Definition mechanisms in extensible programming languages

by STEPHEN A. SCHUMAN*

Université de Grenoble
Grenoble, France

and

PHILIPPE JORRAND
Centre Scientifique IBM-France
Grenoble, France

INTRODUCTION

The development of extensible programming languages is currently an extremely active area of research, and one which is considered very promising by a broad segment of the computing community. This paper represents an attempt at unification and generalization of these developments, reflecting a specific perspective on their present direction of evolution. The principal influences on this work are cited in the bibliography, and the text itself is devoid of references. This is indicative of the recurring difficulty of attributing the basic ideas in this area to any single source; from the start, the development effort has been characterized by an exceptional interchange of ideas.

One simple premise underlies the proposals for an extensible programming language: that a "user" should be capable of modifying the definition of that language, in order to define for himself the particular language which corresponds to his needs. While there is, for the moment, a certain disagreement as to the degree of "sophistication" which can reasonably be attributed to such a user, there is also a growing realization that the time is past when it is sufficient to confront him with a complex and inflexible language on a "take it or leave it" basis.

According to the current conception, an extensible language is composed of two essential elements:

1. A base language, encompassing a set of indispensable programming primitives, organized so as to constitute, in themselves, a coherent language.
2. A set of extension mechanisms, establishing a systematic framework for defining new linguistic constructions in terms of already existing ones.

Within this frame of reference, an extended language is that language which is defined by some specific set of extensions to the given base language. In practice, definitions can be pyramided, using a particular extended language as the new point of departure. Implicit in this approach is the assumption that the processing of any extended language program involves its systematic reduction into an equivalent program, expressed entirely in terms of the base language.

Following a useful if arbitrary convention, the extension mechanisms are generally categorized as either semantic or syntactic, depending on the capabilities that they provide. These two types of extensibility are the subjects of subsequent sections, where models are developed for these mechanisms.

Motivations for extensible languages

The primary impetus behind the development of extensible languages has been the need to resolve what has become a classic conflict of goals in programming language design. The problem can be formulated as

---

* Present address: Centre Scientifique IBM—France
power of expression versus economy of concepts. Power of expression encompasses both "how much can be expressed" and "how easy it is to express". It is essentially a question of the effectiveness of the language, as seen from the viewpoint of the user. Economy of concepts refers to the idea that a language should embody the "smallest possible number" of distinguishable concepts, each one existing at the "lowest possible level". This point of view, which can be identified with the implementer, is based on efficiency considerations, and is supported by a simple economic fact: the costs of producing and/or using a compiler can become prohibitive. Since it is wholly impractical to totally disregard either of these competitive claims, a language designer is generally faced with the futile task of reconciling two equally important but mutually exclusive objectives within a single language.

Extensible languages constitute an extremely pragmatic response to this problem, in the sense that they represent a means of avoiding, rather than overcoming this dilemma. In essence, this approach seeks to encourage rather than to suppress the proliferation of programming languages; this reflects an increasing disillusionment with the "universal language" concept, especially in light of the need to vastly expand the domain of application for programming languages in general. The purpose of extensible languages is to establish an orderly framework capable of accommodating the development of numerous different, and possibly quite distinctive dialects.

In an extensible language, the criteria concerning economy of concepts are imposed at the point of formulating the primitives which comprise the base language. This remains, therefore, the responsibility of the implementer. Moreover, he is the one who determines the nature of the extension mechanisms to be provided. This acts to constrain the properties of the extended languages subsequently defined, and to effectively control the consistency and efficiency of the corresponding compilers.

The specific decisions affecting power of expression, however, are left entirely to the discretion of the user, subject only to the restrictions inherent in the extension mechanisms at his disposal. This additional "degree of freedom" seems appropriate, in that it is after all the language user who is most immediately affected by these decisions, and thus, most competent to make the determination. The choices will, in general, depend on both the particular application area as well as various highly subjective criteria. What is important is that the decision may be made independently for each individual extended language.

At the same time, the extensible language approach overcomes what has heretofore been the principal obstacle to a diversity of programming languages: incompatibility among programs written in different languages. The solution follows automatically from the fact that each dialect is translated into a common base language, and that this translation is effected by essentially the same processor.

Despite the intention of facilitating the definition of diverse languages, the extensible language framework is particularly appropriate for addressing such significant problems as machine-to-machine transferability, language and compiler standardization, and object code optimization. The problems remain within manageable limits, independent of the number of different dialects; they need only be resolved within the restricted scope of the base language and the associated extension mechanisms.

Evolution of extensible languages

An extensible language, according to the original conception, was a high level language whose compiler permitted certain "perturbations" to be defined. Semantic extension was formulated as a more flexible set of data and procedure declarations, while syntactic extension was confined to integrating the entities which could be declared into a pre-established style of expression. For the most part, the currently existing extensible languages reflect this point of departure.

