Syntactic Identification of Partial Information in Pattern Databases

Thomas W. Shepard
Larry S. Davis

Computer Sciences Department
University of Texas at Austin
Austin, Texas 78712

December 1980 TR-80-8

This research was supported in part by funds derived from the National Science Foundation under Grant ENG-7904037.

Abstract

This paper considers the organization of pattern databases when a formal grammar is available whose language contains the items in the database. The organization is based on the derivations of database elements, and allows access to the database given only partial pattern queries. Algorithms for computing all partial parses of pattern queries and for database access based on those partial queries are presented. The database access algorithm is based on fast string matching algorithms applied to string representations of the derivations of database items and pattern queries. The results of some computational experiments on computer generated pattern databases are presented.

1. Introduction

It has become apparent during the past several years that one of the major impediments towards constructing large pattern analysis systems is the lack of any practical theory and implementation of pattern databases. The pattern database problem constitutes an aggravated instance of the general database problem because not only must access to data be provided based upon symbolic data queries (e.g., retrieve, from a cartographic database, all counties in California with population greater than 50,000), but access must also be provided based on non-symbolic and more pattern-like data queries (e.g., are there any characters in a Chinese character database which look like this one).

This paper will be specifically concerned with pattern access to pattern databases. We define the pattern database problem to be the design of

1) logical storage organization for a class of patterns, and

2) access algorithms for retrieving that subset of the database which matches a pattern query.

There are a variety of factors which complicate the pattern database problem. First, if the patterns in the database contain geometric information (such as positions,

lengths, orientations, etc.) then the access algorithms may have to be invariant to certain geometric pattern transformations - e.g., we might be required to match a pattern independent of its orientation of size in the query.

Second, the pattern query may not be an exact match to any pattern in the database. The access methods must then be capable of computing nearest, or adequately near, matches. This "inexactness" often results from noise and systematic errors in the perceptual system which constructed the pattern query. For example, a computer vision system may imperfectly segment a character from the printed page; or, the pattern query may have been recieved over a noisy channel. In the latter case, one may be able to construct a precise model of pattern query errors and formally integrate it into the solution of the pattern database problem. In the former case, where the errors are probably due to current technological limitations in computer vision, a more informal, ad hoc, situation would have to be advised.

Finally, the pattern query may correspond to only a subpattern of any pattern in the database. In such situations, the database access algorithms must be capable of retrieving a pattern which only partially matches the pattern query. Partial pattern queries can be due to either systematic errors in the pattern percieving system or inherent properties of the entire pattern perception environment. For example, recognition of objects in the

three-dimensional world based on the information in a single, two-dimensional image will necessarily involve access to the database by partial pattern queries since not all of the points on the surface of a solid object can be seen in a single view.

In this paper, we will present a solution to the pattern database problem based on a syntactic model of the patterns to be stored in the database. Syntactic, or structural, pattern representations have had widespread application in pattern and image recognition. Fu [1] discusses some of these applications which include fingerprint recognition, character recognition, and industrial inspection tasks. Many of these applications utilize only minor modifications of ordinary string grammars; however, others require generalizations to grammars whose languages may be trees or even graphs.

Unidentified patterns are often analyzed using existing parsing algorithms from formal language theory. However, extensions are often required either to parse the higher-dimensional structures, or to allow for inexact matching. In pattern database applications, it would also be necessary to recognize a pattern based on only partial information (as discussed above). To our knowledge, no work has been reported on parsing algorithms which can be applied when only a partial pattern is available for analysis.

2.0 Identification of partial patterns

In this section, we consider the design of a syntactic database. The design will allow pattern queries which correspond to only partial patterns of any pattern in the database, but will not support inexact matching.

In what follows, we will assume that we are given:

1) a set, $D = \{d_1,\ldots,d_n\}$ of strings, d_i, and

2) a string grammar $G = \langle P, V_T, V_N, S\rangle$, where $D \subseteq L(G)$

P is the set of productions of G, V_T is the set of terminal symbols, V_N is the set of non-terminal symbols, S is the start symbol and L(G) is the language of G.

We will restrict G to be a stratified context-free grammar (SCFG). In an SCFG, every symbol, v, in $V = V_T \cup V_N$ has a level number (ℓn) from 0 to m associated with it. If $v \in V_T$, then $\ell n(v) = 0$. In addition, $n(S) = m$, and in any production $v := a \in V^*$, if $\ell n(v) = k$, then every symbol, v', in a has $\ell n(v') = k-1$. SCFG's were developed in [2] as the basis for a pattern analysis system. (Actually, the grammars used in [2] were much more elaborate than the SCFG defined above.)

In section 2.1 we will develop an algorithm for syntactic analysis of partial patterns. Section 2.2 will present a database organization where patterns are retrieved

from the database using a modification of a fast string matching algorithm (Knuth, Morris, Pratt [3]) applied not to the data strings themselves, but to the strings representing their derivations. Section 2.3 contains the results of timing tests of these algorithms as well as baseline timing data.

2.1 Partial parsing

In this section we define the problem of the analysis of partial information and present a solution. Section 2.1.1 presents a formal definition of a partial parse. A practical method of finding partial parses of a string is developed in section 2.1.2.

2.1.1 Definition of a partial parse

Pattern queries to the syntactic database system consist of elements of V^*. If $d = \delta_1,\ldots,\delta_r \in D$, and if $q = q_1,\ldots q_s$ is a pattern query, then we say that q <u>matches</u> d if, for some j, $q_i = \delta_{i+j}$, $i = 1,\ldots,s$. The task of the pattern database system is to find all $d \in D$ such that q matches d.

In order to perform this task, a "parse" (or "parses") of q must be computed, since the database is organized around the derivations of strings rather than the strings themselves. More precisely, we must compute all <u>partial parses</u> of q. Partial parses are defined below using syntax tree representation of a derivation.

Let $A = \{a_i\}_{i=1}^{|V|}$ be any symbols such that $a_i \notin V$. Then the string q' is an <u>extension</u> of q if $q' = \alpha\ q\ \beta$, where $\alpha \in A^*$ and $\beta \in A^*$.

Let $= \alpha_1\alpha_2\ldots\alpha_r$, $\alpha_i \varepsilon A$, $1 \leq i \leq r$. Then the function m: $A \rightarrow V_T$ is called a symbol map, and we let $m(\alpha)$ denote the string $m(\alpha_1)m(\alpha_2)\ldots m(\alpha_r)$.

Similarly, if $R \varepsilon (V \cup A)^*$, then $m(R)$ is obtained by applying m to the a_i which appear in R.

Definition 1: Let $q' = \alpha\ q\ \beta$ be an extension of q. Then we say that the string

$$S \Rightarrow R_1 \Rightarrow R_2 \Rightarrow \ldots \Rightarrow R_n = q'$$

is an e-derivation of q' if there exists a symbol map, m, such that

$$S \Rightarrow m(R_1) \Rightarrow m(R_2) \Rightarrow \ldots \Rightarrow m(R_n) = m(\alpha)qm(\beta)$$

is a derivation of $m(\alpha)qm(\beta)$.

Definition 2: Let T be the syntax tree for an e-derivation of some extension, q', of $q \varepsilon V_T^*$. Then the q-reduced syntax tree of T, T(q), for that e-derivation is obtained by 1) deleting from T all subtrees whose tip nodes consist entirely of symbols from A, then 2) recursively deleting the root node if it has only one son.

Finally, we arrive at

Definition 3: The left-to right, bottom-up traversal of a q-reduced syntax tree for an e-derivation of an extension of q is a partial parse of q.

To illustrate these definitions, we use the grammar G1 $= \langle P, V_T, V_N, S\rangle$ where

$$V_T = \{u, d\}$$

$V_N = \{S, V, P\}$

$P = \{(1)\ S \rightarrow PV$

$(2)\ S \rightarrow VP$

$(3)\ P \rightarrow ud$

$(4)\ P \rightarrow udu$

$(5)\ V \rightarrow du\}$

Consider the strings $q=uud$, $\alpha=d'$, and $\beta=u'$. Then $q'=\alpha q \beta = d'uudu'$ is an extension of q. Let m be a symbol map such that $m(\alpha)=m(d')=d$ and $m(\beta)=m(u')=u$. Then

$$S \rightarrow VP \rightarrow Vudu' \rightarrow d'uudu' = q'$$

is an e-derivation of q' since

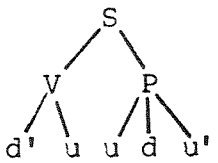
$$m(S) \rightarrow m(VP) \rightarrow m(Vudu') \rightarrow m(d'uudu') = m(\alpha)qm(\beta)$$

which can be written as

$$S \rightarrow VP \rightarrow Vudu \rightarrow duudu$$

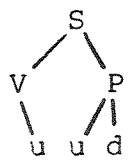
which is a derivation of $m(\alpha)qm(\beta)=duudu$ in G1. The syntax tree for the e-derivation is

T:



so that

T(q):



is the q-reduced syntax tree of T.

In this case, a traversal of either tree yields the same result since no part of the tree was pruned by the q-reduction.

A second example uses the grammar G2 = $\langle P, V_T, V_N, S \rangle$ where

V_T = {a, b, f, g, x, y}

V_N = {S, P, V, U, D}

P = {(1) S → PP

(2) S → VVP

(3) P → UD

(4) P → UUD

(5) V → DU

(6) U → af

(7) U → bg

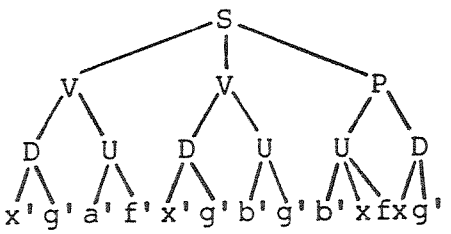
(8) U → bxf

(9) D → xg

(10) D → yaf }

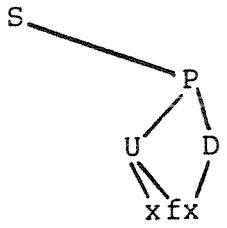
Consider the strings q=xfx and q'=x'g'a'f'x'g'b'g'b'x f x g'. Then the syntax tree for the e-derivation of q' is

T:



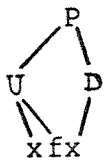
Deleting the subtrees whose tip nodes consist entirely of

primed terminals, we get



Since the root node, S, has only one son, S is deleted, leaving

T(q):



which is a q-reduced syntax tree of T. A left-to-right, bottom-up traversal of T(q) yields the productions 8 9 3, since these productions are applied at nodes U, D, and P respectively.

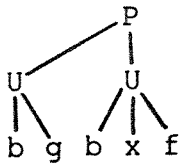
2.1.2 Generating partial parses

In order to generate all partial parses of a string q, it is not actually necessary to find all the q-extensions of q and prune the parse trees of these q-extensions. In this section we describe a parser which can efficiently find all partial parses of a given string with respect to an SCFG.

Given a string $q \in V_T^*$, the parser lists the production (by number) in a left-to-right bottom-up traversal of all

q-reduced syntax trees for q. The parser can insert zeroes in this list at all positions where it is known that one or more productions would have appeared in a traversal of the non-reduced syntax tree.

For example, given the grammar G2, one partial parse of the string bgbxf is 7 8 0 4, corresponding to the q-reduced tree



In this case the zero corresponds to the D generated by (4) P → UUD, because the D is a non-terminal which in this case produces nothing.

The parser uses a structure called a hypograph which corresponds to the grammar. It is similar to an and-or tree and contains exactly one node for each symbol (called symbol nodes) and one node for each production (called production nodes). At the bottom of each symbol node are arcs to each one of the productions on whose left side that symbol appears. At the bottom of each production node is a set of ordered arcs each of which points to a node corresponding to a symbol appearing on the right side of the production; these arcs are regarded as an "anded" group, meaning that all the symbols pointed to by those arcs must be matched for this production to match.

In the following we will refer to "downarcs" and "uparcs" of a node. A downarc of a node, N, is an arc which connects N to a node below it (in the figures), while an uparc on N would connect N to node above it. The hypograph for grammar G1 is shown in figure 1. The following example develops the method of obtaining partial parses with this structure.

