Formal systems and analysis of context sensitive languages*

Z. J. Ghandour

Philadelphia Scientific Centre, IBM, 3401 Market Street, Philadelphia, Pennsylvania 19104, USA

Canonic systems, which are applied variants of Post's canonical systems have been used to define sets of strings of symbols recursively. A hierarchy of classes of canonic systems is obtained by introducing restrictions on the general form of canonic systems. One of the classes contains canonic systems which generate only recursive sets including all context sensitive languages. An algorithm has been programmed in APL to parse strings over the alphabet of languages specified by canonic systems of the class mentioned above. A general approach has been taken such that the canonic systems and the parsing algorithm cover a wide variety of applications, as e.g., in constructing syntax directed translators, proof checking systems and linguistical analysis. Some mathematical properties of the classes of canonic systems are also presented.

(Received July 1971)

1. Introduction

This paper describes a process that I have constructed to test whether or not a string of characters belongs to a set of strings that has been defined recursively. If the string belongs to the set, the process gives the structure of the string based on the rules used to generate it. Canonic systems, which are applied variants of Post's canonical systems and Smullyan's elementary formal systems, are used to define sets recursively. Logicians have used canonical systems to give self-contained definitions of mathematical systems in the sense of Smullyan (1961). Similarly, a programming language can be treated as a logistic system, i.e., a set of axioms and a set of rules to generate theorems from the axioms. By Church's thesis, any effective process can be defined recursively.

In their general form, canonic systems are capable of describing any recursively enumerable set. I have obtained a hierarchy of classes of canonic systems by introducing restrictions on the canonic systems. The hierarchy on canonic systems provides an alternative to Chomsky's hierarchy on grammars and to the hierarchy on sequential devices. Theorems about the hierarchy of canonic systems and some of its nodes are presented in this paper.

General parsing algorithms (Feldman and Gries, 1968) used in actual translators are characterised by the use of classes of grammars which are subsets of the class of context free grammars, e.g., simple phrase structure grammars, predictive grammars, dependency grammars, precedence grammars, operator grammars and LR(K) grammars.

However, most computer languages are context sensitive and cannot be generated by context free grammars. For example, the statement GOTO 3 in a FORTRAN program is not syntactically correct unless there exists a statement with a statement label 3 in the same program. A similar problem arises with declarative statements. Any rule requiring that two or more constituent phrases of a construction be identical is beyond the expressive capability of context free grammars.

The restricted canonic systems described in this paper, for which a corresponding parsing algorithm has been constructed, can generate any context sensitive language. Canonic systems specifying context sensitive languages remain as clear and concise as canonic systems specifying context free languages.

Syntax directed translators involve parsing and evaluating strings in source languages. The evaluation algorithm is based on the structure analysis of the string. Syntax directed translators need the grammars of languages in recogniser-oriented form, while the users of the languages need the grammars in generator-oriented form. The canonic systems, which are generator-oriented, and the corresponding parsing algorithms satisfy both needs.

One can construct canonic systems to define proofs in a variety of mathematical systems. E.g., one can define the set of all strings which are proofs in implicational calculus. The parsing algorithm parses strings over the symbols of the canonic system. If the string is a valid proof, the algorithm generates a tree structure representing the derivation of the proof; otherwise, it indicates that the string is not a proof.

Section 2 defines canonic systems and the restrictions introduced on their form. Section 3 gives results of some mathematical properties of various classes of canonic systems. Section 4 describes the parsing algorithm, which maps any string in the language into the structure of the string. Any string not in the language is mapped into a special symbol 0. Appendix 1 presents a canonic system which describes a context sensitive subset of a computer language. An example of a parse of a string in that language is given. Appendix 2 presents a canonic system which describes proofs in a modified implicational calculus. An example of a parse of a string which is a proof in that calculus is given.

2. Canonic systems

This section talks about canonic systems. The definition of canonic systems given here is similar to Smullyan's definition of elementary formal systems (Smullyan, 1961).

Some terminology is introduced before defining canonic systems. A *string* is a succession of one or more signs. The *empty string* consists of no signs. An *alphabet* is a set of signs. A *string over* an alphabet A is a string that contains only the signs of A. A *language* is a set of strings over an alphabet.

A *Canonic System* C over an alphabet V (terminal characters) is the collection of the following:

1. The alphabet V. The elements of V are the *symbols* of the system.
2. An alphabet of characters called *variables*.
3. An alphabet of characters called *predicates*.
4. An alphabet of characters called *grouping signs*.
5. A finite number of strings which are canons according to the definition given below.

The alphabets 1, 2, 3, and 4 are mutually disjoint. Their elements are the *signs* of the system C.

Canons are strings over the alphabet of signs which specify the rules for the recursive generation of strings over V. Before

*This paper is based on part of a dissertation presented to the Faculty of the Graduate School of Yale University in candidacy for the degree of Doctor of Philosophy.

Downloaded from https://academic.oup.com/comjnl/article/15/3/229/480603 by guest on 19 April 2024

defining what a canon is, some terminology must be introduced.

A *word* is a string of symbols. A *compound word* is either a word or a compound word followed by '&' followed by a word. '&' is a grouping sign that is interpreted as conjunction.

A *term* is a string over the alphabet of symbols and variables. A *compound term* is either a term or a compound term followed by '&' followed by a term.

A *remark* is a term followed by a predicate sign. A *compound remark* is a compound term followed by a predicate sign.

A *premiss* is a string consisting of a single compound remark or several compound remarks separated by the grouping sign '&'. Each such remark is called a *premiss remark*. The empty string is a premiss. A *conclusion* is a non-empty compound remark. A canon is a string of finite length consisting of a premiss followed by the grouping sign '→' followed by a conclusion followed by the grouping sign '⋅⋅'. An *instance of a canon X* in canonic system C is any string obtained from X by substituting strings over V for all variables in X. A canonic system can be viewed as having a possibly infinite number of canon instances represented by a finite number of canons.

