AN INTRODUCTION TO SNePS

Stuart C. Shapiro
Computer Science Department
Indiana University
Bloomington, Indiana 47401

TECHNICAL REPORT NO. 31

AN INTRODUCTION TO SNePS

STUART C. SHAPIRO

REVISED: DECEMBER, 1976

An Introduction to SNePS

Stuart C. Shapiro
Computer Science Department
Indiana University
Bloomington, Indiana 47401

Abstract

SNePS (Semantic Network Processing System) is a system for building directed graphs with labelled nodes and edges and locating nodes in such graphs according to graph patterns. Rather then being a general system for processing labelled digraphs, SNePS is restricted in certain ways, appropriate for its intended use--to model "semantic" or "cognitive" structures. SNePS may be used interactively by a human to explore various approaches to semantic representation, or it may be used as a collection of functions by a more complete natural language understanding program. This paper gives a user-oriented introduction to SNePS.

Introduction

SNePS (Semantic Network Processing System) is a system for building directed graphs with labelled nodes and edges and locating nodes in such graphs according to graph patterns. Rather than being a general system for processing labelled digraphs, SNePS is restricted in certain ways, appropriate for its intended use -- to model "semantic" or "cognitive" structures. SNePS, a revised versio of MENTAL [Shapiro 1971a, 1971b], is written in LISP 1.6 and runs on a DEC-system-10.

Edge labels represent binary semantic relations which are used to structure the network and about which no information can be stored in the network. For example, the _cases_ of Fillmore, 1968, might be such labels. The user of SNePS is free to choose and declare his own set of labels. There are two kinds of edges, _regular_ edges and _auxiliary_ edges. Regular edges come in pairs, one representing a _descending_ relation, the other the _ascending_ converse of the first. Auxiliary edges do not come in pairs and do not have converses. If a path of descending edges goes from node n to node m, we will say that node n _dominates_ node m.

There are three kinds of nodes in the network: _constant_, _non-constant_, and _auxiliary_ nodes. Auxiliary nodes are connected to each other and/or to other nodes only by auxiliary edges. Constant nodes represent semantic concepts, including anything about which information may be stored in the network. Nodes which dominate no other node are called _atomic_ nodes. Atomic constants are called _base_ nodes and atomic non-constants are called _variable_ nodes. Non-atomic nodes are called _molecular_ nodes. A molecular node which dominates any variable node is called a _pattern_. Molecular nodes that do not dominate variable nodes are called <u>assertions</u>. An assertion is also a constant in that it represents a particular semantic concept.

Nodes are labelled with LISP atoms. Some nodes are labelled by the user. Others are labelled by SNePS using Mxxxx where xxxx is a series of digits. Normally the user labels only base nodes, although it is possible to create user labelled molecular nodes. It is impossible to create user labelled variable nodes.

A major restriction designed into SNePS is that the user cannot add a new edge connecting two already existing nodes. This would amount to changing an assertion or concept into a different one. An alternative view of this restriction is that whenever a relationship between two or more existing nodes is added, a node representing that this relationship holds is also added. An implication of this restriction is that it is impossible to have an edge connecting two user labelled nodes.

One common use of auxiliary nodes is as <u>SNePS variables</u>. These are to be distinguished both from LISP variables and from variable nodes. An atom may have a LISP value and also be the label of a SNePS variable with a different value, which will always be a set of one or more nodes. The system also creates auxiliary nodes with the same labels as edges to maintain certain information about them.

Several SNePS variables are pre-defined in and maintained by the system. They and their values are:

NODES	The set of SNePS nodes
VARBL	The set of variable nodes

RELST	The set of descending edge labels
AUXRELST	The set of auxiliary edge labels

Auxiliary edges are used by SNePS for the purposes listed below. Additional ones may be declared by the user. Generally this is done to put "hangings" (see Friedman, 1973. Brown et al, 1974 added this feature to their version of SNePS, calling it a property as opposed to a relation) on nodes, i.e. to point from some SNePS node to some LISP structure that is not a node. Auxiliary edges pre-defined by the system and their uses are:

CONV	Points from the label of a regular edge to the label of its converse edge
:VAL	Points from a SNePS variable to each node in the list that is its value.
:VAR	Points from each variable node to the auxiliary node T.
:SVAR	Points from each pattern node to each variable node that it dominates.

There are mechanisms for the user to create temporary nodes. These are not placed in the NODES or VARBL sets, and unlike normal SNePS nodes, when a temporary node, t, is created with a regular edge pointing to another node, n, no converse edge is created pointing from n to t. Temporary SNePS labelled nodes may be accessed by making them (or nodes dominating them) the value of SNePS variables or LISP variables, but once they are no longer accessible, they disappear (are garbage collected).

The SNePS user language consists of a set of functions for which the unquote convention (see Bobrow and Raphael, 1974) holds. An atom refers to itself unless it is unquoted. A list is either a SNePS function reference or a list of elements which can be atoms, unquoted atoms or SNePS function references.

There are several types of unoqutes:

*FOO	The previously assigned SNePS value of the SNePS variable, FOO.
#FOO	A newly created constant node, which is assigned as the new SNePS value of FOO.
$FOO	A newly created variable node, which is assigned as the new SNePS value of FOO.
%FOO	A newly created temporary variable node, which i assigned as the new SNePS value of FOO.
?FOO	The SNePS value of FOO is determined during search as described below.
(<s-func> <term>...)	If <s-func> is a SNePS function, the SNePS value of the form
(↑ <sexp>)	The LISP value of <sexp>.