2.1.2.1 Example: using the hypograph

The parser utilizes two stacks. The first stack (which is displayed in the figures below the hypograph) accumulates the production numbers which will appear in the derivation. It is called the derivation stack. The second stack (to the right of the hypograph) holds backtracking information: each entry contains a node and (when necessary) a downarc from that node. This stack is called the backtrack stack.

In this example we will describe the computation of all partial parses of the string uud using the grammar G2. The following paragraphs are all numbered; paragraph 1.i is illustrated in Figure 1.i.

1.1. The first two uparcs from node u were searched, but failed because in the productions they point to (i.e., downarc 1 of production 3 and downarc 1 of production 4),

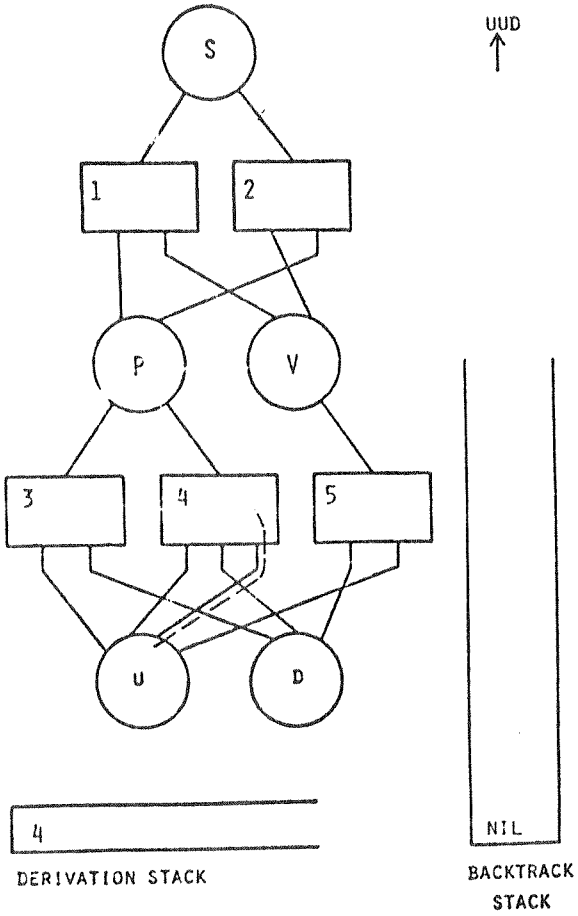


FIGURE 1.1

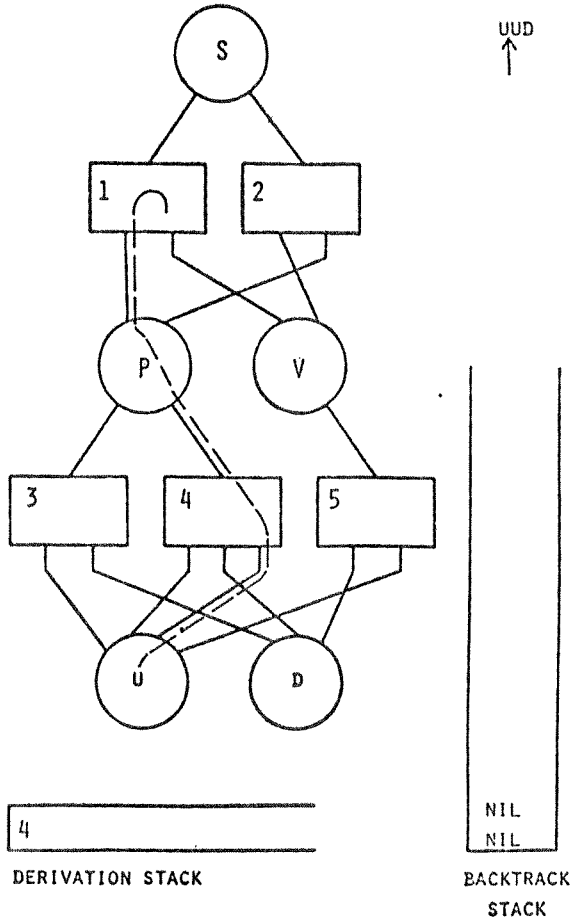


FIGURE 1.2

the next downarc from those productions (downarc 2 of productions 3 and 4) points to node d which doesn't match the input string of position 2. Uparc 3 from node u is searched next. Nil is pushed on the backtrack stack to mark entry into the bottom of a production node. Production 4 is pushed on the derivation stack.

1.2. The search continues up to node 1 where entry is again marked by pushing nil on the stack. The parser prepares to match the remaining downarc(s) of node 1.

1.3. The only symbol found down arc 2, node 1 was a d with no alternatives, causing backtracking. Returning to node 1, nil was popped, indicating that a failure return should be made rather than a search for alternatives down the return arc. The parser returned to uparc 1, node P, and followed uparc 2 to node 2. Nil was pushed on the backtrack stack and production 2 on the derivation stack, since P is the last symbol in production 2.

1.4. The parser moves up to node S, but there are no uparcs to investigate, and there are symbols left in the input string. The parser is searching for alternatives but has none. A failure return is made.

1.5. The production stack is popped, and now the parser attempts to back through production 2's downarcs looking for alternatives. But nil is popped from the backtrack stack, so the parser makes a failure return instead.

1.6. Returning to uparc 2, node P, the parser has run out of uparcs and must make a failure return from node P (this is the same situation as in 1.4 at node S). Backing into node 4, the derivation stack is popped. Nil is popped from the backtrack stack, so node 4 makes a failure return.

1.7. Uparc 4, node u is searched up to node S, which again fails. However, productions 5 and 1 are pushed onto the derivation stack.

1.8. Backing through node 1, the derivation stack is popped and nil is popped from the backtrack stack. Returning failure to node V, the parser investigates uparc 2, pushing nil as it enters node 2 at downarc 1.

