SETL Newsletter No. 75 June 1, 1972
BALM Bulletin No. 13 E. Milgrom

Some Thoughts on the Use of BALM to Implement SETL

The purpose of this newsletter is to present some thoughts derived from my use of the BALM language and system to implement a preliminary version of SETL called BALMSETL (see SETL Newsletter #66). These remarks are mostly suggestions for modifications to (improvements of?) BALM; a few conclusions have been drawn concerning some aspects of SETL brought to light by this implementation.

1. BALM -- general remarks.

1.1 My general opinion is that BALM has proven to be very useful as a tool to produce a first implementation of SETL. It has, however, become a truism to state that the present (interpretative) version of BALM is excruciatingly slow and large. Even with the gain in speed of 15 expected from the compiling version which should be available soon, the system will still be slow and, of course, it will be even larger.

I believe that the time has come to investigate the reasons for these drawbacks. If this is not done very soon, I think that BALM will lose its appeal and that its future will be rather limited. Some suggestions for tools to investigate the preformance of BALM are mentioned below.

If it is verified that the main reason for the slowness of the BALM simulator is the amount of time spent unpacking instructions and branching to the appropriate code, I would suggest the introduction of higher-level (i.e. more powerful) BALM machine instructions and the suitable modification of the compiler or a re-design of the BALM machine.

1.2 A dilemma which faces all language designers is the choice between high and low level of expressiveness. BALM tends to incorporate both very high level and rather low level features. My opinion is that some low level features are missing in order for a user to be able to use the language for systems programming.

I think the language should include several levels of detail; the lower levels should provide optimal declarative features to specify facts which the compilershould be able to take advantage of whenever possible. Even lower level features woul be related to storage structure organization; not everybody likes the way every structure must be represented in terms of lists and vectors; one should be able to design, within the limitations clearly indicated by the system (most notably for garbage collection purposes), one's own storage structures.

1.3 The storage management mechanisms are incomplete, a few additional features would make programming easier and more efficient (see 2.3).

1.4 Name scoping rules are not as barbaric as many seem to think: one can get easily accustomed to them. The question is rather whether one should get accustomed to unconventional name-scoping rules. Indeed, it is not clear what benefits are derived from the BALM name-scoping: if no special advantages are obtained, I would suggest that the Algol-60 or PL/1 conventions be adopted to avoid unnecessary confusion.

1.5 Error handling facilities are poor. There should exist a (slower and larger) debugging version of the system with exhaustive compile-time and run-time tests. Moreover, since many errors are detected during code-generation, it would be useful to number the lines of the source text and pass these numbers to the code-generator in order to localize such errors more easily.
This version of the system should include the tracing features of section 2.1 below and all kinds of other traces such as variables trace (selective or not) procedure calls trace, etc.

1.6 In conclusion, it would seem that a number of language features of BALM are derived from implementation constraints: they stem directly from the desire to keep the translator as simple and straightforward as possible. Compiler simplicity should not, however, be the prime goal of the implementators of BALM. Rather, a more complete frame of "conventional" features should support the BALM language (comments, macros, goto's, name-scoping, handling of parameters, ...).

2. Some suggestions for BALM

2.1 Measurement tools

In the absence of a complete explanation for the time and space requirements of the system, I would suggest the creation of a special version of the simulator incorporating exhaustive measurement tools. In fact, a complete trace should be available, including in various degrees of detail the space requirements and giving also accurate timing figures. The space requirements involve the symbol table, the stack and the heap. As an elaboration of the current facilities provided by BLM4STAT for the heap alone, one should be able to get a message every time a block of heap is required which is larger than a given integer N. The present feature which involves giving a message every time a total of N words have been requested should be retained.
It should also be useful to be able to obtain the total space requirements for every procedure called in a run (stack and heap), together with a cumulative history of CPU cycles for every procedure.

2.2 High-Level vs. Low-Level

In order to be useful as a systems programming tool of acceptable efficiency, BALM should allow more precise control over a number of essential features. One should be able to inform the compiler explicitl of certain properties of the data IF ONE WISHES TO DO SO. I propose therefore the introduction of optional declarations allowing for instance the specification of the type of a variable in order to allow a more complete syntax check at compile-time. Similarly, the declaration of operators should include optional types for the operands.
Another source of inefficiency is the high level of the structures used to form aggregates of data. It should be possible to create new types of values at the same level as the BALM primitives. If the structure of these primitives was known and if one had access to the fields of the machine words (mode, type, information field, etc), it would be possible, for instance, to create the SETL type "blank atom" by creating a new value for the type field instead of representing blank atoms as vectors with 3 components (4 words). Such a facility, admittedly to be used with caution

by the more sophisticated user, would also enable one to create, say, long integers or short character strings in a much more efficient way. In other words, a lower level data definition facility would be very useful in systems programming work.

2.3 Storage Management

2.3.1 Free_storage_list

I think that the language should include a RELEASE command to return a structure which is known to be inactive to a free storage list, or a number of such lists: one for list elements, one for vectors of size N, one for vectors of size cN etc.
A variable allocation policy would then be used to obtain a new requested block either from the heap or from the free storage lists. Garbage collection would occur only when both the heap and the lists are exhausted.
This would allow the system to take advantage of the programmer's knowledge that some area of core can be released.

