American Journal of Computational Linguistics *Microfiche 37 : 28*

P L A T O N -- A New Programming Language for Natural Language Analysis

MAKOTO NAGAO AND JUN-ICHI TSUJII

Department of Electrical Engineering
Kyoto University
Kyoto, Japan

ABSTRACT

PLATON (Programming LAnguage for Tree OperatioN) - ne facilities of pattern matching and flexible backtracking, l. The language is developed to simplify writing analysis programs l language The pattern matching process has the facility to extract sub- from the input sentence and invoke semantic and contextual checking fu Int actions between syntactic and other components are easily obt =. If processing results in a failure, a message which expresses t: se of failure will be sent up. The control will be modified accord Thi enables us to write fairly complicated non-deterministic prog in manner. An example of structural analysis using PLATON is als ed

I Introduction

In this paper we describe a new programming language which is designed to facilitate the writing of natural language grammars. A simple structural analysis program using this language is given as an example. There are two key issues in analyzing natural language by computer: 1) how to represent knowledge (semantics, pragmatics) and the state (context) of the world, and 2) how to advance the programming technology appropriate for syntactic-semantic, syntactic-contextual interface. The point in designing a programming language is to make this kind of programming less painful.

Traditional systems which represent grammars as a set of rewriting rules usually have poor control mechanisms, and flexible interaction between the syntactic and other components is not possible. Systems in which rules of grammars are embedded in procedures, on the other hand, make it possible to intermix the syntactic and semantic analyses in an intimate way. However, these systems are apt to destroy the intelligibility and regularity of natural language grammars, because in these systems both rules and their control mechanisms are contained in the same program.

Recently various systems for natural language analysis have been developed T. Winograd's (1971) "PROGRAMMAR" is a typical example of procedure oriented systems. In this system the syntactic and other components can interact closely in the course of analyzing sentences. However, details of the program are lost in the richness of this interaction. LINGOL, developed by V. Pratt (1973) at MIT, is a language appropriate to syntax-semantics interface and in which it is easy to write grammars in the form of rewriting rules. The TAUM group at Montreal University (1971) has evolved a programming language named System-Q in which expressions of trees, strings and lists of them can be matched against partial expressions (structural patterns) containing variables and can be transformed in any arbitrary fashion.

The augmented transition network (ATN) proposed by W Woods (1970) from our point of view gives an especially good framework for natural language analysis systems. One of the most attractive features is the clear discrimination between grammatical rules and the control mechanism. This enables us to develop the model by adding various facilities to its control mechanism.

The ATN model has the following additional merits:

1. It provides power of expression equivalent to transformational grammars.

2. It maintains much of the readability of context-free grammars.

3. Rules of a grammar can be changed easily, so we can improve them through a trial-and-error process while writing the grammar.

4. It is possible to impose various types of semantic and pragmatic conditions on the branches between states. By doing this, close interactions between the syntactic and other components can be easily accomplished.

However ATN has the following shortcomings, especially when we apply it to the parsing of Japanese sentences:

1. It scans words one-by-one from the leftmost end of an input sentence, checks the applicability of a rule, and makes the transition from one state to another. This method may be well suited for English sentences, but because the order of words and phrases in Japanese sentences is relatively free, it is preferable to check the applicability of a rule by a flexible pattern-matching method. In addition, without a pattern-matching mechanism, a single rewriting rule of an ordinary grammar is often to be expressed by several rules belonging to different states in Woods ATN parser.

2. An ATN model essentially performs a kind of top-down analysis of sentences. Therefore recovery from failures in prediction is most difficult.

Considering these factors, we developed PLATON (a Programming LAnguage for Tree-OperatioN), which is based on the ATN model and has various additional capabilities such as pattern-matching, flexible backtracking, and so on. As in System-Q and LINGOL, PLATON's pattern-matching facility makes it easy to write rewriting rules. Moreover, it extracts substructures from the inputs and invokes appropriate semantic and contextual checking functions.

These may be arbitrary LISP functions defined by the user, the arguments of which are the extracted substructures.

A backtracking mechanism is also necessary for language understanding as in other fields of artificial intelligence. During the analysis, various sorts of heuristic information should be utilizable. At any stage, analysis based on criteria which may relate to syntactic, semantic or contextual considerations taken separately may be unreliable. The result which fulfils all the criteria, however, will be a correct one. The program should be designed such that it can choose the most satisfactory rule from many candidates according to the criteria at hand. In further processing, if the choice is found to be wrong by other criteria, the program must be able to backtrack to the point at which the relevant decision was made. In PLATON we can easily set up arbitrary numbers of decision points in the program. Then, if subsequent processing results in some failure, control will come back to the points relevant to the cause of the failure.

II. Pattern-matching

Before proceeding to the detailed description of PLATON, we will explain the representation schema for input sentences and parsed trees. The process of analyzing a sentence, roughly speaking, may be regarded as the process of transforming an ordered list of words to a tree structure, which shows explicitly the interrelationships of each word in the input sentence. During the process, trees which correspond to the parts already analyzed, and lists which have not been processed yet, may coexist together in a single structure. We therefore wish to represent such a coexisting structure of trees and lists. A list structure is a structure in which the order of element is not changeable. On the other hand, a tree structure consists of a single

root node and several nodes which are tied to the root node by distinguishable relations. Because relations between the root and the other nodes are explicitly specified, the order of nodes in a tree is changeable except for the root node which is placed in the leftmost position. Different matching schemas will be applied to trees and lists.

The formal definition of such coexisting structures is as follows. <structure> is the fundamental data-structure into which all data processed by PLATON must be transformed. Hereafter we refer to this as the "structure" The formal definition of <structure> is:

<structure> ::= <tree> | <list>

<list> ::= (* <structures>)

<structures> ::= |<structure> < structures >

<tree> ::= < node> | (<node> < branches>)

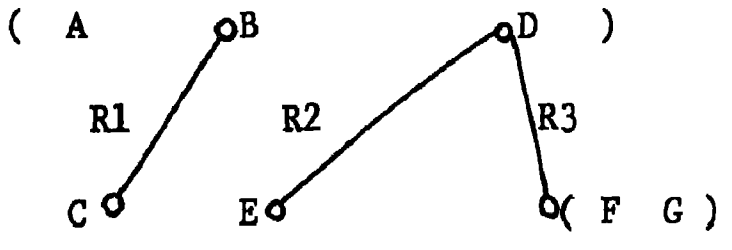
<branches> ::= <branch> | < branch> < branches >

<branch> ::= (< relation> <tree>)

< node> ::= <list> | ARBITRARY LISP-ATOM

< relation > ::= ARBITRARY LISP-ATOM

A simple example is shown in Figure 1.