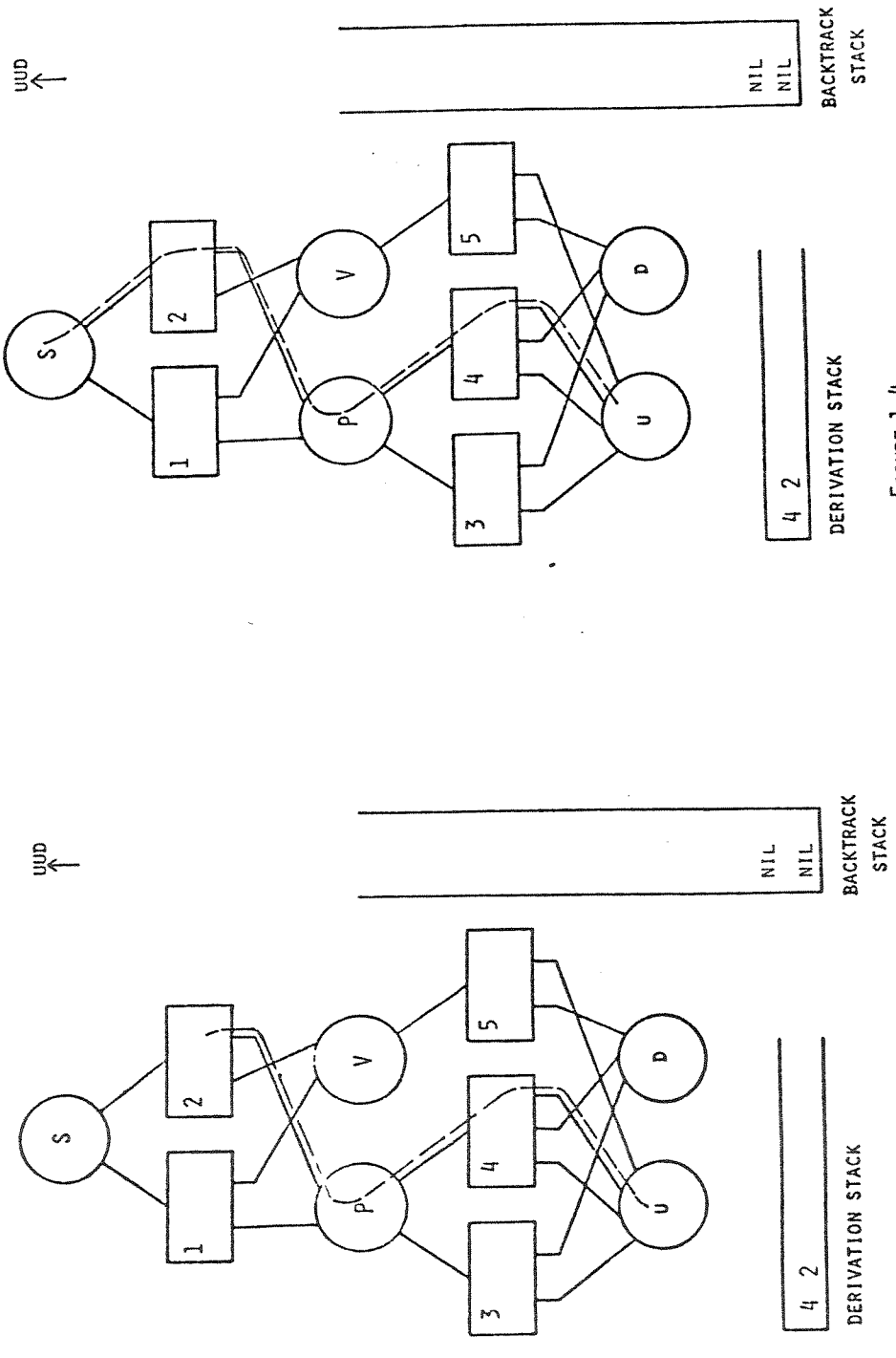


FIGURE 1.4

FIGURE 1.3

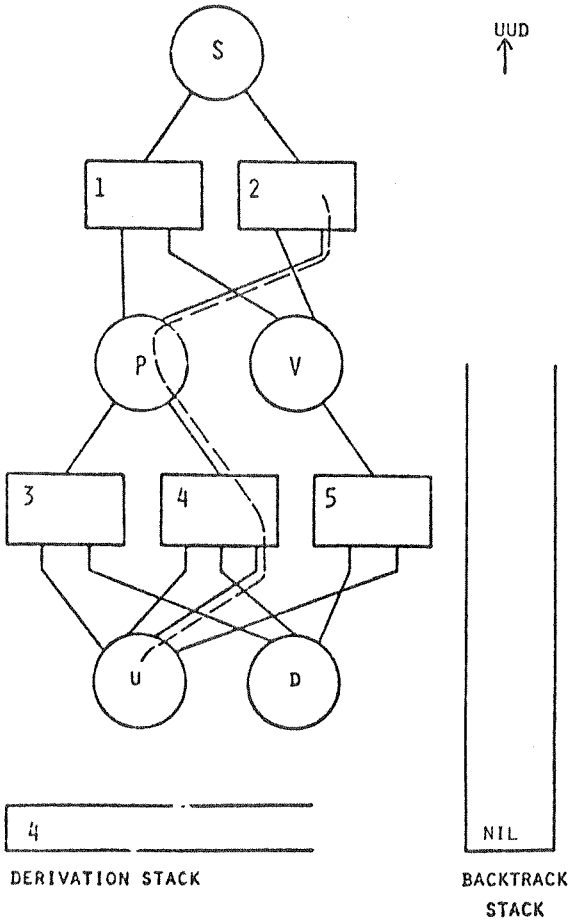


FIGURE 1.5

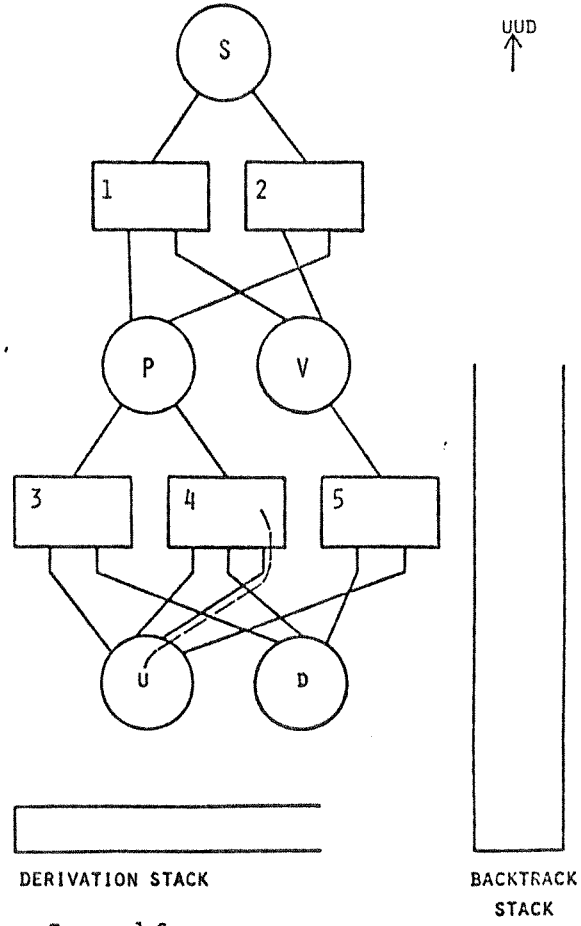


FIGURE 1.6

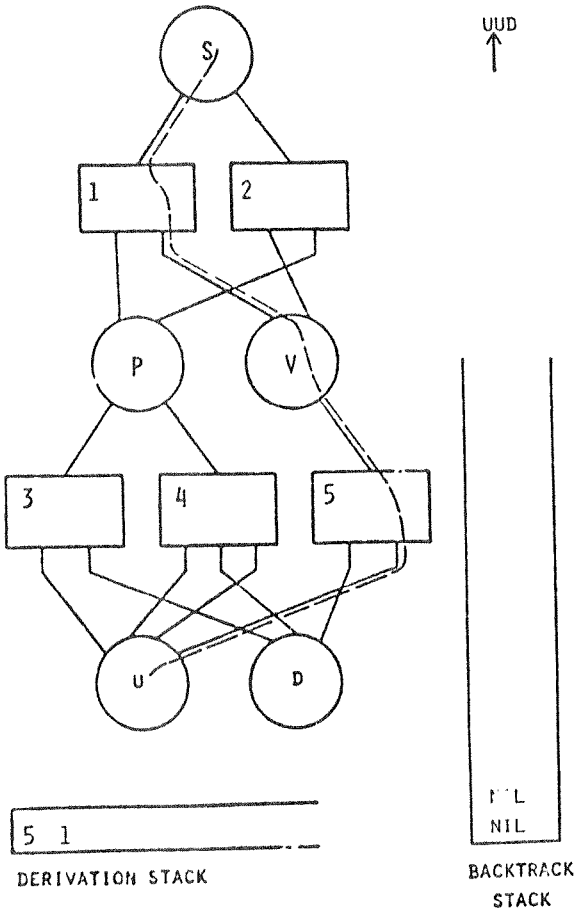


FIGURE 1.7

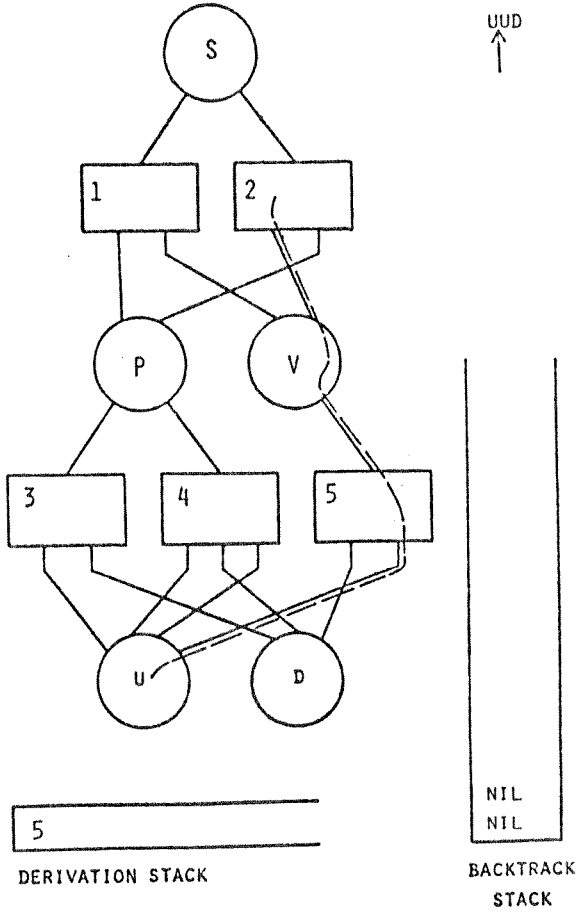


FIGURE 1.8

1.9. Searching downarc 2 from node 2, the parser pushes nil as it enters downarc 1, node 3. This will be used later to stop backtracking through the downarcs of node 3. Node u is entered and found to match the input string. (u,←) is pushed on the backtrack stack.

1.10. (3,1) and (d,←) are pushed on the backtrack stack.

1.11. (3,2) is pushed on the backtrack stack. Production 3 is put on the derivation stack since we have successfully matched production 3. (P,1) is pushed, indicating that downarc 1 of node P has matched.

1.12. Returning successfully to node 2, (2,2) is pushed and production 2 is put on the derivation stack. Since there is no input left, the parser does not need to move any higher up the tree, because the symbol above this production could be part of many productions which would all be found as alternatives. The derivation stack can be output since it just accounts for the input. However, there may still be alternatives to be found below this node: a substring may have more than one parse even though the grammar is non-ambiguous. After the derivation stack is printed, therefore, an artificial failure is forced so as to cause backtracking.

1.13. The derivation stack is popped and (2,2) and (P,1) are popped from the backtrack stack.

1.14. Production 3 is popped from the derivation stack. (3,2), (d,←), (3,1), and (u,←) are popped, with no alternatives being found. The parser is preparing to return from node u.

1.15. Backing into node 3, the parser tries to backtrack through any previous downarcs. Popping nil from the stack, however, it sees that no alternatives remain. It executes a failure return.

1.16. Returning to downarc 1, node P, the parser tries to see if a P can be accounted for in any other ways, so it looks at downarc 2 to production 4.

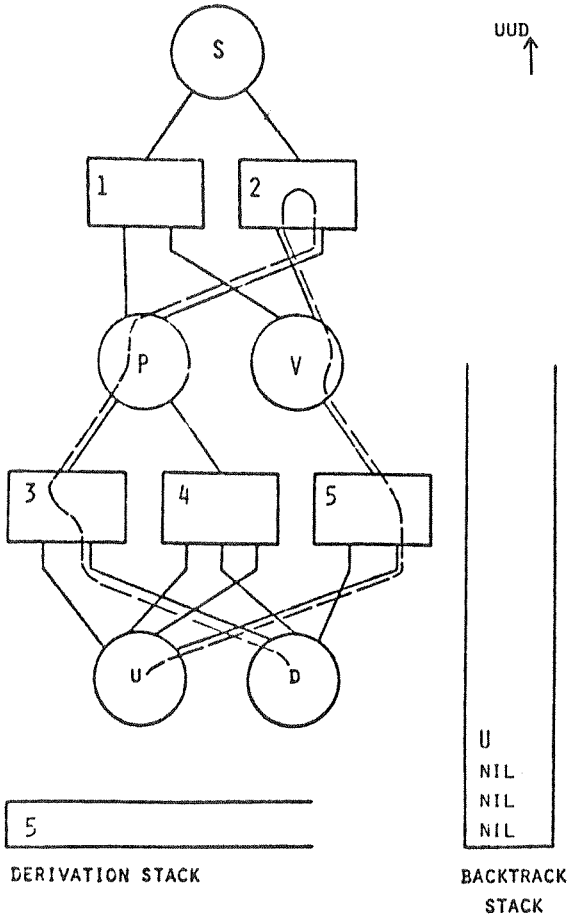


FIGURE 1.9

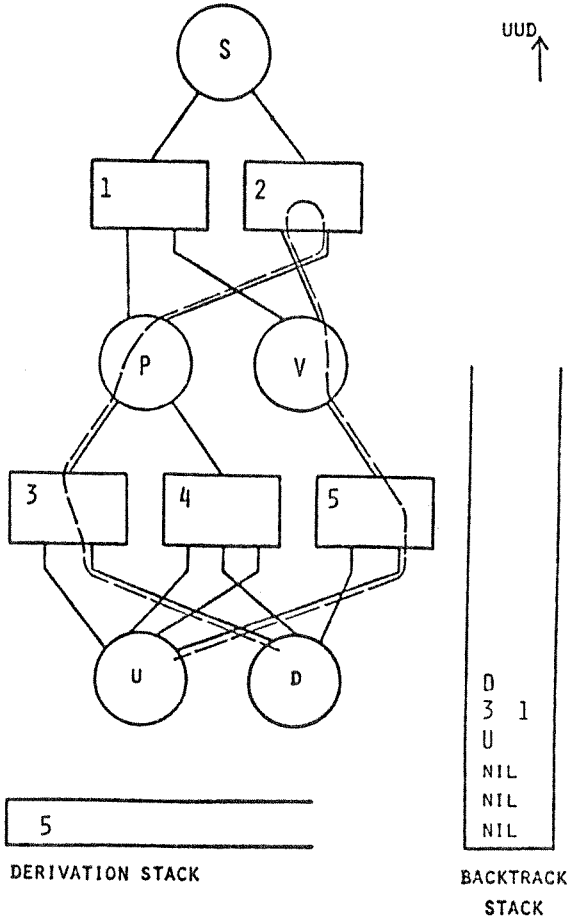


FIGURE 1.10

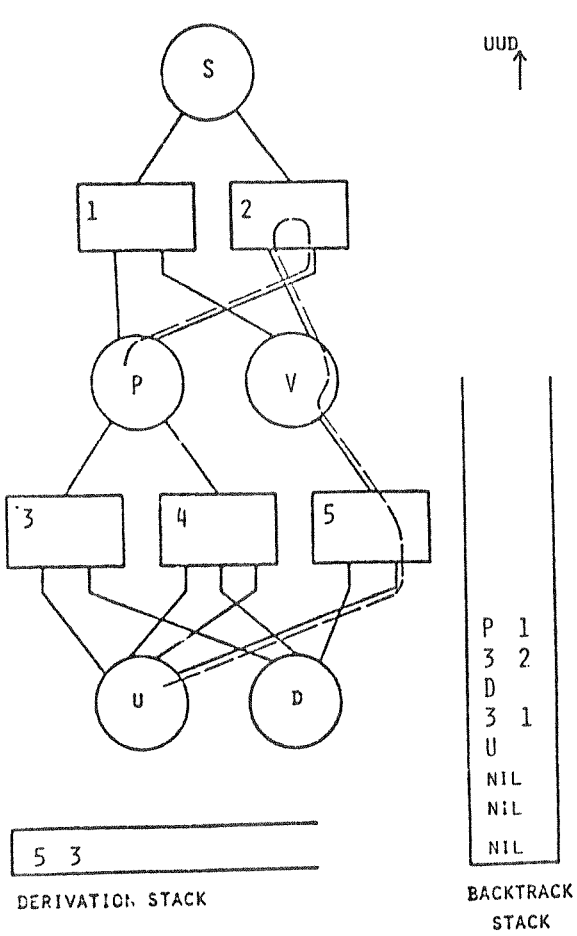


FIGURE 1.11

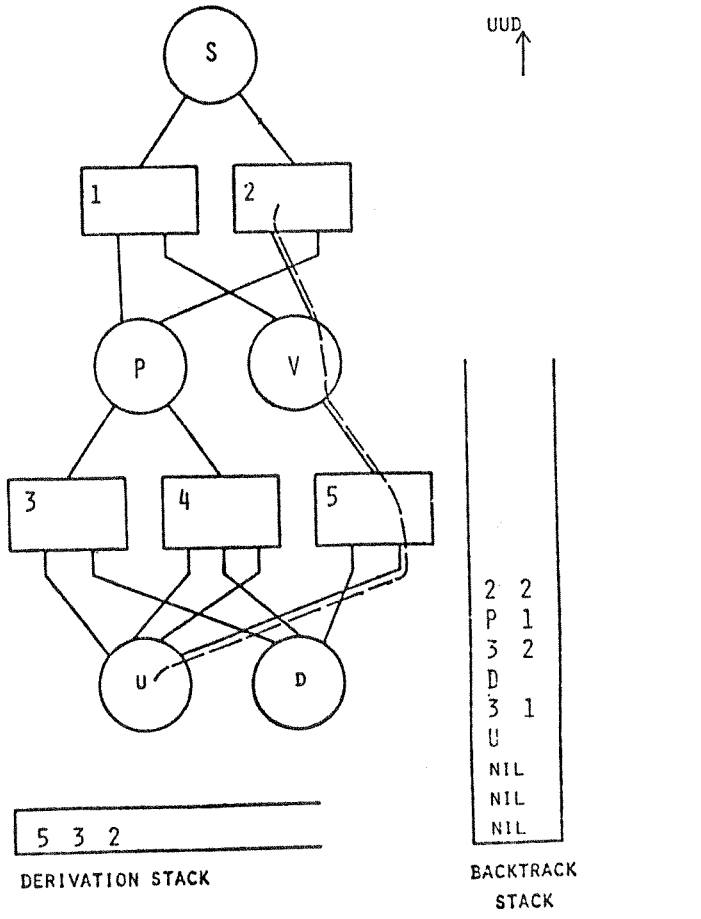


FIGURE 1.12

(OUTPUT DERIVATION STACK)