Unlike Smullyan's (1961) elementary formal systems and Post's (1943) canonical systems, a canon, as defined above, consists of a conjunction of premiss remarks and a conjunction of conclusion remarks. For a conclusion remark to hold, only those premiss remarks which are relevant to the conclusion remark must be satisfied. A premiss remark, P, is *relevant* to a conclusion remark, R, if there exists a variable that occurs both in P and R, or occurs both in P and another premiss remark relevant to R. Consider, e.g., the canon

$$\alpha P_1 \& \rho P_2 \& \omega P_3 \& \alpha\rho P_4 \rightarrow \omega \& \alpha P_5 \cdot\cdot \, .$$

where α, ρ, and ω are variables. ωP_3 is the only premiss remark relevant to the conclusion remark ωP_5. The premiss remarks which are relevant to the conclusion remark αP_5 are

$$\alpha P_1 \& \rho P_2 \& \alpha\rho P_4 \, .$$

At this point, the variations on Smullyan's and Post's systems can be considered simply as a means for abbreviating several canons into one. Consider a canon N with n remarks in its conclusion. It is an abbreviation for n canons. The correspondence between N and the n canons is as follows: Each remark R in the conclusion of N is a conclusion of a canon whose premiss consists of the remarks in the premiss of N which are relevant to R.

An *axiom* of a canonic system C is a remark R which occurs in the conclusion of a canon N of C such that N has no remarks in its premiss relevant to R. I.e., an axiom is a remark which occurs in the conclusion of an unabbreviated canon that has an empty premiss. A *provable string* of C is any string which is an axiom of C or is derivable from the unabbreviated canons of C by a finite number of applications of a rule of substitution and a modified rule of modus ponens. I.e., an instance of a canon is obtained by substituting words for variables in the canon using the rule of substitution. If all the remarks in the premiss of the instance of the canon are provable, the conclusion in the instance of the canon is also provable by applying the modified rule of modus ponens.

Let P be a predicate in a canonic system C over V. Let S be a set of strings over V. P is said to *represent* S in C iff for every string x over V, the following condition holds:

$$x \in S \longleftrightarrow xP \text{ is provable in } C \, .$$

Let P be a designated predicate in a canonic system C. A language L defined by the canonic system C is the set of strings represented by P in C.

An example of a canonic system that defines the context sensitive language $L = \{a^m b^n a^m b^n | m, n \geq 1\}$ is the following:

$$\alpha P_1 \rightarrow a \& \alpha a P_1 \cdot\cdot \quad (1)$$

$$\alpha P_2 \rightarrow b \& \alpha b P_2 \cdot\cdot \quad (2)$$

$$\alpha P_1 \& \rho P_2 \rightarrow \alpha\rho\alpha\rho P_3 \cdot\cdot \quad (3)$$

where a and b are symbols; P_1, P_2, and P_3 are predicate signs; &→⋅⋅ are grouping signs; and α and ρ are variables.

The set L is represented by the predicate P. E.g., consider the set A represented by predicate P. Canon (1) is an abbreviation for the two canons:

$$\rightarrow aP_1 \cdot\cdot \quad (4)$$

$$\alpha P_1 \rightarrow \alpha a P_1 \cdot\cdot \quad (5)$$

aP_1 is provable because it is an axiom. It follows from no premiss in canon (4). Hence, the string a is in the set A.

Using the substitution rule, one obtains as an instance of canon (5) the following:

$$aP_1 \rightarrow aaP_1 \cdot\cdot$$

where the string a is substituted for the variable α at each occurrence of α. Applying the rule of modus ponens on the above instance and the fact that aP_1 is provable, it follows that aaP_1 is provable.

One can interpret canonic systems as generator systems. A canonic system generates strings. There are different levels of generation of strings. At level 0, no strings are generated and the sets of strings represented by the predicates of the canonic system are empty.

At level 1, the strings generated are the axioms of the system. I.e., a string xP, where x is a word and P is a predicate, is generated at level 1 iff xP is an axiom. The string x is added to the set represented by the predicate P which was obtained at level 0. So, the sets of strings at level 1 are the sets obtained at level 0 updated by the addition of the words of remarks generated at level 1.

At level n, the strings generated are the remarks in the conclusions of instances of canons whose premiss remarks have already been generated at levels 0 through $n - 1$. Again, the sets of strings at level n are the sets of strings obtained at level $n - 1$ updated by the addition of the words of remarks generated at level n.

A canon is said to have *cross reference* if there exists a variable having a multiple occurrence in a conclusion remark of the canon. Canon (3), in the above example, has cross reference because the variables α and ρ have multiple occurrences in the conclusion remark of the canon. The first occurrence of the variable is said to be involved with *forward reference*; the latter occurrences are said to be involved with *back reference*.

For any one instance of canon (3), the same words must be substituted for each occurrence of α. Similarly for each occurrence of ρ. According to canon (3), any string of a's followed by any string of b's followed by the same string of a's followed by the same string of b's is a string in the set represented by the predicate P.

Consider a canon C_1. Let T be a term in the conclusion remark of C_1. Let the first sign in T be a variable α. If α is the term of a premiss remark which has a predicate the same as the predicate being defined by C_1, T is said to be *left recursive*. C_1 is a canon with left recursion. E.g., the canon $\alpha P \rightarrow \alpha b P$ is left recursive.

Canonic systems, in general, are capable of defining any recursively enumerable set. The general problem of deciding whether or not a string belongs to such sets is recursively unsolvable. Restrictions have been introduced on the form of the canons of canonic systems and a hierarchy of classes of canonic systems has been obtained.

Class 1 Canonic Systems are obtained by introducing the following restrictions on the general canonic systems:

Let T_1 be a premiss remark and T_2 a conclusion remark in the same canon such that T_1 and T_2 are relevant. The restrictions (necessary for relevant remarks only) are:

1. The length of T_1 must not exceed the length of T_2.
2. Each variable in T_1 must occur in T_2.
3. The number of occurrences of a variable in T_1 must not

Downloaded from https://academic.oup.com/comjnl/article/15/3/229/480603 by guest on 19 April 2024

exceed the number of occurrences of the same variable in T_2.

Canonic systems of this class are permitted to have cross reference and premiss terms of length greater than one.

Class 2 *Canonic Systems* are obtained by introducing the following restrictions on Class 1 canonic systems: The term of each premiss remark must consist of just one variable. The class of languages generated by such canonic systems remains closed under intersection.

Class 3 *Canonic Systems* are obtained by introducing the following restriction on Class 2 canonic systems:

No two premiss remarks in the same canon have identical terms. In this class we allow cross reference in the conclusion only.

Class 4 *Canonic Systems* are obtained by introducing the following restriction on Class 3 canonic systems:

No variable can have a multiple occurrence in any one term of the conclusion of any canon. The class of languages generated by such canonic systems is equivalent to the class of context free languages.

3. Some mathematical properties of classes of canonic systems

Some theorems about the mathematical properties of the various classes of canonic systems will be stated. The proofs of the theorems will be sketched sufficiently to allow the interested reader to complete the proofs on his own.