Description with Examples

The following description will contain many examples of SNePS usage. In all examples, lines beginning with ** are the first lines of the user's input to SNePS. Subsequent input lines begin with *. Lines without these prompts are SNePS output. Diagrams of the network will be displayed in which the newly created structures will be enclosed in dotted lines. Regular edges will be show as labelled solid lines with arrowheads, auxiliary edges as labelle dashed lines with arrowheads. SNePS nodes will be shown as small circles with labels, auxiliary nodes as labels only. All examples are for the purpose of describing SNePS and are not to be taken as this author's complete current proposal for the actual contents of a model of human semantic memory.

The user declares regular edge labels with the SNePS function DEFINE, which is given pairs of relations. The first of each pair is considered to be the descending relation. Each label is stored as an auxiliary node with the auxiliary relation CONV to its converse label. Each descending relation is added into the set which is the SNePS value of the SNePS variable RELST.

```
**(DEFINE MEMBER MEMBER* CLASS CLASS*))
(MEMBER MEMBER*)
(CLASS CLASS*)
(DEFINED)

**((DEFINE A A* V V* O O* I I*))
(A A*)
(V V*)
(O O*)
(I I*)
(DEFINED)
```

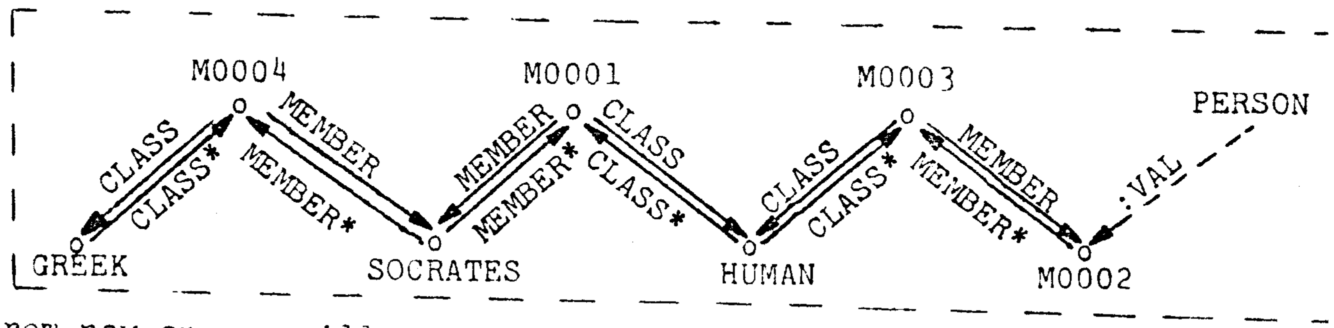
A node is created and its associated network built by the BUILD function. The value of the BUILD function is a list of the created node. The arguments to BUILD are, alternately, an edge label and a node or set of nodes. The second example below demonstrates one of the unquotes.

```
**((BUILD MEMBER SOCRATES CLASS HUMAN))
(M0001)

**((BUILD MEMBER #PERSON CLASS HUMAN))
(M0003)

**((BUILD MEMBER SOCRATES CLASS GREEK))
(M0004)
```

The network built by these instructions is shown below.



From now on, we will not show ascending relations in diagrams of the network, although they should be assumed to be present.

The user may have pieces of the network printed for his inspection by using the DESCRIBE function.

```
**((DESCRIBE M0001 M0003))
(M0001 (CLASS (HUMAN))(MEMBER (SOCRATES)))
(M0003 (CLASS (HUMAN))(MEMBER (M0002)))
(DUMPED)
```

The function FIND is used to locate nodes in the network.

```
**((FIND MEMBER SOCRATES CLASS HUMAN))
(M0001)
```

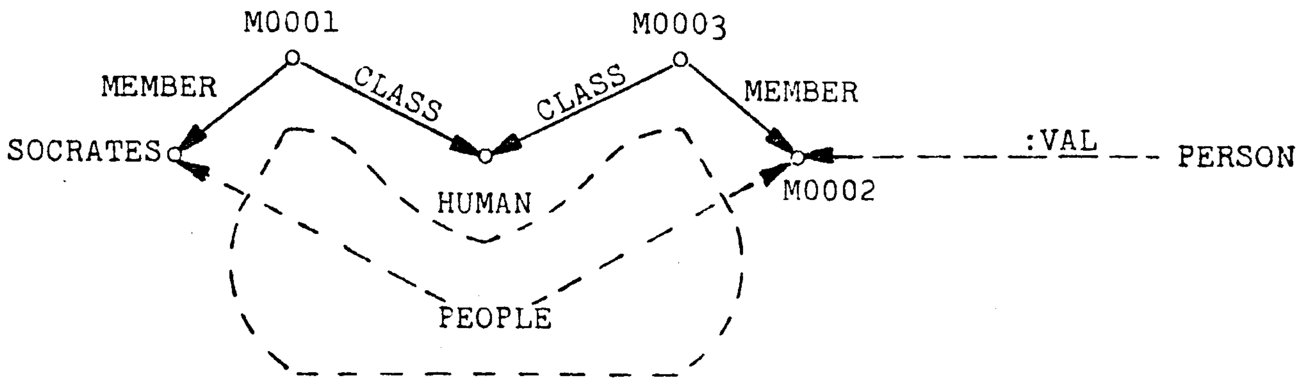
The value of FIND is a list of the located nodes, so calls to FIND may be embedded in other functions.

```
**((FIND MEMBER* (FIND CLASS HUMAN)))
(M0002 SOCRATES)

**((FIND MEMBER*(FIND CLASS HUMAN)
*       MEMBER*(FIND CLASS GREEK)))
(SOCRATES)

**((FIND MEMBER ?PEOPLE CLASS HUMAN))
(M0003 M0001)
```

