PLATON—a New Programming Language for Natural Language Analysis

Makoto Nagao[*] and Jun-ichi Tsujii[*]

Abstract

A new programming language named PLATON (Programming LAnguage for Tree OperatioN), which has the facilities of pattern matching and flexible backtracking, is described. The language is developed in order to make it easy to write an analysis program of natural language. The pattern matching process not only checks whether a rewriting rule is applicable or not, but also extracts sub-strings from the input sentence and invokes appropriate semantic and contextual checking functions. We can set up arbitrary numbers of decision points in a program. If a failure occurs, the control will be changed appropriately according to the cause of the failure. By means of using this mechanism, we can write fairly complicated non-deterministic programs in a simple manner.

1. Introduction

In this paper, we describe a new programming language which is designed in order to make it easy to write natural language grammars. The augmented transition network (ATN) proposed by W. Woods gives a good framework for natural language analysis systems. One of the most attractive features is the clear discrimination between grammar rules and the control mechanism. So we developed PLATON by adding various facilities to this model.

2. Pattern-matching

During the analysis, 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, should be able to represent a coexisting structure of trees and lists. Now we show the formal definition of such a structure (Tab. 1). The structure is the

This paper first appeared in Japanese in Joho—Shori (Journal of the Information Processing Society of Japan), Vol. 15, No. 9 (1974), pp. 654～661.

[*] Department of Electrical Engineering, Kyoto University

fundamental data-structure, into which all of the objects processed by PLATON must be transformed. Two lists which have the same elements but different ordering of them should be regarded as different structures. On the contrary, two tree structures such as (A (R1 B)(R2 C)) and (A (R2 C)(R1 B)) are regarded as an identical one. Moreover, structural patterns are permitted in PLATON, which contain variable expressions. PLATON-interpreter matches such structural patterns against the structure under processing, 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 pattern are indicated as :X. Some examples are shown in fig. 1.

```
<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
```

Tab. 1. Formal Definition of structure

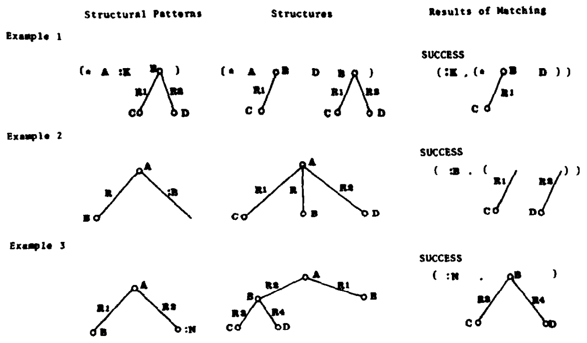

fig. 1. Examples of Pattern-Matching

3. Basic Operations of PLATON

A grammar is represented as a directed graph. Each branch represents a rewriting rule and each state has several numbers of branches ordered according to the preference of the rules (see fig. 2). When the control jumps to a state, it checks the rules associated with the state 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. By these operations, we can leave the analysis of substructures to different states. (see section-4)

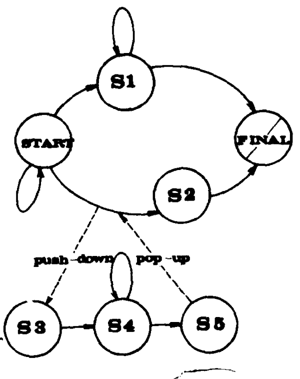

fig. 2. State Diagram

Each rule in PLATON is represented by a sixlet shown in fig. 3. The strx corresponds to the left side of a rule and indicates a structural pattern on which the

A Rule of PLATON

```
(<pcon> <strx> <con> <trans> <act> <end>)
                                     |
         ┌───────────────────────────┘
         │  ((NEXT  <state-name> <stry>)
         └─>{(NEXTB  <state-name> <stry>)
            {(POP  <stry>)
            ((FM  <failure-message>)
```

fig. 3. A Rule of PLATON

rule is applicable. The stry in end -part represents the right side of a rule. The variables used in strx are bound to corresponding substructures when matching succeeds. The scope of this binding is limited to the inside of a rule. On the other hand, we can store certain kinds of results of a rule in registers and refer them in different rules. This kind of variables, which we call registers, are represented by symbol /X.

pcon and con also check the applicability of a rule. At pcon -part, we can check the context of registers set by previous processing. In con -part we can check semantic and contextual co-ordination between substructures by pre-defined functions. For example, suppose

```
strx = ( ( ADJ ( TOK  :N ))(N (TOK  :N1 ))  :I)
con  = ( SEM  :N  :N1 )
```

Here SEM is a function defined by the user which checks the semantic co-ordination between the adjective :N and the noun :N1.