Theorem 1
Class 1 $\geq$ Class 2 $>$ Class 3 $>$ Class 4.

Proof
By construction Class 1 $\geq$ Class 2 $\geq$ Class 3 $\geq$ Class 4.

The containments must be shown to be proper among Classes 2, 3 and 4.

The language $L = \{a^m b^n a^m b^n | m, n \geq 1\}$ cannot be generated by any canonic system of Class 4. However, L can be generated by the canonic system on page 230 which is of class 3.

Later on, the emptiness problem is proved to be decidable for Class 3 and undecidable for Class 2, the latter proved by showing that the problem of whether or not the intersection of two Class 2 languages is empty is recursively unsolvable. Class 2 languages are closed under intersection. If Class 3 languages are not closed under intersection, then clearly there are Class 2 languages which are not in Class 3. If Class 3 languages are closed under intersection, then there exist two Class 2 languages whose intersection is not decidable whether empty or not and hence at least one of such two languages is not in Class 3. Therefore, Class 2 properly contains Class 3.

The problem of whether or not Class 1 properly contains Class 2 is an open question.

Theorem 2
The class of languages generated by canonic systems of Class 4 is equivalent to the class of context free (CF) languages (Chomsky's type 2).

Lemma 1
Any language L generated by a canonic system of Class 4 is a context free language. I.e., there exists a context free grammar G that can generate L.

Proof of Lemma 1
Interpret the predicate signs as non-terminal characters V_N, and the symbol signs as terminal symbols V_T. The designated predicate corresponds to the designated non-terminal character S of the CF grammar. A canon of the form $\rightarrow aA\cdot\cdot$ is interpreted as a rule of G of the form $A \rightarrow a$. A canon of the form $\alpha B \& \rho C \rightarrow \alpha a \rho A\cdot\cdot$ is interpreted as a rule of the form $A \rightarrow BaC$. The left-hand side of the rule consists of a single element of V_N. The element is the predicate defined by the canon. The right-hand side of the rule consists of the term of the conclusion remark with each variable replaced by the predicate it is associated with in the premiss of the canon. The above interpretation can be extended to any canon in any canonic system of Class 4. Hence, the set of strings of symbols represented by the designated predicate of the canonic system consists of elements that are S-derived in some CF grammar. Therefore, any language defined by a canonic system of Class 4 is context free.

Lemma 2
Any CF language can be generated by a canonic system of Class 4.

Proof of Lemma 2
The proof is similar to that of Lemma 1.

Theorem 3
Any context sensitive language (Chomsky's type 1) can be generated by a canonic system of Class 1.

Proof
A phrase structure grammar can be viewed as a semi-Thue system with restrictions on the pairs of strings

$$(A_1, B_1), (A_2, B_2), \ldots, (A_n, B_n)$$

used in the production rules

$$PA_iQ \rightarrow PB_iQ \quad i = 1, \ldots, n$$

of the system where P and Q are arbitrary strings. The restrictions are: (1) The A_i must have at least one non-terminal symbol in them. (2) One of the A_i must be the designated symbol S of the phrase structure grammar. The language generated by a phrase structure grammar of this sort is the set of all terminal strings that are equivalent to S.

A phrase structure grammar is context sensitive if for all $i = 1, \ldots, n \quad |A_i| \leq |B_i|$. Any such restricted semi-Thue system can be interpreted as a canonic system of Class 1 in the following way:

The alphabet of symbols of the canonic system contain the terminal and non-terminal symbols of the context sensitive grammars. Let V be a predicate representing the set of all possible strings over the symbols of the canonic system. Let S be the designated non-terminal symbol of the context sensitive grammar or equivalently S be the starting symbol of the semi-Thue system to which the strings in the generated language are equivalent. Let L be the corresponding designated predicate in the canonic system. A rule $PA_iQ \rightarrow PB_iQ$ of the semi-Thue system is interpreted as a canon:

$$\alpha V \& \ \rho V \ \& \ \alpha A \ \rho L \rightarrow \alpha B \ \rho L\cdot\cdot$$

The canon

$$\rightarrow S\,L\cdot\cdot$$

is also included in the canonic system. The language generated by such a canonic system is the set of all theorems which are strings over the terminal symbols. The above canons satisfy the restrictions of Class 1. Any context sensitive grammar can be interpreted as a canonic system as described above. Therefore, any context sensitive language can be generated by a canonic system of Class 1.

Since Class 4 canonic systems and context free grammars are equivalent, the decidability and closure properties of context free grammars are also the properties of Class 4 canonic systems. We mention below some other properties. E.g., Class 3 is closed under the following operations: Union, catenation, Kleene closure, string reversal, intersection with regular sets, and homomorphism. Class 3 is not closed under intersection and complementation. The proofs of the above assertions are simpler and more straightforward than the proofs of compar-

Downloaded from https://academic.oup.com/comjnl/article/15/3/229/480603 by guest on 19 April 2024

able assertions about Chomsky's grammars and hence will be left to the reader. Among the reasons for this simplicity is that canonic systems define relations among terminal strings directly while in Chomsky's grammars there are non-terminal symbols appearing in the intermediary strings.

A derivation is *repetitious* if in the subtree representing the derivation there is a subtree whose root and some node in it represent the same predicate. If *SP* is provable in *C*, then *SP* is a theorem that can be derived. The length of the derivation is equal to the length of the longest path from the root representing *P* to the leaves representing *S*. It can be easily shown that if a derivation in a canonical system of Class 3 is non-repetitious and if *C* has *n* predicates, then the length of the derivation is not greater than *n*. Further, it can be shown that if *C* is a canonic system in Class 3 and L_C (the language generated by *C*) is non-empty, then there exists an *S* in L_C and a non-repetitious derivation of *SP*. Therefore, in a finite number of steps one can decide whether or not the language generated by a canonic system of Class 3 is empty. Also, in a striaghtforward manner one can show that a language *L* generated by a canonic system of Class 3 is infinite iff there exists a string in *L* whose length of derivation *l*, is $n < l < 2n$ where *n* is the number of predicates in the canonic system.

One can show that the problem of deciding whether or not the intersection of two languages generated by canonic systems of Class 2 is empty is recursively unsolvable by reducing the Post correspondence problem to it. Also, one can easily show that the class of languages generated by canonic systems of Class 2 is closed under intersection. Therefore, the emptiness problem for Class 2 canonic systems is undecidable.