This last is a simple use of the ? unquote. It requires that each located node have a MEMBER relation to some node and places all these nodes in the SNePS value of PEOPLE. This results in the following addition to the network.



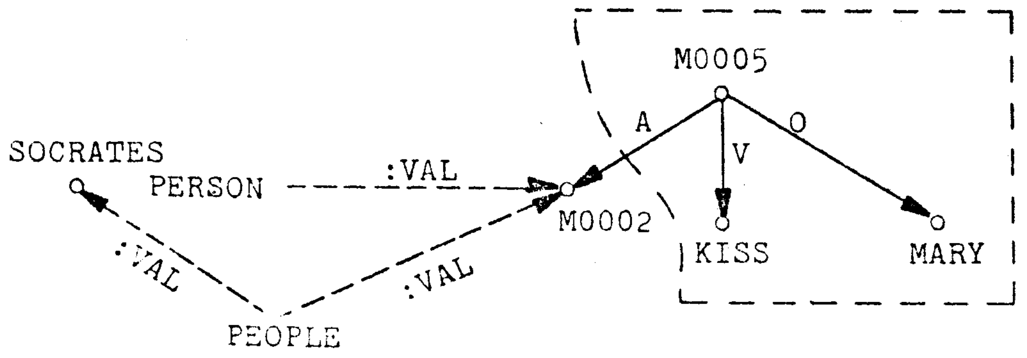
To simply print the value of a SNePS variable, the following use of the * unquote suffices.

```
**(*PERSON)
(M0002)

**(*PEOPLE)
(M0002 SOCRATES)
```

Assigned variables may also be used within functions.

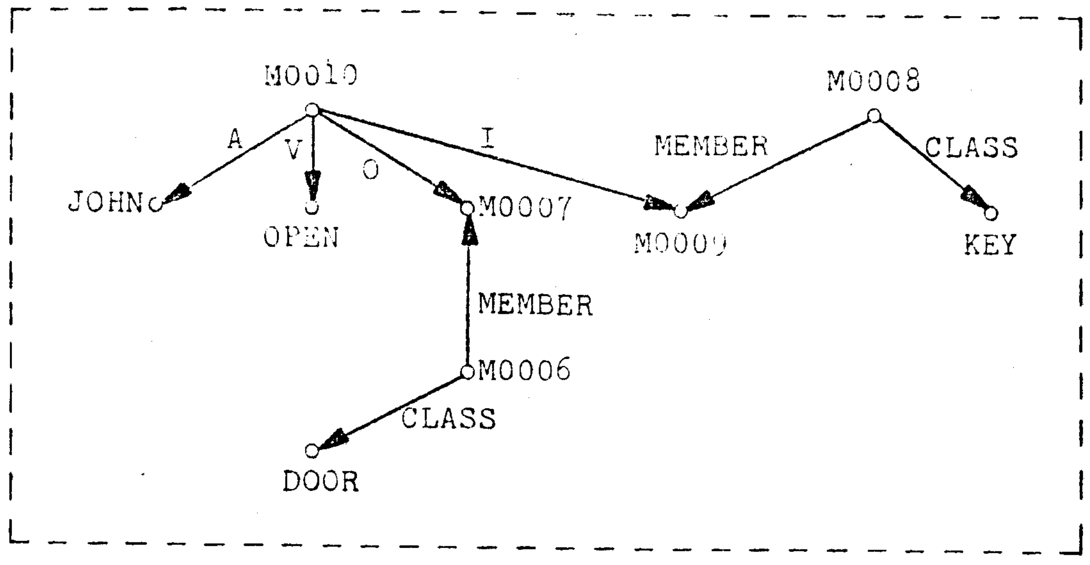
```
**((BUILD A *PERSON V KISS O MARY))
(M0005)
```



```
**((FIND O* (FIND A *PEOPLE V KISS)))
(MARY)
```

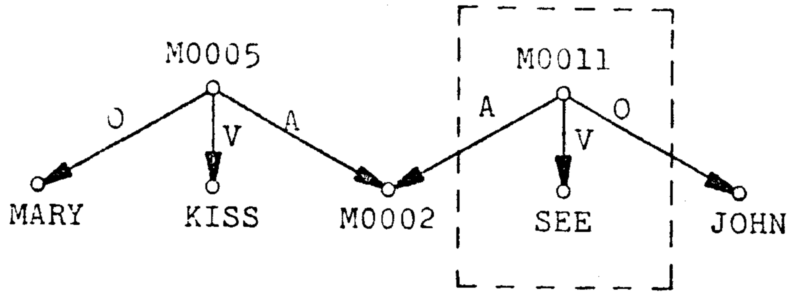
BUILDs may be embedded within BUILDs to simulate the several sentences that underlie a single surface sentence. For example, a simplified representation of "John opens a door with a key" might be:

```
**((BUILD A JOHN V OPEN
*          O (BUILD MEMBER* (BUILD CLASS DOOR))
*          I (BUILD MEMBER* (BUILD CLASS KEY))))
(M0010)
```



