SNePS 2.7 USER'S MANUAL[1]

Stuart C. Shapiro
and
The SNePS Implementation Group

Department of Computer Science and Engineering
University at Buffalo, The State University of New York
201 Bell Hall
Buffalo, NY 14260-2000

February 11, 2008

[1]The development of SNePS was supported in part by: the National Science Foundation under Grants IRI-8610517 and REC-0106338; the Defense Advanced Research Projects Agency under Contract F30602-87-C-0136 (monitored by the Rome Air Development Center) to the Calspan-UB Research Center; the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB DC 20332 under Contract No. F30602-85-C-0008, which supported the Northeast Artificial Intelligence Consortium (NAIC); NASA under contract NAS 9-19335 to Amherst Systems, Inc.; ONR under contract N00014-98-C-0062 to Apple Aid, Inc.; the U.S. Army CECOM Intelligence and Information Warfare Directorate (I2WD) through a contract with CACI Technologies and through Contract #DAAB-07-01-D-G001 with Booze·Allen & Hamilton; and CUBRC under prime contract FA8750-06-C-0184 between CUBRC and U.S. Air Force Research Laboratory, Rome, NY.

Over the years, many people have contributed to the design and implementation of SNePS, and to the writing of successive versions of the SNePS User's Manual. They constitute "The SNePS Implementation Group" cited on the title page, and I am grateful to them. They are listed here. If I have inadvertently omitted anyone's name, or have mispelled anyone's name, please let me know, and I will correct it for the next printing of this Manual.

Syed S. Ali
Michael J. Almeida
Charles W. Arnold
Robert J. Bechtel
Sudhaka Bharadwaj
Jong S. Byoun
Alistair E. Campbell
Scott S. Campbell
Hans Chalupsky
Chung M. Chan
Joongmin Choi
Chi C. Choy
Soon Ae Chun
Maria R. Cravo
Dmitriy Dligach
Zuzana Dobes
Gerard F. Donlon
Nicholas E. Eastridge
Elissa Feit
David Forster
Richard B. Fritzson
James Geller
Susan M. Haller
Richard G. Hull
Haythem Ismail
Frances L Johnson
Steven D. Johnson
Darrel L. Joy
Sudha Kailar
Michael W. Kandefer
Deepak Kumar
Stanley C. Kwasny
John S. Lewocz
Naicong Li
John D. Lowrance
Christopher Lusardi
Anthony S. Maida
Mark D. Malamut
Nuno Mamede
João P. Martins
Pedro A. Matos
Donald P. McKay
James P. McKew
Ernesto J. Morgado
William A. Neagle
Jeannette G. Neal
Jane Terry Nutter
Rafail Ostrovsky
Sandra L. Peters
Carlos Pinto-Ferreira
William J. Rapaport
Victor H. Saks
Harold L. Shubin
Reid G. Simmons
Benjamin R. Spigle, Jr.
Rohini K. Srihari
William M. Stanton
Jennifer M. Suchin
Lynn M. Tranchell
Jason C. Van Blargan
Nicholas F. Vitulli
Diana K. Webster
Janyce M. Wiebe
Albert Hanyong Yuhan
Martin J. Zaidel

Stuart C. Shapiro

Contents

List of Figures **vii**

1 Introduction **1**
- 1.1 General . . . 1
- 1.2 What's New . . . 2
- 1.3 System Portability . . . 3
- 1.4 Commands and Environments . . . 4
- 1.5 Types of Nodes . . . 5
- 1.6 Contexts . . . 5
- 1.7 SNePSUL Variables . . . 6

2 SNePSUL Commands **7**
- 2.1 Context Specifiers . . . 7
- 2.2 Loading SNePS . . . 7
- 2.3 Entering and Leaving SNePS . . . 7
- 2.4 Using Auxiliary Files . . . 8
  - 2.4.1 Reading/Writing Files . . . 8
  - 2.4.2 Writing/Altering Source Files For Use With SNePS 2.6 and ACL 6 . . . 9
- 2.5 Relations . . . 10
  - 2.5.1 Reduction Inference . . . 10
  - 2.5.2 Path-Based Inference . . . 11
- 2.6 Operating on Contexts . . . 13
- 2.7 Building Networks . . . 13
- 2.8 Deleting Information . . . 14
- 2.9 Functions Returning Sets of Nodes or of Unitpaths . . . 15
- 2.10 Displaying the Network . . . 16
- 2.11 Retrieving Information . . . 17

3 SNIP: The SNePS Inference Package **19**
- 3.1 Representing and Using Rules . . . 19
  - 3.1.1 Connectives . . . 19
  - 3.1.2 Quantifiers . . . 21
  - 3.1.3 Recursion . . . 22
- 3.2 Tracing Inference . . . 23

4 SNeRE: The SNePS Rational Engine **25**
- 4.1 Acting . . . 25
- 4.2 Primitive Acts . . . 26
- 4.3 Associating Primitive Action Nodes with Their Functions . . . 33
- 4.4 Complex Acts . . . 34

4.5 Goals . . . 37
4.6 The Execution Cycle: Preconditions and Effects . . . 38

5 Program Interface 45
5.1 Transformers . . . 45
5.2 With-SNePSUL Reader Macro . . . 45
5.2.1 Controlling the Evaluation of SNePSUL Forms Generated by #! . . . 46
5.2.2 Example Use of #! . . . 47
5.3 Defining New Commands . . . 49

6 SNePSLOG 53
6.1 SNePSLOG Basics . . . 53
6.2 SNePSLOG Syntax . . . 53
6.3 SNePSLOG Semantics . . . 56
6.3.1 Semantics of SNePSLOG Commands . . . 56
6.3.2 Semantics of wffNameCommands . . . 59
6.3.3 Semantics of wffCommands . . . 60
6.3.4 Semantics of SNePSLOG Wffs . . . 61
6.4 SNIP in SNePSLOG . . . 62
6.4.1 Rules of Inference . . . 62
6.4.2 Recursion . . . 64
6.5 SNeRE in SNePSLOG . . . 65
6.5.1 SNePSLOG Versions of SNeRE Constructs . . . 65
6.5.2 Restrictions . . . 66
6.6 The Tell-Ask Interface . . . 67
6.7 The Java-SNePS API . . . 68

7 Procedural Attachment 69

8 SNeBR: The SNePS Belief Revision System 71
8.1 Hypotheses, Contexts, and Belief Spaces . . . 71
8.2 Responsibilities of SNeBR . . . 71
8.3 Recognizing a Contradiction . . . 71
8.4 Identifying Possible Culprits . . . 72
8.5 Choosing a Culprit . . . 72
8.5.1 Automatic Culprit Choosing . . . 72
8.5.2 Assisted Culprit Choosing . . . 72
8.6 Disbelief Propagation . . . 73
8.7 Warning About a Belief Space Known to be Contradictory. . . . 73

9 SNaLPS: The SNePS Natural Language Processing System 75
9.1 Top-Level SNaLPS Functions . . . 75
9.2 The Top-Level SNaLPS Loop . . . 76
9.2.1 Input to the SNaLPS Loop . . . 76
9.2.2 SNaLPS Variables . . . 76
9.3 Syntax and Semantics of GATN Grammars . . . 77
9.3.1 Arcs . . . 78
9.3.2 Actions . . . 79
9.3.3 Preactions . . . 80
9.3.4 Terminal Actions . . . 80
9.3.5 Forms . . . 80
9.3.6 Tests . . . 81

9.3.7 Terminal Symbols . . . 81
9.4 Morphological Analysis and Synthesis . . . 82
9.4.1 Syntax of Lexicon Files . . . 82
9.4.2 Functions for Morphological Analysis . . . 83
9.4.3 Functions for Morphological Synthesis . . . 84
9.5 Examples . . . 85
9.5.1 Producing Parse Trees . . . 85
9.5.2 Interacting with SNePS . . . 90

10 SNePS as a Database Management System 97
10.1 SNePS as a Relational Database . . . 97
10.1.1 Project . . . 97
10.1.2 Select . . . 100
10.1.3 Join . . . 100
10.2 SNePS as a Network Database . . . 101
10.3 Database Functions . . . 102

University at Buffalo Public License ("UBPL") Version 1.0 103
1. Definitions. . . . 103
2. Source Code License. . . . 104
2.1. The Initial Developer Grant. . . . 104
2.2. Contributor Grant. . . . 105
3. Distribution Obligations. . . . 105
3.1. Application of License. . . . 105
3.2. Availability of Source Code. . . . 105
3.3. Description of Modifications. . . . 105
3.4. Intellectual Property Matters . . . 106
3.5. Required Notices. . . . 106
3.6. Distribution of Executable Versions. . . . 106
3.7. Larger Works. . . . 107
4. Inability to Comply Due to Statute or Regulation. . . . 107
5. Application of this License. . . . 107
6. Versions of the License. . . . 107
6.1. New Versions . . . 107
6.2. Effect of New Versions . . . 107
6.3. Derivative Works . . . 107
6.4. Origin of License . . . 108
7. DISCLAIMER OF WARRANTY . . . 108
8. Termination . . . 108
9. LIMITATION OF LIABILITY . . . 109
10. U.S. government end users . . . 109
11. Miscellaneous . . . 109
12. Responsibility for claims . . . 110
13. Multiple-licensed code . . . 110
Exhibit A - University at Buffalo Public License. . . . 111

Index 112

List of Figures

4.1 Three ways of associating action nodes with action functions. `M3`, `M4`, and `M5` are act nodes, with action nodes `SAY`, `M1`, and `B1` respectively. 35

9.1 Graphical version of the example GATN. 86
9.2 SNePS network after running the example. 95

10.1 Fragment of SNePS network for the Supplier-Part-Project Database 98

Chapter 1

Introduction

1.1 General

SNePS is a logic-, frame- and network-based knowledge representation, reasoning and acting system. The name "SNePS" originally was an acronym of "The Semantic Network Processing System,".

A semantic network, roughly speaking, is a labeled directed graph in which nodes represent entities, arc labels represent binary relations, and an arc labeled R going from node n to node m represents the fact that the entity represented by n bears the relation represented by R to the entity represented by m.

SNePS is called a *propositional* semantic network because every proposition represented in the network is represented by a node, not by an arc. Relations represented by arcs may be thought of as part of the syntactic structure of the node they emanate from. Whenever information is added to the network, it is added in the form of a node with arcs emanating from it to other nodes.

Each entity represented in the network is represented by a unique node. This is enforced by SNePS 2 in that whenever the user specifies a node to be added to the network that would look exactly like one already there, in the sense of having the same set of arcs going from it to the same set of other nodes, SNePS 2 retrieves the old one instead of building the new one.

SNePSUL, the SNePS User Language, is the lowest-level command language for using SNePS. It is a Lispish language, usually entered by the user at the top-level SNePSUL read-eval-print loop, but it can also be called from Lisp code or from GATN arcs. The SNePSUL chapters of this manual follow the style of Guy Steele's COMMON LISP book, and assume that the reader is familiar with that book and with COMMON LISP. The organization of this manual has been retained from the time when SNePSUL was the standard way to interact with SNePS. However, the use of SNePSLOG is now recommended.

SNIP (Chapter 3), the SNePS Inference Package, interprets certain nodes as representing reasoning rules, called *deduction rules*. SNIP supports a variety of specially designed propositional connectives and quantifiers, and performs a kind of combined forward/backward inference called *bi-directional* inference.

SNePSLOG (Chapter 6) is a logic programming interface to SNePS, and provides direct access in a predicate logic notation to almost all the facilities provided by SNePSUL. It is now the recommended way to interact with SNePS.

SNeBR (Chapter 8), the SNePS Belief Revision system, recognizes when a contradiction exists in the network, identifies possible culprits, and performs disbelief propagation. It eliminates contradictions automatically in some cases, and helps the user to do so in the general case.

SNaLPS (Chapter 9), the SNePS Natural Language Processing System, consists of a morphological analyzer, a morphological synthesizer, and a Generalized Augmented Transition Network (GATN) Grammar interpreter/compiler. Using these facilities, one can write natural language (and other) interfaces for SNePS.

1.2 What's New

SNePS 2 differs in several respects from its predecessor, now called SNePS-79, mostly because of theoretical decisions that were made since SNePS-79 was implemented.

SNePS 2.1 differs from SNePS 2.0 by including belief revision as a standard feature.

SNePS 2.3 includes some techniques for making node-based inference faster, and includes SNeRE (Chapter 4).

SNePS 2.4 includes: a change in how contexts and sets of contexts are implemented that should improve the speed of the system; `deducetrue`, `deducefalse`, `deducewh`, and `deducewhnot` (Section 2.11); the Tell-Ask interface (Section 6.6); and SNePSLOG Mode 3, which allows SNePSLOG syntax to be used to build SNePS networks using as flexible a choice of relations as may be done using SNePSUL syntax (Section 6.3.1).

SNePS 2.5 includes: a change in how nodes are implemented and how sets of nodes are ordered that should improve the speed of the system; `activate` (Section 2.7); the SNePSLOG `perform` command (Section 6.3.1); a revised semantics for `when-do`, with the old `when-do` now renamed `whenever-do` (Section 4.1).

SNePS 2.6 code has been modified so that it can be loaded into ALLEGRO CL version 6.X and used with its default case mode of "case-sensitive-lower" (all predefined COMMON LISP symbols have lower-case names, and the case of characters typed into the LISP listener is left as originally typed) **or** loaded into earlier versions of ALLEGRO CL, or other versions of COMMON LISP that use "case-insensitive-upper" mode (all predefined COMMON LISP symbols have upper-case names, and the case of characters typed into the LISP listener is changed to upper case). Some SNePS symbols may look different in the two different modes, such as `M1` vs. `m1`. The output for the examples in this manual was generated from earlier versions of SNePS, so it is typically in uppercase (except where special formatting was used to generate the output).

SNePS 2.6.1 Previously, the functions `+` and `&` took only two arguments. Now they can take zero or more. The function `show` has been added. The SNeRE mental action `believe` now checks for and `disbelieves` more contradictory beliefs than before. The SNePSLOG commands `activate`, `activate!`, `ask`, `askifnot`, `askwh`, and `askwhnot` have been added. There have been several other bug fixes. Documentation of the SNePSLOG command `list-wffs` has been added to the manual; it was previously available. Section 6.5, "SNeRE in SNePSLOG," has been added to this manual, some material describing features that were never implemented in SNePS 2 has been deleted from the manual, and there have been other editorial changes. A complete description of what's new in SNePS 2.6.1 is at `http://www.cse.buffalo.edu/sneps/Downloads/releaseNotes261.html`.

SNePS 2.7 The SNePSLOG parser has been completely rewritten, improved, and made more robust. The chapter on SNePSLOG has been completely rewritten. SNePSLOG is now the preferred interface to SNePS. Normal output from SNePSLOG now shows the wffName. The SNePSLOG command `define-terms` has been added, and `define-frame` given an optional string argument, which, when used together, provide a facility for giving a natural language gloss of SNePS terms. The tell-ask interface has been improved, and its documentation has been moved to §6.6. Previously `askwh` and `askwhnot` would return a simple list of terms, even if the query had more than one free variable. Now, they return a list of substitutions even if the query has only one free variable. (Similarly for the SNePSUL commands `deducewh` and `deducewhnot`). A Java-SNePS API has been added to give Java programs access to SNePS via the tell-ask interface, see §6.7. The final cases in which the inference system could get into infinite recursion have been eliminated—see §3.1.3 and §6.4.2. The SNePSUL `intext` command and the new SNePSLOG `load` command silently load files. Previously, one could

not have a node named with a string the Lisp reader could not handle, such as `"#"`, and `node-to-lisp-object` did not handle nodes with names like `"5 pounds"`. Both these problems are now fixed. A node's name can now be any Lisp number, string, or symbol, and `node-to-lisp-object` is able to handle it. Previously, inference and acting tracing was on by default, now they are off by default. Previously, in `add-to-context`, only the wffs in termSet that had already been introduced to the KB as hypotheses were added as hypotheses of the context. Now, all the wffs listed as to be added are asserted into the context. An procedural attachment facility has been added. It is described in Chapter 7. The guarded acts in `snif` and `sniterate` can now take sets of conditions and sets of acts. `snif` and each loop of `sniterate` still performs one act one of whose guards holds. The mental acts `adopt` and `unadopt` are now available for adopting and unadopting policies, and should be used for those purposes instead of `believe` and `disbelieve`. A chapter on SNeBR (Chapter 8) has been added. XGinseng has been eliminated because it was implemented in Garnet, which has not been maintained, and was difficult to use. SNePS networks may now be displayed by `show`, either via: `dot`; or `JUNG` and `JIMI`, depending on the installers choice (both require separate downloading). If both versions are installed, the user may choose which one to use by setting the value of the global variable `cl-user:*use-gui-show*`.

There have been other bug fixes, improvements to speed, and improvements to user messages. A complete description of what's new in SNePS 2.7 is at `http://www.cse.buffalo.edu/sneps/Downloads/releaseNotes262.html`.

1.3 System Portability

SNePS 2 is written in ANSI COMMON LISP (two exceptions are noted in the following paragraph). Hence, every proper implementation of ANSI COMMON LISP should be sufficient to run SNePS 2. In particular, SNePS 2 should run successfully using the following:

- UNIX operating system
- LINUX operating system
- Apple/Macintosh operating system
- Microsoft Windows operating system
- Allegro Common Lisp (Franz Inc.)
- Lucid (or Sun) Common Lisp
- GNU CLISP
- CMU Common Lisp
- Macintosh Common Lisp
- Harlequin LispWorks

The `JUNG/JIMI` version of `show`, and the Java-SNePS API require some facilities peculiar to Franz's ACL. For installations that do not have that compiler, an executable file is available for downloading from `http://www.cse.buffalo.edu/sneps/Downloads/`. That executable provides all the features of SNePS 2.7. Every other feature of SNePS 2.7 is written in ANSI Common Lisp.

1.4 Commands and Environments

A SNePSUL *command* is classified according to its role either as a *procedure* or as a *function*. A *procedure* is a command that performs some action but returns nothing, using the COMMON LISP `(values)` function. A *function* is a command that always returns some value, possibly after having performed some action as a side effect. A function is implemented directly as a Lisp function.

A command is also classified according to the environment(s) in which it may legally appear. A procedure can be entered only at the top level of SNePSUL. A function, however, may appear in many different environments. The five environments are:

1. The top level of SNePS 2
2. A *relation-set* position embedded in a command
3. A *node-set* position in `build`
4. A *node-set* position in `find` or `findassert`
5. A *node-set* position in any of the other commands

Finally, a command can be classified according to the relation between its position and the position of its arguments in the input line.

Most commands have an arbitrary number of arguments. They are called *prefix commands*, because they can only be entered using Cambridge prefix notation:

(*prefix-command argument* ... *argument*).

Some two-argument commands can be entered in infix position, and so are called *infix commands*. When an infix command is used in infix position, SNePS rearranges the input line to transform the form into a prefix form. Precedence is always from left to right. An infix command can be used as

(*infix-command argument argument*)

or as

argument infix-command argument

with no parentheses.

Since SNePS always remembers the result of the last top-level function, an infix command can also be used as

infix-command argument

in which case SNePS recalls the result of the last function and makes it the first argument for the infix command before rearranging the form to the prefix notation.

Similarly, some one-argument commands can be entered in postfix position and therefore are called *postfix commands*. A postfix command can be used as

(*postfix-command argument*)

or as

argument postfix-command

with no parentheses, or just as

postfix-command

in which case the result of the last function is used as argument.

Another kind of one-argument command, called *macro commands*, have one-character names and are used as

macro-command argument

with no parentheses, and preferably with no space between the command and the argument. Before passing it to the evaluator, the SNePS reader expands this form to a standard Cambridge prefix form.

1.5 Types of Nodes

There are four types of nodes in the SNePS network: base, variable, molecular, and pattern.

Base nodes are distinguished by having no arcs emanating from them. A base node may be created by the user's referring to it by name in the proper context. In such a case, the name of a base node can be any Lisp symbol. If a number is used, the node's name is a symbol whose symbol-name is a string of the characters that makes up the number. If a string is used, the node's name is the symbol whose symbol-name is that string. A base node may also be created using the `#` macro command, in which case the node's name is `B`x, where x is some integer. A base node is assumed to represent some entity—individual, object, class, property, etc. It is assumed that no two base nodes represent the same, identical entity. One may, of course, introduce an equality or equivalence relation and the rules for using them. In that case the introduced equality or equivalence relation is weaker than the identity relation just referred to. This is the most basic way that SNePS assumes an intensional representation—no two nodes are intensionally identical even though they might be extensionally equivalent.

Variable nodes also have no arcs emanating from them, but represent arbitrary individuals or propositions, in much the same way that logical variables do. Variable nodes are created using the `$` macro command. The name of a variable node is `V`x, where x is some number.

Molecular nodes and pattern nodes have arcs emanating from them. Molecular nodes may represent propositions, including rules, or "structured individuals." A molecular node that represents a proposition may be *asserted* or *unasserted*. Pattern nodes represent arbitrary propositions or arbitrary structured individuals, and are similar to open sentences in predicate logic. Pattern nodes and unasserted molecular nodes are created by the `build` function. Asserted molecular nodes are created by the `assert` function. An unasserted molecular node may be asserted by using the `!` postfix command. The name of a pattern node is `P`x, where x is a number. The name of a molecular node is `M`x, where x is a number. The name of an asserted molecular node is printed with a suffix of `!`.

Once any node is created, it may be referred to by its name. It is not necessary to include the `!` suffix to refer to an asserted molecular node. In fact, its use is always interpreted as a call to the `!` command, which will assert the node even if it wasn't previously asserted.

1.6 Contexts

A *context* is a structure with three components: 1) a set of hypotheses; 2) a restriction set; 3) a set of names. The set of hypotheses is a set of nodes which are the assumptions of the context. The set of hypotheses is the determining component of the context in the sense that no two contexts will have the same set of hypotheses. The restriction set is a set of sets of nodes, such that the union of any of these sets with the set of hypotheses of the context forms a set of hypotheses from which a contradiction has been derived (*i.e.* a set of hypotheses known to be inconsistent). The set of names is a set of symbols each of which functions as a name of this context.

A context name intensionally defines a context, which is extensionally defined by its set of hypotheses. The SNePSUL user always refers to contexts by name, and may add assertions to, or remove assertions from a context. Actually, such changes do not change contexts (extensionally defined), but change the context that

the name refers to. The system takes care of such details, and the SNePSUL user may normally think of a context name as always referring to the same context.

The user is always working in a particular context, called the *current context*. The current context for a particular SNePSUL command may be specified by an optional argument to the command. Otherwise, all commands are carried out with the *default context* as current context. By default, this context is named `default-defaultct`.

In SNePS 2.3 and later versions, a proposition node is not simply asserted or unasserted—it is either asserted or unasserted in each context. The `!` suffix will be printed with a node's name when that node is asserted in the current context. An *hypothesis* is a node that was asserted by the user using `assert` or `!`, rather than being asserted only because it was derived during inference. An hypothesis is always an hypothesis of one or more context; it may also be asserted in other contexts, and might be unasserted in still other contexts.

Every node is said to be *in* zero or more contexts. A node n is in a context c in any of the following cases:

- n is one of the hypotheses that define c.
- n has been derived from a set of assumptions that is a subset of the set of hypotheses of c.
- n is dominated by a node in c.

1.7 SNePSUL Variables

SNePSUL, the SNePS User Language, has variables which are entirely distinct from SNePS variable nodes. The value of a SNePSUL variable is always a set of objects, `nil` if nothing else. A SNePSUL variable may be given a value with the `?`, `#`, or `$` macro commands, or with the `=` infix command. The value of a SNePSUL variable is obtained by using the `*` macro command. SNePSUL variables created and maintained by SNePS are:

`nodes`	The set of all nodes in the network.
`assertions`	The set of all nodes in the network that were asserted by the user.
`patterns`	The set of all pattern nodes in the network.
`varnodes`	The set of variable nodes in the network.
`relations`	The set of defined arc labels.
`variables`	The set of SNePSUL variables.
`defaultct`	The name of the default context.
`commands`	The set of SNePSUL commands.
`topcommands`	The set of commands valid at SNePS top-level.
`bnscommands`	The set of commands valid at node-set positions in build-type commands.
`fnscommands`	The set of commands valid at node-set positions in find-type commands.
`rscommands`	The set of commands valid at relation-set positions in commands.
`nscommands`	The set of commands valid at at node-set positions in other commands.

