Standards

S. GORN, Editor; E. LOHSE, Asst. Editor, Information Interchange

EULER: A Generalization of ALGOL, and its Formal Definition: Part I*

Niklaus Wirth and Helmut Weber†
Stanford University, Stanford, California

A method for defining programming languages is developed which introduces a rigorous relationship between structure and meaning. The structure of a language is defined by a phrase structure syntax, the meaning in terms of the effects which the execution of a sequence of interpretation rules exerts upon a fixed set of variables, called the Environment. There exists a one-to-one correspondence between syntactic rules and interpretation rules, and the sequence of executed interpretation rules is determined by the sequence of corresponding syntactic reductions which constitute a parse. The individual interpretation rules are explained in terms of an elementary and obvious algorithmic notation. A constructive method for evaluating a text is provided, and for certain decidable classes of languages their unambiguity is proved. As an example, a generalization of ALGOL is described in full detail to demonstrate that concepts like block-structure, procedures, parameters, etc. can be defined adequately and precisely by this method.

CONTENTS

(Part I)

(Part II)

I. Introduction and Summary

When devising a new programming language, one inevitably becomes confronted with the question of how to define it. The necessity of a formal definition is twofold: The users of this language need to know its precise meaning, and also need to be assured that the automatic processing systems, i.e., the implementations of the language on computers, reflect this same meaning equally precisely. ALGOL 60 represented the first serious effort to give a formal definition of a programming language [1]. The structure of the language was defined in a formal and concise way (which, however, was not in all cases unambiguous), such that for every string of symbols it can be determined whether it belongs to the language ALGOL 60 or not. The meaning of the sentences, i.e., their effect on the computational process, was defined in terms of ordinary English with its unavoidable lack of precision. But probably the greater deficiency than certain known imprecise definitions was the incompleteness of the specifications. By this no reference is made to certain intentional omissions (like specification of **real** arithmetic), but to situations and constructs which simply were not anticipated and therefore not explained (e.g., dynamic **own** arrays or conflicts of names upon procedure calls). A method for defining a language should therefore be found which guarantees that no unintentional omissions may occur.

How should meaning be defined? It can only be explained in terms of another language which is already well understood. The method of formally deriving the meaning of one language from another makes sense, if and only if the latter is simpler in structure than the former. By a sequence of such derivations a language will ultimately be reached where it would not be sensible to define it in terms of anything else. Recent efforts have been conducted with this principle in mind.

Böhm [3] and Landin [4, 5] have chosen the λ-calculus as the fundamental notation [6, 7], whose basic element is the function, i.e., a well-established concept. The motivation for representing a program in functional form is to avoid a commitment to a detailed sequence of basic steps representing the algorithm, and instead to define the meaning or effect of a program by the equivalence class of algorithms represented by the indicated function. Whether it is worthwhile to achieve such an abstract definition of meaning in the case of programming languages is not discussed here. The fact that a program consists basically of

* Part II of this paper, which gives the formal definition of "EULER: An Extension and Generalization of ALGOL," will appear in the February, 1966 issue of the *Communications of the ACM*.

This work was partially supported by the National Science Foundation (GP 4053) and the Computation Center of Stanford University.

† Present address: IBM Corporation, Endicott, N. Y.

single steps remains, and it cannot even be hidden by a transliteration into a functional notation: The sequence is represented by the evaluations of nests of functions and their parameters. An unpleasant side-effect of this translation of ordinary programming languages into λ-calculus is that simple computer concepts such as assignment and jumps transform into quite complicated constructs, this being in obvious conflict with the stated requirement that the fundamental notation should be simple.

Van Wijngaarden describes in [8, 9] a more dynamic approach to the problem: The fundamental notation is governed by only half a dozen rules which are obvious. It is in fact so simple that it is far from being a useful programming notation whatsoever, but just capable enough to provide for the mechanism of accepting additional rules and thus expanding into any desirable programming system. This method of defining the meaning (or, since the meaning is imperative: effect) of a language is clearly distinct from the method using functional notations, in that it explicitly makes use of algorithmic action, and thus guarantees that an evaluating algorithm exists for any sentence of the language. The essence of this algorithm consists of first scanning the ordered set of rules defining the structure of the language, and determining the applicable structural designations, i.e., performing an *applicability scan*, and then scanning the set of rules for evaluating the determined structural units, i.e., performing an *evaluation scan*. The rules are such that they may invoke application of other rules or even themselves. The entire mechanism is highly recursive and the question remains, whether a basically subtle and intricate concept such as recursion should be used to explain other programming languages, including possibly very simple ones.

The methods described so far have in common that their basic set of fundamental semantic entities does not resemble the elementary operations performed by any computational device presently known. Since the chief aim of programming languages is their use as communication media with computers, it would seem only natural to use a basic set of semantic definitions closely reflecting the computer's elementary operators. The invaluable advantage of such an approach is that the language definition is itself a processing system and that implementations of the language on actual machines are merely adaptations to particular environmental conditions of the language definition itself. The question of correctness of an implementation will no longer be undecidable or controversial, but can be directly based on the correctness of the individual substitutions of the elementary semantic units by the elementary machine operations.

It has elsewhere been proposed (e.g. [10]) to let the processing systems themselves be the definition of the language. Considering the complexity of known compiler systems this seems to be an unreasonable suggestion, but if it is understood as a call for systematizing such processing systems and representing them in a notation independent from any particular computer, then the suggestion appears in a different light.

The present paper reports on efforts undertaken in this direction. It seems obvious that the definition of the structure, i.e., the syntax, and the definition of the meaning should be interconnected, since structural orderings are merely an aid to understanding a sentence. In the presented proposal the *analysis* of a sentence proceeds in parallel with its *evaluation*: Whenever a *structural unit* is discovered, a corresponding *interpretation rule* is found and obeyed. The syntactic aspects are defined by a Phrase Structure System (cf. [2, 11, 12]) which is augmented by the set of interpretation rules defining the semantic aspects. Such an augmented Phrase Structure Language is subsequently called a *Phrase Structure Programming Language*, implying that its meaning is strictly imperative and can thus be expressed in terms of a *basic algorithmic notation* whose constituents are, e.g., the fundamental operations of a computer.

Although here the processes of syntactic analysis and semantic evaluation are more clearly separated, the analogies to the van Wijngaarden proposal are apparent. The parsing corresponds to the applicability scan, the execution of an interpretation rule to the evaluation scan. However, this proposal advocates the strict separation between the rules which define the language, i.e., its analysis and evaluation mechanisms, and the rules produced by the particular program under evaluation, while the van Wijngaarden proposal does not distinguish between language definition and program. Whether the elimination of this distinction which enables—and forces—the programmer to supply his own language defining rules, is desirable or not must be left unanswered here. The original aim of this contribution being the development of a proposal for a fixed language, it would have been meaningless to eliminate it.

Section II contains the descriptions of an algorithmic notation considered intuitively obvious enough not to necessitate further explanation in terms of more primitive concepts. This notation will subsequently be used for the definition of algorithms and interpretation rules, thus playing a similar role for the semantic aspects as did BNF for the syntactic aspects of Algol 60. The function of this notation is twofold: (1) it serves to precisely describe the analysis and evaluation mechanisms, and (2) it serves to define the basic constituents of the higher level language. E.g., this basic notation contains the elementary operators for arithmetic, and therefore the specifications of the higher level language defer their definition to the basic algorithmic notation. It is in fact assumed that the definition of **integer** arithmetic is below the level of what a programming language designer is concerned with, while **real** arithmetic shall very intentionally not be defined at all in a language standard. The concepts which are missing in the basic notation and thus will have to be defined by the evaluation mechanisms are manifold: The sequencing of operations and operands in expressions, the storage allocation, the block structure, procedure structure, recursivity, value- and name-parameters, and so forth.

Section III starts out with a list of basic formal definitions leading to the terms "Phrase Structure System," "Phrase Structure Programming Language" and "Meaning." The notation and terminology of [12] is adopted here as far as possible. The fact that the nature of meaning of a programming language is imperative, allows the meaning of a sentence to be explained in terms of the changes which are affected on a certain set of variables by obeying the sentence. This set of variables is called the *Environment* of the Programming Language. The definition of the meaning with the aid of the structure and the definition of the evaluation algorithm in terms of structural analysis of a sentence demand that emphasis be put on the development of a constructive algorithm for a syntactic analysis. Section III is mainly devoted to this topic. It could have been entirely avoided, had a reductive instead of a productive definition of the syntax been chosen. By a *productive* syntactic definition is meant a set of rules illustrating the various constructs which can be generated by a given syntactic entity. By a *reductive* syntactic definition is meant a set of rules directly illustrating the reductions which apply to a given sentence. A reductive syntax therefore directly describes the analyzer, and recently some compilers have been constructed directly relying on a reductive syntactic description of the language [13]. A language definition, however, is not primarily directed toward the reader (human or artificial) but toward the writer or creative user. His aim is to construct sentences to express certain concepts or ideas. The productive definition allows him to derive directly structural entities which conform to his concepts. In short, his use of the language is primarily synthetic and not analytic in nature. The reader then must apply an analytic process, which in turn one should be able to specify given the productive syntactic definitions. One might call this a transformation of a productive into a reductive form, a synthetic into an analytic form.