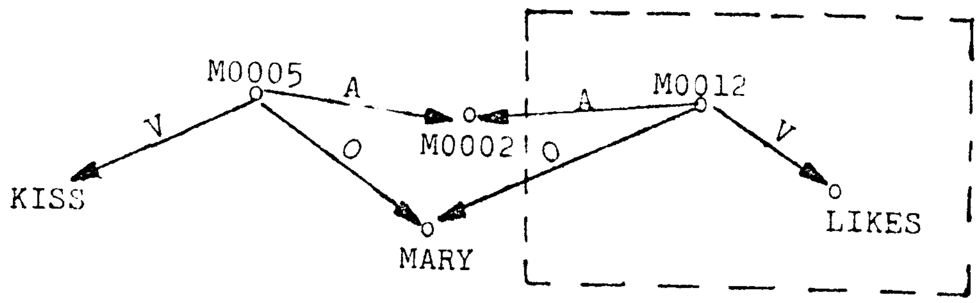
FINDs may be embedded within BUILDs to simulate descriptive phrases that refer to previously stored concepts. For example, a representation of "The person who kissed Mary sees John" might be:

```
**((DESCRIBE(BUILD A (FIND A* (FIND V KISS O MARY))
*                    V SEE O JOHN)))
(M0011 (O (JOHN))(V (SEE))(A (M002)))
```

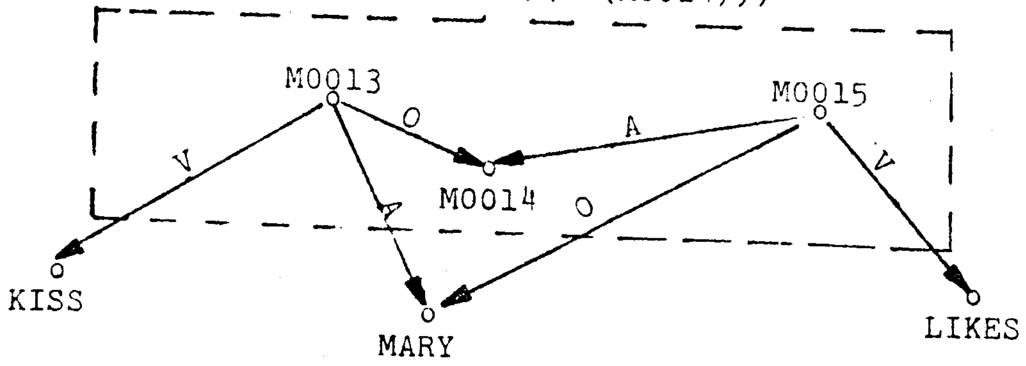


Occasionally, we desire to use a perviously built node, but are not really sure it exists. We want to FIND it if it does exist, but BUILD it if it doesn't. The FINDORBUILD function serves this purpose. For example the first use of FINDORBUILD below FINDS a node, whereas the second use BUILDs one.

```
**((DESCRIBE (BUILD A (FINDORBUILD A* (FINDORBUILD V KISS O MARY
*                       V LIKES O MARY)))
(M0012 (O (MARY))(V (LIKES)) (A (M0002)))
```



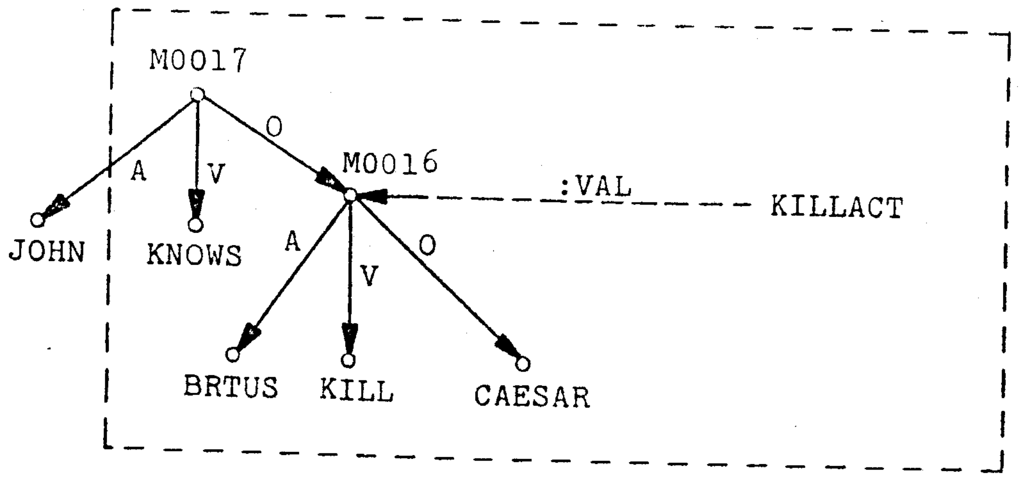
```
**((DESCRIBE (BUILD A (FINDORBUILD O* (FINDORBUILD A MARY V KISS
*                       V LIKES O MARY)))
(M0015 (O (MARY))(V (LIKES))(A (M0014)))
```



