INTRODUCTION

The purpose of this paper is to present a scheme for employing definitional or "macro" features in a higher level programming language. The emphasis will not be on defining the syntactic augments and precise interpretation of such features in any particular programming language and/or operating environment but, rather, on developing the compiler mechanisms for handling the definition and call of such macros and then indicating the kinds of extensions one might propose to current programming languages in order to usefully employ these kinds of facilities.

Macro facilities have been incorporated for some time in most machine language assembly systems. More recently, macro systems have been developed for string and text handling both as independent systems as well as within the framework of some other systems. However, there have been relatively few attempts to incorporate the ability to extend a higher level programming language via macro type facilities. Some exceptions to this are the DEFINE facility included in the various JOVIAL languages which allowed the substitution of an arbitrary character string upon each occurrence (during lexical analysis) of an identifier previously DEFINEd. The PL/I language has a quite elaborate facility for text-editing type macros which are employed prior to lexical and syntactic analysis of the source text (and which thus require special lexical and syntactic analysis machinery to be available as a preprocessor); further, the PL/I facility for definition of "arbitrary linear mapping function" is a primitive macro facility which is employed after syntactic analysis. The proposal by Galler and Perlis suggests an interesting extension to ALGOL to allow for user-controlled syntactic and semantic augments to ALGOL.

One of the reasons for the lack of macro facilities in higher level programming languages may be that we can identify at least four distinct kinds of macro facilities which might be introduced, each with quite definite advantages and disadvantages. Thus, we submit that to speak of "adding macro facilities" is merely confusing; one must indicate with some precision just where, when, and how he proposes to add such facilities. Very briefly, we identify three times during the compilation process where macros might be added as follows:

Preceding Lexical Analysis—for editing text and incorporating filed or otherwise prepared text into a program prior to any lexical or syntactic analysis of the program text.

During Syntactic Analysis—allowing the introduction of new syntactic structure and
the corresponding semantic structure into the language.

Following Syntactic Analysis—particularly useful for introducing “open coding” for various procedures and functions, such as mapping functions for array elements.

Historically, the schemes and mechanisms we discuss in this paper have arisen out of our experience in dealing with a variety of compiler-compilers (or, more properly, compiler-building systems) and using them to construct translators for a variety of standard and special-purpose programming languages, including ALGOL and PL/I. The compiler-building system on which most of the ideas developed herein are based is the TRANGEN system and its compiler-building language TRANDIR.6,7

As indicated above, we are not going to frame our discussion around any particular programming language; however, if a focus is desired, our thinking is more often than not with respect to languages like ALGOL or PL/I. When we need to reference the programming language being compiled, we shall refer to it simply as language Lp. The ideas will be developed with reference to a proposed compiler model and compiling system which are more-or-less easily particularized to most of the current programming languages. By a compiler model we mean a division of the compiling task into a collection of conceptually distinct parts or phases; we are not suggesting that every compiler must be constructed with these phases, nor do we regard the phases as corresponding to “memory loads” or “tape movements.” Rather, they are conceptual divisions of the compiling task into pieces we choose to distinguish and with respect to which we will explain the various definitional facilities with which this paper is concerned.

The parts of the compiler model can be represented by the following diagram:

```
Lexical Analysis — Syntactic Analysis — Interpretation of the Parse
|
Optimization — Code Selection — Formatting and Output
```

A brief description of the function of each of the parts follows:

**Lexical Analysis**—the process of interfacing with the source of program text and identifying, isolating, and disposing of the terminal symbols and token symbols occurring in the program text, and outputting a sequence of descriptors of these terminal and token symbols. By terminal symbols we mean the basic characters and strings of characters comprising the alphabet of the language, e.g., “+”, “,”, “begin”, “procedure”, and the like; by token symbols we have in mind the strings of characters comprising individual members of such (terminal) classes as identifiers and literals (e.g., “ALPHA”, “1.5604E-3”, “true”, and the like). By “converting” and “disposing of” the token symbols we mean whatever process is required to provide us with an entry in a symbol table or a literal table and produce the resultant descriptor pointing to that entry.

**Syntactic Analysis**—the process of identifying the syntactic units (with respect to some particular syntax) occurring in the stream of descriptors produced by the lexical analysis process, resulting in a parse of the program text.

**Interpretation of the Parse**—the process of generating that representation of the computation which will provide the basis for whatever optimization and subsequent generation of machine (or other) coding is to follow. In this paper we will restrict ourselves to a “pseudo-code” or “computation tree” representation of the computation; prefix or suffix representation, direct machine code representation, and other variations are possible with the mechanisms described, but we shall not here consider them further.

**Optimization**—the process of preparing the pseudo-code representation and gathering a variety of useful information prior to the actual generation of machine coding. Such things as elimination of common subexpressions, flow analysis, development of statistics to guide register allocation and so on are included here.

**Code Selection**—the process of inspecting the pseudo-code representation of the computation and, using the information developed during the optimization phase, generating the sequence of machine coding for carrying out the computation.

**Formatting and Output**—the process of
preparing the final machine code representation (e.g., relocatable binary) and other information (e.g., some form of symbol table) from the machine code as produced by the code selection process in the formats required for punching, running, filing, or whatever disposition is to be made of the resultant program.

The compiler for $L_P$ will be represented by means of “translation programs” in two programming languages which we will describe below: the base language $L_B$ and the descriptive language $L_D$. We shall associate various portions of the translation programs with the various parts of the compiler model; the actual compilation will be done by the compiling system performing the set of actions or statements given in $L_B$ and $L_D$ by whatever means it may choose to do them (interprettively, from a machine code version resulting from compiling a program in $L_B$, or whatever); the only assumption we make is that any $L_D$ program (i.e., the part of the compiler represented in $L_D$) is translated into an equivalent $L_B$ program (by one of the facilities in the compiling system) prior to the translation of the $L_B$ program into a form which will “run” the compiling system.

