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

JUNE, 1975

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.

An Introduction to SNePS

Stuart C. Shapiro
Computer Science Department
Indiana University
Bloomington, Indiana 47401
March, 1975

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 version of MENTAL [Shapiro 1971a, 1971b], is written in LISP 1.6 and runs on a DECsystem-10.

There are two basic types of nodes in the network, constant nodes and variable nodes. Constant nodes represent semantic concepts, including anything about which information may be stored in the network. Variable nodes are used to store graph patterns in the network. In addition, SNePS maintains some of its own information by using auxiliary nodes which may be connected to the network, but are not under the control of the user.

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 his own set of labels which are declared in pairs, one member of each pair being a descending relation, the other being the ascending converse of the first. Auxiliary nodes are connected to each other and/or to other nodes by relations controlled by SNePS which might not have converse relations stored. A node from which relations descend represents an assertion and may be considered an instance of some n-ary relation with labelled argument positions. An assertion node is also a constant node. A node from which it is possible to descend to one or more variable nodes (via a path of length 1 or more) represents a graph pattern.

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.

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. An unquoted atom is a SNePS variable. These are to be distinguished from both LISP variables and from variable nodes. An atom may have a LISP value and a different SNePS value, which will be a set of one or more nodes. A SNePS variable is maintained as an auxiliary node with the relation :VAL to each node in its value.

There are several types of unquotes:

*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.
$FOO	A newly created variable node, which is assigned as the new SNePS value of FOO.
?FOO	The SNePS value of FOO is determined during search as described below.
($ <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. All examples are for the purpose of describing SNePS and are not to be taken as this author's proposal for the actual contents of a model of human semantic memory.

The user declares 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 relation CONV to its converse label. Each descending relation is added into the set which is the SNePS value of the special 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
```

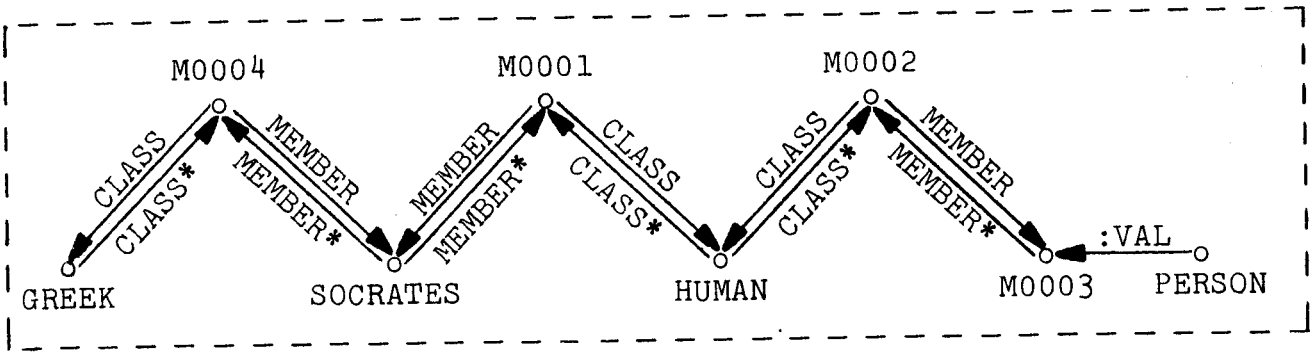
A node is created and its associated network built by the BUILD function. The value of the BUILD function is a list of the created node. The arguments to BUILD are, alternately, an edge label and a node or set of nodes. The second example below demonstrates one of the unquotes.

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

**((BUILD MEMBER #PERSON CLASS HUMAN))
(M0002)

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

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

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

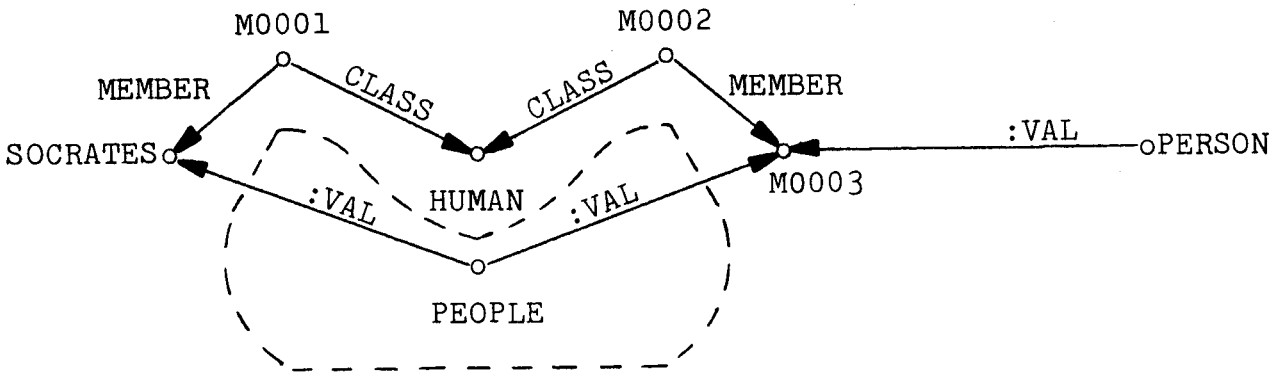
The value of FIND is a list of the located nodes, so calls to FIND may be embedded in other functions.