Coexisting Structure of Trees and Lists

(* A (B ,(R1 C)) (D (R2 E)
(R3 (* F G))))

Corresponding Expression in PLATON

Figure 1 Expression of Structure in PLATON

Two lists which have the same elements but different orderings (for example, (* A B C) and (* A C B)), should be regarded as different structures. On the other hand, two tree structures such as (A (R1 B) (R2 C)) and (A (R2 C)(R1 B)) are regarded as identical. Besides the usual rewrite rules which treat such strings, structural patterns which contain variable expressions are permitted in PLATON. The PLATON-interpreter matches structural patterns containing variable expressions against the structure under process and checks whether the specified pattern is found in it. At the same time, the variables in the pattern are bound to the corresponding substructures.

Variables in patterns are indicated as :X (X is an arbitrary LISP atom). The following can be expressed by variables in the above definition of <structure> .

(1) arbitrary numbers of <structures> , that is to say, list elements in the definition of <list> (Figure 2, Ex. 1). We can also specify the

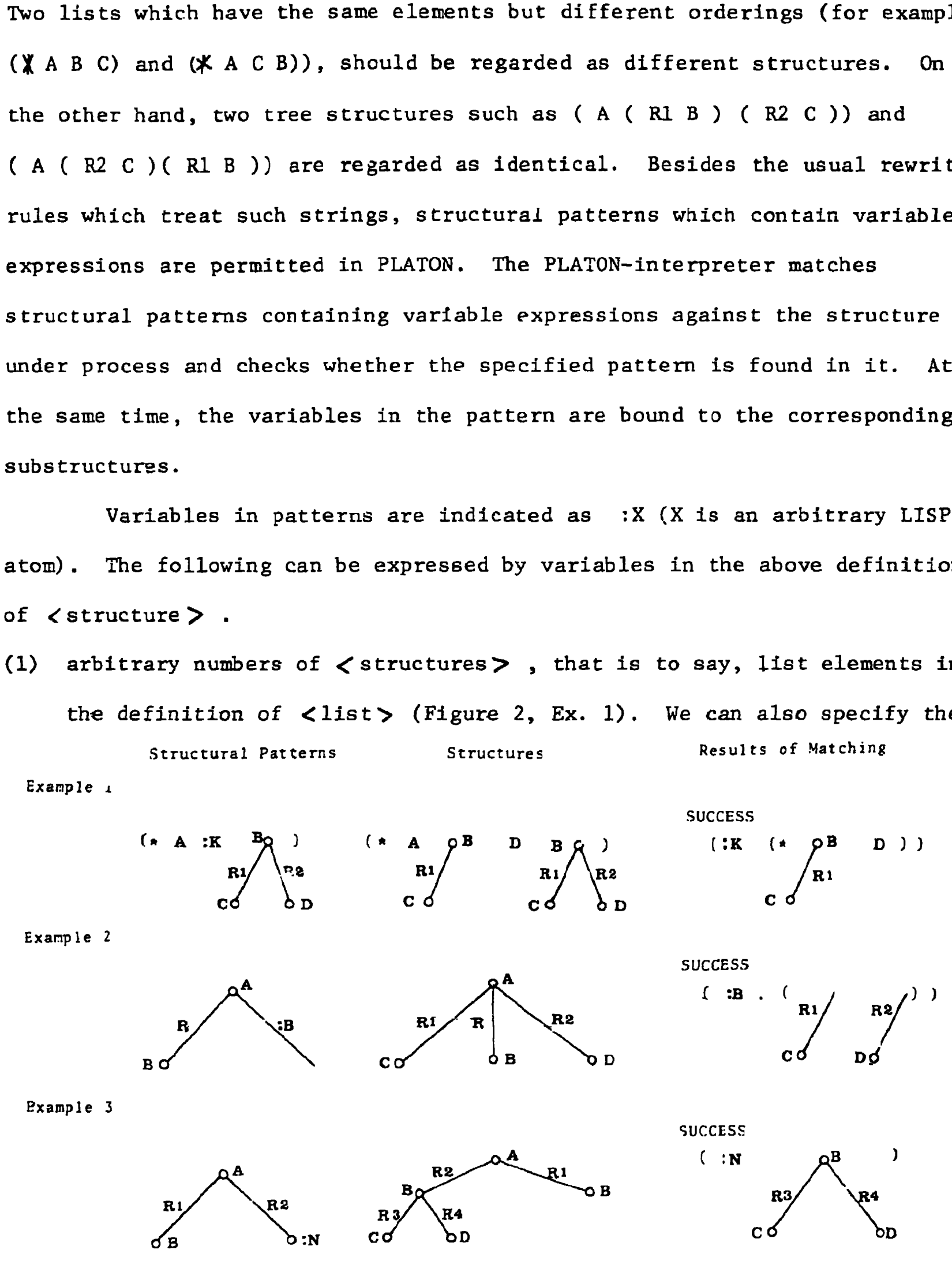


Figure 2 Illustration of Matching

number of list elements by indicating variables as :X+number. For example, the variable :D2 will match with two elements in a list.

(2) arbitrary numbers of <branches> , in the definition of <tree> (Figure 2, Ex. 2).

(3) <tree> in the definition of <branch> (Figure 2, Ex. 3).

We shall call such structural patterns <structure-1 > . By using the same variable several times in a pattern, we can express a structure in which the same sub-structure appears in two or more different places. The character '!' in a list indicates that the next element following the character is optional.

III Basic Operations of PLATON

A grammar, whether generative or analytical, is represented as a directed graph with labeled states and branches. There is one state distinguished as the Start State and a set of states called Final States. Each branch is a rewriting rule and has the following elements:

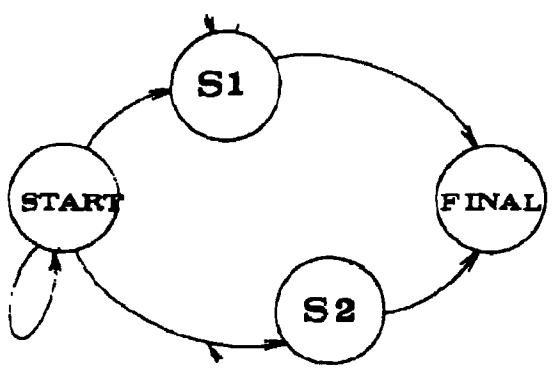
(1) applicability conditions of the rule, typically represented as a structural pattern

(2) actions which must be executed, if the rule is applicable

(3) a structural pattern into which the input structure should be transformed.

Each state has several branches ordered according to the preference of the rules. When the control jumps to a state, it checks the rules associated with the state one-by-one until it finds an applicable rule. If such a rule is found, the input structure is transformed into another structure specified by the rule and the control makes the state transition.

In addition to the above basic mechanism the system is provided with push-down and pop-up operations. The push-down operation is such that in the process of applying a rule, several substructures are extracted from the whole structure by variable binding mechanisms of pattern-matching. Then each is analyzed from a different state. The pop-up operation is such that after each substructure is analyzed appropriately, control comes back to the suspended rule and execution continues. Using these operations, embedded structures can be handled easily (See Figure 3).



