A FORMAL DEFINITION OF THE SCAN ALGORITHM OF THE TRAC T-64 LANGUAGE

G. E. Whitney, Western Electric Company, Princeton, N. J.
C. N. Mooers, Rockford Research Institute Inc., Cambridge, Mass.

1. INTRODUCTION

This paper presents a formal definition of the scan algorithm of the TRAC† language. The definition does not extend to a complete processor for the language since the definition of an evaluator for expressions is not given. This paper provides a rigorous starting point from which a complete processor could be constructed.

1.1 Background of the language.

TRAC language was developed for interactive text or string processing. Its basic structure is such that it can be implemented on small as well as large computers. The defining report for the language is [1]. Additional explanatory information is found in [2]. TRAC language has certain features not found in many other languages because of a dynamic relationship between the scanning of the text of a program‡ and the evaluation of an expression once it has been successfully parsed. The essence of this relationship is given in figure 1.

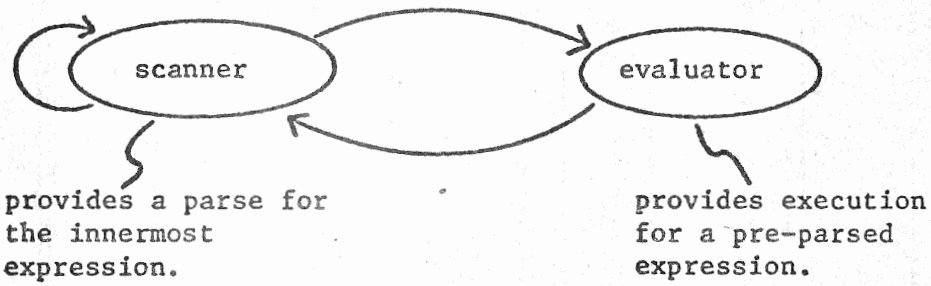


Fig. 1. Simplified state diagram of a TRAC language processor.

The scanner proceeds from left to right. Prior to evaluation only a sufficient portion of the source text is processed to complete the parsing of the innermost expression. At this point control is transferred to the eval-

---

† Trade mark and service mark of the Rockford Research Institute Inc.

‡ An input string is called a script rather than a program since in TRAC language both source program and data are intermixed on the same input.

uator. When evaluation is complete, side effects if any will have been recorded, the source text will have been altered, and control is transferred back to the scanner. From this point the process repeats. Inherent within this processor is the ability to execute functions recursively. The description of the scan algorithm must include a means of indicating a branch to the evaluator and subsequent return to the scanner. The scanner itself must provide the ability not only to parse the source text but also to translate the text as necessary in proparation for expression evaluation. This separation of scanning and evaluating is similar to the ideas of inner and outer syntax as defined in [3].

1.2 An example of the TRAC language processor.

Figure 2 shows the relationship between the two major strings which provide the environment for the TRAC processor. The neutral/active string is regarded as residing on a pair of unbounded pushdown tapes, where the boundary between the tapes is indicated by the scan pointer.

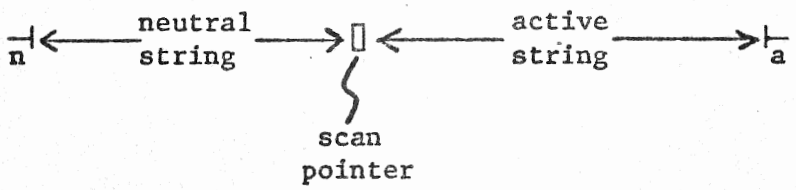


Fig. 2. The schema for the instantaneous description utilized by the TRAC language processor.

The following example is intended to make the body of the paper more readable for those not familiar with [1]. In its initial configuration, the neutral/active string of the TRAC processor contains $\dashv_n \square\#(ps,\#(rs)) \vdash_a$. In this configuration, $\dashv_n$ is the left end marker for the neutral string; □ marks the boundary between the neutral and the active string; $\vdash_a$ is the right end marker for the active string. The symbol □ is called the <u>scan pointer</u>. For this example, primitive functions will be enclosed by #(and).

The operation of the processor is as follows:

(1) Read-string:

The innermost primitive expression of the initial configuration is found by movement of the scan pointer. This expression is #(rs) which is the read-string or read-in function. This expression is then evaluated and causes an input string to be read from the typewriter keyboard or other input device. Let the input string be #(ad,5,7)' . Within this string, the apostrophe ' is called the metacharacter, and it signifies the termination of the input string. As a result of evaluation, the expression #(rs) is repalced by the input string #(ad,5,7) and the metacharacter is omitted.

(2) Add:

Now the innermost expression is #(ad,5,7) . In its evaluation, it is replaced by its value 12, which is the sum of the arguments 5 and 7.

(3) Print-string:

Now the innermost expression is #(ps,12) where ps stands for the print-string primitive function. When evaluated, this expression prints the value 12. Since there are no further functions to evaluate, the processor reloads itself with the initial configuration.

1.3 Goals in the formal definition of a language.

The goal of formal definition of a programming language is primarily to achieve mathematical precision in the statements which describe the language. Its main value is in man to man-communication about the syntax and semantics of a language. This communication has value both for implementors and users of the language. Since it is difficult to achieve the desired precision with natural language, other special languages are defined and used as vehicles for definition. This was done for PL/I, see [4]. Because of the nature of TRAC language, a new grammar model, called a type-T grammar†, has been developed as a vehicle for the definition of the scan algorithm. Care has been taken throughout this presentation to show the relationship between the pre-

sent definition and that given in [1]. The present definition is intended to be precise, compact, complete and independent of the space and speed performance of specific implementations of the algorithm.

---

† A type-T grammar is an adaption of a type-0 grammar. This adaptation allows a type-T rule when invoked to preserve a terminal string during the evaluation of the rule even when this terminal string is specified in the rule by means of a context-free category. Type-T rules are defined in full in section 3. below.

The following conventions about phrase structure languages will be assumed throughout. For α having as its range, any set of strings, then:

$\{\alpha\}^* = \{\alpha\}^+ \cup \{\varepsilon\}$ where ε is the empty string, and where

$\{\alpha\}^+ = \bigcup_{i=1}^{\infty} \alpha^i$ where $\alpha^1=\alpha$ and $\alpha^{i+1}=\alpha^i\alpha$.

Let string be an element in the set {category ∪ terminal}*. A type-0 grammar has rules of the form "string-1 -> string-2". For a type-1 grammar the length of string-2 must not be less than the length of string-1. In type-2 (i.e. context-free) grammar rules, string-1 is restricted to a category.

If string-2 -> string-4 is a rule, then the application of this rule to the instantaneous description "string-1 string-2 string-3" produces the result "string-1 string-4 string-3". This operation is called a generation step and is denoted by:

string-1 string-2 string-3 $\Rightarrow$ string-1 string-4 string-3 ,

where $\Rightarrow$ is a transitive relation between a pair of instantaneous descriptions. A generation sequence of n-1 steps between n instantaneous descrip- (ID's) is denoted by ID-1 $\overset{*}{\Rightarrow}$ ID-n where $\overset{*}{\Rightarrow}$ indicates the intervening sequence of steps. A grammar defines a language as the set of all terminal strings which can be produced by applying its rules when the initial configuration for each generation sequence is the sentence symbol.

A model based on a type-0 grammar was used in [5,6] to define a variety of context-free parsers. A form of type-0 grammar was used in [7] to partially define the TRAC scan algorithm. Complete definitions of the 0,1 and 2 grammar types can be found in [8] on page 15. A type-0 grammar model is equivalent to the reduction language of Floyd as reported in [9] section II.B, Introduction and subsection 5.