Chapter 2

SNePSUL Commands

2.1 Context Specifiers

In a number of commands described in this chapter, part of the syntax is *context-specifier*, and the semantics mentions the context specified by *context-specifier*. In every such case, the possible syntax of *context-specifier*, and what context is specified by each possibility is:

omit If the *context-specifier* is omitted, the specified context is the default context (the value of `*defaultct`).

`:context` The context specified is the default context (the value of `*defaultct`).

`:context` *context-name* The context specified is that named *context-name*, which must be a symbol.

`:context` *nodeset context-name* The context specified is that named *context-name*, which is initialized to be the context whose set of hypotheses is the value of *nodeset*, which must be a SNePSUL expression that evaluates to a set of proposition nodes.

`:context all-hyps` The context specified is the one whose set of hypotheses is the set of all hypotheses—all assertions entered by the user.

2.2 Loading SNePS

Ask whoever maintains SNePS at your site how to load SNePS. Typically, this involves running COMMON LISP, and then loading SNePS.

2.3 Entering and Leaving SNePS

The commands in this section move the user between the SNePSUL evaluator and the COMMON LISP evaluator. Although every SNePSUL function is a COMMON LISP function, the SNePSUL loop provides certain special facilities, so it is best to be in the proper top-level loop for extended work.

`(sneps )`
Lisp function that brings the user into the SNePS read-eval-print loop.

`(lisp )`
SNePSUL function that returns the user to the Lisp evaluator.

`^`
SNePSUL command that causes the next form to be evaluated by Lisp.

`^^`
SNePSUL command that puts the user into an embedded Lisp read-eval-print loop until the next occurrence of the form `^^`, whereupon the user is returned to the SNePSUL loop.

2.4 Using Auxiliary Files

2.4.1 Reading/Writing Files

The commands in this section provide for the use of auxiliary files for the storage of networks or of sequences of commands.

`(outnet` *file*`)`
Stores the current network on the *file* in a special SNePS format. The syntax for the file specification is machine dependent.

`(innet` *file*`)`
If *file* was created by a call to `outnet`, the current network will be initialized to the one stored on *file*. **Note:** innet rewrites the entire network and several SNePSUL variables, so it cannot be used to combine several networks. An error message is issued if *file* is not in the appropriate format.

`(intext` *file*`)`
Reads a sequence of SNePSUL commands from the *file* and executes them, without doing any printing. All assertions specified by the file are done as one batch at the end of the loading process, and they are all asserted into the current context.

`(demo &optional` *file pause*`)`
Reads from the *file*, echoes it, and behaves as if that stream had been typed directly into SNePS. (You can even call `demo` recursively.) If *file* is a string of length 1 that does not name a file or is a symbol whose name is a string of length 1, then a menu of possible demonstrations is printed, and the user may pick one of them. If *file* is an integer, and the menu lists at least that many demonstrations, the one with that number will be run. If *pause* is given, its value may be any of `t`, `b`, `bv`, `a`, `av`, or `n`. If *pause* is `t`, `b`, or `bv`, SNePS will pause before each input command is read. If *pause* is `a`, or `av`, SNePS will pause just after each input is read, but before it is executed. If *pause* is omitted or is `n`, SNePS will not pause at all. If *pause* is `av` or `bv`, a pause message will be printed when the pause occurs; otherwise the message will not be printed. If both arguments are omitted, the menu will be shown, and *pause* defaults to `av`. When SNePS pauses, the following commands are available:

`h,?`	Print this help message
`l,^`	Enter Lisp read/eval/print loop
`s,%`	Enter SNePS toplevel loop,
`o,:`	Enter SNePSLOG
`c`	Continue without pausing
`p`	Set pause control
`q`	Quit this demo
`a`	Quit all demos
any other key	Continue the demo

All these commands are also available inside demo files. This enables you, for example, to turn on pausing at some interesting point in your demo and to run quickly through all the setup stuff, or turn pausing off, or enter a Lisp top-level somewhere or whatever. Here are the commands that allow you to do that. (These are not SNePSUL commands, but they are specially interpreted demo control commands. The DC stands for demo control):

```
dc-pause-help
dc-lisp
dc-sneps
dc-snepslog
dc-no-pause
dc-set-pause ...takes an argument, e.g., (dc-set-pause av)
dc-read-pause
dc-quit
dc-quit-all
```

All commands except `dc-set-pause` are atomic. They can be given in upper or lower case, and they are available in SNePSLOG and the parser as well. However, the way the parser reads input they have to be followed by a "`.`" if sentences are terminated that way, and `dc-set-pause` has to be given as `DC-SET-PAUSE bv.` because the function `parser::atn-read-sentence` collects tokens into a list automatically.

2.4.2 Writing/Altering Source Files For Use With SNePS 2.6 and ACL 6

ACL 6 vs. Other Versions of Lisp

In any LISP other than ACL 6, SNePS 2.6 should run like SNePS 2.5, with no noticable differences. As described in Section 1.2, ACL 6 differs in the case of its pre-defined symbols and input. This means that some source files that ran successfully in SNePS 2.5 might not run successfully in SNePS 2.6 using ACL 6.

To insure portability across LISP systems, any new source files should be written per the advice in the SNeRG Technical Note 30, "Notes on Converting to ACL 6", by Stuart C. Shapiro, which can be found as Reference Number 2001-5 at:
`http://www.cse.buffalo.edu/sneps/Bibliography/`
This will be referred to from now on as SNeRG TN 30.

If a programmer wishes to load a *pre-existing* SNePS input file using SNePS 2.6 running in ACL 6, they have two options[1]:

1. Make sure the file (and any other files involved) are ACL 6 compatible — refer to the SNeRG TN 30 described above

2. Wrap the main file (i.e. the single file which contains SNePS input and/or loads any other input files) in the code shown below.

Code Wrap For ACL 6 and SNePS 2.6 Compatibility

If a pre-existing source file does not run successfully using ACL 6 and SNePS 2.6 and altering *all* the files involved is not desirable, the following code can be wrapped around the main input file — this should result in a successful run. Some minor code changes might be necessary (per SNeRG TN 30), but any inconsistency of case (e.g. `NIL` vs. `nil`) in SNePS code or in the input lines will be adjusted by the wrap. The code wrap is intended to be read at the top LISP level.

Insert the following code at the **beginning** of the main source file:

[1] NOTE: These suggestions work if the pre-existing file runs successfully using SNePS 2.5. They are especially important if the run includes loading altered SNePS code. Older files might need further adjusting.

```
;;; adjustment for ACL6
#+(and allegro-version>= (version>= 6 0))
(sneps:adjust-for-acl6 :before)
```

Insert the following code at the **end** of the main source file:

```
;;; adjustment for ACL6
#+(and allegro-version>= (version>= 6 0))
(sneps:adjust-for-acl6 :after)
```

2.5 Relations

By *relation* in this manual, we mean any relation used to label network arcs. Therefore "relation" and "arc label" are used interchangeably. Whenever an arc labelled R goes from node x to node y, SNePS considers an arc labelled $R-$ to go from y to x. Relation names ending in the character `#\-` are reserved for this "reverse arc" or "converse relation" labelling. Therefore no relation name may end with a `#\-`. The term *relation* always refers to a normal, "forward" arc label. We will use the term *unitpath* to mean either a relation name or the name of its converse relation.

`(define` $\{$*relation*$\}^*$`)`
Defines each *relation* to be an arc label. The name of a relation must not end in the character `#\-`. Each *relation* is added to the SNePSUL variable `relations`. An informative message is given if a relation has previously been defined. Initially, SNePS has a set of relations defined as if the following had been executed:

```
(define forall exists pevb
        min max thresh threshmax emin emax etot
        ant &ant cq dcq arg default
        if when vars suchthat do
        condition then else
        action act plan goal precondition effect
        object1 object2)
```

For uses of the predefined relations, see Sections 3.1.1, "Connectives," 3.1.2, "Quantifiers," and Chapter 4, "SNeRE."

`(undefine` $\{$*relation*$\}^*$`)`
Undefines each *relation*. If any *relation* is being used in the current network, the arcs are not removed from the network structure, but they do become undefined. `undefine` is most useful in correcting typographical errors in calls to define.

2.5.1 Reduction Inference

An asserted node with a certain set of arcs emanating from it implies another node with a subset of those arcs. Using this implication to derive new nodes is called "reduction inference," and is implemented and used by `deduce`. For example,

```
* (describe (assert member (snoopy rover) class (dog animal)))
(M1! (CLASS ANIMAL DOG) (MEMBER ROVER SNOOPY))
(M1!)
 CPU time : 0.08

* (describe (deduce member snoopy class dog))
```

```
(M2! (CLASS DOG) (MEMBER SNOOPY))
(M2!)
 CPU time : 0.05

* (describe (assert agent john act gives object book-1 recipient mary))
(M3! (ACT GIVES) (AGENT JOHN) (OBJECT BOOK-1) (RECIPIENT MARY))
(M3!)
 CPU time : 0.08

* (describe (deduce agent john act gives object book-1))
(M4! (ACT GIVES) (AGENT JOHN) (OBJECT BOOK-1))
(M4!)
 CPU time : 0.05
```

Warning: According to Shapiro, 1991,[2] if you `build` a node that is implied via reduction inference by an already asserted node, the new node will automatically be asserted. This is not implemented in the current version of SNePS 2.

2.5.2 Path-Based Inference

Path-based inference allows an arc between two nodes to be inferred from the presence of a path of arcs between them. The various versions of `find` as well as `deduce` will use any path-based inference rules that have been declared.

`(define-path` $\{$*relation path*$\}^*$`)`
Declares the path-based inference rule,

$$\forall(n_1, n_2)path(n_1, n_2) \Rightarrow relation(n_1, n_2).$$

I.e., if a path of arcs specified by *path* is in the network going from node n_1 to node n_2, then the single arc labelled by *relation* is inferred as going from node n_1 to node n_2. See the following subsection for the syntax of *path*. No *relation* may have more than one path-based inference rule for it at any time. This is not a restriction, since a disjunction of paths is also a path. **Warning:** A path-based inference rule will not be expanded recursively. I.e., no relation (or converse relation) in the path will be expanded even if a path-based inference rule has been declared for it. A subtle implication of this is that it is almost always proper to do `(define-path` *relation* `(or` *relation new-path*`))`, so that explicit occurrences of *relation* will be recognized.

`(undefine-path` $\{$*relation path*$\}^*$`)`
Deletes the given path-based inference rules.

Syntax and Semantics of Paths

A *unitpath* is simply a single arc followed in the forward or the reverse direction. A *path* can be a sequence of unitpaths, or a more complicated way of getting from one node to another. Keep in mind the distinctions between *relation*, *unitpath*, and *path*, since there are places where it matters.

unitpath ::= relation
Any single arc *relation* is also a *unitpath*.

[2]S. C. Shapiro, Cables, paths and "subconscious" reasoning in propositional semantic networks. In J. Sowa, Ed. *Principles of Semantic Networks: Explorations in the Representation of Knowledge.* Morgan Kaufmann, San Mateo, CA, 1991, 137–156.

unitpath ::= relation`-`
If R is a relation from node x to node y, then R`-` is a unitpath from y to x.

path ::= unitpath
Any single arc, either forward or backward, is a *path*.

path ::= `(converse` *path*`)`
If P is a path from node x to node y, then `(converse` P`)` is a path from y to x.

path ::= `(compose` $\{$*path* `|` `!`$\}^*$`)`
If $x_1, \ldots, x_n$ are nodes and P_i is a path from x_i to x_{i+1}, then `(compose` P_1 `...` P_{n-1}`)` is a path from x_1 to x_n. **Note:** If the symbol `!` appears between P_{i-1} and P_i, then x_i must be asserted in the current context. **Examples:** 1) After doing `(build member socrates class man)`, the path `(compose member- class)` goes from `socrates` to `man`, but the path `(compose member- ! class)` doesn't. However, after doing `(assert member socrates class man)`, both paths exist. 2) `(find (compose !) *nodes)` is a way to find all nodes that are asserted in the current context.

path ::= `(kstar` *path*`)`
If path P composed with itself zero or more times is a path from node x to node y, then `(kstar` P`)` is a path from x to y.

path ::= `(kplus` *path*`)`
If path P composed with itself one or more times is a path from node x to node y, then `(kplus` P`)` is a path from x to y.

path ::= `(or` $\{$*path*$\}^*$`)`
If P_1 is a path from node x to node y or P_2 is a path from x to y or . . . or P_n is a path from x to y, then `(or` P_1 P_2 `...`P_n`)` is a path from x to y.

path ::= `(and` $\{$*path*$\}^*$`)`
If P_1 is a path from node x to node y and P_2 is a path from x to y and . . . and P_n is a path from x to y, then `(and` P_1 P_2 `...`P_n`)` is a path from x to y.

path ::= `(not` *path*`)`
If there is no path P from node x to node y, then `(not` P`)` is a path from x to y. **Warning:** Belief revision will not work for nodes that were inferred via path-based inference that used `not` arcs.

path ::= `(relative-complement` *path path*`)`
If P is a path from node x to node y and there is no path Q from x to y, then `(relative-complement` P Q`)` is a path from x to y. **Warning:** Belief revision will not work for nodes that were inferred via path-based inference that used `relative-complement` arcs.

path ::= `(irreflexive-restrict` *path*`)`
If P is a path from node x to node y, and $x \neq y$, then `(irreflexive-restrict` P`)` is a path from x to y.

path ::= `(exception` *path path*`)`
If P is a path from node x to node y and there is no path Q from x to y with length less than or equal to the length of P, then `(exception` P Q`)` is a path from x to y.

path ::= `(domain-restrict (`*path node*`)` *path*`)`
If P is a path from node x to node y and Q is a path from x to node z, then `(domain-restrict (`Q z`)` P`)` is a path from x to y.

path ::= `(range-restrict` *path* `(`*path node*`))`
If P is a path from node x to node y and Q is a path from y to node z, then `(range-restrict` P `(`Q z`))` is a path from x to y.

path ::= `(`*path**`)`
If P_1 is not one of the symbols `and, converse, compose, exception, kstar, kplus`, `not`, `or`, `relative-complement`, `irreflexive-restrict`, `domain-restrict`, or `range-restrict,` then
`(`P_1 ... P_{n-1}`)` is equivalent to `(compose` P_1 ... P_{n-1}`)`.

2.6 Operating on Contexts

`(set-context` *nodeset [symbol]*`)`
Creates a context whose hypothesis set is *nodeset* (which cannot contain pattern nodes). If `symbol` is given, that is made the name of the context; otherwise `*defaultct` becomes the name of the context.

`(set-default-context` *context-name*`)`
Changes the default context (the value of `*defaultct`) to be *context-name*.

`(add-to-context` *nodeset [context-name]*`)`
Adds the nodes of *nodeset* into the hypothesis set of the context, *context-name*. If *context-name* is omitted, adds the hypotheses to `*defaultct`.

`(remove-from-context` *nodeset [context-name]*`)`
Removes the nodes of *nodeset* from the hypothesis set of the context, *context-name*. If *context-name* is omitted, removes the hypotheses from `*defaultct`.

`(list-context-names )`
Prints a list of all valid context names.

`(describe-context` *[context-name]*`)`
Prints the hypothesis set, restriction set, and all names of the context named *context-name*. If *context-name* is omitted, prints the information on `*defaultct`.

`(list-hypotheses` *[context-name]*`)`
Returns the hypothesis set of the context named *context-name*. If *context-name* is omitted, returns the hypothesis set of `*defaultct`.

2.7 Building Networks

The commands of this section add information to the network, either in the form of a node, a node and some arcs, or an assertion tag. It is not possible to add just an arc to the network. Isolated nodes cannot be added to the network, so the commands `#` and `$` can only be used within the lexical context of a `build`, `assert`, or `add`.

We will use the term *wire* to mean a labelled arc and the node it points to. So a molecular node has a set of wires coming out of it.

`(build` {*relation nodeset*}* `)`
`(assert` {*relation nodeset*}* *context-specifier* `)`
`(add` {*relation nodeset*}* *context-specifier* `)`
Puts a node in the network with an arc labelled *relation* to each node in the following *nodeset*, and returns a singleton set containing the built node. The new node is added to the value of the SNePSUL variable `nodes`. If this new node would look exactly like an already existing node, i.e., would have exactly the same set of wires emanating from it, then no node is built, but a singleton set containing the extant node is returned. `build` creates an unasserted node. `assert` is just like `build`, but creates the node as an asserted node (an hypothesis), and adds it to the hypothesis set of the context specified by *context-specifier*. `add` is just like `assert`, but, in addition, triggers forward inference. **Note:** where *relation* is specified in the syntax, neither a converse relation nor a non-unit path is allowed. `build` is not a top-level SNePSUL command in SNePS 2.

`(activate` {*nodeset*}* *context-specifier* `)`
Finds all the nodes that dominate the nodes in *nodeset* (including the nodes in *nodeset* themselves), and that are asserted in the context specified by *context-specifier*, and triggers forward inference on them.

`(!`*node context-specifier* `)`
A postfix command that asserts *node* in the context specified by *context-specifier*, and returns a singleton set containing *node*.

`(assert ...` *context-specifier*`)`	is equivalent to	`(! (build ...)` *context-specifier*`)`.
`(build ...)!`	is equivalent to	`(assert ...)`.

`#`*symbol*
A macro command that creates a new base node, assigns a singleton set containing the new node as the value of the SNePSUL variable *symbol*, and returns that set. This may not be used at the top-level SNePSUL loop, since that would create an isolated node. (Note: The `#` macro is smart enough to guess whether you want to create a base node or whether the standard COMMON LISP dispatching macro is intended. This means that the `#!` "with-snepsul" syntax is available at the SNePS top level, as well as in GATN grammars, etc., and that other common uses, such as `#'` for functions, are available too.)

`$`*symbol*
A macro command that creates a new variable node, assigns a singleton set containing the new node as the value of the SNePSUL variable *symbol*, and returns that set. This may not be used at the top-level SNePSUL loop, since that would create an isolated node.

2.8 Deleting Information

The commands of this section delete information from the network, and are mainly intended for use after mistakes or when debugging.

`(erase` {*nodeset*}* `)`
`(silent-erase`{*nodeset*}* `)`
Removes all nodes in all *nodeset*s from the network along with any nodes that become isolated in the process (that is, all nodes which no longer have any arcs connected to them), and all nodes that were dominated by nodes it erases that are not also dominated by other nodes. Refuses to delete nodes that have arcs coming into them. `silent-erase` is like `erase`, but does no printing.

`(resetnet` *[reset-relations?]*`)`
Reinitializes the network to the state in which no nodes have been built. If *reset-relations?* is `t`, the set of SNePS relations is reset to the pre-defined ones; If *reset-relations?* is `nil` (default), the defines relations and declared path-based inference rules remain as is.

`(clear-infer-all )`
Deletes any information placed in the "active connection graph" version of the network by SNIP. I.e., all deduction rules are returned to their unactivated state as if no inference had yet been performed. It is recommended that `clear-infer` be used instead. See below.

`(clear-infer )`
Like `clear-infer-all`, but retains some pointers from rules to their instances that makes node-based inference faster. `clear-infer` is recommended over `clear-infer-all` unless there is a specific reason to use the latter.

2.9 Functions Returning Sets of Nodes or of Unitpaths

The functions described in this section neither add to nor delete from the network. Rather, they compute and return sets either of nodes or of unitpaths.

`(`$\{node\}^*$`)`
A list of nodes at the top level of the SNePSUL loop, or in a context where a node set is required, is treated as an expression whose value is a set of the nodes in the list.

`(*` *symbol*`)`
A macro command function which returns the set of nodes in the value of the SNePSUL variable *symbol*.

`(list-nodes` *[context-name]*`)`
Returns the set of all nodes that are in the context named *context-name*. If *context-name* is omitted, returns the set of all nodes that are in `*defaultct`.

`(^` *S-expression*`)`
The set of nodes obtained by evaluating the Lisp *S-expression*.

`(&` $nodeset^*$`)`
Infix function that returns the intersection of the *nodeset*s.

`(+` $nodeset^*$`)`
Infix function that returns the union of the *nodeset*s.

`(-` *nodeset nodeset*`)`
Infix function that returns the set of nodes in the first *nodeset* but not in the second *nodeset*.

`(=` *nodeset symbol*`)`
Infix function that assigns the *nodeset* to be the value of the SNePSUL variable *symbol*.

`(_` *nodeset unitpathset*`)`
Infix function that returns the set of those nodes in the *nodeset* which do not have any of the unitpaths in the *unitpathset* emanating from them.

(> *unitpathset symbol*)
Infix function that assigns the *unitpathset* to be the value of the SNePSUL variable *symbol*.

({*unitpath*}*)
A list of unitpaths in a context where a unitpathset is required, is treated as an expression whose value is a set of the unitpaths in the list.

2.10 Displaying the Network

The commands in this section are various ways of printing, or otherwise displaying, the information in the network.

(dump {*nodeset*}* *context-specifier*)
Prints the name of each node in the *nodeset* that is in the context specified by *context-specifier*, along with all arcs going from it or into it, and the nodes that each arc points to or from. For a complete dump of the network, execute `(dump *nodes :context all-hyps)`.

(describe {*nodeset*}* *context-specifier*)
Similar to dump, but: describes only the molecular and pattern nodes in the *nodeset*s; describes all molecular and pattern nodes dominated by nodes it describes; describes any node at most once—the second and later times, only the node's name is printed.

(full-describe {*nodeset*}* *context-specifier*)
Similar to describe, but also shows the context(s) each node is asserted in. Unlike `dump` and `describe`, `full-describe` can describe nodes that are not in any context.

(show *nodeset**)
(show *nodeset** &key :file :format)
Displays the network connected to the nodes in the *nodesets* in a graphical form. The first version uses `JUNG` and `JIMI`, and produces a graph that can be manipulated by hand. The second version saves a specification of the network in the DOT language to a file, from which an output file is produced via the `dot` compiler. The *file* keyword argument specifies the base name of the `.dot` and output files (by default a temporary file). The *format* keyword specifies the format of the output file, which must be either `:gif` (default) or `:ps`. The output file is displayed via either `xv` or `gv`. `dot` produces a static figure. Only one of these versions of `show` is available; which one, depends on the SNePS installer. Neither `dot`, `JUNG`, nor `JIMI` are part of the SNePS distribution. `dot` is part of the Graphviz package which can be downloaded from `http://graphviz.org/`). `JUNG` and it's associated packages, `Xerxes`, `Colt`, and `Jakarta Common Collections`, can be downloaded from `http://jung.sourceforge.net/`. `JIMI` can be downloaded from `http://java.sun.com/products/jimi/`. The `JUNG/JIMI` version of `show` requires either that SNePS is being run under Franz's ACL, or that one of the executable packages of SNePS is being used. The SNePS installer may install either the `dot` version or the JUNG/JIMI version, both, or neither. If both are installed, the user can dynamically pick the version to be used by setting the global variable ¡code¿cl-user:*use-gui-show*¡/code¿/indexctcl-user:*use-gui-show* to ¡code¿t¡/code¿ for the JUNG/JIMI version, or to ¡code¿nil¡/code¿ for the ¡code¿dot¡/code¿ version.

(surface {*nodeset*}*)
Generates a description of each node in each *nodeset* using the currently loaded GATN grammar starting in state `g`.

2.11 Retrieving Information

The functions in this section find nodes in the network, and return them.

`(find` $\{$*path nodeset*$\}^*$ *context-specifier*`)`
`(findassert` $\{$*path nodeset*$\}^*$ *context-specifier*`)`
`(findconstant` $\{$*path nodeset*$\}^*$ *context-specifier*`)`
`(findbase` $\{$*path nodeset*$\}^*$ *context-specifier*`)`
`(findvariable` $\{$*path nodeset*$\}^*$ *context-specifier*`)`
`(findpattern` $\{$*path nodeset*$\}^*$ *context-specifier*`)`
Returns the set of nodes in the specified context such that each node in the set has every specified *path* going from it to at least one node in the accompanying *nodeset*. `(find class (man greek))` will find nodes with a `class` arc to either `man` or `greek`, whereas `(find class man class greek)` will find nodes with `class` arcs to both `man` and `greek`. `find` returns all appropriate nodes in the specified context; `findassert` returns only asserted nodes; `findconstant` returns only base or molecular nodes; `findbase` returns only base nodes; `findvariable` returns only variable nodes; `findpattern` returns only pattern nodes.