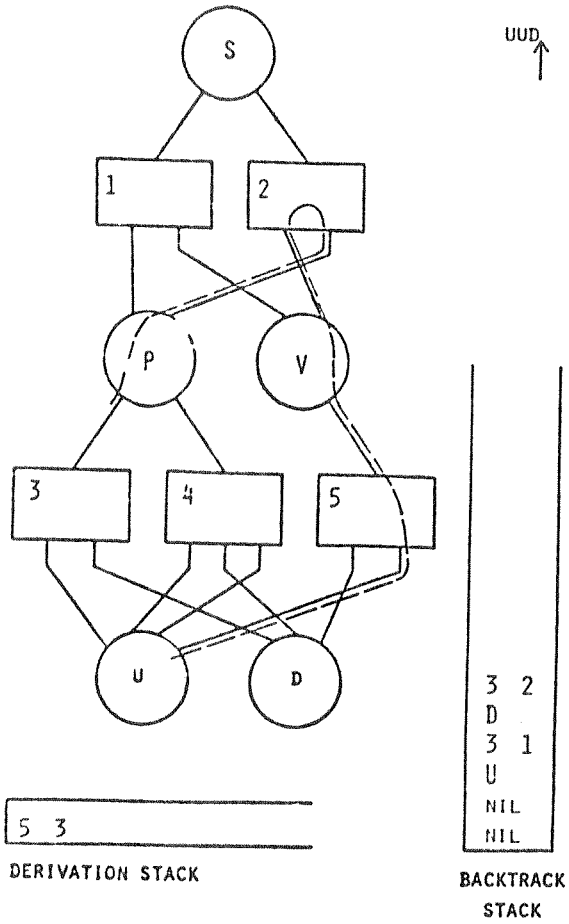


FIGURE 1.13

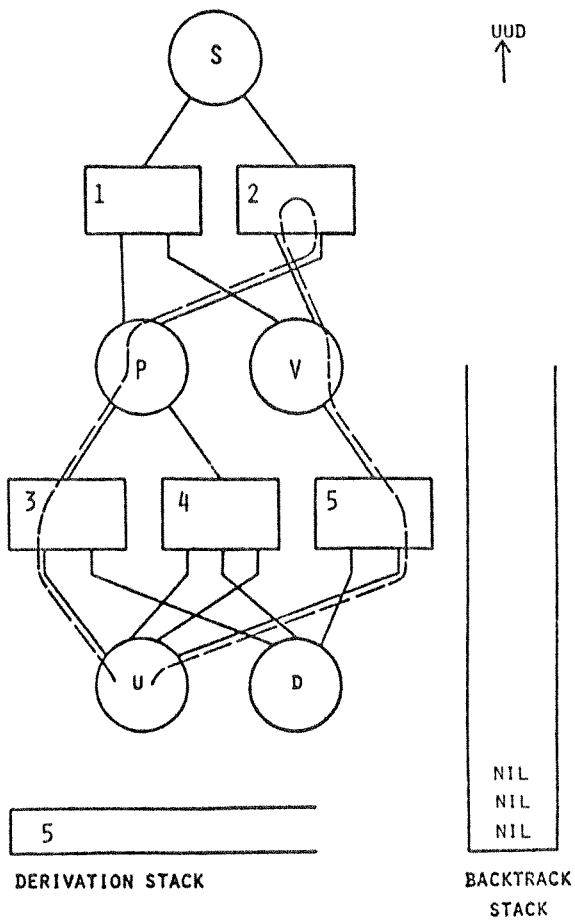


FIGURE 1.14

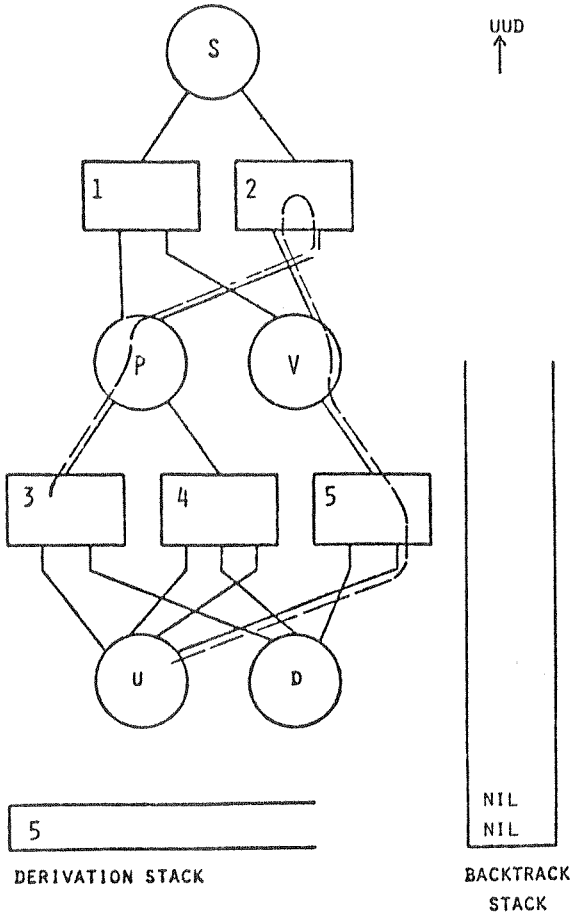


FIGURE 1.15

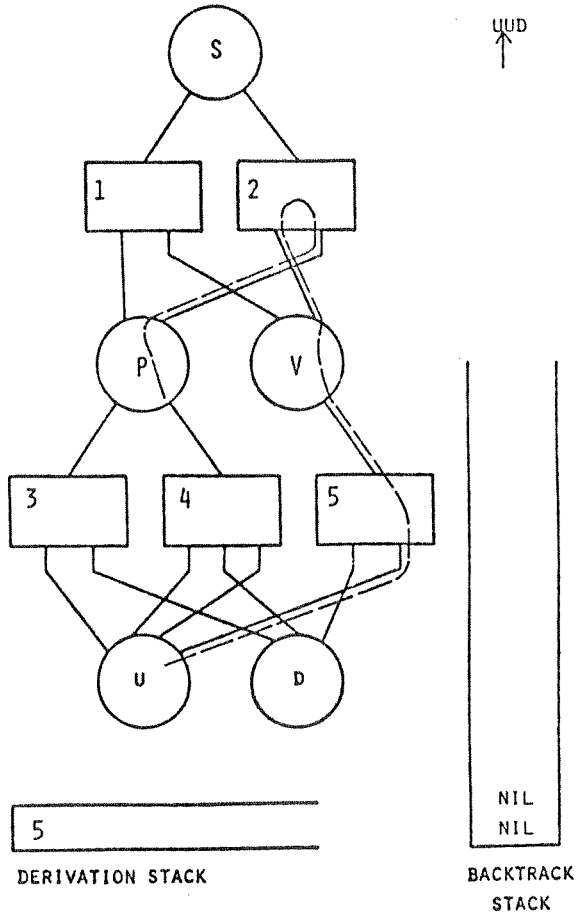


FIGURE 1.16

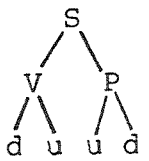
1.17. The first two downarcs of node 4 match, and there is now no more input. Thus, the parser has not seen anything so far to cause 4 to fail, but is has partially succeeded. Since the input could be partial information, we cannot rule out the possibility that if more input had been available, production 4 would have matched. So production 4 is pushed on the derivation stack and a success return is made.

1.18. (P,2) and (2,2) are pushed on the backtrack stack. Production 2 is pushed on the derivation stack. Again, the derivation stack is printed, and a mock failure is forced.

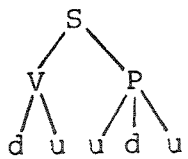
1.19. No alternatives were found at downarc 2, node 2, so backtracking occurs.

1.20. Nil is popped, so a failure return is made to uparc 2, node V. There are no more uparcs, so a failure return is made to node 5. Nil is popped, causing a return to node u. There are no more uparcs here either, so the parser is finished.

The two derivations which the parser produced, 5 3 2 and 5 4 2, correspond to the syntax trees



and



Although for this simple grammar the partial parses actually charcterize the entire language strings of which the input string is a substring, for more complex grammars (such as G2 used in section 2.3) this will not ordinarily be the case.