1.4 Methodology of formal definition.

A definition consists of a statement, or sequence of statements, written in a meta-language about some object being defined. A non-trivial meta-language must be capable of defining an infinite class of objects. The meta-language must include the elements of the object language as a proper subset of its elements and must have other elements at the meta level which are syntactically independent of the object language, otherwise definition would not be possible. A definition may permit a recursive reference to

an object being defined, but this recursion must be within a level of definition and not across levels of definition. In the process of definition, there arises naturally a hierachy of language levels, with the higher level standing in a meta relationship to the one immediately below it. The highest meta-language must be natural language. This paper will utilize three levels. The first or top level is English which is used to define the second level languages which are the context-free grammars and the type-T grammars. These grammars in turn are used to define the mapping rules between successive configurations of the neutral/active string. The set of these mapping rules contitute the scan algorithm which is the third or target level of definition.

1.5 Non-procedural algorithms.

The term rule will be used to denote a conditional statement and an associated action which is to be performed when the condition is satisfied. Such a set of rules is usually tested for satisfaction of the conditions in a fixed sequence. This case appears in figure 3 under the title "Sequential Evaluation". Departures from this fixed sequence could be provided for by the use of a goto cause a branch to a rule which is not the next one in the sequence. However a non-procedural algorithm consists of a set of rules which can be evaluated in any sequence. Conceptually such a set of rules should be evaluated in parallel to see if the conditions for more than one rule are satisfied. This case appears in figure 3 under the title "Parallel Evaluation". If more than one rule can be satisfied for any configuration of the system, then the set of rules is non-deterministic. On the other hand, if there is at most one rule which can be satisfied for every configuration of the system, then the set of rules is determinstic. If an algorithm of n rules is both deterministic and non-procedural, then the rules can be ordered in n! different sequences all of which are logically equivalent when the rules are tested for satisfaction in a fixed sequence.

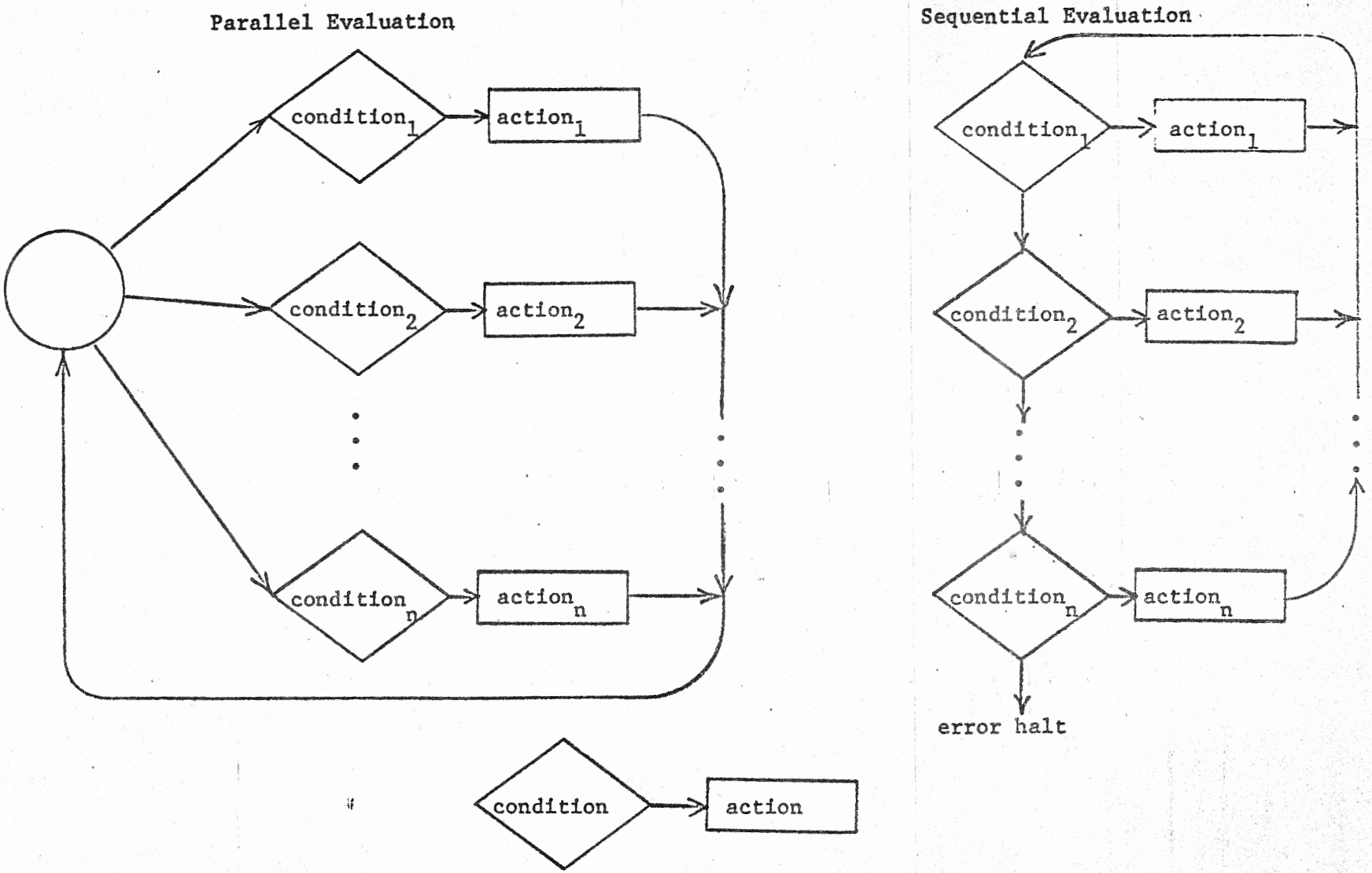


Fig. 3. Alternative methods of rule evaluation for non-procedural algorithms.

1.6 Relationship between a non-procedural algorithm and a grammar.

In order to explain the relationship between a non-procedural algorithm and a grammar it is necessary to introduce the concept of an abstract machine having a finite control and a fixed number of auxiliary storage tapes. This abstract machine has a set of rules which define its operation and comprise its finite control. If the machine has only one state, each rule when evaluated returns control back to the one controlling state. From the discussion given in 1.5 above, it should be evident that a one-state machine is equivalent to a non-procedural algorithm.

A conventional grammar is equivalent to a one-state machine which defines a mapping from a fixed initial configuration into a set of terminal strings. This fixed initial configuration consists of a distinguished category called the sentence symbol. An extended grammar will be defined as a grammar for which the initial configuration is not fixed but is allowed to range over a a set of strings composed of a scan pointer and terminals. Processing will be understood to halt when no rule can be applied. An extended grammar then defines a mapping from one set of terminal strings into another. Depending on the grammar, this mapping may be either a partial or a total function.

If the rules of a non-procedural algorithm define a mapping from one set of strings to another, then for every such algorithm there exists a directly equivalent extended grammar. For the remainder of the paper, the term grammar is to be understood in the more specific sense of extended grammar.

1.7 Definition problems peculiar to the TRAC class of languages.

Since by definition TRAC language accepts all strings as legal input, a grammar defining merely legal strings is trivial. A more complex grammar could define a parse of the entire input script without considering the effects which interactive execution has in altering the script. However, such a parse would actually be erroneous for many non-trivial cases. These factors

indicate that a context-free grammar of TRAC language scripts would not be a meaningful definitional device. Also, direct extensions of context-free notation such as table grammars [10] are excluded and for the same reasons. The stages of redefinition of the TRAC language to be carried out in this paper are given in figure 3. The ocntext-free categories are defined first. These categories are then utilized to provide a non-procedural restatement of the original algorithm (Section 2.). This algorithm is subsequently restated as a type-T grammar (Section 4.), and as a decision table and its flow chart (Section 6.).

1.8 The form of context-free rules.

Certain context-free categories are used in the statement of the scan algorithm. Each of these categories is defined by a composite rule of the form:

category ::= expression-1
.
.
.
::= expression-n .

