26

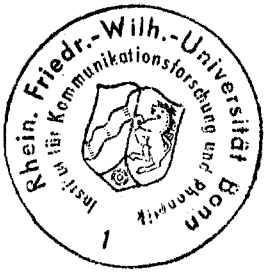
1965 International Conference on Computational Linguistics

SPECIFICATIONS FOR A TREE PROCESSING LANGUAGE

by

R. Tabory, F. L. Zarnfaller

International Business Machines Corporation
Thomas J. Watson Research Center
Yorktown Heights, New York



SPECIFICATIONS FOR A TREE PROCESSING LANGUAGE

by

R. Tabory

F. L. Zarnfaller

International Business Machines Corporation
Thomas J. Watson Research Center
Yorktown Heights, New York

ABSTRACT: Description of trees and strings - both composed of lexical values - matching of trees and strings against partially or totally defined pattern and operations on trees and strings form the essential part of the Processor. The notion of string has been extended to strings embedded in or extracted from trees, and a modified version of the COMIT rewriting rule applied to these strings. Variables ranging over lexical values, strings and trees can be introduced and operated on according to a "pushdown" principle. Besides, variables can be operated either in the "connected" or "autonomous" mode, depending whether their connections with some parent structure are remembered or not. Variable value structures can be matched against patterns or compared among themselves. Transformations on trees and strings are defined, allowing for the development of a given data structure into a new one. All these features and devices were defined by extracting the elementary steps with which linguists compose their operations and by generalizing these steps reasonably. The resulting language specifications are proposed for discussing the solution of a class of non-numerical programming problems.

Introduction

Recent developments in computational linguistics have shown the need for a convenient programming formalism enabling linguists to handle data processing problems of natural languages. Especially tree and an extended kind of string processing have to be made conceptually easy for the linguistically oriented computer user. This paper proposes specifications for such a language; it should be thought of - theoretically only, since its implementation is not planned presently- as a procedure package attached to NPL (New Programming Language).

The reason for this is the desire to take full advantage of the highlights of a modern higher level programming language as well as of the devices any general purpose programming system is in possession of. Consequently, the operations described in this paper should be thought of as being complemented by the full power of NPL, the most interesting features of which - from the viewpoint of non-numerical processing - are: recursive procedure calls, dynamic storage allocation; numeric, character string and bit string data arranged in structures and multi-dimensional arrays, etc.

Description of trees and strings, both composed of lexical values, and operations on them form the essential part of the Processor. The word "processor" covers both a programming language, and a data organizing system in the computer memory. As far as string descriptions and string processing is concerned, the influence of COMIT was dominant in elaborating the relevant part of the Processor, but the notion of string has been extended to strings embedded in or extracted from trees. Other existing programming languages and linguistic programming systems were also taken into consideration, they are

all mentioned in the Reference List.

The definition of the features of the Processor has been accomplished by extracting the elementary steps with which linguists compose their higher level syntactic operations and by a reasonable generalization of these steps. Some of these devices are, however, fairly general and any kind of tree processing, not only a syntactically oriented one, would introduce them.

Data and Variables

There are three basic kinds of data the Processor can handle: lexical values, strings and trees. A lexical value is a character string; a string (sometimes referred to as "lexical" string, as opposed to a character string) is an ordered sequence of lexical values separated by some conventionally chosen delimiter; a tree is a partially ordered set of lexical values, where each element has one and only one ancestor, except the "root" of the tree which has no ancestor at all. Besides, elements sharing the same ancestor are called siblings and such a sibling set is totally ordered. Thus, each element of the tree has at most one "left" and at most one "right" sibling.

Lexical values can be numerically or logically tagged. In order to avoid multiple tagging a tag is a fixed length decimal integer, each digit of which can be processed separately. Besides a digit can be logically interpreted as True or False, according to whether it is non-zero or zero.

Variables of the form Ln, Sn and Tn - where n is an integer freely chosen by the user - take, respectively, lexical values, strings and trees as values. We shall see that these variables have value lists sometimes, instead of a single value.