4. The parsing algorithm

Some terminology is introduced before describing the parsing algorithm. A string *B* is a *substring* of a string *A* if *B* occurs in *A*. Let *B* and *C* be substrings of *A*. *B precedes C* in *A* if *B* and *C* are disjoint and *B* occurs before *C* in *A*, or if *B* and *C* are not disjoint and *B* is a substring of *C*.

A *structure* of a string *A* is an ordered *n*-tuple *T*. Each coordinate of *T* specifies a set to which a substring of *A* belongs. There is always one coordinate in *T* that specifies the set to which *A* belongs. The ordering on *T* is obtained as follows: Assume the substrings *B* and *C* of *A* belong to the set S_1 and S_2, respectively. If *B* precedes *C*, S_1 corresponds to a coordinate lower than the one corresponding to S_2.

A *parsing algorithm* computes a function that maps any string in a language into the structure of the string. Any string not in the language is mapped into 0.

Languages, as defined in Section 2, can be generated by canonic systems. Consider a canonic system which generates a language *L*. A parsing algorithm is an algorithm that can process a string over the alphabet of *L* and decide based on the canonic system whether or not the string is in *L*. If the string is in *L*, the algorithm generates an array which specifies the tree representation of the structure of the string. A parsing algorithm that can handle canonic systems of Class 1 has been programmed in APL.

A context sensitive language is specified by the canonic system CROSSFEF in Appendix 1. A string

'SV:A = 1 ÷ R; D = A; GOTO SV;.'

in the language is parsed by the algorithm. The generated array which specifies the tree representation of the structure of the string is presented in Appendix 1. A pictorial representation of the structure of the string is depicted in **Fig. 5.**

Strategy of the parsing algorithm

Let *P* be a predicate in a canonic system *C* over *V*, *S* be a set of strings over *V*, and *P* be the predicate representing *S* in *C*. From the condition that a string *X* over *V* is in the set *S* iff *XP*

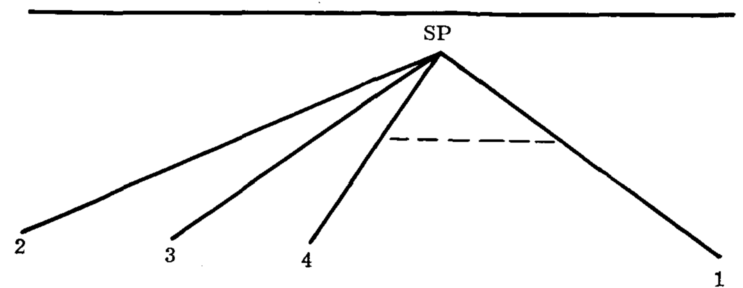


Fig. 1

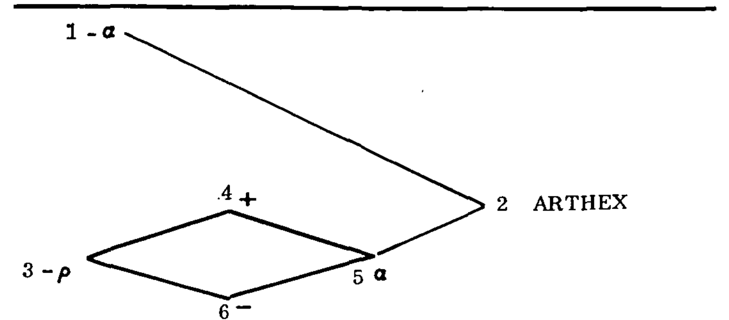


Fig. 2

is provable in *C*, it follows that, checking whether or not a string *X* belongs to a set *S*, is equivalent to checking whether or not *XP* is provable in *C*.

Consider, for example, the canonic system that defines another context sensitive language $L = \{a^n b^n a^n | n \geq 1\}$:

$$\alpha P_1 \rightarrow a\&\alpha aP_1 \quad (1)$$

$$\alpha P_2 \rightarrow b\&\alpha bP_2 \quad (2)$$

$$\alpha P_1 \,\&\rho P_2 \,\&\alpha\rho\alpha P_3 \rightarrow aba\&\alpha a\rho b\alpha aP_3 \quad (3)$$

where *a* and *b* are symbols; α, ω, and ρ are variables; → & are grouping signs; and P_1, P_2, and P_3 are predicates.

The set *L* is represented by P_3. aP_1 is provable because it is an axiom. It follows from no relevant premiss in canon (1). Similarly,

bP_2 is provable (from canon (2))

$abaP_3$ is provable (from canon (3))

A substitution rule generates canon instances from canons. Using the substitution rule, one obtains as an instance of canon (3) the following:

$$aP_1 \,\&\, bP_2 \,\&\, abaP_3 \rightarrow aabbaaP_3\,.$$

All the relevant premiss remarks are satisfied. Therefore, by modus ponens, $aabbaaP_3$ is provable. Canons of the form of canon (3), which has a term in a premiss remark consisting of more than one sign, are said to have a *multi sign premiss term.*

The parsing algorithm presented in this paper reflects the process of proof checking described above. Let *P* be the predicate that represents, in a canonic system *C*, the set of all well-formed programs in some programming language. A string *S* is a well-formed program iff *SP* is provable in *C*.

Suppose the algorithm has to check whether or not a string *S* is a well-formed program. The algorithm sets a pointer to point to the left-most symbol in *S*. It also sets *P* as a goal, i.e., it sets checking whether or not *SP* is provable as a goal. This is represented by node 1 of **Fig.** 1. Each of the nodes 2, 3, . . . represents a conclusion remark in the canon defining *P*.

The algorithm proceeds according to the remark represented by node 2. The first sign in that remark is set as a 'subgoal'. If the sign is a symbol and that symbol is the same as the symbol pointed to in *S*, we say that the subgoal is *recognised.* Then, the next sign in the remark is set as a subgoal. If the sign is a syntactical variable, the predicate of the premiss remark which has that variable as its term is set as a subgoal. The predicate represented by the subgoal is defined by a canon and

Downloaded from https://academic.oup.com/comjnl/article/15/3/229/480603 by guest on 19 April 2024