The base language $L_B$ is the language in which we describe all of the manipulations which are performed on the data representing the source text to produce, in the end, the data representing the machine (or other) code result. The data being manipulated consists of a collection of tables in which are recorded the relevant properties or attributes of the items or symbols being manipulated (identifiers, literals, operations, and the like) and a list of symbol descriptors. The list of symbol descriptors represents the program text being compiled in any of its intermediate forms: string of source characters, list of lexical units, syntax tree, sequence of pseudo-instructions, sequence of machine instructions, and so on. Thus during syntactic analysis, an identifier will be represented by a symbol descriptor which points to the symbol table entry for the identifier, plus an indication of its “current syntactic type” (primary, term, expression, etc.); a computation may be represented by a symbol descriptor which points to a sequence of pseudo-instructions comprising the computation, again plus an indication of syntactic type. We may think of a symbol descriptor as a quadruple of integers: a table code and the line number within the table which contains the properties of the symbol; a type field containing current syntactic type; and a single bit indicator which we shall term large indicating whether or not the descriptor is the last argument of some pseudo instruction.

Language $L_B$ can be thought as containing, as a sub-language, a more-or-less standard algebraic language including assignment statements, relations, if-then-else statements, control transfer statements, statement labels, a block structure similar to ALGOL or PL/1, and the like. The numbers which it manipulates are integers which are fields of tables, fields of descriptors, literals, working variables, and so on. $L_B$ has, in addition to the algebraic sub-language, a language for performing pattern matching and replacement over sequences of symbol descriptors. Much of the syntactic analysis and the code selection portions of a compiler are written using the pattern-replacement part of $L_B$, however, there is a complete algebraic language available when it is desired—for handling declarations, determining fixed point scaling, error recovery, preparing relocatable binary output, and so on.

The descriptive language $L_D$ is a declarative language in which the syntax of $L_P$ and a certain portion of the “semantics” of $L_P$ are given. We might think of $L_D$ as a syntax-describing language such as BNF to which we have added facilities for representing various manipulations corresponding to the different syntactic constructions.

The compiling system is that collection of programs and data, under an executive system, which can carry out all the language processing, language executing, and data handling sketched above. Specifically, there will be provisions for:

1. translation of language $L_D$ into language $L_B$, and subsequent editing, modification, and so on of a language $L_B$ program;
2. translation of language $L_B$ programs into appropriate table layouts and code (machine code, interpretive code, some mix of machine and interpretive, etc.);
3. a means for executing the code resulting from (2) (e.g., an interpreter);
4. a lexical analyzer which performs as suggested above; and
5. an executive or operating system which can attend to all the details of inputting, filing, allocating, loading, binding, sequencing, outputting, and the like which might be required.
SYNTACTIC ANALYSIS

Before discussing, in the next section, the descriptive language $L_D$ in which the syntax and, to some extent, the semantics of the programming language $L_P$ are provided, it is important to understand how the syntactic analysis is accomplished in our compiler model and how that portion of the language $L_B$ program representing the syntactic analysis may be derived (mechanically) from syntax rules encoded in language $L_D$. For the purpose of this section we can think of the syntax language as BNF with a few notational changes (to make the representation of the analysis program in language $L_B$ easier) plus a minor extension. The notational changes are as follows: for syntactic categories we will use simple identifiers, eliminating the $[ ]$ brackets normally used with BNF; the terminal symbols of the language will be singly quoted. The minor extension is to allow the declaration of those syntactic categories which are considered tokens—that is, recognized automatically by lexical analysis. An example including simple expressions over integers and an assignment statement is:

```
TOKENS (IDENT, INTEGER)
PRIMARY ::= IDENT | INTEGER
FACTOR ::= PRIMARY | '(' EXPR ')' 
TERM ::= FACTOR | TERM '*' FACTOR
EXPR ::= TERM | EXPR '+' TERM
ASSG ::= IDENT '=' EXPR
```

Syntactic analysis will be accomplished by transforming syntax rules such as the above into a reductions analysis program in language $L_B$. We cannot give a transformation which will work for an arbitrary grammar; the technique which we will discuss depends upon the language being a precedence language. One might ask at this point why we do not utilize one of the standard "syntax driven" (predictive) analyzers to perform the syntactic analysis. Our reasons are, basically, that error recovery is much more easily handled with the scheme proposed here and that once a language has been verified as being a precedence language, we are certain that the language is not ambiguous. Our feeling is that programmers who are provided with facilities to extend the syntax of a language should also be given the measure of protection that guaranteed nonambiguity provides.

Precedence languages (as distinct from operator precedence languages) were introduced by Wirth and Weber and include most of the current programming languages. The basic idea of a simple precedence language is a language such that each pair of syntactic categories and/or terminal symbols enjoy at most one of the three relations: $=$ (equal), $<$ (yielding), or $>$ (taking) precedence. Any pair of symbols which appear adjacent in some syntactic rule have equal precedence; any terminal symbol appearing left adjacent to a syntactic category symbol in some rule yields precedence to all the symbols which can be leftmost symbols of any construction of that category, and so on. In the example above some of the precedence relations are:

\[
\begin{align*}
'( & = EXPR = ')' \\
TERM & = 'e' = FACTOR \\
'\ast' & < PRIMARY \\
'\ast' & < '(' \\
FACTOR & > 'e' \\
\end{align*}
\]

e tc.

1 since PRIMARY and '(' may be the leftmost symbols of a FACTOR
2 since FACTOR may be the rightmost symbol of a TERM

Details are provided in Refs. 10 and 11.

