token as subroutine or function name as three occurences.

bb) List the cards on which suspicious variables occur along with the variable name.

bc) If a variable is suspicious, check for a variable of like spelling, and list any such along with the suspicious variable.

c) Sophisticated compile-time analysis of the type of values which variables will assume can detect a useful variety of bugs of both the misinitialisation and misspelling types; a consideration which underscores the potential importance of this type of analysis.

d) The present crash messages would be considerably more useful if they gave the symbolic names of the routines found in the call - chain at the time of crash. This can be done easily, using the following scheme:
When the SETLB preprocessor encounters a ' define sub...' or 'definef sub...' it can record the name 'sub',and,on beginning its second pass, insert an initial call of the form record (sub, = sub).
This call can append a pair consisting of the internal BALM subroutine pointer and the external subroutine name to a 'subroutine list', which can be consulted by the crash-message generator to enable subroutine names to be included in crash messages.

Alternatively, the crash message processor can search the symbol table for a variable whose value is identical with a subroutine pointer, and take the external name of this variable as the external subroutine name.

e) An automatic analysis of name-scope relationships, with appropriate print-outs, might be useful.

Bugs of category ii require extensive rethinking, but can hardly be cured by system improvements. There is probably not much that can be done about bugs of type iii before the new SETL system becomes available.

Code consisting largely of expressions to be evaluated seems to be considerably less bug-prone than procedural code realising equivalent logical function. This suggests an effort to expand the 'expression' part of the SETL syntax and semantics.

2. Turnaround, interactive use. The significant proportion of bugs that can be found easily makes some method for 'quick fix' highly desirable. Here obvious improvements could come from greater efficiency(a factor of 5, or still better 10, would significantly affect the range of experiment possible) and smaller memory size. Code-paging appears to offer interesting possibilities. The garbage-collector 'space exhausted' routine should probably request more space, or perform a SAVESETL operation allowing subsequent continuation in a larger space, rather than simply terminating.

The biggest gains in quick - fixability would come if it were possible to get down to interactive sizes. Online subroutine patching is desirable. For this purpose, the following scheme might be used.

a) On entry to any subroutine, save (stack) the values of all global variables accessed by the subroutine. (Only those which might be modified need be saved.)

b) When a crash is detected, return control to the user's console. He will then have the option of typing

```
FIX  SUB
```

where 'SUB' is some subroutine name. This will back up the subroutine call stack to the first occurence of SUB on the stack (if none such occurs, it will clear the stack and prepare for a complete program rerun). At the same time, all variable values will revert to the values which they had immediately before the first stacked call to SUB. The source text of SUB will be retrieved, and put into an edit mode. After editing, SUB will be retranslated, and execution can resume with entry to the new version of SUB.

3. Subroutine libraries and the creation of an Extensive Programming Environment:

The abstract character of SETL should (and to some extent, already seems to) make library subroutines usable at a much higher level of function than is now the case. Because of SETL's abstractress, there exists a real chance that independently written routines manipulating compound data structures should cohere. In languages of the FORTRAN-PL/1 level this highly unlikely since discrepancies in data layouts, pointer systems, etc. tend strongly to cause catastrophic incompatibilities.

This consideration suggests that the SETL user should be kept in contact with an extensive library of subroutines, and be able in a simple way to call any one of these routines into his 'workspace'. Fully satisfactory implementation of this idea will undoubtedly require the development of a sophisticated, systematic namescoping scheme like that outlined in Newletter 76. In the following paragraphs, a much more primitive scheme, but one whose implementation for use with the presently existing SETLB would be relatively easy, will be proposed.

Note that the scheme envisaged here will only become practical after paging of code blocks is implemented, so that one no longer pays a severe penalty for including in frequently executed subroutines in one's workspace.