It results from the above described data definitions that data can be embedded in other data; a lexical value can be a string constituent or a tree element, a string can be a portion of a broader string or extracted from a tree, a tree can be a subtree of a larger tree. In order to handle this property of the data, a variable can be operated in two modes: the autonomous or the connected mode. At

each instant all variables introduced have a well defined "autonomy status" determining the mode in which each variable is operated on. This status is the same for all values of a variable. The meaning of the two modes is the following:

In the connected mode all values of a variable maintain their connections with the larger data structure to which they belong. In other words a connected variable "remembers" its connections with some parent structure and this feature can be used for further processing of that parent structure. Moreover, these connections are automatically updated when the parent structure undergoes a modification. The kind of updating depends on the way the particular value was assigned to the variable and on the modification occurring in the parent structure.

In the autonomous mode variables don't have connections at all. They refer to data that are independent of other data, except data embedded in the autonomous structures in question. Variables carrying input data into the computer are made autonomous by the Processor. Variables introduced in various statements (except input statements) are set to the connected mode; the statement

CREATE variable

creates an autonomous variable with an empty value list. In order to change the status of a variable the statement

AUTONOMY variable

can be used. Upon execution of this statement data referred to by the variable are copied down in some part of the storage and the variable is made autonomous. To change the status in the other direction, from autonomous to connected, some later described

transformation statements have to be used. Generally, some statements may change the autonomy status of a variable. These changes will be described with these commands.

It is possible to check the autonomy status of a variable. The predicate

B AUTONOMY variable

performs this function. B stands for a logical variable (variable ranging over bit strings of length one). B is set to True if the argument variable is autonomous, to False otherwise.

Value Assignment Statements

In the forthcoming chapters statements will be described which assign values to variables. If the variable to which values are assigned has already a non-empty value list the new values assigned to this variable will be added to the list, at the beginning of it, in a "push-down" manner.

Value assignment statements use also variables sometimes, as arguments of the statement. In such a case the first item on the value list of the argument variable is taken into consideration, and referred to hereafter as "the value" of the variable, though other values may follow it on the list. If the value list is empty, the value assignment statement amounts to a non-operation.

Parent referencing is another problem that arises naturally in value assignment statements, that is to say the problem of naming the data where from the variable will get values assigned, according to the specifications of the statement. Two cases exist:

(a) The parent reference is mandatory in the value assignment

command. This is the case when the statement has no sense at all without a parent reference.

(b) The parent reference is optional. In this case the following rules prevail: if it is mentioned, the value assigned is entirely contained in the specified parent reference, or else (if no suitable values are found within the reference) the assignment statement is a non-operation. If it is not mentioned, values will be collected from the largest autonomous data to which the argument variable value of the value assignment statement is connected.

Parent References are of the form (in Vn), where Vn is a variable name (Tree or String). In the forthcoming chapters they will be mentioned, in statement descriptions, as "Optional Parent Reference" or "Mandatory Parent Reference". The word "parent" might be replaced by "string", or "tree", etc. when a restrictive condition of this nature prevails, according to the nature of the statement.

Value Assignment to Lexical Variables

Assigning a value to a lexical variable can take place, in principle, in three possible ways:

(1) Absolute Assignment - The lexical value collected from a tree or a string is designated by its "absolute location" in the parent structure, i.e., its Iverson's index vector in the tree case, or its constituent serial number in the string case. (See below for detailed explanation.)

(2) Associative Assignment - This kind of value assignment takes place by naming that lexical value in the parent tree or string, which the user desires to assign as variable value. In this case

the variable may be supplied with a value list, instead of a single value. This happens when the specified lexical value occurs more than once in the parent structure. The value list will then contain entries having identical lexical values but each of these entries having different connections with the parent structure.

(3) Relative Assignment - This type of assignment uses another variable, as argument, and determines the value to be assigned in function of the value of the argument variable.

We are going now into the details of value assignment to lexical variables. From the autonomy status point of view the following rules prevail (also valid for value assignment to string and tree variables): if the variable to which value is assigned is currently operated in the connected mode, connections with the parent structure will be conserved in the usual way. If the variable is autonomous, these connections are -lost and only lexical values (without their "position information") and possible tags are transmitted.