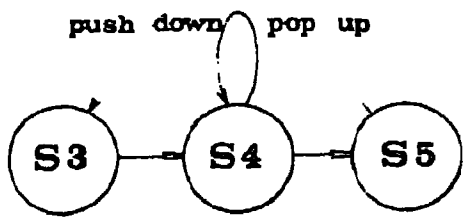


Figure 3 State Diagram

Table 1 shows the formal definition of a grammar of PLATON (See following page). It shows that branches or rewriting rules in an ATN parser correspond to six-tuples (i.e., <pcon> , <strx> , <con> , (<trans>), (<acts>), <end>). <strx> corresponds to the left side of a rewriting rule and describes the structural pattern to which a rule is applicable. <strx> is, by definition

(1) / or

(2) structure-1

TABLE 1 Formal Definition of Grammar in PLATON

```
<grammar>  :: = ( <states> )

<states>   :: = <state> | <state> <states>

<state>    :: = ( <state-name>  <rules> )

<rules>    :: = |<rule> <rules>

<rule>     :: = ( <pcon> <strx> <con> ( <trans> )( <acts> ) <end> )

<trans>    :: = | <transit><trans>

<transit>  :: = (( <state-name> <structure-2> { <register-name> | <variable-name> }) <errorps>

<errorps>  :: = | <errorp> <errorps>

<errorp>   :: = ( <failure-message> <act> <pros> )

<pros>     :: = <pro> | <pro> <pros>

<pro>      :: = (EXEC <trans> ) |(TRANS ( <state-name> <stry> ))

<end>      :: = (NEXT <state-name> <stry> )
                |(NEXTB <state-name> <stry> )
                |(POP <stry> ) | (FM <failure-message> )

<acts>     :: = | <act><acts>

<act>      :: = <form> | (SR <register-name> <form> )
                | (SU <register-name> <form> )
                |(SD <register-name> <form> )

<strx>     :: = <structure-1> | /

<stry>     :: = <structure-2> | /

<pcon>,<con>:: = <form>

<form>     :: = (GR <register-name> ) | (GV <variable- name> )
              |(TR <structure-2> ) |(TR /) | ARBITRARY LISP FORM

<variable- :: = :X (X is an arbitrary LISP atom)
 name>

<register- :: = /X (X is an arbitrary LISP atom)
 name>
```

shows that a rule is applicable no matter what the structure under process is The variables used in < structure-1 > are bound to corresponding substructures when matching succeeds. The results of Example 1 (See Figure 2) indicate that the variable :K is bound to the substructure (* (B (R1 C)) D)

The scope of variable binding is limited to within the realm of the particular rule. The same variable name in different rules has different interpretations. In this sense, :X-type variables in <structure-1> are called Local Variables. On the other hand, we can store certain kinds of results from the application of rules in registers and refer back to them in different rules. These constitue variables which we call registers. They are represented by the symbols /X (X is an arbitrary LISP atom).

Besides the pattern-matching, <pcon> and <con> can also check the applicability of a rule. Certain parts of the results from the application of previous rules are contained in registers, not in the structure. We can check the contents of these registers by using <pcon>-part functions like GR, GU, etc. (these functions are listed in Table 2) and other LISP functions defined by the usual LISP function, DEFINE. (See following page for Table 2.)

Semantic and contextual co-ordination between substructures can be checked by using appropriate functions in the <con>-part of a rule. Semantic and contextual analyses cannot be expressed in the form of simple rewriting rules. These analyses have differing requirements such as lexical information about words which may in turn represent knowledge of the world and contextual information which may express the state of the world. We can use arbitrary LISP-forms in the <con>-part, according to what semantic and contextual models we choose.

TABLE 2 Functions of PLATON

Function	Argument	Effect	Value
SR	<register-name> LISP - <form>	SR stores the result of the evaluation of the 2nd argument in the register.	the result of the evaluation of the 2nd argument
SV	<variable-name> LISP - <form>	SV stores the result of the evaluation of the 2nd argument in the variable	the result of the evaluation of the 2nd argument
GR	<register-name>	GR get the content of the register	the content of the register
GV	<variable-name>	GV gets the value of the variable	the value of the variable
TR	<structure-2> or /	TR transforms the variables and registers in the structural pattern into their values.	the transformed structure
SU	<register-name> LISP - <form>	SU sets the reigster of the higher level processing	the result of the evaluation of the 2nd argument
SD	<register-name> LISP - <form>	SD sets the register of the lower level processing.	the result of the evaluation of the 2nd argument
GU	<register-name>	GU gets the content of the register of the higher level.	the content of the register
PUSHR	<register-name> LISP - <form>	PUSHR is defined as the following. (SR <register-name> (CONS <form> (GR <register-name>)))	the result of the evaluation of the 2nd argument

For example, suppose

strx = (ADJ (TOK :N))(N(TOK :N1)) :I)

con = (SEM :N :N1)

Here TOK is the link between a word and its part of speech. :N and :N1 are the words of an input sentence. SEM is a function defined by the user which checks the semantic co-ordination between the adjective :N and the noun :N1. By this function SEM, we can search, if necessary, through both lexical entries and the contextual data bases.

With this approach, if a certain syntactic pattern is found in the input structure, it is possible for an appropriate semantic function to be called. Hence the intimate interactions between syntactic and semantic components can be obtained easily without destroying the clarity of natural language grammars.

Arbitrary LISP-forms can be also used in < act > -portion. They will be evaluated when the rule is applied. If necessary, we can set intermediate results into registers and variables by using the functions listed in Table 2

< end > comprises four varieties, and rules are divided into four types according to their < end > types.

1. NEXT-type: The < end > is in the form (NEXT <state-name> <stry>). The <stry> corresponds to the right side of a rewriting rule, and represents the transformed structure. A rule of this type causes state-transition to the <state-name> , when it is applied.
2. NEXTB-type: This rule also causes state-transition. Unlike with the NEXT-type, state-saving is done and if further processing results in some failures, control comes back to the state where this rule is applied. The environments, that is, the contents of various registers will be restored, and the next rule belonging to this state will be tried

3. POP-type: The <end> -part of this type is in the form (POP <stry>) When it is applied, the processing of this level is ended and the control returns to the higher level with the value <stry> .
4. FM-type: The <end >-part of this type is in the form (FM <failure-message >). The side effects of the processing at this level, that is, register settings and so on, are cancelled (see section 4).

In <stry > we can use two kinds of variables, that is, the variables used in <strx > and registers. We find this structural pattern, called <structure-2 > , more suitable for writing transformational rules than Woods BUILDQ-operation. By way of illustration consider the following:

input string	= (* C D E (A (R1 (* B))) F G)
strx	= (* :I (A (R1 :N)) :J)
stry	= (* (A (R1 (* :I :N))(R2 /REG)) :J)
the content of /REG	= (G (R3 H))

As the result of matching, the variables :I, :N and :J are bound to the substructures (* C D E), (* B) and (* F G) respectively. The result of evaluating the < stry > is

(* (A (R1 (* C D E B))(R2 (G (R3 H)))) F G).

