LITTLE Newsletter 29 September 1, 1973

A Medium-Level Semantic Environment Based on LITTLE

J. Schwartz

Table of Contents

1. Introduction.

This newsletter distills a series of reflections on BALM, and proposes a similar, but possibly more powerful and more easily learned, semantic environment.

BALM extracts very considerable semantic power from a very lean set of sources. Moreover, it attains quite a decent level of efficiency when compiled. For these reasons, it repays study. We see the following bases for its success.

i. It incorporates a garbage collector.

ii. Peephole-optimized translation of BALM-machine code proves to yield reasonably efficient target machine code.

iii. BALM's recursive semantics makes it easy to manipulate parse trees and hence to express the BALM compiler in BALM.

iv. Simple precedence parsing meshes quite nicely with the 'MEANS' tree-macro scheme, to attain a surprisingly high degree of syntactic extensibility.

Against these indisputable advantages we may set a number of disadvantages:

a. The BALM-machine operations are somewhat too far from physical machine operations and too lean for it to be possible to create highly efficient target code using BALM. Thus it is not reasonable to use BALM to write either its own garbage collector, a BALM interpreter, or an interpreter for another language. Part of the problem here is that the very dynamic nature of BALM makes global optimization difficult; at any rate, no global optimizer is part of the present BALM system.

b. Too much detail concerning internal syntax-tree forms must be learned in making BALM extensions, especially the case of languages involving high degrees of syntactic transformation (consider, for example, the case of a language incorporating a rich system of declarations).

Other, less central criticisms of BALM will be implicit in some of the detailed proposals to follow. The semantic environment to be proposed will preserve feature (b) of BALM, and will permit peephole-optimized translation (of a very wide variety of source languages, into LITTLE; the use of LITTLE makes available globally optimized final translation down to machine level.) We will abandon the BALM machine, the use of a fixed system of recursion and parameter passing (BALM feature (iii)), and the particular style of precedence parsing used in BALM (feature (iv)). In justification of abandoning feature (iii), we note that nearly all the BALM machine operations are quite trivial. The creation of an interpreter for the BALM machine or any other pseudo-machine opcode system of similar functional level is routine, or at any rate no more difficult than the translation of the pseudo-ops of such a machine into BALM-machine operation sequences. Note also that we shall outline a scheme which makes it relatively easy to define a compiler for a pseudo-machine opcode system once an interpreter for the pseudo-ops is available. In justification of abandoning (iv), note that pure precedence parsing imposes a cramped syntactic style, deficient especially in the level of its diagnostics. It may therefore be argued that a more flexible parsing system, as for example advancing top-down parsing, should be used; in a supportive semantic environment, it will certainly not be hard to write parsers of this style for a variety of languages.

2. 'Heap'- and 'Stack'-related Semantics; Atomic Types.

We proceed to propose a semantic environment as a follow-on to BALM. Oversimplifying,we may define this semantic environment by the phrase, 'LITTLE, plus a stack, a heap, and garbage collection.' The stack can be represented in LITTLE by two global variables STACK (an array of indefinite length), STACKTOP (a pointer to the current top of STACK) and a function RESERVE (which ensures that a specified part of the stack is protected against over-write by the competing *heap*).

The semantics proposed for the heap-related functions are essentially those implemented by Hank Warren's SRTL garbage collector. Specifically, heap blocks of specified size can be allocated; once allocated, a block is referenced by a variable of type *pointer* (a 'TACK'ed variable, in the current SRTL notation). A heap block contains both an initial pointer-free portion and a terminal part containing pointers, each word of the terminal part containing 1,2, or 3 pointers plus a field indicating the number of pointers present. A polished 'extended LITTLE' syntax would allow pointers to be declared directly, perhaps in the syntactic form

```
POINTER P(INITIALPARTSIZE);
```

Following such a declaration, one could allocate a new heap block to P, perhaps using the syntax

```
P = ALLOCATE(INITIALPARTLENGTH,TERMINALPARTLENGTH);
```