It is nonetheless true that the basic research underlying the development of extensible languages has taken on the character of an attempt to isolate and generalize the various "component parts" of programming languages, with the objective of introducing the property of "systematic variability". A consequence of this effort has been the gradual emergence of a somewhat more abstract view of extensible languages, wherein the base language is construed as an inventory of essential primitives, the syntax of which minimally organizes these elements into a coherent language. Semantic extension is considered as a set of "constructors" serving to generate new, but completely compatible primitives; syntactic extension permits the definition of the specific structural combinations of these primitives which are actually meaningful. Thus, extensible languages have progressively assumed the aspect of a language definition framework, one which has the unique property that an operational compiler exists at each point in the definitional process.

Accordingly, it is increasingly appropriate to regard extensible languages as the basis for a practical language definition system, irrespective of who has responsibility for language development. Potentially, such an environment is applicable even to the definition of non-
extensible languages. Heretofore, it has been implied that any given extended language was itself fully extensible, since its definition is simply the result of successive levels of extension. In conjunction with the progressive generalization of the extension capabilities, however, one is naturally led to envision a complementary set of restriction mechanisms, which would serve to selectively disable the corresponding extension mechanisms.

The intended function of the restriction mechanisms is to eliminate the inevitable overhead associated with the ability to accommodate arbitrary extension. They would be employed at the point where a particular dialect is to be "frozen". In effect, such restriction mechanisms represent a means of imposing constraints on subsequent extensions to the defined language (even to the extent of excluding them entirely), in exchange for a proportional increase in the efficiency of the translator. The advantage of this approach is obvious: the end result of such a development process is both a coherent definition of the language and an efficient, operational compiler.

Within this expanded frame of reference, most of the extensible languages covered by the current literature might more properly be considered as extended languages, even though they were not defined by means of extension. This is not unexpected, since they represent the results of the initial phase of development. The remainder of this paper is devoted to a discussion of the types of extension mechanisms appropriate to this more evolved interpretation of extensible languages. The subject of the next section is semantic extensibility, while the final section is concerned with syntactic extensibility. These two capabilities form a sort of two-dimensional definition space, within which new programming languages may be created by means of extension.

**SEMANTIC EXTENSIBILITY**

In order to discuss semantic extensibility, it is first necessary to establish what is meant here by the semantics of a programming language. A program remains an inert piece of text until such time as it is submitted to some processor: in the current context, a computer controlled by a compiler for the language in which the program is expressed. The activity of the processor can be broadly characterized by the following steps:

1. Recognition of a unit of text.
2. Elaboration of a meaning for that unit.
3. Invocation of the actions implied by that meaning.

According to the second of these steps, the notion of meaning may be interpreted as the link between the units of text and the corresponding actions. The set of such links will be taken to represent the semantics of the programming language.

As an example, the sequence of characters "3.14159" is, in most languages, a legitimate unit of text. The elaboration of its associated meaning might establish the following set of assertions:

- this unit represents an object which is a value.
- that value has a type, which is real.
- the internal format of that value is floating-point.
- the object will reside in a table of constants.

This being established, the actions causing the construction and allocation of the object may be invoked. The set of assertions forms the link between the text and the actions; it represents the "meaning" of 3.14159.

With respect to the processor, the definition of the semantics of a language may be considered to exist in the form of a description of these links for each object in the domain of the language. When a programming language incorporates functions which permit explicit modification of these descriptions, then that language possesses the property of semantic extensibility. These functions, referred to as semantic extension mechanisms, serve to introduce new kinds of objects into the language, essentially by defining the set of linkages to be attributed to the external representation of those objects.

**Semantic extension in the domain of values: A model**

The objects involved in the processing of a program belong, in general, to a variety of categories, each of which constitutes a potential domain for semantic extension. The values, in the conventional sense, obviously form one such category. In order to illustrate the overall concept of semantic extensibility, a model for one specific mechanism of semantic extension will be formulated here. It operates on a description of a particular category of objects, which encompasses a generalization of the usual notion of value. For example, procedures, structures and pointers are also considered as values, in addition to simple integers, booleans, etc.

These values are divided into classes, where each class is characterized by a mode. The mode constitutes the description of all of the values belonging to that class. Thus the mode of a value may be thought of as playing a role analogous to that of a data-type. It is
assumed that processing of a program is controlled by syntactic analysis. Once a unit of text has been isolated, the active set of modes is used by the compiler to elaborate its meaning. Typically, modes are utilized to make sure that a value is employed correctly, to verify that expressions are consistent, to effect the selection of operations and to decide where conversions are required.

The principal component of the semantic extension mechanism is a function which permits the definition of new modes. Once a mode has been defined, the values belonging to the class characterized by that mode may be used in the same general ways as other values. That is to say, those values can be stored into variables, passed as parameters, returned as results of functions, etc.

The mode definition function would be used like a declaration in the base language. The following notation will be taken as a model for the call on this function:

\[
\text{mode } u \text{ is } T \text{ with } \pi;
\]

The three components of the definition are:

1. the symbol clause "mode \( u \)",
2. the type clause "is \( T \)",
3. the property clause "with \( \pi \)."