A composite rule which has n rightside expressions is said to be composed of n primitive rules each of the form:

category ::= expression-i .

Each expression-i is a string consisting either of the symbol ε or of an arbitrary number of either categories or terminals. The symbol ε stands for the string of length zero , also called the empty string. Each category appearing within a rule has the form:

< λ >

where λ stands for any string composed of letters and blanks.

1.9 Terminals and undefined categories.

With no loss of rigor, the categories <digit>, <letter>, <format character> and <user delimiter> will remain formally undefined. They are to be understood

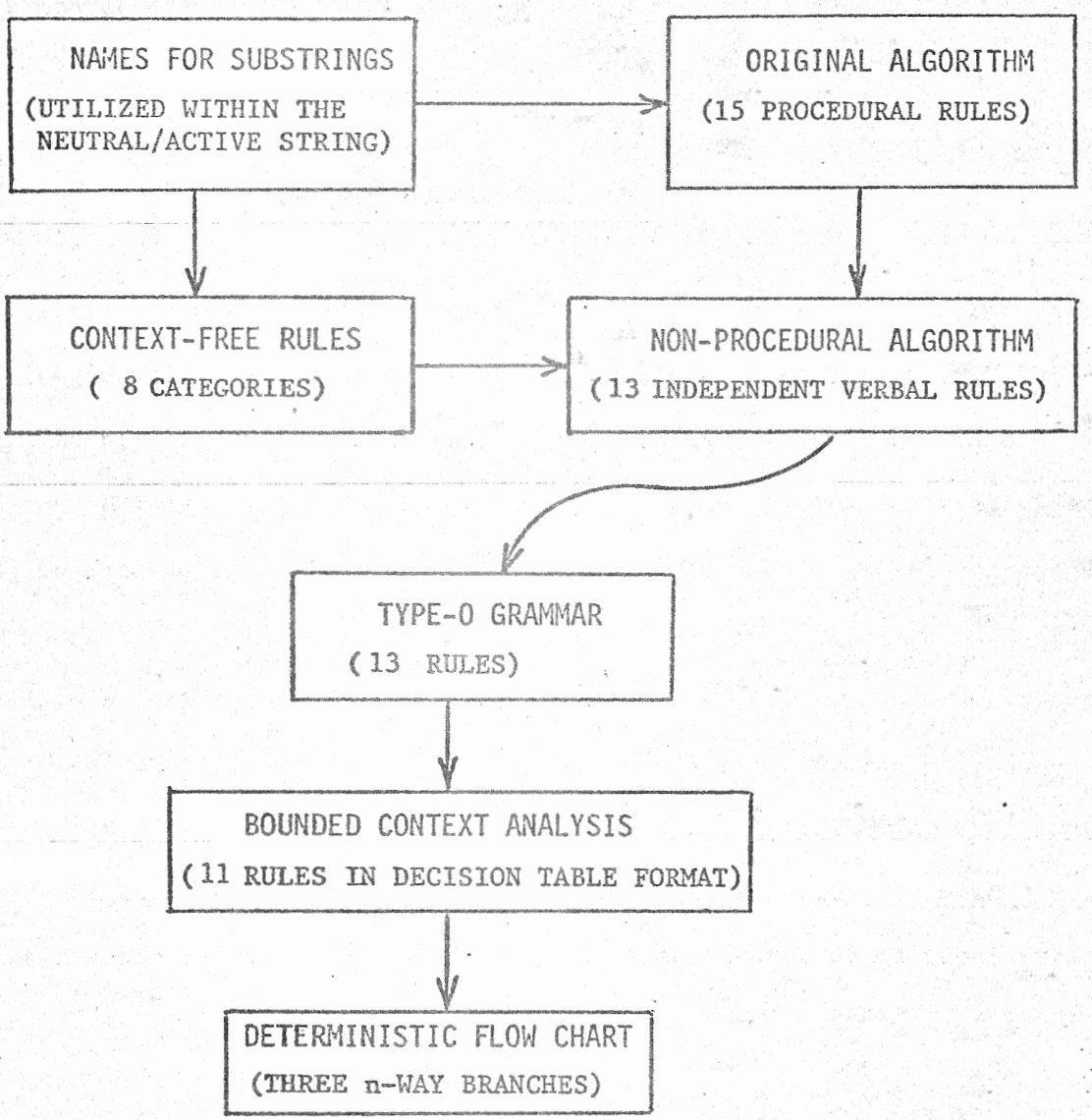


Fig. 4. Outline of the stages of redefinition of TRAC language scan algorithm.

as disjoint categories each of which has as its range a set of terminals. The category <digit> includes the ten numeric digits as terminals. The category <letter> includes the alphabetic characters (either upper case or lower case or both) as terminals. The category <user delimiter> is to be understood to include period and space and : and ; and other special characters, but specifically to exclude comma and (and) and #. There are two additional terminals which are not part of the TRAC language but which are necessary for complete precision in formal definition. These terminals are $\dashv_n$ and $\vdash_a$ which are respectively the left end marker for the neutral string and the right end marker for the active string. Note that the symbol called the metacharacter is not a context-free category since it is a terminator of an input string and as such does not appear in the definition of the scan algorithm.

1.10 Rules which define the context-free categories.

<idling procedure> ::= #(ps,#(rs))

<text character> ::= #
::= ,
::= <digit>
::= <letter>
::= <user delimiter>

<balanced string> ::= ε
::= (<balanced string>)
::= <balanced string> <balanced string>
::= <text character>
::= <format character>

<unbalanced string> ::= (<unbalanced part>

<unbalanced part> ::= (
::= <balanced string>
::= <unbalanced part> <unbalanced part>

```
<script character>  ::=  (
                    ::=  )
                    ::=  <text character>
                    ::=  <format character>

 argument>          ::=  ε
                    ::=  <script character>
                    ::=  <script character> <argument>

 rgument sequence>  ::=  <argument>
                    ::=  <argument> , <argument sequence>
```

.11 Unbalanced strings.

ote that an <unbalanced string> is always defined to have an excess of open parens. Also such a string always begins with (. The case of a string with an excess of close parens can occur, but it is handled without the need for a special category to name the substring which is involved.

1.12 Omitted categories.

There are no special categories to name and define the neutral string and the active string. These omissions are intentional. The schematic relationship between these strings and the scan pointer and the respective end markers is given in figure 2. A context-free definition of these strings is not meaningful because interaction between the scan algorithm and the evaluator dynamically alters the contents of both strings.

1.13 Representation of a grammar as a syntax graph.

The context-free grammar rules of 1.10 can be displayed as a syntax graph. Such a graph appears in figure 6. The rules by which such a graph can be constructed are given in figure 5 where five specific cases are itemized. The explanation of these five cases is as follows:

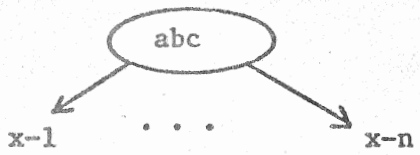
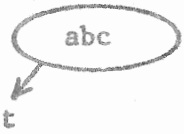
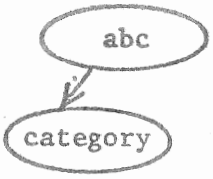
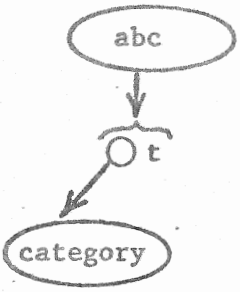
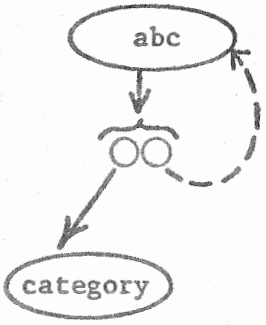
(1) A category in a grammar defined by a composite rule, composed of n primitive rules is represented in the graph by an elipse enclosing the category name with < and > omitted and with n directed arcs leading away from the elipse, one arc for each primitive rule.