Variables may be assigned a value by use of an infix assignment operator. This simulates the use of a pronoun to refer to a previously described concept.

```
**((BUILD A BRUTUS V KILL O CAESAR) = KILLACT)
(M0016)

**((BUILD A JOHN V KNOWS O *KILLACT))
(M0017)
```



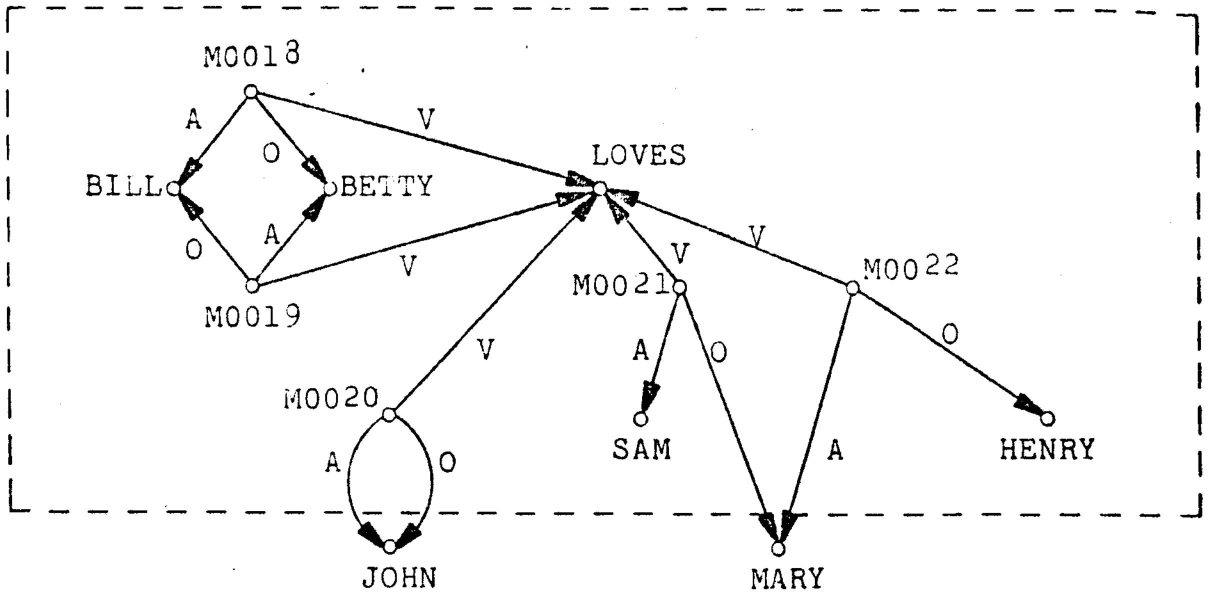
Another infix operator is relative complement, for which the symbol "-" is used.

```
**((BRUTUS CAESAR MARY) - (JOHN MARY))
(BRUTUS CAESAR)
```

We will further demonstrate the used of relative complement and the ? unquote after building some more structure.

```
**((BUILD A BILL V LOVES O BETTY))
(M0018)
**((BUILD A BETTY V LOVES O BILL))
(M0019)
**((BUILD A JOHN V LOVES O JOHN))
(M0020)
**((BUILD A SAM V LOVES O MARY))
(M0021)
**((BUILD A MARY V LOVES O HENRY))
(M0022)
```

The resultant structure is:



To find lovers who are loved, we can do:

```
**((FIND A* (FIND V LOVES) = L O* *L))
(MARY JOHN BILL BETTY)
```

To find lovers who are not loved, we use relative complement.

```
**((FIND A* *L) - (FIND O* *L))
(SAM)
```

To find those who love themselves, we use the ? unquote. Notice that if we consider the FIND instruction to be a pattern, the located nodes represent instantiations of that pattern such that the ? variable has a valid substitution in that instantiation. The nodes that can substitute for the variable go into the set that becomes the variable's value.

```
**((FIND A ?NARCISSIST V LOVES O ?NARCISSIST))
(M0020)

**(*NARCISSIST)
(JOHN)
```

The ? variable operates properly across embedded FINDS such as we could use to find lovers whose love is returned by the beloved.

```
**((FIND A* (FIND V LOVES O ?BELOVED)
*        O* (FIND V LOVES A ?BELOVED)))
(JOHN BILL BETTY)

**(*BELOVED)
(JOHN BETTY BILL)
```

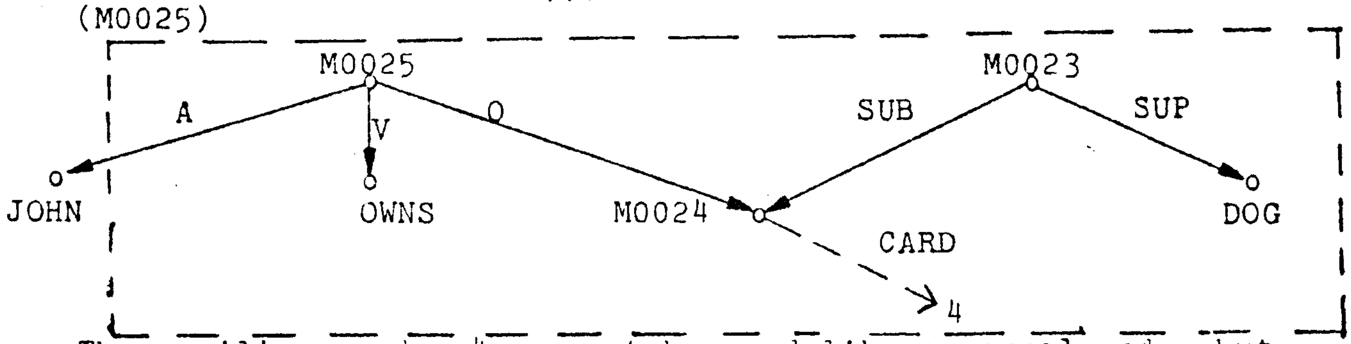
Using the variables assigned above, we can find unrequited lovers.

```
**((FIND A* *L) - *BELOVED)
(MARY SAM)
```