The transformation method derived subsequently is largely based on earlier work by R. W. Floyd described in [14]. The grammars to which this transformation applies are called *Precedence Grammars*. The term "Precedence Syntax" is, however, redefined, because the class of precedence grammars described in [14] was considered to be too restrictive, and even unnecessarily so. In particular, there is no need to define the class of precedence grammars as a subclass of the "Operator grammars." Several classes of precedence grammars are defined here, the order of a precedence grammar being determined by the amount of context the analysis has to recognize and memorize in order to make decisions. This classification relates to the definition of "Structural Connectedness" described in [15], and provides a means to effectively determine the amount of connectedness for a given grammar.

Also in Section III, an algorithm is described which decides whether a given grammar is a precedence grammar, and if so, performs the desired transformation into data representing the reductive form of the grammar.

A proof is then provided of the *unambiguity* of precedence grammars, in the sense that the sequence of syntactic reductions applied to a sentence is unique for every sentence in the language. Because the sequence of interpretation rules to be obeyed is determined by the sequence of syntactic reductions, this uniqueness also guarantees the unambiguity of meaning, a crucial property for a programming language. Furthermore, the fact that all possible reductions are described exhaustively by the syntax, and that to every syntactic rule there exists a corresponding interpretation (semantic) rule, guarantees that the definition of meaning is exhaustive. In other words, every sentence has one and only one meaning, which is well defined, if the sentence belongs to the language. Section III ends with a short example: The formal definition of a simple programming language containing expressions, assignment statements, declarations and block-structure.

A formal definition of an extension and generalization of Algol 60 is presented in Section IV. It will demonstrate that the described methods are powerful enough to define adequately and concisely all features of a programming language of the scope of Algol 60. This generalization is a further development of earlier work presented in [16].

II. An Elementary Notation for Algorithms

This notation is used in subsequent chapters as a basis for the definitions of meaning of more complicated programming languages.

A program is a sequence of imperative statements. In the following paragraphs the forms of a statement written in this elementary notation are defined and rules are given which explain its meaning. There exist two different kinds of statements: the *Assignment Statement* and the *Branching Statement*.

The Assignment statement serves to assign a new value to a variable whose old value is thereby lost. The successor of an Assignment Statement is the next statement in the sequence. The Branching Statement serves to designate a successor explicitly. Statements may for this purpose be labeled.

A. The Assignment Statement. The (direct) Assignment Statement is of the form:

$$v \leftarrow E$$

v stands for a *variable* and E for an *expression*. The meaning of this statement is that the current value of v is to be replaced by the current value of E.

An expression is a construct of either one of the following forms:

$$x, \quad \circ x, \quad x \, \Theta \, y, \quad r$$

where x, y, stand for either *variables*, *literals* or *lists*, $\circ$

stands for a *unary operator*, Θ stands for a *binary operator* and r stands for a *reference*. The value of an expression involving an operator is obtained by applying the operator to the current value(s) of the operand(s). A reference is written as $@v$, where v is the referenced variable.

The *indirect* Assignment Statement is written as

$$v \cdot \leftarrow E$$

and is meant to assign the current value of the expression E to the variable, whose reference is currently assigned to the variable v.

1. *Literals.* A literal is an entity characterized by the property that its value is always the literal itself. There may exist several kinds of literals, e.g., numbers, logical constants (Boolean), and symbols. Furthermore there exists the literal Ω with the meaning "undefined." Numeric constants are denoted in standard decimal form. The logical constants are **true** and **false**.[1]

A symbol or character is denoted by the symbol itself enclosed in quote marks ("). A list of symbols is usually called a string. Other types of literals may arbitrarily be introduced.

2. *Lists.* A list is an entity denoted by

$$(E, F, \cdots, G)$$

whose value is the ordered set of the current values of the expressions E, F, $\cdots$, G, called the elements of the list. A list can have any number of elements (including 0), and the elements are numbered with the natural numbers starting with 1.

3. *Variables.* A variable is an entity uniquely identified within a program by a name to which a value can be assigned (and reassigned) during the execution of a program. Before the first assignment to a variable, its value shall be Ω.

If the value of a variable consists of a sequence of elements, any one element may be designated by the variable name and a subscript, and thus is called a subscripted variable. The subscript is an expression, whose current value is the ordinal number of the element to be designated. Thus, after $a \leftarrow (1, 2, (3, 4, 5), 6)$, $a[1]$ designates the element "1", $a[3]$ designates the element (3, 4, 5), and therefore $a[3][2]$ designates the second element of $a[3]$, i.e., "4". The notation a_i shall be understood equivalent to $a[i]$, $a_{i,j}$ equivalent to $a[i][j]$, etc.

4. *Unary Operators.* Examples of unary operators are:

$-x$ yields the negative of **x**.
c x yields the value of the variable whose reference is currently assigned to x (instead of **c**x we also use $x\cdot$).
abs x yields the absolute value of x.
integer x yields x rounded to the nearest integer.
tail x yields the list x with its first element deleted.
isli x yields **true**, if x **is** a **list**, **false** otherwise.

A further set of unary operators is the set of typetest operators which determine whether the current value of a variable is a member of a certain set of literals. The resulting value is **true**, if the test is affirmative, **false** otherwise. For example,

isn x, current value of x **is** a number
isb x, current value of x **is** a logical (Boolean) constant
isu x, current value of x **is** Ω (undefined)
isy x, current value of x **is** a symbol

A further set of unary operators is the set of conversion operators which produce values of a certain type from a value of another type. For example,

real x yields the number corresponding to the logical value x;
logical x inverse of **real** (**true** ↔ 1, **false** ↔ 0 shall be assumed); Conversion operators between numbers and symbols shall not be defined here, although their existence is assumed, because the notation does not define the set of symbols which may possibly be used.

5. *Binary Operators.* Examples of binary operators are:

$+ - \times$ designating addition, subtraction and multiplication in the usual sense. The accuracy of the result in the case of the operands being nonintegral numbers is not defined.
$/$ denoting division in the usual sense. The accuracy of the result is not defined here. In case of the denominator being 0, the result is Ω.
$\div$ denoting division between the rounded operands with the result being truncated to its integral value.
mod yields the remainder of the division denoted by $\div$.
& yields the concatenation of two lists, i.e., (x) & (y) = (x, y).
$=$ yields **true**, if the two scalar operands are equal, **false** otherwise.
$\uparrow$ denoting exponentiation, i.e., $x \uparrow y$ stands for x^y.

The classes of unary and binary operators listed here may be extended and new types of literals may be introduced along with corresponding typetest and conversion operators.

B. The Branching Statement. There are Simple and Conditional Branching Statements.

1. *The Simple Branching Statement.* It is of the form

go to l

where l stands for a label. The meaning is that the successor of this statement is the statement with the label l. Labeling of a statement is achieved by preceding it with the label and a colon (:) The label is a unique name (within a program) and designates exactly one statement of the program.

2. *The Conditional Branching Statement.* It is of the form

if E **then go to** l

where l is a label uniquely defined in the program and E is an expression. The meaning is to select as the successor to the Branching Statement the statement with the label l, if the current value of E is **true**, or the next statement in the sequence, if it is **false**. For notational convenience a

[1] The boldface letters have to be understood as one single symbol.

statement of the form

$$\text{if } \neg\, E \text{ then go to } l$$

where ¬ stands for not, shall be admitted and understood in the obvious sense.

Notational standards are not fixed here. Thus the sequence of statements can be established by separating statements by delimiters, or by beginning a new line for every statement. The Branching Statement and the labeling of statements may be replaced by explicit arrows, thus yielding block diagrams or flowcharts.

III. Phrase Structure Programming Languages

A. Notation, Terminology, Basic Definitions. Let $\mathcal{V}$ be a given set: the *vocabulary*. Elements of $\mathcal{V}$ are called *symbols* and denoted here by capital Latin letters, S, T, U, etc. Finite sequences of symbols—including the empty sequence (Λ)—are called *strings* and are denoted by small Latin letters, x, y, z, etc. The set of all strings over $\mathcal{V}$ is denoted by $\mathcal{V}^*$. Clearly $\mathcal{V} \subseteq \mathcal{V}^*$.

A *simple phrase structure system* is an ordered pair $(\mathcal{V}, \Phi)$, where $\mathcal{V}$ is a vocabulary and Φ is a finite set of syntactic rules φ_i of the form $U \rightarrow x$ where $x \neq U$, $U \in \mathcal{V}$, $x \in \mathcal{V}^*$.