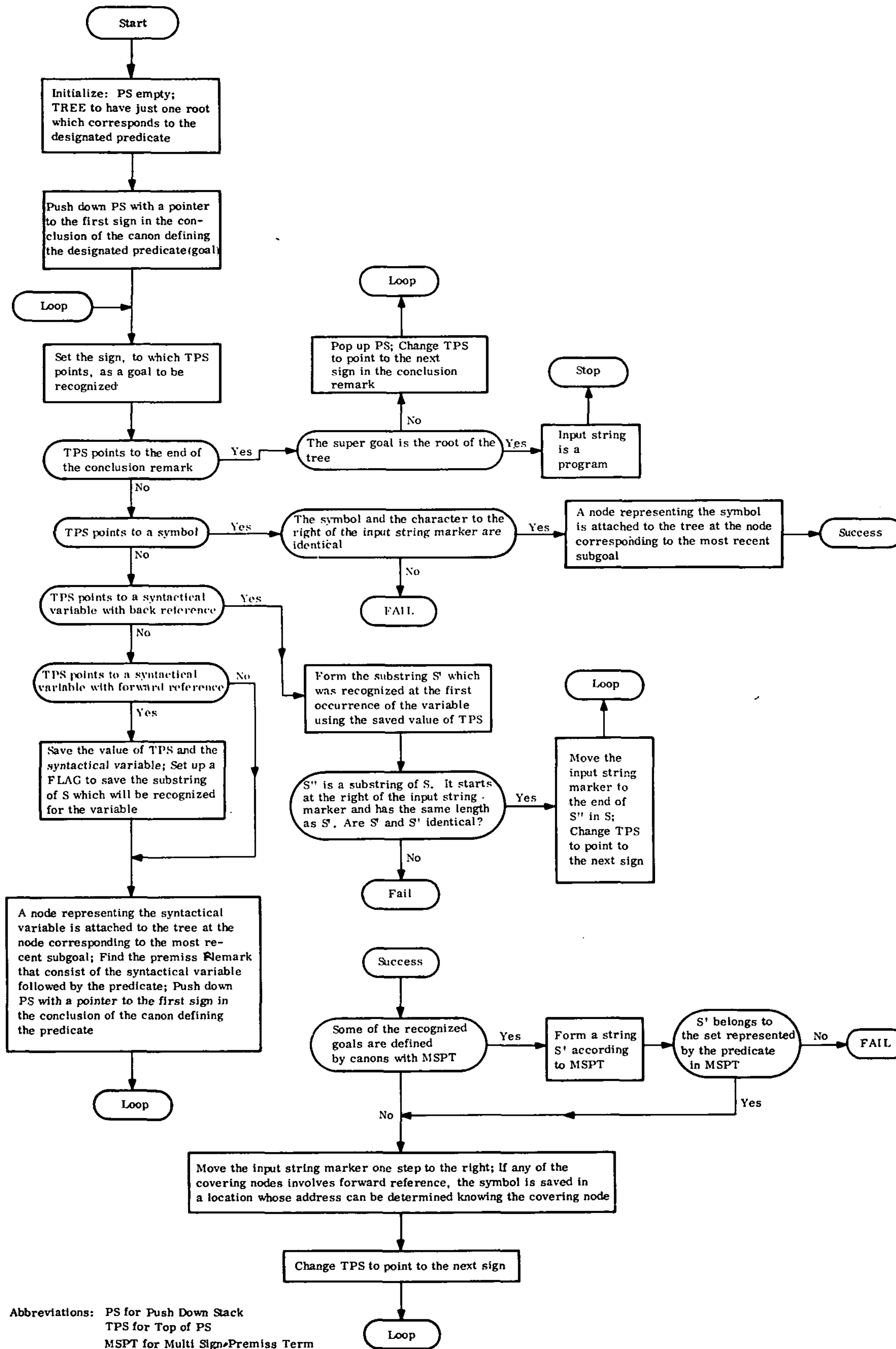


Fig. 3

the procedure to recognise it is the same as for recognising P. Therefore, the algorithm may set up subgoals of subgoals .... If a subgoal is successfully recognised, the algorithm steps to the next sign in the conclusion remark. *Successful recognition* of a subgoal is achieved if all signs in the conclusion remark of the canon defining the subgoal are recognised. In this scheme, left recursion poses a special problem which is treated in detail later on in the paper.

If a subgoal, representing the first sign of a conclusion remark, is not recognised, the algorithm tries an alternate remark in the conclusion of the same canon. If all such remarks have been tried without success, the algorithm backs up to the last sign successfully recognised and tries an alternative. Accompanying this back-up is a corresponding back-up of the pointer associated with the string S. If no alternatives exist and backing-up is not possible, then the string S is not a well-formed program.

Downloaded from https://academic.oup.com/comjnl/article/15/3/229/480603 by guest on 19 April 2024

If a variable, α, occurs more than once in a term of a conclusion remark, α is involved with cross reference. In the process of recognising the term, it is not sufficient to recognise a substring of the input string as of the predicate type represented by α. In the tree structure of string S, the leaves of the vertex representing α at one occurrence must be the same as those of the vertex representing α at any other occurrence in the term. Therefore, when a variable is involved with forward reference, the leaves of the vertex representing the variable are saved in a location whose address can be determined at the later occurrences of the variable.

If the recognised predicate is defined by a canon that has multi-sign premiss term, the algorithm has to perform one further check before reporting successful recognition of the predicate. The algorithm constructs the string represented by the multi-sign premiss term. E.g., suppose the canon has a premiss remark

$$a_x a_y a_z P_n$$

and S_x, S_y, and S_z are the substrings of the input string S that correspond to a_x, a_y, and a_z, respectively. The algorithm sets as a goal the checking of whether or not $S_x S_y S_z P_n$ is provable. The algorithm calls itself recursively. When operating at inner levels, the algorithm returns only a value of 0 or 1; i.e., it acts only as a *recogniser*.

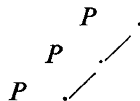
Major features of the parsing program

So far the strategy of the algorithm has been described. The flowcharts shown in **Fig. 3** and **Fig. 4** depict, in general, the sequence of events in the program. The canonic systems are mapped into a matrix before being used by the program.

Left recursion poses a problem for top-to-bottom parsing algorithms. E.g., consider the canon

$$\alpha P \rightarrow a\&\alpha bP \ .$$

Suppose it is required to check whether or not the string ab belongs to the set represented by the predicate P. A top-to-bottom parsing algorithm sets P as a goal to recognise. According to the second term in the conclusion, the algorithm sets P as a subgoal. Again, according to the same term, the algorithm sets P as a subgoal of the subgoal. This process is repeated indefinitely.