(2) When the rightside of a primitive rule is a terminal, then the terminal appears in the graph unchanged and with no arc leading away from it.

(3) When a primitive rule consists of simply of a category on the leftside and a category on the rightside, then the rule is represented by a single directed arc linking two elipses.

(4) When the rightside of a primitive rule is a string of more than one symbol then the arc will point to a horizontal brace which spans the graph representation of that rightside string. In this representation terminals stand for themselves but categories are replaced by a circle with an arc leading out the the actual elipse where that category is defined.

(5) If a category is defined recursively, the arc indicating this is given as a broken line.

Case	Grammar Format	Graph Format
(1)	<abc> ::= x-1 ⋮ ::= x-n	
(2)	<abc> ::= t	
(3)	<abc> ::= <category>	
(4)	<abc> ::= <category> t	
(5)	<abc> ::= <category> <abc>	

x-1 , x-n are arbitrary rightsides of a rule.

t is an arbitrary terminal.

Fig. 5. Equivalences between grammar format and graph format for the representation of context-free syntax rules.

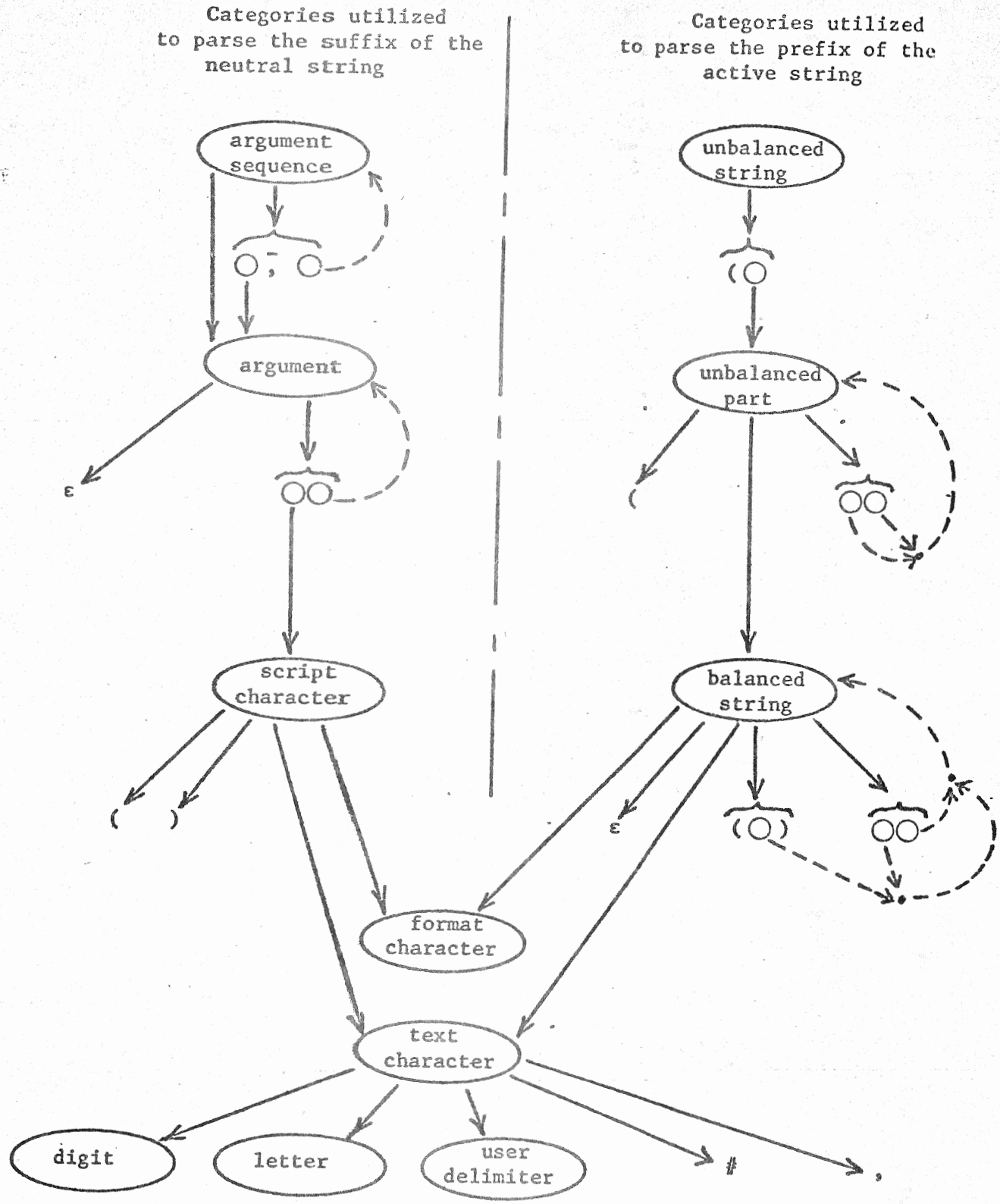


Fig. 6. Syntax graph of context-free categories.

2.0 A NON-PROCEDURAL STATEMENT OF THE SCAN ALGORITHM

In [1] the scan algorithm is given as a procedure consisting of fifteen rules. These rules are in the form of conditional statements and are linked together by goto's. Before this algorithm can be stated as a type-T grammar it is necessary to convert this procedural definition into a set of non-procedural rules, i.e. rules for which no specific order of execution is required. The result of this conversion appears in 2.1 to 2.9 where the scan algorithm is defined in the form of nine non-procedural rules. Appendix I includes a table showing the relationship between this algorithm and the one given in [1].

S C A N A L G O R I T H M

The TRAC language scan algorithm operates on strings contained between the end markers $\dashv_n$ and $\vdash_a$ with [] as the scan pointer. This scan environment is shown in figure 2.

2.1 If the active string is empty, then delete the neutral string, reload the <idling procedure> and position [] to the left of the first character of the <idling procedure>.

2.2 When a (is encountered immediately to the right of [], different actions are possible depending on the context:

2.2.1 If there is a string of the form (<balanced string>) immediately to the right of [] then the enclosing parens of this <balanced string> are removed and [] is moved to the right of the <balanced string>.

2.2.2 If the character on the right of [] is (and this (is the first character of an <unbalanced string> then the <idling procedure> is reloaded because one or more close parens are missing.

2.3 If the character on the right of [] is a <format character> it is deleted.

2.4 If the two characters to the right of [] are #(, then they are replaced by $\overline{(}$ and [] is moved to its right.

2.5 If the three characters to the right of [] are ##(, then they are replaced by $\overline{(}$ and [] is moved to its right.

2.6.1 If the character to the right of [] is # and if the character to the right of the # is neither # nor (, then [] is moved to the right of #.

2.6.2 If the two characters to the right of [] are ## and if the character to their right is not (, then [] is moved to the right of the leading # .

2.7 If the character on the right of [] is a comma, it is marked as $\overline{,}$ and [] is moved to its right.

2.8 If the character on the right of [] is) , then a leftward scan of the neutral string will determine which of the following three alternatives is to apply. [] itself is not to be moved during this leftward scan.

2.8.1 If the leftward scan encounters $\overline{(}$ prior to $\overline{\overline{(}}$ or $\dashv_n$ then the intervening string is the <argument string> for the active function evaluation. The evaluator replaces $\overline{(}$<argument string> []) with the <value> returned and positions [] to the left of this <value>.