In order to enable the user to assign an arbitrary lexical value to a variable (not necessarily collected from a tree or a string), the following statement is permissible:

Ln = LEXICAL lexical / m

The lexical value "lexical" is assigned to Ln. m is an optional integer representing a tag. Words written in upper case letters represent - throughout this paper - statement keywords.

Absolute value assignments to lexical variables are of the form:

(1) Ln = (m_1, m_2, ... m_k) Mandatory Tree Reference

(2) Ln = LEFT k Mandatory String Reference

(3) Ln = RIGHT k " " "

In the tree case, (m_1, m_2, . . . m_k) denotes an Iverson's index vector; each m_i is an integer, m_1 = 1 and it denotes the root of the tree, m_2 selects the m_2 -th sibling among the immediate successors of the root, m_3 selects the m_3 -th sibling among the immediate successors of the tree element selected by the previous portion of the index vector, and so on, until m_k selects a final tree element that will become the value to be assigned to Ln.

In the string case, k is an integer and LEFT k of RIGHT k assigns, respectively, the k -th constituent from the left or from the right end of the string, as value to Ln.

Associative value assignments to lexical variables are of the form:

Lm = lexical / tag Mandatory Parent Reference

In the right side of the statement "lexical" denotes a lexical value and "tag" the description of a tag. One of the two descriptions "lexical" and "tag" might be omitted. The parent structure is searched for all elements or constituents that match the specified lexical value and/or the specified tag. The selected elements are added to the value list of Ln. A tag specification is of the form: a, a_2, . . . a_n where n is the number of digits in the tag; each a_i is either an integer or an integer variable specifying the required digit in the i-th position, or one of the signs:

(1) * meaning any integer in that position

(* - k_1, k_2, . . . k_m) " " " " " " except the enumerated ones (each k_i is an integer)

$(k_1 / k_2 / \ldots / k_m)$ meaning one of the enumerated integers.

Relative value assignments to lexical variables are of the form:

Ln = NEIGHBOR Lm Optional Parent Reference

where NEIGHBOR stands for one of the following keywords: ANCESTOR, L-SIBLING, R-SIBLING, L-SUCCESSOR, R-SUCCESSOR. The interpretation of these keywords depends on the Parent Reference, whether it is explicitly stated or not. If the Parent Structure is a tree, the interpretation is the following:

(1) ANCESTOR: the nearest tree element to which the value of Lm is connected in the direction of the root of the tree.

(2) L-SIBLING: the element preceding the value of Lm, in its sibling set.

(3) R-SIBLING: the element following the value of Lm in the sibling set.

(4) L-SUCCESSOR: the first element in the sibling set whose ancestor is the value of Lm.

(5) R-SUCCESSOR: the last element in the sibling set whose ancestor is the value of Lm.

If the Parent Structure is a string, L-SIBLING and L-SUCCESSOR both denote the constituent to the left of the value of Lm, while R-SIBLING and R-SUCCESSOR denote the string constituent to its right. ANCESTOR denotes the leftmost constituent in the string.

In all these statements the argument variable Lm has to be currently operated in the connected mode. The selected value, depending on which keyword is used and what the parent structure is, is

assigned to the value list of Ln.

In obvious cases the "empty" value might be assigned to Ln.

Value Assignment to String Variables

String value assignments are also absolute, associative or relative. The autonomy rules are analogous to the ones used in value assignments to lexical variables, except if stated otherwise. In order to enable the user to assign an arbitrary value to a string variable - not necessarily collected from a parent structure - the following statement is permissible:

Sn = CONCATENATE AAA . . . AA

where each A stands for one of the following possibilities:

(1) A lexical value with an optional tag (of the form lexical / tag)

(2) a lexical variable

(3) a string variable.

The statement performs the concatenation in the order shown in the statement body, by taking the first items on the value lists of the variables involved. Parent connections of these values are not taken into consideration and Sn is made automatically autonomous (if it wasn't already), its value is the concatenated set of lexical values, with possible tags.