The subroutine library will consist of

a) A catalog file, organised on an alphabetised keyword basis, from which routine names and short descriptions can be retrieved online.

b) A subroutine source-text file. From this file, the text of subroutines can be retrieved. This file will be broken up into named 'sublibraries', and the retrieval of a subroutine will be by sublibrary and subroutine name. Subroutines can be associated into named 'groups', the group name being sufficiant to retrieve all the subroutines of the group. With each subroutine there will be provided a comment indicating the routines which it calls, and all global variables which it modifies. To include a subroutine (or group) into a users code, one will type an instruction having approximately the following form:

```
include ⟨library - name⟩ (⟨routine/group - specifier * 1⟩);  or
include ⟨library- name⟩ (⟨routine/group - specifier * 1⟩ +
                ⟨variable - specifier * 1⟩);
```

Here

```
⟨routine/group - specifier⟩ →, ⟨*name⟩ |, ⟨*name⟩/⟨*name⟩
⟨variable - specifier⟩→, ⟨*name⟩/⟨*name⟩
```

The intended semantics are deducible from consideration of the following example. If one writes

```
include optlibrary (dg, intof/intervlof + intervals/ints,
                                                cesor/sucessor);
```

one specifies that the routines 'dg' and 'intof' from the specified library are to be included in one's program. The first of these is to retain its name; the second is to be renamed 'intervlof', presumably to avoid a name conflict. All refereces to 'intof', occuring in text retrieved from the library, are to be changed to read 'intervlof', this essentially means that a parameterless macro is to be applied during the 'include' process. Similarly, the global variables 'intervals' and'cesor' are to be renamed 'ints' and 'sucessor' ; this allows them to be referenced globally from the program in which they are to be included. All other global variables occuring in the original 'dg' or 'intov' are to be renamed in some manner protecting them from accidental reference.

4. Input - output issues; Data object library.

A variety of powerful output utilities is desirable, especially if some general,'format driven' utilities can be developed. Two dimensional and tabular output forms are especially desirable; Dave Shields 'print as map' and 'program graph print', as well as J. Schwartz 'tree print' are initial steps in this direction. A logically formatted input utility, allowing tables and compound structures of various kinds to be punched in convenient, minimally punctuated, ways would also be desirable.

Reading sets in their present print format is very annoying, and would be catastrophic if it became necessary to find an item in a large set. The print routine should use some standard alphabetisation procedure to arrange the items to be printed.

The following proposal describes one possible formatted read scheme. Strings read would be broken into substrings using blanks as delimiters. Formats would have the following external syntax, and a correspond corresponding internal (abstract) syntax:

```
⟨format⟩ → ⟨format-part⟩ | [ ⟨delimited-part * 1⟩ ]
⟨format-part⟩ → s ⟨format-item-string⟩ | ⟨format-item-string⟩
⟨delimited-part⟩ → ⟨* symbol⟩⟨* symbol⟩⟨format-part⟩
⟨format-item-string⟩ → ( ⟨format-item *1 ⟩ ) |
                               ( s ⟨format-item * 1⟩ )
⟨format-item⟩ → ⟨* integer⟩⟨format-part⟩ | ⟨* integer⟩ |
                              ⟨* integer⟩ s |
                              ⟨* symbol⟩⟨format-part⟩ | ⟨* symbol⟩ |
                              ⟨* symbol⟩ s
```

Two examples are:

```
    (1 (.)   /s (1 . ) )
```

and

```
[ ⟨ ⟩ ( ⟩ (, ) )      ≤ ≥ (≥ ( , )  ].
```

The semantic reading-rules associated with the syntactic structure of a format is as follows. Each format item dictates the reading of a number of tokens, which, on reading, are formed into substructures. An item of the form

$\langle$*integer$\rangle$

dictates the reading of a fixed number of tokens.
An item of the form

$\langle$*integer$\rangle$ s

dictates the reading of a fixed number of tokens, and the formation of them into a set (consisting precisely of the tokens read). In general the presence of an s in a format-item indicates that the tokens (or groups of tokens, see below) read by the format-item are to be formed into a set rather than inserted into a tuple.

An item of the form

$\langle$*integer$\rangle$ $\langle$format$\rangle$

dictates the reading of a fixed number of groups of items, each to be read in the manner specified by the format occuring within the item. Thus, for example, if we read A B C D E F in the format

(2 s (3))

we get $\langle$ $\underline{<}$: A,B,C$\underline{>}$, $\underline{<}$: D,E,F $\underline{>}$$\rangle$.

If we read in format s (3 (2)) we get

<: $\langle$A,B$\rangle$, $\langle$C,D$\rangle$, $\langle$E,F$\rangle$ $\underline{>}$.