The property clause may be omitted.

The symbol clause

In the symbol clause, a new symbol \( u \) is declared as the name of the mode whose description is specified by the other clauses. For example,

\[
\text{mode complex is ...}
\]

may be used to introduce the symbol complex. In addition, the mode to be defined may depend on formal parameters, which are specified in the symbol clause as follows:

\[
\text{mode matrix (int } m, \text{ int } n \text{) is ...}
\]

The actual parameters would presumably be supplied when the symbol is used in a declarative context, such as

\[
\text{matrix (4, 5)A;}
\]

The type clause

In the type clause, \( \tau \) specifies the nature of the values characterized by the mode being defined. Thus, \( \tau \) is either the name of an existing mode or a constructor applied to some combination of previously defined modes. There are assumed to be a finite number of modes predefined within the base language. In the scope of this paper, int, real, bool and char are taken to be the symbols representing four of these basic modes, standing for the modes of integer, real, boolean and single character values, respectively. Thus, a valid mode definition might be:

\[
\text{mode integer is int;}
\]

The model presented here also includes a set of mode constructors, which act to create new modes from existing ones. The following list of constructors indicates the kinds of combinations envisioned:

1. Pointers
   A pointer is a value designating another value. As any value, a pointer has a mode, which indicates that:
   - it is a pointer.
   - it is able to point to values of a specified class.

   The notation \( \text{ptr } M \) creates the mode characterizing pointers to values of mode \( M \). For example,

   \[
   \text{mode ppr is ptr ptr real;}
   \]

   specifies that values of mode \( ppr \) are pointers to pointers to reals.

2. Procedures
   A procedure is a value, implying that one can actually have procedure variables, pass procedures as parameters and return them as the results of other procedures. Being a value, a procedure has a mode which indicates that:
   - it is a procedure.
   - it takes a fixed number (possibly zero) of parameters, of specified modes.
   - it returns a result of a given mode, or it does not return any result.

   The notation \( \text{proc } (M_1, \ldots, M_n) M \) forms the mode of a procedure taking \( n \) parameters, of modes \( M_1 \ldots M_n \) respectively, and returning a value of mode \( M \). As an example, one could declare

   \[
   \text{mode trigo is proc (real)real;}
   \]
for the class of trigonometric functions, and then

\[
\text{mode trigocompose is proc (trigo, trigo)trigo;}
\]

for the mode of functions taking two trigonometric functions as parameters, and delivering a third one (which could be the composition of the first two) as the result.

3. Aggregates
Two kinds of aggregates will be described:

a. the *tuples*, which have a constant number of components, possibly of different modes;
b. the *sequences*, which have a possibly variable number of components, but of identical modes.

a. Tuples
The mode of a tuple having \( n \) components is established by the notation \([M_1s_1, \ldots, M_ns_n]\), where \( M_1 \ldots M_n \) are the modes of the respective components, and \( s_1 \ldots s_n \) are the names of these components, which serve as selectors. Thus, the definition of the mode \text{complex} might be written.

\[
\text{mode complex is [real \( \text{rp} \), real \( \text{ip} \]);}
\]

If \( Z \) is a complex value, one might write \( Z.\text{rp} \) or \( Z.\text{ip} \) to access either the real part or the imaginary part of \( Z \).

b. Sequences
The mode of a sequence is constructed by the notation \text{seq}(e)M, where \( e \) stands for an expression producing an integer value, which fixes the length of the sequence; the parenthesized expression may be omitted, in which case the length is variable. The components, each having mode \( M \), are indexed by integer values ranging from 1 to the current length, inclusively. The mode for real square matrices could be defined as follows:

\[
\text{mode rsqmatrix (int n) is seq (n) seq (n) real;}
\]

If \( B \) is a real square matrix, then the notation \( B(i)(j) \) would provide access to an individual component.

4. Union
The union constructor introduces a mode characterizing values belonging to one of a specified list of classes. The notation \( \text{union} (M_1, \ldots, M_n) \) produces a mode for values having any one of the modes \( M_1 \ldots M_n \). Thus, if one defines

\[
\text{mode procir is proc (union (int, real));}
\]

this mode describes procedures taking one parameter, which may be either an integer or a real, and returning no result. A further example, using the tuple, pointer, sequence and union constructors, shows the possibility of recursive definition:

\[
\text{mode tree is [char root, seq ptr union (char, tree) branch];}
\]

The list of mode constructors given above is intended to be indicative but not exhaustive. Moreover, it must be emphasized that the constructors themselves are essentially independent of the nature and number of the basic modes. Consequently, one could readily admit the use of such constructors with an entirely different set of primitive modes (e.g., one which more closely reflects the representations on an actual machine). What is essential is that the new modes generated by these constructors must be usable in the language in the same ways as the original ones.