If the rule is a POP-type one, then this structure will be returned to the higher level processing. If it is NEXT- or NEXTB-type, then the control will move to the specified state with this structure.

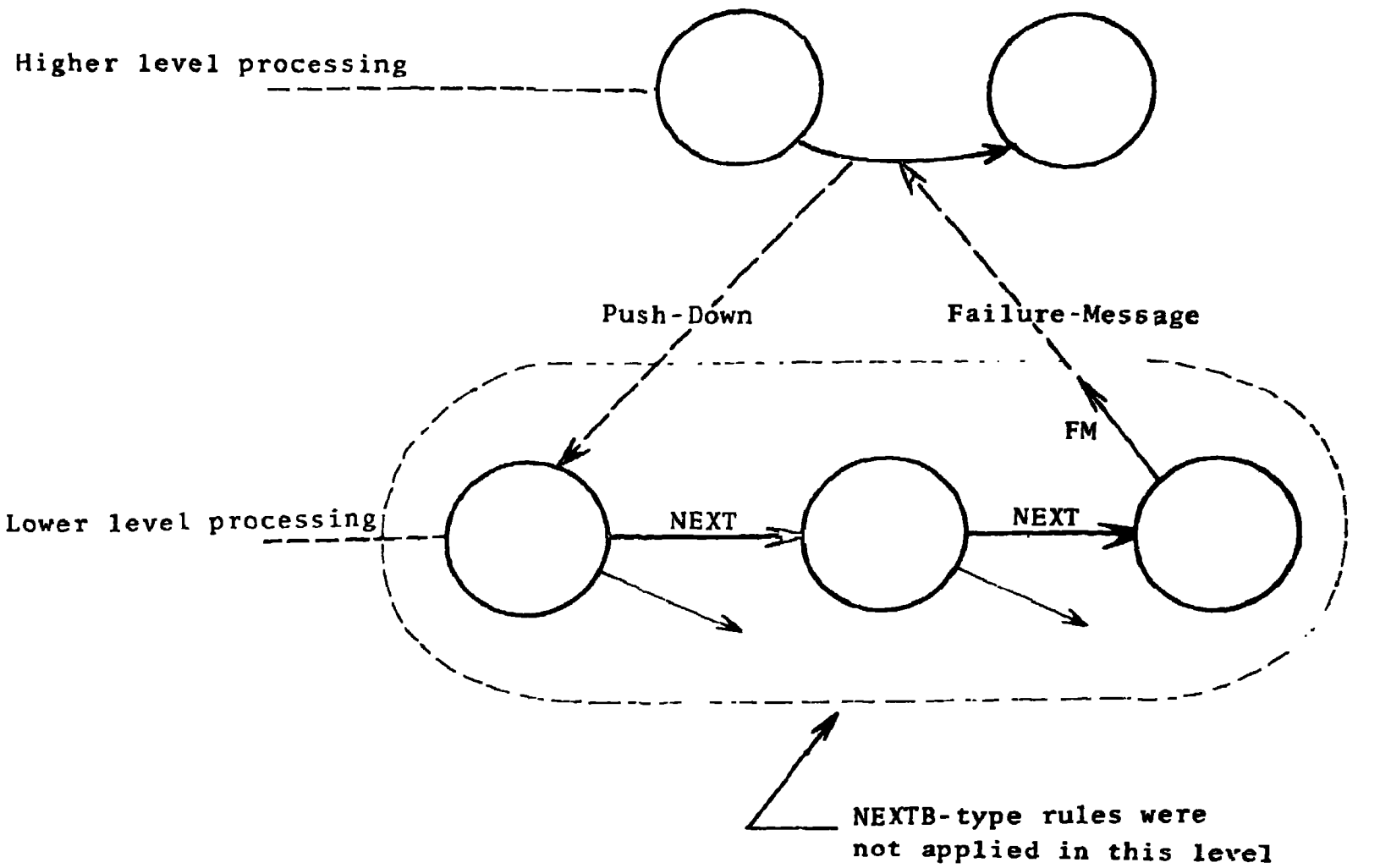
IV Push-down and Pop-up Operations

By means of NEXTB-type rules, we can set up decision points in a program. We can also do this by using push-down and pop-up operations. A rule in PLATON finds particular syntactic clues by its structural description

< strx > ; and at the same time, extracts substructures from the input string From the structural description it is predicted that the substructures may have particular constructions, that is, they may comprise noun phrases, relative clauses or whatever. It is necessary to transfer the substructures to states appropriate for analyzing these constructions predicted and to return the analyzed structures back into the appropriate places In PLATON, these operations can be described in the < trans > -part of a rule. For example, suppose the < trans > -part of a rule is

(((S1 :K :K)) ((S2 (* :I :J)/REG)))

When the control interprets this statement, the substructures corresponding to the variable :K and (* :I :J) are transferred to the states S1 and S2 respectively If the processings starting from these states are normally completed (by a POP-type rule), then the results are stored in the variable :K and the register /REG. In this manner, by means of the push-down and pop-up mechanisms, substructures can be analyzed from appropriate states. Processing from these states, however, may sometimes result in failure. That is, predictions that certain relationships will be found among the elements of substructures may not be fulfilled. In such instances the pushed down state will send an error-message appropriate to the cause of the failure by an FM-type rule. An FM-type rule points out that a certain error has occurred in the processing. If NEXTB-type rules were used in the previous processing at this level, control will go back to the most recently used NEXTB-type rule. If NEXTB-type rules were not used at this processing level, the error-message specified by the FM-type rule will be sent to the < trans > part of the rule which directed this push-down operation (see Figure 4)