To handle left recursion the terms in the conclusion are arranged to have the term with the left recursion at the end of the canon. The algorithm does not try to parse the input string according to the left recursive terms unless a previous term has been successfully recognised. If a previous term has been successfully recognised, the algorithm tries to recognise a longer substring of the input according to the term with left recursion. The first sign in the left recursive term is already recognised. The second sign in the left recursive term is set as a goal to be recognised. The algorithm keeps repeating this process as long as the algorithm is recognising successfully the predicate defined by the left recursive canon. A modified version of the 'Harvard Shaper' mechanism (Kuno, 1965) may be used as an alternative method for handling left recursion.

Consider a compound remark which has more than one term. Suppose some of the terms have similar substrings. The compound remark can be abbreviated to eliminate repetition of identical substrings. This kind of abbreviation is called factoring. The grouping signs '\ /' are used like parenthesis to specify factoring. E.g., the compound remark

$$a_1 a_2 b_2 a_3 \ \& \ b_1 a_2 b_2 a_3 \ \& \ a_1 a_2 a_3 \ \& \ b_1 a_2 a_3 P$$

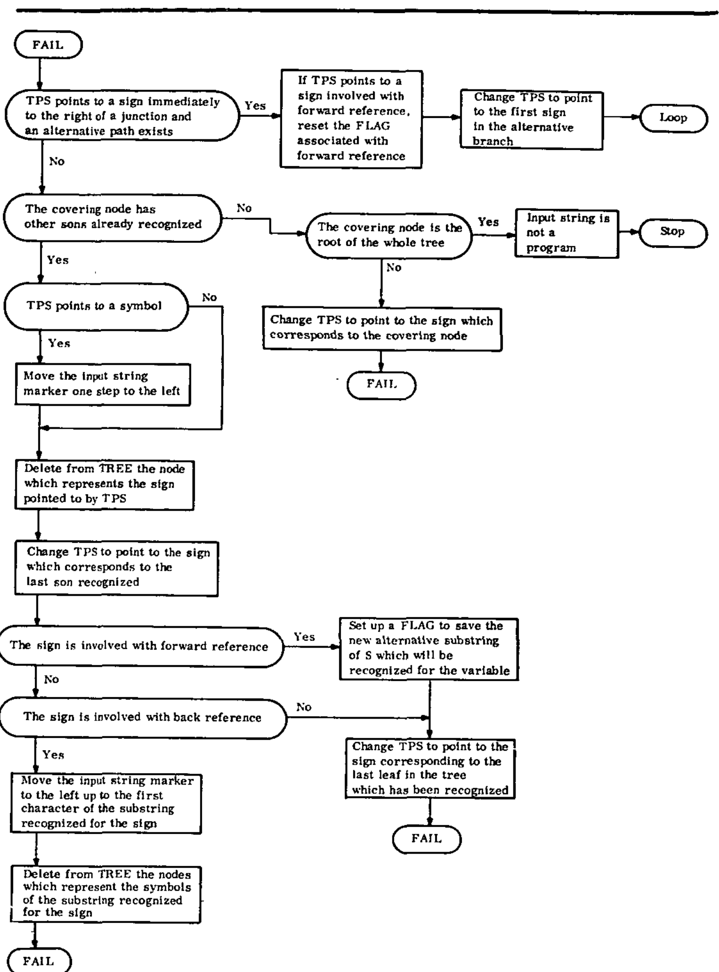


Fig. 4

can be abbreviated to

$$\backslash a_1 \ \& \ b_1 / a_2 \backslash b_2 \& / a_3 P \ .$$

Factoring makes canonic specifications more concise. It speeds up the parsing process considerably. It permits a canon to have more than one left recursive term in its conclusion.

To further facilitate the parsing, each variable must be defined in such a way as to permit the determination of the substring it represents. Therefore, we add the following requirement to Class 1 canonic systems: Any variable that appears in the conclusion of a canon must be defined explicitly in the premiss of the canon. I.e., the variable must be the term of some premiss remark in the canon. This requirement does not, in any way, restrict Class 1 canonic systems since it can be met always by having the predicate in the premiss remark represent the set of all strings over the alphabet of the canonic system.

A canon may have a compound remark as a conclusion. The sequence of steps taken by the parsing algorithm through the conclusion is not necessarily linear from left to right. Consider, e.g., the canon

$$\alpha term\&parthex \rightarrow \alpha\&\rho\backslash + \& - /\alpha arthex \ .$$

The conclusion has the following remarks:

$$\alpha \ arthex, \ \rho + \alpha \ arthex, \text{ and } \rho - \alpha \ arthex \ .$$

The algorithm follows the sequence depicted in the special directed graph shown in **Fig. 2.**

A directed graph is *special* if there is an order on some of its edges. A vertex which is an initial point of two or more edges is called a *junction*. E.g., in Fig. 2, vertex 3 is a junction. The edges coming out of a junction are ordered according to the occurrence, from left to right, of their terminal vertices in the conclusion of the canon. E.g., the order on the edges coming out of junction 3 in Fig. 2 is 34 then 36. There is always a virtual vertex to the left of the graph.

Downloaded from https://academic.oup.com/comjnl/article/15/3/229/480603 by guest on 19 April 2024

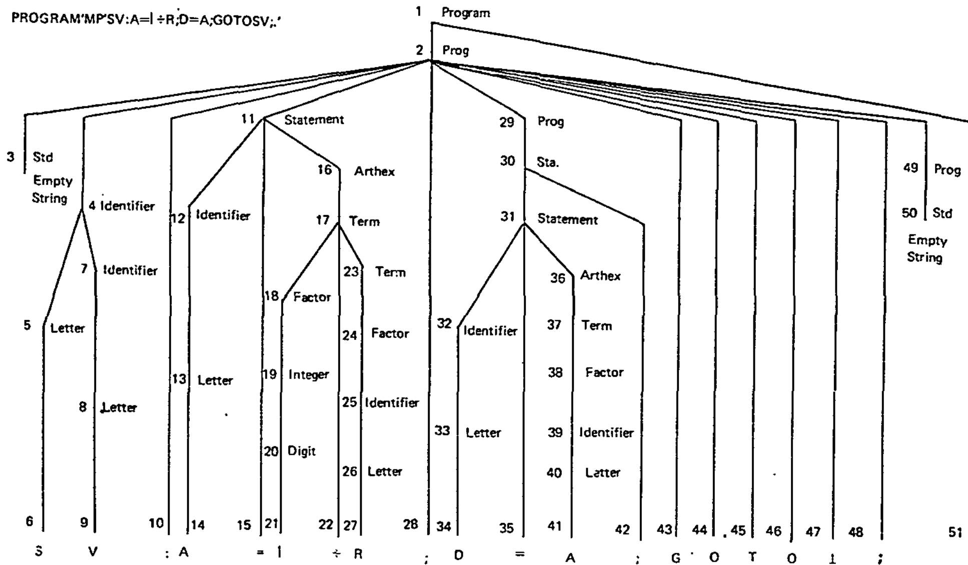


Fig. 5

P is a vector for stepping through special directed graphs. Each element of *P* corresponds to a canon defining a predicate which is set as a goal to be recognised. The value of an element of *P* is the positional number of the vertex to which *P* points in the corresponding graph.