The property clause

The property clause "with \( \pi \)" when present, specifies a list of properties possessed by the values of the mode being defined. These properties identify a sub-class of the values characterized by the mode in the type clause. Two kinds of properties are introduced for the present model: transforms and selectors.

1. Transforms
The transforms provide a means of specifying the actions to be taken when a value of mode \( M_1 \) occurs in a context where a value of mode \( M_2 \) is required \( (\left\langle M_1 \right\rangle \neq \left\langle M_2 \right\rangle) \). If \( M \) is the mode being declared, then two notations may be used to indicate a transform:

a. "from \( M_1 \) by \( V.E_1 \)," which specifies that a value of mode \( M \) may be produced from a value of mode \( M_1 \) (identified by the bound variable \( V \)) by evaluating the expression \( E_1 \).
b. "into $M_2$ by $V.E_2$" which specifies that a value of mode $M_2$ may be produced from a value of mode $M$ (identified by the bound variable $V$) by evaluating the expression $E_2$. The following definitions provide an illustration:

```plaintext
mode complex
is [real rp, real ip]
with from real
by x. [x, 0.0],
into real
by y. (if y.ip = 0
  then y.rp
  else error);

mode imaginary
is [real ip]
with from complex
by x. (if x.rp = 0
  then x.ip
  else error),
into complex
by y. [0.0, y.ip];
```

By the transforms in the above definitions, all of the natural conversions among real, complex, and imaginary values are provided. It must be noted that the system of transformations specified among the modes may be represented by a directed graph, where the nodes correspond to the modes, and the arcs are established by the from and into transforms. Thus, the rule to decide whether the transformation from $M_1$ into $M_2$ is known might be formulated as follows:

i. There must exist at least one path from $M_1$ to $M_2$.
ii. If there are several paths, there must exist one which is shorter than all of the others.
iii. That path represents the desired transformation.

2. Selectors
The notation "take $M_1$s as $V.E$" may appear in the list of properties attached to the definition of the mode $M$. It serves to establish the name "$s$" as an additional selector which may be applied to values of mode $M$ to produce a value of mode $M_1$. Thus, if $X$ is a value of mode $M$, then the effect of writing "$X.s$" is to evaluate the expression $E$ (within which $V$ acts as a bound variable identifying the value $X$) and to transform its result into a value of mode $M_1$.

As an example, the definition of complex might be augmented by attaching the following two properties:

- `take real mag as Z. (sqrt (Z.rp^2 + Z.ip^2)),`
- `take radian ang as Z. (arctan (Z.ip/Z.rp));`

The mode `radian` is presumed to be defined elsewhere, and to properly characterize the class of angular values.

As with the case of the constructors, the properties presented here are intended to suggest the kinds of facilities which are appropriate within the framework established by the concept of mode.

In summary, it must be stressed that the model developed here is applicable only to one particular category of objects, namely the values on which a program operates. Clearly, there exist other identifiable categories which enter into the processing of a program (e.g., control structure, environment resources, etc.). It is equally appropriate to regard these as potential domains for semantic extensibility. This will no doubt necessitate the introduction of additional extension mechanisms, following the general approach established here. As the number of mechanisms is expanded, the possibility for selective restriction of the extension capabilities will become increasingly important. The development of the corresponding semantic restriction mechanisms is imperative, for they are essential to the production of specialized compilers for languages defined by means of extension.

**SYNTACTIC EXTENSIBILITY**

A language incorporating functions which permit a user to introduce explicit modifications to the syntax of that language is said to possess the property of syntactic extensibility. The purpose of this section is to establish the nature of such a facility. It is primarily devoted to the development of a model which will serve to characterize the mechanism of syntactic extension, and permit exploration of its definitional range.

It must be made explicit that, when speaking of modifications to the syntax of a language, one is in fact talking about actual alterations to the grammar which serves to define that syntax. For a conventional language, the grammar is essentially static. Thus, it is conceivable that a programmer could be wholly unaware of its existence. The syntactic rules, which he is nonetheless constrained to observe (whether he likes them or not), are the same each time he writes a pro-
gram in that language, and no deviation is permitted anywhere in the scope of the program. The situation is significantly different for the case of a syntactically extensible language. This capability is provided by means of a set of functions, properly imbedded in the language, which acts to change the grammar. Provided that the user is cognizant of these functions and their grammatical domain, he then has the option of effecting perhaps quite substantial modifications to the syntax of that language during the course of writing a program in that language; this is in parallel with whatever semantic extensions he might introduce. In effect, the grammar itself becomes subject to dynamic variation, and the actual syntax of the language becomes dependent on the program being processed.

The syntactic macro mechanism: A model

The basis of most existing proposals for achieving syntactic extensibility is what has come to be called the syntactic macro mechanism. A model of this mechanism is introduced at this point in order to illustrate the possibilities of syntactic extension. The model is based on the assumption that the syntactic component of the base language, and by induction any subsequent extended language, can be effectively defined by a context-free grammar (or the equivalent BNF representation). This relatively simple formalism is adopted as the underlying definitional system despite an obvious contradiction which is present: a grammar which is subject to dynamic redefinition by constructs in the language whose syntax it defines is certainly not “context-free” in the strict sense. Therefore, it is only the instantaneous syntactic definition of the language which is considered within the context-free framework.