Here, the parameter INITIALPARTSIZE gives the normal word size into which the initial pointer-free portion of the block allocated to P will be divided. The syntactic form P(I) could be used, sinister or dexter, to reference the I-th word of the initial part of the allocated block; the syntactic form P[I] could be used to reference the I-th word of the block's terminal portion. The word P[I] can hold a number of pointers (in the SRTL garbage collector, up to 3). It is reasonable to provide several standard field names (such as APOINTER, BPOINTER, CPOINTER) to reference these fields, and an associated special fieldname NUMPOINTERS to reference the 'flag' field within P[I] which states the number of pointers contained in P[I]. Finally, one will wish to provide special field names which extract the initial part length and the final part length of the block allocated to P.

Syntactically much less polished but semantically almost equivalent facilities can be provided in the present LITTLE system by using macros. It is necessary to represent a pointer P by two associated names, which we shall call P and PPTR. The first is a macro, the second a variable which can hold an index referencing the start of a block allocated out of the heap. The following LITTLE macro makes available a kind of 'declaration' which can be used to create additional pointers.

```
+* POINTER(P,PPTR) = TACK(PPTR);
     MACDEF(P(I) = HEAP(I+PPTR)) **
```

(See SETL Newsletter 73 for a description of the macros TACK and MACDEF used here).

Most of the subfield extractors mentioned above are provided by macros already available in the SRTL library.

ALLOCATE can be made available as a subroutine. Note that the POINTER macro given above does not allow the construction P[I] to be used; instead, one must write P(I+OFFSET), where the variable OFFSET locates the first word of the second part of the block allocated to P.

It is appropriate to provide a number of important atomic data types in our extended LITTLE. Real numbers, always of some fixed machine-dependent size REALSIZE are desirable, as are signed integers involving the usual machine dependencies (sum of positive integers may be negative, and sum of negative integers may be positive, in the 'overflow' case; the bit format of a negative integer is inscrutable). A possible form of declaration for such quantities is:

SIZE A(REAL), I(INTEGER);

A few conversion functions are naturally associated with signed integer and real quantities.

SIGNAB(I)

converts a quantity of declared type INTEGER to a LITTLE bit-string representing the same quantity in sign-magnitude form, and vice-versa. This same function yields the sign and magnitude of the mantissa of a quantity declared as type REAL; the function

ESIGNAB(R)

yields the sign and magnitude of the exponent of a REAL quantity R. The function

MAKREAL(E,M)

forms a real quantity R from two bit-strings E and M, which respectively give the exponent and the mantissa of R in sign-magnitude form.

Lexical forms for real and signed-integer constants must of course be provided.

Since even the most elementary language-processing applications are apt to require it, and since it is easy to do so, we propose to provide SETL atoms of type (long, heap-stored) *string* as part of our expanded LITTLE. Because it is equally easy, and to head off the creation of SETL-nonstandard bit-string types, we also propose to provide SETL atoms of type (long) *bitstring*. In our expanded LITTLE, these objects (which are stored in the heap area) will be of declared type pointer. With strings and bitstrings, the operations *extract*, *insert*, *catenate*, and *copy* should be provided;

each of these operators yielding an object of the same (SETL heap) type as the object to which it is applied. Modified *extract* and *insert* operators, which obtain or insert a LITTLE character or bit string from (or to) a SETL character or bit string are desirable also. Boolean operators applying to long bit-strings are appropriate, as are equality and inequality functions for long character and bit strings. Finally, since it is easy and probably useful to do so, we may wish to provide standard hash functions applying to all the types of atoms which have been mentioned; this will facilitate and standardize the use of hashed searches.

Of course our extended LITTLE will, like standard LITTLE, include a full complement of I/O primitives.