2.3.2 Space request from the stack

In many cases, a procedure requests a number of words as a local workpile, to be deleted on exit. Such a workpile would be most naturally implemented by allocating some space on the run-time stack. However BALM does not allow one to request space for a vector or a list on the stack or on the heap but according to a stack discipline. In consequence, such workpiles are allocated on the heap, used for a few cycles and then are left for the garbage collector to recover. This is a most inefficient way of handling temporary storage: it is the source of most of the garbage generated during a BALMSETL run.
Similarly, system functions which return a vector or a list (e.g. VFORMS, LFROMV) should be directed whether to allocate this vector or that list on the stack or in the heap.

2.4 Name Protection

The facilities currently available are inadequate: the kind of protection they provide is not the one which is most desirable. In fact, what should exist is a scheme where, after execution of a FREEZE statement, all names previously defined can be accessed

only on a read-only basis (no assignments). A statement such as THAW would allow one to make adjustments to protected variables until the execution of the next FREEZE. It should also be possible to FREEZE and THAW specific variables selectively.

2.5 Pointer vs. Values, Copying, Parameters

The fact that pointers are used extensively but are never mentioned (and are not even a legal data type) makes it difficult to remember at all times what the implications are of an assignment statement. If the values involved are simple (integers, labels, booleans) then assignment involves copying of the value; in other cases (compound values: vectors, lists, character-strings), the assignment involves copying a pointer to the structure. It is my experience that, even though the rule is simple, its implications are usually complicated enough to encourage the programmer to use an inordinate amount of copying -- just in case. This situation is made even more unbearable by the rather unconventional behavior of the parameters of procedures: these parameters cannot be modified by the execution of the procedure, but if they are compound values, they can be modified in some ways, but not entirely. As was the case for the name-scoping rules, I fail to see what advantages are derived from the rule governing the behavior of parameters: I think that a more traditional approach (even at greater cost to the translator and the simulator) would be beneficial for the users. It is worth pointing out that these rules are not modifiable by means of simple extensions and that all languages derived from BALM inherit then its peculiarities.
To return to the copy problem, I would agree that one does not want the assignment statement to imply automatic copying of structures, since this rule lacks generality. However, the present situation is confused because of the different treatment of simple and compound values. A more general scheme should allow both copying of values (simple or compound) and sharing of values (simple or compound): maybe this would help reduce the confusion.

2.6 Labels, Goto's, Returns, etc.

The current restrictions on the use of labels and goto statements seem to have been imposed with the aim of deterring users once and for all from using transfers. One could replace every DO-END block (within which no labels are allowed) by a BEGIN-END block,

were it not for the fact that this would prevent one from using RETURN statements within that block to exit from an enclosing block. This situation occurs frequently; the language should thus be modified accordingly. The simplest way should be to make DO-blocks and BEGIN-blocks equivalent but for their behavior with respect to RETURN. Thus local variables could be associated with DO-blocks and a RETURN from such a block would involve the popping of the stack.

2.7 Miscellaneous notes

2.7.1 It would be worth while, from the BALMSETL point of view, to introduce a number of new BALM_machine_operators: these would replace BALM coded procedures which are exuecuted frequently and/or which generate a lot of garbage.

-- Equality operation for all the basic BALM data types.
-- Copy operation for all the basic BALM types.
-- A generalized conversion routine (from type to type) for all the basic BALM types.
-- A general hashing function for primitive BALM types.
-- Equality test for lists or vectors independently of the order of the elements.
-- An operation returning the number of elements of a list.

2.7.2 Other proposed_modifications_

-- A null statement should be available to make it legal to write

```
...
A = A + 3,
END
```

-- The macro facility should be generalized, in particular, it should be possible to introduce lexical macro's to handle the cases where one or both patterns are not parsable, e.g.

```
, END    MEANS    END
```

3. Some Notes Concerning SETL

3.1 A number of limited tests have been done to check the time and space requirements of BALMSETL, in particular regarding the operations on sets and tuples. These tests have enabled me to decrease significantly the time and space requirements of some of the operations. It seems, however, that the basic data structure for sets, designed for efficient function application operations, is poorly suited for all the other set-operations. Since function application with more than three variables is rather infrequent, I would suggest limiting the nesting of the representation to 2 or 3 levels. It might also be worthwhile generating sets of tuples without nesting and converting such sets to the nested form at the first occurrence of a function applicating operation. Also, the current representation favors the first element(s) of a tuple and is so asymmetric that finding all the elements of a set with a given second component becomes horrendous.

3.2 The method which consists of increasing the hash tables of sets by doubling their size (originally 8) every time a certain loadfactor is reached causes a lot of re-hashing and storage allocation when the sets are very large. In all cases where a set is built element by element, one should try to determine the size of the set before actually building it (whenver possible) and that information should be used to determine the original size of the hash table.