```
**((FIND MEMBER* (FIND CLASS HUMAN)))
(M0003 SOCRATES)

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

**((FIND MEMBER ?PEOPLE CLASS HUMAN))
(M0002 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.



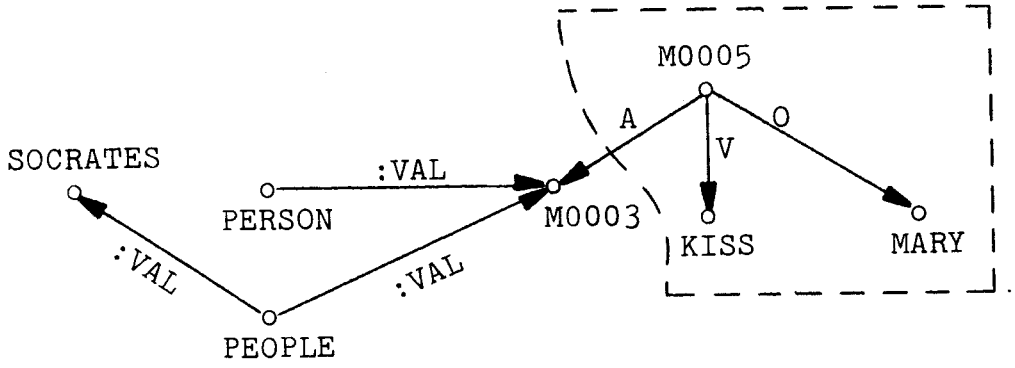
To simply print the value of a SNePS variable, the following use of the * unquote suffices.

```
**(*PERSON)
(M0003)

**(*PEOPLE)
(SOCRATES M0003)
```

Assigned variables may also be used within functions.

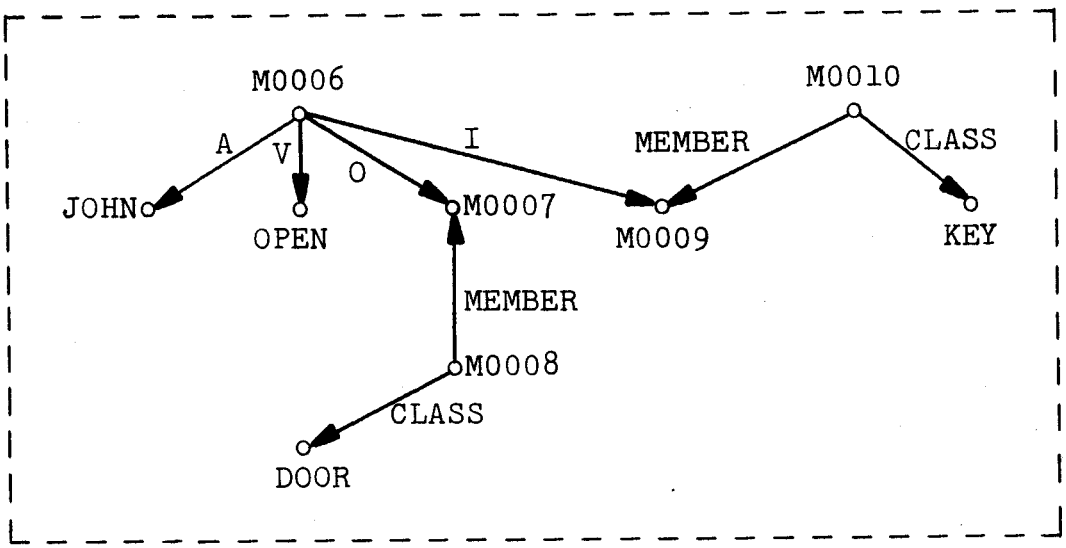
```
**((BUILD A *PERSON V KISS O MARY))
(M0005)
```