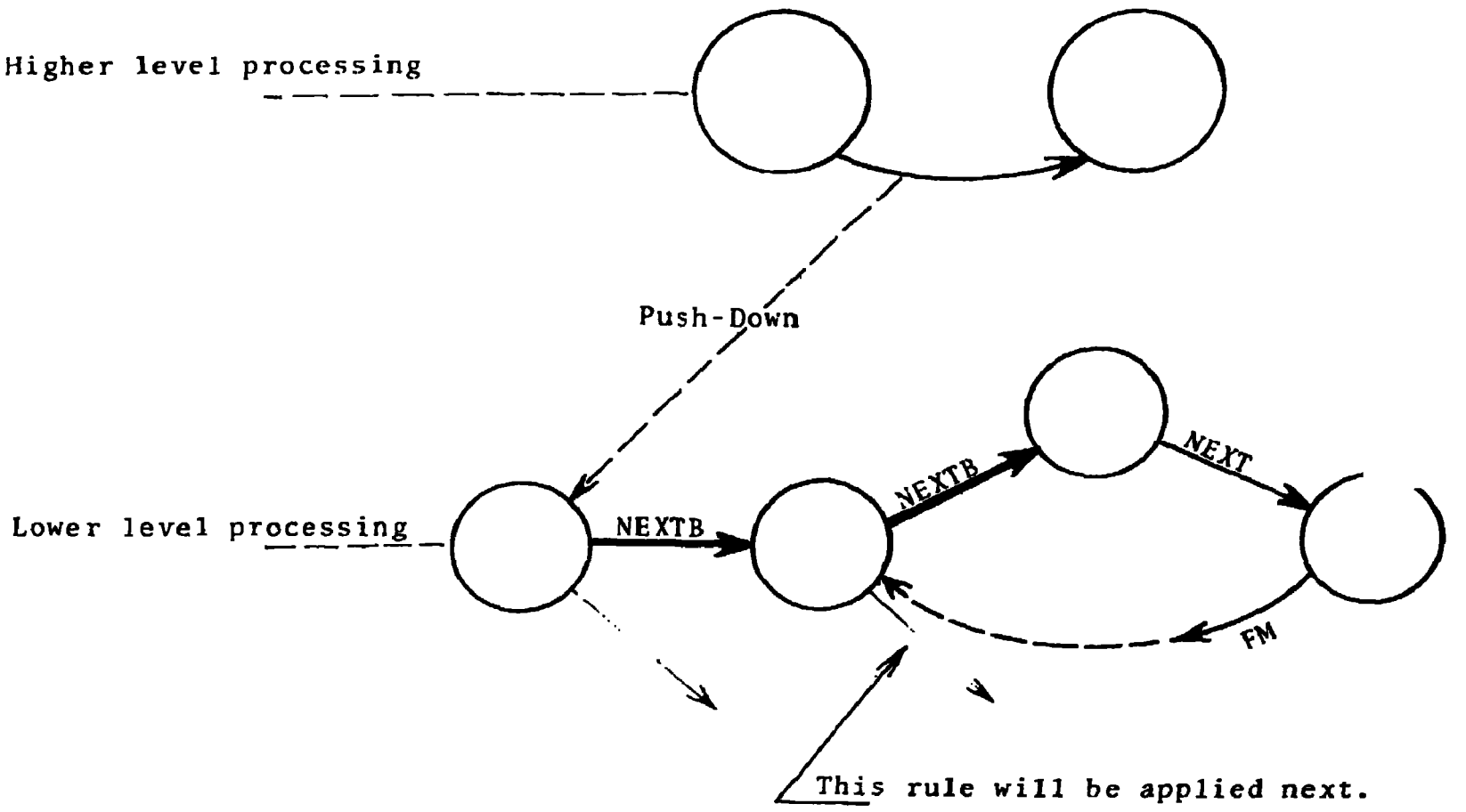


Figure 4 Illustration of Backtracking

According to these error-messages, control-flow can be changed appropriately. For example, we can direct processings by describing the < trans > -part in the following way.

```
( (( S1 :K :K )( ERR1 ( EXEC (( S5 :K :K )) (( S6 (* :I :J ) /REG ))))
                    ( ERR2 (TRANS ( S8 /)))
    ((S2 (* :I :J )  /REG ))  )
```

In the above example, the processing of the substructure :K from the state S1 will produce one of the following three results. According to the returned value, the appropriate step will be taken:

(1) Normal return: the processing of :K is ended by a POP-type rule. The result is stored in the variable :K and the next push-down performed, that is (* :I :J) will be transferred to the state S2.

(2) Return with an error-message: the processing of :K results in a failure and an FM-type rule sends up an error-message. If the message is ERR1, then :K and (* :I :J) will be analyzed from the states S5 and S6 respectively (EXEC-type). If it is ERR2, the interpreter will give up the application of the present rule, and pass the control to another state S8 (TRANS-type). If it is neither ERR1 nor ERR2, the same step as (3) will be taken

(3) Return with the value NIL: the processing from the state S1 will send up the value NIL if it runs into a blind alley, that is, there are no applicable rules. The interpreter will give up the application of the present rule and proceed to the next rule attached to this state.

Mechanisms, such that control flow can be appropriately changed according to the error-messages from lower level processings are not found in Woods ATN parser. We can obtain flexible backtracking facilities by combining these mechanisms with NEXTB-type rules.

V A Simple Example

We are now developing a deductive question-answering system with natural language inputs -- Japanese sentences. The internal data-base is assumed to be a set of deep case structures of input sentences. We adopted and modified Fillmore's (1968) case grammar to analyze the input of Japanese sentences. Japanese is a typical example of an SOV-language in which the object and other constituents governed by a verb usually appear before the verb in a sentence. A typical construction of a Japanese sentence is shown in Figure 5.

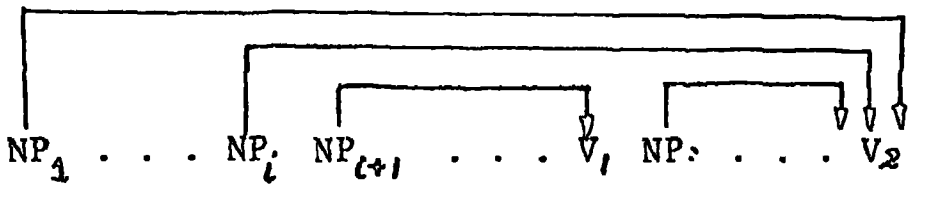


Figure 5 Typical Construction of a Japanese Sentence

A verb may govern several noun phrases preceding it. A relative clause modifying a noun may appear in the form -- verb + noun -- The right boundary of the clause is easily identified by finding the verb. The left boundary is often much more difficult to identify. In Figure 5, the noun phrase NP_{i+1} is a case element of the verb V On the other hand, the noun phrase NP_i is governed by the verb V_2 Because the rule of projections holds in Japanese as in other languages, all the noun phrases between NP_{i+1} and V are governed by V , and the noun phrases before NP_i are governed by V_2 However, in the course of analysis, such boundaries cannot be determined uniquely. The analysis program fixes a temporary boundary and proceeds to the next step in processing If the temporary boundary is not correct, the succeeding processing will fail and the control wil come back to the point

at which the temporary boundary was fixed.