For $\varphi = U \rightarrow x$, U is called the *left part* and x the *right part* of φ.

y directly produces z ($y \dashrightarrow z$) and conversely *z directly reduces* into y, if and only if there exist strings u, v such that $y = uUv$ and $z = uxv$, and the rule $U \rightarrow x$ is an element of Φ.

y produces z ($y \xrightarrow{*} z$) and conversely *z reduces* into y, if and only if there exist a sequence of strings $x_0, \cdots, x_n$, such that $y = x_0$, $x_n = z$ and

$$x_{i-1} \dashrightarrow x_i \ (i = 1, \cdots, n; n \geq 1).$$

A *simple phrase structure syntax* is an ordered quadruple $\mathcal{G} = (\mathcal{V}, \Phi, \mathcal{B}, A)$, where $\mathcal{V}$ and Φ form a phrase structure system; $\mathcal{B}$ is the subset of $\mathcal{V}$ such that none of the elements of $\mathcal{B}$ (called basic symbols) occurs as the left part of any rule of Φ, while all elements of $\mathcal{V}-\mathcal{B}$ occur as left part of at least one rule; A is the symbol which occurs in no right part of any rule of Φ.

The letter U always denotes some symbol $U \in \mathcal{V} - \mathcal{B}$.

x is a *sentence* of $\mathcal{G}$, if $x \in \mathcal{V}^*$ (i.e., x is a string of basic symbols) and $A \xrightarrow{*} x$.

A *simple phrase structure language* $\mathcal{L}$ is the set of all strings x which can be produced by $(\mathcal{V}, \Phi)$ from A: $\mathcal{L}(\mathcal{G}) = \{x \mid A \xrightarrow{*} x \wedge x \in \mathcal{V}^*\}$. Let $U \xrightarrow{*} z$. A *parse* of the string z into the symbol U is a sequence of syntactic rules $\varphi_1, \varphi_2, \cdots, \varphi_n$, such that φ_j directly reduces z_{j-1} into z_j ($j = 1, \cdots, n$), and $z = z_0$, $z_n = U$.

Assume $z_k = U_1U_2 \cdots U_m$ (for some $1 < k < n$). Then z_i ($i < k$) must be of the form $z_i = u_1u_2 \cdots u_m$, where for each $l = 1 \cdots m$ either $U_l \xrightarrow{*} u_l$, or $U_l = u_l$. Then the *canonical form* of the section of the parse reducing z_i into z_k shall be $\{\varphi_1\}\{\varphi_2\} \cdots \{\varphi_m\}$, where the sequence $\{\varphi_l\}$ is the canonical form of the section of the parse reducing u_l into U_l. Clearly $\{\varphi_l\}$ is empty, if $U_l = u_l$, and is canonical, if it consists of one element only.

The canonical parse is the parse which proceeds strictly from *left to right* in a sentence, and reduces a leftmost part of a sentence as far as possible before proceeding further to the right. In general, there may exist several canonical parses for a sentence, but every parse has only one canonical form.

An *unambiguous syntax* is a phrase structure syntax with the property that for every string $x \in \mathcal{L}(\mathcal{G})$ there exists exactly one canonical parse.

It has been shown that there exists no algorithm which decides the ambiguity problem for any arbitrary syntax. However, a sufficient condition for a syntax to be unambiguous is subsequently derived, and a method is explained to determine whether a given syntax satisfies this condition.

An *environment* $\mathcal{E}$ is a set of variables whose values define the meaning of a sentence.

An *interpretation rule* ψ defines an action (or a sequence of actions) involving the variables of an environment $\mathcal{E}$.

A *phrase structure programming language* $\mathcal{L}_p(\mathcal{G}, \Psi, \mathcal{E})$ is a phrase structure language $\mathcal{L}(\mathcal{G})$, where $\mathcal{G}(\mathcal{V}, \Phi, \mathcal{B}, A)$ is a phrase structure syntax, Ψ is a set of (possibly empty) interpretation rules such that a unique one to one mapping exists between elements of Ψ and Φ, and $\mathcal{E}$ is an environment for the elements of Ψ. Instead of $\mathcal{L}_p(\mathcal{G}, \Psi, \mathcal{E})$ we also write $\mathcal{L}_p(\mathcal{V}, \Phi, \mathcal{B}, A, \Psi, \mathcal{E})$.

The *meaning* m of a sentence $x \in \mathcal{L}_p$ is the effect of the execution of the sequence of interpretation rules $\psi_1, \psi_2, \cdots, \psi_n$ on the environment $\mathcal{E}$, where $\varphi_1, \varphi_2, \cdots, \varphi_n$ is a parse of the sentence x into the symbol A and ψ_i corresponds to φ_i for all i.

It follows immediately that a programming language will have an unambiguous meaning, if its underlying syntax is unambiguous. As a consequence, every sentence of the language has a well defined meaning. A *sentence* $x_1 \in \mathcal{L}_p(\mathcal{G}_1, \Psi_1, \mathcal{E})$ is called *equivalent* to a sentence $x_2 \in \mathcal{L}_p(\mathcal{G}_2, \Psi_2, \mathcal{E})$ (possibly $\mathcal{G}_1 = \mathcal{G}_2$, $\Psi_1 = \Psi_2$), if and only if $m(x_1)$ is equal to $m(x_2)$. A *programming language* $\mathcal{L}_p(\mathcal{G}_1, \Psi_1, \mathcal{E})$ is called *equivalent* to $\mathcal{L}_p(\mathcal{G}_2, \Psi_2, \mathcal{E})$, if and only if $\mathcal{L}_{p1} = \mathcal{L}_{p2}$ and for every sentence x, $m_1(x)$ according to $(\mathcal{G}_1, \Psi_1)$ is equal to $m_2(x)$ according to $(\mathcal{G}_2, \Psi_2)$.

B. Precedence Phrase Structure Systems. The definition of the meaning of a sentence requires that a sentence must be parsed in order to be evaluated or obeyed. Our prime attention is therefore directed toward a constructive method for parsing. In the present section, a parsing algorithm is described. It relies on certain relations between symbols. These relations can be determined for any given syntax. A syntax for which the relation between any two symbols is unique, is called a *simple precedence syntax*. Obviously, the parsing algorithm only applies to precedence phrase structure systems. It is then shown that any parse in such a system is unique. The class of precedence phrase structure systems is only a restricted subset among all phrase structure systems. The definition of precedence relations is subsequently generalized with the effect that the class of precedence phrase structure systems is considerably enlarged.

1. *Parsing Algorithm for Simple Precedence Phrase Structure Languages.* In accordance with the definition of the canonical form of a generation tree or of a parse, a parsing algorithm must first detect the leftmost substring of the sentence to which a reduction is applicable. Then the reduction is to be performed and the same principle is applied to the new sentence. In order to detect the leftmost reducible substring, the algorithm to be presented here makes use of previously established noncommutative relations between symbols of $\mathcal{V}$ which are chosen according to the following criteria:

(a) The relation $\doteq$ holds between all adjacent symbols within a string which is directly reducible.

(b) The relation $<$ holds between the symbol immediately preceding a reducible string and the leftmost symbol of that string.

(c) The relation $>$ holds between the rightmost symbol of a reducible string and the symbol immediately following that string.

The process of detecting the leftmost reducible substring now consists of scanning the sentence from left to right until the first symbol pair is found so that $S_i > S_{i+1}$, then to retreat back to the last symbol pair for which $S_{j-1} < S_j$ holds. $S_j \cdots S_i$ is then the sought substring; it is replaced by the symbol resulting from the reduction. The process then repeats itself. At this point it must be noted that it is not necessary to start scanning at the beginning of the sentence, since all symbols S_k for $k < j$ have not been altered, but that the search for the next $>$ can start at the place of the previous reduction.

In the following description of the algorithm in pseudo ALGOL the original sentence is denoted by $P_1 \cdots P_n$. k is the index of the last symbol scanned. For practical reasons, all scanned symbols are copied and renamed $S_1 \cdots S_i$. The reducible substring therefore will always be $S_j \cdots S_i$ for some j. Internal to the algorithm, there exists a symbol $\perp$ initializing and terminating the process. To any symbol S of $\mathcal{V}$ it has the relations $\perp < S$ and $S > \perp$.

We assume that $P_0 = P_{n+i} = \perp$.

```
comment  Algorithm for syntactic analysis of a sentence $P$ on
  the basis of precedence relations;
$S_0 := P_0$;  $i := 0$;  $k := 1$;
while $P_k \neq$ '$\perp$' do
begin $i := j := i+1$;  $S_i := P_k$;  $k := k+1$;
  while $S_i > P_k$ do
  begin while $S_{j-1} \doteq S_j$ do $j := j-1$;
    $S_j :=$ Leftpart $(S_j \cdots S_i)$;  $i := j$
  end
end
```

The function denoted by Leftpart $(S_j \cdots S_i)$ requires that the reducible substring is identified in order to obtain the symbol resulting from the reduction. If furthermore the parsed sentence is to be evaluated, then the interpretation rule ψ_l corresponding to the syntactic rule φ_l: $U \rightarrow S_j \cdots S_i$ is to be identified and executed.

2. *Algorithm to Determine the Precedence Relations.* The definition of the precedence relations can be formalized in the following way:

(a) For any ordered pair of symbols (S_i, S_j), $S_i \doteq S_j$, if and only if there exists a syntactic rule of the form $U \rightarrow xS_iS_jy$, for some symbol U and some (possibly empty) strings x, y.