The most essential element of the syntactic macro mechanism is the function which establishes the definition of a syntactic macro. It must be a legitimate linguistic construct of the base language proper, and its format would likely resemble any other declaration in that language. The following representation will be used to model a call on this function:

\[ \text{macro } \phi \text{ where } \pi \text{ means } \rho; \]

The respective components are:

- \( \phi \), the production;
- \( \pi \), the predicate; and
- \( \rho \), the replacement.

The macro clause would be written in the form

\[ \text{macro } C \rightarrow 'phrase' \]

where \( C \) is the name of a category (non-terminal) symbol in the grammar, and the phrase is an ordered sequence, \( S_1 \ldots S_n \), such that each constituent is the name of a category or terminal symbol. Thus the production in a macro clause corresponds directly to a context-free production. The \textit{where} and \textit{means} clauses are optional components of the definition, and will be discussed below.

A syntactic macro definition differs from an ordinary declaration in the base language in the sense that it is a function operating directly on the grammar, and takes effect immediately. In essence, it incorporates the specified production into the grammar. Subsequent to the occurrence of such a definition in a program, syntactic configurations conforming to the structure of the phrase are acceptable wherever the corresponding category is syntactically valid. This will apply until such time as that definition is, in some way, disabled. As an example, one might include a syntactic macro definition starting with

\[ \text{macro FACT} \rightarrow 'PRIM!' \]

for the purpose of introducing the factorial notation into the syntax for arithmetic expressions. Within the scope of that definition, the effect would be the same as if the syntactic definition of the language (represented in BNF) incorporated an additional alternative

\[ (\text{factor}):=\ldots\mid (\text{primary})! \]

Thus, in principle, a statement of the form

\[ c := n!/(n-m)! * m!; \]

might become syntactically valid according to the active set of definitions.

The production

The role of the production is to establish both the context and the format in which “calls” on that macro may be written. The category name on the left controls where, within the syntactic framework, such calls are permitted. One may potentially designate any category which is referenced by the active set of productions. The phrase indicates the exact syntactic configuration which is to be interpreted as a call on that particular macro. In general, one may specify any arbitrary sequence (possibly empty) of symbol names. The constituents may be existing symbols, terminals which were not previously present, or categories to be defined in other macros. This is of course, subject to the constraint that the grammar as a whole must remain both
well-formed and non-ambiguous, if it is to fulfill its intended function.

In addition, the macro clause serves to declare a set of formal parameters, which may be referenced elsewhere in the definition. Each separate call on that macro can be thought of as establishing a local syntactic context, defined with respect to the complete syntax tree which structurally describes the program. This context would be relative to the position of the node corresponding to the specified category, and would include the immediate descendants of that node, corresponding to the constituents of the phrase. At a call, the symbol names appearing in the production are associated with the actual nodes occurring in that context. Thus, the terminal names represent an individual instance of that terminal, and the category names represent some complete syntactic sub-tree belonging to that category. In order to distinguish between different formal parameters having the same name, the convention of subscripting the names will be adopted here; this notation could readily be replaced by declaration of unique identifiers.

The replacement

The means clause specifies the syntactic structure which constitutes the replacement for a call on that particular macro. It is written in the form

\[
\text{means} \ '\text{string}'
\]

where the string is an ordered sequence, composed of either formal parameters or terminal symbol names. An instance of this string is generated in place of every call on that macro, within which the actual structure represented by a formal parameter is substituted for every occurrence of its name. If the complete syntactic macro definition for the factorial operator had been

\[
\text{macro} \ \text{FACT} \rightarrow \text{PRIM}_1 \ !
\]

\[
\text{means} \ '\text{factorial (PRIM}_1')';
\]

then each call on this macro would simply be replaced by a call on the procedure named "factorial", assumed to be defined elsewhere.

When present, the means clause establishes the semantic interpretation to be associated with the corresponding production; if absent, then presumably the construct is only an intermediate form, whose interpretation is subsumed in some larger context. The "meaning," however, is given as the expansion of that construct into a "logically lower level language". While the replacement may be expressed in terms of calls on other syntactic macros, these will also be expanded. In effect, the meaning of every new construct introduced into the language is defined by specifying its systematic expansion into the base language. Accordingly, one might consider syntactic extension merely as a means for permitting a set of "systematic abbreviations" to be defined "on top of" the base language.