The groups of tokens read by the various format items of a format-item-string are made into a tuple, or into a set if the format-item-string begins with the symbol 's'. This tuple (or set) is the read-result of the format-item-string.

A format item of the form

⟨* symbol⟩

reads a succession of tokens, up to the first occurence of an instance of ⟨* symbol⟩. These are formed into a tuple, or into a set if the ⟨* symbol⟩ is immediately followed by an 's'.

A format-item of the form

⟨* symbol⟩ ⟨format⟩

reads a sucession of groups, up to the first occurence of an instance of ⟨* symbol⟩. Each group read is of the layout indicated by the ⟨format⟩ occuring in the format-item. This produces a sequence of read-results which are formed either into a tuple, or into a set if the ⟨* symbol⟩ is immediately followed by an 's'.

Some examples: if

A B C D E F .   B D E .   D A E

is read in the format s (1 . s), the following set results:

$$\leq \langle A \leq B,C,D,E,F \geq \rangle, \quad \langle B, \leq:D,E \geq \rangle, \langle D, \leq:A,E \geq \rangle \geq .$$

this is a reasonable input format to use in reading, let us say, text defining the structure of a graph. If X Y Z. A B C D. E F G H / is read in the format (1 (.) /s (1 .)), then the read-result R

$$\langle\langle X,Y,Z\rangle, \leq: \langle A \langle B\ C\ D \rangle\rangle, \langle E, \langle F,G,H\rangle\rangle \geq\rangle .$$

This is a reasonable format in which to read information describing a function of two variables; the actual function F can then be expressed in terms of R by writing

$$F = \leq \langle (R(1)\ (N),\ Y(1),\ (Y(2))(N) \rangle,$$
$$1 < = N < = \downarrow R(1),\ Y \rightarrow R\ (2) \geq ;$$

The semantics of formats of the simple form ⟨format-part *⟩ should be clear from the preceeding remarks.

More complex formats, having the structure

[⟨delimited-part *1⟩]

are used for reading recursive structures; we shall call them recursive formats. Their semantics is as follows. Each ⟨delimited-part⟩ which occurs will have the form

⟨* symbol⟩ ⟨* symbol⟩ ⟨format-part⟩

we call the first ⟨* symbol⟩ of the ⟨delimited-part⟩ its <u>opener,</u> and shall refer to it as s1 in the next few paragraphs. The second ⟨* symbol⟩ of the ⟨delimiter-part⟩ we call its <u>closer;</u> this will be designated as s2. The ⟨format-part⟩ occuring in a delimiter-part we call the read format part of the delimited part.

The read format of the ⟨delimited-part⟩ occuring first within a recursive format F will be called the <u>prime</u> <u>read</u> format of F.

To read a string of tokens using a recursive format F we proceed as follows. We begin to read in the prime read format F of F; since F' is a non - recursive format, the conventions which have been explained above apply. Whenever the opener- symbol of a ⟨delimited - part⟩ P is encountered, we recursively suspend the presently governing read format, and begin to read in the read-format of P. We continue to use this read-format until the closer-symbol of P encountered, at which time we revert to whatever read format P' was previously in use. The read-result R collected between the moments at which the opener-symbol and the closer-symbol of P were encountered is treated as if it were a single token read by P' .

An example will clarify the effects attained by the recursive conventions just explained.
First suppose that we use the recursive format

$$[\; \langle \; \rangle \; (\; > (1, \;) \;) \; \leq \; \geq \; (\; \geq \; s \; (\; 1, \;) \;) \;] \; .$$

This will read in something very like the normal SETLB read format. The reader may verify that, according to the above rules, input

(1) $\langle$ A, B, C, $\leq$ D, E, F, $\langle$ G, H $\rangle \geq$, I $\rangle$

will be read to create the tuple which (1) designates. A variant format can be used to use blanks instead of commas as delimiters, thus simplifying input perparation. This format is

$$[\langle \; \rangle \; (\rangle) \quad \leq \; \geq \; s \; (\; \geq \;) \;] .$$

if the input

$\langle$ A B C $\leq$ D E F $\langle$ G H$\rangle$ $\geq$ I$\rangle$

is read in this format, the tuple (1) will be the read result.