2.8.2 If the leftward scan encounters $\overline{\overline{(}}$ prior to $\overline{(}$ or $\dashv_n$ then the intervenening string is the <argument string> for neutral function evaluation. The evaluator replaces $\overline{\overline{(}}$ <argument string> []) with either a <success value> or a default value> depending on whether it succeeds or fails. If the evaluation succeeds, [] is positioned to the right of <success value>. If the evaluation fails, [] is positioned to the left of the <default value>.

2.8.3† If the leftward scan encounters $\dashv_n$ prior to $\overline{(}$ or $\overline{\overline{(}}$ then the close paren to the right of [] is deleted†.

2.9 If the character on the right of [] is a <text character>, then [] is moved to the right of this <text character>.

† This rule conforms to [1] but current standard TRAC language processors are implemented as follows: "If the leftward scan encounters $\dashv_n$ prior to ⟅ or ⟆ then the active string and the neutral string are deleted, and the ⟨idling procedure⟩ is reloaded with □ positioned on its left ".

‡ Since the current contents of the ⟨neutral string⟩ can never be evaluated, the processor will be more efficient if the ⟨neutral string⟩ is also deleted by this rule.

2.10 Non-procedural aspects of the algorithm.

Note that the logical conditions under which each of the above nine rules may be applied are independent of the sequence in which previous successful rules were applied. The rules are to be understood as logically unordered and no rule refers to another rule by means of a goto statement. The rules also form a partitioned set in which the conditions for the successful evaluation of rule are disjoint from the conditions for the success of every other rule. In addition, the rules are complete since they provide an exhaustive cover for all possible cases of strings in the environment of figure 2.

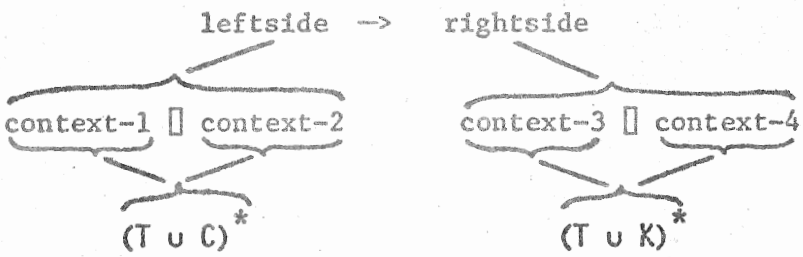
This method of specification does not in general assure that there will exist a direct deterministic processor. That is a processor which can make the correct choice every time without backtracking. In general when this method is used, it may be possible that there exists some situation in which more than one rule may apply. However, in section 6. below it will be shown that this non-procedural algorithm actually is deterministic. The nine rules may be ordered in any one of 9! different ways. In all of these possible orderings, the operation of the algorithm is identical. In an actual implementation an optimum ordering of the rules could be obtained by statistical considerations on the frequency of the execution of each rule, together with a cost factor for each rule based upon the space and speed characteristics of the target machine.

3. DEFINITION OF A TYPE-T GRAMMAR

In Section 4., which follows, the scan algorithm is presented as a set of thirteen type-T grammar rules. The purpose of this section is to define and explain the notation to be employed in the grammar rules of 4. These type-T rules are numbered (4.1 to 4.9) in one-to-one correspondence with the verbal non-procedural statement of each rule given above (2.1 to 2.9).

3.1 Type-T rule schema.

The schema for type-T rules is given in figure 7. Unless stated otherwise



Note that the quantification given for the leftside has been simplified.

Fig. 7. Schema for type-T rules.

quantification is as given in figure 7.

Notation:

T is the set of terminals.

C is the set of context-free categories. The extended-class of categories given in 3.2 below are also context-free categories.

K = $\{c_1, c_2, \ldots, c_n\}$, which is defined for a particular rule as the set of categories appearing on the leftside of that rule. Thus figure 7 requires that the quantification of the "rightside" be restricted to those categories which also appear in the corresponding "leftside" of the same rule.

V = T ∪ C, this is the set of the total vocabulary.

Quantification:

The precise quantification of "leftside" within a type-T rule is not given in figure 7. This quantification is given by:

"leftside" is in $\{V[] \cup []V \cup V^+[]V^+\}$.

This added precision assures that no leftside of a rule will be represented

by [] standing alone.

Syntactic Restriction:

For a given rule and for c_i, c_j in K, then $c_i \neq c_j$ for all $i \neq j$. Synonyms will : allowed for the context-free categories so that the restriction does not tually restrict the parse of the leftside.

mantic Restriction:

the same category appears on both the leftside and on the rightside of a ule, then the exact terminal string spanned by the leftside category is pre- erved during the evaluation of the rule. However the position of that string ithin the context may have been alatered in accordance with the specification iven by the rightside. This is explained in detail in 3.3 to 3.7 below.

Remark:

There do not exist any type-T categories in this grammar system. All categ- ories must be defined by some context-free grammar. This fact indicates the type of a scan which is nexessary to determine if the conditions of a rule are satisfied. A pushdown automaton will be powerful enough to carry out the required trial parse. An example of the type of parsing required by a type-T rule is given in figure 8.

3.2 Extended-class of categories.

In listing the rules of a grammar, it is sometimes much more compact to define a category by ennumerating what is not in the category rather than by the usual method of ennumerating what is in the category. For this purpose, the following definition is given:

$$\langle \neq \{ \text{ list } \} \rangle = \Sigma - \{ \text{ list } \}.$$

The expression { list } is to be replaced by specific terminals in the form:

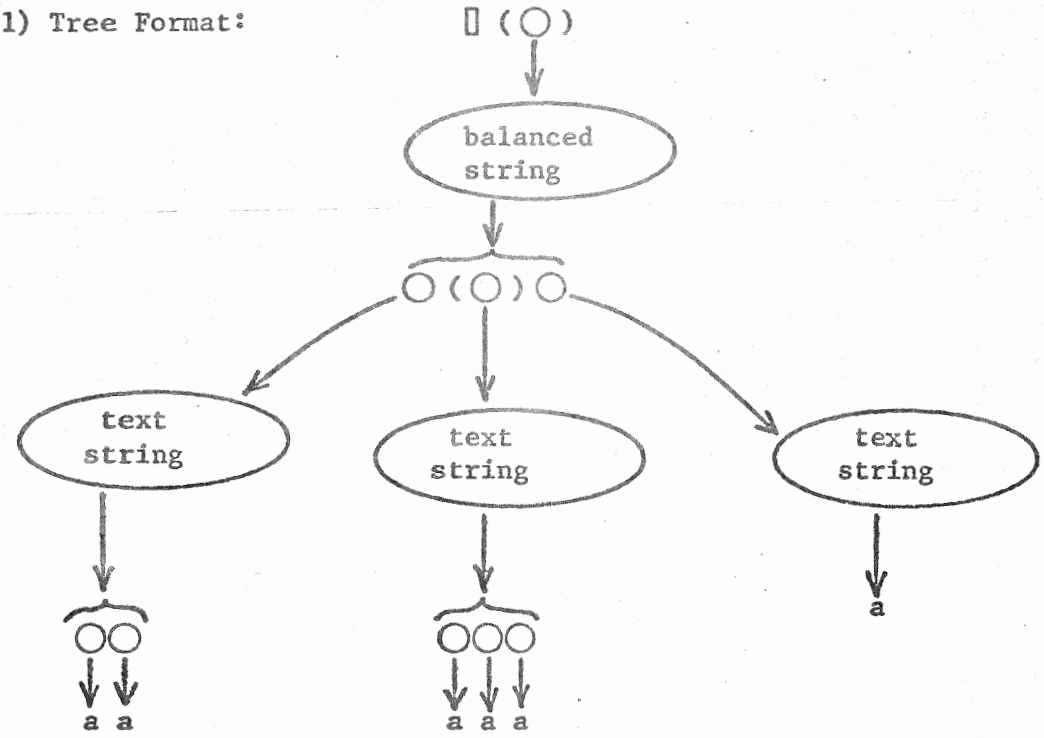
$$\{ \text{terminal}_1 \mid \cdots \mid \text{terminal}_n \}.$$

For example the category $\langle \neq \{\#|(\} \rangle$ means: "any symbol in Σ except # or (". When "list" is a single terminal the surrounding { and } are omitted.