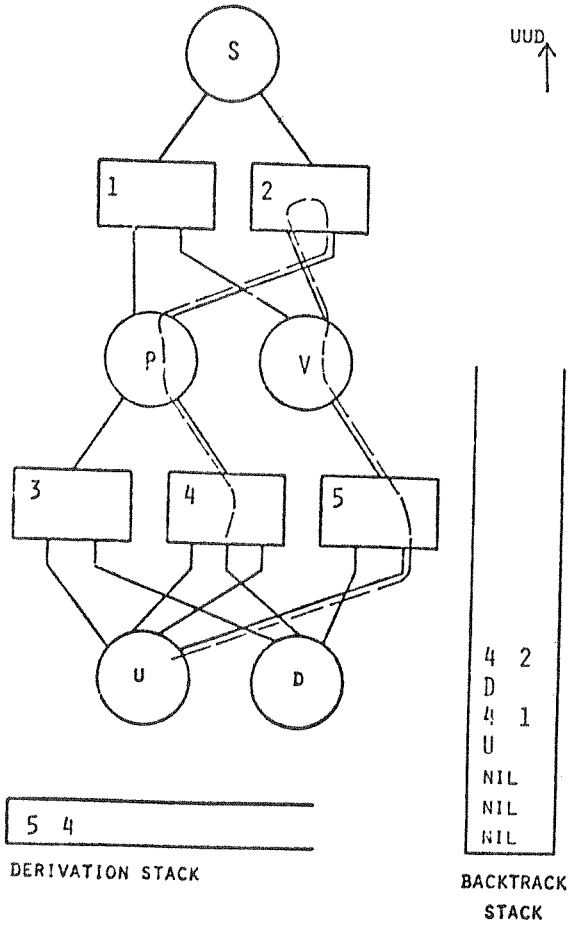


FIGURE 1.17

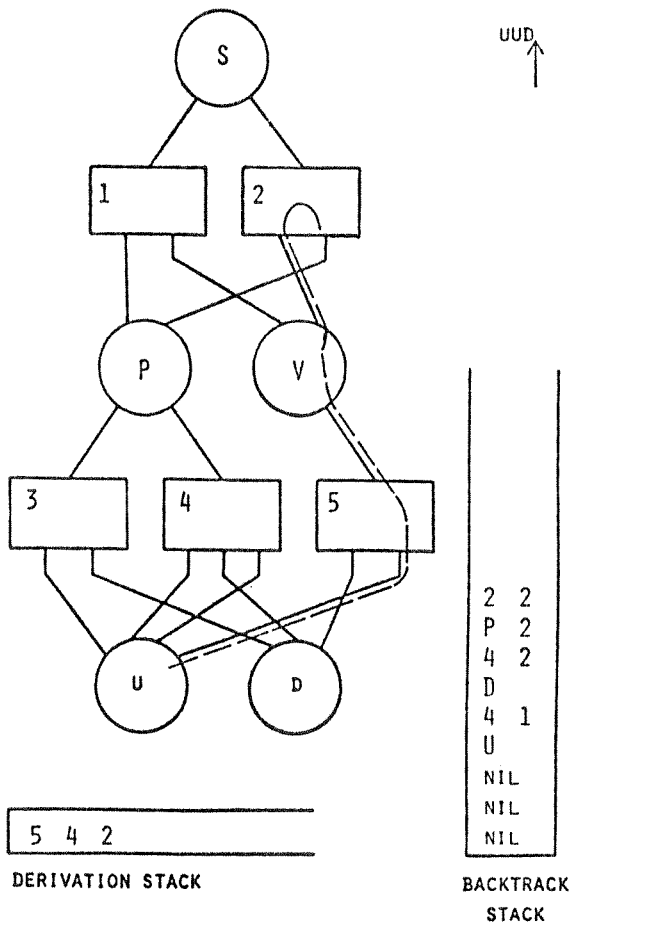


FIGURE 1.18

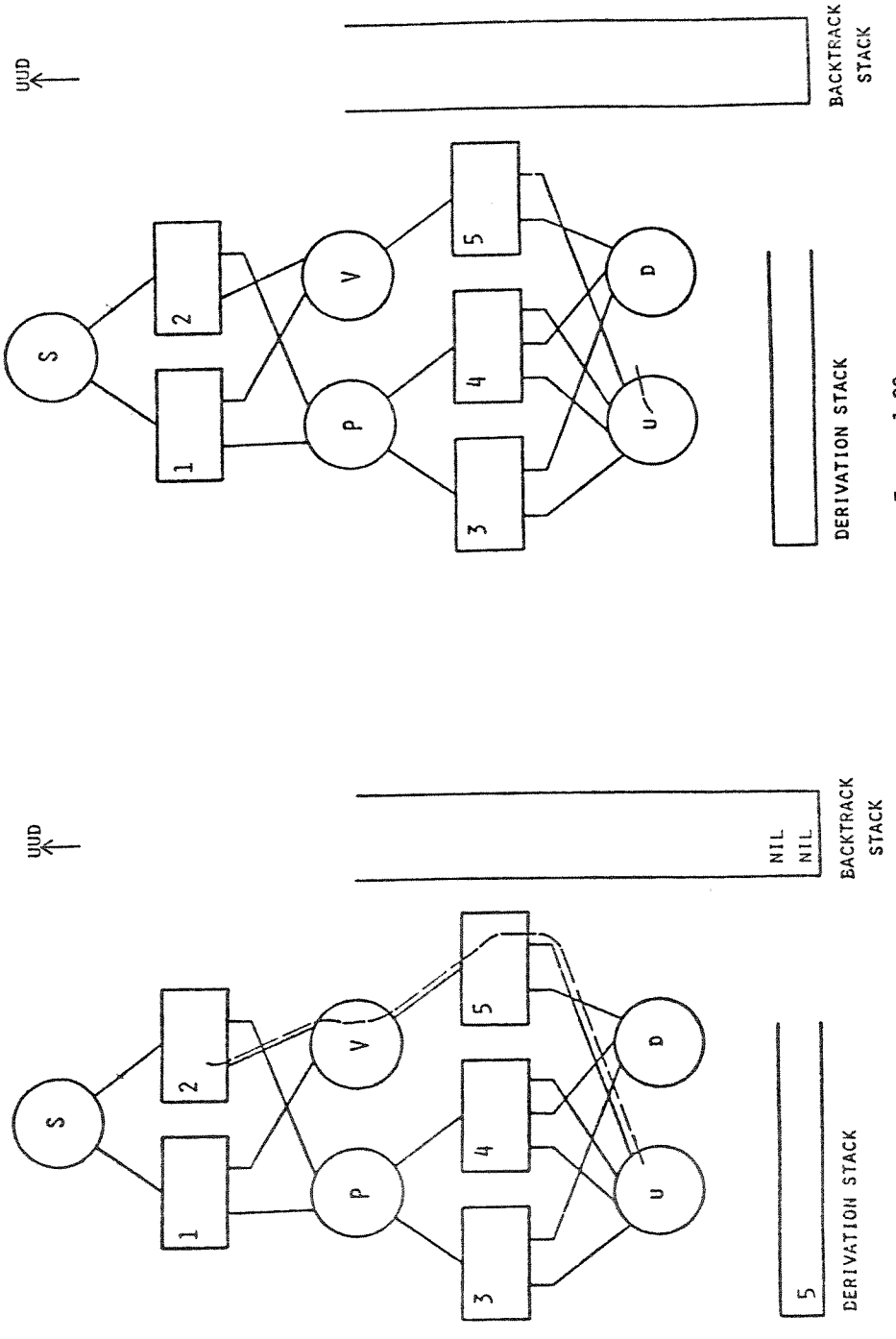


FIGURE 1.19

FIGURE 1.20

2.1.2.2 Development of the algorithms

We will now develop the algorithms for partial parsing by categorizing the actions described in example 3. Each of these actions will then be described in terms of the procedures which perform them.

There are two types of nodes, symbol nodes and production nodes. For both types there are three ways in which the path can enter a node, each with its associated actions:

1) entering from above (not backtracking)

a) symbol nodes

* In example 1.1, node d is entered from above twice: once from production node 3, and once from production node 4.
* 1.3: the path enters node V, then node d.
* 1.9: node P is entered as the parser searches node 2, downarc 2. The path then enters node u.
* 1.10: node d is entered.
* 1.17: the first two downarcs of node 4 point to nodes u and d, which are entered.

Upon entering a terminal symbol node from above, the parser must compare that node's symbol with the current input symbol. Upon

entering a non-terminal symbol node from above, the parser must scan the downarcs of that symbol node; we will refine the definition of this action later.

b) production nodes

* 1.3: the path enters production node 5 from above.

* 1.9: node 3 is entered from above

* 1.17: node 4 is entered

Upon entering a production node from above, the parser must scan the downarcs of that node, starting with the first downarc; we will refine the definition of this action later.

2) entering from below

a) symbol nodes

* 1.1: the parser starts by entering node u from below.

* 1.2: node P is entered from below on the way up to node 1.

* 1.4: node S is entered.

* 1.7: The path enters node V and node S.

When a symbol node is entered from below, the

parser must then search all the uparcs of that node. For example, in 1.1 node u is entered and its first two uparcs are searched, and the search of the third uparc is begun; in 1.7 the search of the third uparc ends and the search of the fourth uparc begins. All the uparcs are searched, even if a successful partial parse was found in the search of an earlier uparc; the search of a later uparc might reveal an alternative partial parse.

b) production nodes

* 1.1: Node 3 is entered through the first downarc. Node 4 is also entered through its first downarc. Then node 4 is entered through its third downarc.

* 1.2: Node 1 is entered through its first downarc.

* 1.7: Node 5 is entered through its second downarc. Node 1 is also entered through its second downarc.

* 1.8: Node 2 is entered through its first downarc.

Upon entering a production node from below, the parser must scan the downarcs following the one through which that node was entered. In 1.7, there are no downarcs following the one through which node 5 was entered, so

there is nothing to scan. In 1.8, however, there is a downarc following the one through which node 2 was entered, and this remaining arc must be "scanned". This scan is almost identical to the scan of production downarcs when the producion node is entered from above, the only difference being the downarc at which the scan is initiated. Again, we will defer the detailed description of this action until later.

3) entering from above (backtracking)

a) symbol nodes

* 1.13: The parser pops (P,1) from the backtrack stack and enters node P from above. More importantly, the downarc popped at the same time (downarc 1 of node P) tells the parser which downarc to backtrack through upon entering node P.

* 1.14: Node d is entered from above (backtracking); there are no alternatives to be taken at a terminal symbol node. Node u is also entered from above (backtracking).

* 1.19: Nodes d and u are again entered, this time from production 4.

When backtracking into a symbol node from above, the parser must search for alternatives through the downarc that is

popped with that node from the backtrack stack. If no alternatives are found, the remainder of the symbol downarcs must be scanned. This is almost the same as the scan of the symbol downarcs when the symbol is entered from above (not backtracking), differing only in the downarc with which the scan is started.

b) production nodes

* 1.14: Node 3 is entered from above (backtracking).
* 1.19: Node 4 is entered from above, while backtracking down from downarc 2, node 2.

Upon entering a production node from above (backtracking), the parser performs a backtrack scan of that production's downarcs; this happens to be a natural part of the production scan already mentioned, and a discussion of this will also be left until later.

Corresponding to these actions are six procedures:

SFA - enter a symbol from above

PFA - enter a production from above

SFB - enter a symbol from below

PFB - enter a production from below

SFAB - enter a symbol from above, backtracking

PFAB - enter a production from above, backtracking

There are also two auxiliary procedures:

SCANS - scan symbol downarcs (called by SFA and SFAB)

SCANP - scan production downarcs (called by PFA, PFB and PFAB)

The parser prepares for a partial parse by finding the first input symbol (and a pointer to the corresponding symbol node) in a list of terminal symbols of the grammar. The pointer to this terminal symbol node is passed to SFB to start the parse by entering this node from the bottom; a return from this call means that all possible partial parses have been output.

Before describing these procedures in detail, we will define some hypograph data structure terminology.

A hypograph is made up of two types of basic structures: nodes and half-arcs. Each node contains

1) a pointer to the first uparc in that node (call this pointer UPARC) and

2) a pointer to the first downarc in that node (call this pointer DOWNARC).

In figure 2, which represents part of the hypograph for grammar G1, UPARC for node P points to the structure corresponding to uparc 1, node P; DOWNARC points to the structure corresponding to downarc 1, node P. In addition to these two fields, each production node contains the number of this production in the grammar (this item is called PRODNUMBER), and each terminal symbol node contains the symbol it represents (this item is called SYMBOL).

There is one half-arc in the data strucure for each of the two ends of each arc in the hypograph. In figure 2 there is a half-arc for downarc 2, node P, and a half-arc for uparc 1, node 4; this pair of half-arcs represents the arc between node P and node 4. Each half-arc contains three pointers:

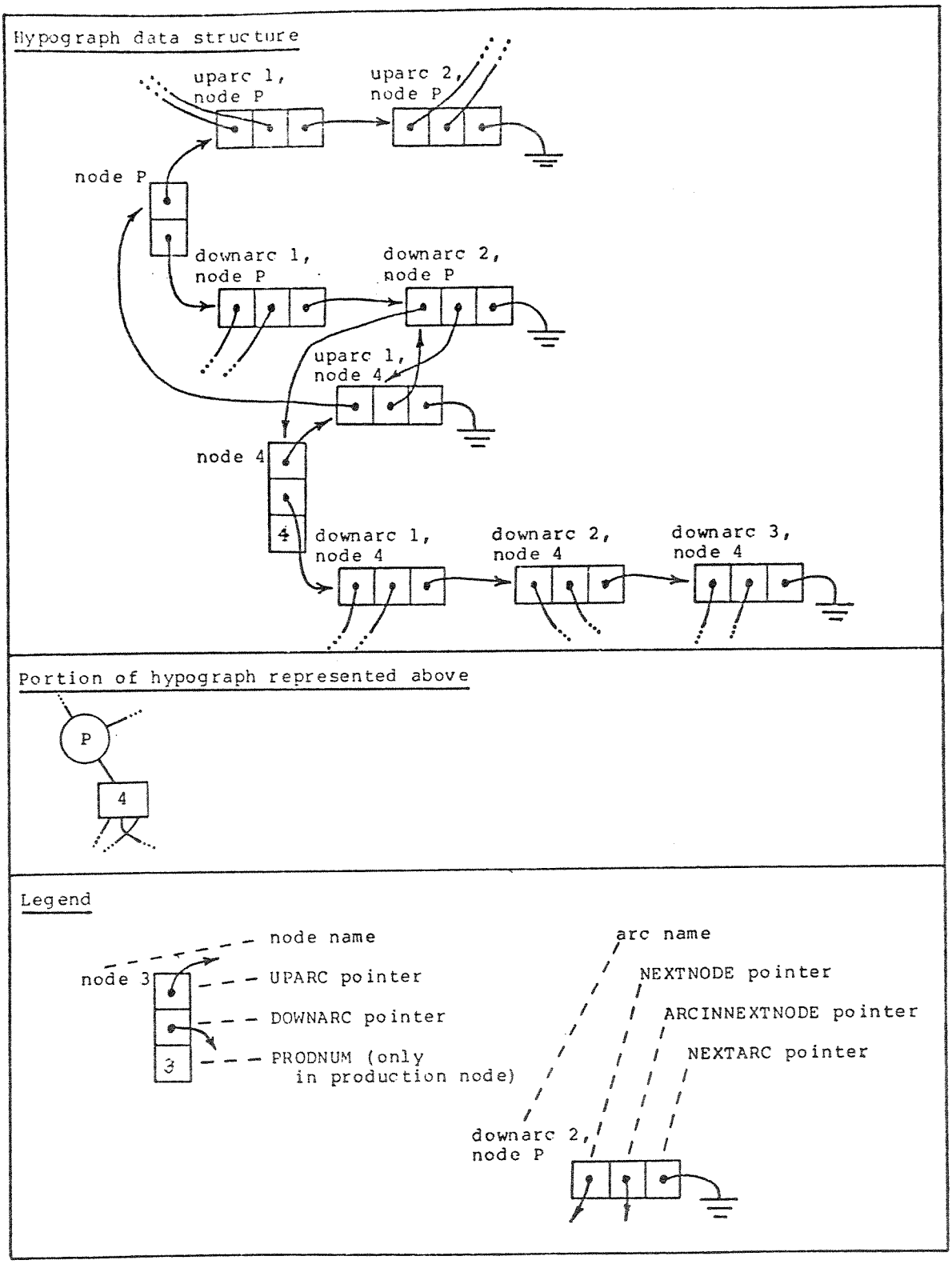

Hypograph data structure
uparc 1, node P
uparc 2, node P
node P
downarc 1, node P
downarc 2, node P
uparc 1, node 4
node 4
4
downarc 1, node 4
downarc 2, node 4
downarc 3, node 4
Portion of hypograph represented above
P
4
Legend
node name
node 3
UPARC pointer
DOWNARC pointer
3
PRODNUM (only in production node)
arc name
NEXTNODE pointer
ARCINNEXTNODE pointer
NEXTARC pointer
downarc 2, node P

Figure 2

1) A pointer to a node (this pointer is called NEXTNODE). Note that in figure 2, NEXTNODE of downarc 2, node P points to node 4.

2) A pointer to the other member of this half-arc pair (this pointer is called ARCINNEXTNODE). Thus, downarc 2, node P points to uparc 1, node 4, and vice versa.

3) A pointer to the next half-arc in the same node as this half-arc (this pointer is called NEXTARC). All the downarcs in a node are connected in a linked list; the DOWNARC pointer of this node points to the head of the list, and the NEXTARC of the last half-arc is nil. (Node 4 in figure 4 points to a list of three downarcs.) All the uparcs of a node are connected in a similar fashion, except that the UPARC pointer of the node points to the head of the list.

2.1.2.3 Procedure listings and descriptions

We now describe the eight procedures of the partial parser. When a reference is made to a specific line in a procedure, we will include, in parentheses, the procedure name and line number. A listing of each procedure, with

line numbers, can be found after the description of that procedure.

A description of the auxiliary scan routines, which has been thus far delayed, will now be presented.

SCANS is passed a pointer to a downarc in a symbol node (call this parameter ARC) (SCANS 1, called from SFA 14 and SFAB 15). It then calls PFA with the node pointed to by ARC (SCANS 5). If PFA fails, SCANS makes ARC the next downarc (i.e., NEXTARC) (SCANS 7). SCANS then repeats from the beginning if it is not yet at the end of the list of downarcs (ARC<>nil) (SCANS 4 to SCANS 8). If a call to PFA is successful, SCANS returns the ARC which was successful.

```
SCANS  1 procedure SCANS (var ARC, var SUCCESS);
SCANS  2       (* scan symbol downarcs *)
SCANS  3   begin
SCANS  4     repeat
SCANS  5       PFA (ARC^.NEXTNODE, SUCCESS);
SCANS  6       (* if the search down this ARC failed,
                    get the next downarc *)
SCANS  7         if not SUCCESS then ARC:=ARC^.NEXTARC;
SCANS  8       until SUCCESS or (ARC=NIL);
SCANS  9     end (* SCANS *);
```

SCANP has two modes, forward and backtracking. On entry SUCCESS=true indicates forward mode and SUCCESS=false indicates backtrack mode. At exit SUCCESS=true means the production downarcs successfully matched, and SUCCESS=false means the downarcs failed to match in order and there are no alternatives.

When called with forward mode, it is also passed ARC, a pointer to the first of a list of production downarcs to be searched (SCANP 1, called from PFB 5 and PFA 5). Nil is pushed on the backtrack stack to mark the beginning of the scan (SCANP 6). SFA is called with ARC.NEXTNODE (SCANP 20), and if successful, this ARC is pushed on the stack (SCANP 25), ARC is set equal to ARC.NEXTARC (SCANP 26), and the process is repeated (SCANP 13 to SCANP 30); if SFA fails at any point, SCANP enters backtrack mode (SCANP 28,29). While in forward mode, SCANP will exit if ARC=nil (all arcs have been scanned) or if there is no input left (SCANP 14 to SCANP 17). If SCANP finishes with ARC<>nil (this implies forward mode and no input left) then zero is pushed on the derivation stack if ARC^.NEXTNODE points to a non-terminal (SCANP 31 to SCANP 33); this means that if there had been more input, there would be one or more additional production numbers on the derivation stack upon exiting SCANP.

When called with backtrack mode (SCANP 1, called at PFB 13 and PFAB 7), SCANP pops the derivation stack if there is a zero on top (SCANP 10 to SCANP 12). When SCANP is in backtrack mode it pops ARC from the backtrack stack (SCANP 9 and SCANP 28,29) and calls SFAB (SCANP 21,22); this is repeated until SFAB succeeds or ARC=nil. In the first case SCANP enters forward mode (SCANP 24 to SCANP 27). In the second case SCANP exits, indicating that there are no more alternatives to be found.

```
SCANP  1 procedure SCANP (ARC, var SUCCESS);
SCANP  2       (* scan procedure downarcs *)
SCANP  3   begin
SCANP  4     if SUCCESS
SCANP  5       then begin (* forward mode *)
SCANP  6         PUSHBACKTRACK(NIL,NIL);
SCANP  7         end
SCANP  8       else begin (* backtrack mode *)
SCANP  9         POPBACKTRACK(DUMMYNODE,ARC);
SCANP 10         if ARC<>NIL
SCANP 11           (* if zero was left on the derivation
                      stack last time, pop it *)
SCANP 12             then if DERISTACK[DERISTACKLOC]=0
                       then POPDERI;
SCANP 13     while (ARC<>NIL)
                 and not(SUCCESS and NOINPUTLEFT) do begin
SCANP 14         (* i.e., quit looping only if
SCANP 15             ARC=NIL
SCANP 16             or
SCANP 17             we have SUCCESS and
                       there is no input left *)
SCANP 18       if SUCCESS
SCANP 19         then (* forward mode *)
SCANP 20           SFA(ARC^.NEXTNODE,SUCCESS)
SCANP 21         else (* backtracking mode *)
SCANP 22           SFAB(SUCCESS);
SCANP 23       if SUCCESS (* of SFA or SFAB *)
SCANP 24         then begin (*forward mode: get next ARC*)
SCANP 25           PUSHBACKTRACK(NIL,ARC);
SCANP 26           ARC:=ARC^.NEXTARC;
SCANP 27           end
SCANP 28         else (*backtrack mode: get previous ARC*)
SCANP 29           POPBACKTRACK(DUMMYNODE,ARC);
SCANP 30       end (* while etc. *);
SCANP 31     if (ARC<>NIL)
SCANP 32       then if (not TERMINAL(ARC^.NEXTNODE))
SCANP 33         then PUSHPROD(0);
SCANP 34     end (* SCANP *);
```

Each of the procedures SFA, PFA, SFB, and PFB is passed a pointer to the node to be entered (this parameter is called NODE).

SFA has a part for processing terminal symbol nodes and a part for processing non-terminal symbol nodes. It is called from SCANP 20.

If this is a terminal node, and NODE^.SYMBOL does not match the current input symbol, SFA returns SUCCESS=false (SFA 11). If it does match, this NODE is pushed on the backtrack stack (SFA 7), the cursor is advanced one character (SFA 8), and SFA returns SUCCESS=false (SFA 9).

If this is not a terminal node, then SCANS is called with a pointer to the first downarc of this node (SFA 13,14). If SCANS is successful, then the current ARC (i.e., the one SCANS was successful with) is pushed on the backtrack stack and SFA returns SUCCESS=true. If SCANS fails, SFA returns SUCCESS=false.

```
SFA  1 procedure SFA (NODE, var SUCCESS);
SFA  2     (* symbol from above *)
SFA  3   begin
SFA  4     if TERMINAL(NODE)
SFA  5       then if CURRENTCHAR=NODE^.SYMBOL
SFA  6         then begin
SFA  7           PUSHBACKTRACK(NODE,NIL);
SFA  8           MOVCURSORFORWARD;
SFA  9           SUCCESS:=TRUE;
SFA 10           end
SFA 11         else SUCCESS:=FALSE
SFA 12       else begin
SFA 13         ARC:=NODE^.DOWNARC;
SFA 14         SCANS (ARC, SUCCESS);
SFA 15         if SUCCESS then PUSHBACKTRACK(NODE,ARC);
SFA 16         end;
SFA 17     end (* SFA *);
```

PFA is called at SCANS 5 or SFAB 11. PFA first calls SCANP with the forward mode and a pointer to the first downarc of this node (PFA 5). If SCANP is successful, PFA pushes the production number of this node on the derivation stack (PFA 8) and pushes this node on the backtrack stack (PFA 9). It then returns (returning the value of SUCCESS that SCANP returned).

```
PFA  1 procedure PFA (NODE, var SUCCESS);
PFA  2     (* production from above *)
PFA  3   begin
PFA  4     SUCCESS:=TRUE; (* forward mode *)
PFA  5     SCANP (NODE^.DOWNARC, SUCCESS);
PFA  6     if SUCCESS
PFA  7       then begin
PFA  8         PUSHDERI(NODE^.NODENUMBER);
PFA  9         PUSHBACKTRACK(NODE,NIL);
PFA 10         end;
PFA 11     end (* PFA *);
```

SFB is called to initiate a partial parse. It is also called at PFB 10. SFB calls PFB once for each uparc in this node. PFB is passed a pointer to the node to be entered and a pointer to the arc in that node through which the node is to be entered (SFB 6). After all uparcs have been searched (SFB 5 to SFB 8) SFB exits.

```
SFB  1 procedure SFB(NODE);
SFB  2     (* symbol from below *)
SFB  3   begin
SFB  4     ARC:=NODE^.UPARC;
SFB  5     while ARC<>NIL do begin
SFB  6       PFB (ARC^.NEXTNODE, ARC^.ARCINNEXTNODE);
SFB  7       ARC:=ARC^.NEXTARC;
SFB  8       end;
SFB  9     end (* SFB *);
```

PFB is called (at SFB 6) with a pointer to a node (parameter NODE) and a pointer to the downarc through which the node is to be entered (parameter ARC). Since this arc has already matched, SCANP is called in forward mode with a pointer to the next downarc (PFB 5) (note that SCANP in forward mode takes care of pushing nil on the backtrack stack to mark the beginning of a scan). If SCANP is successful, then PFB pushes the production number of this node on the derivation stack (PFB 7); if there is no input left the derivation stack is printed (PFB 9), otherwise SFB is called with a pointer to the node pointed to by the only uparc of this production node (PFB 10); the derivation stack is popped (PFB 11) (to remove this production number); and SCANP is called in backtrack mode (PFB 13). This block (from the pushing of the procedure number on, i.e. PFB 6 to PFB 14) is repeated until SCANP fails, i.e. there are no more alternatives. PFB then returns.

```
PFB  1 procedure PFB (NODE, ARC);
PFB  2     (* production from below *)
PFB  3   begin
PFB  4     SUCCESS:=TRUE; (* forward mode *)
PFB  5     SCANP (ARC^.NEXTARC, SUCCESS);
PFB  6     while SUCCESS do begin
PFB  7       PUSHDERI(NODE^.NODENUMBER);
PFB  8       if NOINPUTLEFT
PFB  9         then PRINTDERIVATIONSTACK
PFB 10         else SFB(NODE^.UPARC^.NEXTNODE);
PFB 11       POPDERI;
PFB 12       SUCCESS:=FALSE; (* backtrack mode *)
PFB 13       SCANP (DUMMYARC, SUCCESS);
PFB 14       end;
PFB 15     end (* PFB *);
```

The backtrack procedures are now described.