`?`*symbol*
May be used in any `find` function in place of a *nodeset*, to stand for "any node." The scope of these symbols is the outermost `find` function and all embedded `find` functions. After return of the outermost `find` function, *symbol* will be a SNePSUL variable whose value will be the set of nodes it matched.

`(deduce [`*numb*`]` $\{$*relation nodeset*$\}^*$ *context-specifier*`)`
`(deducetrue [`*numb*`]` $\{$*relation nodeset*$\}^*$ *context-specifier*`)`
`(deducefalse [`*numb*`]` $\{$*relation nodeset*$\}^*$ *context-specifier*`)`
`(deducewh [`*numb*`]` $\{$*relation nodeset*$\}^*$ *context-specifier*`)`
`(deducewhnot [`*numb*`]` $\{$*relation nodeset*$\}^*$ *context-specifier*`)`
Like `findassert`, but uses SNIP to back-chain on any deduction rules in the specified context. `deducetrue` returns all inferred nodes that satisfy the specification. `deducefalse` returns all inferred nodes that satisfy the negation of the specification. `deduce` returns all inferred nodes that satisfy the specification, and inferred nodes that satisfy the negation of the specification. `deducewh` returns a list of substitutions for the free variables in the specification indicating the set of nodes that would be returned by `deducetrue`. `deducewhnot` returns a list of substitutions for the free variables in the specification indicating the set of nodes that would be returned by `deducefalse`. Note that only *relation*s may appear in the specification, not any other unitpaths or paths. Neither may `?`*symbol* variables appear in the specification. The *numb* argument is optional. If *numb* is omitted, then deduce continues until no more answers can be derived. If *numb* is a single integer, it specifies the total number of answers requested. If *numb* is zero, no inference is done—only answers already in the network are returned. Otherwise, *numb* must be a list of two numbers, `(`*npos nneg*`)`, and deduction terminates after at least $npos$ positive and $nneg$ negative instances are derived.

Chapter 3

SNIP: The SNePS Inference Package

Automatic inference may be triggered using the function `deduce` (see Section 2.11), a generalization of `find`, or the function `add` (see Section 2.7), a generalization of `assert`. In order for these to accomplish anything, deduction rules must exist in the network. A deduction rule is a network structure dominated by a rule node. A rule node represents a logical formula of molecular nodes, using connectives and quantifiers.

3.1 Representing and Using Rules

Rules are placed in the network with the `assert` and `add` commands (see Section 2.7). The arcs needed to build rules are predefined by SNePS.

3.1.1 Connectives

Connectives are the means by which simple propositions are compounded to make more complicated ones. In classical logic, this compounding is accomplished by use of standard connectives such as & (AND) and $\vee$ (OR). A number of disadvantages exist in using standard connectives in SNePS, primarily because of their binary nature and the size of the network needed to store representations with standard connectives. To avoid these problems, SNePS uses non-standard connectives. These non-standard connectives are as adequate as standard connectives, but they take arbitrarily large sets of arguments and express common modes of human reason simply. The non-standard connectives are: and-entailment, or-entailment, numerical entailment, andor, and thresh. An explanation of each connective follows.

And-Entailment

$\{A_1, \ldots, A_n\}$ &$\Rightarrow$ $\{C_1, \ldots, C_m\}$ means that the conjunction of the antecedents implies the conjunction of the consequents. An and-entailment rule is built with the SNePSUL command:

(assert &ant $(A_1, \ldots, A_n)$
        cq $(C_1, \ldots, C_m)$)

Use An asserted and-entailment may be used in forward or backward inference to conclude that one or more of its consequents is to be asserted.

Or-Entailment

$\{A_1, \ldots, A_n\} \vee\!\!\Rightarrow \{C_1, \ldots, C_m\}$ means that the disjunction of the antecedents implies the conjunction of the consequents. An or-entailment rule is built with the SNePSUL command:

(assert ant $(A_1, \ldots, A_n)$
cq $(C_1, \ldots, C_m)$)

Note: or-entailment is more efficient than and-entailment, so if there is only one antecedent, use `ant` rather than `&ant`.

Use An asserted or-entailment may be used in forward or backward inference to conclude that one or more of its consequents is to be asserted.

Numerical Entailment

$\{A_1, \ldots, A_n\}\ i\!\Rightarrow \{C_1, \ldots, C_m\}$ means that the conjunction of any i of the antecedents implies the conjunction of the consequents. In other words, if i or more of the antecedents are true, then all of the consequents are true. A numerical-entailment rule is built with the SNePSUL command:

(assert thresh i
&ant $(A_1, \ldots, A_n)$
cq $(C_1, \ldots, C_m)$)

Use An asserted numerical-entailment may be used in forward or backward inference to conclude that one or more of its consequents is to be asserted.

AndOr

$\bigvee\!\!\!\bigwedge_i^j \{P_1, \ldots, P_n\}$ means that at least i and at most j of the n propositions are true. An andor rule is built with the SNePSUL command:

(assert min i max j
arg $(P_1, \ldots, P_n)$)

The following special cases of andor are representations of standard connectives: $i = j = n$ is AND; $i = j = 0$ is a generalization of NOR; and $i = j = 1$ is a generalization of EXCLUSIVE OR.

Use An asserted and-or may be used in forward or backward inference to conclude that one or more of its arguments is to be asserted, or that the negation of one or more of its arguments is to be asserted. An unasserted and-or for which $i = j =$ *number of arguments* will be asserted during backward inference if all its arguments are asserted.

Thresh

$\Theta_i^j\{P_1, \ldots, P_n\}$ means that either fewer than i or more than j of the n propositions are true. j may be omitted, in which case it defaults to $n - 1$. A thresh rule is built with the SNePSUL command:

(assert thresh i threshmax j
arg $(P_1, \ldots, P_n)$)

If $i = 1$ and j is omitted, the thresh is a generalization of equivalence.

Use An asserted thresh may be used in forward or backward inference to conclude that one or more of its arguments is to be asserted, or that the negation of one or more of its arguments is to be asserted.

3.1.2 Quantifiers

Quantifiers permit the use of variables in deduction rules. The relations `forall` and `exists`, are predefined quantifier relations. They are used to point to variable nodes, indicating for which values of the variable node the rule holds. `forall` and `exists` represent universal and existential quantifiers, respectively. SNePS 2 uses restricted quantification, which means that every quantified expression must have a restriction as well as a scope.

The Universal Quantifier

$\forall(x_1,\ldots,x_n)\{R_1(x_1),\ldots,R_n(x_n)\} : \{P_1(x_1,\ldots,x_n),\ldots,P_m(x_1,\ldots,x_n)\}$ means that for every substitution, $\sigma = \{t_1/x_1,\ldots,t_n/x_n\}$ for which the following conditions hold

- t_i satisfies the restriction $R_i, 1 \leq i \leq n$
- $t_i \neq t_j$ whenever $i \neq j$
- t_i does not already occur in the rule, $1 \leq i \leq n$

$P_i(x_1,\ldots,x_n)\sigma, 1 \leq i \leq m$ is true. There may be fewer restrictions than variables if some restriction contains more than one variable free, as long as every variable occurs in at least one restriction. A universally quantified rule is built with the SNePSUL command:

(assert forall $(x_1,\ldots,x_n)$
&ant $(R_1(x_1),\ldots,R_n(x_n))$
cq $(P_1(x_1,\ldots,x_n),\ldots,P_m(x_1,\ldots,x_n))$)

The first occurrence of a variable must be preceded by the `$` macro, and subsequent occurrences must be preceded by the `*` macro.

If there is only one restriction, `ant` should be used instead of `&ant`.

Use Universal instantiation has been implemented, but not universal generalization.

The Existential Quantifier

The existential quantifier has not yet been implemented in SNePS 2. However, it is not needed, because Skolem functions can be used instead.

Whenever an existentially quantified variable y is bound within the scope of universally quantified variables $x_1,\ldots,x_n$, y can be replaced by the Skolem function $f(x_1,\ldots,x_n)$, as long as f is used nowhere else. The existential quantifier that binds y can then be eliminated.

So, to represent an existentially quantified variable in SNePS 2, define a set of arcs, say `Skf, a1, a2, ...`, and replace the variable node by a molecular node with the `ai` arcs going to the universally quantified variables whose scopes contain the existentially quantified variable, and with the `Skf` arc going to a new base node that serves as the Skolem function. The Skolem function node may be named mnemonically.

For example to represent the formula

$$\forall x(Man(x) \Rightarrow \exists y(Woman(y) \wedge Loves(x,y)))$$

you might do

```
(assert forall $man
           ant (build member *man class man)
            cq ((build member (build Skf loved-by a1 *man) = thiswoman
                      class  woman)
                (build agent *man act loves object *thiswoman)))
```

The Numerical Quantifier

${}_k\exists_i^j(x_1, \ldots, x_n)\{R_1(x_1), \ldots, R_n(x_n)\}P(x_1, \ldots, x_n)$ means that of the k substitutions, $\sigma = \{t_1/x_1, \ldots, t_n/x_n\}$ for which the following conditions hold

- t_i satisfies the restriction $R_i, 1 \leq i \leq n$
- $t_i \neq t_j$ whenever $i \neq j$
- t_i does not already occur in the rule, $1 \leq i \leq n$

between i and j of them also make $P(x_1, \ldots, x_n)\sigma$ true. There may be fewer restrictions than variables if some restriction contains more than one variable free, as long as every variable occurs in at least one restriction.

A numerically quantified rule is built with the SNePSUL command:

(assert emin i emax j etot k pevb $(x_1, \ldots, x_n)$
&ant $(R_1(x_1), \ldots, R_n(x_n))$
cq $P(x_1, \ldots, x_n)$)

The first occurrence of a variable must be preceded by the `$` macro, and subsequent occurrences must be preceded by the `*` macro.

The Uniqueness Principle for Variables

Currently, the Uniqueness Principle, that every entity represented in the network is represented by a unique node, is not enforced by SNePS for variables. Therefore, it is advised that the Uniqueness Principle for variables be followed by the SNePSUL user as a matter of style. This should be done as follows. Every restriction R used in a restricted quantifier should have a series of variables, $x_1^R, x_2^R, \ldots$ Every rule that uses R once should use x_1^R as its variable. A rule that uses the restriction R more than once should use x_1^R in the first use of R, x_2^R in the second use of R, etc. This can be done by using the `$` macro to create each variable node the first time the restriction occurs, and the `*` macro on all subsequent occasions, including subsequent rules. For example, the two rules "Every dog is a pet" and "Every dog hates every cat" might be entered as follows, assuming that the restrictions $Dog(x)$ and $Cat(y)$ have not previously been used in the network:

```
(assert forall $dog1
        ant    (build member *dog1 class dog)
        cq     (build member *dog1 class pet))
(assert forall (*dog1 $cat1)
        &ant   ((build member *dog1 class dog)
                (build member *cat1 class cat))
        cq     (build agent *dog1 act hates object *cat1))
```

3.1.3 Recursion

Recursive rules such as

```
(assert forall($x $y $z)
        &ant  ((build rel ancestor arg1 *x arg2 *y)
               (build rel ancestor arg1 *y arg2 *z))
          cq  (build rel ancestor arg1 *x arg2 *z))
```

may be used without causing an infinite loop.

Infinite loops caused by backward chaining on rules such as

```
(assert forall $x
        ant    (build member (build fn motherOf fnarg *x)
                      class  duck)
        cq     (build member *x class  duck))
```

or by forward chaining on rules such as

```
(assert forall $x
        ant    (build member *x class number))
        cq     (build member (build fn successor fnarg *x)
                      class  number)
```

are terminated under the control of the global parameters `*depthCutoffBack*` and `*depthCutoffForward*`, respectively. If a subgoal is generated during backward chaining whose depth, in terms of arc paths, exceeds `*depthCutoffBack*`, it is not pursued. Also, if a result is generated during forward chaining whose depth, in terms of arc paths, exceeds `*depthCutoffForward*`, it is not pursued. `*depthCutoffBack*` and `*depthCutoffForward*` are each set by default to 10, and can be changed independently via `setf`.

3.2 Tracing Inference

The variable and functions described in this section let you turn on and off various ways of tracing SNIP's activities. Following these traces requires various degrees of knowledge of how SNIP is implemented. Implementation details, however, are beyond the scope of this manual.

`*infertrace*`
This variable controls an inference trace that is readily understandable by the SNePSUL user. When this inference tracing is enabled, a message is printed whenever: a `deduce` is done; a sub-goal is generated during backward inference; a sub-goal matches a stored assertion; a rule fires. The message indicates which of these is happening, and prints one or more proposition nodes or instantiated pattern nodes. The possible values of `*infertrace*` are:

`nil` This inference tracing is disabled.

`t` Default. Nodes are printed using `describe`.

`surface` Nodes are printed using `surface`. (See Section 2.10.)

`(ev-trace` *process-name** `)`
A SNePSUL top-level command for tracing MULTI processes. If called with one or more arguments (unquoted), it turns on event tracing of those named processes. If called with no arguments, and some processes are being traced, it returns a list of processes being traced. If called with no arguments, and no processes are being traced, it turns on event tracing of all processes. Following these traces requires a knowledge of how SNIP is implemented. They are intended for implementing and debugging new features of SNIP.

`(unev-trace` *process-name** `)`
A SNePSUL top-level command for turning off event tracing of MULTI processes. If called with one or more arguments (unquoted), it turns off event tracing of those named processes. If called with no arguments, it turns off all event tracing.

`(in-trace` *process-name** `)`
A SNePSUL top-level command for tracing MULTI processes. If called with one or more arguments (unquoted), it turns on initiation tracing of those named processes. If called with no arguments, and some

processes are being traced, it returns a list of processes being traced. If called with no arguments, and no processes are being traced, it turns on initiation tracing of all processes. Following these traces requires a knowledge of how SNIP is implemented. They are intended for implementing and debugging new features of SNIP.

`(unin-trace` *process-name** `)`
A SNePSUL top-level command for turning off initiation tracing of MULTI processes. If called with one or more arguments (unquoted), it turns off initiation tracing of those named processes. If called with no arguments, it turns off all initiation tracing.

`(multi::print-regs` *process*`)`
Function that prints the registers of the individual *process* and their current values. Assumes a knowledge of how SNIP is implemented. Intended for implementing and debugging new features of SNIP.
`snip::send-request`
`snip::send-reports`
These two functions may profitably be traced by someone familiar with how SNIP is implemented. Tracing `snip::send-request` will show the requests being sent, the nodes they are sent to, and the queue of pending processes. Tracing `snip::send-reports` will show the reports being sent, the channels they are being sent through, and the reports coming out of the channels.

Chapter 4

SNeRE: The SNePS Rational Engine

4.1 Acting

SNeRE, The SNePS Rational Engine, is a package that allows for the smooth incorporation of acting into SNePS-based agents. SNeRE recognizes a node with an `action` arc to be a special kind of node called an *act node*. Since an act usually consists of an action and one or more objects of the action, an act node usually has additional arcs pointing to the nodes that represent the objects of the action. These additional arcs are generally labelled `object1` ... `object`n, where n is the number of objects the action is performed on. The relations `object1` and `object2` are pre-defined. If more `object`i are needed, the user must `define` them.

There are three ways to initiate acting. The first is by use of the SNePSUL command `perform`.

`(perform` *`actnode context-specifier`*`)`
Causes the *actnode* to be performed. Deductions and assertions triggered during the performance will be made in the specified context.

```
* (perform (build action say object1 "Hello" object2 "there"))
Hello there
 CPU time : 0.15
```

(How the `say` action is defined will be explained below.)

The other two ways to initiate action are during inference:

1. If a node of the form `M`:{⟨`whenever, p`⟩, ⟨`do, a`⟩} or of the form `M`:{⟨`when, p`⟩, ⟨`do, a`⟩}, where `p` is a proposition node and `a` is an act node, is in the network, and forward inference causes both `M` and `p` to be asserted, then `a` is performed.

   ```
   * (describe
        (assert whenever (build agent Stu state is location here)
                      do (build action say object1 "Hello" object2 "Stu.")))
   (M3! (DO (M2 (ACTION SAY) (OBJECT1 Hello) (OBJECT2 Stu)))
    (WHENEVER (M1 (AGENT STU) (LOCATION HERE) (STATE IS))))
   (M3!)
    CPU time : 0.04
   ```

```
* (describe
     (assert when (build agent Stu state is location here)
                do (build action say object1 "I see" object2 "you're here.")))
(M5! (DO (M3 (ACTION SAY) (OBJECT1 Hello) (OBJECT2 Stu)))
 (WHEN (M2 (AGENT STU) (LOCATION HERE) (STATE IS))))
(M5!)
 CPU time : 0.03

* (add agent Stu state is location here)
Hello Stu
I see you're here
 CPU time : 0.06
```

The difference between `when` and `whenever` is that if the proposition `p` is `disbelieved` (*see* below) and readded, the act controlled by `whenever` will be performed again, but the act controlled by `when` won't.

```
* (perform (build action disbelieve
                  object1 (build agent Stu state is location here)))
 CPU time : 0.18

* (add agent Stu state is location here)
Hello Stu
 CPU time : 0.02
```

2. If an asserted node of the form `M!`: $\{\langle$`if, p`$\rangle, \langle$`do, a`$\rangle\}$, where `p` is a proposition node and `a` is an act node, is in the network, and SNIP back-chains into `p`, then `a` will be performed.

```
* (describe (assert if (build agent who state is location here)
                    do (build action say object1 "Who's" object2 "here?")))
(M7! (DO (M6 (ACTION SAY) (OBJECT1 Who's) (OBJECT2 here?)))
 (IF (M5 (AGENT WHO) (LOCATION HERE) (STATE IS))))
(M7!)
 CPU time : 0.19

* (deduce agent who state is location here)
Who's here?
 CPU time : 0.12
```

4.2 Primitive Acts

The only acts that can actually be performed are *primitive acts*—those whose actions are *primitive actions,* which, themselves, are associated with *primitive action functions.* Several primitive action functions are predefined. The user may define additional ones by using the function `define-primaction`:

`(define-primaction` *action* `(`*relation*$_1$ `...`*relation*$_n$`)` $\{$*form*$\}^*$`)`
This defines *action* to be a LISP function of arity n, whose list of lambda variables is (*relation*$_1$ ... *relation*$_n$), and whose body is $\{$*form*$\}^*$. When the function is called, each lambda variable will be bound to a node set. For example the action function for `say`, used in the examples above, was defined by:

```
(define-primaction say (object1 object2)
  "Print the the argument nodes in order."
  (format t "~&~A ~A~%"
          (sneps:choose.ns object1)
          (sneps:choose.ns object2)))
```

The predefined primitive action functions, and what they do are:

Functions for Mental acts

(believe *object1*), where *object1* must be a proposition node. The following special cases of belief revision are first performed:

- If (M*n*! (MIN 0) (MAX 0) (ARG ... *object1* ...)) is currently asserted, it is disbelieved.
- If (M*n*! (MIN *i*) (MAX 1) (ARG *object1 otherprop* ...)) (for any *i*) and *otherprop* are currently asserted, *otherprop* is disbelieved.

Then *object1* is asserted, and forward inference is done with it.

```
* (assert min 0 max 0
          arg (build agent Stu state is location here))
(M2!)
 CPU time : 0.09

* (describe (deduce agent $who state is location here))
(M2! (MIN 0) (MAX 0) (ARG (M1 (AGENT STU) (LOCATION HERE) (STATE IS))))
(M2!)
 CPU time : 0.18

* (perform (build action  believe
                  object1 (build agent Stu state is location here)))
 CPU time : 0.06

* (describe (deduce agent $who state is location here))
(M1! (AGENT STU) (LOCATION HERE) (STATE IS))
(M1!)
 CPU time : 0.06
```

(disbelieve *object1*), where *object1* must be a proposition node. *object1* is removed-from-context.

```
* (describe (deduce agent $who state is location here))
(M1! (AGENT STU) (LOCATION HERE) (STATE IS))
(M1!)
 CPU time : 0.07

* (perform (build action  disbelieve
                  object1 (build agent Stu state is location here)))
 CPU time : 0.03

* (describe (deduce agent $who state is location here))
 CPU time : 0.07
```

Functions for Control acts

(do-all *object1*), where *object1* is a set of one or more act nodes, causes all of the act nodes to be performed in some arbitrary order.

```
* (perform
   (build action  do-all
          object1 ((build action say object1 "Hello" object2 "Bill")
                   (build action say object1 "Hello" object2 "Stu"))))
Hello Stu
Hello Bill
 CPU time : 0.32
```

(do-one *object1*), where *object1* is a set of one or more act nodes, causes an arbitrary one of the act nodes to be performed. If the variable `snip::*choose-randomly*` is `T` (default), `do-one` will choose its act randomly; if it is `NIL`, it will choose deterministicly, which might be desirable during debugging. (See page 43 for clarification.)

```
* (perform
   (build action  do-one
          object1 ((build action say object1 "Hello" object2 "Bill")
                   (build action say object1 "Hello" object2 "Stu"))))
Hello Bill
 CPU time : 0.15
```

(snsequence *object1 ...objectn*), where *object1 ...objectn* are act nodes, causes *object1 ...objectn* to be performed in that order.

```
* (perform
   (build action  snsequence
          object1 (build action say object1 "Hello" object2 "Bill")
          object2 (build action say object1 "Hello" object2 "Stu")
          object3 (build action say object1 "Hello" object2 "Oscar")))
Hello Bill
Hello Stu
Hello Oscar
 CPU time : 0.31
```

Warning: In the current version of SNePS, if two *object*s of `snsequence` are the same act, it will only be done once, so instead of

```
(build action snsequence
       object1 a1 ...
       objecti ai objecti+1 ai+1 ...
       objectj ai ... objectn an)
```

one should use

```
(build action snsequence
       object1 a1 ...
       objecti ai
       objecti+1 (build action snsequence object1 ai+1 ...
                                          objectj-i ai ... objectn-i an)
```

(snif *object1*), where *object1* is a set of *guarded acts,* and a guarded act is either of the form $\{\langle$`condition,` *p*$\rangle$, $\langle$`then,` *a*$\rangle\}$, or of the form $\{\langle$`else,` *elseact*$\rangle\}$, where *p* is

a proposition node and *a* and *elseaact* are act nodes. `snif` chooses at random one of the guarded acts whose `condition` is asserted, and performs its act. If none of the `conditions` is asserted and the `else` clause is present, the *elseact* is performed.

```
* (describe (assert agent "Stu" state is location here))
(M6! (AGENT Stu) (LOCATION HERE) (STATE IS))
(M6!)
 CPU time : 0.02

* (describe (deduce agent $who state is location here))
(M6! (AGENT Stu) (LOCATION HERE) (STATE IS))
(M6!)
 CPU time : 0.17

* (perform
    (build action  snif
           object1
           ((build condition
                   (build agent "Bill" state is location here)
                   then
                   (build action say object1 "Hello" object2 "Bill"))
            (build condition
                   (build agent "Stu" state is location here)
                   then
                   (build action say object1 "Hello" object2 "Stu"))
            (build else
                   (build action say
                          object1 "No one's" object2 "here!")))))
Hello Stu
 CPU time : 0.50
* (perform (build action  disbelieve object1 M6))
 CPU time : 0.02

* (perform
   (build action  snif
          object1
          ((build condition
                  (build agent "Bill" state is location here)
                  then
                  (build action say object1 "Hello" object2 "Bill"))
           (build condition
                  (build agent "Stu" state is location here)
                  then
                  (build action say object1 "Hello" object2 "Stu"))
           (build else
                  (build action say
                         object1 "No one's" object2 "here!")))))
No one's here!
 CPU time : 0.27
```

(sniterate *object1*), where *object1* is a set of guarded acts. If at least one of the guard's `conditions` is asserted, `sniterate` performs the `act` of a random one of the guards whose