The set of terminals (denoted by Σ) will include the three undefined terminal categories given in 1.9 above and the end markers. Thus,

$$\Sigma = \{\langle\text{digit}\rangle \mid \langle\text{letter}\rangle \mid \langle\text{user delimiter}\rangle \mid \# \mid (\mid) \mid , \mid \bar{(} \mid \bar{)} \mid \bar{,} \mid \dashv_n \mid \vdash_a \}\ .$$

1) Tree Format:



(2) Stratified Format:

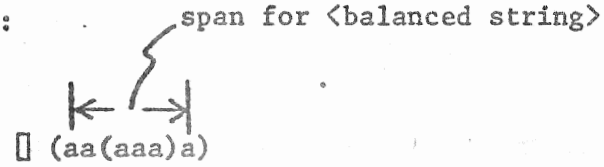


Fig. 8. Example of the parse produced to match the leftside of a type-T rule.

3.3 The computation step defined by a type-T rule.

A context-free language is so defined that its generation sequence will terminate. A type-T grammar as used here does not define a terminating sequence, but rather each rule defines a computation step in a potenially infinite sequence. As in phrase structure languages, a transitive step is denoted by $\Rightarrow$ and a sequence of steps by $\overset{*}{\Rightarrow}$.

If []a -> a[] is a rule then bc[]ad ⇒ bca[]d is a legal step which results from the application of the rule. If c[]a -> f[] is a rule then bc[]ad ⇒ bf[]d is a legal step.

3.4 Context-free categories within a type-T rule.

Figure 9 gives an example of the operation of a type-T rule in which the same category appears on both the leftside and the rightside of a rule.

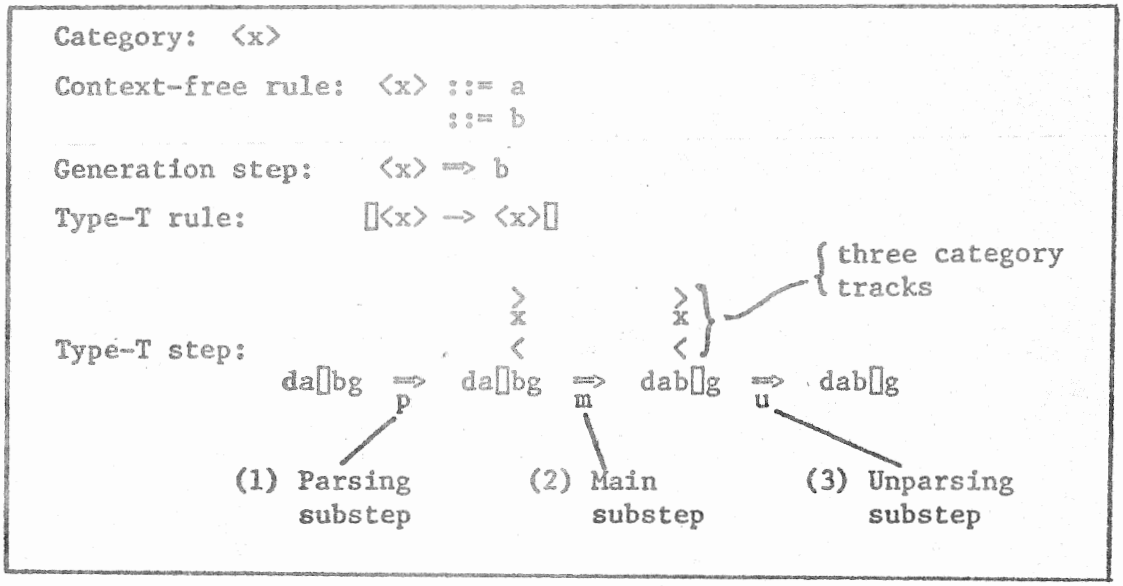


Fig. 9. Preservation of a terminal character under a category during the operation of the three substeps within a type-T step.

This system consists of a single category <x> and one type-T rule, []<x>-> <x>[]. The type-T step is shown broken down into its three substeps. Note that throughout the mapping, the terminal b was preserved (i.e. not changed to a or to <x>), although this is not explicitly stated by the type-T rule. The explanation of each substep is as follows:

(1) Parsing substep = $\underset{p}{\Rightarrow}$:

This substep discovers a local parse which is a match for the category <x>. This match provides satisfaction for the leftside of the type-T rule. The new parse information is stored on the three category tracks above the terminal track.

(2) Main substep = $\underset{m}{\Rightarrow}$:

This substep carries out the operation indicated by the rule in accordance with the restriction that "the terminal string under a category is preserved".

(3) Unparsing substep = $\underset{u}{\Rightarrow}$:

This substep deletes the parsing information from the three category tracks, so that the tracks will be empty for the application of the next rule.

The number of categories in a finite grammar must be finite. Thus although the names identifying actual categories may be composed of several letters, it is always possible to recode the category names onto some finite alphabet in which each category is represented by a single unique letter. Hence for the examples of figure 9, 10 and 11 only single letter categories are employed. In an actual grammar such a restriction is unnecessary.

3.5 A category which spans an arbitrary number of characters.

A special utilization of the three "category tracks" is required in the case where a category spans more than a single terminal character. An example of this case is given in figure 10. Note that < on the first category track marks the beginning of the span, and that > on the third category track marks the end of the span. The category name, "y" is repeated once for each terminal in the string it spans. This method of processing is equivalen to handling parsing trees by their nod names. Figure 11 gives a summary of track utilization. In this figure, each of the terminals within the span of the category <z> is marked by a z on track C2. The terminals under each z are preserved unaltered during the evaluation of the rule.

Category: ⟨y⟩

Context-free rule: ⟨y⟩ ::= a
::= a⟨y⟩

Generation sequence: $\langle y \rangle \overset{*}{\Rightarrow}$ aaaa

Type-T rule: []⟨y⟩ → ⟨y⟩[]

Type-T Step: b[]aaaac $\underset{p}{\Rightarrow}$ b[]aaaac $\underset{m}{\Rightarrow}$ baaaa[]c $\underset{u}{\Rightarrow}$ baaaa[]c

Fig. 10. A type-T step utilizing a category which spans a string.

Generation sequence: $\langle z \rangle \overset{*}{\Rightarrow} b_1 b_2 \cdots b_n$

Type-T rule: []⟨z⟩ → ⟨z⟩[]

category-tracks	C3	>
	C2	z z ••• z
	C1	<
terminal-track	T	•••aa[]$b_1 b_2$ ••• $b_n$cc•••

Parsed string: $b_1 b_2 \cdots b_n$

Unparsed strings: •••aa and cc•••

< on C1 marks the beginning of the parse span.

zz•••z marks the scope of the parse span.

> on C3 marks the end of the parse span.

Fig. 11. Summary of the utilization of the category tracks.

3.6 Case of the empty string.

In the context-free grammar given in 1.10 above, cases occur where for some category ⟨x⟩, ⟨x⟩ $\overset{*}{\Rightarrow}$ ε. In this case there would be no place on the tape to write ⟨x⟩ over ε, since ε does not occupy a character position. Without giving a formal proof† it is claimed that this case can be handled without altering the definitional power of the system.