3. Parsing, Interpretation, Compilation.

We propose to use a parsing scheme built on the type of advancing top-down parser already used for parsing LITTLE. (A detailed description of this type of parser is found in Stephanie Brown's Top-Down Metaparser User's Guide.) However, we suggest a few variations in overall approach which should further improve the case of use of this parsing scheme.

a. All the parsers will use a fixed *parse interpreter*. The parse interpreter (written in LITTLE) will be called as a subroutine from a *parse controller*. This controller routine will normally incorporate all the 'rpak' generator code of our present approach; in addition, it will transmit all required arguments to the parse interpreter, call the lexical scanner and pass tokens to the parse interpreter, and 'own' the parse interpreter stack, thus making a fixed parse interpreter useable with several separate input languages. The parse controller will build a body of *interpretable text* or *pseudo-code* and pass it to an *interpreter-compiler* either for interpretation or for compilation. (Note, however, that this interpretable text could also be passed to a high-level program analyzer-optimizer, allowing high-level optimization to be performed prior to either interpretation or low-level compilation-optimization.)

A pseudo-code interpreter normally examines the pseudo-operation field of each pseudo-code item submitted to it, does an indexed transfer to the particular code sequence which interprets this item, and, on completing the interpretation of the item, loops back to examine the next pseudo-code item in turn. To convert an interpreter to a compiler, the following scheme, which treats ordinary (i.e., non-transfer) pseudo-code items one way, and transfer items in a more complex way, can be used:

a. Instead of interpreting an ordinary pseudo-op item, emit (into an output code file) the code which would otherwise interpret it.

b. To handle a pseudo-code item of transfer type, emit code which will cause an execution-time transfer having a target label corresponding exactly to the point from interpretation would continue were the pseudo-code being interpreted. Note that for this to be convenient it may be necessary to include explicit 'label' pseudo-items in all pseudo-code text.

c. Pseudo-code items of procedure-call type raise more serious semantic problems, which we approach as follows. In order not to impose the detailed LITTLE call semantics (in particular, its argument-transmission features) on other languages compiled via the system we are describing, we will often prefer to compile pseudo-operations of call type as LITTLE "GO-TO"s rather than as LITTLE "CALL"s. For this to be possible, we must be able to assign labels (in particular, return labels) to variables. Thus, the extended LITTLE language must include an instruction which transfers control to the value of a variable. Similarly we will normally wish to avoid having the LITTLE rules governing actual parameter to formal parameter size correspondence influence the other languages compiled via the system we are describing. For this reason we will often demand that all data objects appearing in emitted code be global and of one of a few allowed sizes. This means that argument transmission patterns, as well as the manner in which the variously structured data objects of the pseudo code being compiled are represented in terms of much more highly standardized LITTLE objects, will be shown explicitly in the LITTLE code which we emit.

To facilitate the writing, in LITTLE, of pseudo-code compilers of the kind just described, and to make easy the transition between interpreters and compilers for the same type of pseudo-code, we propose the following syntactic and semantic extensions, which seem quite adequate though they do not support dynamic definability to the same extent as do more standard interpretive macroprocessors:

A line L of LITTLE code prefixed by the symbol '→' will be emitted rather than compiled. Other lines will be executed in the normal way. Emission will be governed by the following conventions.

a. Names N appearing in L but not declared in the LITTLE subprocedure P containing L will be emitted literally, as strings.

b. If a name N appears in L and is declared in P its <u>value</u> V will be converted to a string and emitted in place of the name. The manner in which V is converted will depend on the manner in which N has been declared. If N is declared using an ordinary SIZE statement, then its value will be converted to a positive integer in decimal representation. The values of names declared as signed integers (resp. real quantities) will be emitted in signed decimal (resp. approximate scientific decimal) form.

c. An auxiliary declaration calling for Hollerith string conversion will be provided. This allows names to be passed to P as parameters. Note that our conventions also allow new tokens to be formed by concatenation.

To convert an interpreter to a compiler, one will in the simplest cases have little more to do than to prefix some of the the lines of the interpreter with the sign '→' thereby causing them to be emitted rather than executed.