(b) For any ordered pair of symbols (S_i, S_j), $S_i < S_j$, if and only if there exists a syntactic rule of the form $U \rightarrow xS_iU_ly$, for some U, x, y, U_l, and there exists a generation $U_l \xrightarrow{*} S_jz$, for some string z.

(c) For any ordered pair of symbols (S_i, S_j), $S_i > S_j$, if and only if (1) there exists a syntactic rule of the form $U \rightarrow xU_kS_jy$, for some U, x, y, U_k, and there exists a generation $U_k \xrightarrow{*} zS_i$ for some string z, or (2) there exists a syntactic rule of the form $U \rightarrow xU_kU_ly$, for some U, x, y, U_k, U_l, and there exist generations $U_k \xrightarrow{*} zS_i$ and $U_l \xrightarrow{*} S_jw$ for some strings z, w.

We now introduce the sets of leftmost and rightmost symbols of a nonbasic symbol U by the following definitions:

$$\mathcal{L}(U) = \{S \mid \exists z(U \xrightarrow{*} Sz)\}$$

$$\mathcal{R}(U) = \{S \mid \exists z(U \xrightarrow{*} zS)\}.$$

Now the definitions (a)–(c) can be reformulated as:

$$\begin{aligned}
&\text{(a)} \quad S_i \doteq S_j \leftrightarrow \exists\varphi(\varphi\colon U \rightarrow xS_iS_jy)\\
&\text{(b)} \quad S_i < S_j \leftrightarrow \exists\varphi(\varphi\colon U \rightarrow xS_iU_ly) \wedge S_j \in \mathcal{L}(U_l)\\
&\text{(c)} \quad S_i > S_j \leftrightarrow \exists\varphi(\varphi\colon U \rightarrow xU_kS_jy) \wedge S_i \in \mathcal{R}(U_k)\\
&\qquad\qquad \vee \exists\varphi(\varphi\colon U \rightarrow xU_kU_ly)\\
&\qquad\qquad \wedge S_i \in \mathcal{R}(U_k) \wedge S_j \in \mathcal{L}(U_l)
\end{aligned}$$

These definitions are equivalent to the definitions of the precedence relations, if Φ does not contain any rules of the form $U \rightarrow \Lambda$, where Λ denotes the empty string.

The definition of the sets $\mathcal{L}$ and $\mathcal{R}$ is such that an algorithm for effectively creating the sets is evident. A symbol S is a member of $\mathcal{L}(U)$, if (a) there exists a syntactic rule φ: $U \rightarrow Sx$, for some x, or (b) there exists a syntactic rule φ: $U \rightarrow U_1x$, and $S \in \mathcal{L}(U_1)$; i.e.,

$$\mathcal{L}(U)$$
$$= \{S \mid \exists\varphi\colon U \rightarrow Sx \vee \exists\varphi\colon U \rightarrow U_1x \wedge S \in \mathcal{L}(U_1)\}.$$

Analogously,

$$\mathcal{R}(U)$$
$$= \{S \mid \exists\varphi\colon U \rightarrow xS \vee \exists\varphi\colon U \rightarrow xU_1 \wedge S \in \mathcal{R}(U_1)\}.$$

The algorithm for finding $\mathcal{L}$ and $\mathcal{R}$ for all symbols $U \in \mathcal{V} - \mathcal{B}$ involves searching Φ for appropriate syntactic rules.

The precedence relations can be represented by a matrix $\mathbf{M}$ with elements $\mathbf{M}_{ij}$ representing the relation between the ordered symbol pair (S_i, S_j). The matrix clearly has many rows and columns as there are symbols in the vocabulary $\mathcal{V}$.

Assuming that an arbitrary ordering of the symbols of $\mathcal{V}$ has been made ($\mathcal{V} = \{S_1, S_2, \cdots, S_n\}$), an *algorithm* for the determination of the *precedence matrix* $\mathbf{M}$ can be indicated as follows:

For every element φ and Φ which is of the form

$$U \rightarrow S_1 S_2 \cdots S_m$$

and for every pair S_i, S_{i+1} $(i = 1 \cdots m-1)$ assign

(a) $\doteq$ to $\mathbf{M}_{i,i+1}$,

(b) $\lessdot$ to all $\mathbf{M}_{i,k}$ with column index k such that $S_k \in \mathcal{L}(S_{i+1})$,

(c) $\gtrdot$ to all $\mathbf{M}_{k,i+1}$ with row index k such that $S_k \in \mathcal{R}(S_i)$,

(d) $\gtrdot$ to all $\mathbf{M}_{l,k}$ with indices l, k such that $S_l \in \mathcal{R}(S_i)$ and $S_k \in \mathcal{L}(S_{i+1})$.

Assignments under (b) occur only if $S_{i+1} \in \mathcal{V} - \mathcal{B}$, under (c) only if $S_i \in \mathcal{V} - \mathcal{B}$, and under (d) only if both S_i, $S_{i+1} \in \mathcal{V} - \mathcal{B}$, because $\mathcal{L}(S)$ and $\mathcal{R}(S)$ are empty sets for all $S \in \mathcal{B}$.

A syntax is a *simple precedence syntax*, if and only if at most one relation holds between any ordered pair of symbols.

3. *Examples*

(a) $\mathcal{G}_1 = (\mathcal{V}_1, \Phi_1, \mathcal{B}_1, S)$ $\quad \Phi_1 : S \rightarrow H”$
$\mathcal{V}_1 = \{S, H, \lambda, ”\}$ $\quad H \rightarrow ”$
$\mathcal{B}_1 = \{\lambda, ”\}$ $\quad H \rightarrow H\lambda$
$\quad H \rightarrow HS$

Assume that S stands for "string" and H for "head," then this phrase structure system would define a string as consisting of a sequence of string elements enclosed in quotation marks, where an element is either λ or another (nested) string:

U	$\mathcal{L}(U)$	$\mathcal{R}(U)$
S	$”H$	$”$
H	$”H$	$”\lambda S$

$\mathbf{M}$	S	H	λ	$”$
S	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
H	$\doteq$	$\lessdot$	$\doteq$	$\textcircled{\leq}$
λ	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$”$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$

Since both $H \doteq ”$ and $H \lessdot ”$, $\mathcal{G}_1$ is not a precedence syntax. It is intuitively clear that either nested strings should be delineated by distinct opening and closing marks ($\mathcal{G}_2$), or that no nested strings should be allowed ($\mathcal{G}_3$).

(b) $\mathcal{G}_2 = (\mathcal{V}_2 ; \Phi_2, \mathcal{B}_2, S)$ $\quad \Phi_2 : S \rightarrow H’ \; (\varphi_1)$
$\mathcal{V}_2 = \{S, H, \lambda, ‘, ’\}$ $\quad H \rightarrow ‘ \; (\varphi_2)$
$\mathcal{B}_2 = \{\lambda, ‘, ’\}$ $\quad H \rightarrow H\lambda \; (\varphi_3)$
$\quad H \rightarrow HS \; (\varphi_4)$

U	$\mathcal{L}(U)$	$\mathcal{R}(U)$
S	$‘H$	$’$
H	$‘H$	$‘\lambda S’$

$\mathbf{M}$	S	H	λ	$‘$	$’$
S	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
H	$\doteq$	$\lessdot$	$\doteq$	$\lessdot$	$\doteq$
λ	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$‘$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$’$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$

$\mathcal{G}_2$ is a precedence syntax.

(c) $\mathcal{G}_3 = (\mathcal{V}_1, \Phi_3, \mathcal{B}_1, S)$
$\Phi_3 : S \rightarrow H”$
$\quad H \rightarrow ”$
$\quad H \rightarrow H\lambda$

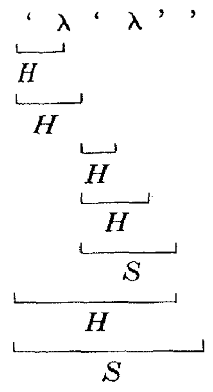
U	$\mathcal{L}(U)$	$\mathcal{R}(U)$
S	$”H$	$”$
H	$”H$	$”\lambda$

M	S	H	λ	$”$
S				
H			$\doteq$	$\doteq$
λ			$\gtrdot$	$\gtrdot$
$”$			$\gtrdot$	$\gtrdot$

$\mathcal{G}_3$ is a precedence syntax.

As an illustration for the parsing algorithm, we choose the parsing of a sentence of $\mathcal{L}(\mathcal{G}_2)$:

	$‘\,\lambda\,‘\,\lambda\,’\,’$
φ_2:	$H\,\lambda\,‘\,\lambda\,’\,’$
φ_3:	$H\,‘\,\lambda\,’\,’$
φ_2:	$H\,H\,\lambda\,’\,’$
φ_3:	$H\,H\,’\,’$
φ_1:	$H\,S\,’$
φ_4:	$H\,’$
φ_1:	S



4. *The Uniqueness of a Parse.* The three previous examples suggest that the property of unique precedence relationship between all symbol pairs be connected with uniqueness of a parse for any sentence of a language. This relationship is established by the following theorem.