`condition` is asserted, and then performs the entire `sniterate` again. When none of the guards has an asserted `condition`, if there is an *`elseact`* it is performed, and the `sniterate` terminates.

```
* (describe (deduce agent $who state is location here))
(M1! (AGENT Bill) (LOCATION HERE) (STATE IS))
(M2! (AGENT Stu) (LOCATION HERE) (STATE IS))
(M1! M2!)
 CPU time : 0.10

* (perform
   (build
     action  sniterate
     object1 ((build
                condition (build agent "Bill" state is location here)
                then
                (build action snsequence
                       object1
                       (build action say object1 "Hello" object2 "Bill")
                       object2
                       (build
                         action  disbelieve
                         object1
                         (build agent "Bill" state is location here))))
              (build
                condition (build agent "Stu" state is location here)
                then
                (build action snsequence
                       object1
                       (build action say object1 "Hello" object2 "Stu")
                       object2
                       (build
                         action disbelieve
                         object1
                         (build agent "Stu" state is location here))))
              (build
                else (build action say
                            object1 "That's" object2 "all")))))
Hello Stu
Hello Bill
That's all
 CPU time : 0.83
```

`(withall` *`vars suchthat do`* `[`*`else`*`])`, where *`vars`* is a set of variable nodes, *`suchthat`* is a proposition with *`vars`* free, *`do`* is an act node with *`vars`* free, and *`else`* is an act node with no free variables. `withall` finds all substitutions for *`vars`* for which *`suchthat`* is asserted, and performs all those instances of *`do`*. If there are no such substitutions and *`else`* is present, it is done.

```
(describe (deduce agent $who state is location here))
 CPU time : 0.08

* (perform
   (build action   withall
          vars     $x
          suchthat (build agent *x state is location here)
          do       (build action say object1 "Hello" object2 *x)
          else     (build action say object1 "No one's" object2 "here")))
No one's here.
 CPU time : 0.34

* (describe (assert agent "Stu" state is location here))
(M3! (AGENT Stu) (LOCATION HERE) (STATE IS))
(M3!)
 CPU time : 0.08

* (describe (assert agent "Bill" state is location here))
(M4! (AGENT Bill) (LOCATION HERE) (STATE IS))
(M4!)
 CPU time : 0.07

* (perform
   (build action   withall
          vars     $x
          suchthat (build agent *x state is location here)
          do       (build action say object1 "Hello" object2 *x)
          else     (build action say object1 "No one's" object2 "here")))
Hello Bill.
Hello Stu.
 CPU time : 0.54
```

(withsome *vars suchthat do* [*else*]), where *vars* is a set of variable nodes, *suchthat* is a proposition with *vars* free, *do* is an act node with *vars* free, and *else* is an act node with no free variables. withsome finds some substitution for *vars* for which *suchthat* is asserted, and performs that instance of *do*. If there is no such substitution, and *else* is present, it is performed.

```
* (describe (deduce agent $who state is location here))
(M3! (AGENT Stu) (LOCATION HERE) (STATE IS))
(M4! (AGENT Bill) (LOCATION HERE) (STATE IS))
(M3! M4!)
 CPU time : 0.19

* (perform
   (build action   withsome
          vars     $x
          suchthat (build agent *x state is location here)
          do       (build action say object1 "Hello" object2 *x)
          else     (build action say object1 "No one's" object2 "here")))
Hello Stu.
 CPU time : 0.43
```

```
* (perform
   (build action  disbelieve
          object1 (build agent "Stu" state is location here)))
 CPU time : 0.03

* (perform
   (build action  disbelieve
          object1 (build agent "Bill" state is location here)))
 CPU time : 0.03

* (perform
   (build action   withsome
          vars     $x
          suchthat (build agent *x state is location here)
          do       (build action say object1 "Hello" object2 *x)
          else     (build action say object1 "No one's" object2 "here")))
No one's here.
 CPU time : 0.42
```

Notice that `withall` and `withsome` only operate on entities already believed to satisfy the *suchthat* criterion. If you want to operate on entities discovered in the future to satisfy the *suchthat* criterion, use `when-do` or `whenever-do`, and note that they only perform on new beliefs:

```
* (describe (assert member ("Stu" "Bill") class person))
(M1! (CLASS PERSON) (MEMBER Bill Stu))
(M1!)
 CPU time : 0.07

* (describe (assert agent "Stu" state is location here))
(M2! (AGENT Stu) (LOCATION HERE) (STATE IS))
(M2!)
 CPU time : 0.06

* (describe
   (assert forall $p
           ant (build member *p class person)
           cq  (build when (build agent *p state is location here)
                      do   (build action  say
                                  object1 "Hello new"
                                  object2 *p))))
(M3! (FORALL V1) (ANT (P1 (CLASS PERSON) (MEMBER V1)))
 (CQ
  (P4 (DO (P3 (ACTION SAY) (OBJECT1 Hello new) (OBJECT2 V1)))
   (WHEN (P2 (AGENT V1) (LOCATION HERE) (STATE IS))))))
(M3!)
 CPU time : 0.19
```

```
* (perform
   (build action withall
          vars $x
          suchthat (build min 2 max 2
                          arg ((build member *x class person)
                               (build agent *x state is location here)))
          do (build action say object1 "Hello old" object2 *x)))
Hello old Stu
 CPU time : 0.60

* (clear-infer)
(Node activation cleared. Some register information retained.)
 CPU time : 0.01

* (add agent "Bill" state is location here)
Hello new Bill
 CPU time : 0.31
```

(The `(clear-infer)` was needed to mark the change in time and discourse context.)

4.3 Associating Primitive Action Nodes with Their Functions

SNeRE will recognize an act node by its `action` arc to another node. However, if that latter node represents a primitive action, it must be associated with a primitive action function. To provide flexibility in the representation of primitive actions, the user is obliged to explicitly associate primitive action nodes with their functions.

`(attach-primaction` {*`action-node-form action-function-name`*}*`)`
`action-node-form` must be a SNePSUL form that evaluates to a node *`n`* or a singleton nodeset `(`*`n`*`)`, and *`action-function-name`* must be a symbol that was defined to name a primitive action function *`f`*. These are associated with each other so that when an act node whose action node is *`n`* is performed, *`f`* is applied to the nodesets at the end of the arcs whose relations are the lambda variables of *`f`*.

As an example, we will establish three primitive actions using a variety of representation schemes.

```
* ^^
--> (define-primaction sayfun (object1 object2)
      "Print the the argument nodes in order followed by a period."
      (format t "~&~A ~A.~%"
              (first (ns-to-lisp-list object1))
              (first (ns-to-lisp-list object2))))
SAYFUN
--> (define-primaction exclaimfun (object1 object2)
      "Print the the argument nodes in order followed by an exclamation mark."
      (format t "~&~A ~A!~%"
              (first (ns-to-lisp-list object1))
              (first (ns-to-lisp-list object2))))
EXCLAIMFUN
--> (define-primaction questionfun (np vp)
      "Print the the argument nodes in order followed by a question mark."
      (format t "~&~A ~A?~%"
              (first (ns-to-lisp-list np))
```

```
               (first (ns-to-lisp-list vp))))
QUESTIONFUN
--> (attach-primaction
      say sayfun
      (= (build lex "exclaim") exclaim) exclaimfun
      (find entity- (assert entity (\# 'question) expression "question"))
      questionfun)
T
--> ^^
 CPU time : 0.17

* (perform (build action say object1 "Hello" object2 "Stu"))
Hello Stu.
 CPU time : 0.18

* (perform (build action *exclaim object1 "Hello" object2 "Bill"))
Hello Bill!
 CPU time : 0.14

* (perform
 (build action *question np "Who's" vp "there"))
Who's there?
 CPU time : 0.16
```

Figure 4.1 shows the three act nodes that were performed. The `attach-primaction` call shown above associated action node `SAY` with the primitive action function `sayfun`, action node `M1` with the primitive action function `exclaimfun`, and action node `B1` with the primitive action function `questionfun`.

The user must remember to use `attach-primaction` to associate action nodes even with the built-in primitive action functions she intends to use. As a reminder, the built in action functions are listed in Table 4.1. The `achieve` primitive action function will be described below.

Table 4.1: Built-in Primitive Action Functions

believe	disbelieve	achieve
do-one	do-all	snsequence
snif	sniterate	
withsome	withall	

4.4 Complex Acts

An act that is not a primitive act is called a *complex act.* If SNeRE is asked to perform a complex act, it will try to infer a *plan* to carry out the act. A *plan* in the SNeRE formalism is represented by any act node, but especially one whose action is a control action. A node of the form `M:`$\{\langle$`plan, p`$\rangle, \langle$`act, a`$\rangle\}$, where `a` is a complex act node, and `p` is a plan node, represents the proposition that the plan represented by `p` is the way to perform the complex act represented by `a`. Having inferred some plans for carrying out a complex act, SNeRE will perform `do-one` on them.

To illustrate the use of complex acts, we will first define `say` as a one object action function, and associate action nodes with the functions `say`, and `snsequence`.

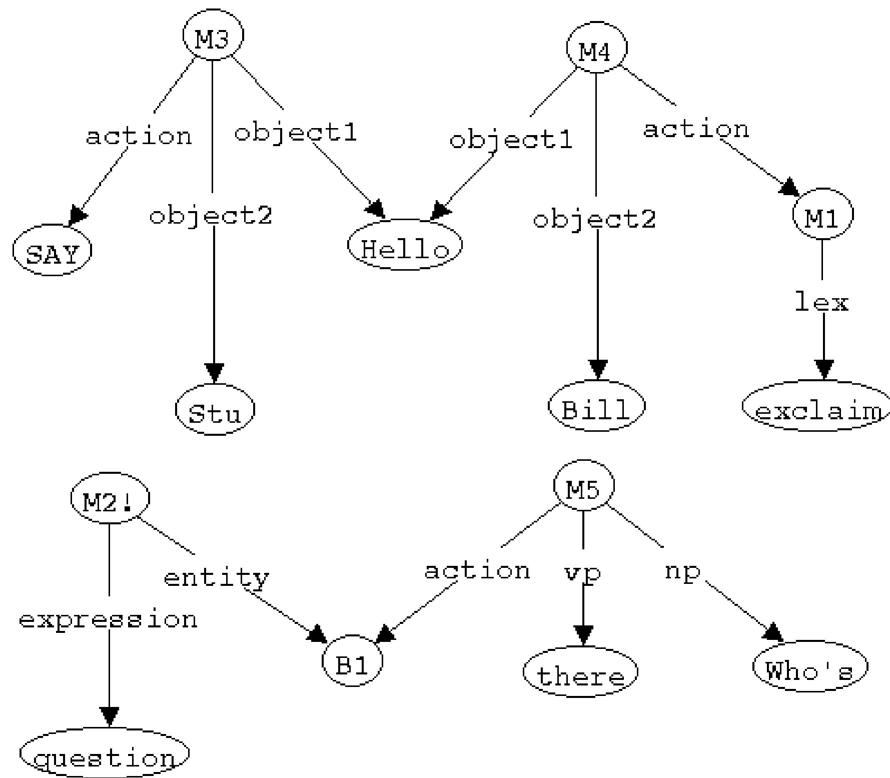

Figure 4.1: Three ways of associating action nodes with action functions. `M3`, `M4`, and `M5` are act nodes, with action nodes `SAY`, `M1`, and `B1` respectively.

```
^^
--> (define-primaction say (object1)
      "Print the object."
      (format t "~&~A" (sneps:choose.ns object1)))
SAY
--> (attach-primaction
      say say
      snsequence snsequence)
T
--> ^^
 CPU time : 0.05
```

Then, we will give a rule that says the way to greet a person is to `sayHi`, then say the person's name (and assert that Stu and Bill are people).

```
* (describe (assert forall $person
                    ant    (build member *person class person)
                     cq    (build act  (build action greet object1 *person)
                                  plan (build action  snsequence
                                              object1 sayHi
                                              object2 (build action  say
                                                             object1 *person)))))

(M1! (FORALL V1) (ANT (P1 (CLASS PERSON) (MEMBER V1)))
 (CQ
  (P5 (ACT (P2 (ACTION GREET) (OBJECT1 V1)))
   (PLAN
    (P4 (ACTION SNSEQUENCE) (OBJECT1 SAYHI)
     (OBJECT2 (P3 (ACTION SAY) (OBJECT1 V1))))))))
(M1!)
 CPU time : 0.18

* (describe (assert member (Stu Bill) class person))
(M2! (CLASS PERSON) (MEMBER BILL STU))
(M2!)
 CPU time : 0.06
```

We will give three plans for `sayHi`.

```
* (describe (assert act  sayHi
                    plan (build action say object1 "Hello")))
(M4! (ACT SAYHI) (PLAN (M3 (ACTION SAY) (OBJECT1 Hello))))
(M4!)
 CPU time : 0.09

* (describe (assert act  sayHi
                    plan (build action say object1 "Hi")))
(M6! (ACT SAYHI) (PLAN (M5 (ACTION SAY) (OBJECT1 Hi))))
(M6!)
 CPU time : 0.06
```

```
* (describe (assert act  sayHi
                    plan (build action say object1 "Hiya")))
(M8! (ACT SAYHI) (PLAN (M7 (ACTION SAY) (OBJECT1 Hiya))))
(M8!)
 CPU time : 0.09
```

and finally, `greet` Stu and Bill.

```
* (perform (build action greet object1 Stu))
Hiya
STU
 CPU time : 1.34

* (perform (build action greet object1 Bill))
Hello
BILL
 CPU time : 1.37
```

A complex act node may be represented by a node with no `action` arc emanating from it, as long as a plan can be derived for it.

```
* (describe (assert act  ask
                    plan (build action say object1 "Who's there?")))
(M7! (ACT ASK) (PLAN (M6 (ACTION SAY) (OBJECT1 Who's there?))))
(M7!)
 CPU time : 0.05

* (perform ask)
Who's there?
 CPU time : 0.33
```

4.5 Goals

In the SNeRE formalism, a *goal* is a proposition that the SNeRE agent is trying to bring about. The action of trying to bring about a goal is called "`achieve`":

(achieve *object1*), where *object1* must be a proposition node, is performed by finding plans for achieving *object1*, and performing a `do-one` on them.

The plans for achieving goals are given by assertions of the form `M!`:$\{\langle$`goal, g`$\rangle, \langle$`plan, p`$\rangle\}$, which says that `p` is a plan for achieving the goal `g`.

```
* (describe
   (assert forall $person
           ant    (build member *person class person)
            cq    (build goal (build agent *person state is location here)
                         plan (build action call object1 *person))))
(M22! (FORALL V5) (ANT (P23 (CLASS PERSON) (MEMBER V5)))
 (CQ
  (P26 (GOAL (P24 (AGENT V5) (LOCATION HERE) (STATE IS)))
   (PLAN (P25 (ACTION CALL) (OBJECT1 V5))))))
(M22!)
 CPU time : 0.18
```

```
* (describe (assert forall $person
                    ant    (build member *person class person)
                     cq    (build act  (build action call object1 *person)
                                  plan (build action  snsequence
                                              object1 (build action  say
                                                             object1 "Come here")
                                              object2 (build action  say
                                                             object1 *person)))))
(M24! (FORALL V6) (ANT (P27 (CLASS PERSON) (MEMBER V6)))
 (CQ
  (P31 (ACT (P28 (ACTION CALL) (OBJECT1 V6)))
   (PLAN
    (P30 (ACTION SNSEQUENCE) (OBJECT1 (M23 (ACTION SAY) (OBJECT1 Come here)))
     (OBJECT2 (P29 (ACTION SAY) (OBJECT1 V6))))))))
(M24!)
 CPU time : 0.19

* (perform (build action  achieve
                  object1 (build agent Bill state is location here)))
Come here
BILL
 CPU time : 1.76
```

4.6 The Execution Cycle: Preconditions and Effects

SNeRE acting may be understood by the following pseudo-definition of `perform`, although the actual implemention is different.

```
perform(act):
   preconds := set of preconditions of act;
   unachieved-preconditions := preconds - {p | p ∈ precond & p is deduceable};
   if unachieved-preconditions ≠ nil
      then perform(snsequence(doall({a | p ∈ unachieved-preconditions & a = achieve(p)}),
                              act))
      else {effects := effects of act;
            if act is primitive
               then {apply(primitive-function(act), objects(act));
                     doall({a | p ∈ effects & a = believe(p)})}
               else {plans := plans for carrying out act;
                     perform(snsequence(do-one(plans),
                                        doall({a | p ∈ effects & a = believe(p)})))}}.
```

Notes and comments:

- A trace of the acting system is printed when the global variable `*plantrace*` is set to `T`. If `*plantrace*` is set to `'surface`, then nodes are sent to the GATN generator starting at state `G` for printing. If `*plantrace*` is `NIL`, no trace is printed. This was the setting for the previous examples in this chapter, but the default is `T`.
- The preconditions of an act, `a` are all `p` for which propositions of the form `M!:{⟨act, a⟩, ⟨precondition, p⟩}` are deduceable.

```
* (describe
     (assert forall *person
             ant    (build member *person class person)
              cq    (build act          (build action greet object1 *person)
                           precondition (build agent    *person
                                               state    is
                                               location here))))
(M34! (FORALL V6) (ANT (P27 (CLASS PERSON) (MEMBER V6)))
 (CQ
  (P40 (ACT (P39 (ACTION GREET) (OBJECT1 V6)))
   (PRECONDITION (P38 (AGENT V6) (LOCATION HERE) (STATE IS))))))
(M34!)
 CPU time : 0.13

* (perform (build action greet object1 Stu))
About to do
((M9 (ACTION (GREET)) (OBJECT1 (STU))))

I wonder if the act
((M9 (ACTION (GREET)) (OBJECT1 (STU))))
has any preconditions...

The act
((M9 (ACTION (GREET)) (OBJECT1 (STU))))
has a precondition:
((M35! (ACT (M9 (ACTION (GREET)) (OBJECT1 (STU))))
  (PRECONDITION (M33! (AGENT (STU)) (LOCATION (HERE)) (STATE (IS))))))
It is satisfied.

The act
((M9 (ACTION (GREET)) (OBJECT1 (STU))))
has a plan:
((M12! (ACT (M9 (ACTION (GREET)) (OBJECT1 (STU))))
  (PLAN
   (M11 (ACTION (SNSEQUENCE)) (OBJECT1 (SAYHI))
    (OBJECT2 (M10 (ACTION (SAY)) (OBJECT1 (STU))))))))

Intending to do
((M14 (ACTION (DO-ONE))
  (OBJECT1
   (M11 (ACTION (SNSEQUENCE)) (OBJECT1 (SAYHI))
    (OBJECT2 (M10 (ACTION (SAY)) (OBJECT1 (STU))))))))

Now doing: DO-ONE
((M11 (ACTION (SNSEQUENCE)) (OBJECT1 (SAYHI))
  (OBJECT2 (M10 (ACTION (SAY)) (OBJECT1 (STU))))))

Chose to do the act
((M11 (ACTION (SNSEQUENCE)) (OBJECT1 (SAYHI))
  (OBJECT2 (M10 (ACTION (SAY)) (OBJECT1 (STU))))))
```

```
About to do
((SAYHI))

I wonder if the act
((SAYHI))
has any preconditions...

The act
((SAYHI))
has no preconditions:

The act
((SAYHI))
has the following plans:
((M4! (ACT (SAYHI)) (PLAN (M3 (ACTION (SAY)) (OBJECT1 (Hello)))))
 (M6! (ACT (SAYHI)) (PLAN (M5 (ACTION (SAY)) (OBJECT1 (Hi)))))
 (M8! (ACT (SAYHI)) (PLAN (M7 (ACTION (SAY)) (OBJECT1 (Hiya))))))

Intending to do
((M15 (ACTION (DO-ONE))
  (OBJECT1 (M3 (ACTION (SAY)) (OBJECT1 (Hello)))
   (M5 (ACTION (SAY)) (OBJECT1 (Hi))) (M7 (ACTION (SAY)) (OBJECT1 (Hiya))))))

Now doing: DO-ONE
((M3 (ACTION (SAY)) (OBJECT1 (Hello)))
 (M5 (ACTION (SAY)) (OBJECT1 (Hi)))
 (M7 (ACTION (SAY)) (OBJECT1 (Hiya))))

Chose to do the act
((M3 (ACTION (SAY)) (OBJECT1 (Hello))))

About to do
((M3 (ACTION (SAY)) (OBJECT1 (Hello))))

I wonder if the act
((M3 (ACTION (SAY)) (OBJECT1 (Hello))))
has any preconditions...

The act
((M3 (ACTION (SAY)) (OBJECT1 (Hello))))
has no preconditions:

Hello

About to do
((M10 (ACTION (SAY)) (OBJECT1 (STU))))

I wonder if the act
((M10 (ACTION (SAY)) (OBJECT1 (STU))))
has any preconditions...
```

```
The act
((M10 (ACTION (SAY)) (OBJECT1 (STU))))
has no preconditions:

STU
 CPU time : 3.10
```

- The current version of SNeRE will never give up trying to achieve the preconditions of an act it is trying to perform, even if some precondition is impossible to achieve.
- The effects of an act, *a* are all *e* for which propositions of the form `M!`:{⟨`act`, *a*⟩, ⟨`effect`, *e*⟩} are deduceable.

```
* (describe
   (assert forall *person
           ant    (build member *person class person)
            cq    (build act    (build action call object1 *person)
                         effect (build agent    *person
                                       state    is
                                       location here))))

(M36! (FORALL V6) (ANT (P27 (CLASS PERSON) (MEMBER V6)))
 (CQ
  (P41 (ACT (P28 (ACTION CALL) (OBJECT1 V6)))
   (EFFECT (P38 (AGENT V6) (LOCATION HERE) (STATE IS))))))
(M36!)
 CPU time : 0.11

* (perform (build action call object1 Bill))
About to do
((M27 (ACTION (CALL)) (OBJECT1 (BILL))))

I wonder if the act
((M27 (ACTION (CALL)) (OBJECT1 (BILL))))
has any preconditions...

The act
((M27 (ACTION (CALL)) (OBJECT1 (BILL))))
has no preconditions:

The act
((M27 (ACTION (CALL)) (OBJECT1 (BILL))))
has a plan:
((M31! (ACT (M27 (ACTION (CALL)) (OBJECT1 (BILL))))
  (PLAN
   (M30 (ACTION (SNSEQUENCE))
    (OBJECT1 (M23 (ACTION (SAY)) (OBJECT1 (Come here))))
    (OBJECT2 (M17 (ACTION (SAY)) (OBJECT1 (BILL))))))))
```

```
Intending to do
((M40 (ACTION (DO-ONE))
  (OBJECT1
   (M30 (ACTION (SNSEQUENCE))
    (OBJECT1 (M23 (ACTION (SAY)) (OBJECT1 (Come here))))
    (OBJECT2 (M17 (ACTION (SAY)) (OBJECT1 (BILL))))))))

Now doing: DO-ONE
((M30 (ACTION (SNSEQUENCE))
  (OBJECT1 (M23 (ACTION (SAY)) (OBJECT1 (Come here))))
  (OBJECT2 (M17 (ACTION (SAY)) (OBJECT1 (BILL))))))

Chose to do the act
((M30 (ACTION (SNSEQUENCE))
  (OBJECT1 (M23 (ACTION (SAY)) (OBJECT1 (Come here))))
  (OBJECT2 (M17 (ACTION (SAY)) (OBJECT1 (BILL))))))

About to do
((M23 (ACTION (SAY)) (OBJECT1 (Come here))))

I wonder if the act
((M23 (ACTION (SAY)) (OBJECT1 (Come here))))
has any preconditions...

The act
((M23 (ACTION (SAY)) (OBJECT1 (Come here))))
has no preconditions:

Come here

About to do
((M17 (ACTION (SAY)) (OBJECT1 (BILL))))

I wonder if the act
((M17 (ACTION (SAY)) (OBJECT1 (BILL))))

has any preconditions...
The act
((M17 (ACTION (SAY)) (OBJECT1 (BILL))))

has no preconditions:

BILL

Now doing: DO-ALL
((M38 (ACTION (BELIEVE))
  (OBJECT1 (M25 (AGENT (BILL)) (LOCATION (HERE)) (STATE (IS))))))

Believe
(M25! (AGENT BILL) (LOCATION HERE) (STATE IS))
 CPU time : 2.07
```

- The current version of SNeRE will believe that all effects of an act are achieved, even though this may be a naive assumption.
- The current version of the primitive action function `do-all` is

  ```
  (define-primaction do-all (object1)
    (do.ns (act object1)
           (schedule-act act)))
  ```

  but the user could redefine it if she wanted to make a more intelligent decision about the order in which the acts should be performed.
- The current version of the primitive action function `do-one` is

  ```
  (define-primaction do-one (object1)
    (schedule-act
     (lisp-list-to-ns (if snip::*choose-randomly*
                         (nth (random (cardinality.ns object1))
                              (ns-to-lisp-list object1))
                       (first (ns-to-lisp-list object1))))))
  ```

  but the user could redefine it if she wanted to make a more intelligent decision about which act should be performed.