Emission-compilation has several advantages (in regard to efficiency) over interpretation. Compilation draws together, into physical sequence, operations which will be performed serially. More significantly, it makes manifest the distinction between calculations which need only be performed once (at 'compile time') and calculations which will have to be performed repeatedly (during 'execution time'). Normally an interpreter cannot spend a great deal of time examining the context in which an operation is to be interpreted so as to find an efficient manner of interpretation. An emitter-compiler can do just this, since a single instruction emitted by it may be executed repeatedly. Thus a good deal of peephole optimization can be associated with a compiler-emitter. Moreover, the code which is emitted can be subjected to global optimization before being converted to machine code.

'Local' or 'peephole' optimization can easily be performed by an emitter-compiler program. To provide this type of optimization, one adds to the emitter-compiler code which looks for special, efficiently compilable local sequences in the pseudo code sequence being processed, and emits modified target code for these sequences. It is natural in doing this to employ a small amount of 'lookahead', i.e., to examine pseudo-instructions following a few places behind that for which code is next to be emitted. An emitter-compiler may also incorporate some appropriately designed finite-state automaton, which can keep track of the changing context (determined by pseudo-operations already processed and code already emitted) in which code is to be emitted.

The occurrence of a label in the pseudo-code stream, and especially the occurrence of a label which is the target of a backward branch, creates problems for a peephold optimizer, which may be forced to make severely restrictive 'worst case' assumptions about the situation prevailing immediately after a label has been passed. This difficulty can be remedied if a global optimizer is made available. We may regard the global optimizer as a program which attaches auxiliary information to each label. This information permits the peephole optimizer to make more favorable assumptions than would otherwise be possible concerning the context prevailing immediately after a label has been passed; in many cases, this greatly improves the quality of code that can be generated.

The type of code fragment which a post-emitter optimization process OPT will find it easiest to deal with is the straight-line fragment containing no imbedded loops or subroutine calls (forward branches are not so problematical as loops). On the other hand, it is relatively less important to compile (rather than to interpret) code containing loops than to compile straight line code, since all that is obtained is an 'outer loop' rather than an 'inner loop' benefit. It is important, however, to transmit to OPT all information defining the variables used, modified, etc. in an interpreted loop. This suggests the following approach to the transformation of interpretable text into compilable code:

a. Handle simple, straight-line interpreter sequences and short interpreter sequences not containing loops by emitting equivalent code.

b. Handle complex interpreter sequences, especially those containing embedded subroutine calls or loops, by emitting calls to interpretive subroutines. In emitting such calls, also emit any additional data necessary to give the global optimizer OPT (which will process the emitted code) whatever information it needs concerning the effect of the calls.

4. Name Protection; File and Overlay Semantics.

The incremental compilation of subroutines containing ever-new variable names creates the danger of accidental name collisions, to prevent which a system of name protection is required. The BALM name protection mechanism is limited and not fully satisfactory, and we shall now propose a scheme which it is hoped will be more adequate. This name protection scheme will function within a larger semantic framework in which file and program overlay facilities will also be supported. These latter facilities are sufficient to make our extended LITTLE system relatively independent of external file cataloging and loader programs.

In the scheme now to be described, names will be known only within single overlays; the overlay will also define the framework within which names are protected. For this reason, the next few paragraphs of discussion assume some single overlay O as their underlying logical frame of reference. Blocks of code can be added to O by incremental compilation. Immediately prior to such an addition, O will contain the code produced by earlier compilations. Within O, certain *permanent name groups* will be open. Only the 'permanent names' in these groups will be accessible during the compilation of additional code. During a single compilation, a group of several subprocedures, plus one main program, can be added to O. In accordance with the latest LITTLE namescoping conventions, all names global to more than one of these procedures must be declared and dimensioned within one of several NAMESET groups prefixed to the subroutine. To allow these names to be identified with names global within prior compilations, we allow names declared within a NAMESET group to be identified with names belonging to some permanent name group of O.

Specific conventions might be as follows. A nameset N, with defining text opened by

NAMESET <name>;

might be allowed to contain declarations of the form

ACCESS <name_1>, ..., <name_k>;

which lists various permanent name groups and makes them available within the nameset. *Identifying declarations* having the form

<name_1> = <name_2>_<permanent_group_name>;

would also be allowed within the defining text of N. Such a declaration identifies the name $name_1$ of N with the name $name_2$ of the permanent name group P indicated in the declaration. Where $name_2$ is unambiguous even if P is not mentioned, the shorter form

<name_1> = <name_2>;

could be permitted.