3.7 Notation for transfer to the evaluator.

The symbol —e—> will be used in place of —> to indicate that the execution of a rule so designated will require a transfer of control temporarily to the evaluator before the scan-algorithm can resume its processing. The categories ⟨value⟩, ⟨success value⟩ and ⟨default value⟩ will be allowed on the rightside of a rule marked with —e—>. These three categories will represent strings of terminal symbols but the exact specification of what these strings contain lies outside the scope of this paper. This information is dependent upon the formal definition of the evaluator itself. However ⟨success value⟩ and ⟨default value⟩ are mutually exclusive in the sense that one or the other must equal ε. With this restriction in mind then the following holds:

⟨value⟩ = ⟨success value⟩ ⟨default value⟩ .

---

† To see that this is true, note that ⟨x⟩ can appear only a finite number of times on the rightside of the various context-free rules. The rule ⟨x⟩ ::= ε can be deleted and its effect can be retained by adding a set of parallel rules to the grammar. These new rules are added by using the formula ⟨x⟩=ε and performing a uniform substitution throughout the rightsides of the other rules in the original grammar. This expanded grammar will define the same language and the category ⟨x⟩ will no longer generate ε. The naming significance of ⟨x⟩ will be unaltered except with respect to ε. This same expansive substitution must also be performed with respect to the tules of the type-T grammar.

4. TYPE-T GRAMMAR FOR THE SCAN ALGORITHM OF THE TRAC LANGUAGE

	Syntax	Comments
4.1	$\square \vdash_a \rightarrow \dashv_n \square \#(ps,\#(rs)) \vdash_a$	Re-initialize to the idling procedure.
4.2.1	$\square$ (<balanced string>) $\rightarrow$ <balanced string> $\square$	Remove protecting parens from a balanced string.
4.2.2	$\square$ <unbalanced string> $\vdash_a \rightarrow \dashv_n \square \#(ps,\#(rs)) \vdash_a$	Re-initialize due to missing close paren.
4.3	$\square$ <format character> $\rightarrow \square$	Delete unprotected format character.
4.4	$\square \#(\rightarrow \overline{(}\ \square$	Mark the open paren of an active function.
4.5	$\square \#\#(\rightarrow \overline{\overline{(}}\ \square$	Mark the open paren of a neutral function.
4.6.1	$\square \#\langle\neq \{\#\mid()\rangle \rightarrow \#\ \square\ \langle\neq \{\#\mid()\rangle$	Step past a #.
4.6.2	$\square \#\#\langle\neq ()\rangle \rightarrow \#\ \square\ \#\langle\neq ()\rangle$	Step past a #.
4.7	$\square$, $\rightarrow \overline{,}\ \square$	Mark a comma which separates arguments.
4.8.1	$\overline{(}$ <argument string> $\square$) $\xrightarrow{e}$ $\square$ <value>	Execute active function evaluation.
4.8.2†	$\overline{\overline{(}}$<argument string> $\square$) $\xrightarrow{e}$ <success value> $\square$ <default value>	Execute neutral function evaluation.
4.8.3‡	$\dashv_n$ <argument string> $\square$) $\rightarrow \dashv_n$ <argument string> $\square$	Delete excess close paren.
4.9	$\square$ <text character> $\rightarrow$ <text character> $\square$	Step past a text character.

† <success value> and <default value> are mutually exclusive in the sense that one or the other must equal ε, the empty string.

‡ See footnote on 2.8.3 for an alternate interpretation.

5. EXAMPLE OF SCRIPT PROCESSING

In table I a sequence of eight instantaneous descriptions (ID's) are given which illustrate configurations during the processing of a script in which both input and output occur. The following additional conventions are adopted:

$\dashv_i$ and $\vdash_i$ are end markers for the input string.

' is the metacharacter for the "read string" function.

$\dashv_p$ and $\vdash_p$ are the end markers for the printer string.

Table II gives a detailed explanation of the processing which occurs between each pair of instantaneous descriptions. Σ^* represents any string.

ID	Neutral/active string	Input string	Printer string
1	$\dashv_n \Sigma^* \square \vdash_a$	$\dashv_i \#\#(rs))'),('\vdash_i$	$\dashv_p \vdash_p$
2	$\dashv_n \square\#(ps,\#(rs)) \vdash_a$	no change	no change
3	$\dashv_n (ps,\overline{(rs\square)}) \vdash_a$	"	"
4	$\dashv_n (ps,\overline{\square\#\#(rs))}) \vdash_a$	$\dashv_i),('\vdash_i$	"
5	$\dashv_n (ps,\overline{\overline{(rs\square)}})) \vdash_a$	no change	"
6	$\dashv_n (ps,),(\square)) \vdash_a$	$\dashv_i \vdash_i$	"
7	$\dashv_n \square) \vdash_a$	no change	$\dashv_p),(\vdash_p$
8	$\dashv_n \square \vdash_a$	"	no change

Table I. Typical sequence of instantaneous descriptions (ID's).

Processing Step	Sequence of † type-0 rules	Comments
$ID_1 \Rightarrow ID_2$	1	Reinitialize.
$ID_2 \overset{*}{\Rightarrow} ID_3$	4,9,9,7,4,9,9	Scan for close paren.
$ID_3 \Rightarrow ID_4$	8.1	Execute active "read string" function.
$ID_4 \overset{*}{\Rightarrow} ID_5$	5,9,9	Scan for close paren.
$ID_5 \Rightarrow ID_6$	8.2	Execute neutral "read string" function.
$ID_6 \Rightarrow ID_7$	8.1	Execute active "print string" function.
$ID_7 \Rightarrow ID_8$	8.3	Delete excess close paren entered into ID_4 by the active "read string" function. The next step is rule 4.1.

† The initial 4. has been deleted from each of the rule numbers in this column.

Table II. Details of the processing steps utilized in the example of table I.

6. BOUNDED CAONTEXT ANALYSIS OF THE TYPE-T RULES

Some of the rules of Section 4. require a scan over string of unbounded length to complete the evaluation of a rule. However such unbounded scans are not necessary to determine which rule of the set will succeed and hence is to be applied next. The deterministic logic necessary to achieve this decision requires at most the examination of three characters† to the right of the <scan pointer>. A summary of this analysis is given as an extended entry decision table in Table III. The corresponding actions appear in Table IV. This decision table is converted into one possible flow chart in figure 12. The use of a horizontal format for the rules in the decision table allows the three characters under scan to be viewed in their natural positions, i.e. horizontal and left to right.

---

† This is known as a bounded context scan and in the notation of Knuth is LR(3). See [9] sections II.C1 and II.C3 for the required definitions.