THEOREM. *The given parsing algorithm yields the canonical form of the parse for any sentence of a precedence phrase structure language, if there exist no two syntactic rules with the same right part. Furthermore, this canonical parse is unique.*

This theorem is proven, if it can be shown that in any sentence its directly reducible parts are disjoint. Then the algorithm, proceeding strictly from left to right, produces the canonical parse, which is unique, because no reducible substring can apply to more than one syntactic rule.

The *proof* that all directly reducible substrings are disjoint is achieved indirectly: Suppose that the string $S_1 \cdots S_n$ contains two directly reducible substrings (a) $S_i \cdots S_k$ and (b) $S_j \cdots S_l$, where $1 \leqq i \leqq j \leqq k \leqq l \leqq n$. Then because of (a) it follows from the definition of the precedence relations that $S_{j-1} \doteq S_j$ and $S_k \gtrdot S_{k+1}$, and because of (b) $S_{j-1} \lessdot S_j$ and $S_k \doteq S_{k+1}$. Therefore this sentence cannot belong to a precedence grammar.

Since in particular the leftmost reducible substring is unique, the syntactic rule to be applied is unique. Because the new sentence again belongs to the precedence language, the next reduction is unique again. It can be shown by induction, that therefore the entire parse must be unique.

From the definition of the meaning of a phrase structure programming language it follows that its *meaning is un-*

ambiguous for all sentences, if the underlying syntax is a precedence syntax.

5. *Precedence Functions.* The given parsing algorithm refers to a matrix of precedence relations with n^2 elements, where n is the number of symbols in the language. For practical compilers this would in most cases require an extensive amount of storage space. Often the precedence relations are such that two numeric functions (f, g) ranging over the set of symbols can be found, such that for all ordered pairs (S_i, S_j):

(a) $f(S_i) = g(S_j) \leftrightarrow S_i \doteq S_j$
(b) $f(S_i) < g(S_j) \leftrightarrow S_i < S_j$
(c) $f(S_i) > g(S_j) \leftrightarrow S_i > S_j$

If these functions exist and the parsing algorithm is adjusted appropriately, then the amount of elements needed to represent the precedence information reduces from n^2 to $2n$.

In example $\mathcal{G}_2$, e.g., the precedence matrix can be represented by the two functions f and g, where

S =	s	h	λ	‘	’
$f(S)$ =	3	1	3	3	3
$g(S)$ =	1	2	1	2	1

A precedence phrase structure syntax for which these precedence functions do not exist is given presently:

$$\mathcal{V} = \{A, B, C, \lambda, [,]\}$$
$$\mathcal{B} = \{\lambda, [,]\}$$
$$\begin{aligned} \Phi:\ & A \rightarrow C\ B\ ] \\ & A \rightarrow [\ \ ] \\ & B \rightarrow \lambda \\ & B \rightarrow \lambda\ A \\ & B \rightarrow A \\ & C \rightarrow [\end{aligned}$$

It can be verified that this is a precedence syntax and in particular the following precedence relations can be derived:

$$\lambda < [, \qquad [> [, \qquad [\doteq], \qquad \lambda >].$$

Precedence functions f and g would thus have to satisfy

$$f(\lambda) < g([) < f([) = g(]) < f(\lambda)$$

which clearly is a contradiction. Precedence functions therefore do not exist for this precedence syntax.

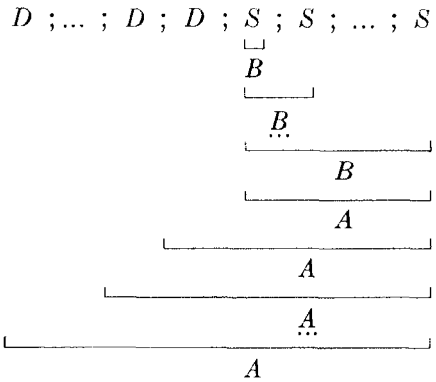
6. *Higher Order Precedence Syntax.* It is the purpose of this section to redefine the precedence relationships more generally, thus enlarging the class of precedence phrase structure systems. This is desirable, since for precedence languages a constructive parsing algorithm has been presented which is instrumental in the definition of the meaning of the language. The motivation for the manner in which the precedence relationships will be generalized is first illustrated in an informal way by means of examples. These examples are phrase structure systems which for one or another reason might be likely to occur in the definition of a language, but which also violate the rules for simple precedence syntax.

Example 1

$$\mathcal{V} = \{A, B, ;, S, D\}$$
$$\mathcal{B} = \{;, S, D\}$$
$$\begin{aligned} \Phi:\ & A \rightarrow B \\ & A \rightarrow D\ ;A \\ & B \rightarrow S \\ & B \rightarrow B\ ;S \end{aligned}$$

$S \in \mathcal{L}(A)$, thus $; < S$, and also $; \doteq S$.

This syntax produces sequences of D's separated by ";", followed by a sequence of symbols S, also separated by ";". A parse is constructed as follows:



The sequence of S's is defined using a left-recursive definition while the sequence of D's is defined using a right-recursive definition. The precedence violation occurs, because for both sequences the same separator symbol is used.

The difficulty arises when the symbol sequence ";S" occurs. It is then not clear whether both symbols should be included in the same substring or not. The decision can be made, if the immediately *preceding* symbol is investigated.

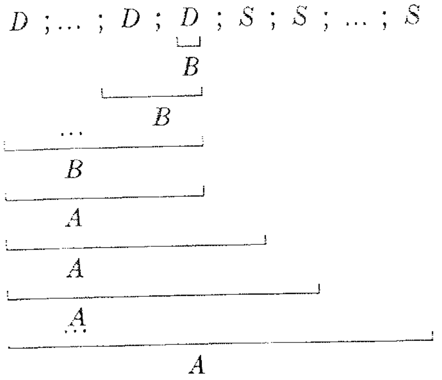
In other words, not only two single symbols should be related, but a symbol and the string consisting of the two previously obtained symbols. Thus $B; \doteq S$ and $D; < S$.

Example 2

$$\mathcal{V} = \{A, B, ;, S, D\}$$
$$\mathcal{B} = \{;, S, D\}$$
$$\begin{aligned} \Phi:\ & A \rightarrow B \\ & A \rightarrow A\ ;S \\ & B \rightarrow D \\ & B \rightarrow D\ ;B \end{aligned}$$

$D \in \mathcal{R}(A)$, thus $D > ;$ and also $D \doteq ;$.

This syntax produces the same strings as the preceding one, but with a different syntactic structure:



Here the same difficulty arises upon encountering the symbol sequence "D;". The decision whether to include both symbols in the same syntactic category or not can be reached upon investigating the *following* symbol. Explicitly, a symbol should be related to the subsequent string of two symbols, i.e., $D \doteq ;D$ and $D \gtrdot ;S$.

Example 3

$$\mathcal{V} = \{A, B, \lambda, ;, [,]\}$$
$$\mathcal{B} = \{\lambda, ;, [,]\}$$
$$\Phi\colon A \to B \,;\, B$$
$$B \to [\,A\,]$$
$$B \to [\,\lambda\,]$$
$$B \to \lambda$$

Since $\lambda \in \mathcal{L}(A)$ and $\lambda \in \mathcal{R}(A)$: $[\ \lessdot \lambda$ and $\lambda \gtrdot\ ]$, but also $[\ \doteq \lambda$ and $\lambda \doteq\ ]$. In this case the following relations must be established to resolve the ambiguity:

$$[\ \doteq \lambda],\qquad [\ \lessdot \lambda;,\qquad ;\lambda \gtrdot\ ],\qquad [\lambda \doteq\ ].$$

This syntax therefore combines the situations arising in Examples 1 and 2. Obviously, examples could be created where the strings to be related would be of length greater than 2. We will therefore call a precedence phrase structure system to be of order (m, n), if unique precedence relations can be established between strings of length equal to or less than m and strings of length equal to or less than n. Subsequently, a more precise definition is stated. A set of extended rules must be found which define the generalized precedence relations. The parsing algorithm, however, remains the same, with the exception that not only the symbols S_i and P_k be related, but possibly the strings $S_{i-m} \cdots S_i$ and $P_k \cdots P_{k+n}$.

The definitions of the relations $\lessdot$, $\doteq$, $\gtrdot$ is as follows: Let $x = S_{-m} \cdots S_{-1}$, $y = S_1 \cdots S_n$, let $u, v, u', v' \in \mathcal{V}^*$ and $U, U_1, U_2 \in \mathcal{V} - \mathcal{B}$. Then

(a) $x \doteq y$, if and only if there exists a syntactic rule $U \to uS_{-1}S_1v$, and $uS_{-1} \xrightarrow{*} u'x$, $S_1v \xrightarrow{*} yv'$;

(b) $x \lessdot y$, if and only if there exists a syntactic rule $U \to uS_{-1}U_1v$, and $uS_{-1} \xrightarrow{*} u'x$, $U_1v \xrightarrow{*} yv'$;

(c) $x \gtrdot y$, if and only if there exists a syntactic rule $U \to uU_1S_1v$, and $uU_1 \xrightarrow{*} u'x$, $S_1v \xrightarrow{*} yv'$, or there exists a syntactic rule $U \to uU_1U_2v$ and $uU_1 \xrightarrow{*} u'x$, $U_2v \xrightarrow{*} yv'$.