Given a simple precedence language, we can parse a string (program) in that language, left to right, as follows: suppose that the given string (of terminal or token symbols) is $I_1, I_2, \ldots, I_M$ and that at some time during the parse we have reduced this string to

\[ P_1 \ldots P_N / I_1 \ldots I_M \]

where $P_1 \ldots P_N$ are terminal or syntactic type symbols comprising the "parse stack" and $I_1 \ldots I_M$ are the input symbols which have not yet entered into the analysis (except for supplying right context). Then the next step depends upon whether $P_N > I_j$ on the one hand or $P_N = I_j$ or $P_N < I_j$ on the other ($P_N$ not having any precedence relation with $I_j$ indicated an error in the input string). That is, if $P_N > I_j$ then there exists a phrase $P_1 \ldots P_N$ determined by $P_{i-1} < P_1 = P_{i+1} = \ldots = P_N$, and, by this, we mean that there exists some syntactic category $R$ and syntax rule

\[ R ::= P_1 \ldots P_N \]
We "reduce" the string by

\[ \begin{align*}
P_1 \ldots P_{i-1} P_i & \rightarrow P_1 \ldots P_{i-1} R / I_j \ldots I_M \\
or \quad P_1 \ldots P_N & \rightarrow R / \ldots
\end{align*} \]

and proceed, comparing R and Ij. In the other case, 
P_{i-1} < I_i \text{ or } P_{i-1} = I_i, \text{ we effectively increment } N \text{ and \ j by one and set } P_{N+1} = I_j,=i.e. bring the next input symbol (Ij) into our "parsing stack"; the "reduction" can be depicted:

\[ \begin{align*}
P_1 \ldots P_N / I_j I_{j+1} \ldots I_M \\
or \quad P_N / I_j \rightarrow P_N I_j / I_{j+1} \ldots I_M
\end{align*} \]

A language which is not a simple precedence language may be a higher order precedence language. That is, a language in which, in general, the precedence relations hold, not between pairs of single symbols, but between strings of symbols, where the number of symbols in the left (right) string does not exceed some constant m(n). For minimal m,n a language is said to be an (m,n)-precedence language ((1,1)-precedence thus being equivalent to simple precedence). The parsing scheme sketched above also works for higher order precedence languages so long as the proper number of symbols are inspected.*

Now, let us think of the I's and P's as symbol descriptors. Each I is either a pointer to the terminal symbol table, or is a pointer to the literal or symbol table and contains the appropriate syntactic type code; each P is like an I or is a pointer to a computation and, again, carries the appropriate syntactic type code. Adding some machinery for disposing of the phrases which are encountered, we might have replacement rules (almost "statements" in language L_b) such as

\[ \begin{align*}
\ldots \text{TERM} / \text{*} \ldots & \rightarrow \ldots \text{TERM} \text{*} / \\
\ldots \text{EXPR} \text{+} \text{TERM} & \rightarrow \ldots \text{EXPRSEMIT (PLUS, EXPR, TERM)} / \ldots
\end{align*} \]

where EMIT (PLUS, EXPR, TERM) causes "output" of a PLUS pseudo-operation with two arguments, these being whatever descriptors were successfully typed EXPR, and TERM in the pattern, and returns as result a descriptor of the output (i.e.,

one having the table code of the "computation" table and line being where the PLUS was placed in the "current output area" of the computation table); the prefix EMIT$ indicates that the result is to be syntactically typed (i.e., coded in its type field) EXPR.

It is a straightforward task to generate a set of such replacements from the syntax rules for a (generally higher order) precedence language. Usually (see Ref. 11 for details), we have one replacement corresponding to each alternate construction of each syntactic type, plus a replacement for each possible following symbol in order to "move" input into the parsing stack, all ordered appropriately to insure that reductions are not made prematurely. If we have the complete set of rules plus some mechanism to scan all the patterns to find the one applicable and then to make the corresponding replacement, we have a syntactic analyzer. However, it is better to introduce some control, allowing us to avoid looking at all patterns each "cycle", by means of the following techniques:

1. Grouping (and ordering) all replacements with the same symbol "on top of the parse stack" (i.e., immediately left of the /);
2. Attaching a label to the first replacement rule of each group.
3. Introducing the following "actions" which can follow a replacement rule: TRY (label) which indicates that the pattern indicated is to be tried next; ERROR which announces an error, and EXIT, which terminates the process.
4. Following each rule with TRY(t) where t is the label corresponding to the new top of parse stack (resulting from the replacement), or with ERROR or with EXIT if an error condition maintains or the "largest syntactic type" has been recognized.

Now presuming a control mechanism which, "pointing" to some replacement, tries the pattern and, upon success, performs the replacement and the action(s) following and, upon failure, tries the next pattern, we have a language L_b program for syntactic analysis. The program for the simple example given above is (where we have taken a few liberties to show the kind of program our language L_d to L_b translator would produce):

* Note that m,n are maxima for the entire language; for "most" symbol pairs it will probably be sufficient to inspect just those symbol pairs, and not strings of symbols of which the left (right) is rightmost (leftmost).
The BNF-like syntax language introduced in the previous section is our starting point for language $L_D$. That is, language $L_D$ admits data descriptions of the various data (tables and the like), declaration of pseudo-operations, and so on to be included in the compiler; allows specification of the syntax of language $L_P$ in the above notation; and has provisions for the specification of “semantics”, the various actions which are to be taken whenever a certain syntactic construction is found (i.e., the reduction is performed). The $L_B$ program which results from translation of an $L_D$ program contains the $L_D$ data declarations intact plus a reductions analysis program within which the “semantics” have been imbeded for analysis of language $L_P$ program text. We will describe $L_D$ in this section by presuming a “basic” $L_D$ which allows specification of syntax rules in a manner similar to that sketched in the previous section and assume it has provisions for handling data descriptions and, given this basic $L_D$, we will motivate and describe a number of “extensions”.