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

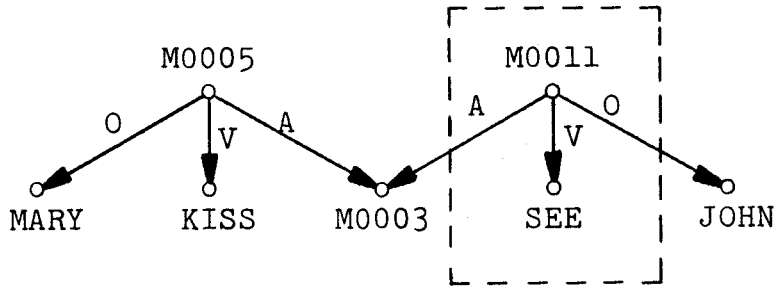
BUILDs may be embedded within BUILDs to simulate the several sentences that underlie a single surface sentence. For example, a simplified representation of "John opens a door with a key" might be:

```
**((BUILD A JOHN V OPEN
*         O (BUILD MEMBER* (BUILD CLASS DOOR))
*         I (BUILD MEMBER* (BUILD CLASS KEY))))
(M0006)
```



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:

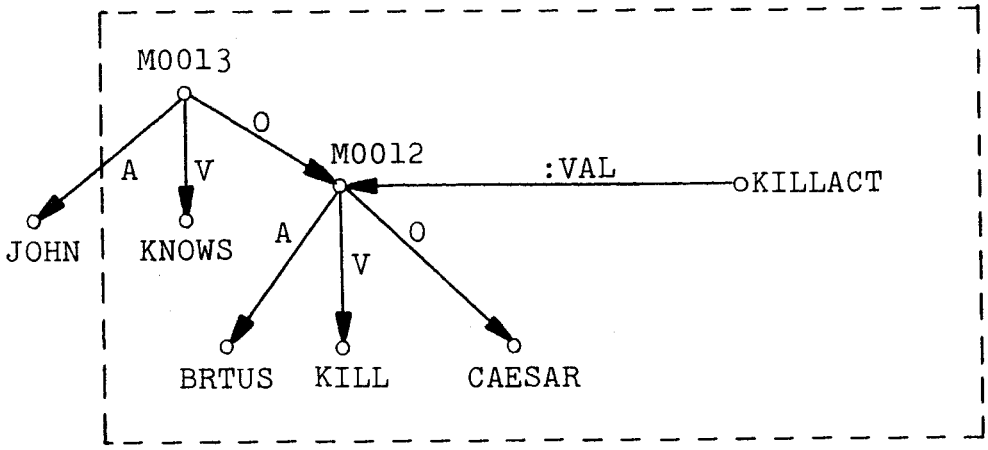
```
**((BUILD A (FIND A* (FIND V KISS O MARY))
*           V SEE O JOHN))
(M0011)
```



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)
(M0012)
```

```
**((BUILD A JOHN V KNOWS O *KILLACT))
(M0013)
```



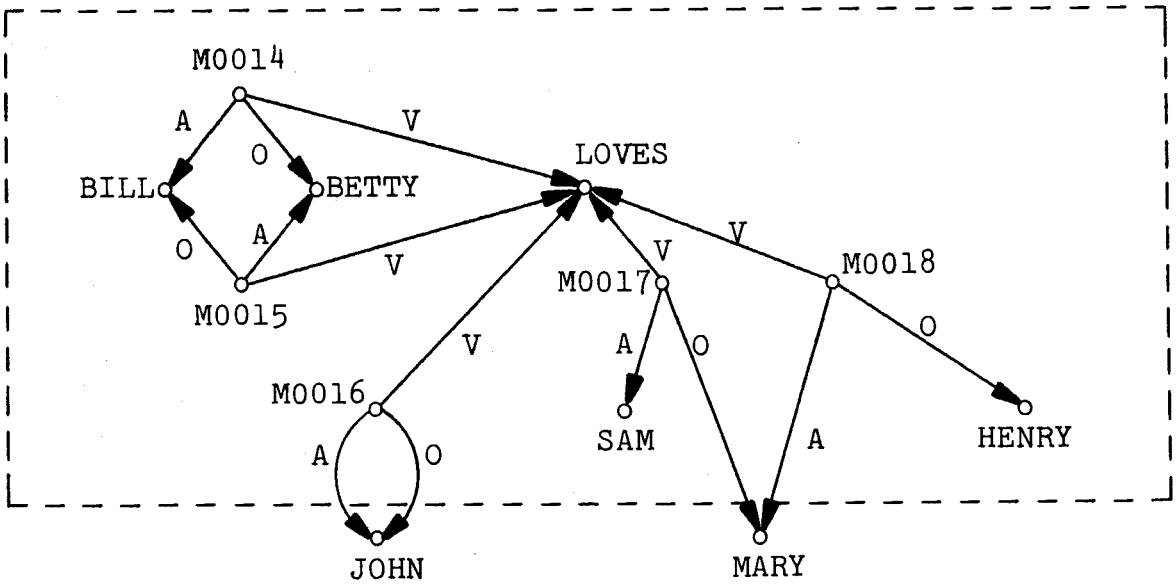
Another infix operator is relative complement, for which the symbol "-" is used.