Absolute value assignment to string variables is of the form:

Sn = LEVEL k Mandatory Tree Reference

k is an integer selecting the k-th level of the tree referred to. The k-th level of a tree is the string whose constituents are at distance k from the root, the distance being measured in number of subsequent ancestors up to the root.

Lm being a connected variable, relative assignment to string variables takes place with the help of a statement of the form:

Sn = WORD Lm Optional Parent Reference

WORD stands for one of the keywords enumerated below; the interpretation of these keywords depends on the Parent Structure (explicite or implied) being a tree or a string. The table below enumerates all the cases:

Keyword	Tree Case	String Case
ANCESTOR	The ancestor string of the argument element is the one that connects it to the root of the tree.	The longest string in which the argument is a constituent.
SUCCESSOR	The successor string of the argument element is the sibling set having the argument as ancestor element.	Same as ANCESTOR
L-SUCCESSOR	The L-successor string is the one composed of the sequence of subsequent L-SUCCESSOR elements in the downward direction.	The left portion of the string upto the argument constituent.
R-SUCCESSOR	Same definition as above, but taking the subsequent R-SUCCESSOR elements.	The right portion of the string from the argument constituent.
SIBLING	The sibling string of the argument is the successor string of its ancestor element.	same as ANCESTOR
L-SIBLING	The part of the sibling string preceding the argument.	Same as L-SUCCESSOR

Keyword	Tree Case	String Case
R-SIBLING	The part of the sibling string following the argument.	Same as R-SUCCESSOR
TERMINAL	The terminal string of the argument element is composed of elements having empty successor strings and on whose ancestor strings the argument element is a constituent.	Same as ANCESTOR

All these statements assign the selected string as value to Sn.

Associative assignment to string variables takes place with a statement of the form Sn = STRING (k_1, k_2 string structure specification) Mandatory String Reference. String structure specification is a string over the alphabet of lexical values and a few special signs; - we are going to call it a metastring. It describes the internal composition of the Parent Reference. k_1 and k_2 stand for integers or integer variables denoting the k_1 -th and the k_2 -th constituent of the metastring. The portion of the metastring falling between these constituents delimits a portion of the parent string. Upon execution of the statement this portion of the Parent String is attributed as value to Sn. In case of more than one portion referred to in the parent string, the leftmost one is taken into consideration. In case of "no-match" Sn gets the empty value.

The metastring's internal composition is very close to the one used by the programming language COMIT. If AAAAA . . . AAAA is the form of such a metastring, where A stands for one of its constituents, the following context-free grammar (or Backus Normal Form) would generate it:

A⟶b/b/. . ./b	(meaning an arbitrary choice among the enumerated b's)
A⟶$ - b.b..b.b	(meaning any string except the ones described by the b's.)
b ⟶ $n	(meaning n arbitrary consecutive constituents in the parent string; n stands for an integer. $0 denotes the empty string.)
b ⟶ $	meaning an arbitrary string within the parent string, the empty string included.)
b ⟶	any lexical value / tag description combination as described in a previous chapter.
b⟶ Ln or Sn	(meaning the first value of the variable involved, tag included, and the position, if the variable is connected.)

Value Assignment to Tree Variables

In order to enable the user to assign an arbitrary value to a tree variable, the following statement is permissible (the equivalent of the LEXICAL and CONCATENATE statements in the previous cases):

Tn = STRUCTURE k_1 A k_2 A. . . . k_n A

where each k_i is a level number (see COBOL, NPL) and each A stands for one of the following possibilities:

(1) A lexical value with an optional tag

(2) A lexical variable

(3) A tree variable

If A is a tree variable the subsequent level number should not be superior to the preceding one. The statement builds a tree according to the structure shown by the level numbers, by taking the first items on the value lists of the variables involved, without position information. This tree is assigned as value to Tn, by making Tn automatically autonomous.