The first extension is to allow the attachment of “semantics” or “interpretation of the parse” to each syntactic construction. This is accomplished by adjoining to a construction a bracketed “interpretation” or “output specification”. Using the previous example we might write:

```plaintext
TOKENS (IDENT, INTEGER)

PRIMARY ::= IDENT | INTEGER

FACTOR ::= PRIMARY | 'IDENT' [IDENT]

TERM ::= PRIMARY | 'INTEGER' [INTEGER]

EXPR ::= TERM | 'PLUS' [TERM]

ASSG ::= IDENT = IDENT [IDENT]
```

The $L_D$ to $L_B$ translator will arrange for the appropriate prefixing (e.g., $EXPR$EMIT(PLUS,EXPR,TERM)) and “type promotion” when no bracketed interpretation is given (e.g., ...PRIMARY / ... $FACTOR$PRIMARY / ... ). We note here that it is not necessary that pseudocode output be specified; an $L_D$ description which would output a “syntax tree” is given by:
DEFINITIONAL FACILITIES INTO HIGHER LEVEL PROGRAMMING LANGUAGES

TOKENS (IDENT, INTEGER)

PRIMARY ::= IDENT [EMIT(IDENT)] | INTEGER [EMIT(INTEGER)]

FACTOR ::= PRIMARY [EMIT(PRIMARY)] | ('EXPR') [EMIT('(', EXPR, ')')]

TERM ::= FACTOR [EMIT(FACTOR)] | TERM ' *' FACTOR [EMIT(TERM, '*', FACTOR)] and so on.

We now proceed to introduce another extension. Matching some syntactic construction is not always enough; that is, there may exist identical constructions of several different syntactic types (e.g. identifier in ALGOL). We thus allow a predicate to be attached to a syntactic construction and arrange that both the matching of the pattern (i.e., recognition of that construction) and the truth of the predicate are required in order that the corresponding reduction be made. A predicate is any Boolean expression in language LB which involves "and", "or", and "not" combinations of relational over the integers, fields of tables, and working variables which have been declared. As an example we might have

REALVAR ::= IDENT WHEN DATATYPE(IDENT) EQ REALTYPE;

INTVAR ::= IDENT WHEN DATATYPE(IDENT) EQ INTTYPE;

and so on.

Here IDENT in the context DATATYPE-(IDENT) is understood to mean the line field (index into the symbol table) of the descriptor syntactically typed IDENT; and DATATYPE () has presumably been previously declared as a field of the symbol table.*

The next extension is to augment the language by allowing "computations" (in language L₀) to be attached to a construction—computations to be carried out as the corresponding reduction is performed. Again for example, we might have

DECL ::= 'REAL' IDENT DO DATATYPE(IDENT) = REALTYPE; | 'INTEGER' IDENT DO DATATYPE(IDENT) = INTTYPE;

and so on.

Another extension, this time to the way in which we can write a syntax rule, will help alleviate some of the trouble and confusion which recursive syntactic constructions sometimes introduce. Consider for example, the two rules

EXPR ::= EXPR | EXPR ' + ' TERM

ARGLIST ::= EXPR | ARGLIST ',' EXPR

The first construction may be entirely proper and desirable; that is, it specifies the left associativity of the addition operator as well as indicating that '+' has lower "binding power" than the operators (presumably '*' etc.) used to make TERMS. However, the second construction is generally confusing in that the expressions making up an argument list are "equal" = i.e., have neither left nor right grouping. A preferable syntactic form would be

ARGLIST ::= EXPR {',' EXPR}

where {...} is interpreted "choose zero, one, two, etc. occurrences of ... ". We would like the result of recognition of this construction to be a list of the EXPRs recognized with which we can transact further. We extend L₀ by admitting such constructions as

ARGLIST ::= EXPR {',' EXPR} [EMIT(LIST, EXPR)]

where LIST is a defined pseudo-operation and EXPR in the context EMIT(LIST, EXPR) denotes a (first-in-first-out) queue of the descriptors successively recognized as EXPRs. Language L₀ has facilities for dealing with such lists: counting elements, accessing individual or groups of elements, and so on.

As another example we might have (for reasons which will become clear when we discuss computational macros)

REF ::= IDENT [EMIT(IDENT)] | IDENT ('EXPR {',' EXPR}') [EMIT(IDENT, EXPR)]

Finally, we require an extension which will help us to recover from errors detected by the syntactic

* It should be noted that we are not intending to use this predicate mechanism to discriminate various data types—we are here assuming that lexical analysis assigns the appropriate data (i.e., syntactic) type automatically from information in the symbol table. We also note that the division of the testing into a determination of structure followed by application of predicates to certain structural elements is similar to the scheme employed by Fenichel in the FAMOUS System.12
analysis. We note that the language \( L_D \) to \( L_B \) translator will supply rules of the form

\[
\rightarrow \text{ERROR}
\]

as the last rule of a group which may not otherwise contain a rule guaranteed to be successful. The extension is to admit statements of the form

\[
\text{OTHERWISE (comp)} \text{ stuff}
\]

where "comp" is any syntactic type or terminal and "stuff" contains interpretations, predicates, or computations similar to any correct construction; the \( L_D \) to \( L_B \) translator will insure that the "stuff" is done rather than an error being signalled. An example:

\[
\text{OTHERWISE (EXPR) DO BEGIN}
\]

\[
\begin{align*}
\text{TRY(EXPR. R)} \\
\text{EXPR. R} & \rightarrow \ldots \left(\text{EXPR} / \ldots \right) \\
\rightarrow & \ldots \left(\text{EXPR}^* / \ldots \right) \text{TRY(RPAR. T)} \\
\rightarrow & \text{PRINT ("INCORRECT OPERATOR FOLLOWING EXPRESSION")}; \\
\text{TRY(RECOVER)} & \text{ END}
\end{align*}
\]

what we are trying to suggest here are several things, to wit:

1. \( L_B \) has a facility for printing (error) comments;
2. replacement rules can be directly programmed as "computations" in \( L_B \);
3. a series of "computations" can be bracketed by BEGIN END brackets; etc.