A tree is a directed graph which contains no cyclic paths and which has at most one branch entering each vertex. A vertex *i* which branches to a vertex *j* is called the father (or the covering vertex) of vertex *j*. Vertex *j* is called the son of vertex *i*. The son vector of a vertex *j* is an ordered set of all sons to *j*. A well ordered tree *T* is a tree such that the set of all sons of any vertex of *T* is ordered.

The trees generated by the parsing algorithm are well ordered. A well-ordered tree will simply be referred to as a tree.

A tree *T* is completely defined by specifying, for each vertex *a* of *T*, the son vector of *a*. The above specification is given in the form of a two-dimensional array. The vertices of a tree are labelled. The *i*th row of the two-dimensional array specifies the son vector of vertex *i*. The order on the sons of vertex *i* is the same as the order on the elements of the *i*th row of the array.

However, to make the array specification of the tree more legible, additional information is provided. The array is augmented by three columns at the left.

The first column is an index column for the array. The entry in the first column at the *i*th row is *i*, the label of the vertex whose son vector is specified in the *i*th row. Each vertex in the tree represents a predicate or a symbol. If the vertex is a leaf, it represents a symbol. Otherwise, it represents a predicate. The second column consists of strings of literals. The entry in the second column at the *i*th row is the predicate or symbol represented by the *i*th vertex. The third column provides

CROSSREF*

```
α⎕PROG⎕→α.⎕PROGRAM⎕¨

α⌊ρ⎕PROG⌈ω⎕STATEMENT⎕⌈⊥⎕IDENTIFIER⎕⌈⊤⎕STA⎕→⊤\⊥:GOTO⊥;α
    ⌊GOTO⊥;α⊥:ω;ρ⌊⊥:ω;αGOTO⊥;ρ⌊/ ⎕PROG⎕¨

α⎕STATEMENT⎕⌈ρ⎕STA⎕→α;\ρ⌊/⎕STA⎕¨

α⎕IDENTIFIER⎕⌈ρ⎕ARTHEX⎕→α=ρ⎕STATEMENT⎕¨

α⎕TERM⎕⌈ρ⎕ARTHEX⎕→α\\+⌊-/ρ⌊/⎕ARHTEX⎕¨

α⎕FACTOR⎕⌈ρ⎕TERM⎕→α\\×⌊÷/ρ⌊/⎕TERM⎕¨

α⎕IDENTIFIER⎕⌈ρ⎕INTEGER⎕⌈ω⎕ARTHEX⎕→α⌊ρ⌊(ω)⎕FACTOR⎕¨

α⎕LETTER⎕⌈ρ⎕IDENTIFIER⎕→ α\ρ⌊/⎕IDENTIFIER⎕¨

α⎕DIGIT⎕⌈ρ⎕INTEGER⎕→α\ρ⌊/⎕INTEGER⎕¨

A⌊B⌊C⌊D⌊E⌊F⌊G⌊H⌊I⌊J⌊K⌊L⌊M⌊N⌊O⌊P⌊Q⌊R⌊S⌊T⌊U⌊V⌊W⌊X⌊Y⌊Z⎕LETTER⎕¨

1⌊2⌊3⌊4⌊4⌊6⌊7⌊8⌊9⌊0⎕DIGIT⎕¨
```

See Appendix 1 on next page.

Downloaded from https://academic.oup.com/comjnl/article/15/3/229/480603 by guest on 19 April 2024

information about the fathers of vertices. The entry in the third column at the *i*th row is the vertex label of the father of the *i*th vertex.

If the recognised variable is involved with back reference, its son vector will not be specified explicitly in the array. The row in the array corresponding to such variables consists of four elements. The first element is the column index of the row. The second element is the label of the vertex representing the first occurrence of the variable. The third element is the variable itself. The fourth element is the vertex label of the covering node.

Again, consider the example in Appendix 1. The parsing algorithm set PROGRAM as a goal to be recognised. This in turn, according to the first canon of the canonic system, set PROG as a subgoal. PROG is recognised according to the remark ⊤⊥:ω;αGOTO⊥;ρ□PROG□ in the conclusion of the second canon. At the first occurrence of the variable ⊥, SV is recognised as the identifier represented by ⊥. Since ⊥ occurs later in the same remark, SV is saved. At the second occurrence of ⊥, the algorithm does not only recognise an identifier but also checks that it is the same identifier recognised at the earlier occurrence of ⊥.

Appendix 1

A context sensitive language, CROSSREF, is defined as a canonic system. The language cannot be defined by a context free grammar. To any GOTO statement in the program there must exist a corresponding statement having a label the same as the transfer label in the GOTO statement. Otherwise, the GOTO statement is not legal (see page 235).

where

A . . . Z 0 . . . 9 . : ; = + − × ÷ ( ) are symbols.

α ⊥ ⊤ ω ρ are variables.

⌈ ⌊ \ / → ¨ are grouping signs*. The sign ¨ indicates the end of a canon. The □ sign is used to indicate the predicate signs. In this system any string over the alphabet delimited by two □ signs is a predicate. Because the sign & is not available on the APL terminal, we used the sign ⌊ to represent conjunction between terms and the sign ⌈ to represent conjunction between remarks.