When called from SCANP 22, SFAB first pops (NODE,ARC) from the backtrack stack (SFAB 4) and then determines whether this is a terminal node (SFAB 5).

If this is a terminal symbol node, then there are no alternatives; the cursor is backed up one character (SFAB 7) and SFAB returns SUCCESS=false (SFAB 8).

If this is a nonterminal symbol node, then SFAB backs down the last successful downarc by calling PFAB (SFAB 11); if this is not successful, then SCANS is called with a pointer to the next downarc (SFAB 15). If either PFAB or SCANS is successful, then the ARC which matched and the current NODE are pushed onto the backtrack stack (SFAB 17).

```
SFAB  1 procedure SFAB(var SUCCESS);
SFAB  2     (* symbol from above, backtrack *)
SFAB  3   begin
SFAB  4     POPBACKTRACK (NODE, ARC);
SFAB  5     if TERMINAL(NODE)
SFAB  6       then begin
SFAB  7         MOVCURSORBACK;
SFAB  8         SUCCESS:=FALSE;
SFAB  9         end
SFAB 10       else begin
SFAB 11         PFAB (SUCCESS);
SFAB 12         if not SUCCESS
SFAB 13           then begin
SFAB 14             ARC:=ARC^.NEXTARC;
SFAB 15             if ARC<>NIL then SCANS (ARC, SUCCESS);
SFAB 16             end;
SFAB 17         if SUCCESS then PUSHBACKTRACK (NODE, ARC);
SFAB 18         end;
SFAB 19     end (* SFAB *);
```

PFAB, called from SFAB 11, pops NODE from the backtrack stack (PFAB 4) and pops one item from the derivation stack (PFAB 5). It then calls SCANP with backtrack mode (PFAB 7). If SCANP is successful, PFAB pushes the production number of NODE on the derivation stack (PFAB 10) and NODE on the backtrack stack (PFAB 11). It then returns (returning the value of SUCCESS that SCANP returned).

```
PFAB  1 procedure PFAB(var SUCCESS);
PFAB  2     (* production from above, backtrack *)
PFAB  3   begin
PFAB  4     POPBACKTRACK (NODE, DUMMARC);
PFAB  5     POPDERI;
PFAB  6     SUCCESS:=FALSE; (* backtrack mode *)
PFAB  7     SCANP (DUMMYARC, SUCCESS);
PFAB  8     if SUCCESS
PFAB  9       then begin
PFAB 10         PUSHDERI(NODE^.NODENUMBER);
PFAB 11         PUSHBACKTRACK(NODE,NIL);
PFAB 12         end;
PFAB 13     end (* PFAB *);
```

2.2 Database organization and access

In this section we develop the database organization and access algorithms. Let $D=\{d_1,d_2,\ldots,d_n\}$ be the set of database strings and let $P=\{P_1,P_2,\ldots,P_n\}$ be the parses of those strings. Each P_i is a sequence of productions $\{P_{ij}\}_{j=1}^{n_i}$.

Given P and a partial parse p_x of a string x, we need to determine all strings $A \in D$ such that x is a substring of A. There are two cases to consider.

1) If p_x contains no zeroes, then we need only to find all strings d_i such that p_x is a contiguous substring of P_i.

2) If there are r-1 zeroes in p_x such that $p_x=p_x^1\ 0\ p_x^2\ 0\ \ldots\ 0\ p_x^r$ then we determine all strings A such that

$$p_x^1 = P_{i,j_1}\ P_{i,j_1+1}\ \cdots\ P_{i,j_1'}$$
$$p_x^2 = P_{i,j_2}\ P_{i,j_2+1}\ \cdots\ P_{i,j_2'}$$
$$\vdots$$
$$p_x^r = P_{i,j_r}\ P_{i,j_r+1}\ \cdots\ P_{i,j_2'}$$

$$j_{s+1} > j_s'+1, \qquad 1\leq s\leq r-1$$
$$j_t \leq j_t', \qquad 1\leq t\leq r$$

(Note that we cannot have two zeroes adjacent in p_x. The

parser outputs a zero if input is exhausted while scanning the downarcs of a production; the production is then considered (partially) matched so the parser outputs the production number immediately after the zero.)

The database access algorithm proceeds by first finding $P' \subseteq P$ such that for each $P_k \in P'$, every non-zero production number in p_x is also in P_k. This is accomplished as follows. For each production j we define $Q_j \subseteq P$ such that $P_k \in Q_j$ if and only if P_k contains production j. Each of the sets Q_j is stored as part of the database. The algorithm computes P' by taking the intersection of all Q's whose associated productions occur in p_x.

Now that the search has been narrowed to P', the algorithm considers each $P_A \in P'$. P_A is searched for the first occurrence of $p_x^1 = P_{A,j_1} \cdots P_{A,j_1'}$ (as defined above); if no such substring is found then x is not a substring of A, so P_A is discarded and the next element of P' is considered. If p_x^1 <u>is</u> found to be a subpart of P_A, then the identification algorithm attempts to find the first occurrence of p_x^2 in P_A, starting at $P_{A,j_1'+2}$. We could have started at $P_{A,j_1'+1}$, but since the zero in p_x between p_x^1 and p_x^2 means that <u>at least one</u> production is missing in p_x, we can skip over one production before starting the search for p_x^2. As with p_x^1, if p_x^2 is not found in P_A,

then P_A is discarded; otherwise the process is continued with the remaining parts of p_x. If p_x^1 through p_x^r match, then x is a substring of A.

The identification algorithm is implemented in procedure IDENTIFY. The procedure has a bit vector for each production in the grammar, with a bit for every database string. A "1" means that this production is in the parse of the corresponding string; these vectors implement the sets Q above. The array PRODNUMS contains the partial parse of the string to be identified. The procedure INTERSECTPRODUCTIONS intersects all the bit vectors corresponding to the nonzero production numbers in PRODNUMS; the resultant vector (corresponding to P') is stored in POSSIBLESET. Members of POSSIBLESET are indices to parses of known strings.

The procedure GETPRODNUMS also breaks up PRODNUMS into PATTERN[1], PATTERN[2], ... PATTERN[NPATTERNS] (corresponding to $p_x^1, p_x^2, \ldots p_x^r$). The procedure MATCHPARSE tries to match these PATTERNs in order against DB[SENTENCENUM], for each SENTENCENUM in POSSIBLESET (the array DB corresponds to the set P, with DB[i] being the parse of the i^{th} database string). GETNEXTSENTENCENUM is a function which performs the task of picking a SENTENCENUM from POSSIBLESET to pass to MATCHPARSE.

```
IDENT  1 procedure IDENTIFY;
IDENT  2   begin
IDENT  3     GETPRODNUMS;
IDENT  4     INTERSECTPRODUCTIONS;
IDENT  5     while GETNEXTSENTENCENUM do MATCHPARSE;
IDENT  6     end (* IDENTIFY *);
```

K, the position at which MATCHPARSE starts trying to match DB[SENTENCENUM], is initially 1 (MATP 3). I, also initially 1 (MATP 4), is the index of the database PATTERN to be matched. MATCHPARSE then starts a loop (MATP5): it calls the Knuth-Morris-Pratt algorithm [3] (KMP at MATP 6 to MATP 12) to match PATTERN[I] starting at position K in DB[SENTENCENUM], then i.crements I (MATP 13); the loop continues until either MATCH fails, or all the PATTERNs have matched (MATP 15). After each successful return of MATCH, the value of K is one plus the position of the last character to match in DB[SENTENCENUM]. This is where the next PATTERN might start matching, but since there is a zero between this PATTERN and the next, there is at least one missing production; on the next loop MATCH will skip over at least one production in DB[SENTENCENUM], so K is incremented (MATP 14).

If all the PATTERNS match, then SENTENCENUM is an index to the name of a string of which the input is a substring (MATP 16).

```
MATP  1 procedure MATCHPARSE;
MATP  2   begin
MATP  3     K:=1; (* start matching at character 1
                     of DB[SENTENCENUM] *)
MATP  4     I:=1; (* start with PATTERN[1] *)
MATP  5     repeat
MATP  6       LASTMATCH := KMP (PATTERN[I],
MATP  7           PATTERNLEN[I], (* length of PATTERN[I] *)
MATP  8           NEXT[I], (* artifact of
                              Knuth-Morris-Pratt *)
MATP  9           DB[SENTENCENUM],
MATP 10           DBITEMLEN[SENTENCENUM], (* length of
                                             DB[SENTENCENUM] *)
MATP 11           K (* position in DB[SENTENCENUM]
                       at which to start *)
MATP 12           );
MATP 13       I:=I+1; (* next PATTERN *)
MATP 14       K:=K+1; (* skip over an item
                         in DB[SENTENCENUM] *)
MATP 15       until (not LASTMATCH) or (I>NPATTERNS);
MATP 16     if LASTMATCH then PRINTNAME(SENTENCENUM);
MATP 17     end (* MATCHPARSE *);
```

2.3 Results

Timimg tests were made using grammar G2, which describes a language of 1440 strings. Databases of 200, 500, 800, 1100, and 1400 strings were constructed. For each database size, strings were selected at random to construct the database. This was done five times for each size database. From each of these 25 databases 4 sets of 100 substrings were chosen at random. One set consisted of entire sentences (100% of each string chosen), and the other three sets contained fractions of 75%, 50%, and 25% of each string chosen. The substrings were picked at random from within each string chosen.

The 100 substrings of each of the 4 sets were identified relative to the database from which they were chosen. Two methods were used and average timing data was recorded in both cases. The first method was the one presented in sections 2.1 and 2.2. In the second method, the strings, rather than their derivations, formed the database, and the Knuth-Morris-Pratt algorithm was applied directly to the strings themselves. Thus, the comparative timing results should indicate the advantages (or disadvantages) of using the syntactic model, i.e., computing the partial parses of the pattern queries.

For these tests we employed a modification of the procedure IDENTIFY (which was discussed in section 2.2).

Since for each partial parse of a query string one pass is made on the database, it is possible that the algorithm will reconsider database strings which were previously identified (relative to a different partial parse) as containing the query string. The modification consists simply of removing from consideration any database string already identified as a superstring of the current query string. This modification was found to improve performance by 25% for small string fractions. The improvement for complete strings was almost negligible.

Times from the five runs were combined to calculate an average for each of the twenty combinations of database sizes and string fractions. The results are presented in table 1. These times, in milliseconds, represent average time to process a single query string. Four different timing values are indicated in table 1:

1) T_P, the time to find all partial parses

2) T_K, the time to identify these parses using the method of section 2.2

3) T_G, the sum of (1) and (2), that is, the total time to identify the query string

4) T_S, the time to identify the query string using the Knuth-Morris-Pratt method applied directly to the string, rather than to its

	Database Size					String Fraction
	200	500	800	1100	1400	
T_P	18.07	17.38	18.70	17.85	18.30	.25
T_K	50.75	120.11	199.13	269.77	343.13	
T_G	68.82	137.49	217.83	287.62	361.42	
T_s	29.49	73.08	117.58	159.62	205.19	
T_P	28.96	28.66	29.41	29.60	28.79	.50
T_K	28.07	65.74	110.66	145.92	180.51	
T_G	57.03	94.40	140.07	175.52	209.30	
T_s	33.21	81.95	132.04	180.98	229.86	
T_P	34.91	34.61	34.61	35.79	35.49	.75
T_K	15.21	32.94	54.00	74.36	89.53	
T_G	50.12	67.55	88.60	110.15	125.02	
T_s	33.92	83.04	133.56	182.60	233.42	
T_P	25.87	26.05	24.96	25.71	25.91	1.00
T_K	8.05	18.33	27.79	38.28	48.98	
T_G	33.92	44.38	52.75	63.99	74.89	
T_s	32.94	81.00	128.41	176.69	224.79	

Table 1. Execution times in ms. per string for various random databases, access strings, and string fractions

derivation.