A stronger way of putting this is that our "operational expectation" is that much of the syntax and semantics of an \( L_P \) can be provided via \( L_D \) and result automatically in an \( L_B \) program for syntactic analysis and interpretation of the parse. However, since the "final" program is in \( L_B \) we are free to modify, extend, etc. the \( L_B \) program to account for syntactic vagaries (non-precedence situations), error recovery, and the like either by modifying the \( L_B \) program resulting from \( L_D \) translation or by inserting this kind of thing into \( L_D \) by the device sketched above. Thus one could have a more-or-less "pure" \( L_D \) description of a language available to a user as a reference document; the details of error recovery, handling of special cases, and so on might be represented only in a modified version of the \( L_B \) program resulting from translation of the \( L_D \) program, and supplied only to "experts". The point is that both declarative and imperative versions of the syntactic analysis and parse interpretation are available in a similar form, surely an aid to documentation.

**ADDITION OF DEFINITIONAL OR "MACRO" FACILITIES TO A LANGUAGE**

With the compiler model and other machinery outlined above in mind, we now turn to the question of imbedding the macro definition and call facilities into a given language \( L_P \). We will organize our discussion around the time at which the macro facilities are employed and thus discuss

- **Text Macros**—which are employed prior to lexical analysis;
- **Syntactic Macros**—which are called during syntactic analysis; and
- **Computational Macros**—which are called subsequent to syntactic analysis.

We now proceed to a discussion of each of these three types.

**Text Macros**

By text macros we mean macro facilities which, from the point of view of our compiler model, are contained in the interface which supplies program text to the lexical analyzer. Indeed, one might take either TRAC \(^1\) or GPM \(^2\) (they are very similar devices) as perfectly satisfactory pre-lexical macro devices essentially as they stand. In any event, since these kinds of macros are really outside the scope of the compiler model we have described and, particularly, since two excellent macro systems which are sufficient for this purpose have been described in the literature, we shall not discuss this type of macro further.

**Syntactic Macros**

What we have in mind here is a facility which allows one to define what are in effect new syntactic structures in terms of the given syntactic structures of the language \( L_P \) and previously defined "syntactic macros". We require three things: a syntactic augment to \( L_P \) to allow the definition and call of syntactic macros; a mechanism, within our syntactic analyzer, for "recording" the definition of the syntactic macros; and, a mechanism within our syntactic analyzer (and lexical analyzer) to handle the call of these macros.
The definition of syntactic macros will equate a “macro form” which is some syntactic structure—a string of terminal and token symbols mixed with identifiers, previously declared as formal parameters bearing a given syntactic type—with a “defining string” which is a string of characters intermixed with the formal parameter names. Upon recognition of the macro form, the syntactic analyzer will submit the corresponding defining string to the lexical analyzer after placing the actual parameters occurring in the macro call on an argument list. The lexical analyzer will then lexically analyze the defining string in a normal fashion with the exception that occurrences of the formal parameter names (as identifier tokens) will result in the output of the descriptor corresponding to the actual parameter (already processed and stored on the argument list) rather than a descriptor for that identifier. This lexical output will precede the normal lexical output (or previously arranged macro output, if the macros are defined recursively). Note that we presume that the lexical analyzer deals only with a single descriptor of the actual parameter, however complex (syntactically) it might have been in the macro call, and merely substitutes this descriptor for the occurrence of the corresponding formal parameter in the defining string, nothing more. The computation tree or whatever resulted from the analysis of the actual parameter within the macro call is not touched.

We will assume that syntactic macro definitions occur at the head of a program (or of a block, if you want), since the syntactic analyzer may be considerably modified (re-constructed) by such definitions and it might be prudent to restrict such operations to the beginning of a compiler run. (Clearly, generally useful syntactic macro extensions to a compiler could result in the “extended” compiler being filed under some new name to save the overhead of re-constructing the compiler “on the fly” each time it is called.)

Let us now propose an extension of the syntax of L_p to allow definition and call of two varieties of syntactic macros. (Each particular L_p would doubtless be handled somewhat differently; the intention here is only to propose something we can make reference to in the sequel). For this we will use the BNF notation:

```
s-macro-parameter-decl ::= LET [identifier] BE [syntactic type]
s-macro-definition ::= MACRO [macro form] MEANS ['defining string'] |
```
to the lexical analyzer and the actual parameters with their associated names being placed on an argument list. The % [macro call] "phrase" will then be completely wiped off the parse stack and syntactic analysis will continue. Note that % is a special terminal which takes precedence over any symbol which might appear immediately to the left of it (to insure that the macro is expanded to provide the proper right context for the symbol preceding the % in the L_p text). Example

LET N BE INTEGER
MACRO MATRIX (N) MEANS 'ARRAY(1:N, 1:N)' is essentially equivalent to a syntactic definition of the form

MACRO::='%' 'MATRIX' '('INTEGER')'

where the macro syntactic type is handled differently from other syntactic types in that % takes precedence over everything and recognition of the phrase "% ... " causes elimination of the phrase (after "interpretation") rather than reduction to a single descriptor typed MACRO.

A call of the macro of the form

% MATRIX (25)

in most contexts will result in the equivalent of having written:

ARRAY (1:25, 1:25)

in that context. Similarly

MACRO A MEANS '+B*C+

followed by

X = Y %A Z

will result in the equivalent of having written

X = Y+B*C+Z

The definition and use of the MACRO macros must be done with some care. In particular, the right-most component of the [macro form] must be carefully chosen so that the complete [macro form] will be recognized as a phrase. For example, the right-most component could be a terminal symbol in L_p which will take precedence over the symbol following the macro call in all contexts in which it might appear. The use of an identifier which is treated as a new terminal symbol as right-most component would eliminate any trouble, since its precedence will automatically be taken as greater than any symbol which may succeed it. Also, one could introduce a special right delimiter (matching the % left delimiter) and insist on both left and right delimitation of the MACRO macro call.

We note that the text macros and the MACRO macros are similar in that both require some sort of trip character to announce their call. Indeed, in many situations the choice of one type over the other might be quite arbitrary. However, there are situations in which the MACRO macros are clearly preferable. One situation is when the verification of the correct syntactic type of the actual parameter is desired prior to the expansion of the macro, for example to allow better control in announcing to the user the exact circumstance in which an error occurred. We note also that the form of macro call for text macros would very likely be quite rigid (e.g., # (name, arg, ..., arg) in TRAC and § name, arg, ..., arg; in GPM) while the form of MACRO macro calls is not restricted except for the position of the trip character, %.