Now we will show a simple example of structural analysis by PLATON The example explains how the backtracking facility is used in analyzing Japanese sentences. Because we want to visualize the operations of PLATON without bothering with microscopic details of Japanese sentences, we will take an imaginary problem as an example. The parser which is written in PLATON is described in another paper by M Nagao and J. Tsujii (1976)

An input string is assumed to be a list. The elements of the list are integers and trees are in the form of (X (SUM 0)). Here 'X' may be regarded as a term modified by 'SUM 0' These two kinds of elements are arranged in an arbitrary order, except that the last element is the tree (X(SUM 0)). The following is an example of an input string:

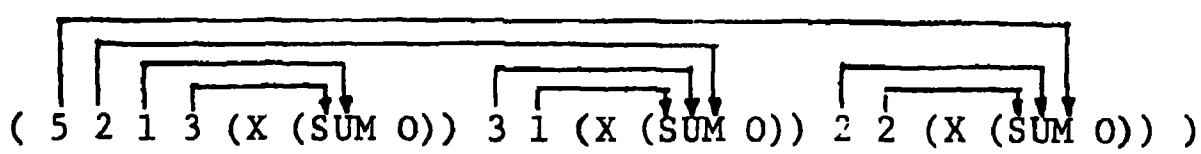
(5 2 1 3 (X (SUM 0)) 3 1 (X (SUM 0)) 2 2 (X (SUM 0)))

Figure 6 An Example String to be Analyzed

The result of the transformation is expected to be in the following form:

((X (SUM 4)) (X SUM 6)) (X (SUM 9)))

This result is regarded as representing the following relationships between integers and 'X'.



The number associated with an 'X' by the relation 'SUM' shows the sum of the integers which are governed by the 'X'. We can look upon the relations between integers and an 'X' as the relations between noun phrases and the verb in Japanese sentences. The result of the analysis is assumed to satisfy the following conditions.

(1) Governor-governed relationships between integers and an 'X' must obey the projection rule (i.e., clauses do not overlap).

(2) As a simulation of a semantic restriction, we attach a condition that the sum of the integers governed by an 'X' should not exceed ten.

(3) As a simulation of a contextual restriction, we attach the condition that a result (* (X (SUM num-1)) (X (SUM num-2)) . . . . (X (SUM num-N))) should maintain the relation, num-1 $\leqq$ num-2 $\leqq$ . . . . $\leqq$ num-N.

A set of rules is shown in the following. The corresponding state-diagram is shown in Figure 7.

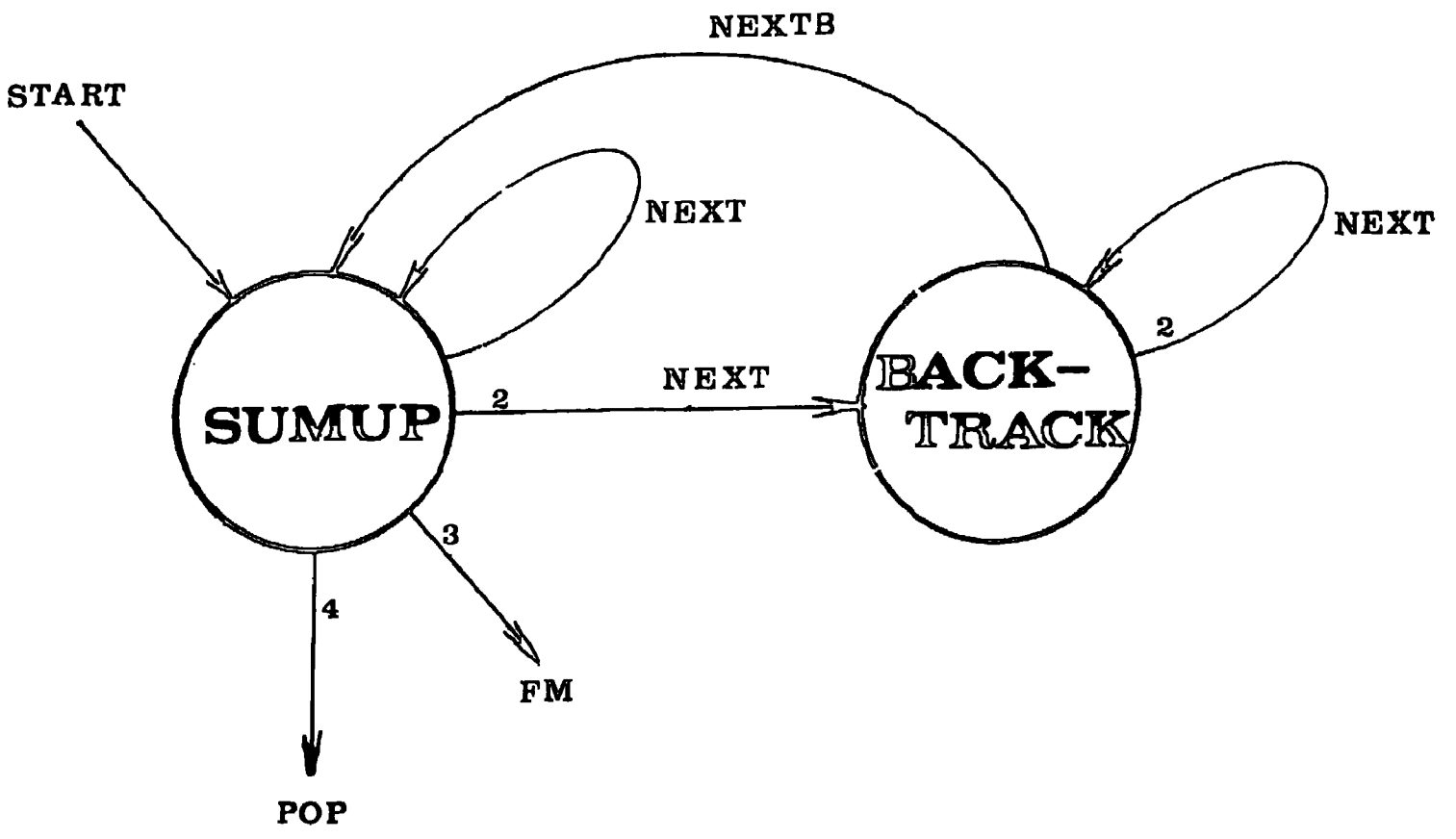


Figure 7 State Diagram of a Simple Example

```
SUMUP -1-  strx: = (*  :I :I1 (X (SUM :N)) :J)
           con:  = ( GREATERP 10 (PLUS :N :I1))
           act: = ( (SV :N (PLUS :N :I1 ))
                    (PUSHR  /REG :I1) )
           end: = (NEXT SUMUP (*  :I (X (SUM :N)) :J) )
      -2-  strx : = (*  :I (X (SUM :N)) :J)
           con: = (CONTEXTCHECK /RESULT (TR (X (SUM :N))))
           act: = NIL
           end: = (NEXT BACKTRACK /)
      -3-  strx: = (*  :I (X (SUM :N)) :J)
           con: = T
           act: = NIL
           end: = (FM-ERROR)
      -4-  strx: = (* )
           con: = T
           act: = ( (SR /RESULT (CONS 'X /RESULT )) )
           end: = (POP /RESULT)
BACKTRACK
      -1-  strx: = (*  :I (X (SUM :N)) :J)
           con: = T
           act: = ( (SR /REG NIL)
                    (SR /RESULT (APPEND /RESULT ( TR (X (SUM :N))))) )
           end: = (NEXTB SUMUP (*  :I :J ))
      -2-  strx: = (*  :I (X (SUM :N)) :J)
           con: = T
           act: = ( (POPR /TEMP /REG)
                    (SV :N (MINUS :N /TEMP)) )
           end: = (NEXT BACKTRACK (*  :I /TEMP (X (SUM :N)) :J) )
```

