2.PAK: A SNOBOL-BASED PROGRAMMING LANGUAGE FOR ARTIFICIAL INTELLIGENCE APPLICATIONS

JOHN MYLOPOULOS, NORMAN BADLER, LUCIO MELLI AND NICHOLAS ROUSSOPOULOS

DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF TORONTO

Descriptive Terms: Programming language, SNOBOL, backtracking, relational data base, artificial intelligence.

Abstract

This paper describes a programming language for Artificial Intelligence applications which offers

(a) a data base, in the form of a collection of labeled directed graphs where knowledge can be stored

(b) pattern directed information retrieval and pattern invoked function calls

(c) primitive statements which enable the user to construct flexible searching algorithms.

The language is an extension of SNOBOL in its design and implementation and uses SNOBOL's string pattern matching facilities for its own (graph) pattern matching.

1. Introduction

l.pak was designed and implemented in order to facilitate AI research at the University of Toronto. Its design was influenced by other languages designed for similar reasons such as C0NNIVF.R[1,2] , PLANNFR[3], QA4[4], SAIL[5,6], and our decision to implement it as an extension of SNOBOL and keep it relatively inexpensive (in terms of the time and space required for the execution of programs).

This paper only discusses the main features of *l.pak,* how they can be used, their relation to features offered by other languages for AI , and the success of the current implcmentation. More details on the language are available elsewhere [7] It is assumed that the reader is familiar with the basic features of SNOBOL [8].

2. The Data Base

The data base for each *l.pak* program consists of a collection of directed, labeled graphs (hereafter graphs) such as that shown in FIG. 1. A list of transitive and/or intransitive edges is associated with each node, which define either properties of (the object represented by) the node or relations which hold between the node and other elements of the same graph.

A linear order exists for the nodes of each graph defined by special edges labeled NEXT.

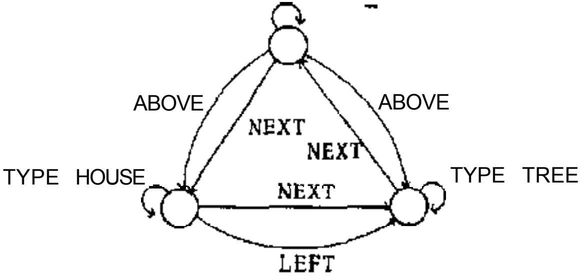


FIG. 1.

Edge labels (properties) consist of one or more atoms separated by underscores. The first atom is the attribute of the property, while subsequent atoms are its modifiers.

Information can be retrieved from the data base by matching (graph) patterns against it.

Patterns arc specified in terms of sequences of path descriptions. For example, the pattern

<$X,(LEFT_AARB?YJ, (FAR).SW-

contains one path description which will match paths that

(a) Begin at the node which is the value of atom X ,

(b) Move along an edge whose label has LEFT as attribute and is followed by at least one modifier,

(c) Move along an edge whose label has FAR as attribute,

(d) End at node $W.

If such a path connects nodes $X and $W for a particular data base, the pattern match succeeds and the modifier of LEFT is assigned to atom Y , while the graph pattern match returns $W , the last node vi?ited during the match. Patterns are evaluated by the function SEARCH.

AARB is a special property pattern which will match any atom. *l.pak* offers several other built-in property patterns and operators similar to those offered by SNOBOL to help the user specify classes of edge labels in his graph patterns Thus (graph) pattern matching is essentially driven by property pattern matching and can be easily implemented using the string pattern matching facilities in SNOBOL.

Note that there may be several paths which will match the same pattern. For example, the pattern match

SEARCH(<$X,(LEFT),(ABOVE^FAR)>)

could return node n1 or nZ In FIG. 2. The pattern match however will return just one of the two nodes .

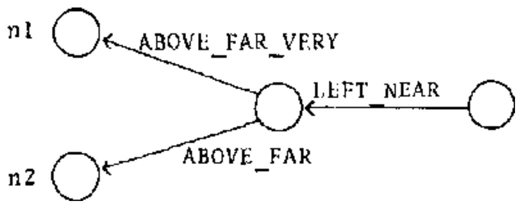


FIG. 2.

We will show later how to generate all the matching alternatives.

Information can be added to or deleted from the data base through the special functions ADD and DEL.