```
**((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))
(M0014)
**((BUILD A BETTY V LOVES O BILL))
(M0015)
**((BUILD A JOHN V LOVES O JOHN))
(M0016)
**((BUILD A SAM V LOVES O MARY))
(M0017)
**((BUILD A MARY V LOVES O HENRY))
(M0018)
```

The resultant structure is:



To find lovers who are loved, we can do:

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

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))
(M0016)

**(*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 BETTY BILL)

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

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

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

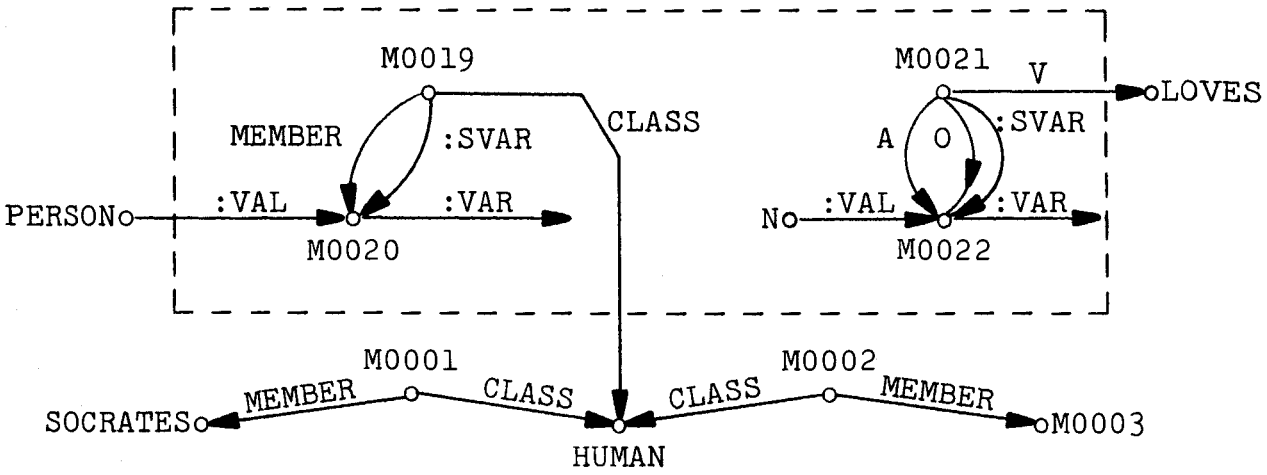
Storing and Using Patterns

When a $ unquoted variable is encountered, a variable node is created and made the value of the variable. This is the only way a variable node can be created. A variable node has the special indicator :VAR, which we will diagram as

A node from which it is possible to follow a path of descending relations to a variable node is called a pattern node and has the special relation :SVAR to each variable node reachable from it. The instructions

```
**((BUILD MEMBER $PERSON CLASS HUMAN))
(M0019)
**((BUILD A $N V LOVES O *N))
(M0021)
```

build the structure:



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

The pattern M0019 is a stored version of the function (FIND MEMBER ?M0020 CLASS HUMAN) and the pattern M0021 is a stored version of (FIND A ?M0022 V LOVES O ?M0022). These pattern nodes may be used by use of the function NFIND.