Note that names declared equivalent to previously existing permanent names should not be either SIZE'd, dimensioned, or equivalenced in the namesets in which they appear, since they will inherit their semantic characteristics from preceding declarations.

All the code presented for compilation during one single pass of the proposed incremental compiler will be optimized at once by a global process which operates across subroutine boundaries. This global optimization process can, of course, make use of information concerning previously compiled subroutines if the LITTLE system keeps such information available. Subroutines compiled in later compilation passes can call upon subroutines compiled in early passes, but not vice-versa.

Note that we propose to compile and place subroutines in the static manner presently characteristic of LITTLE rather than in the dynamic BALM manner. Static compilation is somewhat preferable to dynamic from the point of view of efficiency, but yields a substantially less flexible semantic environment. In particular, static compilation permits neither the redefinition of previously compiled code blocks nor the software-controlled paging out of inactive subroutines. We propose to obtain semantic effects of this type, to the limited extent that they are ordinarily necessary, by use of an overlay feature, to be described below. However, it must be admitted that this design decision deserves review.

Following the body of code to be compiled during a single phase of incremental compilation, we place a series of declarations which modify the permanent name set information to be held with O, deleting some names and namesets, and adding others. Proposed forms for these declarations, together with brief explanations of their intended effects, are as follows:

A. Purge.

PURGE <permanent_nameset_name>;

removes a nameset from the list of permanent namesets, making all names referenced from within this nameset inaccessible subsequently.

PURGE <permanent_nameset_name> (<name_list>);

where <name_list> is a list of names belonging to the indicated permanent nameset PNS, keeps PNS in existence, but removes the listed names from PNS, making them inaccessible subsequently. Similarly,

PURGE <permanent_nameset_name>(-<name_list>);

removes all names but those listed from PNS, making them inaccessible subsequently.

To establish new permanent namesets, we provide a composite declaration whose first line is

PERMANENT <permanent_nameset_name>;

This can then be followed by declarations of the form

ACCESS <name_1>, ..., <name_2>;

which list various global namesets and make them available within the permanent nameset PNS. Identifying declarations having the form

<name_1> = <name_2>_<nameset_name>;

would also be permitted within the defining text of PNS.

Such a declaration identifies the name $name_1$ of PNS with the name $name_2$ of the nameset indicated in the declaration.

We propose the following semantic conventions and primitives for file manipulation.

A. At any moment, some particular collection of file names will be known to the extended LITTLE system. Writing onto a file of unknown name will create a file with this name, initially rewound and empty.

An attempt to read from a file of unknown name will be treated as an error; however, a primitive allowing LITTLE to REQUEST the operating system to supply a file and give it a specified name will be provided.

B. The primitive

```
DROP F;
```

makes the file name F unknown.

```
COPY (F,F')
```

copies F to F' and leaves both files rewound.

```
APPEND(F,F')
```

appends F to F'. (More flexible serial file primitives, providing record marks of various levels, and alling records to be copied, are probably desirable.)

Some files cataloged in the LITTLE system will be flagged as *overlays*. Such files have a special structure; in particular, control can be transferred to them. For transfer of control between overlays, we propose the following conventions and primitives.

a. At any moment, an overlay O contains a certain family of compiled code blocks. Moreover, the variables owned by, and accessible to, those code blocks have certain values. In addition, the common recursion STACK used by the code blocks in O makes available certain information. Note that the information available within an overlay falls into three classes: STATIC information, stored in variables and arrays declared within individual subroutines; information stored on the common recursion STACK; and information stored in the common HEAP, and accessed through variables declared as POINTER. One of the code blocks contained in O is O's *main program*; the other code blocks are subprocedures.

b. A block of code compiled to O (either by O itself or by some other overlay O') consists of a main program (possibly null) and a group of subprocedures. The subprocedures are added to the total collection of subprocedures available within O; STATIC variables referenced within these new procedures and declared (via the 'permanent names' mechanism discussed above) to be identical with variables in previously existing procedures are identified in appropriate fashion with previous variables.