Chapter 5

Program Interface

5.1 Transformers

These functions convert Lisp objects to SNePS nodes, and vice versa.

`(apply-function-to-ns` *fn ns*`)`
Converts the node set *ns* to a list of lisp objects, applies the function *fn* to that list, then converts the result to a node set, and returns that.

`(lisp-list-to-ns` *list*`)`
Returns a set of nodes whose identifiers look like the printed representations of the objects in the list *list*.

`(ns-to-lisp-list` *ns*`)`
Returns a list of Lisp objects corresponding to the SNePS nodes in the node set *ns*.

`(node-to-lisp-object` *nde*`)`
Returns a Lisp object corresponding to the SNePS node *nde*. The Lisp object will be either a number or a symbol.

`(lisp-object-to-node` *obj*`)`
Returns a SNePS node whose identifier looks like *obj*.

5.2 With-SNePSUL Reader Macro

The with-snepsul reader macro is provided so that users can easily incorporate calls to SNePSUL commands within Lisp code.

`(#[`i`]! (`*snepsul-form*$_1$ ... *snepsul-form*$_n$`))` The form following `#[`i`]!` is taken to be a list of SNePSUL forms, each of which will be executed just as if it had been typed that way at the SNePS prompt, regardless of the package in which the `#[`i`]!` form is read. References to Lisp variables can be made via a ~ reader macro mechanism (similar to the comma within backquote syntax). Results of ~ expansions will be automatically interned into the SNePSUL package (i.e., any symbols that might be part of such a result), unless explicitly specified otherwise. All the special reader syntax available at the SNePS top-level is available, too.

The semantics of the ~ syntax is:

~ *s-expression*:
S-expression will be read with ordinary reader syntax and at execution time it will be evaluated and its value inserted into the SNePSUL expression. If the value is a symbol or a list containing symbols then these symbols will be interned into the SNePSUL package first.
Ex: `#!((describe ~'(m1 m2)))` will act like `(describe (m1 m2))`.

`~@` *s-expression*:
Just like ~ but the value of the s-expression has to be a list which will be spliced into the SNePSUL expression. Any symbols occuring as leaves in the list will be interned into the SNePSUL package first. Ex: `#!((describe ~@'(m1 m2)))` will act like `(describe m1 m2)`.

~~ *s-expression*:
Just like ~ but symbols in the value will not be interned into the SNePSUL package.

`~~@` *s-expression*:
Just like `~@` but symbols in the value will not be interned into the SNePSUL package.

CAUTION: The ~ syntax can only be used within SNePSUL forms, but not to denote multiple forms, e.g., while `#!(~com1 ~com2 ~com3)` is legal (as long as the runtime values of $\texttt{com}_i$ represent proper SNePSUL commands), `#!(~@commands)` is not!!

Supplying an optional digit argument can be used to select a specific evaluation function, or to suppress output:

arg	eval function	silent	syntax
no arg	topsneval	no	`#!(....)`
1	topsneval	no	`#1!(....)`
2	eval	no	`#2!(....)`
3	topsneval	yes	`#3!(....)`
4	eval	yes	`#4!(....)`

For example, `#4!((build relation node))` will use the function `eval` to evaluate the form (hence `build` can be used!!), and will suppress any output generated by the snepsul command.

5.2.1 Controlling the Evaluation of SNePSUL Forms Generated by #!

`(defvar *with-snepsul-eval-function* #'with-snepsul-standard-eval`
"The value of this variable has to be a function of two arguments, an *eval-function* and a *form* to which the function should be applied. Binding this variable to different functions can implement various different evaluation behaviors, such as normal evaluation, tracing, top-level-like echoing, evaluating and printing the result, etc., when the *form* gets evaluated inside `with-snepsul-eval`.")

The following evaluation functions are available:

`(with-snepsul-standard-eval` *function form*`)`
Standard function used by `with-snepsul-eval` to evaluate *form* with evaluation *function*.

`(with-snepsul-trace-eval` *function form*`)`
Does not actually evaluate *form*, only prints it for debugging purposes.

`(with-snepsul-toplevel-style-eval` *function form*`)`
Evaluates the SNePSUL *form* using *function* and returns the result. Additionally, prints the prompt, *form*, result and timing information just like the top-level SNePS loop does—good for monitoring the execution of the actual SNePSUL commands.

5.2.2 Example Use of #!

```
> (in-package 'user)
#<Package "USER" 79D15E>

> (defun myassert (relation nodes)
    (let ((base-node-var 'mybase))
      #!((define ~relation ~~relation myrel snip::test)
         (assert ~relation (~@nodes) snip::test #~base-node-var)
         (describe ~@(setq nodes (cdr nodes)))
         (assert ~~relation (~~@nodes) myrel *~base-node-var)
         (assert arg (build myrel hans ~relation franz)
                 myrel *mybase)
         (describe *nodes))))
MYASSERT

;; If the variable *with-snepsul-eval-function* is bound to the
;; function #'with-snepsul-trace-eval then the generated SNePSUL
;; expression will only be printed, but not actually executed:

> (let ((sneps:*with-snepsul-eval-function*
          #'sneps:with-snepsul-trace-eval))
    (myassert 'relrel '(hans franz otto)))
(SNEPS:DEFINE SNEPSUL::RELREL ;; Note "snepsulization" with single ~
              USER::RELREL    ;; Package preservation with double ~
              SNEPSUL::MYREL  ;; Unqualified symbols go into SNePSUL
              SNIP::TEST)     ;; Qualified symbols keep their package
(SNEPS:ASSERT SNEPSUL::RELREL
              (SNEPSUL::HANS SNEPSUL::FRANZ
                             SNEPSUL::OTTO)
              SNIP::TEST
              ;; had to replace the |'s with !'s here (comment problem)
              (SNEPS:!#! 'SNEPSUL::MYBASE))    ;; Combination of # and ~
(SNEPS::DESCRIBE SNEPSUL::FRANZ SNEPSUL::OTTO)
(SNEPS:ASSERT USER::RELREL (USER::FRANZ USER::OTTO)
              SNEPSUL::MYREL (SNEPS:* 'SNEPSUL::MYBASE))
(SNEPS:ASSERT SNEPS:ARG (SNEPS:BUILD SNEPSUL::MYREL SNEPSUL::HANS
                                     SNEPSUL::RELREL SNEPSUL::FRANZ)
              SNEPSUL::MYREL (SNEPS:* 'SNEPSUL::MYBASE))
(SNEPS::DESCRIBE (SNEPS:* 'SNEPS:NODES))

;; Now actually run it:
> (myassert 'relrel '(hans franz otto))
(FRANZ)
(OTTO)

(B1)
(FRANZ)    ;; user::franz
(FRANZ)    ;; snepsul::franz
(HANS)
(M1! (RELREL FRANZ HANS OTTO))
```

```
(M2! (RELREL FRANZ OTTO))
(M3 (MYREL HANS) (RELREL FRANZ))
(M4! (ARG (M3)))
(M5! (RELREL FRANZ HANS OTTO)
     (TEST B1))
(M6! (MYREL B1)
     (RELREL FRANZ OTTO))
(M7! (ARG (M3)) (MYREL B1))
(B1 FRANZ FRANZ HANS M1! M2! M3 M4! M5! M6! M7! OTTO OTTO)
SNEPS:DEFAULT-DEFAULTCT

;; Here's what the definition of myassert looks like:
> (ppdef 'myassert)
(LAMBDA (RELATION NODES)
  (BLOCK MYASSERT
    (LET ((BASE-NODE-VAR 'MYBASE))
      (PROGN ;; progn generated by #!
        (SNEPS::WITH-SNEPSUL-EVAL
            `(SNEPS:DEFINE ,(SNEPS::SNEPSULIZE RELATION) ,RELATION
              SNEPSUL::MYREL SNIP::TEST)
          #'SNEPS:TOPSNEVAL NIL)
        (SNEPS::WITH-SNEPSUL-EVAL
            `(SNEPS:ASSERT ,(SNEPS::SNEPSULIZE RELATION)
              (,@(SNEPS::SNEPSULIZE NODES)) SNIP::TEST
              (SNEPS:!#! ',(SNEPS::SNEPSULIZE BASE-NODE-VAR)))
          #'SNEPS:TOPSNEVAL NIL)
        (SNEPS::WITH-SNEPSUL-EVAL
            `(SNEPS::DESCRIBE
              ,@(SNEPS::SNEPSULIZE (SETQ NODES (CDR NODES))))
          #'SNEPS:TOPSNEVAL NIL)
        (SNEPS::WITH-SNEPSUL-EVAL
            `(SNEPS:ASSERT ,RELATION (,@NODES) SNEPSUL::MYREL
              (SNEPS:* ',(SNEPS::SNEPSULIZE BASE-NODE-VAR)))
          #'SNEPS:TOPSNEVAL NIL)
        (SNEPS::WITH-SNEPSUL-EVAL
            `(SNEPS:ASSERT SNEPS:ARG
              (SNEPS:BUILD SNEPSUL::MYREL SNEPSUL::HANS
               ,(SNEPS::SNEPSULIZE RELATION) SNEPSUL::FRANZ)
              SNEPSUL::MYREL (SNEPS:* 'SNEPSUL::MYBASE))
          #'SNEPS:TOPSNEVAL NIL)
        (SNEPS::WITH-SNEPSUL-EVAL
            '(SNEPS::DESCRIBE (SNEPS:* 'SNEPS:NODES))
          #'SNEPS:TOPSNEVAL NIL)))))
```

5.3 Defining New Commands

`(defsnepscom` *command* `([({`*arg*`}*)] [`*environments*`] [`*eval-args*`])` {*body-form*}* `)`

`defsnepscom` is a Lisp macro that defines SNePSUL commands. All standard SNePSUL commands such as `find`, `assert`, `deduce`, etc., are defined via `defsnepscom`. More importantly, `defsnepscom` is the only way to define commands which will be recognized as legal SNePSUL commands at the SNePS top level. The syntax of `defsnepscom` is very similar to that of a standard `defun` or `defmacro`.

command is a Lisp symbol which serves as the command name, e.g., `find`, `deduce`, `my-find`, `isa`, etc. *command* will get exported automatically from its home package and imported into the SNePSUL package, hence, even if the command was defined in a different package, it can be used at the SNePS top level without package qualifiers. The only catch is that if *command* is the name of a standard COMMON LISP function as in the case of `find` or `assert`, then that symbol has to be shadowed in its home package with the COMMON LISP function `shadow` before the command gets defined.

`({`*arg*`}*)` is an optional argument list in the standard COMMON LISP syntax. An actual call to the command has to be legal according to that argument list, otherwise an error will occur. The only difference to the standard `defun` style of specifying argument lists is an extra level of nesting as shown in the examples below.

The optional second argument *environments* defines the places in which the command can legally appear. An environment is basically a specification of a location in which a command can be used. For example, some commands can only be used at the top level, some commands can never be used at the top level but only inside some other command, some commands can only be used within `find` commands, etc. See Section 1.4 for more information on environments. *environments* can either be `:all` to define *command* as legal in all possible environments, or it can be a subset of `(top rs bns fns ons rearrange)` specified as a list, which will make it legal in the specified environments. These abbreviations indicate environments as specified in the following table.

`top`	The top level of SNePS 2
`rs`	A *relation-set* position embedded in a command
`bns`	A *node-set* position in `build`
`fns`	A *node-set* position in `find` or `findassert`
`ons`	A *node-set* position in any of the other commands
`rearrange`	The command is an infix or postfix command.

A third possibility, which is probably the one most commonly used, is to supply the name of an already existing command, in which case *command* will be legal in all environments in which the supplied command is legal. *environments* defaults to `(top)`. According to the specified *environments*, `defsnepscom` automatically updates the SNePSUL variables `commands`, `topcommands`, etc. (See Section 1.7.)

By default, commands defined with `defsnepscom` do not evaluate their arguments. If one wants command arguments to be evaluated before they get passed (similar to the behavior of standard functions defined with `defun`), one has to specify the optional third argument, *eval-args* as `t`.

*body-form*s are a sequence of body forms, possibly including a documentation string and declarations just as in a normal `defun`. The value/s of the last form will be returned.

Here are some examples:

Example 1:

```
* ^^   ;; escape to the Lisp level, since 'defnepscom' is not a SNePSUL command

--> (defsnepscom mylist ((first &optional second &rest others))
       (list first second others))
```

```
T
--> ^^ ;; back to the SNePS top level

 CPU time : 0.03

* (mylist apples oranges hans franz)  ;; let's try it out:

(APPLES ORANGES (HANS FRANZ))

 CPU time : 0.01
```

Note, that we did not have to quote `apples` and `oranges` in the example above, because *eval-args* was not specified as `t`.

Example 2:

```
* ^^

--> (defsnepscom isa ((who what) assert)
      "Asserts that WHO is a WHAT."
      #!((assert member ~who class ~what)))
T
--> ^^

 CPU time : 0.08

* (describe (isa hans student))

(M1! (CLASS STUDENT) (MEMBER HANS))

(M1!)

 CPU time : 0.02
```

The `isa` command defined above takes two arguments who and `what`, and it is legal in all places where the `assert` command is legal, because we specified `assert` as the value of *environments*. The body of the command has a documentation string just as a normal `defun`, and it uses the `#!` with-snepsul reader macro (see Section 5.2) to easily call the SNePSUL command `assert` in the body of the command definition.

Example 3:

```
* ^^

--> (defsnepscom lex-build ((word) (top bns) t)
      "Builds a node with a `lex' arc to a WORD node."
      #2!((build lex ~word)))
T
--> ^^

 CPU time : 0.25
```

```
* (lex-build (progn (format t "Word: ") (read)))
Word: Lucy

(M1)

 CPU time : 0.03
```

This last example command uses all the features: It has an argument list, it explicitly specifies two environments in which the command will be legal, and it evaluates its arguments which is the reason why we could call it with the little interactive input specification. Note, that this command `builds` (not `asserts`) a node, and that it will be available as a top-level command because we specified `top` as one of the environments. For good reasons, the standard `build` command is not a top-level command, hence, in this example we forced SNePS to do something which is normally not allowed.

By convention, every command that returns a node set should return a context as a second value which will be used to display the node set. Commands which use an application of the `#!` reader macro as their last body form will achieve this automatically. Otherwise, a form such as `(values nodes crntct)` has to be used as the last body form.

`defsnepscom` is available in all standard SNePS packages. Hence, it can normally always be used without a package qualifier. If it is used in a non-standard package it should be written as `sneps:defsnepscom`.

`(undefsnepscoms` $\{$*commands*$\}^*$`)`
Undefines all *commands*. The function definitions of the individual commands will not be removed, but the listed commands will not be available as SNePSUL commands anymore. For example, the following will undo the inappropriate definition of Example 3 above:

```
* (^ (undefsnepscoms lex-build))

(T)

 CPU time : 0.00

* (lex-build Lucy)

SNePS ERROR: Invalid top SNePSUL form: (LEX-BUILD LUCY)
Occurred in module TOP-EVALUATOR in function TOPSNEVAL

Do you want to debug it? n

*
```

Chapter 6

SNePSLOG

6.1 SNePSLOG Basics

SNePSLOG is a logic programming interface to SNePS. That is, almost everything that can be done interactively using SNePSUL can be done interactively using SNePSLOG, just with a syntax that looks more like traditional symbolic logic than SNePSUL does. Use of SNePSLOG rather than SNePSUL is recommended for the SNePS novice.

To enter SNePSLOG, load SNePS and evaluate

```
(snepslog)
```

To leave SNePSLOG, execute the SNePSLOG command

```
lisp
```

The default Common Lisp package for symbols read by the SNePSLOG reader is `snepslog`.

The full details of the SNePSLOG syntax is in §6.2. The semantics are given in §6.3.

6.2 SNePSLOG Syntax

The SNePSLOG syntax is described in Tables 6.1 and 6.2 using the Extended BNF notation. Object language terminal symbols are in `this font`. Grouping parentheses are (and). Alternatives are separated by the | character. Square brackets [and] surround optional material. The Kleene star, *, indicates zero or more repetitions. The Kleene plus, +, indicates one or more repetitions. Object language parentheses are (and). The object language comma is `,`. The object language underscore character is _. The symbols i, j, and k are non-terminal symbols representing integers. Material starting with a semicolon is a comment indicating a restriction on the syntax.

Note that a SNePSLOGsymbol may be any Lisp symbol, string, or number. If a SNePSLOGsymbol is a symbol, it is interned in the `snepslog` package. If a SNePSLOGsymbol is a string, it is coerced into a symbol whose symbol-name is the original string, and is interned in the `snepslog` package. If a SNePSLOGsymbol is a number, it is coerced into a symbol whose symbol-name is the Lisp printer representation of the original number, and is interned in the `snepslog` package. Two SNePSLOGsymbols are considered the same by SNePSLOG if and only if the Lisp being used considers the symbols constructed by this algorithm to be the same. When SNePSLOG prints a SNePSLOGsymbol, it does not print string-quotes nor escape characters.

Table 6.1: The Syntax of SNePSLOG Commands

<table>
<tr><td>command</td><td>::=</td><td>wffNameCommand | snepslogCommand | wffCommand</td></tr>
<tr><td>wffNameCommand</td><td>::=</td><td>wffName terminalPunctuation</td></tr>
<tr><td>wffCommand</td><td>::=</td><td>wff terminalPunctuation ; wff must not be an atomic symbol</td></tr>
<tr><td>snepslogCommand</td><td>::=</td><td>% SNePSULcommand | ^ LispForm | ^^</td></tr>
<tr><td></td><td></td><td>| activate wff [.]</td></tr>
<tr><td></td><td></td><td>| activate! wff [terminalPunctuation]</td></tr>
<tr><td></td><td></td><td>| add-to-context SNePSLOGsymbol termSet [.]</td></tr>
<tr><td></td><td></td><td>| ask wff [terminalPunctuation]</td></tr>
<tr><td></td><td></td><td>| askifnot wff [terminalPunctuation]</td></tr>
<tr><td></td><td></td><td>| askwh wff [terminalPunctuation]</td></tr>
<tr><td></td><td></td><td>| askwhnot wff [terminalPunctuation]</td></tr>
<tr><td></td><td></td><td>| clear-infer [.]</td></tr>
<tr><td></td><td></td><td>| clearkb [.]</td></tr>
<tr><td></td><td></td><td>| define-frame SNePSLOGsymbol LispList [LispString] [.]</td></tr>
<tr><td></td><td></td><td>| define-path SNePSRelation SNePSPath [.]</td></tr>
<tr><td></td><td></td><td>| demo [filePath | ? | i] [t | b | bv | a | av | n] [.]</td></tr>
<tr><td></td><td></td><td>| describe-context [SNePSLOGsymbol] [.]</td></tr>
<tr><td></td><td></td><td>| describe-terms [pTermSet] [.]</td></tr>
<tr><td></td><td></td><td>| expert [.]</td></tr>
<tr><td></td><td></td><td>| lisp [.]</td></tr>
<tr><td></td><td></td><td>| list-asserted-wffs [SNePSLOGsymbol] [.]</td></tr>
<tr><td></td><td></td><td>| list-contexts [.]</td></tr>
<tr><td></td><td></td><td>| list-terms [pTermSet] [.]</td></tr>
<tr><td></td><td></td><td>| list-wffs [.]</td></tr>
<tr><td></td><td></td><td>| load filePath [.]</td></tr>
<tr><td></td><td></td><td>| normal [.]</td></tr>
<tr><td></td><td></td><td>| perform atomicTerm [.]</td></tr>
<tr><td></td><td></td><td>| remove-from-context SNePSLOGsymbol pTermSet</td></tr>
<tr><td></td><td></td><td>| set-context SNePSLOGsymbol [pTermSet]</td></tr>
<tr><td></td><td></td><td>| set-default-context SNePSLOGsymbol</td></tr>
<tr><td></td><td></td><td>| set-mode-1 [.]</td></tr>
<tr><td></td><td></td><td>| set-mode-2 [.]</td></tr>
<tr><td></td><td></td><td>| set-mode-3 [t | nil] [.]</td></tr>
<tr><td></td><td></td><td>| show [pTermSet] [.]</td></tr>
<tr><td></td><td></td><td>| trace [SNePSLOGfunction]* [.]</td></tr>
<tr><td></td><td></td><td>| undefine-path SNePSRelation [.]</td></tr>
<tr><td></td><td></td><td>| unlabeled [.]</td></tr>
<tr><td></td><td></td><td>| untrace [SNePSLOGfunction]* [.]</td></tr>
<tr><td>SNePSLOGfunction</td><td>::=</td><td>inference | acting | translation | parsing</td></tr>
<tr><td></td><td></td><td>| LispSymbol</td></tr>
</table>

Table 6.2: The Syntax of SNePSLOG Wffs

<table>
<tr><td>wff</td><td>::=</td><td>infixedTerm | entailment | prefixedTerm</td></tr>
<tr><td>infixedTerm</td><td>::=</td><td>prefixedTerm [(and | or | <=>) prefixedTerm]+</td></tr>
<tr><td>entailment</td><td>::=</td><td>termSet (=> | v=> | &=> | i=>) termSet</td></tr>
<tr><td>pTermSet</td><td>::=</td><td>termSet ; but taken to denote all terms that match</td></tr>
<tr><td>termSet</td><td>::=</td><td>prefixedTerm | { termSequence }</td></tr>
<tr><td>termSequence</td><td>::=</td><td>prefixedTerm [, prefixedTerm]*</td></tr>
<tr><td>prefixedTerm</td><td>::=</td><td>negatedTerm | andorTerm | allTerm | nexistsTerm
| threshTerm | atomicTerm</td></tr>
<tr><td>negatedTerm</td><td>::=</td><td>~ atomicTerm</td></tr>
<tr><td>andorTerm</td><td>::=</td><td>andor (i, j)termSet ; $0 <= i <= j$</td></tr>
<tr><td>threshTerm</td><td>::=</td><td>thresh (i [, j])termSet ; $0 <= i <= j$</td></tr>
<tr><td>allTerm</td><td>::=</td><td>all (symbolSequence) (wff) ; wff must not be an atomic symbol</td></tr>
<tr><td>nexistsTerm</td><td>::=</td><td>nexists nexistsParameters (symbolSequence) (termSet : termSet)</td></tr>
<tr><td>nexistsParameters</td><td>::=</td><td>(i , j , k) | (_ , j , _) | (i , _ , k)</td></tr>
<tr><td>atomicTerm</td><td>::=</td><td>wffName | qvar | SNePSLOGsymbol
| withsome/allTerm ; in Mode 3 only
| (qvar | SNePSLOGsymbol) (termSetSequence)
| (wff)</td></tr>
<tr><td>withsome/allTerm</td><td>::=</td><td>(withsome | withall) (symbolSequence , termSet , termSet [, termSet])</td></tr>
<tr><td>termSetSequence</td><td>::=</td><td>termSet [, termSet]*</td></tr>
<tr><td>symbolSequence</td><td>::=</td><td>SNePSLOGsymbol [, SNePSLOGsymbol]*</td></tr>
<tr><td>wffName</td><td>::=</td><td>wffi ; wffi must already name a wff.</td></tr>
<tr><td>qvar</td><td>::=</td><td>? SNePSLOGsymbol</td></tr>
<tr><td>SNePSLOGsymbol</td><td>::=</td><td>wffi | Lispsymbol | Lispstring | Lispnumber</td></tr>
<tr><td>terminalPunctuation</td><td>::=</td><td>. | ! | ?? | ? [(i [j])]</td></tr>
</table>

6.3 SNePSLOG Semantics

6.3.1 Semantics of SNePSLOG Commands

Here we present a list of the SNePSLOG commands, with a description of what they do. See Table 6.1 for a concise list of the SNePSLOG commands and their syntax.