The input string is the list shown in Figure 6. Since the start state is SUMUP, the first rule attached to this state is applied. This rule will find the leftmost 'X' and an integer just before the 'X' (by SUMUP -1-, strx). The variable :I1 is bound to this integer. This integer is added to the sum of the integers, :N, if the total does not exceed ten (SUMUP -1-, con). PUSHR, used in the < act > -part, is a PLATON function which puts the second argument on the head of the first argument (SUMUP -1-, act) After this rule is applied, the control will enter the state SUMUP again (SUMUP -1-, end). That is, this rule is applied until there are no integers before the first 'X' or the sum of the integers exceeds ten. As the result, the environment is the following:

structure under processing

= (* 5 (X (SUM 6)) 3 1 (X (SUM 0)) 2 2 (X (SUM 0)))

relationship temporarily fixed between integers and 'X'

= (5 2 1 3 X 3 1 X 2 2 X)

content of /REG

= (2 1 3)

The second rule of SUMUP will be applied next. This rule checks by its < con> part whether the result at hand satisfies the third condition, that is, the contextual restriction. Because the content of /RESULT is NIL, the function CONTEXTCHECK returns the value T (SUMUP -2-, con). So this rule is applicable. Control makes the state-transition to the state BACKTRACK (SUMUP -2-, end) Because the first rule of BACKTRACK is a NEXTB-type rule, state-saving is performed. That is, the following environment is saved:

content of /REG = (2 1 3)

content of /RESULT = NIL

structure under processing =

(* 5 (X (SUM 6)) 3 1 (X (SUM 0)) 2 2 (X (SUM 0)))

By this rules, the registers /REG and /RESULT are set as follows (BACKTRACK -1-, act).

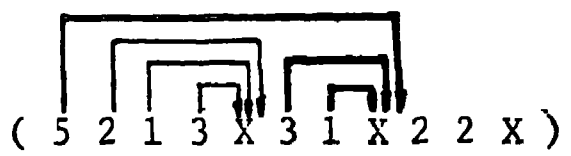
/REG : = NIL

/RESULT: = ((X (SUM 6)))

And the structure is transformed to

(✱ 5 3 1 (X (SUM 0)) 2 2 (X (SUM 0)))

A NEXTB-type rule causes a state transition as does a NEXT-type rule. So control returns to the state SUMUP (BACKTRACK -1-, end). At this state, a process similar to the one described above is performed. As a result, the following governor-dependent relationships are established.



Here the bold lines indicate the newly established relationships. By the first rule of BACKTRACK the following environment is saved.

content of /REG = (5 3 1)

content of /RESULT = ((X (SUM 6)))

structure under processing = (✱ (X (SUM 9)) 2 2 (X (SUM 0)))

And /REG and /RESULT are set as the following (BACKTRACK -1-, act).

/REG: = NIL

/RESULT: = ((X (SUM 6)) (X (SUM 9)))

The transformed structure is (BACKTRACK -1-, end)

(✱ 2 2 (X (SUM 0)))

The control is transferred to the state SUMUP. By applying the first rule of this state repeatedly on the above structure the following structure is obtained.

(✱ (X SUM 4)))

However this result does not satisfy the contextual restriction. So the application of the second rule of SUMUP fails because the function CONTEXTCHECK used in <con> -part returns the value NIL (SUMUP -2-, con) That is:

contextcheck [((X (SUM 6))(X (SUM 9))) : (X (SUM 4))] = NIL

The third rule, therefore, will be applied next. Because this rule is a FM-type rule (SUMUP -3-, end), it causes an error and control comes back to the point at which a NEXTB-type rule was applied most recently. The saved nvironment is restored. This is:

/REG: = (5 3 1)

/RESULT: = ((X (SUM 6)))

structure under processing: = (* (X (SUM 9)) 2 2 (X (SUM 0)))

Then by applying the second rule of BACKTRACK, the governor-governed relationship established lastly in the previous process is cancelled. The structure and the register /REG are changed as below (BACKTRACK -2-, act):

/REG: = (3 1)

structure under processing: (* 5 (X (SUM 4)) 2 2 (X (SUM 0)))

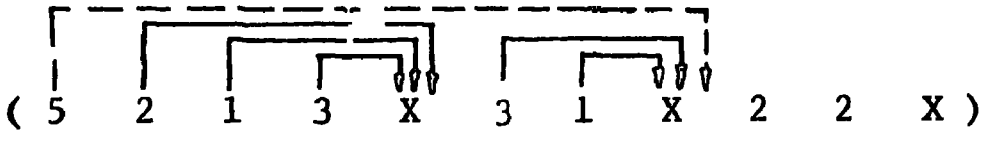
Control enters the BACKTRACK state again. The application of the first rule saves the environment:

content of /REG = (3 1)

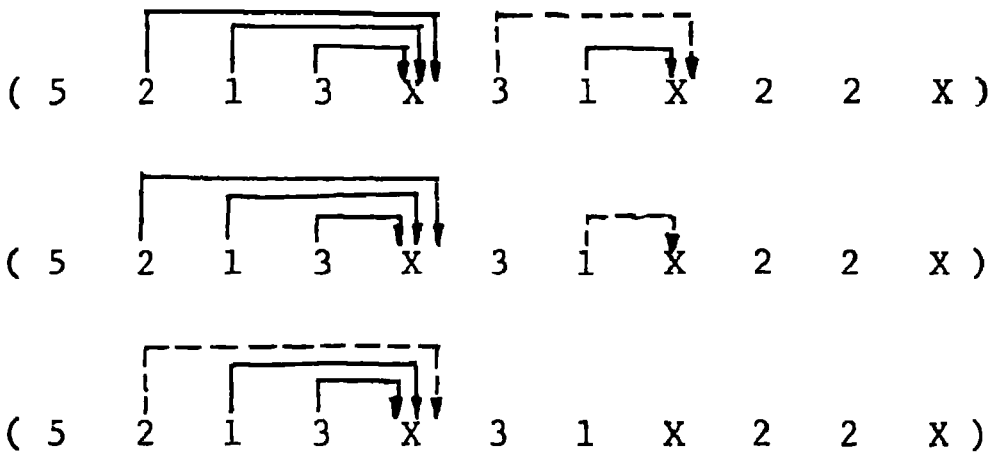
content of /RESULT = ((X (SUM 6))

structure under processing = (* 5 (X (SUM 4)) 2 2 (X (SUM 0)))