Absolute, associative and relative value assignments to tree variables cannot be so clearly separated as in the previous cases. The following statement amalgamates various kinds of assignments:

Tn = TREE Lm, A Optional Tree Reference

where Lm is a connected variable and A is an optional list of constituents of the form $(a_1, a_2, \ldots a_k)$. Each a_i stands for one of the following possibilities:

(1) A lexical variable

(2) A lexical value and/or a tag description (see a previous chapter on the specifications)

The effect of the statement is the following: a subtree is selected whose root is the value of Lm. If the A-list is empty, this entire subtree is assigned, as value, to Tn. If the A-list is not empty, all the subtree elements matched by the A-list are considered terminal (i.e. their successors are neglected) and the assigned tree value stops - at its bottom - with these matched elements. In case of conflict the element closer to the root is considered terminal. The autonomy rules are like in the previous cases.

The following statement provides for obtaining the parent tree of a string or lexical variable:

Tn = PARENT Lm or Sm

The broadest autonomous tree containing the argument is assigned, as value, to Tn.

Movement of Variables

Variable values have to be moved, erased, transmitted, etc. The following statements provide for these operations:

Vn = PUSHDOWN Vm, k

where Vn and Vm are variables of the same type and k is an unsigned integer. The first k values on the value list of Vm are added to the value list of Vn, by "pushing down" the previous values on the list of Vn. The value list of Vm remains unchanged.

The statement

PUSHUP Vn, k

erases the first k values on the list of Vn. In both cases, if k is superior to the number of values on the list, the operation stops after the exhaustion of values.

Structure Questions

The user can question the structure of a variable value by comparing it to a given or partially specified pattern. The general form of such a question is:

B Assignment Statement

where B is a logical variable (bit string of length one) and Assignment Statement stands for any of the assignment statements described in the previous chapters. The interpretation of an assignment statement in a structure question is different, however, from its original interpretation. An assignment statement is of the form:

Vn = specification

and the = sign assigns a value to Vn, according to the specification. When an assignment statement enters a structure question no value assignment takes place, the = sign means equality between the variable value and the specification. In case of equality B is set to True, to False otherwise.

Below we list these structure questions with their interpretation. (In the list the various types of parent references are abbreviated.)

Assignment Statement	Structure Question Interpretation
Ln = LEXICAL lexical / tag	Is the value of Ln equal to "lexical" and to the tag?
Ln = (m_1, m_2 . . m_k) MTR	Does Ln occupy the position specified by the index vector in the specified parent tree?
Ln = LEFT or RIGHT k MSR	Does Ln occupy the k-th constituent position in the specified parent string?
Ln = lexical / tag MPR	Does Ln match lexical / tag and is it in the specified parent structure?
Ln = NEIGHBOR Lm OPR	Is the position of Ln in agreement with the specification?
Sn = CONCATENATE AAA. . .A	Is Sn composed of the constituents specified, position information being disregarded?
Sn = LEVEL k MTR	Does Sn correspond to the k-th level string in the specified parent tree?
Sn = STRING (k_1, k_2, specif.) MSR	Does Sn match the specification?

Assignment Statement	Structure Question Interpretation
Sn = WORD Lm OPR	Is the position of Sn in agreement with the specification?
Tn = STRUCTURE k_1AK_2A. . . k_nA	Is the tree value Tn composed as indicated by the structure description, position information disregarded?
Tn = TREE Lm, A OTR	Is Tn the subtree extending as indicated?
Tn = PARENT Lm or Sm	Are Lm or Sm located in Tn?

All these structure questions compare the first item on the value list of the variable involved with the pattern specification. If the user desires to search through the whole value list he can formulate his structure question as:

B EXTRACT Assignment Statement

In this case the whole value list is searched through and the first item encountered satisfying the pattern specification is extracted and put at the top of the value list, while B is set to True. If no item satisfying the pattern specification is encountered B is set to False and the value list remains unchanged.

In a previous chapter the CREATE statement was described that creates an autonomous variable with an empty value list. The question

B CREATE variable

sets B to True, if the variable's value list is empty, to False,otherwise.