- `%` *SNePSULcommand*
  Executes the *SNePSULcommand*, and prints the result. The default Common Lisp package for symbols in the *SNePSULcommand* is `snepsul`.

- `^` *LispForm*
  Evaluates the *LispForm*, and prints the result.

- `^^`
  Enters a Lisp read-eval-print loop. To leave the loop, type `end` or `^^`.

- `activate` wff
  Performs forward inference on all asserted propositions that dominate the wff.

- `activate!` wff
  Asserts wff, and performs forward inference on it, and on all asserted propositions that dominate it.

- `add-to-context` SNePSLOGsymbol termSet
  Adds the wffs in termSet as hypotheses in the context named SNePSLOGsymbol.

- `ask` wff
  Performs backward inference on wff and prints the inferred positive instances of it.

- `askifnot` wff
  Performs backward inference on wff, and prints the inferred negative instances of it.

- `askwh` wff
  Performs backward inference on wff, and prints a list of substitutions, which, when applied to wff yield asserted wffs.

- `askwhnot` wff
  Performs backward inference on wff, and prints a list of substitutions, which, when applied to the negation of wff yield asserted wffs.

- `clear-infer`
  Deletes any information placed in the "active connection graph" version of the network.

- `clearkb`
  Empties the knowledge base.

- `define-frame` SNePSLOGsymbol $(\texttt{rel}_0\ \texttt{rel}_1\ \ldots\texttt{rel}_n)$ [*LispString*]
  If SNePSLOG is in Mode 3, this declares that every SNePSLOG term of the form $\texttt{P}(\texttt{x}_1, \ldots, \texttt{x}_n)$ is to be represented by a node of the form $\{\langle \texttt{rel}_0, \{\texttt{P}\}\rangle, \langle \texttt{rel}_1, \texttt{x}_1\rangle, \ldots, \langle \texttt{rel}_n, \texttt{x}_n\rangle\}$. If $\texttt{rel}_0$ is `nil`, then the node will have no arc pointing to `P`. One function symbol may be associated with at most one frame, and it must be possible to uniquely determine the form of the SNePSLOG term from each frame. The *LispString*, if present, must contain the substring $\texttt{"[rel}_i\texttt{]"}$ for each non-`null` $\texttt{rel}_i$, and will be used by `describe-terms` to construct a gloss of the term.

- `define-path` *SNePSRelation SNePSPath*
  Defines a path-based inference rule. See §2.5.2.

- `demo` [*filePath* | `?` | i] [`t` | `b` | `bv` | `a` | `av` | `n`]
  The input is taken from the file specified by *filePath*, until the end of file is reached. The input is then reset to the previous input stream. Notice that embebbed demos are allowed. If *filePath* `?` or omitted, a menu of possible demonstrations will be printed, and you will be able to choose one of them. If *filePath* is an integer, and the menu lists at least that many demonstrations, the one with that number will be run. For the meaning of the various pause controls (`t`, `b`, `bv`, `a`, `av`, and `n`), see page 8.

- `describe-context` [SNePSLOGsymbol]
  Lists the details of the context named SNePSLOGsymbol. If SNePSLOGsymbol is omitted, describes the default context.

- `describe-terms` [pTermSet]
  This is only useful in Mode 3. If pTermSet is omitted, all the closed functional terms in the knowledge base are described. If pTermSet is included, all, but only, those closed functional terms that match the term patterns in pTermSet are described. The description of an individual constant is itself, The description of a functional term is formed from the *LispString* included when the frame is defined for the term's function symbol. The description is the *LispString* with every instance of `"[rel`$_i$`]"` replaced by the description of the filler(s) of the `"[rel`$_i$`]"` slot. For example, after the frame definitions:

  ```
  define-frame mother (nil motherof) "the mother of [motherof]"
  define-frame female (property object) "[object] is [property]"
  ```

  the description of `female(mother(Betty))` will be

  ```
  The mother of Betty is female.
  ```

- `expert`
  Turns on the expert mode, in which the wffName of listed terms is shown, as in normal mode (*cf*), and, in addition, when a SNePSLOG proposition is printed, its support set is shown.

- `lisp`
  Leaves the SNePSLOG loop, and returns to the Lisp listener.

- `list-asserted-wffs` [SNePSLOGsymbol]
  Lists all propositions that are asserted in the context named SNePSLOGsymbol. If no argument is given, the default context is used.

- `list-contexts`
  Lists the names of all the contexts that have been defined since the last time the knowledge base was cleared.

- `list-terms` [pTermSet]
  If pTermSet is omitted, all the closed functional terms in the knowledge base are printed. If pTermSet is included, all, but only, those closed functional terms that match the term patterns in pTermSet are printed.

- `list-wffs`
  Lists all propositions that are asserted in any context. That is, all propositions that have been asserted as hypotheses or have been derived, regardless of which context they are in.

- `load` *filePath*
  Executes the contents of the specified file as a series of SNePSLOG commands, without doing any printing. All assertions specified by the file are done as one batch at the end of the loading process, and they are all asserted into the current context.

- `normal`
  Returns to the (default) normal mode. In the normal mode, each term is printed using its SNePSLOG representation, and preceded by its wff name, which is `wff`n, for some integer n. Note that `wff`n is the same node referred to in SNePSUL as `m`n. The wff name is followed by an exclamation mark (`!`) if and only if the term is a proposition asserted in the current context.

- `perform` atomicTerm
  If SNePSLOG is in Mode 3, the act denoted by atomicTerm (either an individual constant or ground functional term) is performed. (*See* §6.5.)

- `remove-from-context` SNePSLOGsymbol pTermSet
  Removes the terms that match the patterns in pTermSet from the context named SNePSLOGsymbol.

- `set-context` SNePSLOGsymbol [pTermSet]
  Defines a context named SNePSLOGsymbol, and sets its initial set of hypotheses to be the terms that match the patterns in pTermSet. Note that pTermSet could be empty or omitted, in which case, the new context is initialized with an empty set of hypotheses.

- `set-default-context` SNePSLOGsymbol
  Makes the context named SNePSLOGsymbol the current, default, context.

- `set-mode-1`
  The knowledge base is cleared, and SNePSLOG is put into Mode 1 (the default mode). In this mode, every term of the form $\mathtt{P}(\mathtt{x}_1, \ldots, \mathtt{x}_n)$ is represented by a node of the form

  $$\{\langle \mathtt{r}, \{\mathtt{P}\}\rangle, \langle \mathtt{a1}, \{\mathtt{x}_1\}\rangle, \ldots, \langle \mathtt{a}_n, \{\mathtt{x}_n\}\rangle\}$$

  Mode 1 is the mode to use when there is no specific reason to use Mode 2 or Mode 3. It requires less set-up effort for the user than Mode 3, because no frames need to be defined. Mode 1 is the only mode that allows a variable (`qvar`) to be used as the function symbol of a query or term pattern. Inference in Mode 1 is less efficient than in the other modes because more terms match any given pattern.

- `set-mode-2`
  The knowledge base is cleared, and SNePSLOG is put into Mode 2. In this mode, every proposition of the form $\mathtt{P}(\mathtt{x1}, \ldots, \mathtt{xn})$ is represented by a node of the form

  ```
  {<| rel P|, {P}>, <|rel-arg#P1|, {x1}>, ..., <|rel-arg#Pn|, {xn}>}
  ```

  Mode 2 is a compromise between Modes 1 and 3. It requires no more user set-up effort than Mode 1 does; inference is more efficient than in Mode 1, because fewer terms match any given pattern; but, unlike in Mode 1, variables may not be used as function symbols in queries.

- `set-mode-3`
  The knowledge base is cleared, and SNePSLOG is put into Mode 3. In this mode, the user must specify how terms are represented by using `define-frame`. Inference can be more efficient if the user uses a wide variety of frames. This mode facilitates path-based reasoning, and is required if the SNePSLOG version of SNeRE (§6.5) is to be used. In this mode, SNePSLOG syntax may be used to build almost any SNePS network that can be built using SNePSUL.

- `show` [pTermSet]
  Displays the knowledge base in graphical form as a network, If pTermSet is omitted, all the closed functional terms in the knowledge base are printed. If pTermSet is included, all, but only, those terms that match the term patterns in pTermSet are printed. Depending on the SNePS installer's choice, `show` either uses `dot` or `JUNG` and `JIMI`. `dot` produces a static figure. `JUNG`/`JIMI` produces a graph that

can be manipulated by hand. Neither `dot`, `JUNG`, nor `JIMI` are part of the SNePS distribution. `dot` is part of the Graphviz package which can be downloaded from `http://graphviz.org/`). `JUNG` and it's associated packages, `Xerxes`, `Colt`, and `Jakarta Common Collections`, can be downloaded from `http://jung.sourceforge.net/`. `JIMI` can be downloaded from `http://java.sun.com/products/jimi/`. The SNePS installer may install either the `dot` version or the JUNG/JIMI version, both, or neither. If both are installed, the user can dynamically pick the version to be used by setting the global variable ¡code¿cl-user:*use-gui-show*¡/code¿/indexctcl-user:*use-gui-show* to ¡code¿t¡/code¿ for the JUNG/JIMI version, or to ¡code¿nil¡/code¿ for the ¡code¿dot¡/code¿ version.

- `trace` [SNePSLOGfunction]*
  If SNePSLOGfunction is
  `inference`: Inference tracing is turned on.
  `acting`: Tracing of acting is turned on.
  `translation`: The translation of each SNePSLOG command into SNePSUL is shown.
  `parsing`: The parsing of each SNePSLOG command is traced.
  The name of any Lisp function: That function is traced.

- `undefine-path` *SNePSRelation*
  Deletes the path-based inference rule from the *SNePSRelation.*

- `unlabeled`
  Turns on unlabeled mode, in which, when a term is printed, neither its wffName (see normal mode), nor its support set (see expert mode) is printed.

- `untrace` [SNePSLOGfunction]*
  If SNePSLOGfunction is
  `inference`: Inference tracing is turned off.
  `acting`: Tracing of acting is turned off.
  `translation`: The translation of each SNePSLOG command into SNePSUL is not shown.
  `parsing`: The parsing of each SNePSLOG command is not traced.
  The name of any Lisp function: That function is not traced.

6.3.2 Semantics of wffNameCommands

A wffNameCommand is a wffName followed by one of the terminalPunctuation marks: ".", "!", "??", or "?". A wffName is a symbol made up of "`wff`" followed by an integer. WffNames are assigned to terms by SNePSLOG. If the wffName `wff`*i*, for some *i*, has already been assigned to a term, then using `wff`*i* in a wffNameCommand or in any SNePSLOG expression is equivalent to using the term that it names. Actually, it is even better. Whenever SNePSLOG parses a new variable (not just a new occurrence of a variable), it creates a new variable node (*see* S1.5), even if the variable has the same name as a previous one. So a variable-containing term in one wffCommand will never be the same, identical, term as one included in a previous wffCommand. However, the wffName assigned to a term will always refer to that term. Note the difference between the two techniques here:

```
: all(x)(Robin(x) => Bird(x)).
  wff1!:  all(x)(Robin(x) => Bird(x))

: Source(all(x)(Robin(x) => Bird(x)), "World Book").
  wff3!:  Source(all(x)(Robin(x) => Bird(x)),World Book)
```

```
: list-terms
  wff1!:  all(x)(Robin(x) => Bird(x))
  wff2:   all(x)(Robin(x) => Bird(x))
  wff3!:  Source(all(x)(Robin(x) => Bird(x)),World Book)

: clearkb
Knowledge Base Cleared

: all(x)(Robin(x) => Bird(x)).
  wff1!:  all(x)(Robin(x) => Bird(x))

: Source(wff1, "World Book").
  wff2!:  Source(all(x)(Robin(x) => Bird(x)),World Book)

: list-terms
  wff1!:  all(x)(Robin(x) => Bird(x))
  wff2!:  Source(all(x)(Robin(x) => Bird(x)),World Book)
```

If a wffName `wff`*i* is used before having been assigned to a term, it will be interpreted as being an individual constant. It will continue to be that individual constant **until** SNePSLOG assigns it to a term. After that, it will always refer to its assigned term, and no longer be recognized as the individual constant.

An assigned wffName in a wffNameCommand refers to its assigned term, and the meaning of the wffNameCommand depends on the terminalPunctuation in exactly the same way as for wffCommands (*see* §6.3.3).

6.3.3 Semantics of wffCommands

A wffCommand is a wff followed by one of the terminalPunctuation marks: ".", "!", "??", or "?". The effect of the wffCommand depends on the terminalPunctuation as follows:

`.` The wff is asserted into the knowledge base as an hypothesis of the current context, and it is printed.

`!` The wff is asserted into the knowledge base as an hypothesis of the current context, and forward inference is done on it. It and all wffs newly asserted as a result of this wffCommand are printed.

`??` If the wff is asserted in the knowledge base, it is printed; otherwise nothing is printed. If qvars (*see* Table 6.2) occur in the wff, they are taken as free variables, and all ground instances that are asserted are printed.

`?` [`(` *i* [*j*] `)`] Backward inference is done on the wff. If qvars (*see* Table 6.2) occur in the wff, they are taken as free variables. If either instances of the wff or of its negation are inferred (or already asserted), those wffs are printed. If no instances of it or its negation are asserted or inferred nothing is printed. If `(i)` is included after the question mark, backward inference stops as soon as at least `i` instances of the wff or its negation are inferred. If `(i j)` is included after the question mark, backward inference stops as soon as at least `i` positive instances and `j` negative instances of the wff are inferred.

Forward and backward inference create and use an active connection graph (acg). The acg prevents infinite recursion when recursive rules are used, and answers some queries without further inference. It also focusses the interaction on the current reasoning problem. However, occasionally it causes inferences not to be made even though the knowledge base contains enough information to make them. In that latter case, it may help to perform the `clear-infer` SNePSLOG Command, and then try again.

6.3.4 Semantics of SNePSLOG Wffs

In this section, we will mainly present the intended semantics of SNePSLOG wffs. We realize, however, that any particular user might have different semantics in mind. If SNePS is useful for that user under those semantics that's fine with us!

We intend a SNePS knowledge base to represent the mental entities conceived of by some individual agent. Some of those entities will be propositions, and some propositions will be asserted, that is, believed by the agent.

A SNePS knowledge base has one or more contexts. A context has a set of hypotheses, which are propositions that were introduced to the system (agent) without justification—typically by the user using a wff-Command terminated by "`.`" or "`!`". At any time, one context is the current context. The agent believes all the propositions that are hypotheses in the current context, as well as all propositions that have been derived from those hypotheses. The set of hypotheses and derived propositions that are currently believed is called the current belief space. Every currently believed proposition is also called "asserted", and when its wffName is printed, it is terminated by an "`!`".

Every well-formed expression of SNePSLOG is a term of the language. Expressions that look to the logically-trained user like formulas or sentences are actually proposition-valued terms (even though SNePSLOG uses `wff`*`i`* for its "wffName"). The significance of this is that, because terms may be arguments of terms, and may have variables ranging over them, without leaving first-order logic, metapropositions (propositions about propositions) are allowed in SNePSLOG.

Given that background, we will now discuss the intended semantics of SNePSLOG expressions.

An **individual constant**, expressed in SNePSLOG as a Lisp symbol, string, or number, denotes a mental entity. The unique names assumption holds, meaning that no two mental entities may be considered to be entirely equal.

A **function symbol**, also expressed in SNePSLOG as a Lisp symbol, string, or number, denotes a conceptualized function or relation in the domain. (Recall that what is considered a truth-functional relation in standard logic is a proposition-valued function in SNePS.) Function symbols do not have fixed arity in SNePSLOG. In fact `p(a,b,c)` implies `p(a,b)` by reduction inference. The user must be careful to always give a function symbol the same number of arguments if this behavior is not wanted.

A **functional term**, consisting of a function symbol and its arguments, denotes a mental entity, possibly a proposition, in the domain, namely that mental entity that results from applying the denotation of the functional term to the denotations of the arguments. No two functional terms that are syntactically different denote the same mental entity. This is called the Uniqueness Principle.

Functional terms with **sets as arguments** takes the elements of the sets conjunctively. For example, `motherOf({John, Jane, Tom, Betty})` is the mother of John, Jane, Tom, and Betty. Also `Isa({Rover, Fido}, {dog, pet})` is the proposition that Rover and Fido are both dogs and pets.

`andor(`*`i`*`,`*`j`*`){P1, ..., Pn}` denotes the proposition that at least *`i`* and at most *`j`* of `P1, ..., Pn` are true.

`P1 and ...and Pn` is an abbreviation of `andor(`*`n`*`,`*`n`*`){P1, ..., Pn}`.

`P1 or ...or Pn` is an abbreviation of `andor(`*`1`*`,`*`n`*`){P1, ..., Pn}`.

`thresh(`*`i`*`, `*`j`*`){P1, ..., Pn}` denotes the proposition that either fewer than *`i`* or more than *`j`* of `P1, ..., Pn` are true.

`thresh(`*`i`*`){P1, ..., Pn}` is an abbreviation of `thresh(`*`i`*`, `*`n-1`*`){P1, ..., Pn}`.

`P1 <=> ...<=> Pn` is an abbreviation of `thresh(`*`1`*`,`*`n-1`*`){P1, ..., Pn}`.

`{P1, ..., Pn} `*`i`*`=> {Q1, ..., Qm}` denotes the proposition that if any *`i`* of P1, . . . , Pn are true, than so are Q1, . . . , and Qm.

`{P1, ..., Pn} => {Q1, ..., Qm}` is an abbreviation of `{P1, ..., Pn} 1=> {Q1, ..., Qm}`.

`{P1, ..., Pn} v=> {Q1, ..., Qm}` is an abbreviation of `{P1, ..., Pn} 1=> {Q1, ..., Qm}`.

{P1, ..., Pn} &=> {Q1, ..., Qm} is an abbreviation of {P1, ..., Pn} *n*=> {Q1, ..., Qm}.

all(x1, ...xn)(P(x1,...,xn)) denotes the proposition that every ground instance of P(x1, ..., xn) that obeys the Unique Variable Binding Rule (UVBR) is true. A ground instance of P(x1,...,xn) obeys UVBR if no term already in P(x1,...,xn) substitutes for any of the variables x1,..., xn, and if no one term substitutes for more than one variable.

nexists(*i*,*j*,*k*)(x1, ..., xn)({P1, ..., Pnn} : {Q1, ...Qmm} denotes the proposition that there are k substitution instances (obeying UVBR) of x1, ..., xn that satisfy P1, ..., Pnn, and of them, at least *i* and at most *j* also satisfy Q1, ...Qmm.

nexists(*i*,_,*k*)(x1, ..., xn)({P1, ..., Pnn} : {Q1, ...Qmm} abbreviates nexists(*i*, *k*, *k*)(x1, ..., xn)({P1, ..., Pnn} : {Q1, ...Qmm} **nexists(_,*j*,_)(x1, ..., xn)({P1, ..., Pnn} : {Q1, ...Qmm}** denotes the proposition that there are at most k substitution instances (obeying UVBR) of x1, ..., xn that satisfy P1, ..., Pnn and also Q1, ...Qmm.

In all cases "... are true" does not mean the same as "... are believed". Truth is from the point of view of the agent. For example, if the agent believes p() => q() (that is, the wff p() => q() is asserted in the current belief space), and also believes p(), it still might not believe q() if the "rule" p() => q() hasn't "fired", but when the rule p() => q() does fire, the agent will believe q(). (That is, when the agent "reasons" using p() => q(), it will conclude q().)

6.4 SNIP in SNePSLOG

6.4.1 Rules of Inference

The implemented rules of inference of SNePSLOG are listed in this section. Not all logically possible operations have been implemented. Each connective and quantifier could have introduction rules and elimination rules. Each of those could operate while forward chaining, while backward chaining, and during bi-directional inference. There are two aspects to bi-directional inference:

1. If a proposition forward chains into one antecedent of a rule, backward chaining is performed on the other antecedents, and first, if necessary, on the rule itself. This backward inference triggered by forward inference operates whenever forward inference operates.

2. If backward inference creates a subgoal that is not satisfied, the subgoal remains as an active process until the next time clear-infer is issued. If the subgoal is later matched during forward inference, the backward inference that created it is resumed. Forward inference that causes one of these backward-inference processes to be resumed is called "forward-in-backward chaining" below. Forward-in-backward chaining operates for whatever rules of inference normal forward chaining operates, but sometimes it is operational even though normal forward chaining isn't.

Reduction inference and path-based inference are used when open terms are matched during forward and backward chaining. The Unique Variable Binding Rule is also enforced during matching. According to this rule, two different variables in one wff cannot be instantiated by different terms, and no variable in a wff can be instantiated by another term in its wff.

Whenever a set of wffs is indicated in this section, the wffs are listed in a convenient order. For example, in the andor(*n*,*n*)-Elimination rule, any wff can be inferred, not just the first one listed.

Reduction Inference: If *t*, *t1*, ..., *tn* are terms, α and β are sets of terms, $\beta \subset \alpha$, and $t \in \alpha$, then

1. *P*(*t1*, ..., α, ..., *tn*) $\vdash$ *P*(*t1*, ..., β, ..., *tn*)
2. *P*(*t1*, ..., α, ..., *tn*) $\vdash$ *P*(*t1*, ..., *t*, ..., *tn*)

Path-Based Inference: If

1. `define-frame` *P*`(`*r r1 ...ri ...rn*`)` has been done,
2. `define-path` *ri* ρ has been done,
3. *P*`(`*a1*`, ...,` *ai*`, ...,` *an*`)` is asserted,
4. and the path ρ exists in the network from *ai* to *b*,

then *P*`(`*a1*`, ...,` $\{$*ai*`,`*b*$\}$`, ...,` *an*`)` can be derived.

`andor(n,n)`-Elimination: `andor(`*n*`,`*n*`)`$\{$*P1*`, ...,` *Pn*$\}\vdash$ *P1* operates during both forward and backward chaining.

`andor(n,n)`-Introduction: *P1*`, ...,` *Pn* $\vdash$ `andor(`*n*`,`*n*`)`$\{$*P1*`, ...,` *Pn*$\}$ operates during backward chaining and forward-in-backward chaining.

`andor(0,0)`-Elimination: `andor(0,0)`$\{$*P1*`, ...,` *Pn*$\}$ $\vdash$ `~`*P1* operates during both forward and backward chaining.

`andor(0,0)`-Introduction:

1. `~`*P1*, ..., `~`*Pn* $\vdash$ `andor(`*0*`,`*0*`)`$\{$*P1*`, ...,` *Pn*$\}$ operates during backward chaining and forward-in-backward chaining.
2. If *P* is asserted with the origin set α, and `~`*P* is asserted with the origin set β, and *H* is in $\alpha \cup \beta$ and is chosen as the culprit of the contradiction, then `~`*H* is derived in the origin set $(\alpha \cup \beta) \setminus \{H\}$.

`andor(i,j)` (*i*$< n$, *j*> 0) -Elimination:

1. `andor(`*i*`,`*j*`)`$\{$*P1*`, ...,` *Pn*$\}$, *P1*, ..., *Pj* $\vdash$ `~`*Pn*, for $0 \leq i \leq j < n$, $0 < j < n$, operates during both forward and backward chaining.
2. `andor(`*i*`,`*j*`)`$\{P_1$`, ...,` $P_n\}$, `~`$P_1, \ldots,$ `~`$P_{n-i} \vdash P_n$, for $0 < i < n$, $i \leq j \leq n$, operates during both forward and backward chaining.

`thresh`-Elimination:

1. `thresh(`*i*`,`*j*`)`$\{P_1$`, ...,` $P_n\}$, $P_1, \ldots, P_i$, `~`$P_{i+1}, \ldots,$ `~`$P_{i+n-j-1} \vdash P_n$, operates during both forward and backward chaining.
2. `thresh(`*i*`,`*j*`)`$\{P_1$`, ...,` $P_n\}$, $P_1, \ldots, P_{i-1}$, `~`$P_i, \ldots,$ `~`$P_{i+n-j-1} \vdash$ `~`P_n, operates during both forward and backward chaining.

***i*`=>`-Elimination:** $\{$*P1*`, ...,` *Pn*$\}$ *i*`=>` $\{$*Q1*`, ...,` *Qm*$\}$`,` *P1*`, ...,` *Pi* $\vdash$ *Q1* operates during both forward and backward chaining.