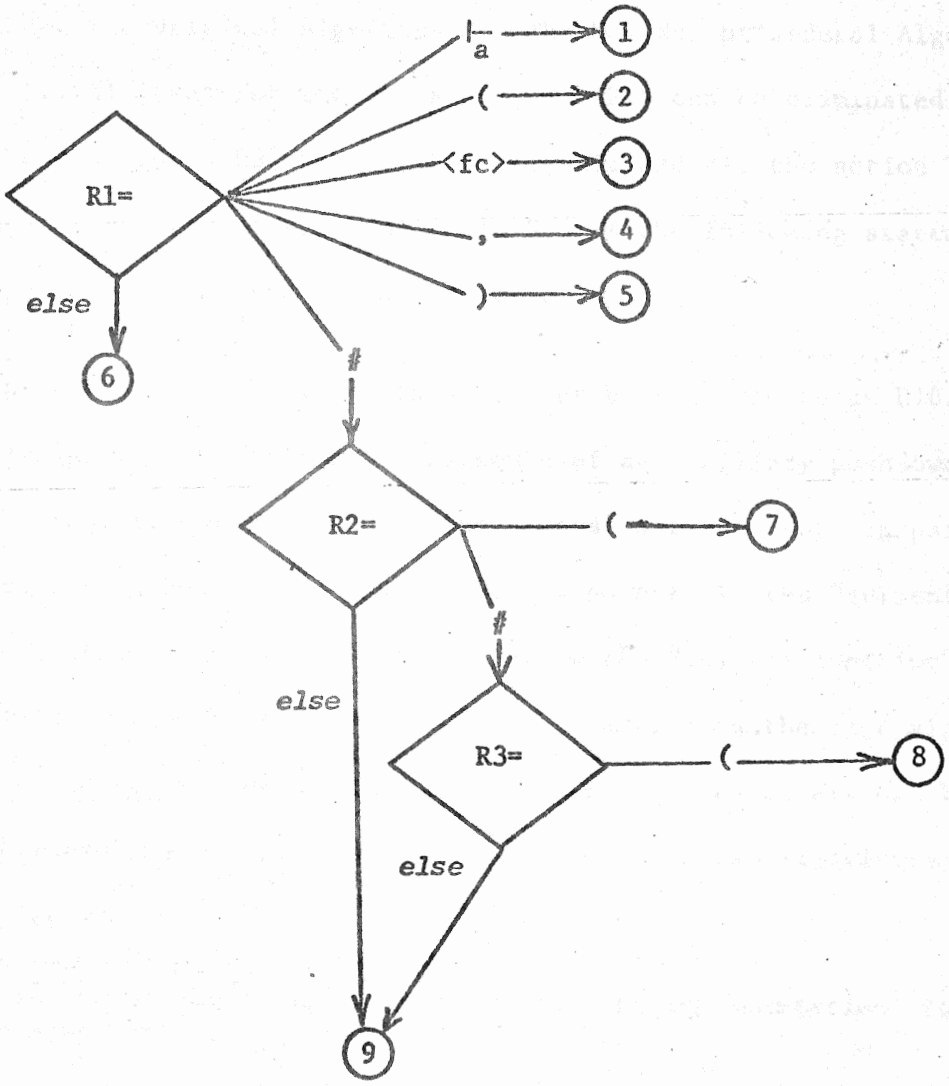
Conditions			Actions
R1	R2	R3	
$=\|_{a}$			1
=(			2
=<fc>			3
=#	=(		4
=#	=#	=(	5
=#	≠(		6
=#	≠#		6
=#	=#	≠(	6
=,			7
=)			8
=<tc>			9

<fc>= <format character>.
<tc>= <text character>.
R1, R2 and R3 stand for the first, second and third characters to the right of the scan pointer.

Table III. Extended entry decision table for bounded context rules in horizontal format.

Action	Definition	Rule
1	Re-initialize to the idling procedure.	4.1
2	Remove the protecting parens from a <balanced string> and place the scan pointer immediately to the right of that <balanced string>; if the parse discovers <unbalanced string> execute re-initialization.	4.2
3	Delete unprotected <format character>.	4.3
4	Mark the open paren of an active function and step past it while deleting the #.	4.4
5	Mark the open paren of a neutral function and step past it while deleting the ##.	4.5
6	Step past a # which is not followed by (or by #(.	4.6
7	Mark a comma as an argument separator and step past it.	4.7
8	Execute a leftward scan which will determine which of the following actions is to be taken:	
	(a) Evaluate an active function.	4.8.1
	(b) Evaluate a neutral function.	4.8.2
	(c) Delete an excess close paren.	4.8.3
9	Step past a <text character>.	4.9

Table IV. Details of actions referred to in Table III.



<fc> stands for <format character>

Bounded context schema = [] R1 R2 R3

(x) indicates "action x".

Fig. 12. Flowchart for bounded context decision logic.

A P P E N D I X I

Relationship Between The Original Algorithm And The New Non-procedural Algorithm.

Let Rn (for n=1,...,16) stand for the rules of [1]. R15 can be eliminated by adding to the end of rules R3, R4, R7, R9, R11, R12 and R13 the action specified by R15. In [1] immediately following R15 is the following statement:

> "Extra close parens are ignored and are deleted at the end of a procedure."

For the purpose of this appendix, let this statement be referred to as R16.

In [1] there is informally implied the presence of an auxiliary pushdown used to record and to retrieve pointers to the neutral string. Note in particular the statements in R4, R5, R6 and R8 about a pointer to the "current location" and the statement in R8 about a pointer to the "current function"†. This implied auxiliary pushdown has been replaced by adding to the tape alphabet the marked characters $\overline{(}$ and $\overline{\overline{(}}$ and $\overline{,}$. These marked characters can be retrieved by a leftward scan. The relationship between the two algorithms is summarized in Table V.

† It was not the intent of [1] to specify the details of implementation for a TRAC language processor.

New Rule	Context	Old Rule	Relation†
2.1	re-initialize	R1,R14	D
2.2	□(	R2	I.1
2.3	□<format character>	R3	D
2.4	□#(	R5	D
2.5	□##(	R6	D
2.6	□# [not R5,R6]	R7	D
2.7	□,	R4	D
2.8	□)	R8,R10,R13	I.3
2.8.1	$\overline{(}$ ...□)	R11	D
2.8.2	$\overline{\overline{(}}$ ...□)	R12	I.3
2.8.3	$\overline{|}_n$ ...□)	R16	I.2
2.9	otherwise	R9	D

† D = there exists an obvious and direct relation between the rules in the two systems.
I.x = see this appendix at Section I.x for comments on this case.

Table V. Relationship between new rules and old rules.

I.1 Balanced strings.

In R2 no provision was made for the case where the expected close paren is omitted. Rule 2.2.2 makes explicit provision for this situation.

I.2 Close paren.

In rule 2.8 a leftward scan for $\overline{(}$ or $\overline{\overline{(}}$ or $\overline{\underline{n}|}$ replaces the "retrieve pointer" operation of R8. In the non-procedural algorithm, the case of $\overline{\underline{n}|}$ is handled by 2.8.3 while in [1] it is defined by R16. See also footnote to 2.8.3, in this paper.

I.3 Neutral evaluation.

The rule 2.8.2 is so stated as to take into account a special case which occurs when a neutral function returns a <default value>. This case is covered in [1] by the non-procedural statement:

> "The overflow value (in this paper called <default value>) is always treated as if it were produced by an active function."

This is found in [1] on page 218 under "Arithmetic Functions". The new rule 2.8.2 provides for this case directly.

ACKNOWLEDGMENT

The authors wish to thank Mr. P. Hess of Western Electric Co., Princeton for his constructive comments and in particular for pointing out an error in the processing of # in an earlier formulation of the scan algorithm.

REFERENCES

[1] Mooers, C. N. TRAC, a procedure-describing language for the reactive typewriter. *Comm. ACM 9* (Mar. 1966),215-219.

[2] ————. How some fundamental problems are treated in the design of the TRAC language. *Symbol Manipulation Languages and Techniques*, Bobrow (Ed.). North-Holland Pub. Co., Amsterdam, 1968, pp. 178-190.

[3] Wilkes, M. V. The outer and inner syntax of a programming language. *The Computer Journal 11* (Nov. 1968), 260-263.

[4] Bandat, K. On the formal definition of PL/I. *Proc. AFIPS 1968 SJCC, Vol 32*, pp. 363-373.

[5] Griffiths, T. V. and Petrick, S. R. On the relative efficiencies of context-free grammar recognizers. *Comm. ACM 8* (May 1965), 289-300.

[6] ————. Top-down versus bottom-up analysis. *Proc. IFIP Congress*, Edinburgh, 1968, pp. B80-B85, in Software I.

[7] Williams, R. A concise notation for the TRAC scanning algorithm. *SIGPLAN Notices, ACM* (July-Aug. 1968), 31-32.

[8] Hopcroft, J. E., and Ullman, J. D. *Formal Languages and their Relation to Automata.* Addison-Wesley, Reading, Mass., 1969.

[9] Feldman, J., and Gries, D. Translator writing systems. *Comm. ACM 11* (Feb. 1968), 77-113.

[10] Whitney, G. E. An extended BNF for specifying the syntax of declarations. *Proc. AFIPS 1968 SJCC, Vol 34.*