There are many situations in which neither the text nor the MACRO macros are adequate. In particular, the requirement for a trip character in the call required to keep the syntactic analysis from going askew may be distasteful in many cases. The SMACRO form of syntactic macro differs from the MACRO form in that a syntactic type of the macro form is provided (AS[synthetic type]) and thus the call of an SMACRO can be accomplished without using the special % character. While the declaration of the MACRO and SMACRO macros and the handling of their calls, once they have been recognized, are very similar, the actual recognition of the two kinds of macros is quite different. That is, the MACRO macro call has the % character to trigger recognitions while with the SMACRO macro calls the handling of the recognition is just as though the macro had been declared (in L_p) as another construction of the given syntactic type and thus enjoys all the attendant advantages and disadvantages. As an example we can write the first example above as:

LET N BE INTEGER
SMACRO MATRIX(N) AS ATTRIBUTE MEANS 'ARRAY(1:N, 1:N)'

and call the macro via, for example:

... MATRIX(25) WALDO ...

so long as the calling string is in an allowed context for the (presumed previously defined for L_p) syn-
DEFINITIONAL FACILITIES INTO HIGHER LEVEL PROGRAMMING LANGUAGES

The macro definition in $L_0$ of the form

\[
\text{ATTRIBUTE} ::= \text{`MATRIX'} \text{`(`INTEGER')'}
\]

and accompanying interpretation or semantics accomplishing whatever $\text{ARRAY}(\ldots)$ accomplishes.

As another example of the SMACRO form, suppose that $L_P$ has arithmetic expressions (EXPR), relation operators (RELOP), and relations (RELATION) as syntactic types, and also allow ANDing and ORing of relations, where the syntax for RELATION was given as

\[
\text{RELATION} ::= \text{EXPR} \text{ RELOP } \text{EXPR}
\]

with appropriate accompanying interpretation. By the next recognition of the macro form the parse stack would be

\[
\text{EXPR} \text{ RELOP } \text{EXPR} \text{ RELOP } \text{EXPR}
\]

\[
\downarrow \quad \downarrow
\]

\[
\text{PLUS } A = 1 \quad \text{PLUS } C = 1
\]

and so on.

It is clear that some measure of care must be taken in the use of syntactic macros and it is implicitly assumed here that this kind of tool would probably be placed only in the hands of a reasonably sophisticated user. We do, however, submit that the methods incorporated here for syntactic analysis would allow not only "catching" but more-or-less reasonably useful "commenting" upon violations to the given syntax (via $L_0$) by poor use of the SMACRO facility.

By virtue of the particular technique which we propose to utilize for syntactic analysis it appears possible, by saving an appropriately encoded representation of the syntax rules and the precedence relations for $L_0$, to handle syntactic macros by "incremental compiler changes" if the representation of the $L_8$ program which "runs" the reductions analysis is as appropriately organized coding for an interpretive system.

It would, in principle, be possible to allow the attachment of more "semantics" to a [macro form] than just the [defining string]. That is, language $L_D$ (or $L_8$) fragments appropriately delimited could presumably be handled along with or in place of the [defining string] by utilizing the presumed facilities of the compiling system to reconstruct the compiler even while it is processing $L_P$ programs containing $L_8$ statements. However, this approach seems practically infeasible and we are led to propose some "built in macros" to accomplish this function for some of the more "practical" cases; we will discuss one of these and mention a few others to give the flavor.

There is one particular facility which would be extremely useful and which cannot be handled by the syntactic macros (except by some real trickery in defining $L_P$ originally) as defined above, namely the introduction of new data types. The problem is that these would necessitate the ability to make appropriate symbol table (type field) entries. We thus propose the following "built in" macro, called via

\[
\% \text{DEFINETYPE} ([\text{identifier}], [\text{syntactic type}])
\]

which associates the given syntactic type code (very likely a new syntactic type rather than one of the syntactic type given for $L_0$) with the [identifier] in the symbol table.

Given this, plus the ability to "talk about" these type codes via

\[
\text{LET } [\text{identifier}] \text{ BE } [\text{syntactic type}]
\]

\[
\text{SMACRO } [\text{macro form}] \text{ AS } [\text{syntactic type}] \text{ MEANS } '[\text{defining string}]'
\]

we can then deal with definition and manipulation of new data types. Other built in macros would include some allowing the augmenting of the symbol table by new fields and subsequent reference to those fields, access to and provision for manipulation of those fields of the symbol tables (or other tables) which contain data allocation information, and so on. Ref. 13 describes an extensible version of ALGOL-60 based on the ideas in this paper and contains several elaborate examples of the use of the various macro facilities.

Computational Macros

The idea of computational macros is quite straightforward and their use within a compiler is quite cheap (as compared with the manipulations required to handle syntactic macros). The idea is this: we may optionally associate with any Identifier a sequence of pseudo-instructions (actually a computation tree) which contain, in general, some formal parameters as arguments. The completed processing of the "reference" (i.e., the identifier and all the expressions comprising its actual argument list) can then be followed by the macro expansion, i.e. the replacement of the reference by a copy of the computation tree with each formal parameter argument replaced by the corresponding (completely processed) actual pa-
parameter. The computation tree, or macro skeleton, would be defined by associating with the macro name (the identifier whose occurrence in a reference will trigger the macro expansion) the result of syntactically analyzing and interpreting an expression \* which is the “defining string” for the macro. Note here that one difference between the syntactic and the computational macros is that the defining string for syntactic macros is not touched until a macro call, at which time it looks (to within parameter substitution) like raw input; on the other hand, the defining string for a computational macro is analyzed and interpreted into a series of pseudo-instructions on first encounter.