`v=>`-Introduction: If *P1* $\vdash$ $\{$*Q1*`, ...,` *Qm*$\}$, then $\vdash$ $\{$*P1*`, ...,` *Pn*$\}$ `v=>` $\{$*Q1*`, ...,` *Qm*$\}$, operates during backward chaining.

`&=>`-Introduction: If *P1*`, ...,` *Pn* $\vdash$ *Q1*, and ..., and *P1*`, ...,` *Pn* $\vdash$ *Qm* such that *Q1*, and, ..., and *Qm* have an origin set in common, then $\vdash$ $\{$*P1*`, ...,` *Pn*$\}$ `&=>` $\{$*Q1*`, ...,` *Qm*$\}$, operates during backward chaining.

nexists-Elimination:

1. $\texttt{nexists}(i,j,k)(\vec{x})(\{P_1(\vec{x}),\ \ldots,\ P_n(\vec{x})\}:Q(\vec{x}))$,
   $P_1(\vec{a}_1), \ldots, P_n(\vec{a}_1), Q(\vec{a}_1)$,
   $\ldots$,
   $P_1(\vec{a}_j), \ldots, P_n(\vec{a}_j), Q(\vec{a}_j)$,
   $P_1(\vec{a}_{j+1}), \ldots, P_n(\vec{a}_{j+1}) \quad \vdash \quad \tilde{}Q(\vec{a}_{j+1})$
   operates during both forward and backward chaining.
2. $\texttt{nexists}(i,j,k)(\vec{x})(\{P_1(\vec{x}),\ \ldots,\ P_n(\vec{x})\}:Q(\vec{x}))$,
   $P_1(\vec{a}_1), \ldots, P_n(\vec{a}_1), \tilde{}Q(\vec{a}_1)$,
   $\ldots$,
   $P_1(\vec{a}_{k-i}), \ldots, P_n(\vec{a}_{k-i}), \tilde{}Q(\vec{a}_{k-}i)$,
   $P_1(\vec{a}_{k-i+1}), \ldots, P_n(\vec{a}_{k-i+1}) \quad \vdash \quad Q(\vec{a}_{k-i+1})$
   operates during both forward and backward chaining.

all-Elimination:

1. $\texttt{all}(\vec{x})(\texttt{andor}(i,j)\{P1(\vec{x}),\ \ldots,\ Pn(\vec{x})\}), P1(\vec{a}), \ldots, Pj(\vec{a}) \vdash \tilde{}Pn(\vec{a})$, for $0 \leq i \leq j < n$, $0 < j < n$
2. $\texttt{all}(\vec{x})(\texttt{andor}(i,j)\{P_1(\vec{x}),\ \ldots,\ P_n(\vec{x})\}), \tilde{}P_1(\vec{a}), \ldots, \tilde{}P_{n-i}(\vec{a}) \vdash P_n(\vec{a})$, for $0 < i < n$, $i \leq j \leq n$
3. $\texttt{all}(\vec{x})(\texttt{thresh}(i,j)\{P_1(\vec{x}),\ \ldots,\ P_n(\vec{x})\}), P_1(\vec{a}), \ldots, P_i(\vec{a}), \tilde{}P_{i+1}(\vec{a}), \ldots, \tilde{}P_{i+n-j-1}(\vec{a})$ $\vdash P_n(\vec{a})$
4. $\texttt{all}(\vec{x})(\texttt{thresh}(i,j)\{P_1(\vec{x}),\ \ldots,\ P_n(\vec{x})\}), P_1(\vec{a}), \ldots, P_{i-1}(\vec{a}), \tilde{}P_i(\vec{a}), \ldots, \tilde{}P_{i+n-j-1}(\vec{a})$ $\vdash \tilde{}P_n(\vec{a})$
5. $\texttt{all}(\vec{x})(\{P1(\vec{x}),\ \ldots,\ Pn(\vec{x})\}\ i\texttt{=>}\ \{Q1(\vec{x}),\ \ldots,\ Qm(\vec{x})\}),\ P1(\vec{a}),\ \ldots,\ Pi(\vec{a})$ $\vdash Q1(\vec{a})$

operate during both forward and backward chaining.

6.4.2 Recursion

Recursive rules such as

```
all(x,y,z)(ancestor(x,y), ancestor(y,z) &=> ancestor(x,z))
```

may be used without causing an infinite loop.

Infinite loops caused by backward chaining on rules such as

```
all(x)(duck(motherOf(x)) => duck(x))
```

or by forward chaining on rules such as

```
all(x)(number(x) => number(successorOf(x)))
```

are terminated under the control of the global parameters `*depthCutoffBack*` and `*depthCutoffForward*`, respectively. If a subgoal is generated during backward chaining whose depth, in terms of parenthesis nesting, exceeds `*depthCutoffBack*`, it is not pursued. Also, if a result is generated during forward chaining whose depth, in terms of parenthesis nesting, exceeds `*depthCutoffForward*`, it is not pursued. `*depthCutoffBack*` and `*depthCutoffForward*` are each set by default to 10, and can be changed independently via `setf`.

6.5 SNeRE in SNePSLOG

6.5.1 SNePSLOG Versions of SNeRE Constructs

With the introduction of Mode 3 in SNePSLOG (*see* p. 58), almost every structure that can be built *via* SNePSUL can be build *via* SNePSLOG. (For some restrictions, *see* §6.5.2.) Therefore, SNeRE agents can be defined and operated *via* SNePSLOG. In fact, the examples of Chapter 4 are available as a SNePSLOG demo in the distributed version of SNePS.

Each SNeRE caseframe documented in Chapter 4 has a SNeRE version predefined in Mode 3. These are listed below.

Policies: Policies are "propositions" that must be asserted in order to operate as described.

> **`ifdo(p, a)`:** If SNIP backchains into `p`, perform `a`.
>
> **`whendo(p, a)`:** If SNIP forward chains into `p`, perform `a`, and then disbelieve the `whendo`.
>
> **`wheneverdo(p, a)`:** If SNIP forward chains into `p`, perform `a`.

Mental Acts:

> **`believe(p)`:** First, the following special cases of belief revision are performed:
>
> - If `andor(0,0)`{*p*`, ...`} is believed, it is `disbelieved`.
> - If `andor(`*i*`,1)`{*p*`, `*q*`, ...`} is believed and *q* is believed, then *q* is `disbelieved`.
>
> Then `p` is asserted.
>
> **`disbelieve(p)`:** The proposition `p`, which must be a hypothesis, is unasserted.

Control Acts:

> **`achieve(p)`:** If the proposition `p` is asserted, do nothing. Otherwise, use `deduce` to infer plans for bringing about the proposition `p`, and then `do-one` of them.
>
> **`do-all(`{`a1, ..., an`}`)`:** Perform all the acts `a1, ..., an` in a nondeterministic order.
>
> **`do-one(`{`a1, ..., an`}`)`:** Nondeterministically choose one of the acts `a1, ..., an`, and perform it.
>
> **`snif(`{`if(p1,a1), ..., if(pn,an)[, else(da)]`}`)`:** Using `deduce` determine which of the `pi` hold. If any do, nondeterministically choose one of them, say `pj`, and perform `aj`. If none of the `pi` can be inferred, and if `else(da)` is included, perform `da`. Otherwise, do nothing.
>
> **`sniterate(`{`if(p1,a1), ..., if(pn,an)[, else(da)]`}`)`:** Using `deduce` determine which of the `pi` hold. If any do, nondeterministically choose one of them, say `pj`, perform `aj`, and then perform the entire `sniterate` again. If none of the `pi` can be inferred, and if `else(da)` is included, perform `da`. Otherwise, do nothing.
>
> **`snsequence(a1, a2)`:** Perform `a1`, and then perform `a2`.
>
> **`withall(?x, p(?x), a(?x), da)`:** Using `deduce`, determine which, if any, entities satisfy the open proposition `p(?x)`. If any do, say `e1, ..., en`, perform `a(ei)` on each of them in a nondeterministic order. If no entity satisfies `p(?x)`, perform `da`. Note: the question mark must appear to identify the open variable, and the default act `da` must be included.
>
> **`withsome(?x, p(?x), a(?x), da)`:** Using `deduce`, determine which, if any, entities satisfy the open proposition `p(?x)`. If any do, nondeterministically choose one, say `e`, and perform `a(e)`. If no entity satisfies `p(?x)`, perform `da`. Note: the question mark must appear to identify the open variable, and the default act `da` must be included.

Propositions about Acts:

ActPlan(a1, a2): The way to perform the act a1 is to perform the act a2. Typically, a1 will be a simple, but non-primitive act, and a2 will be an act structured using the control acts listed above.

Effect(a, p): The effect of performing the act a is that the proposition p will hold. The SNeRE executive described in §4.6 will automatically cause p to be asserted after a is performed.

GoalPlan(p, a): The act a is a plan for bringing about the proposition p. GoalPlan assertions are inferred by achieve(p) to find plans for achieving p.

Precondition(a, p): In order to be able to perform the act a, the proposition p must hold. It is assumed that the SNeRE agent is able to achieve(p). Before the SNeRE executive described in §4.6 performs any act which has inferrable preconditions, it will attempt to achieve all the preconditions.

6.5.2 Restrictions

Snsequence

The primitive control action snsequence (p. 28) can take an arbitrary number of acts, but SNePSLOG cannot accept a function of an arbitrary number of arguments, so in SNePSLOG Mode 3, snsequence is limited to two argument acts. The way around this is to define versions of snsequence for more than two arguments. This is illustrated for a sequence of four acts below:

```
: set-mode-3

Net reset
In SNePSLOG Mode 3.
...

: ;;; Define the case frame for a 4-act sequence
define-frame snsequence4 (action object1 object2 object3 object4)
snsequence4(x1, x2, x3, x4) will be represented by
     {<action, snsequence4>, <object1, x1>, <object2, x2>,
                                       <object3, x3>, <object4, x4>}

: ;;; but attach it to the original snsequence primitive action
^(attach-primaction snsequence4 snsequence)
t

: ;;; Define a specific primitive action to do four times
define-frame say (action object)
say(x1) will be represented by {<action, say>, <object, x1>}

: ^(define-primaction sayAction (object)
     (print (sneps:choose.ns object)))
sayAction

: ;;; Attach it
^(attach-primaction say sayAction)
t
```

```
: ;;; Test it
perform snsequence4(say(one), say(two), say(three), say(four))

one
two
three
four
```

Zero-Argument Predicates and Functions

SNePSLOG syntax does not accept atomic propositions, but it does accept zero-argument predicates and zero-argument functions as long as the parentheses are included. A way to perform zero-argument actions In SNePSLOG is shown here:

```
: set-mode-3
Net reset
In SNePSLOG Mode 3.
...

: ^^
--> ;;; Define a zero-argument action
(define-primaction reportAction ()
   (princ "I'm here."))
reportAction
--> ;;; Attach it
(attach-primaction report reportAction)
t
--> ^^

: ;;; Define the frame so that the action arc goes to the action.
define-frame report(action)
report(x1) will be represented by {<action, report>, <nil, x1>}

: ;;; Test it
perform report()
I'm here.
```

6.6 The Tell-Ask Interface

The Tell-Ask interface, is an easy way to interface with SNePS from Common Lisp programs, and should be used for this purpose whenever parts of the knowledge base are built via SNePSLOG.

`(snepslog:tell` *`string`*`)`
`string` must be a valid SNePSLOG input. `Tell` gives *`string`* to the SNePSLOG interpreter, does no printing, and returns whatever the SNePSLOG command would return.

`(snepslog:ask` *`string`* `&key verbose)`
`(snepslog:askifnot` *`string`* `&key verbose)`
`(snepslog:askwh` *`string`* `&key verbose)`
`(snepslog:askwhnot` *`string`* `&key verbose)`
`string` must be a valid argument to the SNePSLOG command `ask`, `askifnot`, `askwh`, or `askwhnot`,

respectively. These functions all give *string* to the appropriate SNePSLOG command, and return what the SNePSUL command would print. If `:verbose` is `t`, the functions print the results as well as returning them.

6.7 The Java-SNePS API

Users who are either running SNePS under Franz's ACL, or as the distributed executable have a Java-SNePS API available.

To use SNePS from a Java program:

- include in the Java program the import statements

```
import java.util.HashSet;
import edu.buffalo.sneps.JavaSnepsAPI;
import edu.buffalo.sneps.Substitution;
```

- include the locations of the files `jlinker.jar` and `JavaSnepsAPI.jar` in the Java classpath.

The Java program must create an object of the `JavaSnepsAPI` class. There are two constructors for `JavaSnepsAPI`:

1. `public JavaSnepsAPI(java.lang.String` *`config_file`*`, int` *`interface_port`*`)`
   Creates an instance of `JavaSnepsAPI` using the specified *`config_file`* and *`interface_port`*, and starts SNePS using information specified in the *`config_file`*. The config_file is normally in the `JavaSnepsAPI` subdirectory of the SNePS home directory, and is named `java_sneps_config.config`, but ask the individual who installed SNePS for specifics.

2. `public JavaSnepsAPI(int` *`interface_port`*`)`
   Creates an instance of `JavaSnepsAPI` using the specified *`interface_port`*. It is presumed that the user will then manually start SNePS and invoke the connection from Common Lisp using the function, `snepslog:init-java-sneps-connection`:

   `(snepslog:init-java-sneps-connection` *`port classpath`*`)`
   Initialize a connection between SNePS and Java on the specified *`port`* using the given java *`classpath`* string. Call this function after creating the `JavaSnepsAPI` object in the Java process.

The `JavaSnepsAPI` class provides the methods:

- `void tell(java.lang.String command)`
- `java.util.HashSet<java.lang.String> ask(java.lang.String command)`
- `java.util.HashSet<java.lang.String> askifnot(java.lang.String command)`
- `java.util.HashSet<Substitution> askwh(java.lang.String command)`
- `java.util.HashSet<Substitution> askwhnot(java.lang.String command)`
- `boolean isConnected()`
- `void endLispConnection()`

The `Substitution` class provides the method

- `java.lang.String getValFromVar(java.lang.String var)`.

See the Java Documentation for details on using the API and its classes and methods. The `JavaSnepsAPI` subdirectory of the SNePS home directory contains the documentation and a file that provides an example of using the Java-SNePS API, named `TestAPI.java`. The documentation is also located at `http://www.cse.buffalo.edu/sneps/Docs/javadocs/`.

Chapter 7

Procedural Attachment

Normally, if the system backchains into a proposition-valued function node, it will use inference to determine what instances of the (negation of the) node may be asserted. If, however, the node has an arc labelled `attachedfunction` to a node that has been attached to a function, the system will determine what instances of the (negation of the) node may be asserted by evaluating the attached function. For instance, backchaining into the node

```
(p1 (attachedfunction Sum) (add1 3) (add2 4) (sum v1))
```

could result in a computation to determine that the node

```
(m1 (attachedfunction Sum) (add1 3) (add2 4) (sum 7))
```

should be asserted.

To use the procedural attachment facility, the user must do three things:

1. if using SNePSLOG, define the frames for the predicates to which functions will be attached, if using SNePSUL, define the relations to be used as function arguments;
2. define the attached function;
3. attach the function to a SNePS predicate node.

These steps are more fully explained below.

1. For an example of (1) using the above example, the SNePSLOG user would do:

   ```
   define-frame Sum(attachedfunction add1 add2 sum)
   ```

   and the SNePSUL user would do:

   ```
   (define add1 add2 sum)
   ```

   The relation, `attachedfunction` is defined by the SNePS system.

2. Attached functions must be defined by

   (define-attachedfunction fun (*lambda variables*) &body)

   where:

   (a) `fun` will be the name of the attached function.

(b) each lambda variable must either be a relation used for the arguments of the predicate node, or such a relation enclosed in a pair of parentheses. If the lambda variable is an atomic relation name, it will be bound to the (modified) set of nodes that that relation points to. If it is a relation enclosed in parentheses, the relation symbol will be bound to one element of the (modified) set of nodes that that relation points to (presumably, there will only be one). The way that the nodes will be modified is

i. if the node is a variable, it will be left alone;

ii. if the node's name looks like a Lisp number, the number will be provided;

iii. otherwise, a Lisp symbol whose name is the same string as the node's name will be provided.

In the body of the attached function, the three Lisp predicates `numberp`, `symbolp`, and `sneps:isvar.n` may be used to distinguish the three types of argument.

(c) the attached function must return a list, each of whose members is a list of two members:

i. Either: `snip:pos` to indicate that the instance of the proposition is to be asserted; or `snip:neg` to indicate that the negation of the instance of the proposition is to be asserted;

ii. A substitution of the form `( ... (`*`var term`*`) ...)` to indicate the appropriate instance of the proposition, where each *`var`* is a variable-node argument, and *`term`* is a Lisp object to be converted into a node giving the instance.

If the attached function returns `nil`, that indicates that neither any instance of the proposition nor any instance of its negation is to be asserted.

A simple attached-function definition for the above example is:

```
(define-attachedfunction sumfn ((add1) (add2) (sum))
  (cond
   ;; If all three are numbers, check that it is correct.
   ((and (numberp add1) (numberp add2) (numberp sum))
    (if (cl:= (cl:+ add1 add2) sum)
        '((snip:pos nil))
      '((snip:neg nil))))
   ;; If add1 and add2 are numbers, and sum is a variable,
   ;;     compute the sum.
   ((and (numberp add1) (numberp add2) (sneps:isvar.n sum))
    `((snip:pos ( (,sum . ,(cl:+ add1 add2))) )))
   ;; Else, don't give an answer
   (t nil)))
```

3. The user must attach functions to SNePS predicates using

```
(attach-function node fun node fun ...)
```

For example

```
(attach-function Sum sumfn)
```

After which, the SNePSLOG user can ask questions such as

```
        Sum(3,4,7)?
and     Sum(4,6,?x)?
```

and the SNePSUL user can ask questions such as

```
        (deduce attachedfunction Sum add1 3 add2 4 sum 7)
and     (deduce attachedfunction Sum add1 3 add2 4 sum $x)
```

Chapter 8

SNeBR: The SNePS Belief Revision System

8.1 Hypotheses, Contexts, and Belief Spaces

A proposition may be asserted as an **hypothesis** or as a **derived belief**. An hypothesis is a proposition asserted into the SNePS knowledge base (KB) by the user or by the `believe` mental act. A derived belief is one whose assertion status depended on derivation, via the implemented rules of inference, from other beliefs.

A **context** (*see also* §1.6) is a named structure that contains a set of hypotheses. There is always a **current context** within which all reasoning is performed. (In SNePSUL, many commands take an optional context argument to change this default behavior.)

A **belief space** (BS) defined by a context is the union of the set of hypotheses of the context and the set of derived beliefs that were derived from the hypotheses of the context. The current BS is the BS defined by the current context. Whenever it is said that a proposition is asserted, it should be understood as being asserted in the current BS.

8.2 Responsibilities of SNeBR

SNeBR, the SNePS Belief Revision System has five responsibilities:

1. Recognize when the current BS is contradictory.
2. Identify the possible culprits of a contradiction.
3. Help the user choose the culprit to be blamed for a contradiction.
4. Disbelieve any proposition derived from an hypothesis that is disbelieved (disbelief propagation).
5. Warn the user that is about to make the current BS be one that is already known to be contradictory.

These responsibilities are discussed in the following sections, using the SNePSLOG (Chapter 6) syntax.

8.3 Recognizing a Contradiction

SNeBR recognizes that the current BS is contradictory when both some proposition *p* and `andor(i,0)`{*p*, ...} are asserted (They could be asserted in either order.), or when the SNeRE mental act `believe` is about to assert a proposition that contradicts an already asserted proposition. Thus SNeBR doesn't recognize that

the BS is contradictory in the sense that a contradiction *could* be derived; it only recognizes that contradiction *has been* (or is about to be) explicitly believed. We therefore speak of a BS being "known to be contradictory." Similarly, we speak of a context being known to be contradictory when a contradiction has been derived from a subset of its hypotheses.

8.4 Identifying Possible Culprits

Every proposition that was ever asserted in any BS will have a **support set**, a set of supports, each of which is a triple consisting of an origin tag, an origin set, and a restriction set. If the proposition p has been introduced as an hypothesis, one of its supports will be $\{\langle$`hyp, `$\{p\}$`, `$\{\}\rangle\}$, where `hyp` is the origin tag indicating that p is an hypothesis, $\{p\}$ is the origin set containing p as its only element, and $\{\}$ is the empty restriction set. If a proposition p has been derived, it will have a support containing an origin tag of either `der` or `ext`, and an origin set which will be a set of hypotheses from which p was derived. The SNePS inference engine, SNIP, is an implementation of a version of the logic of relevant entailment, and guarantees that every hypothesis in the origin set was relevantly used to derive p, at least in that way—a proposition may be derived in multiple ways, and so have multiple supports. If a support $\langle\tau, \sigma, \rho\rangle$ has a non-empty restriction set, ρ, the restriction set ρ will be a set of sets of hypotheses, $\{\rho_1, \ldots, \rho_n\}$ such that each set of hypotheses $\sigma \cup \rho_i$ is known to be contradictory.

Whenever a contradictory pair of propositions, p and `andor(`i`,0)`$\{p,$ `...`$\}$ is asserted, the two propositions will have support sets $\{\langle\tau_1, \sigma_1, \rho_1\rangle$`, ..., `$\langle\tau_n, \sigma_n, \rho_n\rangle\}$ and $\{\langle\tau\prime_1, \sigma\prime_1, \rho\prime_1\rangle$`, ..., `$\langle\tau\prime_n, \sigma\prime_n, \rho\prime_n\rangle\}$. Every set of hypotheses $\sigma_i \cup \sigma\prime_j$ is now known to be contradictory, and comprises a set of possible culprits for the contradiction. At least one member of each of those sets must be chosen as a culprit, and removed from the current context, if the current BS is to be restored to a state of not being known to be contradictory.

8.5 Choosing a Culprit

8.5.1 Automatic Culprit Choosing

If the SNeRE act `believe` is performed on the proposition p, it is assumed that belief in p is to take priority over any contradictory belief. Therefore, SNeBR behaves as follows.

1. If `andor(0,0)`$\{p,$ `...`$\}$ is believed as an hypothesis, it is chosen as the culprit. If it is a derived belief, assisted culprit choosing is done.

2. If `andor(`i`,1)`$\{p,$ q`, ...`$\}$ is believed and q is believed as an hypothesis, then q is chosen as the culprit. If q is a derived belief, assisted culprit choosing is done[1].

8.5.2 Assisted Culprit Choosing

If automatic culprit choosing is not done, the user must decide what to do about the contradictory BS with the assistance of SNeBR.

First, SNeBR informs the user that "a contradiction was detected," and specifies the contradictory context and the two contradictory propositions, showing the support set of each.

Then, SNeBR gives the user a choice (not necessarily in the order shown):

1. The user may choose to not do the input that led to the contradiction. The context and BS are left as they were.

[1] In the propositional case, when all proposition-valued functions have arity zero, the contradictory BS is is left as is.

2. The user may choose to continue in an inconsistent context. That situation is not disastrous, since SNePS uses a paraconsistent logic—a contradiction does not imply irrelevant other propositions.

3. The user may choose to repair the context now known to be inconsistent.

In case the user chooses to repair the context, SNeBR then presents the user with each set of possible culprits, as follows.

1. If any of the possible culprits is in the origin set of both contradictory propositions, the user is warned that choosing that as the culprit "might be too extreme a revision."

2. Then SNeBR lists each of the possible culprits, listing all the propositions it supports, and showing how many propositions are in the list. Choosing this as the culprit and disbelieving it will result in all these supported propositions also being removed from the BS (unless they have alternate derivations).

3. The user may then

   - "discard" some of the possible culprits from the context, choosing them as actual culprits;
   - examine all the hypotheses in the context, even those that are not possible culprits;
   - see the culprits already chosen;
   - get instructions;
   - or "quit revising this set" of possible culprits.

   This choice repeats until the user chooses to "quit revising this set".

This entire process is repeated for each set of possible culprits until at least one actual culprit has been chosen for each one.

After identifying at least one actual culprit in each set of possible culprits, the user is shown the remaining list of consistent (really, not known to be inconsistent) hypotheses in the context, and given the same choices as though it were a list of possible culprits. Thereby, the user may disbelieve additional hypotheses, even though they were not responsible for the contradiction.