An important consequence of the fact that a syntactic macro definition is itself a valid element of the base language is that such definitions may occur in the context of a replacement. This is illustrated by the following example, showing how a declaration for "push-down stack" might be introduced:

\[
\text{macro} \ \text{DECL} \rightarrow \text{TYPE}_1 \ \text{stack} [\text{EXPR}_1] \ \text{IDEN}_1;'
\]

\[
\text{means} \ '\text{TYPE}_1 \ \text{array} [1: \text{EXPR}_1] \ \text{IDEN}_1;'
\]

\[
\text{integer} \ \text{level}_1 \ \text{initial} \ 0;
\]

\[
\text{macro} \ \text{PRIM} \rightarrow \text{depth} \ \text{IDEN}_1,'
\]

\[
\text{means} \ '\text{res} (\text{EXPR}_1);'
\]

\[
\text{macro} \ \text{PRIM}_1 \rightarrow \text{IDEN}_1,'
\]

\[
\text{means} \ '\text{if level}_1 \ \text{IDEN}_1 > 0 \text{ then} (\text{IDEN}_1, \text{level}_1 \ \text{IDEN}_1 := \text{level}_1 \ \text{IDEN}_1 - 1;)
\]

\[
\text{else}
\]

\[
\text{error ("overdraw IDEN_1");}
\]

\[
\text{macro} \ \text{REFR} \rightarrow \text{IDEN}_1,'
\]

\[
\text{means} \ '\text{if level}_1 \ \text{IDEN}_1 < \text{depth} \ \text{IDEN}_1 \text{ then}
\]

\[
(\text{level}_1 \ \text{IDEN}_1 := \text{level}_1 \ \text{IDEN}_1 + 1; \text{IDEN}_1 [\text{level}_1 \ \text{IDEN}_1])
\]

\[
\text{else}
\]

\[
\text{error ("overflow IDEN_1");}
\]

Thus a declaration of the form

\[
\text{integer} \ \text{stack} [K] 8;
\]

would generate not only the necessary array for holding the contents of the stack, but also several other declarations, including:

1. An integer variable, named level_8, corresponding to the level counter of the stack. It is initialized to zero on declaration.
2. A literal, written "depth_8," for representing the depth of that stack. Its expansion is given in terms of the operator res, which is taken to mean the result of a previously evaluated ex-
pression, and presumed to be defined accordingly.

3. A macro definition (PRIM) which establishes, by means of a compound expression, the interpretation of the stack name in "fetch-context". This allows one to write "N := S;" for removing the value from the top of the stack S and assigning it to the integer variable N.

4. A macro definition (REFR) which establishes the corresponding "store context" operation. One can then write "S := 5;" to push a new value into the stack.

The predicate

The where clause provides a way of specifying additional conditions which must be satisfied in order that the configuration defined by the phrase constitute a call on that particular macro. Its absence implies that the syntactic structure of the phrase is sufficient to identify a call. When present, it serves to introduce additional selectivity into the definition, which enhances the effect of conditional expansion. It is also a vehicle for enlarging the local syntactic context established at each call on the macro, thereby expanding the set of formal parameters declared within the definition.

As construed here, a predicate would be written as a sequence of calls on specialized logical functions, separated by the usual operators of predicate calculus (conjunction, disjunction, implication, etc.) and grouped by parentheses. The list which follows is indicative of the kind of functions which might be appropriate:

1. \( S_i = S_j \)
   which decides whether the syntactic configurations associated with the two previously declared formal parameters, \( S_i \) and \( S_j \), are structurally equivalent. \( S_j \) may instead be the symbol \( \epsilon \), which is used to decide whether \( S_i \) represents a construct corresponding to the empty phrase.

2. \( S_i \neq S_j \)
   which is the opposite of function (1). The following definition

   \[
   \text{macro BLOC} \rightarrow \text{LABL}_{1}: \begin{align*}
   \text{begin STATLIST}_1 \\
   \text{end LABL}_{1} \\
   \end{align*}
   \]

   where \( \text{LABL}_2 \neq \epsilon \supset \text{LABL}_2 = \text{LABL}_1 \)

   means . . .

   illustrates the use of these functions in a predicate.

3. \( S_i \rightarrow \text{phrase} \)
   where \( S_i \) is a previously declared parameter representing a category, and the phrase is written analogously to that of the production in a macro clause. It verifies whether the immediate substructure of the specified parameter corresponds to the indicated configuration. The constituents of the phrase are also declared as formal parameters. An interesting example is suggested by a peculiarity in PL/1, wherein the relation "7<6<5" is found to be true. A possible remedy might be formulated as follows:

   \[
   \text{macro REL} \rightarrow \text{REL}_{1} < \text{EXPR}_{1} \\
   \text{where REL} \rightarrow \text{EXPR}_{1} < \text{EXPR}_{2} \\
   \text{means} \text{REL}_{1} \land \text{rest} (\text{EXPR}_{2}) < \text{EXPR}_{1};
   \]

   The production in the where clause is assumed to be included in the base language, and "REL A REL" is taken to be syntactically defined elsewhere.