Again, we require three things to handle such macros: a syntactic augment to \(L_p\) to allow the definition of the macros; a mechanism within the syntactic analyzer for "recording" the definition; and, a mechanism for recognizing and effecting the call of the macro. In the above discussion of the presumed symbol table contents and the handling of references we left some connections for handling these macros. First, let us propose the syntactic augments to \(L_p\), to wit:

\[
\text{[c-macro-parameter-dec]} ::= \text{TAKE} \ [\text{identifier}] \ \{\text{AS} \ [\text{syntactic type}]\};
\]

\[
\text{[c-macro-definition]} ::= \text{MAP} \ [\text{identifier}] \ \{([\text{parameter}] \ , [\text{parameter}])\} = \ [\text{expression}] \ \{\text{UPON} \ [\text{syntactic type}]\};
\]

Leaving out the optional AS and UPON parts for the moment, we assume that the “TAKE [identifier];” declares the identifier to be a formal parameter for subsequent definition of c-macros. The identifier in the MAP is assumed to be a variable (optionally subscripted) which is declared in the \(L_p\) program; the [parameter]s are formal parameters. (We could, of course, eliminate the actual declaration of the formal parameters and let their occurrence in the MAP definition indicate their formal parameterhood; subsequent remarks on the optional AS syntactic type will show why we have not chosen to do this.) The expression will be presumed to be any (arithmetic or other) expression allowable in \(L_p\) and will generally contain occurrences of the formal parameters. The mechanisms are then roughly as follows: The [c-macro definition] causes the expression to be syntactically analyzed and interpreted, resulting in a sequence of pseudo-instructions; the only augment required to the syntactic analyzer is to handle the formal parameter identifiers occurring by replacing each with some appropriate descriptor, \(\dagger\) a completely trivial matter. Further, the symbol table entry for the identifier is set to indicate that it names a macro and a pointer is inserted to point to the computation tree produced for the expression. Assume that any reference say [identifier] {[expr] {[expr]}} is then syntactically analyzed to produce a “pseudo-instruction” where the [identifier] descriptor is the “pseudo-operation” and the [expr] descriptors follow it as “arguments”.

Then, noting from the symbol table entry for the [identifier] that it is a macro, the (macro skeleton) computation pointed to by this entry is copied with appropriate replacement of formal parameter descriptors by the actual parameters in the manner suggested above. In the event that references occur within the mapping computation, the mechanism will recursively expand these, and so on.

As an example, suppose that \(A, B, G\) are declared as 2, 1, 0 dimensional arrays; then a computational macro for the "mapping functions" for \(B\) and \(A\) might be given by:

\[
\begin{align*}
\text{TAKE} & \ I; \\
\text{TAKE} & \ J; \\
\text{MAP} & \ B(j) = j^2 + G; \\
\text{MAP} & \ A(i,j) = 25i + B(j+1); \\
\end{align*}
\]

The result of processing these might result in symbol table and descriptor table entries as follows:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Map Type</th>
<th>Macro Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>Normal</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Macro</td>
<td>4</td>
</tr>
<tr>
<td>A</td>
<td>Macro</td>
<td>15</td>
</tr>
<tr>
<td>1: TIMES (\pi_1) = 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4: PLUS (\dagger) G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7: TIMES = 25 (\pi_1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: PLUS (\pi_2) = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13: B (\bigcirc)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15: PLUS (\bigcirc)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here \(\pi_1\) stands for a descriptor of formal parameter \(j\), circled integers refer to previous pseudo-instructions, = 25, etc. indicates the literal 25, etc. A reference to \(A(X+Y,Z)\) then might be parsed to yield:

\[
\begin{align*}
100: & \ \text{PLUS} \ X \ Y \\
103: & \ A \ (100) \ Z
\end{align*}
\]

\(\dagger\) e.g. assume a "formal parameter table" (without content) and use table code to indicate formal parameter-hood and line field to indicate which one.

---

\* Or, more generally, a complete procedure so long as the procedure returns a value and the code selection phase of the compiler can cope with full procedure calls where a variable normally appears.

---
DEFINITIONAL FACILITIES INTO HIGHER LEVEL PROGRAMMING LANGUAGES 635

and then expanded to yield:

106: PLUS @
109: TIMES = 25 100
112: PLUS Z = 1
115: PLUS G
118: TIMES Q!P = 2
122: LOCATE B dB G
126: LOCATE A d A

Here the LOCATE pseudo-instructions indicate a reference to the data element indicated by the first argument and where the constant and variable parts of the displacement from the base of the area containing the values for that data element are given by the second and third arguments.

As another example, consider the following mapping function which provides contiguous storage for a 4 x 4 (upper left) triangular matrix M ( ):

TAKE I;
TAKE J;
MAP M(I,J) = (11-I) * 1/2 + J - 6;

This example raises the whole set of issues to do with storage allocation (presumably the declaration of M as a 4 x 4 array would have resulted in 16 words being reserved for it rather than the 10 we desire), providing, filing, and subsequently obtaining and incorporating into the compiler the description of “global” data, and so on. While we cannot go into these issues here, Ref. 14 describes a rather elaborate data declaration facility which we have added to a subset of PL/I, using the computational macro techniques described above, and Ref. 13 discusses the handling of such allocation within the framework of ALGOL-60.

As a final example which illustrates the use of computational macros for other than data mapping applications, consider the following:

TAKE X AS EXPR;
MAP F(X) = SIN(X) + COS(X);

Now let us touch briefly upon the use of syntactic types in declaring the formal parameters for computational macros and in defining the macros themselves. First, unless the formal parameters are explicitly syntactically typed we will assume that they are [integer-expression]s (or the equivalent). This will allow proper syntactic analysis of mapping expressions, generally the most usual use of computational macros. In general, however, it is clear that syntactic type must be given in order that the expression (which may be a Boolean or string, etc. expression) can be parsed. As to the use of the UPON [syntactic type] facility, we will assume that any given identifier may have a number of interpretations (e.g. as a floating as well as an integer valued variable or as a label as well as an integer array name) and there will be a symbol table entry for each interpretation (MAP) with the corresponding “mapping release” syntactic type recorded. A unique interpretation is chosen on the basis of the syntactic type of the reference by matching this “mapping release” syntactic type.