These results are represented graphically in figure 3. We make the following observations about the results:

1) For a fixed string fraction, T_S increases linearly with the size of the database, which is, of course, consistent with the complexity analysis of the algorithm given in [3].

2) For a fixed string fraction, T_G increases slower than linearly with the size of the database. This may be due to the intersections descibed in section 2.2 which restrict the search of the database.

3) As the string fraction increases, T_G and T_G/T_S decrease. This is because small string fractions tend to have more partial parses than large string fractions, and one database pass must be made for each partial parse.

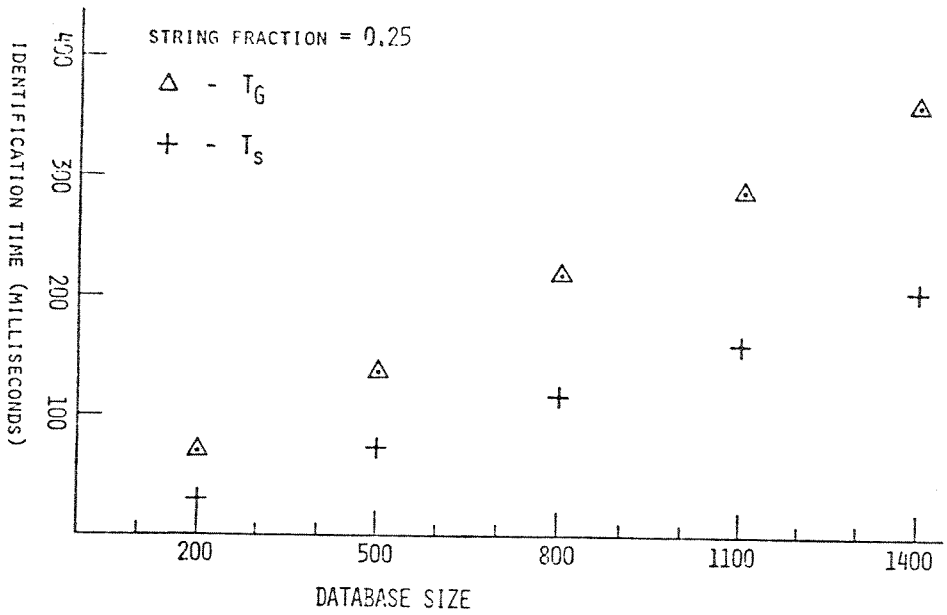


FIGURE 3.1. COMPARISON OF T_G AND T_S AS A FUNCTION OF DATABASE SIZE FOR STRING FRACTION = .25

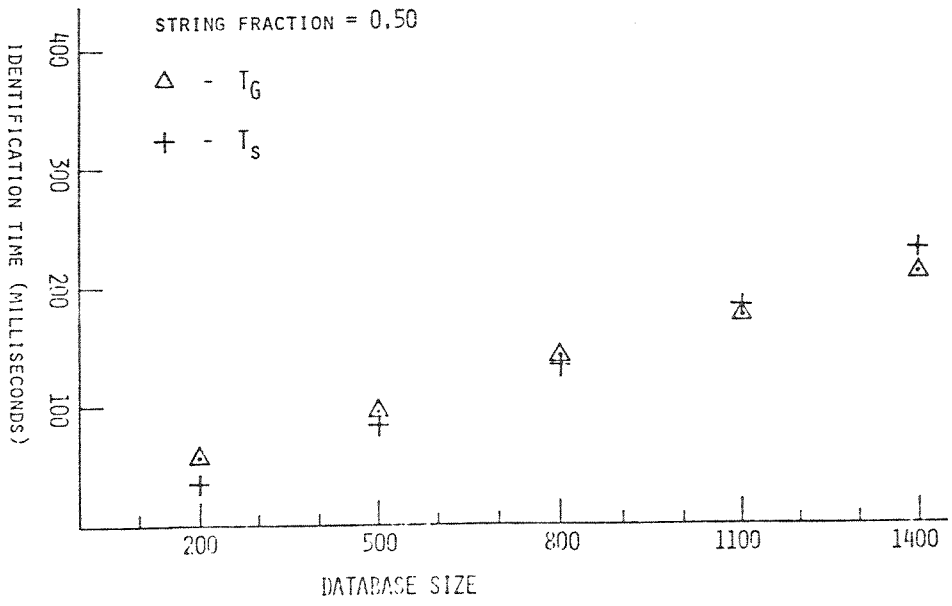


FIGURE 3.2. COMPARISON OF T_G AND T_S AS A FUNCTION OF DATABASE SIZE FOR STRING FRACTION = .50

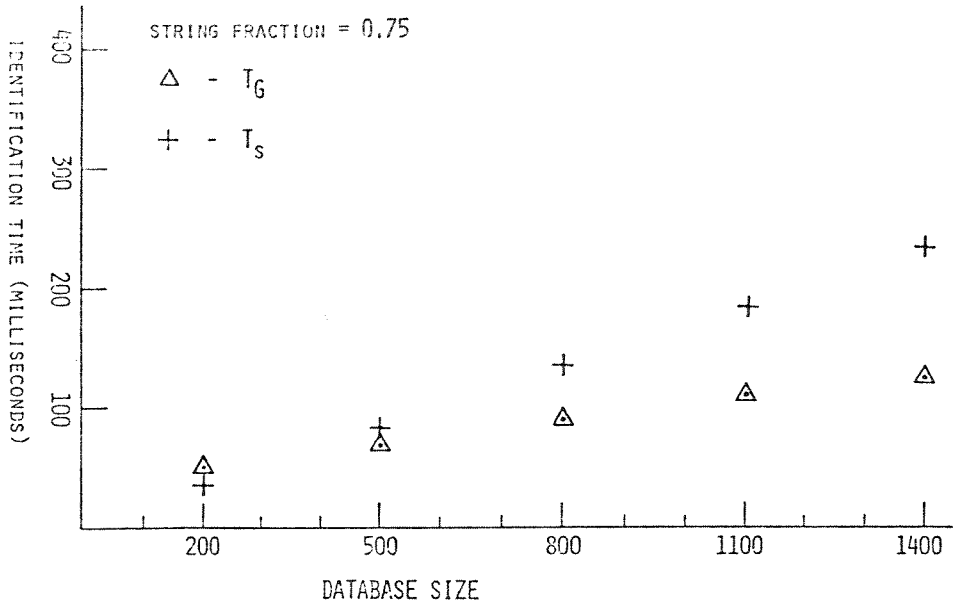


FIGURE 3.3. COMPARISON OF T_G AND T_S AS A FUNCTION OF DATABASE SIZE FOR STRING FRACTION = .75

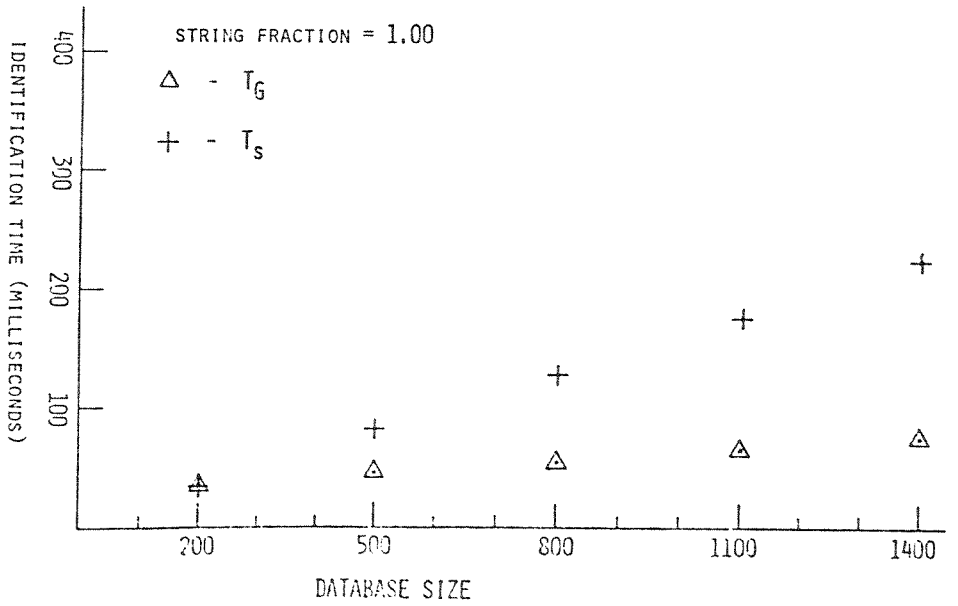


FIGURE 3.4. COMPARISON OF T_G AND T_S AS A FUNCTION OF DATABASE SIZE FOR STRING FRACTION = 1.00

3. Conclusion

The identification of strings using partial parsing and a production oriented database is, on the average over string fraction, slightly faster than identification by a Knuth-Morris-Pratt search of a database of strings. Nonetheless, when the query string is small, the method of this paper requires 75% more time than the more direct method.

With the grammar used in our test cases, the time to perform partial parsing was relatively inconsequential to the total identification time, but realistic two-dimensional grammars will be much more complicated and might cause the partial parsing time to dominate. Methods of speeding up partial parsing will become important.

For example, one simple improvement might consist of keeping at each symbol node a list of positions in the input string at which that symbol has already been found not to match. Since the parser can backtrack and match the leading part of the string in a different way, it can return later to a symbol node which previously failed at the same string location. By checking the list at that node the parser can avoid repeating its previous failure (note that this is similar to some of the improved backtrack procedures described by Gaschnig [4] and Haralick and Elliot [5] and suggests that constraint propagation, as used by Davis and

Henderson [2] might be useful for partial pattern parsing).

Major improvements might also be possible. The parsing algorithm we have presented is a reasonably intuitive, straightforward solution to the partial parsing problem. Perhaps the techniques of more advanced methods, such as LR parsing, could be extended to partial parsing. In addition, stochastic methods might be applied to not only speed up partial parsing, but allow for inexact matching as well (Fu,[6]).

In conclusion, we have studied the partial-pattern database problem. The problem was formally defined in terms of the partial parse of a query string, and a solution was presented. The performance of the algorithm was compared to that of a standard string-matching algorithm and found to be competitive. Finally, we sketched possible methods of improving the speed of our database access algorithms and discussed extensions for handling more realistic partial-pattern database problems.

REFERENCES

1. Fu, K. S., "Introduction to Syntactic Pattern Recognition," in Syntactic Pattern Recognition Applications (ed. K. S. Fu), Springer-Verlag, Berlin, 1977, p.

2. Davis, L. S., and T. C. Henderson, "Hierarchical Constraint Processes for Shape Analysis," to appear in IEEE Transactions on Pattern Analysis and Machine Intelligence

3. Knuth, D. E., J. H. Morris, and V. R. Pratt, "Fast Pattern Matching in Strings," SIAM J. of Comp., vol. 6, no. 2, June 1977

4. Gaschnig, J., "Experimental Case Studies of Backtrack vs. Waltz-type vs. New Algorithms for Satisficing Assignment Problems," Proc. of the 2nd National Conf. of the Canadian Soc. for Computational Studies of Intelligence, Toronto, Canada, July 19-21, 1978

5. Haralick, R. and G. Eliot, "Increasing Tree Search Efficiency for Constraint Satisfaction Problems," 5th Int. Joint Conf. on Artificial Intelligence, Tokyo, Japan, August 1979.

6. Fu, K. S., Syntactic Methods in Pattern Recognition, Academic Press, N.Y., 1974, p.166.