4. \( S_i \rightarrow \text{phrase} \)
   which is analogous to function (3), except that it verifies the substructure at an arbitrary depth, even to the terminal string. An example of its use might be:

   \[
   \text{macro ASGN} \rightarrow \text{REFR}_{1} := \text{EXPR}_{1} \\
   \text{where REFR} \rightarrow \text{EXPR}_{1} < \text{EXPR}_{2} \\
   \text{means} \text{REFR}_{1} < \text{EXPR}_{2} \}
   \]

   These functions are readily generalizable into an extremely powerful pattern-matching mechanism.

5. \( S_i \rightarrow S_j \)
   which determines, in the local syntactic context of the previously declared parameter, \( S_j \), whether the immediate antecedent of \( S_j \) corresponds to the category specified by \( S_i \). Also, \( S_i \) is declared as a formal parameter representing the "father" of \( S_j \).

6. \( S_i \rightarrow S_j \)
   which is simply a generalization of function (5), establishing \( S_i \) as the (nearest) direct antecedent of \( S_j \), regardless of the distance, which belongs to the category named by \( S_i \). For example, to access the name (IDEN) of the procedure in which a particular statement (STAT1) is embedded, one might write a where clause of the
The ellipsis notation is introduced with the framework of functions (3) and (4) to indicate that the structure of the corresponding constituents is irrelevant [and indeed, it may not even be knowable in the contexts that can be established by functions (5) and (6)].

7. \( \exists S_i \leftarrow \text{string} \)
   which is successful on the condition that the string (generated analogously to the replacement string) is directly reducible to the category specified by \( S_i \), which is also declared as a formal parameter to represent the completed sub-tree reflecting the analysis.

8. \( \exists S_i \leftarrow \text{string} \)
   which is analogous to function (7), but the condition is generalized to verify whether the string is reducible (regardless of the depth of the structure) to the specified category. The definition of the "maximum" function, which requires two syntactic macros, provides an interesting example:
   
   \[
   \begin{align*}
   \text{macro} \quad & \text{PRIM}_0 \rightarrow \max (\text{EXPRLIST}_1) \\
   & \text{where } \text{EXPRLIST}_1 \rightarrow \text{EXPR}_1 \\
   & \text{means } \langle \text{EXPR}_1 \rangle; \\
   \text{macro} \quad & \text{PRIM}_1 \rightarrow \max (\text{EXPRLIST}_1) \\
   & \text{where } \text{EXPRLIST}_1 \rightarrow \text{EXPRLIST}_2, \text{EXPR}_2 \\
   & \land \exists \text{PRIM}_2 \leftarrow \max (\text{EXPRLIST}_2) \\
   & \text{means } \langle \text{if } \text{PRIM}_2 > \text{EXPR}_2 \text{ then res PRIM}_2 \text{ else res } \langle \text{EXPR}_2 \rangle \rangle.
   \end{align*}
   \]

9. \( P \) (arguments)
   where \( P \) is the name of a semantic predicate, and the arguments may be either formal parameters or terminal symbols. Such conditions constitute a means of imposing non-syntactic constraints on the definition of a syntactic macro. They are especially applicable in those situations where it is necessary to establish the mode of a particular entity. For example, one might rewrite the factorial definition as follows:
   
   \[
   \begin{align*}
   \text{macro} \quad & \text{FACT}_0 \rightarrow \text{PRIM}_1! \\
   & \text{where } \text{is} _\text{integer} (\text{PRIM}_1) \\
   & \text{means } \langle \text{factorial} (\text{PRIM}_1) \rangle.
   \end{align*}
   \]
   In this form the definition also has the effect of allowing different meanings to be associated with the factorial operator, dependent on the mode of the primary.

10. \( \exists S : F \) (arguments)
   Where \( F \) is the name of a semantic function which conditionally returns a syntactic result. \( S_i \) is also declared as a formal parameter to represent this result. The semantic functions and predicates establish an interface whereby it is possible to introduce syntactic and semantic interdependencies. A likely application of semantic functions would be definitions involving identifiers:
   
   \[
   \begin{align*}
   \text{where } \exists \text{LABL}_1 : \text{newlabel } (\text{IDEN}_1) \ldots
   \end{align*}
   \]
   A particularly intriguing possibility is to provide a semantic function which evaluates an arbitrary expression:
   
   \[
   \begin{align*}
   \text{where } \exists \text{CONST}_1 : \text{evaluate } (\text{EXPR}_1) \ldots
   \end{align*}
   \]
   Obviously, this concept could be expanded to encompass the execution of entire programs, if desired.

It is evident that the role of the \textit{where} clause in a syntactic macro definition is to provide a framework for specifying those properties which effectively cannot be expressed within the context-free constraints. The fashion in which they are isolated allows these aspects to be incorporated without sacrificing all of the practical advantages which come from adopting a relatively simple syntactic formalism as the point of departure. With respect to the model presented here, however, it is nonetheless clear that the definitional power of the syntactic macro mechanism is determined by the power of the predicates.

Operationally, the syntactic macro mechanism can be characterized by three distinct phases, each of which is briefly considered below.