A newly compiled main program replaces the previous main program.

c. A special variable, of type pointer, with the permanent name OPARAMETER, is available in every overlay O. This is used for transmitting parameters to O during *overlay call* (see below). Note that when control passes out of the end of the main program of whatever overlay O is executing, an attempt is made to read additional instructions from the current 'code source' file. Execution terminates if this file is found to be empty (this last convention is the same as that of BALM.)

d. Several overlay call primitives are provided.

i. OVERLAY O(p)

sets the overlay parameter of O to point to a heap item which is a copy of the heap item p, and returns control to O. Note that this type of control transfer is semantically similar to a coroutine call.

ii. OVERCALL $O(f,p_1,\ldots,p_n)$.

Here f must be a string of the form

PERMANENTNAMESETNAME_FUNCTIONNAME .

This string must identify a subroutine entry accessible (through the above-described permanent names mechanism) within O. When executed from within an overlay O' this overlay primitive replaces the main program of O by a program equivalent to

```
POINTER P;
P = FUNCTIONAME (p_1,...,p_n);
OVERLAY O'(P)
```

and transfers control to the newly constructed main routine of O.

The primitive

```
SAVEALL(S)
```

has a string S as its argument. This string names a file known to the operating system within which our extended LITTLE system is supported. When SAVEALL is executed, a copy of the entire state of the LITTLE system is written to the file S, and execution terminates. When the operating system is ordered to re-initiate the extended LITTLE system, this file is passed to a RESUMEALL primitive, the system is restored, and execution proceeds exactly as if the SAVEALL primitive had never been executed. When it is restarted, the LITTLE system accepts a number of files from the operating system, substituting these for a like number of files internal to itself. The 'control card' for re-initiation might have a form something like the following:

```
LITTLE(LITTLE_FNAME_1=SYSTEM_FNAME_1,LITTLE_FNAME_2=SYSTEM_FNAME_2,...)
```

The primitive

```
SAVE(FILENAME,O)
```

where O means an overlay, writes into the named file a body of information saving the whole momentary state of the LITTLE system, except that the main program of O is reduced to null, and O is made the entry overlay for a subsequent resumption. Then, by saving FILENAME (within the external operating system) we make available a file which allows LITTLE to be returned in a known configuration and with control at O.

A file created by copying an existing overlay, or by compiling code into an initially empty file, is flagged as an overlay. Note in this connection that we allow one overlay to compile code into another. This is done by executing a statement

COMPILETO <name>;

before invoking the system compiler. Here, *name* designates a variable, whose value must be a string *s*; *s* names the overlay to which code produced by incremental compilation will then be directed. If *s* is the null string, code produced by compilation will be directed at the currently executing overlay.

We propose the following approach to the management of central memory during incremental compilation. Code blocks, with all their associated STATIC data areas, will be accumulated into low core. At the end of the section of core occupied by routines and STATIC data areas the heap will follow; the system STACK will be placed at the top of the available core area, and will grow toward low core. To gain space for the placement of more code (and STATIC data areas) one can then use something very like an ordinary garbage-collector pass.

It may be desirable to define a form in which standard information, of a type useful to an optimizer, can be kept as part of an overlay O. In such an auxiliary information block one could, for example, indicate the effect of compiled subroutines on data objects named in the permanent names groups associated with O. This would allow more precise optimization of subsequent increments of code. Of course, if such information is kept, it would have to be updated each time an increment of code was added to O.

Simple utilities enabling a program to obtain a copy of the current file catalog, or a list of the permanent namesets and permanent names currently present in any overlay, are bound to be useful.