Additional auxiliary edges may be defined by the user to provide "hangings" on nodes. For example, one might want to use an auxiliary edge pointing to an integer to represent the cardinality of a set. Auxiliary edges may be defined with the DEFINE-AUX function.

```
**((DEFINE-AUX CARD))
CARD
(DEFINED AS AUXILIARY RELATIONS)

**((DEFINE SUB SUB* SUP SUP*))
(SUB SUB*)
(SUP SUP*)
(DEFINED)
**(BUILD A JOHN V OWNS
         O (BUILD SUB* (BUILD SUP DOG)
                  CARD 4)))
(M0025)
```

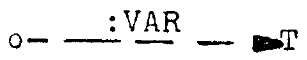


The auxiliary node, 4, cannot be used like a normal node, but it can be retrieved with the use of a ? variable.

```
**((FIND O* (FIND A JOHN V OWNS)
        SUB* (FIND SUP DOG)
        CARD ?X))
(M0024)
**(*X)
(4)
```

Storing and Using Patterns

When a $ unquoted variable is encountered, a variable node is created and made the value of the variable. The variable is also added to the value of the SNePS variable VARBL. This is the only way a variable node can be created. A variable node has the auxiliary relation :VAR to the auxiliary node T. We will diagram this as:

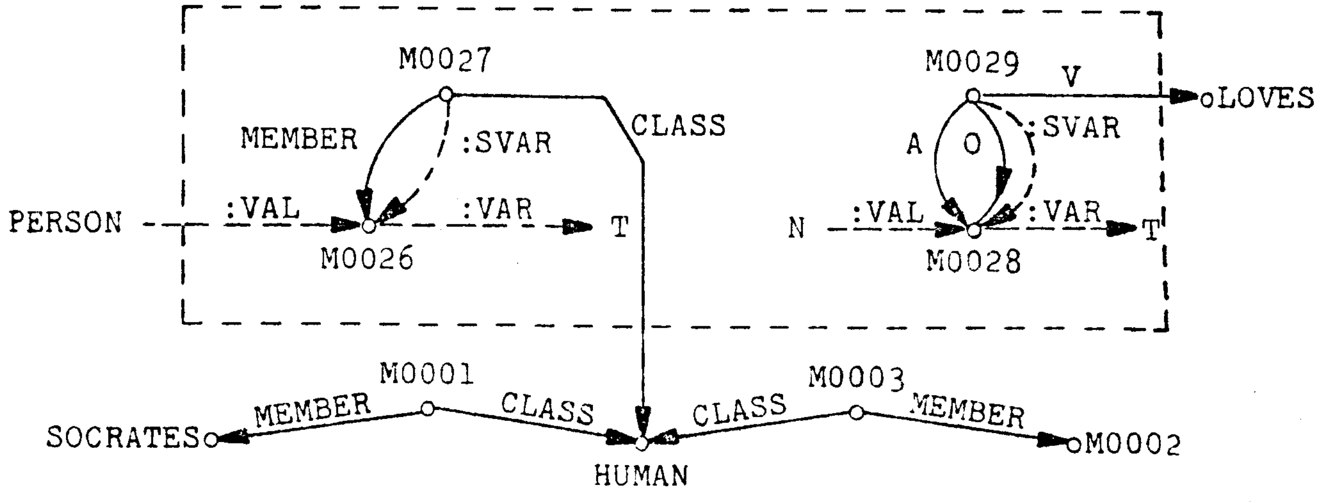


We will allow multiple representations of the auxiliary node T in the diagrams since there is no way to get from T to any other node.

A node which dominates a variable node is called a pattern node. and has the auxiliary relation :SVAR to each variable node dominated by it. The instructions

```
**((BUILD MEMBER $PERSON CLASS HUMAN))
(M0027)
**((BUILD A $N V LOVES O *N))
(M0029)
```

build the structure:



Notice that the variable PERSON has been assigned a new value.

The pattern M0027 is a stored version of the function (FIND MEMBER ?M0026 CLASS HUMAN) and the pattern M0029 is a stored version of (FIND A ?M0028 V LOVES O ?M0028). These pattern nodes may be used by use of the function NFIND.

```
**((NFIND M0027))
(M0027 M0003 M0001 M0026 M0028)
**((FIND A* (NFIND M0029)))
(JOHN M0028)
```

Since NFIND finds generalizations as well as instances, all variable nodes are included in the answer.