\textit{Definition phase}

The definition phase encompasses the different functions incorporated within the base language which act to insert, delete or modify a syntactic macro definition. Together, they constitute a facility for explicitly editing the given grammar, and are employed to form what might be called the \textit{active syntactic definition}. This consists of the productions of the currently active syntac-
tic macros (with their associated predicates and replacements), plus the original productions of the base language. An interesting generalization would be to provide a means of selectively eliminating base language productions from the active syntactic definition, thereby excluding those constructions from the source language; they would still remain part of the base language definition, however, and continue to be considered valid in the context of a replacement. In this fashion, the syntax of an extended language could be essentially independent of the base language syntax, thus further enhancing the definitional power of the syntactic macro mechanism.

**Interpretation phase**

The interpretation phase includes the processing of syntactic macro calls. It consists of three separate operations: (1), recognition of the production; (2), verification of the predicate; and (3), generation of the replacement. Obviously, these operations must proceed concurrently with the process of syntactic analysis, since syntactic macro expansion is incontestably a "compile-time facility". Given the present formulation of the syntactic macro mechanism, some form of what is called "syntax directed analysis" suggests itself initially as the appropriate approach for the analyzer. It must be observed that the character of the analysis procedure is constrained to a certain extent by the nature of the predicates contained within the active syntactic definition. Furthermore, the presence of semantic predicates and functions precludes realization of the analyzer/generator as a pure preprocessor.

In general, there will be the inevitable trade-off to be made between power of definition and efficiency of operation. It is pointless to pretend that this trade-off can be completely neglected in the process of formulating the syntactic definition of a particular extended language. However, deliberate emphasis has been given here to power of definition, with the intention of providing a very general language development framework, one which furnishes an operational compiler at every stage. It is argued that the problem of obtaining an efficient compiler properly belongs to a subsequent phase.

**Restriction phase**

The restriction phase, as construed here, would be a separate operation, corresponding to the automatic consolidation of some active syntactic definition in order to provide a specialized syntactic analyzer for that particular dialect. The degree to which this analyzer can be optimized is determined both by the syntactic complexity of the given extended language, and by the specific constraints on further syntactic extension which are imposed at that point. If subsequent extensions are to be permitted, they might be confined within extremely narrow limits in order to improve the performance of the analyzer; they may be excluded entirely by suppressing the syntactic definition functions in the base language. In either case, various well-defined sub-sets of context-free grammars, for which explicit identification and efficient analysis algorithms are known to exist, constitute a basis for establishing the restrictions. This represents the greatest practical advantage of having formulated the syntactic definition essentially within the context-free framework.

In conclusion, it is to be remarked that syntactic extensibility is especially amenable to realization by means of an extremely powerful extension mechanism in conjunction with a proportionally powerful restrictions mechanism. This approach provides the essential definitional flexibility, which is a prerequisite for an effective language development tool, without sacrificing the possibility of an efficient compiler. In the end, however, the properties of a particular extended language dictate the efficiency of its processor, rather than the converse. This is consistent with the broadened interpretation of extensible languages discussed in this paper.

**BIBLIOGRAPHY**


3 C CHRISTENSEN, C J SHAW Editors *Proceedings of the extensible languages symposium* Boston Massachusetts May 1969 SIGPLAN Notices Vol 4 Number 8 August 1969

4 B A GALLER, A J PERLIS *A proposal for definitions in ALGOL* Communications of the ACM Vol 10 Number 4 pp 204-209 April 1967

5 J V GARWICK *GPL, a truly general purpose language* Communications of the ACM Vol 11 Number 9 pp 634-639 September 1968

6 E T IRONS *Experience with an extensible language* Communications of the ACM Vol 13 Number 1 pp 31-40 January 1970
7 Ph JORRAND
Some aspects of BASEL, the base language for an extensible language facility
Proceedings of the Extensible Languages Symposium
SIGPLAN Notices Vol 4 Number 8 pp 14-17 August 1969

8 B M LEAVENWORTH
Syntax macros and extended translation
Communications of the ACM Vol 9 Number 11 pp 790-793 November 1966

9 M D McILROY
Macro instruction extensions to compiler languages
Communications of the ACM Vol 3 Number 4 pp 214-220 April 1960

10 A J PERLIS
The synthesis of algorithmic systems

11 T A STANDISH
Some features of PPL, a polymorphic programming language
Proceedings of the Extensible Languages Symposium
SIGPLAN Notices Vol 4 Number 8 pp 20-26 August 1969

12 T A STANDISH
Some compiler-compiler techniques for use in extensible languages
Proceedings of the Extensible Languages Symposium
SIGPLAN Notices Vol 4 Number 8 pp 55-62 August 1969

13 A VAN WINGAARDEN B J MAILLOUX
J E L PECK C H A KOSTER
Report on the algorithmic language ALGOL 68
MR 101 Mathematisch Centrum Amsterdam October 1969

14 B WEGBREIT
A data type definition facility
Harvard University Division of Engineering and Applied Physics unpublished 1969