ADD's main function is to create new information with a graph as context. For example, the function call

ADD(<$X,(LEFT),(FAR),?W>)
will create enough new edges and nodes to make the pattern match of its argument against the data base successful. In the process, a (node) value will be assigned to W . ADD will also perform a simple consistency-redundancy test for each new edge attached to a node, with respect to the edge list of that node. Thus an intransitive edge labeled *TYPE HOUSE* will not be added to the edge list of a node which already contains an edge labeled TYPE_H0USE TALL because it is redundant.

An example of a DEL function call is
DEL(<?X,(LEFT),-$Y,(RTGHT)>)
where the special operator '-' specifies that the node or edge it operates on must be deleted. For this function call, a pattern match takes place first, and if unsuccessful the call fails; otherwise the node $Y is deleted along with all the edges that point, to or away from it.

l.pak also offers SNOBOL tables and arrays as means for representing information. Moreover, as in SNOBOL, the user can define his own data types. Explicit list facilities using CONS, CAR and CDR and list notation are provided. Expressions such as
$L = ($A, $B, $C, $D)
may be used to construct lists; here the value of L is a four element list.

3- Program Structure and Control

l.pak statements are similar to their SNOBOL cousins. For example, the statement
A: DEL(<$X, (LEFT) , - (R1GIITJVB0YE) >) :S(A)F(B)
is labeled A , and will attempt to delete the edge whose label matched R1GHT_AB0VE, if the pattern match of <$X,(LEFT),(R]CHT_ABOVE)> against the data base succeeds. If the DEL function call succeeds, control is passed back to A , otherwise control is passed to the statement labeled B .

Each *l.pak* statement will succeed or fail and this can be used for program control. Unless otherwise specified, control will pass to the next statement of a *l.pak* program. Thus backtracking as encountered in PLANNER and QA4 is only offered during graph pattern matching in *l.pak.* A different kind of backtracking will be discussed later.

4. Function Requests

4.1. Explicit function requests

l.pak functions can be defined through PDEFINE. For example, the statement
PDEFINE(NEW(X:NODE]NULL,Y:PROPERTY|PATTERN,? Z)
W_W1,NEW)
defines a function named NEW with three formal parameters; the first must be of type NODE or have as value the null string, the second of type PROPERTY or PATTERN, while the third is called by result. Thus the statement
NEW($X1,$Y1,$Z1)
will fail even before NEW is called because the third argument is not called by result, while
NEW($X1,$Y1,?Z1)
will proceed with the execution of the function call if $X1 is a node or the null string, and $Y1 is a property or a (graph) pattern.

The specification of allowable types for each formal parameter <u>does not</u> mean that the type of function arguments is fixed inside a function call; it only helps to check whether the function request makes sense.

The same function name may be used inside several PDEFINE calls, thus giving the same name to several different functions. This means that a statement such as
NEW($X1,$Y1,?Z)
may cause several function calls until one is found that succeeds. In fact, the function name specified by a function request in *l.pak* can be a string pattern, as in
(T1I|AX) ($X,$Y,?Z)
which will cause the call of any function whose name matches the string pattern (THJAX) (i.e., begins with TH or AX) and whose parameter description is matched by the arguments of the function request.

This feature offers the user flexibility in specifying a function request similar to that offered by theorem provers where each axiom can be considered as a function without a name to be called whenever there is a suspicion that it may be of use.

Unlike SNOBOL, *l.pak* treats field designators of programmer-defined data types, array references and table lookups as special cases of a function request. Thus, if the user writes
CAR($X)
1.pak will try to treat this as a field designator, then as an array reference, then as a table lookup, and if all these attempts fail, it will start calling functions named CAR until one of them succeeds.

4.2 Implicit function requests

It is often the case that information not stored in the data base is actually true in the universe of discourse. For example, a node may represent a house which has been recognized as one of the objects represented in a line drawing, and the question "Does there exist an object to the left of the house?" may be asked. The user may attempt to answer it by writing
SEARCH (<-$X, (LEFT_* FCN) ,?Y>)
where X points to the node representing the house, and *,FCN are special modifiers to be discussed later. If the pattern match fails, either there is no such object, or it exists but it is not represented by a node in the graph where $X is imbedded. We would like to keep the value of the pattern match independent of these two possibilities and have therefore introduced the notion of <u>implicit function requests</u>■ Informally, such function requests will be invoked by the system in an attempt to construct information needed for the successful evaluation of a pattern match whenever the special modifier FCN appears at the end of a property pattern. * is also a special modifier which specifies that modifiers that follow it should be treated differently by *l.pak* than atoms that precede it. In the previous example, if the system has found no edges which leave $X and match LEFT (not LEFT *FCN), it will therefore call functions named" LEFT with implicit argument $X , looking for one that succeeds.