A third example is as follows:

Suppose that we use the read format

```
    [ '('   ')'  s (1  *  (. s) ) //*  s (.) ]
```

to read the input

```
A   (B   C   D  /  E    F  .  G   H  .  I  J  /  .
                K   L .    M    N )    Ø    P
```

The read-result will be

```
≤: A,    ≤: B,  〈 C,  D,  ≤: 〈 E,  F〉,〈G,  H〉, 〈I,  J 〉≥ 〉,
                〈K,  L〉, 〈M,  N〉≥ ,  ≤: Ø,  P ≥ 〉≥ .
```

5. <u>Summary of Comments</u>.

This newletter has been circulated in draft to the members of the SETL group. The following is an attempt to summarise those comments most closely related to the basic issues of useability raised above (A more complete account of comments will be found in Newsletter 83A)

The present practice of interspersing data and program is bad; seperate data files should be used to improve readability.

Concerning libraries and debugging: it is felt that the documentation standards for presently existing SETLB algorithms are too low, and that higher standards should be set (and followed).

A two column format, with comments systematically placed in opposition to related code fragments, might be most desirable. Such a format might be produced by a routine of the 'TIDY' type, which would be a useful aid for library maintainance.

Some of the library functions described above could be provided using UPDATE and Intercom Editor, though nontrivial problems concerning a precuse approach remain to be worked out.

Concerning type i) bugs:
Some bugs could be avoided by having the SETLB processor check the DØ;...; CØMPUTE; scoping.

Subroutine name recovery on crash is probably not difficult to provide, and may be provided soon.

Concerning type ii)bugs. People should read and update SETLNEWS and report problems rather than bypassing them. There may still be some unprotected global names modified from within system routines. These should be protected. A 'local name' default scoping rule rather than the present 'global name' default would be less prone to create bugs. A better solution to the present difficulties concerning labels and transfers within BALM DO/END
and BEGIN/END blocks would also eliminate one class of common system-related pitfalls.

Concerning input-output: it is agreed that formatted input is desirable, since the punctuation presently required is highly tedious and error-prone. The suggestions offered above may be over-elaborate. A simple formatted reader has been coded by D. Shields. This supports formats of the form

```
        FMT = '<I, <I,I,S>,  L >'
```
which then allows, e.g.
```
        10   100   3 ABCD     T.
```
to be read by the statement
```
              IN = RDR (FMT),
```
giving the value
```
              IN = < 10, < 100, 3, 'ABCD'>,  T.>
```

on language extensions: the present character-string package is too primitive, and expanded facitities, providing some sort of pattern-match, would be useful.

BALM does permit error-recovery when space-exhausted; a feature permitting this fact to be exploited may be implemented.

An interactively useable system would be quite desirable, especially for educational uses. It is not clear whether code-paging alone is sufficient to get down to interactive sizes; if excessive paging I/O is to be avoided, it may first be necessary to reduce the inherent size of code blocks by space optimisation and by recoding some of the bulky present BALMSETL procedures at a lower-level.

Various language extensions and modifications were suggested; details will appear in Newsletter 83A. Suggestions included: considerably freer formats for subroutine calls, user-definable object types and operation meanings, user control over lexical conventions, expanded character sets, and left-right symmetric notations for binary relations.

Additional comments

It would be helpful if the store trace could be expanded to a "value assigned" trace which would display the current value assigned to variables in iterators and possibly to each variable in an assignment statement. This becomes more useful as the algorithms become more complex. When the values assumed by iteration variables take on the complexion of multiply nested sets and tuples in a nested set of several iterations, life can become tedious.

Initialization is a problem and a source of annoying error. One should be able to assume that a variable which has not been otherwise initialized has the type which is appropriate for the statement in which it first appears and has the null value for that type. An exception must be made, of course, for type comparison operations (e.g. TYPE. X EQ.) which require prior difinition of X.

A move to implement operator precedence level would be nice but not critical.

Carrying a junk variable at the end of multiple assignments is unappealing.

The suspicious variable threshhold of 3 may be too high. Why not lower the threshhold to one and count subroutines and functions as two.

Clearer messages should be produced when a program crashes because of grossly insufficient memory allocation. Some convenent way of determining minmum core requirements would be useful in avoiding the overestimation /lost run dillemima

The 'macro redefined' warning message should give the line in which the last previous definition or use of a quantity with the same name occurred.