```
**((NFIND M0019))
(M0019 M0002 M0001)
**((FIND A* (NFIND M0021)))
(JOHN M0022)
```

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

```
**(*M0020)
(SOCRATES M0003 M0020)
**(*M0022)
(M0022 JOHN)
```

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:

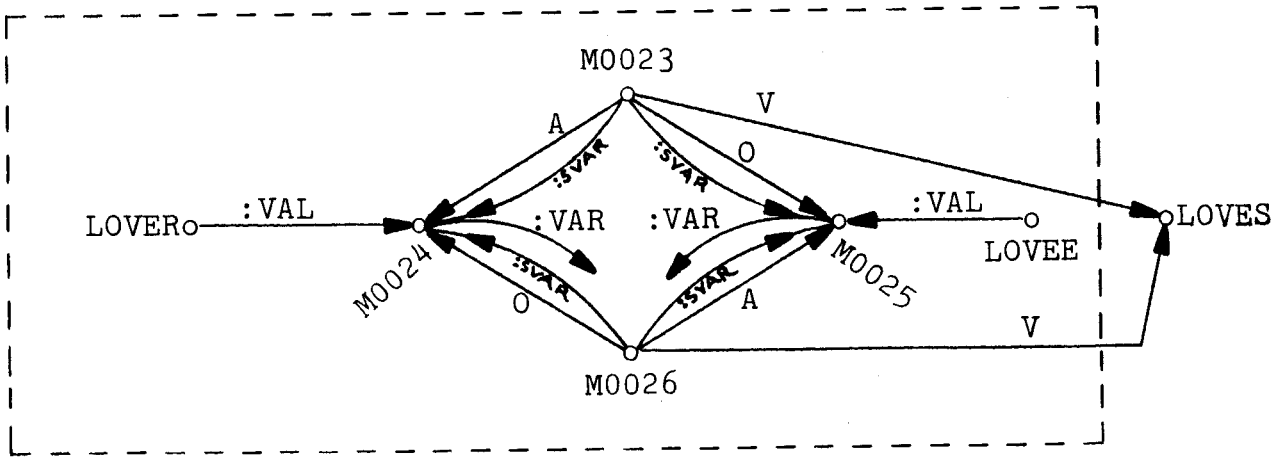
```
**((FIND A* *L) \ (O*))
(SAM)
```

For use with NFIND, we would do the following:

```
**((:VAR :SVAR) = VARIABLES)
(:VAR :SVAR)
**((NFIND M0019) \ (*VARIABLES))
(M0002 M0001)
**(*M0020)
(SOCRATES M0003)
**((FIND A* (NFIND M0021) \ (*VARIABLES)))
(JOHN)
**(*M0022)
(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))
(M0023)
**((BUILD A *LOVEE V LOVES O *LOVER))
(M0026)
```



```
**((NFIND (M0023 M0026)) \ (*VARIABLES))
(M0018 M0017 M0016 M0015 M0014)
**(*M0024)
(HENRY BILL BETTY JOHN SAM MARY)
**(*M0025)
(SAM BETTY BILL JOHN MARY HENRY)
```

The reason for the above result is that both M0023 and M0026 taken separately match any node with V to LOVES. NFIND returns the union of the two sets and the variable nodes, M0024 and M0025, are assigned the union of what they are assigned under each pattern.

The function CNFIND is similar to NFIND except that when it is given a set of patterns, it considers them conjunctively. Common variable nodes serve the same restrictive function as common ? variables serve across embedded FINDs. Compare the above example with the following one:

```
**((CNFIND (M0023 M0026)) \ (*VARIABLES))
(M0016 M0014 M0015)
**(*M0024)
(BILL BETTY JOHN)
**(*M0025)
(BETTY BILL JOHN)
```

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.

There are three 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.

Acknowledgements

I am extremely grateful to Nicholas Vitulli who programmed this implementation of SNePS. I am also grateful to Christopher Charles for his excellent typing and graphics.

References

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

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

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

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

Typing & Graphics by Christopher Charles

Appendix: Summary of SNePS Constructs

Unquote Macro Symbols

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

Function

$	LISP value of argument
DEFINE	Defines edge labels
BUILD	Builds a node
FIND	Locates a node(s)
NFIND	Locates nodes according to a pattern node
CNFIND	Conjunctive version of NFIND
DESCRIBE	Prints a dump-type description of nodes
ERASE	Removes nodes
REMVAR	Unassigns SNePS variables
DELREL	Undefines edge labels

Infix Operators

=	Variable assignment
-	Relative complement
\	Edge label domain restriction

Reserved SNePS Variables

NODES	The set of nodes in the graph
VARBL	The set of variable nodes
RELST	The set of defined descending relations

Edge Labels Used by SNePS (do not have converses)

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