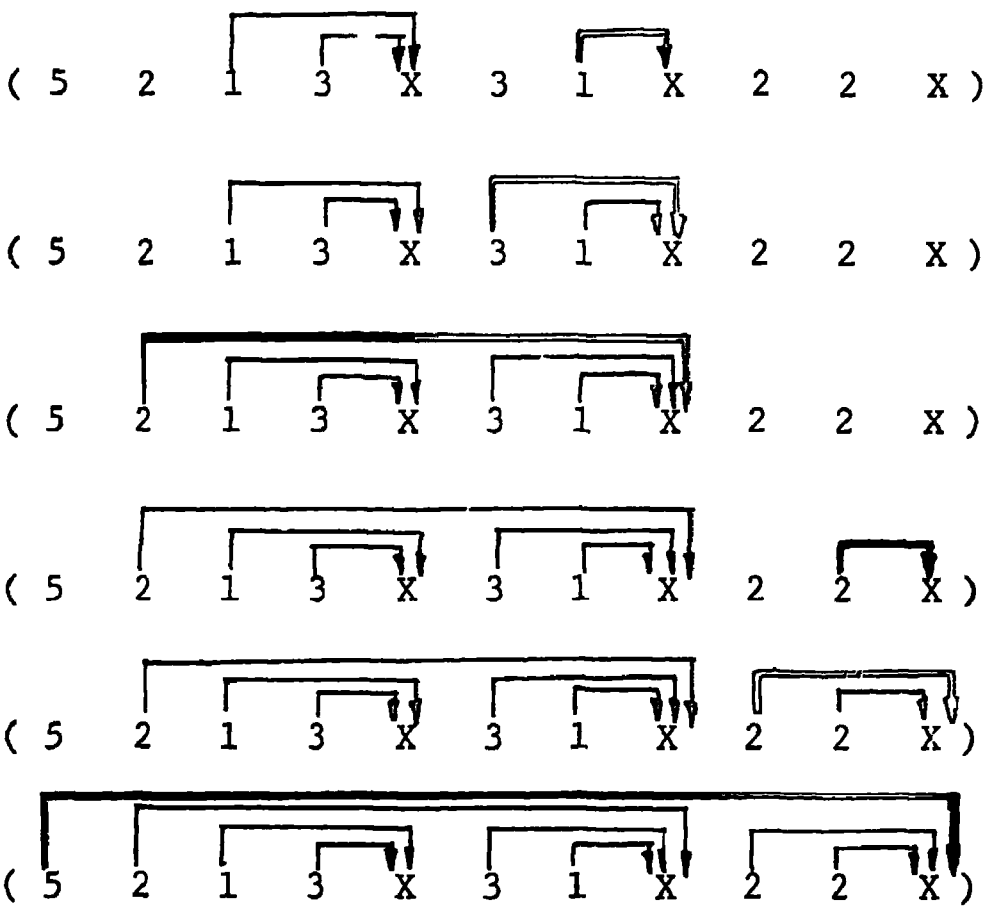
That is, the relationship indicated by the dotted line in the following is cancelled:



Control transits to the state SUMUP (BACKTRACK -1-, end) and a similar process is performed. However, because the governor-governed relationship between the integer 5 and the second 'X' is cancelled, the sum of the integers governed by the first 'X', (2 1 3), is greater than that of the second 'X', (3 1). The contextual condition, therefore, is not fulfilled, and the application of the second rule of SUMUP will not succeed. So the temporarily established relationships will be cancelled one-by-one as follows.



After these relationships have been cancelled, the desired result is obtained by the following sequence.



At the final stage of the processing, the fourth rule of SUMUP a POP-type rule, is applied and returns the value

(* (X (SUM 4)) (X (SUM 6)) (X (SUM 9)))

VI Conclusion

We have described a programming language called PLATON for natural language processing. The language has several additional capabilities beyond the ATN parser of W. Woods.

Grammars written in the language not only maintain clarity of representation but also provide adequately a natural interface between the syntactic component and other components. By means of the pattern-matching facility, we can write grammars in a quite natural manner. And because of the PLATON variable binding mechanism, semantic and contextual LISP functions a easily incorporated in syntactic patterns.

Flexible backtracking mechanisms and push-down operations make complicated non-deterministic processing possible in a very simple way.

We are now developing an analysis program for Japanese using this language. The program can accept fairly complicated sentences in a textbook of elementary chemistry It can utilize the lexical and contextual information of chemistry adequately during the analysis. Such information in our system is expressed in the form of a semantic network similar to that of R. F. Simmons (1973).

Perhaps, PLATON itself must be equipped with more semantics and context-oriented operations such as specified lexical descriptions and functions using them. However, what description method is most efficient, and moreover, what semantic information must be stored in the lexicon, are not yet entirely clear. So, as the first step, PLATON leaves many parts of these problems for

the user to specify by LISP programs. PLATON is written in LISP1.5 and implemented on a FACOM 230-60 at the Kyoto University computing center and a TOSBAC-40 mini-computer in our laboratory. The interpreter of PLATON itself requires only 4.5 K cells.

BIBLIOGRAPHY

D. Bobrow, B. Fraser. "An augmented state transition network analysis procedure", Proc. 1st IJCAI, pp. 557-568, (1969).

A. Colmerauer. "Les systemes-q ou un formalisme pour analyser et synthetiser des phrases sur ordinateur", Project de Traduction automatique de l'Universite de Montreal, TAUM 71, Jan., (1971).

C. J. Fillmore. "The case for case", in Bach and Harms (eds.), Universals in Linguistic Theory, Holt, Rinehart & Winston, pp. 1-90, (1968).

C. Hewitt. "PLANNER: A language for manipulating models and proving theorems in a Robot", in Artificial Intelligence, Washington, D. C , May, (1969)

M. Nagao, J. Tsujii. "Mechanism of deduction in a question answering system with natural language input", Proc. 3rd IJCAI, pp. 285-290, (1973).

M. Nagao, J. Tsujii. "Programming language for natural language processing - PLATON", J.IPSJ, Vol. 15, pp. 654-661, (1974).

M. Nagao, J. Tsujii. "Analysis of Japanese Sentences by Using Semantic and Contextual Information". (forthcoming in AJCL-1976)

V. Pratt. "A linguistic oriented programming language", Proc. 3rd IJCAI, pp. 372-381, (1973).

J. Rulifson, et. al. "QA4-A language for writing problem-solving programs", SRI Technical Note 48, November, (1970).

J. Thorpe, P. Bratley, H. Dewar. "The syntactic analysis of English by machine". In Michie (ed.), Machine Intelligence 3, New York, American Elsevier, (1968).

T. Winograd. "Procedures as a representation for data in a computer program for understanding natural language", MIT Thesis, (1971).

W. Woods. "Augmented transition network grammars for natural language analysis", CACM, Vol. 13, pp. 591-602, (1970).