5. Syntax Macros.

The BALM type of parse-tree oriented syntax macro combines simplicity and power in an attractive way. Macro schemes of this sort can be associated with any parsing scheme which uses tables generated manually and not involving a great deal of delicate 'grammar balancing' ; and provided that complete parse trees for a section of source text are built before any pseudo-code is generated from this section of source text. Thus we may expect to be able to associate such macro schemes either with simple procedence parsers, nodal span parsers, or almost advancing top-down parsers driven directly by a context free grammar unsupplemented by programmed parse-time actions. In such a scheme, we define a macro by establishing an association between two well-formed phrases P_1 and P_2. Then, whenever a node matching P_1 is encountered in a parse tree, it is replaced by a corresponding node matching P_2. For P_1 to be a well formed single clause (or, in a simple precedence grammar, for P_1 to be recognized as a single phrase) it may be necessary to extend the grammar of the language being parsed (or, in a simple precedence parse, to extend the set of known operators and precedences).

A syntax macro system of this type intended to be used with a parser based on a context-free grammar will require the following primitives:

a) A pseudostatement adding productions to the grammar being used.

b) A pseudostatement of some such form as

PHRASE <string$_1$> HASMEANING <string$_2$> END ,

in which *string$_1$* and *string$_2$* are both sequences of intermixed (but distinguishably flagged and individually distinguished) clause type names and literal tokens. This pseudostatement establishes a new syntax macro. To establish this macro,

$string_1$ and $string_2$ are parsed using the context free grammar as it currently stands; each of these strings must be uniquely recognizable as a clause, so that two treelets $treelet_1$ and $treelet_2$ result from parsing these two strings.

c) Subsequently, whenever the structure $treelet_1$ is found in a parse tree, the structure $treelet_2$ is substituted for it.

d) A primitive which drops a macro is bound to be useful.

A syntax macro system (like that of BALM) intended for use with a simple precedence parse is very similar, except that

(a) a pseudostatement defining new operators and precedences is used instead of one defining new grammatical product, and

(b) individually distinguishable symbols designating subclauses (which of course have no types) are used instead of clause type names in $string_1$ and $string_2$.

If instead of the generating top-down parse interpreter proposed at the start of section 3, one elects to use a parser of simple precedence, nodal span, or tree-building top-down type, syntax macros of the BALM type can be made available as a direct part of the parser. In the contrary case, this type of syntax macro can be made available by interposing a precedence-driven preprocesser between whatever parser is used and the lexical scanner feeding it.

6. A Concluding Remark.

A scheme like that which has been proposed seems to me to express much of what is central to the semantic level to which it is addressed, at which level we face the problems of syntactic and semantic definition of monoprocess languages, and of the realization of these languages with a fair degree of efficiency. Of course, the problems connected with optimization are inexhaustible. Moreover, for certain areas of application and certain levels of language, the use of appropriate optimization techniques can improve efficiency very greatly. Moreover, one will continually wish to search for algebras of objects and transoformations useful in connection with particular application areas, the more general, the better; and to seek syntactic forms well adapted to a vareity of application areas. It appears to me that in this, and in optimization studies, we have the truly interesting directions of future monoprocess language work. Note in this connection that SETL derives its interest from the generality and power of the set-theoretic operations which it embodies.

The inclusion of semantic facilities fundamentally different from those customarily provided by monoprocess languages raises other issues; though perhaps in some cases these facilities could be added to the semantic framework described above without straining this framework over-much. The most distinctly different semantic facilities would be those connected with parallel processing, interrupt handling and condition monitoring, process protection and error recovery, and non-deterministic execution. However, non-deterministic execution appears to be a specialized technique useful largely in connection with backtrack algorithms and certain particular kinds of artificial intelligence applications, while the other semantic facilities which have just been mentioned are probably most useful in connection with the creation of operating and real time control systems (and possibly also in connection with discrete simulation languages). For this reason, it is probably well to undertake the actual implementation of these semantic facilities only in connection with fairly extensive operating-system studies, and possibly only in connection with an operating system implementation project.

In a practical vein, it may be remarked that the inclusion into extended LITTLE of generalized left-hand sides (possibly without nesting) and the 'flow' (or 'tree') form of conditional would enhance the language to a noticeable degree.