If the control of when references are “expanded” is vested in some L/# action which is called at the statement or some other level, then a computation tree can be assembled and the mapping expansion of the whole tree “fired off” allowing the mapping release type to be matched not only against the syntactic type of the reference but against those of its ancestors, thus allowing reasonably fine contextual control of different mappings for the same identifier. The stack example in Reference 13 illuminates this notion.

There is another issue which should be mentioned but which will not be pursued very far in the present paper. This is the issue of argument types and numbers of arguments expected of references as they relate to the question of selection of proper mapping (i.e., interpretation). We will presume that the symbol table entry for an identifier, expected to appear with arguments, contains a description of each of these arguments. It would be desirable (it is actually done in the system implementing the language described in Ref. 14) to have an argument matching involved in the searching and decision making to select the “proper” interpretation of a given reference. It is possible to envision a table of actual argument type versus desired argument type containing “degradation factors” (per identifier, per block, per program, or for all the time) to be applied to the measure of desirability of each interpretation of a given reference. Such a table, in conjunction with the mapping release syntactic type, would provide a quite sophisticated criterion for the selection of one from a number of possible interpretations, as well as allow automatic specification of argument conversions, and so on.

It should be remarked that to properly handle such schemes it is desirable to introduce the idea of syntactic classes into the syntax language to allow “grouping” of syntax categories without change of
syntactic type. For example, using \( := \) to denote a syntactic type as, really, a class, we would interpret

\[
\text{EXPR} ::= \text{INTEGEREXPR} \mid \text{REALEXPR} \mid \text{STRINGEXPR} \mid \text{etc.} \\
\text{ARGLIST} ::= \text{EXPR} (',', \text{EXPR})
\]
as allowing an ARGLIST of different kinds of expressions each retaining their peculiar syntactic types, in order that subsequent inspection of the elements of the list as arguments would have the "proper" syntactic type for inspection (to save descending into the computation tree to "dig out" the real type).

**CONCLUSION**

Let us briefly reiterate the point of view which this paper has developed. We have proposed a compiler model, a compiling system, and two programming languages, \( L_D \) and \( L_B \), with which a particular compiler should be more-or-less readily constructable and which result in a program whose efficiency is determined largely by the trouble we choose to take in providing an optimized translation* of language \( L_B \). We have further suggested three types of definitional facilities which will provide the user with a kit of tools to extend a given language to more closely mirror the natural means of exposition in the application area in which he is involved. We want particularly to emphasize that extensions to the user's language need not be based entirely on exploitation of the macro facilities. Indeed, we have left two handles: the ability to add new built in macros and the ability to fall back to language \( L_D \) and rewrite (some portion of) the compiler. Given that the pseudo-operations available provide adequate primitives for representing the extensions desired (and that the optimization and code selection facilities are adequate) this latter facility should be useable by users—not just by compiling system buffs.

As to the status of the various languages, translators, and other facilities discussed in this paper, we do not, at this time, have in hand a complete system with a compiler for some user programming language augmented by all three kinds of macro facilities. On the other hand, these proposals are not merely represented as flights of our fancy, to wit:

1. A compiling system called TRANGEN, and a realization of \( L_B \), called TRANDIR, have existed for some years (and gone through five major language changes and system implementations).\(^6,7\) The TRANDIR/TRANGEN facilities have been used for a number of compilers (PL/1, ALGOL, TRANDIR, FORTRAN IV, a data description language, etc.) on a number of computers (IBM-7094, CDC-1604, GE-635, M-490, etc.)

2. A ("reference" language) version of language \( L_D \) has existed for some time\(^11\) and has been used to translate, by hand, a specification of a language \( L_D \) description of an \( L_P \) into a reductions analysis program in TRANDIR.

3. Text macro facilities are currently available (but not necessarily "hooked in" to a compiler).\(^1,2\)

4. Computational macros have been utilized in a number of compilers; indeed we have developed a number of data description languages based on this idea.\(^14\)

Finally, we ought to remark very briefly on the relation of the SMACRO macro facilities proposed herein to the "definitions in ALGOL" proposed by Perlis and Galler\(^5\) since their proposal is, to our knowledge, the only scheme of any generality described in the literature for syntactic macros. Our two approaches would appear to be reasonably similar. The main differences seem to be three: first, they restrict the syntactic categories which will admit extension to [block head], [type], [assignment statement], [arithmetic expression], and [Boolean expression], while we have not adopted such restrictions; secondly, their [macro forms] (our terminology) are more powerful than those proposed here in that they can accommodate multiple instances of the formal parameters in the [macro form] (meaning multiple occurrences of exactly the same structure in the macro call) because they use a "tree matching" scheme for recognizing the macro call while we use matching of (effectively linear) syntactic constructions; thirdly, our [macro form] is more powerful than theirs in that we allow the use of arbitrary syntactic types as parameters of [macro forms] and use

---

* Clearly, the development of more and more optimal translators for \( L_B \) given, initially, a reasonably straight forward translation into "coding" for an interpretive system, is a task which can proceed in parallel with the development of compilers via programs in \( L_D \) and \( L_P \). The effect on the user should be no more than the fact that his compilations may go faster.
these in recognizing a macro call, while they do not ascribe syntactic type to their formal parameters.

In conclusion, it is our very strong feeling that languages with powerful definitional facilities must be placed in the hands of users. Indeed, the appropriate representation—something natural to the individual—of a problem (or solution) has more to do with the issue of solving problems than merely providing a nicety. To those who question this position, I suggest the following problem in long division:

\[ \begin{array}{c|c|c|c}
     & XLVI & & \\
\hline
    & MCXXIV & & \\
\end{array} \]

REFERENCES