Thus the pattern match
SEARCH(<$X,(DIMENSIONS AARB?Y AARB?Z AARB?W
* FCN)>)

will either match an edge label to the property pattern

```
        DJMENS10NS_AARB?Y  AARB?Z  AARB?W
```

and will return its modifiers through Y, 2 and W , or it will call a function named DIMENSIONS which has three formal parameters, all called by result, which will attempt to find the dimensions of $X . In this case the modifiers of the property for which we are trying to establish an edge were used as arguments of the implicit function call, while the attribute was used as a function name. In the example

```
  SEARCH(<$X,  (TYPE_HOUSE_TALL_VKRY_*__FCN)>)
```

on the other hand, we may want to establish the existence of an (intransitive) edge labeled TYPEJIOUSE_TALL_VERY by first calling the function TYPE with only argument the node $X , then the function HOUSE with arguments $X and the output of TYPE, then TALL and finally VERY. In other words, an attempt to establish an edge in the data base may cause several function calls.

5. Generator Calls

Generators were introduced by PLANNER and used extensively by CONNIVER where they provide one of the most important language constructs. *l.pak* offers them too, as a method of generating alternatives.

A generator is defined by a piece of code and is assigned a name. For example,

```
  GEN.INT,  INTEGER;
       $INT = $INT+1;
       $DOMAIN = C0NS($]NT,$D0MA1N)  : (EXIT);
  END.INT;
```

defines an integer generator named INT. This generator simply considers the first element stored in its DOMAIN list (which is similar to CONNTVER's possibilities list), increments it by 1, stores the result in DOMAIN, and returns it as value. EXIT specifies that execution of this generator call is complete and that its DOMAIN list should be kept active for future calls.

A generator can be used once it has been bound to an atom. Thus

```
                 $X  <= INT(5)
```

assigns the generator INT to X and initializes its DOMAIN list to (5). Now whenever we write X< > , the INT generator will be evaluated returning another integer. Note that there can be several active copies of the same generator, each bound to a different atom. For example,

```
                 $X  <-  INT(51;
                 $Y  <«  INT(7);
            A:   $OUTPUT  =  X<  >  +  Y<  >;
                 LT($OUTPUT,100)   :S(A);
```

will assign to X the generator INT with its DOMAIN list initialized to (5). It will also assign to Y the generator INT with its DOMAIN list initialized to (7). 5+7 will then be evaluated and assigned to OUTPUT. As in SNOBOL, any string assigned to OUTPUT is automatically printed. It is then checked whether the value of OUTPUT is less than 100 and if so 6+8 is added and printed, then 7+9, 8+10 etc.

In general, the user can pass within the angle brackets a list of expressions to be appended to the DOMAIN list of a generator each time he calls it. He can also pass an argument to be used for that particular generator call.

There are several built-in generators. For example, NODES will generate all the nodes that lie at the end of a path which matches a given graph pattern. Thus if we write

```
    $Z  <=  NODES(<$X,(LEFT),(ABOVE_FAR)>)
```

the first time Z< > is encountered with background the graph shown in FIG. 2, it will return, say, node n] , the second time node n2 , and if it is called again it will fail.

6. More on Backtracking

In some cases backtracking (i.e., reset of the program state in case of failure) will not be necessary, but in others it will save the programmer considerable effort. *1.pak* allows a form of backtracking by extending the feature of declaring variables local to a function or a generator in several directions:

(a) All variables which appear in a function or generator body can be declared local for a particular function or generator call by using the keyword VLOCAL during the function definition or generator definition or binding. LOCALness may be specified as dependent on the success of failure of a function or generator call by using SVLOCAL or FVLOCAE instead of VLOCAL. Thus FVLOCAL defines a backtracking situation (i.e., reset in case of failure) for program variables only for "the function or generator it is associated with.

(b) Changes made to the data base can also be declared local by using SDLOCAL, FDLOCAL or DLOCAL.