As an example, a string in the language is parsed. *MP* is a dyadic function which is executed by the parsing algorithm. The first argument of the function is the name of the set which we want to check whether or not it contains the string. The second argument is the string.

```
      'PROGRAM' MP 'SV:A=1÷R;D=A;GOTOSV;.'

1   PROGRAM     0  2  51
2   PROG        1  3  4  10  11  28  29  43  44  45  46  47  48  49
3   STA         2
4   IDENTIFI    2  5  7
5   LETTER      4  6
6   S           5
7   IDENTIFI    4  8
8   LETTER      7  9
9   V           8
10  :           2
11  STATEMEN    2  12  15  16
12  IDENTIFI   11  13
13  LETTER     12  14
14  A          13
15  =          11
16  ARTHEX     11  17
17  TERM       16  18  22  23
18  FACTOR     17  19
19  INTEGER    18  20
20  DIGIT      19  21
21  1          20
22  ÷          17
23  TERM       17  24
24  FACTOR     23  25
25  IDENTIFI   24  26
26  LETTER     25  27
27  R          26
28  ;           2
29  PROG        2  30
30  STA        29  31  42

31  STATEMEN   30  32  35  36
32  IDENTIFI   31  33
33  LETTER     32  34
34  D          33
35  =          31
36  ARTHEX     31  37
37  TERM       36  38
38  FACTOR     37  39
39  IDENTIFI   38  40
40  LETTER     39  41
41  A          40
42  ;          30
43  G           2
44  O           2
45  T           2
46  O           2
47  412
48  ;           2
49  PROG        2  50
50  STA        49
51  .           1
```

The above array represents the structure of the string 'SV:A=1÷R;D=A;GOTOSV;.'

Downloaded from https://academic.oup.com/comjnl/article/15/3/229/480603 by guest on 19 April 2024

Appendix 2

Proof in a modified implicational calculus is defined as a canonic system. The definition is a restricted version of More's (1965) canonic system defining implicational calculus.

*See page 234 for explanation of the ╲ ⫽ notation.

Proof

```
α□TERM□→*⌊*α□TERM□¨
α□TERM□⌈ρ⌊ω□FORMULA□→(ρ⊃ω)⌊α□FORMULA□¨
α□VAN□⌈ρ□FORMULA□→ρ,α⌊□VAN□¨
α⌊ρ⌊Δ⌊∇□VAN□⌈⊥⌊⊤□FORMULA□⌈Δ⊤□BASICLAW□⌈αΔρ-∇.⊥.□PROOF□→αΔρ-∇⊥,.⊤.□PROOF1□¨
α⌊ρ⌊Δ⌊∇□VAN□⌈⊥⌊⊤□FORMULA□⌈ρ⊤□BASICLAW□⌈α-Δρ∇.⊥.□PROOF□→α-Δρ∇⊥,.⊤.□PROOF2□¨
α⌊ρ⌊ω□VAN□⌈⊥□FORMULA□⌈α⊥□BASICLAW□→ραω-.⊥.□PROOF3□¨
α⌊ρ⌊Δ□VAN□⌈⊥⌊⊤□FORMULA□⌈ρ⊥,⊤□BASICLAW□⌈α-Δρ.⊥.□PROOF□→α-Δρ⊥,.⊤.□PROOF4□¨
α⌊ρ⌊ω□FORMULA□→(α⊃ρ),α,ρ⌊(α⊃(ρ⊃ω)),((α⊃ρ)⊃(α⊃ω))⌊ρ,(α⊃ρ)□BASICLAW□¨
α□PROOF1□⌈ρ□PROOF2□⌈ω□PROOF3□⌈Δ□PROOF4□⌈∇□VAN□→∇-..⌊ω⌊α⌊ρ⌊Δ□PROOF□¨
```

where

$*$, – . ( ) ⊃ are symbols.

$\alpha\ \rho\ \omega\ \nabla\ \Delta\ \perp\ \top$ are variables.

⌈ ⌊ → ¨ are grouping signs.

Predicate signs are delimited by □.

As an example, a string which is a proof in the modified implicational calculus is parsed.

'PROOF' MP '(** ⊃ *), **, *–, · (*** ⊃ *).'

1	*PROOF*	0 2
2	*PROOF4*	1 3 25 26 27 28 31 32 33 47
3	*VAN*	2 4 16 17
4	*FORMULA*	3 5 6 11 12 15
5	(	4
6	*FORMULA*	4 7
7	*TERM*	6 8 9
8	*	7
9	*TERM*	7 10
10	*	9
11	⊃	4
12	*FORMULA*	4 13
13	*TERM*	12 14
14	*	13
15	)	4
16	,	3
17	*VAN*	3 18 23 24
18	*FORMULA*	17 19
19	*TERM*	18 20 21
20	*	19
21	*TERM*	19 22
22	*	21
23	,	17
24	*VAN*	17
25	–	2
26	*VAN*	2
27	*VAN*	2
28	*FORMULA*	2 29
29	*TERM*	28 30
30	*	29
31	,	2
32	.	2
33	*FORMULA*	2 34 35 42 43 46
34	(	33
35	*FORMULA*	33 36
36	*TERM*	35 37 38
37	*	36
38	*TERM*	36 39 40
39	*	38
40	*TERM*	38 41
41	*	40
42	⊃	33
43	*FORMULA*	33 44
44	*TERM*	43 45
45	*	44
46	)	33
47	.	2

The above array represents the structure of the proof of '(** ⊃ *), **, –*,· (*** ⊃ *).'

Acknowledgement

The author wishes to thank T. More, Jr. for his many valuable suggestions and contributions.

References

Cheatham, T. E., Jr., and Sattley, K. (1964). Syntax-Directed Compiling, *AFIPS Proc. Spring Joint Computer Conference*, Spartan Books, Baltimore, Maryland, Vol. 25, pp. 31-57.

Chomsky, N. (1959). On Certain Formal Properties of Grammars, *Information and Control*, Vol. 2, pp. 137-167.

Donovan, J. J. (1967). Investigations in Simulation and Simulation Languages, Ph.D. Dissertation, Yale University, New Haven, Connecticut.

Feldman, J., and Gries, D. (1968). Translator Writing Systems, *Comm. ACM*, Vol. 11, No. 2, pp. 77-113.

Ghandour, Z. J. (1968). Formal Systems and Syntactical Analysis, Ph.D. Dissertation, Yale University, New Haven, Connecticut.

Kuno, S. (1965). The Predictive Analyzer and a Path Elimination Technique, *Comm. ACM*, Vol. 8, No. 7, pp. 453-462.

Landweber, P. S. (1964). Decision Problems of Phrase-Structure Grammars, *IEEE Trans. on Electronic Computers*, Vol. EC-13, No. 4, pp. 354-362.

More, T., Jr. (1965). Yale Class Notes on Applied Discrete Mathematics, Spring and Fall, 1965.

Post, E. L. (1943). Formal Reductions of the General Combinatorial Decision Problem, *American Journal of Mathematics*, Vol. 65, pp. 197-215.

Smullyan, R. M. (1961). *Theory of Formal Systems*, Princeton University Press, Princeton.

Downloaded from https://academic.oup.com/comjnl/article/15/3/229/480603 by guest on 19 April 2024