A syntax is said to be a *precedence syntax of order* (m, n), if and only if (a) it is not a precedence syntax of order (m', n') for $m' < m$ or $n' < n$, and (b) for any ordered pair of strings $S_{-m'} \cdots S_{-1}$, $S_1 \cdots S_{n'}$, where $m' \leqq m$ and $n' \leqq n$, either at most one of the three relations $\lessdot$, $\doteq$, $\gtrdot$ holds, or otherwise (b) is satisfied for the pair $S_{-(m'+1)} \cdots S_{-1}$, $S_1 \cdots S_{n'+1}$.

A precedence syntax of order (1, 1) is called a *simple* precedence syntax. With the help of the sets of leftmost and rightmost strings, the definitions of the precedence relations can be reformulated analogously to their counterparts in Section III B2, subject to the condition that there exists no rule $U \to \Lambda$.

$$\text{(a)}\quad x \doteq y \leftrightarrow \exists\varphi(\varphi\colon U \to uS_{-1}S_1v)$$
$$\wedge\ (u'S_{-m} \cdots S_{-2} = u \vee S_{-m} \cdots S_{-2} \in \mathcal{R}^{(m-1)}(u))$$
$$\wedge\ (S_2 \cdots S_nv' = v \vee S_2 \cdots S_n \in \mathcal{L}^{(n-1)}(v))$$
$$\text{(b)}\quad x \lessdot y \leftrightarrow \exists\varphi(\varphi\colon U \to uS_{-1}U_1v)$$
$$\wedge\ (u'S_{-m} \cdots S_{-2} = u \vee S_{-m} \cdots S_{-2} \in \mathcal{R}^{(m-1)}(u))$$
$$\wedge\ (S_1 \cdots S_n \in \mathcal{L}^{(n)}(U_1v))$$
$$\text{(c)}\quad x \gtrdot y \leftrightarrow \exists\varphi(\varphi\colon U \to uU_1S_1v)$$
$$\wedge\ (S_{-m} \cdots S_{-1} \in \mathcal{R}^{(m)}(uU_1))$$
$$\wedge\ (S_1 \cdots S_nv' = v \vee S_2 \cdots S_n \in \mathcal{L}^{(n-1)}(v))$$
$$\vee\ \exists\varphi(\varphi\colon U \to uU_1U_2v)$$
$$\wedge\ (S_{-m} \cdots S_{-1} \in \mathcal{R}^{(m)}(uU_1)$$
$$\wedge\ (S_1 \cdots S_n \in \mathcal{L}^{(n)}(U_2v))$$

$\mathcal{L}^{(n)}(s)$ and $\mathcal{R}^{(n)}(s)$ are then defined as follows:

$$z = Z_1 \cdots Z_n \in \mathcal{L}^{(n)}(Uu) \leftrightarrow \exists k(1 \leqq k \leqq n) \ni (U \xrightarrow{*} Z_1 \cdots Z_k) \wedge (Z_k \cdots Z_nu' = u \vee Z_k \cdots Z_n \in \mathcal{L}^{(n-k)}(u)) \tag{1}$$

$$z = Z_1 \cdots Z_n \in \mathcal{L}^{(n)}(U) \leftrightarrow \exists k(0 \leqq k \leqq n) \ni (U \to Z_1 \cdots Z_ku \wedge Z_k \cdots Z_n \in \mathcal{L}^{(n-k)}(u)) \tag{1a}$$

$$z = Z_n \cdots Z_1 \in \mathcal{R}^{(n)}(uU) \leftrightarrow \exists k(1 \leqq k \leqq n) \ni (u'Z_n \cdots Z_{k+1} = u \vee Z_n \cdots Z_{k+1} \in \mathcal{R}^{(n-k)}(u)) \wedge (U \xrightarrow{*} Z_k \cdots Z_1) \tag{2}$$

$$z = Z_n \cdots Z_1 \in \mathcal{R}^{(n)}(U) \leftrightarrow \exists k(0 \leqq k \leqq n) \ni (U \to uZ_k \cdots Z_1 \wedge Z_n \cdots Z_{k+1} \in \mathcal{R}^{(n-k)}(u)) \tag{2a}$$

These formulas indicate the method for effectively finding the sets $\mathcal{L}$ and $\mathcal{R}$ for all symbols in $\mathcal{V} - \mathcal{B}$. In particular, we obtain for $\mathcal{L}^{(1)}$ and $\mathcal{R}^{(1)}$ the definitions for $\mathcal{L}$ and $\mathcal{R}$ without superscript as defined in Section III B2.

Although for practical purposes such as the construction of a useful programming language no precedence syntax of order greater than (2, 2)—or even (2, 1)—will be necessary; a general approach for the determination of the precedence relations of any order is outlined subsequently.

First it is to be determined whether a given syntax is a precedence syntax of order (1, 1). If it is not, then for all pairs of symbols (S_i, S_k) between which the relationship is not unique, it has to be determined whether all relations will be unique between either (S_jS_i, S_k) or (S_i, S_kS_j), where S_j ranges over the entire vocabulary. According to the outcome, one obtains a precedence syntax of order (2, 1), (1, 2) or (2, 2), or if some relations are still not unique, one has to try for even higher orders. If at some stage it is not possible to determine relations between the strings with the appended symbol S_j ranging over the *entire* vocabulary, then the given syntax is no precedence syntax at all. For example,

$$\mathcal{V} = \{A, B, \lambda, [,]\}$$

$$\mathcal{B} = \{\lambda, [,]\}$$

$$\begin{aligned}\Phi\colon & A \rightarrow B \\ & A \rightarrow [\, B\,] \\ & B \rightarrow \lambda \\ & B \rightarrow [\, \lambda\,].\end{aligned}$$

The conflicting relations are $[\lessdot \lambda$, $[\doteq \lambda$, $\lambda \doteq]$ and $\lambda \gtrdot]$. But a relation between $(S[, \lambda)$ or $(\lambda,]S)$ can be established for no symbol S whatsoever, and between $([, \lambda S_1)$ and $(S_2\lambda,])$ only for $S_1 =]$ and $S_2 = [$. Thus this is no precedence syntax.

Clearly there exist two different parses for the string [λ], namely,

$$\underbrace{[\underbrace{\lambda}_{B}]}_{A} \quad \text{and} \quad \underbrace{\underbrace{[\lambda]}_{B}}_{A}$$

The underlying phrase structure systems in Section III.C and IV are simple precedence phrase structure systems.

C. Example of a Simple Precedence Phrase Structure Programming Language. A simple phrase structure programming language shall serve as an illustration of the presented concepts. This language contains the following constructs which are well known from Algol 60: variables, arithmetic expressions, assignment statements, declarations and the block structure. Themeaning of the language is explained in terms of an array of variables, called the *value stack*, which has to be understood as being associated with the array S which is instrumental in the parsing algorithm. The variable V_i represents the "value" associated with the symbol S_i. E.g., the interpretation rule ψ_{11} corresponding to the syntactic rule φ_{11} determines the value of the resulting symbol **expr-** as the sum of the values of the symbols **expr-** and **term** belonging to the string to be reduced:

φ_{11} : **expr-** → **expr-** + **term**
ψ_{11} : $V_j \leftarrow V_j + V_i$, [V(**expr-**) ← V(**expr-**) + V(**term**)]

Note that the string to be reduced has been denoted by $S_j \cdots S_i$ in the parsing algorithm of Section III.B1. Instead of thus making explicit reference to a particular parsing algorithm, $V_i \cdots V_j$, the values of the symbols $S_i \cdots S_j$, can be denoted explicitly; i.e., instead of V_i and V_j in ψ_{11} one might write V(**term**) and V(**expr-**) respectively. For the sake of brevity, the subscripts i and j have been preferred here.

A second set of variables is called the *name stack*. It serves to represent a second value of certain symbols, which can be considered as a "name." The symbol **decl** is actually the only symbol with two values; it represents a variable of the program in execution which has a name (namely, its associated identifier) and a value (namely, the value last assigned to it by the program). The syntax of the language is such that the symbol **decl** remains in the parse-stack S as long as the declaration is valid, i.e., until the block to which the declaration belongs is closed. This is achieved by defining the sequence of declarations in the head of a block by the right-recursive syntactic rule φ_4. The parse of a sequence of declarations illustrates that the declarations can only be involved in a reduction together with a **body-** symbol after a symbol **body-** has originated through some other syntactic reduction. This, in turn, is only possible when the symbol **end** is encountered. The **end** symbol then initiates a whole sequence of reductions which result in the collapsing of the part of the stack which represented the closing block. On the other hand, the sequence of statements which constitutes the imperative part of a block, is defined by the left-recursive syntactic formula φ_6. Thus a statement reduces with a preceding statement-list into a statement-list immediately, because there is no need to retain information about the executed statement in the value-stack.

This is a typical example where the syntax is engaged in the definition of not only the structure but also the meaning of a language. The consequence is that in constructing a syntax one has to be fully aware of the meaning of a constituent of the language and its interaction with other constituents. Many other such examples appear in Section IV. It is, however, not possible to enumerate and discuss every particular consideration which had to be made during the construction of the language. Only a detailed study and analysis of the language can usually reveal the reasons for the many decisions which were taken in its design.