If the user wants a reset of variables and the data base state, he can use SLOCAL, FLOCAL or LOCAL. This way he has some flexibility in specifying exactly which changes in his program he considers reversible and under what, conditions .

7. Examples

This section describe? three simple *l.pak* programs which demonstrate the features already discussed and point out a few others that arc not as important.

Suppose that we want to define a collection of functions named LEFT which somehow express adequately our own notion of what LEFT means (geometrically). These functions will be defined with respect to a list, $LIST, of objects and it will be assumed that there exists a function REL.LEFT which succeeds or fails depending on whether the object represented by its first argument is to the left of the object represented by its second argument.

First we give a definition of LEFT which postulates its transitivity

```
     PDEFINE(LEFT(TA1L:NODE,HEAD:NODE)Z_AUX,
                                  LEFT,DLOCAL);
 BEGIN.LEFT;
     $Z  <-  NODES(<$TAIL,(LEFT)>);
     $AUX  =  Z<  >    :F(FRETURN);
 A:   ADIH<$AUX,(TRIED)>)     :F(B);
     I DENT($AUX,$HEAD)        :S(RETURN);
 B;   $AUX  -  Z<(<$AUX,(LEFT)>)>  :S(A)F(FRETURN);
 END.LEFT;
```

This function will be evaluated as follows for given $TAIL and $HEAD:

(a) Z is bound to the NODES generator described in section S with its DOMAIN list initialized *to* the pattern <$TAIL, (LF.FT)> . Thus the first call of Z will return a node to the LEFT of $TA1L which is assigned to AUX.

(b) Each node assigned to AUX is labeled TRIED by ADD. If $AUX already has an intransitive edge labeled TRIED, ADD fails (because it cannot change the data base) and another node to the LEFT of $TAIL is assigned to *AUX.*

(c) It is checked whether $AUX is IDENTical to $HEAD, and if so the function call RETURNS;

(d) Otherwise, another node to the LEFT of $TAIL is assigned to AUX and the new graph pattern <$AUX, (LF.FT)> is appended to the DOMAIN list of Z .

When all the nodes to the LEFT of $TAIL have been considered, Z will Start generating nodes to the LEFT of nodes to the LEFT of $TAIL, and this will be repeated until all the nodes to the LEFT of nodes... to the LEFT of $TAIL have been tried. Because of the third argument of PDEFINE, all changes made to the data base will be erased when the call to LEFT is complete, also AUX, Z will be reset to the values they had before the function call.

Thus this definition of LEFT defines a breadth-first search of the graph where $TAIL is imbedded in an attempt to find $HEAD, and it has been formulated by using the generator feature. it is interesting to compare it with the following *l.pak* function which also postulates the transitivity properly of LEFT, by relying on graph pattern matching and implicit function requests:

```
          PDEFINE (LEFT (TAIL: NODE ,1IEAI): NODE),
                                       SLEFT);
 BEGIN.SLEFT;
          SEARCH(<$TAIL,(LEFT),$IIEAD>) :S(RETURN);
          SEARCH(<$TAIL,(LEFT),(LEFT_*_FCN),
                    $HEAD>) :S(RETURN)F(FRETURN);
END.SLEFT;
```

Here it is first checked whether there is an edge with attribute LEFT connecting $TAIL to $HEAD, and if this is not the case, it is checked whether there is a node, say nl , such that there is an edge with attribute LEFT connecting $TA1L to nl , and nl and $HEAD can be connected with an edge labeled LEFT. In checking for the latter condition, the user has specified that implicit function requests involving LEFT-named functions are allowed. This is a recursive definition of transitivity for LEFT in other words. The searching algorithm it defines is depth-first and may enter an infinite loop for graphs representing geometrically strange worlds.

The second definition of LEFT accepts two nodes as arguments and succeeds or fails depending on whether the object represented by the first node is to the LEFT of the object represented by the second

```
         PDEFINE[LEFT(TAIL:NODE,HEAD:NODE)A   B,

                                          T LEFT.)7
 BEGIN.TLEFT;
         SEARCH(<$TAIL,(REPR_AARB?A) :
                   $HEAD,(REPR_AARB?B)>)
                                        :F(FRETURN);
         REL.LEFT($$A,$$B) :S(RETURN)F(FRETURN);
 END.TLEFT;
```

The graph pattern match executed by SEARCH finds the atoms which modify REPR on intransitive edges associated with $TAJL and $HEAD and assigns them to A and B respectively. ':' specifies the end of one path description and the beginning of another. The values *of* the two *atoms* assigned to A *and B* are the elements of $L1ST represented by $TAIL and $HEAD (FIG. 3). Once these atoms are found, REL.LEFT can be called with arguments the objects object.i, object,j, represented in some way.