As indicated above, variable nodes are to pattern nodes what ? variables are to the FIND function. They are assigned values in the same way.

```
**(*M0026)
(M0026 M0002 SOCRATES)
**(*M0028)
(JOHN M0028)
```

To eliminate variable and pattern nodes from the value of NFIND, the "/" infix operator is useful. The left-hand operand of this operator is a set of nodes and the right-hand operand is a set of edge labels. The result is that subset of the given set of nodes containing nodes that do not have any of the given edges emanating from them. For example:

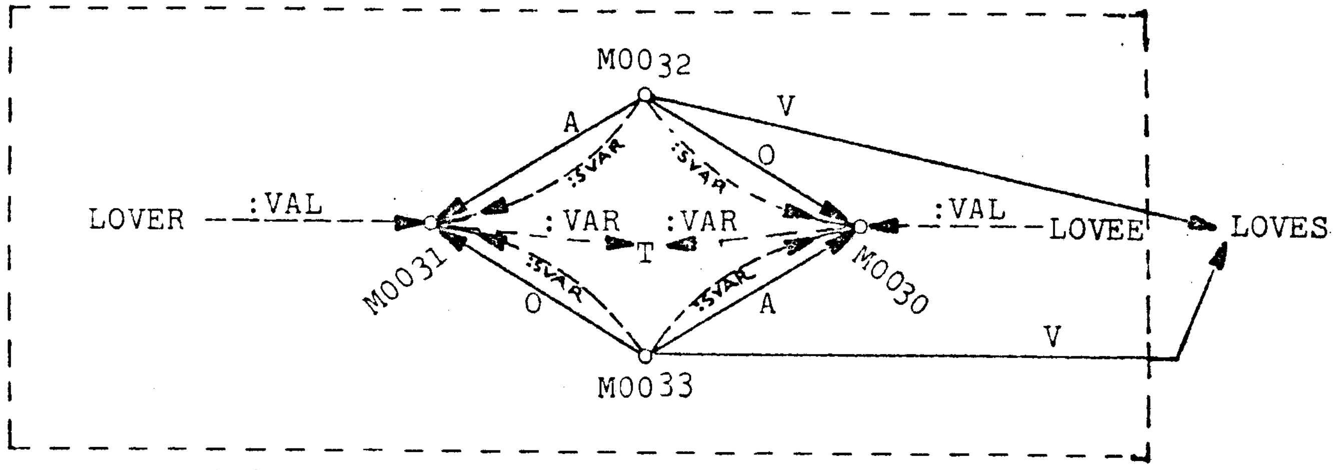
```
**((FIND A* *L) / (O*))
(SAM)
```

For use with NFIND, we would do the following:

```
**((:VAR :SVAR) = VARIABLES)
(:VAR :SVAR)
**((NFIND M0027) \ (*VARIABLES))
(M0003 M0001)
**(*M0026)
(M0002 SOCRATES)
**((FIND A* (NFIND M0029) \ (*VARIABLES)))
(JOHN)
**(*M0028)
(JOHN)
```

If NFIND is given a set of pattern nodes, it finds all nodes that match any pattern of the set.

```
**((BUILD A $LOVER V LOVES O $LOVEE))
(M0032)
**((BUILD A *LOVEE V LOVES O *LOVER))
(M0033)
```



```
**((NFIND M0032 M0033) \ (*VARIABLES))
(M0018 M0019 M0020 M0021 M0022)
**(*M0030)
(SAM BETTY BILL JOHN MARY HENRY)
**(*M0031)
(HENRY BILL BETTY JOHN SAM MARY)
```

The reason for the above result is that both M0032 and M0033 taken separately match any node with V to LOVES. NFIND returns the union of the two sets and the variable nodes, M0030 and M0031, are assigned the union of what they are assigned under each pattern.

NFIND finds generalizations of any molecular node as well as instances of pattern nodes.

```
**((DESCRIBE (NFIND M0018)))
(M0018 (O BETTY))(V (LOVES))(A (BILL)))
(M0033 (A (M0031))(O (M0030))(:SVAR (M0031 M0030))(V (LOVES)))
(M0032 (A (M0030))(O (M0031))(:SVAR (M0030 M0031))(V (LOVES)))
(M0031 (:VAL (HENRY BILL BETTY JOHN SAM MARY))
       (A* (M0033))(O* (M0032))(:VAR (T)))
(M0030 (:VAL (SAM BETTY BILL JOHN MARY HENRY))
       (O* (M0033))(A* (M0032))(:VAR (T)))
(M0028 (:VAL (JOHN))(A* (M0029))(O* (M0029))(:VAR (T)))
(M0026 (:VAL (M0002 SOCRATES))(MEMBER* (M0027))
       (:VAR (T)))
(DUMPED)

**(DESCRIBE (NFIND M0020)))
(M0020 (O (JOHN))(V (LOVES))(A (JOHN)))
(M0033 (A (M0031))(O (M0030))(:SVAR (M0031 M0030))(V (LOVES)))
(M0032 (A (M0030))(O (M0031))(:SVAR (M0030 M0031))(V (LOVES)))
(M0029 (A (M0028))(O (M0028))(:SVAR (M0028))(V (LOVES)))
(M0031 (:VAL (HENRY BILL BETTY JOHN SAM MARY))
       (A* (M0033))(O* (M0032))(:VAR (T)))
(M0030 (:VAL (SAM BETTY BILL JOHN MARY HENRY))
       (O* (M0033))(A* (M0032))(:VAR (T)))
(M0028 (:VAL (JOHN))(A* (M0029))(O* (M0029))(:VAR (T)))
(M0026 (:VAL (M0002 SOCRATES))(MEMBER* (M0027))
       (:VAR (T)))
```

Note that M0029 is not a valid generalization of M0018, although it is of M0020.