Subsequently the formal definition of the simple phrase structure language is given:

$$\mathcal{L}_p = (\mathcal{V}, \Phi, \mathcal{B}, \textbf{program}, \Psi, \mathcal{E})$$

$$\begin{aligned}\mathcal{V} - \mathcal{B} = \{&\textbf{program} \mid \textbf{block} \mid \textbf{body} \mid \textbf{body-} \mid \textbf{decl} \mid \\ &\textbf{statment} \mid \textbf{statlist} \mid \textbf{expr} \mid \textbf{expr-} \mid \textbf{term} \mid \textbf{term-} \mid \\ &\textbf{factor} \mid \textbf{var} \mid \textbf{number} \mid \textbf{digit}\}\end{aligned}$$

$$\begin{aligned}\mathcal{B} = \{&\lambda \mid \textbf{begin} \mid \textbf{end} \mid ; \mid , \mid \leftarrow \mid + \mid - \mid \times \mid / \mid (\mid) \mid \\ &0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid \textbf{new} \mid \perp\}\end{aligned}$$

$$\mathcal{E} = \{S, V, W, i\}$$

Φ:		Ψ:
φ_1 : program	→ ⊥ block ⊥	$\psi_1 : \Lambda$ (empty)
φ_2 : block	→ **begin** body **end**	$\psi_2 : \Lambda$
φ_3 : body	→ body-	$\psi_3 : \Lambda$
φ_4 : body-	→ decl; body-	$\psi_4 : W_j \leftarrow \Omega$
φ_5 : body-	→ statlist	$\psi_5 : \Lambda$
φ_6 : statlist	→ statlist, statment	$\psi_6 : \Lambda$
φ_7 : statlist	→ statment	$\psi_7 : \Lambda$
φ_8 : statment	→ var ← expr	$\psi_8 : V_{V_j} \leftarrow V_i$
φ_9 : statment	→ block	$\psi_9 : \Lambda$
φ_{10} : expr	→ expr-	$\psi_{10} : \Lambda$
φ_{11} : expr-	→ expr- + term	$\psi_{11} : V_j \leftarrow V_j + V_i$
φ_{12} : expr-	→ expr- − term	$\psi_{12} : V_j \leftarrow V_j - V_i$
φ_{13} : expr-	→ −term	$\psi_{13} : V_j \leftarrow -V_i$
φ_{14} : expr-	→ term	$\psi_{14} : \Lambda$
φ_{15} : term	→ term-	$\psi_{15} : \Lambda$
φ_{16} : term-	→ term- × factor	$\psi_{16} : V_j \leftarrow V_j \times V_i$
φ_{17} : term-	→ term- / factor	$\psi_{17} : V_j \leftarrow V_j/V_i$
φ_{18} : term-	→ factor	$\psi_{18} : \Lambda$
φ_{19} : factor	→ var	$\psi_{19} : V_j \leftarrow V_{V_j}$
φ_{20} : factor	→ (expr)	$\psi_{20} : V_j \leftarrow V_{j+1}$
φ_{21} : factor	→ number	$\psi_{21} : \Lambda$
φ_{22} : var	→ λ	$\psi_{22} : t \leftarrow j$ $t \leftarrow t - 1$ **if** $t = 0$ **then** $\xrightarrow{\text{ERROR}}$ **if** $W_t \neq S_j$ **then** $V_j \leftarrow t$
φ_{23} : number	→ digit	$\psi_{23} : \Lambda$
φ_{24} : number	→ number digit	$\psi_{24} : V_j \leftarrow V_j \times 10$ $V_j \leftarrow V_j + V_i$
φ_{25} : decl	→ **new** λ	$\psi_{25} : W_j \leftarrow S_i$ $V_j \leftarrow \Omega$
φ_{26} : digit	→ 0	$\psi_{26} : V_j \leftarrow 0$
φ_{27} : digit	→ 1	$\psi_{27} : V_j \leftarrow 1$
⋮ ⋮		⋮ ⋮
φ_{35} : digit	→ 9	$\psi_{35} : V_j \leftarrow 9$

Notes:

1. The branch in rule ψ_{22} labeled with ERROR is an example for the indication of a "semantic error" in $\mathfrak{L}_p$. By "semantic error" is in general meant a reaction of an interpretation rule which is not explicitly defined. In the example of ψ_{22} the labeled branch is followed when no identifier equal to S_i is found in the W stack, i.e., when an "undeclared" identifier is encountered.

2. The basic symbol λ in $\mathfrak{V}$ is here meant to act as a representative of the class of all identifiers. Nothing will be said about the representation of identifiers.

The matrix **M** of precedence relations (Table I)[2] and the precedence functions f and g (Table II)[2] were determined by a syntax-processor program written in Extended Algol for the Burroughs B5500 computer [17].

REFERENCES

1. Naur, P. (Ed.) Report on the algorithmic language ALGOL 60. *Comm. ACM 3* (May 1960), 299–314.
2. ——. Revised report on the algorithmic language ALGOL 60. *Comm. ACM 6* (Jan. 1963), 1–17.

[2] Tables I and II will appear with Part II of the paper.

(*Continued on page 25*)

Contributions to the Communications of the ACM

The *Communications of the ACM* serves as a newsletter to members about the activities of the Association for Computing Machinery, and as a publication medium for original papers and other material of interest. Material intended for publication may be sent to the Editor-in-Chief, or directly to the Editor of the appropriate department. It will also be considered for the *Journal of the Association for Computing Machinery*.

Contents—Submissions should be relevant to the interests of the Association and may take the form of short contributions or original papers. Short contributions may be published as letters to the editor, or in the news and notices department. Papers should be reports on the results of research, or expositional or survey articles. Research papers are judged primarily on originality; expositional and survey articles are judged on their topicality, clarity and comprehensiveness. Contributions should conform to generally accepted practices for scientific papers with respect to style and organization.

Format—Manuscripts should be submitted in duplicate (the original on bond paper) and the text should be double spaced on *one side* of the paper. Typed manuscripts are preferred, but good reproductions of internal reports are acceptable. Authors' names should be given without titles or degrees. The name and address of the organization for which the work was carried out should be given. If the paper has previously been presented at a technical meeting, the date and sponsoring society should appear in a footnote off the title.

Synopses—Manuscripts should be accompanied by an author's synopsis of not more than 175 words, setting out the essential feature of the work. This synopsis should be intelligible in itself without reference to the paper and should not include reference numbers. The opening sentence should avoid repetition of the title and indicate the subjects covered.

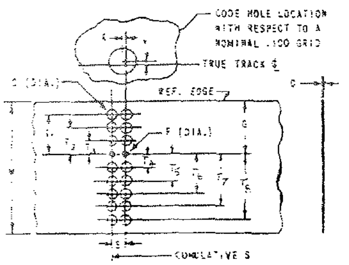
Figures—Diagrams should be on white bond or drafting linen. Lettering should be done professionally with a Leroy ruler (or, if necessary, in *clear black* typing, with carbon reproducible ribbon). Photographs should be glossy prints. The author's name and the figure number should appear on the back of each figure. On publication, figures will be reduced to 3½ inches in width.

Citations—(1) *References to items in periodicals:* These should take the form: author, title, journal, volume number, date, pages. For authors, last names are given first, even for multiple authors; if an editor, the author's name is followed b[illegible] T[illegible] ends with a period, either the period [illegible] or a period for the purpose. The [illegible] names (or their derivatives) starting with capital letters, and it ends with a period. The date is given in parentheses. [illegible] for abbreviations is that recommended by the [illegible] organization. Example:
Jones, R. W., Mark[illegible] T. Programming routines for Boolean functi[illegible] 5-19.
(2) *References to report*[illegible] name(s) and title—same style as for references i[illegible] rt number, city, date.
(3) *References to books:* Author(s) (same style as to periodicals). Title—all principal words start with a capital letter, and the title is underlined so that it will be set in italics. Page or chapter references follow the title, then a period. Publisher, city, year.
(4) *In lengthy bibliographies*, entries must be arranged alphabetically according to authors or editors names, except for those items to which no names can be attached.

Copyright—If material submitted for publication has previously been copyrighted, appropriate releases should accompany the submission. Copyright notices will be inserted when reprinting such material. If the author wishes to reserve the copyright of a computer program, upon his request a copyright notice in his name will be included when the program is published.

Page Charge—Author's institutions or corporations are requested to honor a page charge of $40 per printed page, [illegible] r pages or part thereof, to help defray the cost [illegible] levied [illegible] resear[illegible] riented [illegible] hms, and letters of technical content. [illegible] furnished free of charge. Payment of [illegible] f publication, editorial acceptance of a [illegible] ent or nonpayment.