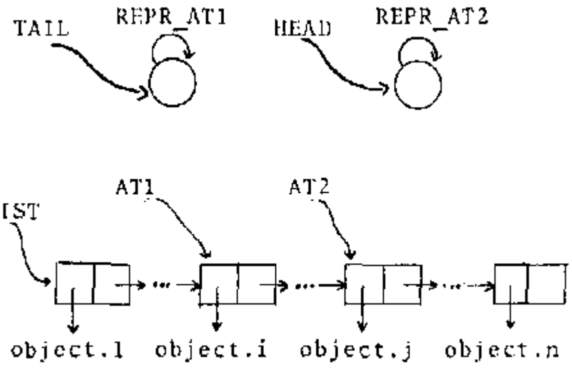


FIG. 3.

A third possible definition of LEFT as a generator returns nodes representing objects to the LEFT of a given node $TAIL which is passed as message to the generator, and is also a global variable:

```
                 $Lt-ir    <=    LIiFTDEF(#$rAILj
```

This statement assigns to LEFT the generator LEFTDEF with its DOMAIN list initialized to the unevaluated expression $TAI1. ('W' keeps its operand unevaluated until it is encountered at execution time, and is therefore similar to the SNOB0L, unary operator '*') - Below we define LEFTDEF.

```
 GEN.LEFTDEF,  NODE;
 GA:    SLEFTDEF  =  SEARCH (– $LEFTDEF , (NEXT)> ) ;
        I DENTf$LEFTDEF,$TAIL)     :S(FRETURN) ;
        SEARL'H(<$TAIL, (LEFT_*_FCN) ,$LEFTDEF>)
                                            :F(GA>;
        $D0MA1N  =  ($LEFTDEF)         : (EXI'I'") ;
 END.LEFTDEF;
```

Whenever this generator is executed, LEFTDEF is first assigned the value of TAIL (this is done automatically by the system). Then the NEXT node is found and assigned to LEFTDEF; NEXT defines a circular order on the nodes of the graph where STAIL is imbedded, and we are using it here in order to traverse the graph. It is checked whether SLEFTDEF is IDENTical to $TAIL, which would mean that we are back at the starting node and there are no more nodes to the LEFT of $TAIL; if not, it is checked whether the current value of LEFTDEF is to the LEFT of $TA1L. Implicit *function* requests are allowed for this check. If the answer is negative, control returns to the statement labeled GA and another node is assigned to LEFTDEF', otherwise the DOMAIN list of LEFTDEF is set to ($LEFTDEE), a one-element list, and we EXIT. Next time this copy of LEFTDEF is called, search will resume with the NEXT node in the graph where $TAIL is imbedded until they have all been considered.

The user can now write

```
    SEARCH(<$X,(LEFT  *  FCN),$Y>)
or    SEARCH(<$X,(LEFT~*_GEN),?Y>)
```

and expect the system to either find the necessary information in the data base or to call the appropriate functions or generators (depending on whether the special modifier is

FCN or GEN) in an attempt to construct it.

8. Discussion

1.pak has been implemented in SPITBOL[9], an efficient version of SNOBOL. SPITBOL uses both a compiler and an interpreter, offers most SNOBOL features (in particular the function EVAL), plus a few more, has a very fast garbage collector and handles user-defined data types very efficiently. It requires approximately ~50K bytes and runs several times faster than the BTL implementation of SNOBOL.

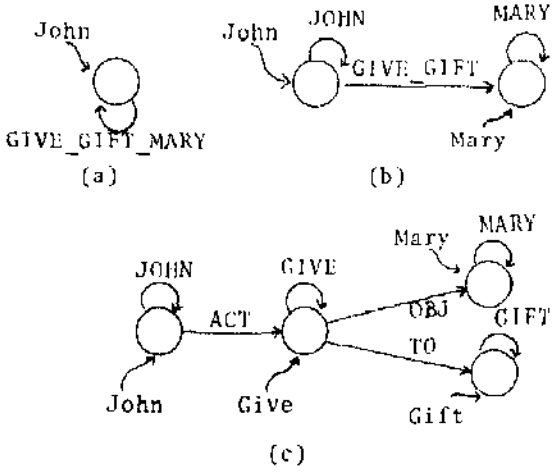
Some of the reasons that led us to choose SNOBOL over other candidate languages are listed below:

(a) It offers pattern matching facilities. This has helped the design and implementation of graph pattern matching; moreover, SNOBOL users will have no difficulty adapting to the graph pattern matching formalism since it is an extension of string pattern matching.