The statement

B AUTONOMY variable

-built in an analogous way - has been described earlier.

Equality Tests

Given two variables Vn and Vm of the same type (both lexical, or both string or both tree) the following statement tests the equality of the variable values:

B Vm = Vn

where B is a bit string of length three. The setting to True or False of this bit string is as follows:

First bit True is the lexical composition of Vm is identical to the one of Vn, tags and autonomy status being disregarded.

Second bit True if the largest parent structures are identical or if both variables are autonomous.

Third bit True if the two index vectors are identical. The index vector of a lexical value has been defined earlier, for string and trees it is - respectively - the index vector of the leftmost constituent and of the root. The parent structures may be different and yet the third bit set to True. If both variables are autonomous the bit is set to True.

Numerical Properties of Data and Variables

The following statements establish connections between variables and their numerical properties:

X being a numeric (decimal) variable the statement

X= NUMBER Vn

assigns to X, as value, the number of items on the value list of Vn. The statement

X = WEIGHT Vn

assigns to X, as value, the number one if Vn is a lexical variable, the

number of constituents or tree elements of the value of Vn, if this latter is a string or tree variable. The statements implementing tag handling are:

X = TAG Ln

which sets X to the tag value of Ln, and

Ln = TAG X

which assigns X, as tag, to the value of Ln.

Transformations

Transformation statements change strings and trees into new strings and trees. During the processing of a transformation data may move, or get deleted. This raises the following problem: variable values are affected by these movements and they must be correspondingly modified, updated, deleted, etc. The relevant conventions are as follows:

Whenever a piece of data moves all variable values equal to or entirely contained in that piece of data, have their connections updated, in function of the new position the piece of data moved occupies. If, instead of being moved, the piece of data is deleted, all variable values equal to or entirely contained in the piece of data deleted are deleted.

Variable values overlapping with data moved or deleted are either modified or deleted. They are deleted in the following cases:

(1) The argument of the assignment statement which defined the value gets deleted. (Ec. Sn = ANCESTOR Ln and Lm gets deleted by a transformation; then the corresponding value of Sn is deleted.)

(2) All string variables defined by a STRING Statement.

In all other cases updating and due modification takes place.

String Transformations

The string transformation statement of the Processor is strongly inspired by the "Rewriting rule" of the COMIT programming language, but it is adapted to strings embedded in trees too.

In the case of an autonomous string the transformation statement is a straightforward adaptation of the COMIT rule. Its form is:

string structure specification = REWRITE rewrite indication

Mandatory String Reference.

The "string structure specification" is identical to the one used in the description of the STRING statement, in the chapter on value assignments to string variables. The "rewrite indication" is a sequence composed of:

(1) Integers sequentially numbering the constituents of the "string structure specification" and showing the new position (or deletion, if an integer is omitted in the "rewrite indication") of these constituents after transformation of the string.

(2) Lexical values and string and lexical variables introducing new constituents into the string to be transformed.

The statement verifies if the "string structure specification" matches the Parent Reference. In case of no match, no transformation takes place. In case of match the string is rewritten in the order of the constituents indicated by the "rewrite indication".

In the case of strings connected to trees two classes of strings have to be distinguished: vertical and horizontal strings. A string

embedded in a tree is horizontal if no constituent of the string is the tree-ancestor element of another constituent. It is vertical in the opposite case.

For vertical strings the form of the REWRITE statement is identical with the one for autonomous strings, its interpretation is, however, different: in case of match, if a constituent is moved or deleted, the entire subtree whose root is the constituent in question, is also moved or deleted, except that branch of the subtree which has an element in the string to be transformed. Moreover, in the new position, the former order of the elements of the moved constituent's successor string is maintained; if an element of this successor string didn't participate in the move - because of the above mentioned restriction - it is replaced by the new left neighbor of the moved constituent in the string.