Arbitrary LISP-forms can be also used in act -part, and, if necessary, we can set intermediate results into registers. end comprises four varieties.

(1) NEXT-type; The stry represents the transformed structure. A rule of this type causes state-transition to the state-name.

(2) NEXTB-type; Rule which also causes state-transition. But unlike 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.

(3) POP-type; When the rule of this type 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 side-effects of the processing at this level, that is, register settings and so on, are cancelled (see section-4).

4. Push-down and Pop-up Operations

A rule in PLATON will find out a certain syntactic clue by its structural pattern strx , and at the same time,extracts substructures from the input string. We can predict that those substructures may have certain constructions, and transfer them to the appropriate states for predicted constructions. The result of analysis given back from these states can be inserted into the appropriate places. For example, suppose trans -part of a rule is

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

When the control interpretes this statement, the substructures corresponding to the variable :K and (:I :J) are transferred to the states S1 and S2 respectively. If the processing from these states is normally ended (by POP-type rule), then the results are stored into the variable :K and the register /REG. In this manner, by means of push-down and pop-up mechanism substructures can be analyzed from appropriate states. Processing from these states, however, may sometimes result in failure. In other words, predictions such that certain relationships must be found among the elements of substructures may not be fulfiled. In this case, the pushed down state will send up an error-message according to the cause of the failure by FM-type rule. If the NEXTB-type rules are used in the previous processing at this level, the control will go back to the most recently used NEXTB-type rule. If the NEXTB-type rules are not used at this processing level, the error-message specified by the FM-type rule will be sent up to the trans -part of the rule which directed this push-down operation. (see fig-4)

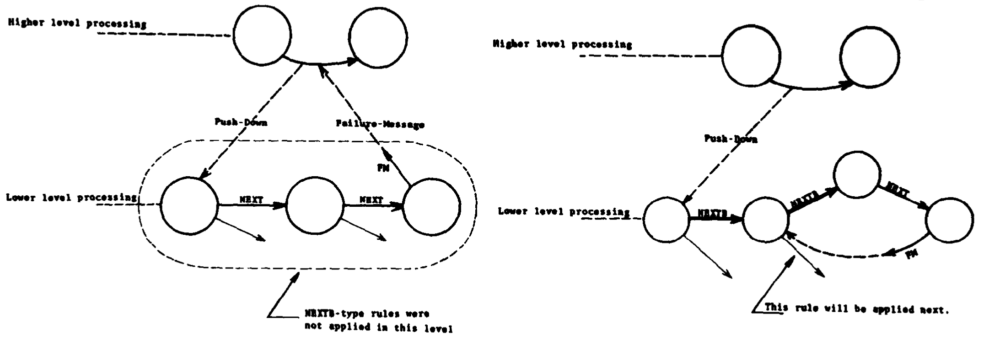

fig. 4.   Illustration of Backtracking

According to these error-messages, the control-flow can be changed appropriately. For example, we can direct such 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 result in one of the following three kinds.

(1) Normal return; the processing of ;K is ended by a POP-type rule. The result is

stored into the variable :K

(2) Return with an error-message; the processing of :K results in some failure and a FM-type rule sends up an error-message. If it is ERR1, then :K and (:I :J) will be analyzed from the state S5 and S6 respectively (EXEC-type). If it is ERR2, the interpreter will 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 there are no applicable rules. Then the interpreter will give up the application of the present rule and proceed to the next rule.

Mechanisms such that the 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.

6. Conclusion

Grammars written by PLATON not only maintain the clarity of representation but also provide adequately the natural interface between syntax and other components. By means of the pattern-matching facility, we can write grammars in a quite natural manner. And by its variable binding mechanism, semantic and contextual LISP functions are easily incorporated in syntactic patterns. Backtracking mechanisms and push-down operations make the complicated non-deterministic processing possible in a very simple way.

PLATON is written in LISP 1.5 and implemented on FACOM 230-60 at Kyoto University computing center and a mini-computer TOSBAC-40 of our laboratory. The interpreter of PLATON itself requirs only 4.5 kcells.

References

1. W. Wood. : 'Augmentex transition network grammars for natural language analysis', CACM, Vol. 13, pp. 591-602, (1970)

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

3. M. Nagao, J. Tsujii : 'PLATON-A NEW PROGRAMMING LANGUAGE FOR NATURAL LANGUAGE ANALYSIS' Proc. 2nd UJCC, pp. 208-214 (1975)

4. M. Nagao, J. Tsujii : 'Analysis of Japanese Sentences by Using Semantic and Contextual Information', Symposium at IIASA, Austria, (1975)