(b) It offers tables and user-defined data types. These features were used extensively during the implementation of the *1.pak* system, and are offered by *1.pak* at very little extra cost.

(c) SNOBOL's control structure is unusual but flexible and well-suited to AI applications programming. *1.pak* offers most of that control structure, in addition to the various shades of the LOCAL feature, function requests, generator calls etc.

Labeled graphs have already been used for the representation of knowledge (e.g., Palme [10], Rumelhart et al [11]). *1.pak* graphs have the additional feature however that the user has a choice of representing a piece of information structurally or as a property. Which form he chooses should depend on how often the parts of this piece of information will be retrieved and manipulated independently of each other. For example, the statement "John gives a gift to Mary" could be represented (rather crudely and with various subtleties of the sentence's meaning ignored) by the graph shown in FIG. 4(a), 4(b) or 4(c), depending on whether we will be referring explicitly to the gift or Mary and their properties, or will simply refer to them as parts of a property John has.



(a) (b)

(c)

FIG. 4.

Unlike PLANNER etc., *1.pak*'s data base offers only partial associativity. This may be inconvenient for the user in certain cases, but it offers him more control over his data base's thirst for memory space. More convential data structures (arrays, tables, user-defined data types) are also available in *1.pak at very little expense for the 1.pak* system since they are mostly handled by SNOBOL.

Graph pattern matching offers many features found in PLANNER in that a similar backtracking mechanism is used and implicit function requests can be considered as consequent theorem calls. If the user agrees with Sussman and McDermott's criticism of PLANNER's backtracking [2], he can switch to a programming style favoring generators where he has more control over the backtracking mechanism he uses. We feel that both features will be found useful.

The *1.pak* implementation uses both a compiler and an interpreter and requires a minimum of ~140K (this includes ~50K for the SPITBOL system). There are plans to use *1.pak* for question-answering, scene analysis and natural language understanding to test it and find which features are useful and which should be made more prominent and which ones should be modified.

Acknowledgements

We would like to thank Carl Hewitt for several useful suggestions, also Walter Berndl for helping us with the implementation and portions of the design of *1.pak*. This research was partially supported by DOC and DRB grants.

References

1. McDermott, D.V., Sussman, G.J. The CONNIVER reference manual. MIT AI Memo. 259.
2. Sussman, G.J., McDermott, D.V. From PLANNER to CONNIVER: a genetic approach. FJCC, 1972, 1171-1180.
3. Hewitt, C. Description and theoretical analysis of PLANNER. MIT AI-TR-258, 1972.
4. Derksen, J., Rulifson, J.F., Waldinger, R.J. The QA4 language applied to robot planning. FJCC, 1972, 1181-1192.
5. Swinehart, D., Sproull, R. SAIL. Stanford AI Project, Operating Note No. 57.2, Janaury 1972.
6. Feldman, J.A., Low, J.R., Swinehart, D.C. and Taylor, R.H. Recent developments in SAIL. FJCC, 1972, 1193-1202.
7. Mylopoulos, J., Badler, N., Melli, L., Roussopoulos, N. An introduction to *1.pak*, a programming language for AI applications. TR-52, Department of Computer Science, University of Toronto, May 1973.
8. Griswold, R.E., Poage, J.F., Polonsky, I.P. The SNOBOL4 programming language. Prentice-Hall, 1971 (second edition).
9. Dewar, R.B.K. SPITBOL, Version 2.0. Illinois Institute of Technology, 1971.

10. Palme, J. Making computers understand natural language. In Artificial Intelligence and Heuristic Programming, Findler N." and Meltzer, B. (Eds .). Edinburgh University Press, 1971.

11. Rumelhart, D,, Lindsay, P.H., Norman, D.A. A process model for long-term *memory.* In Organization of Memory, Tulving, E. and 'Donaldson, W. (Eds .), Academic Press, 1972.