Temporary Nodes

It is occasionally desirable to build a node specifically for NFIND to find generalizations and instances of it. In that case, we would not want the node itself to be part of the answer. Also, if the node is a pattern node, we would not want its variables to be included in the values of all future calls of NFIND. The functic TBUILD and the unquote symbol % are provided for this purpose.

The % unquote is like the $ unquote except that the variable it causes to be created is not added into the value of VARBL so that it will not be found by NFIND.

```
**(%X)
(M0034)
**(*X)
(M0034)
**(*VARBL)
(M0031 M0030 M0028 M0026)
```

TBUILD is like BUILD except that the node it builds is not added into the value of the SNePS variable NODES and no converse edges are added to the network pointing to that node.

```
**((DESCRIBE (TBUILD MEMBER %X CLASS HUMAN) = P))
(M0036 (MEMBER (M0035))(:SVAR (M0035))(CLASS (HUMAN)))
**((FIND CLASS HUMAN))
(M0027 M0003 M0001)
**((NFIND *P))
(M0027 M0003 M0001 M0026 M0028 M0030 M0031)
```

Miscellaneous Functions

There are three functions for removing information from the data base:

(ERASE $node_1$ ... $node_k$)

removes each node from the graph along with any other nodes that thereby become isolated.

(REMVAR $variable_1$ ... $variable_k$)

unassigns each of the listed SNePS variables.

(DELREL $label_1$ ... $label_k$)

undefines each of the labels and their converses as valid edge labels. If any edges with these labels are in the graph, they are not removed. This function is most useful immediately after a typo has been discovered in a call to DEFINE or DEFINE-AUX.

Two functions are useful for saving networks across runs:

(OUTSYS (file.ext))

dumps the current contents of the network onto the file file.ext in a special format.

```
(INSYS (file.ext))
```

initializes the network with the contents of file.ext, which must have been created by OUTSYS. If nodes have already been built and INSYS is called, problems will result.

There are four SNePS variables that are maintained by the system:

(i) The value of NODES is the set of all nodes in the graph.

(ii) The value of VARBL is the set of all variable nodes in the graph.

(iii) The value of RELST is the set of all defined descending relations.

(iv) The value of AUXRELST is the set of all auxiliary edges.

Acknowledgements

I am grateful to Nicholas Vitulli, Nicholas Eastridge, and James McKew who contributed to this implementation of SNePS. Computer service was provided by the IUPUI Computing Facilities. Graphics were done by Christopher Charles and Cathy Poole.

References

Bobrow, D.G., and Raphael, B. 1974. "New Programming Languages for Artificial Intelligence Research." Computing Surveys 6, 3: 153-174.

Brown, J.S.; Burton, R.R.; Bell, A.G. 1974. SOPHIE: A Sophisticated Instructional Environment for Teaching Electronic Troubleshooting (An Example of AI in CAI), AI Report No. 12, Bolt Beranek and Newman Inc., Cambridge, Massachusetts.

Fillmore, C.J. 1968. "The Case for Case." In Bach and Harms, Eds. Universals in Linguistic Theory. Chicago: Holt, Rinehart, and Winston, Inc.

Friedman, D.P. 1973. GROPE: A Graph Processing Language and its Formal Definition, Ph.D. Dissertation and Technical Report No. 20, Department of Computer Science, The University of Texas at Austin.

Shapiro, S.C. 1971a. The MIND System: A Data Structure for Semantic Information Processing, R-837-PR, The Rand Corp., Santa Monica, California.

------------. 1971b. "A Net Structure for Semantic Information Storage, Deduction and Retrieval." 2d International Joint Conference on Artificial Intelligence: Advance Papers of the Conference, British Computer Society, London, 512-523.

Appendix: Summary of SNePS Constructs

Unquote Macro Symbols

*	Previously assigned SNePS value
#	Creates a new constant node
$	Creates a new variable node
%	Creates a new temporary variable node
?	Assigns a variable according to a search

Function

↑	LISP value of argument
DEFINE	Defines edge labels
DEFINE-AUX	Defines auxiliary edge labels
BUILD	Builds a node
TBUILD	Builds a temporary node
FIND	Locates a node(s)
FINDORBUILD	Locates a node(s), but if none exists, builds one
NFIND	Locates nodes according to a molecular node
DESCRIBE	Prints a dump-type description of nodes
ERASE	Removes nodes
REMVAR	Unassigns SNePS variables
DELREL	Undefines edge labels
OUTSYS	Dumps the network onto a file
INSYS	Loads the network from a file

Infix Operators

=	Variable assignment
-	Relative complement
\	Edge label domain restriction

Reserved SNePS Variables

NODES	The set of SNePS nodes
VARBL	The set of variable nodes
RELST	The set of descending edge labels
AUXRELST	The set of auxiliary edge labels

Auxiliary Edge Labels Pre-defined by SNePS

CONV	The converse of an edge label
:VAL	The value of a SNePS variable
:VAR	Indicator of variable nodes
:SVAR	Points from a pattern node to its variable nodes