CHART II. International Comparison of Dimensional Standards for One Inch Perforated Paper Tape



I.S.O. CONVERSION TABLE **

IN.	MM.	IN.	MM.
1.000	25.40	.010	.25
.392	9.96	.004	.10
.100	2.54	.003	.075
.072	1.83	.002	.05
.050	1.27	.001	.025
.046	1.17	.0003	.008
.025	.63	.0002	.005
.015	.38		

NOTES:

** THESE CONVERSIONS WERE PRESENTED IN DOCUMENT ISO/TC 97/SC4 (SEC.-1)27 AND, WHEN APPLICABLE, WERE USED IN THE COMPARISON TABLE BELOW.

△ PREFIXED BY THE NUMBER ISO/TC 97/SC 4/WG 3

□ CZECHOSLOVAKIA ANTICIPATES CHANGING TO 1.000 + .003 AT A LATER DATE.

⊖ NO TOLERANCE ASSIGNED.

‡ WHERE n IS AN INTEGER.

* DEVIATIONS FROM I.S.O. PROPOSALS.

• TOLERANCE ON ".100n" IS SAME AS "Y".

DIMENSIONS AND VALUES	I.S.O. PROPOSALS ISO/TC 97/SC4(SEC.-1)27 DEC., 1964 X3.2/189 JAN. 4, 1965	CHOICES OF MEMBERS WITH REFERENCE TO I.S.O. PROPOSALS									DRAFT I.S.O. PROPOSAL (TURIN'65-6)32 MAY 21, 1965
		UNITED STATES △(U.S.A.-1)14 X3.2/195 JAN. 20, 1965	ITALY △(ITALY-1)16 JAN., 1965 X3.2/209 FEB. 15, 1965	UNITED KINGDOM △(U.K.-1)13 JAN., 1965 X3.2/211 FEB. 16, 1965	CZECHOSLOVAKIA △(CZECH.-1)11 FEB., 1965 X3.2/219 MAR. 3, 1965 X3.2.2/17 MAR. 3, 1965	FRANCE △(FRANCE-1)12 FEB., 1965 X3.2/220 MAR. 3, 1965 X3.2.2/18 MAR. 3, 1965	GERMANY △(GERMANY-1)11 FEB., 1965 X3.2/221 MAR. 8, 1965 X3.2.2/19 MAR. 3, 1965	SPAIN △(SPAIN-1)16 MAR., 1965	E.C.M.A. △(ECMA-1)17 MAR. 31, 1965 X3.2/224 MAR. 31, 1965	JAPAN △(JAPAN-1)23 MAY, 1965	
W	1.000 − .002 OR + .003	1.000 − .003	1.000 ± .002	1.000 ± .002	□*1.000 ± .004	1.000 ± .002	1.000 ± .003	1.000 ± .002	1.000 ± .002	*1.000 ±.004	1.000(NOMINAL)
D	.004 ± .0002 OR ± .0003	.004 ± .0003	.004 ± .0003	⊖*.004	*.004 ± .0004	.004 ± .0003	.004 ± .0003	.004 ± .0003	.004 ± .0002	.004 ±.0002	.004(NOMINAL)
G	.392 ± .004	*.392 ± .003	*.392 ± .005	*.392 ± .003	*.394 ± .004	.392 ± .004	.392 ± .004	.392 ± .004	.392 ± .004	*$.392 {}^{+.004}_{-.008}$	.392 ±.004
T_1 to T_8	‡.100n ± .002	.100n ± .002	.100n ± .002	.100n ± .002	.100n ± .002	.100n ± .002	.100n ± .002	.100n ± .002	.100n ± .002	.100n ±.002	.100n ±.002
F	$.046 {}^{+.002}_{-.001}$ OR $ {}^{+.003}_{-.001}$	$.046 {}^{+.002}_{-.001}$	$.046 {}^{+.002}_{-.001}$	*.046 ± .001	*.047 ± .001	$.046 {}^{+.002}_{-.001}$	$.046 {}^{+.003}_{-.001}$	$046 {}^{+.002}_{-.001}$	$.046 {}^{+.002}_{-.001}$	$.046 {}^{+.003}_{-.001}$	$.046 {}^{+.002}_{-.001}$
C	$.072 {}^{+.001}_{-.002}$ OR ± .002	.072 ± .002	$.072 {}^{+.001}_{-.002}$	*.072 ± .001	*.071 ± .001	.072 ± .002	.072 ± .002	.072 ± .002	$.072 {}^{+.001}_{-.002}$	.072 ±.002	.072 ±.002
X	± .003	± .003	± .003	* ± .002	* ± .002	* NOT GIVEN	± .003	± .003	± .003	±.003	±.003
•Y	± .002	± .002	± .002	± .002	± .002	± .002	± .002	± .002	± .002	-	-
S	.100 ± .002 OR ± .003	.100 ± .003	.100 ± .003	.100 ± .003	* .100 ± .001	.100 ± .003	.100 ± .002	.100 ± .004	.100 ± .003	.100 ±.003	.100 ±.003
CUM. S (10 SPACES)	±.010 OR ±.015	010	± .010	+ .010	* NOT GIVEN	± .010	* NOT GIVEN	+ .010	+ .010	±.010	+.010
CUM. S (50 SPACES)	±.025 TO ±.050	025	± .035	* NOT GIVEN	* ± .020	+ .025	+ .035	+ .025	* NOT GIVEN	±.025	±.025
CUM. S (100 SPACES)	NOT GIVEN	—	—	* ±.050	—	—	—	—	* ± .059	-	-
REL. HUMIDITY	50%	(50 ± 2)%	(50 ± 2)%	(50 ± 2)%	* (65 ± 2)%	(50 ± 5)%	* (65 ± 5)%	* NOT GIVEN	* NOT GIVEN	* 65%	(50 ± 2)%
TEMPERATURE	20°C	*(23 ± 2)°C	(20 ± 2)°C	(20 ± 2)°C	(20 ± 2)°C	* (23 ± 2)°C	(20 ± 2)°C	* NOT GIVEN	* NOT GIVEN	20°C	(23 ± 2)°C

environments may disclose dimensional variance beyond the limits of this standard.

It is highly desirable, for physical interchange of punched tapes, that the paper tape should be punched in an environment near the standard of 50% RH ± 2% RH and 73°F ± 3.5°F, say within the range of 40% to 60% relative humidity. If, however, due to uncontrolled circumstance, the environmental conditions at the time of punching are above 60% RH or below 40% RH, respectively, it may prove necessary to make a special arrangement between the sender and the recipient in order to ensure satisfactory reading.

A3. *Tape Materials Other Than Paper.* The physical dimensions specified in this standard with the exception of tape thickness (paragraph 2.2) also apply where materials other than paper are used.

Expository Remarks

1. Existing and proposed domestic and foreign standards were reviewed. The resolution of domestic differences is shown in the attached Chart I. A summary of international positions is contained in attached Chart II.

In general, the variations between the standards were in minor tolerance differences. A more substantial difference existed with respect to allowable cumulative error in pitch. The cumulative pitch error variation was resolved in favor of the tolerances stated in this Standard because those tolerances were satisfactory for almost all existing equipment.

Achieving international agreement required that the United States specify only the nominal value of width and thickness, increase the tolerance on the distance between the feed hole track and the reference edge and increase the allowable accumulative pitch error.

WIRTH AND WEBER—*cont'd from page 23*

3. Böhm, C. The CUCH as a formal and descriptive language. IFIP Working Conf., Baden, Sept. 1964.
4. Landin, P. The mechanical evaluation of expressions. *Comp. J. 6*, (Jan. 1964), 4.
5. ——. A correspondence between ALGOL 60 and Church's Lambda-Notation. *Comm. ACM 8*, 2, 3 (Feb., Mar. 1965), 89–101, 158–165.
6. Church, A. The calculi of lambda-conversion. *Ann. Math. Studies, 6*, (1941), Princeton, N. J.
7. Curry, A., Feys, R., and Craig, W. *Combinatory Logic.* North-Holland Publ., Amsterdam, 1958.
8. van Wijngaarden, A. Generalized ALGOL. *Ann. Rev. Automat. Programming 3*, 17–26.
9. ——. Recursive definition of syntax and semantics. IFIP Working Conf., Baden, Sept. 1964.
10. Garwick, J. V. The definition of programming languages by their compiler. IFIP Working Conf., Baden, Sept. 1964.
11. Floyd, R. W. The syntax of programming languages—a survey. *IEEE Trans. EC-13* (Aug. 1964), 346–353.
12. Bar-Hillel, Y., Perles, M., and Shamir, E. On formal properties of simple phrase structure grammars. *Z. Phonetik, Sprachwiss. Kommunikationsforsch. 14,* 143–172; also in *Language and Information,* Addison-Wesley Publ., Reading, Mass., 1964.
13. Floyd, R. W. A descriptive language for symbol manipulations. *J. ACM 8* (Oct. 1961), 579–584.
14. ——. Syntactic analysis and operator precedence. *J. ACM 10* (July 1963), 316–333.
15. Irons, E. T. Structural connections in formal languages. *Comm. ACM 7* (Feb. 1964), 67–71.
16. Wirth, N. A generalization of ALGOL. *Comm. ACM 6* (Sept. 1963), 547–554.
17. —— and Weber, H. Title. CS20 Tech. Rep., Computer Science Dept., Stanford, U., Stanford, Calif.