For horizontal strings the form of the REWRITE statement is slightly different: in the "rewrite indication" each constituent must have a left parenthesis somewhere to its left and a right parenthesis somewhere to its right and no other parenthesis can appear between the constituent and these parentheses. In other words the whole "rewrite indication" is simply bracketed. A left parenthesis might be followed by a right parenthesis, with no constituent in between, this couple being placed in the "rewrite indication". After transformation in case of match, constituents between the same couple of parentheses will have the same ancestor element in the tree, in left-to-right order of the parenthesis couples and of the possible ancestor elements of the string.

A couple of parentheses with no constituent in between attributes no constituent of the string to the ancestor element in that position.

Tree Transformations

Three statements ATTACH, DELETE and DETACH provide for the transformation of trees. Variables V referred to below in the description of these statements must be single valued, because of automatic changes in their autonomy status:

(1) ATTACH This statement inserts data in a tree. Its general form is

ATTACH V, P, Ln

where V is an autonomous variable, the value of Ln is a tree element (Ln must be connected) and P stands for one of the words LEFT, RIGHT or UNDER. V is attached to the parent tree of Ln and becomes automatically connected after the attachment. The place of attachment depends on P:

If P is LEFT or RIGHT, V is attached to the left or to the right of the value of Ln, in the sibling string of Ln. In this case Ln cannot be the root of an autonomous tree. If V is a tree, the root is inserted in the sibling string and the remainder of the tree will continue to be attached to the root. All the inserted elements are, of course, connected to the ancestor element of Ln.

If P is UNDER,the specified data will be attached in the successor string of Ln, as its leftmost part, according to the same principles as above.

(2) DELETE This statement deletes data and frees memory in the computer. Its general form is

DELETE V

where V is a variable. If V is autonomous, the corresponding data will entirely vanish. If V is connected, it must be connected to a

tree and the following cases prevail:

(a) V is a lexical value. (In this case it cannot be the root of an autonomous tree.) It is deleted and its descendant tree is attached to the ancestor element of V, in the same sibling position as the one that was occupied by V. Practically, this means that the successor string of V gets inserted at V's place in the sibling string of V, with the remainder of the tree descending from this successor string.

(b) V is a string. All its constituents are deleted except the one that might be the root of an autonomous tree. The descendant trees of the constituents are also deleted.

(c) V is a tree. It is deleted entirely. If the terminal elements (elements with no successors) are not terminal in a larger tree in which the tree to be deleted is embedded, the whole remaining descendant structures of these local terminal elements are deleted.

(3) DETACH This statement detaches data from a tree and holds it in memory. Its general form is

DETACH V

where V is a connected lexical or tree variable. After detachment V becomes automatically autonomous. The detachment of a string can be accomplished with the help of (elementary) statements. The effect of the statement on the parent tree whereupon the detachment takes place is the same as in the DELETE case. Data, however, are not deleted, but separated from their parent tree.

References:

ARMENTI, A. W. et al. — A Data Processing Formalism (Tech. Report 283, Lincoln Lab., MIT, 1962)

BERKELEY, E. et al — The Programming Language LISP (Information International Inc., 1964)

CHOMSKY, N. — On the Notion of "Rule of Grammars" (Proc. of Symp. in Appl. Math., Vol. 12, Am. Math. Soc., 1961)

GENUYS, F. — Commentaires sur le language ALGOL (AFCALTI Seminar on Programming Languages, Paris, 1962)

IVERSON, K. — A Programming Language (John Wiley and Sons, Inc.)

MATTHEWS, G. H. — Analysis by Synthesis in the Light of Recent Developments in the Theory of Grammar (Department of Modern Languages, MIT, 1964)

MOYNE, J. A. — Restrictive Language Defining System (IBM Data Systems Division, Advanced Computer Utilization, ACU-011, 1963)

NEWELL, A. et al. — IPL-5 Manual (Prentice Hall, Inc., 1961)

OETTINGER, A.G. -KUNO, S. — Multiple-path Syntactic Analysis (Proceedings IFIPS 1962)

RADIN, A. -ROGOWAY, H.P. — NPL, Highlights of a New Programming Language (Comm. ACM, Vol. 8, No. 1, 1965)

YNGVE, E. et al — COMIT Programmer's Reference Manual (MIT-RLE Publication)