Finally, the user is given a chance to add new hypotheses to the context. This is especially useful if a rewording of one of the culprits is wanted.

8.6 Disbelief Propagation

Every culprit chosen either automatically or by the user is removed from the set of hypotheses of the current context.

The assertion status of a proposition is computed dynamically whenever it is needed. A proposition is asserted in a given BS if the origin set of at least one of its supports is a subset of the hypothess set of the context of the BS. Thus, if an hypothesis is removed from a context, any derived belief of the BS defined by that context that needed the hypothesis for its derivation is automatically no longer asserted in the BS.

8.7 Warning About a Belief Space Known to be Contradictory.

Every set of possible culprits, α, is a set of hypotheses known to be contradictory. Every context, besides having a set of hypotheses, β, also has a restriction set, $\{\rho_1, \ldots, \rho_n\}$, such that $\beta \cup \rho_i$ is known to be contradictory. When α is discovered to be contradictory, the set $\alpha - \beta$ is added to the restriction set of the context, and any set in the restriction set that is a superset of another set in the restriction set is deleted from the restriction set. Any attempt to create a context whose restriction set contains an empty set is already known to be contradictory, and the user is warned about it.

Chapter 9

SNaLPS: The SNePS Natural Language Processing System

The SNePS Natural Language Processing System consists of a Generalized Augmented Transition Network (GATN) grammar interpreter/compiler and a morphological analyzer/synthesizer.

9.1 Top-Level SNaLPS Functions

The top-level SNaLPS functions are not SNePSUL commands, so to use them from the top-level SNePSUL loop, use the ^ command.

`(atnin` *file* `&key :check-syntax)`
Loads the GATN grammar in *file*. See Section 9.3 for the syntax of the grammar file. If `:check-syntax` is `t`, syntax checks and some simple semantic checks are performed on the grammar as it is input. It reports errors such as undefined target states, pre-actions after actions (on `push` arcs), and syntactically ill-formed arcs. If `:check-syntax` is `nil` (default), this checking is not done.

`(lexin` *file*`)`
Loads a lexicon from *file*. See Section 9.4.1 for the syntax of a lexicon file.

`(parse` *[state] [trace-level]*`)`
Enter the SNaLPS read-parse-print loop. The optional parameters *state* and *trace-level* may appear in either order. If present, *state* must be a symbol, and becomes the initial state of the GATN grammar (the default is `'S`). If present, *trace-level* must be an integer (default, `0`), and becomes the trace level. See below for the effects of the possible trace-levels.

`(break-arc` *state arc-no [break-message]*`)`
Causes a Lisp break to occur on the arc numbered *arc-no* (in the order as input by `atnin`) out of state *state* just before the *test* is performed; if a *break-message* is present it is passed to the Lisp `format` function and printed. In the break package, the contents of registers (and SNaLPS variables) may be examined and modified (by using the GATN actions and forms). Additionally, the current configuration of the parser may be viewed by using `current-configuration`. Resuming, by entering `^^`, `:continue`, or `continue` to the break listener, continues the parse at the point where it was broken. This function, `unbreak-arc`, and `current-configuration` are intended to help debug grammars.

`(unbreak-arc` *state arc-no*`)`
Removes a break from an arc previously set by `break-arc`.

`(current-configuration [trace-level])`
Prints out the current configuration of the GATN parser. Depending on the value of *trace-level* (defaults to `*trace-level*`, see below) the configuration is printed in lesser or greater detail.

9.2 The Top-Level SNaLPS Loop

9.2.1 Input to the SNaLPS Loop

Having executed the `parse` command, the user will enter the SNaLPS top-level read-parse-print loop, the prompt of which is "`:`". At each prompt, the user may enter any of the following:

`^end`: Terminate the SNaLPS read-parse-print loop. If `*terminating-punctuation-flag*` (see below for a description of SNaLPS variables) is non-`nil` then this must be terminated by a terminating punctuation character.

`^` or `^^`: Enter an embedded Lisp read-eval-print loop. This is especially useful to set SNaLPS variables (see below). This Lisp loop is left by entering `continue`, or `:continue`, or `^^`. (If `*terminating-punctuation-flag*` (see below for a description of SNaLPS variables) is non-`nil` then the command for entering an embedded Lisp loop must be terminated by a terminating punctuation character.)

A sentence: A sentence to be parsed is written in the normal fashion—as a sequence of words, not enclosed in parentheses, with punctuation immediately following a word rather than separated by blanks. The initial word may be capitalized. The sentence may extend over several lines either by ending every line except the last with one or more blanks, or by setting the variable `*terminating-punctuation-flag*` to a list of sentence-terminating punctuation marks (See below for a description of SNaLPS variables). Each punctuation mark must be parsed by the grammar as a separate word.

Besides the grammar, the behavior of SNaLPS is affected by a set of variables and by the trace-level. These are described below.

9.2.2 SNaLPS Variables

These variables affect the behavior of SNaLPS, and may be set within an embedded Lisp loop within the SNaLPS loop.

`*all-parses*` If `nil` (default), only the first parse is produced; if `t`, all parses are produced, with the user queried as to whether or not to produce more parses after each one is printed.

`*parse-trees*` If `nil` (default), the contents of all GATN registers are assumed to be sequences (flattened lists), any list structure stored into a register is automatically flattened, and an atom and a list of one atom are considered to be the same; if `t` this flattening is not done, and list structure is preserved.

`*terminating-punctuation-flag*` A list of punctuation marks that will be used to signal the end of a sentence. Each punctuation mark should be enclosed in quote marks to make it a string. E.g. the list might be `("." "?" "!")`. If this variable is `nil` (default), the only way to let a sentence extend over more than one line is to end each line before the last with a blank. If this variable is non-`nil`, the SNaLPS top-level reader will continue to read words from successive lines until a terminating punctuation mark is encountered.

`*trace-level*` An integer indicating what printing is to be done by SNaLPS. The possibilities are listed below. Each trace level also does what every lower level does.

`-2`: SNaLPS does no printing.

`-1`: Prints the time taken by each parse.

`0`: Default. Prints the trace level, the starting state, and the results of each parse. If the result is a SNePS node it is printed using SNEPS::DESCRIBE. Various error messages may also be printed at this trace level.

`1`: A warning is printed if a word is looked up in the lexicon, but not found there.

`2`: Reserved for future trace information.

`3`: Reserved for future trace information.

`4`: Prints the string representation of the original sentence, and prints a trace of the execution of each arc traversed and its associated configuration.

`5`: Prints the current configuration of the GATN parser prior to taking any transitions. Any HOLDs, SENDRs, or LIFTRs associated with a configuration being output are printed.

`6`: Prints the arc currently being attempted, regardless of success. Also prints all blocked arcs.

`7`: Reserved for future trace information.

`8`: Parent configurations of a configuration being output are also printed. This is voluminous, but useful for debugging deeply embedded (via PUSH, CALL, or RCALL) configurations.

`9`: All the acts associated with a configuration that was PUSHed, CALLed, or RCALLed to are printed. This is not normally useful.

`10`: Reserved for future trace information.

Higher trace levels result in more voluminous trace output. For straight-forward trace output, trace level 4 is suggested. For the purposes of printing configurations, if a particular field of the configuration is empty it is not printed regardless of the trace level. For example, if in the configuration being printed, there are no registers, this field is not printed. Note that this does NOT apply to the LEVEL, STATE, and STRING fields, which are always printed regardless of the trace level.

Trace levels up to level 8 can be utilized with little effort and beneficial effects. It is suggested that a novice user closely examine a level 8 trace to become familiar with the operation of the parser.

9.3 Syntax and Semantics of GATN Grammars

In this section, syntactic variables are in *italic* font, optional constituents are enclosed in "[" and "]" brackets, "*" means zero or more occurrences, "+" means one or more occurrences, parentheses and items in `typewriter` font are object language symbols.

gatn-grammar ::= *arc-set*$^+$
The contents of a grammar-file is a sequence of *arc-sets*.

arc-set ::= `(`*state arc*$^+$`)` | `(^^ .` *Lisp-form*`)`
An *arc-set* is either a list whose `car` is a state and whose `cdr` is a list of GATN arcs, or it is a list whose `car` is the symbol `^^` and whose `cdr` is a Lisp form which will be evaluated at grammar load time.

9.3.1 Arcs

arc ::=
The syntax and semantics of GATN arcs follows. Except as noted, all *forms* and *actions* on an arc are evaluated in left-to-right order, in the environment of the arc that they are on.

`(cat` *category test action** *terminal-action* `)`
If the input buffer is empty, the arc blocks. The `lex` register is set to the current word. If any sense(s) of the word have the given lexical *category* in the lexicon, the arc is taken non-deterministically for each sense. For each sense, `*` is set to the root of the word sense, and *test* is performed. If *test* fails, the arc blocks for the given word sense; otherwise, *actions* are done, and the *terminal-action* is taken.

`(call` *state form test preaction-or-action** *register action** *terminal-action* `)`
If the input buffer is empty, the arc blocks. The `*` register is set to the current word, and *test* is performed. If *test* fails, the arc blocks. Otherwise: the *preaction-or-actions* are done; the value of *form* replaces the current word in the input buffer (if the value of *form* is a list, it is spliced in); and parsing continues recursively at *state*. If the recursively called network blocks, this arc blocks also. When that level pops back to this one: the value popped to this level is put into *register*; the value of the `*` register is pushed onto the front of the input buffer; the *actions* are done, and the *terminal-action* is taken.

`(group` *arc*$^+$ `)`
Presumably, at most one *arc* has a *test* that succeeds. That arc is taken. `group` is provided for efficiency only. If a `group` arc is backed into, alternate sub-arcs are not tried, but the entire group fails.

`(jump` *state test action** `)`
The `*` register is set to the current word, and *test* is performed. If *test* fails, the arc blocks. Otherwise the *actions* are done, and parsing continues at *state*.

`(pop` *form test action** `)`
If *test* is non-`nil` and the `hold` register doesn't contain anything put into it on this level, the *actions* are done, and the value of *form* is popped to the level that recursively called this one. If this is the top level (level 1), then not only must *test* be non-`nil` and the `hold` register not contain anything put into it on this level, but also the input buffer must be empty. If this is level 1 and the input buffer is not empty, or this is any level and *test* is `nil` or the `hold` register contains something put into it at this level, this arc blocks.

`(push` *state test preaction** *action** *terminal-action* `)`
If the input buffer is empty, the arc blocks. Otherwise, the `*` register is set to the current word, and `test` is performed. If the *test* is non-`nil`, the *preactions* are done, and parsing continues recursively at *state*. When that level pops back to this one the value popped to this level is put into the `*` register, that value is pushed onto the front of the the input buffer, the *actions* are done, and the *terminal-action* is taken. If the *test* is `nil`, or the recursively called network blocks, this arc blocks.

`(to (`*state [form]*`)` *test action** `)`
If the input buffer is empty, the arc blocks. Otherwise, the `*` register is set to the current word, and `test` is performed. If the *test* is non-`nil`; the *actions* are done; the front symbol in the input buffer is removed from it; if *form* is present, its value is pushed onto the front of the input buffer; and parsing continues at *state*. If the *test* is `nil`, the arc blocks.

`(rcall` *state form test preaction-or-action** *register action** *terminal-action* `)`
If the input buffer is empty, the arc blocks. Otherwise, the `*` register is set to the current word, and `test` is performed. If the *test* is non-`nil` the *preaction-or-actions* are done, the current input buffer is saved and replaced by the value of *form*, and parsing continues recursively at *state*. When that level pops back to this one, the input buffer is restored as it was saved, the value popped to this level is put into *register*, if *register*

is * that value replaces the front of the input buffer, the *actions* are done, and the *terminal-action* is taken. If the *test* is `nil`, or the recursively called network blocks, this arc blocks.

`(tst` *label test action** *terminal-action* `)`
If the input buffer is empty, the arc blocks. Otherwise, the * register is set to the current word, and `test` is performed. If the *test* is non-`nil`, the *actions* are done, and the *terminal-action* is taken. If the *test* is `nil`, the arc blocks. *label* is only there so that different `tst` arcs may be distinguished during tracing.

`(vir` *category test action** *terminal-action* `)`
If the *test* is non-`nil` and the `hold` register contains an entry of the given *category* put into it at this or a higher level, then the arc is taken non-deterministically for each such entry. The entry taken is removed from the `hold` register, put into the * register, and pushed onto the front of the input buffer. Then the *actions* are done and the *terminal-action* is taken. If the *test* is `nil`, or `hold` contains no appropriate entry, this arc blocks.

`(wrd` *word-list test action** *terminal-action* `)`
If the input buffer is empty, the arc blocks. Otherwise, the current word is compared to *word-list*. If it is among the words in *word-list*, the * register is set to the current word, and `test` is performed. If the *test* is non-`nil`, the *actions* are done, and the *terminal-action* is taken. If either the *test* is `nil` or the current word is not on the *word-list*, the arc blocks.

9.3.2 Actions

action ::=
The syntax and semantics of GATN actions follows:

`(addl` *register form*$^+$ `)`
The value of *form* is appended to the front (left end) of *register*. If more than one *form* appears, a list of their values, in the order given, is appended to the front of *register*. If `*parse-trees*` is `nil`, the value of *register* is then flattened and a list of one object is changed to just the single object.

`(addr` *register form*$^+$ `)`
The value of *form* is appended to the end (right end) of *register*. If more than one *form* appears, a list of their values, in the order given, is appended to the end of *register*. If `*parse-trees*` is `nil`, the value of *register* is then flattened and a list of one object is changed to just the single object.

`(hold` *category form* `)`
The value of *form* is put into the `hold` register as an entry whose category is the value of *category*.

`(liftr` *register [form]* `)`
register is set to the value of *form* on the next higher level. If *form* is omitted, *register* at that level is set to the value of *register* at this level. If `*parse-trees*` is `nil`, the value of *register* at the higher level is flattened and a list of one object is changed to just the single object.

`(setr` *register form*$^+$ `)`
register is set to the value of *form*. If more than one *form* appears, *register* is set to a list of their values, in the order given. If `*parse-trees*` is `nil`, the value of *register* is then flattened and a list of one object is changed to just the single object.

S-expression
Any Lisp S-expression not otherwise listed as a GATN action or form evaluates normally.

9.3.3 Preactions

preaction-or-action ::= *preaction* | *action*
A *preaction-or-action* is either a *preaction* or an *action*.

preaction ::= `(sendr` *register form** `)`
The only *preaction* is `sendr`. `sendr` sets the value of *register* on the level about to be called to the value of *form*. If more than one *form* appears, *register* is set to a list of their values, in the order given. If *form* is omitted, *register* at the lower level is set to the value of *register* at this level. If `*parse-trees*` is `nil`, the value of *register* at the lower level is flattened and a list of one object is changed to just the single object.

9.3.4 Terminal Actions

terminal-action ::=
The syntax and semantics of the two terminal actions are:

`(to` *state [form]* `)`
The front symbol in the input buffer is removed from it; if *form* is present, its value is pushed onto the front of the input buffer; and parsing continues at *state*.

`(jump` *state* `)`
Parsing continues at *state*.

9.3.5 Forms

form ::=
The syntax and semantics of GATN forms follows:

`*`
A form consisting of only the symbol `*` evaluates to the value of the `*` register on the current level.

`(buildq` *fragment form** `)`
Evaluates to the given *fragment* list structure, with each special symbol replaced as indicated below:

+: Replaced by the value of the corresponding *form*.

`*`: Replaced by the value of the `*` register.

`(@` *fragment*$^+$ `)`: Each fragment is handled as described here, and the resulting sublists are spliced together.

`(geta` *unitpath form* `)`
Returns the set of SNePS nodes at the end of *unitpath* arcs from all the nodes in the value of *form*. *form* must evaluate to a set of SNePS nodes.

`(getf` *feature [word]* `)`
Looks up *word* in the lexicon, and returns the value of the given *feature*. If *word* is omitted, it defaults to the current word—this is only allowed on `cat` arcs.

`(getr` *register [level-number]* `)`
Evaluates to the value of the given *register*. If *level-number* is present, the value of *register* on the given level is returned. The top level of the network is level number 1, and each push increments the level number by 1. If *level-number* is present, it must be a higher level (smaller integer) than the current level.

`lex`
Evaluates to the current word (first symbol on the input buffer) as it appears before any morphological analysis has been done.

`(nullr` *form*`)`
Evaluates to non-`nil` if the *form* evaluates to `nil`, and to `nil` otherwise.

`(overlap` *form form*`)`
Evaluates to the set-intersection of the values of the two *forms*. For this purpose, an atom is treated as a singleton set containing itself.

register
A form consisting of only the name of a register evaluates to the value of that register on the current level. If the *register* has never been given a value on the current level, its value is `nil`.

S-expression
Any Lisp S-expression not otherwise listed as a GATN form evaluates to its Lisp value. Within a form to be evaluated by Lisp, `buildq, geta, getf, getr, nullr,` and `overlap` are all Lisp functions that operate as described in this Subsection.

9.3.6 Tests

A test fails if it evaluates to `nil`, and succeeds otherwise.

test ::=
The syntax and semantics of GATN tests are:

`(disjoint` *form form*`)`
Evaluates to `(not (overlap` *form form*`))`.

`(endofsentence)`
Returns `t` iff the current input buffer is empty (`nil`), or contains only a terminating punctuation character (as specified by `*terminating-punctuation-flag*`); `nil` otherwise.

`(packageless-equal` *form form*`)`
Equality checker for use in grammars, it should be used rather than Lisp's `equal`. It avoids possible packaging problems associated with the use of Lisp's `equal`.

form
Any GATN form may be used as a test—any non-`nil` value corresponds to True, and `nil` corresponds to False.

`t`
The always True test.

9.3.7 Terminal Symbols

The following is a description of Terminal Symbols of the above grammar (right hand sides are informal English):

category ::= *Any Lisp symbol used in the lexicon as a lexical category.*

feature ::= *Any Lisp symbol used in the lexicon as a lexical feature.*

label ::= *Any Lisp symbol.*

register ::= *Any Lisp symbol used as the name of a GATN register.*

S-expression ::= *Any Lisp S-expression not otherwise recognizable as a GATN form or test.*

state ::= *Any Lisp symbol used to name a GATN state.*

word-list ::= *word* | (*word*$^+$)

word ::= *Any Lisp string potentially used as a lexicon entry or a word appearing in a sentence.*

9.4 Morphological Analysis and Synthesis

9.4.1 Syntax of Lexicon Files

The syntax of the contents of a lexicon file is:

lexicon ::= *lexical-entry**

lexical-entry ::= (*lexeme feature-list*$^+$)
A multi-sense lexeme is given one feature-list for each sense. Lexemes on which morphological synthesis is to be done for generation must have only one feature-list.

feature-list ::= (*feature-pair*$^+$)

feature-pair ::= (*feature* . *value*)

lexeme ::= *Any Lisp string*

feature ::= *Any Lisp atom, but see below for standard features.*

value ::= *Any readable Lisp object, but see below for standard values.*

Standard Lexical Features and Values

The following lexical features will be recognized by GATN arcs and/or the morphological analyzer/synthesizer. With each feature it is shown whether it is used for morphological analysis or synthesis or both. A feature whose value is the default for that feature may be omitted from the feature-list.

`ctgy` The lexical category of the entry. Used for analysis. The following values of this feature are recognized by the morphological analyzer/synthesizer:

- `adj` An adjective.
- `n` A common noun.
- `v` A verb.
- `multi-start` This lexeme is the first word of a multi-word lexeme.
  See the feature `multi-rest` described below.

`multi-rest` A list of the rest of the words (after this one) that form a multi-word lexeme. Each word in the list must be a string. There must also be a lexical entry for the multi-word lexeme as a whole.

`num` The number of a noun or verb. Used for analysis. Its recognized values are `sing` for singular (default), and `plur` for plural.

`past` The past tense form of a verb that is irregular in this form. Used for synthesis. The value must be a string.

`pastp` The past participle form of a verb that is irregular in this form. Used for synthesis. The value must be a string.

`plur` The plural form of a noun that is irregular in this form. Used for synthesis. The value must be a string.

`pprt` A flag indicating whether the lexeme is a past participle. Used for analysis. Possible values are `t` indicating that it is a past participle, and `nil` (default) indicating that it is not.

`pres` The third person singular form of a verb that is irregular in this form. Used for synthesis. The value must be a string.

`presp` The present participle form of a verb that is irregular in this form. Used for synthesis. The value must be a string.

`presprt` A flag indicating whether the lexeme is a present participle. Used for analysis. Possible values are `t` indicating that it is a present participle, and `nil` (default) indicating that it is not.

`tense` The tense of a verb. Used for analysis. Its recognized values are `pres`, indicating present tense (default), and `past`, indicating past tense.

`root` The root, stem, infinitive, or uninflected form of the lexeme. Used for analysis and synthesis. The value must be a string. If omitted, will default to the lexeme itself.

`stative` Whether or not the verb is stative. Used for synthesis. Its recognized values are `nil`, indicating not stative (default), and `t`, indicating stative.

Any additional features and values may be used in a lexicon if a grammar is written to make use of them.

If a lexeme is a root form and has only one feature-list, the lexical entry may be used for both analysis and synthesis. If the principle inflected parts of the noun (plural) or verb (third person singular, past, past participle, present participle) are irregular, those parts must have their own lexical entries, which will only be used for morphological analysis. If a lexeme has multiple lexical entries (because it has multiple lexical categories), they can only be used for analysis. You should give each feature-list its own root form, even if it is a made-up word, and then give each of these root forms a lexical entry with a single feature-list to be used for morphological synthesis. Those feature-lists should have root forms that are correctly spelled forms. The example lexicon in Section 9.5 has a regular and an irregular noun, a regular and an irregular verb, a word (`"saw"`) that is both a noun and a verb, and a multi-word lexeme (`"Computer Science"`).

9.4.2 Functions for Morphological Analysis

`(englex:lookup` *word*`)`

Looks up the word, which must be a string, in the lexicon, and returns its lexical-entry, possibly expanded with default values. If *word* is not in the lexicon, tries to remove prefixes and suffixes until it finds a root form that is in the lexicon, whereupon it returns an appropriately modified lexical-entry. Uses the standard features and values listed above as used for morphological analysis. `englex:lookup` has been imported into the `parser` package for use in grammars, but it is seldom necessary for a grammar-writer to call this function directly, because it is called automatically on appropriate GATN arcs. Nevertheless, when writing a lexicon and grammar, it is useful to use this function to test the lexicon and see which forms `englex:lookup` considers to be regular.

9.4.3 Functions for Morphological Synthesis

The two functions in this section should be used in appropriate places on arcs of GATN generation grammars in order to produce correct surface forms of nouns and verb groups. They both require a loaded lexicon for irregular words, but will assume a word is inflected regularly if it doesn't have a lexical entry. It's a good idea to try them out from a top-level Lisp listener, as they are surprisingly powerful.

(`verbize` *[tense] [number] [person] [mode] [progressive-aspect] [perfective-aspect] [voice] [quality] modal* lexeme)*

Returns a list of strings which forms a verb group. *lexeme* should be a string (but a symbol or node will be coerced into a string), and is presumed to be the root (infinitive) form of a verb. All the other parameters are optional. If *modals* appear, they must immediately precede *lexeme*. The other parameters may appear in any order. The possible values and the default values of the optional parameters are:

tense The tense of the verb group. Possible values are:

- `pres` Present tense. Default.
- `past` Past tense. Alternative: `p`.
- `future` Future tense. Alternatives: `futr`, `ftr`, `fut`.

number The number of the verb group. Possible values are:

- `sing` Singular. Default. Alternative: `singular`
- `plur` Plural. Alternatives: `pl`, `plural`.

person The person of the verb group. Possible values are:

- `p1` First person. Alternatives: `firstperson`, `person1`
- `p2` Second person. Alternatives: `secondperson`, `person2`.
- `p3` Third person. Default.

mode Whether the verb group is in a declarative, interrogative, etc. sentence. Possible values are:

- `decl` Declarative. Default.
- `int` Interrogative. Alternatives: `ynq`, `interrogative`, `interrog`, `ques`, `question`, `intrg`, `Q`.
- `imp` Imperative. Alternatives: `imper`, `imperative`, `impr`, `command`, `request`, `req`
- `inf` Infinitive. Alternatives: `infinitive`, `infin`, `root`
- `gnd` Gerundive. Alternatives: `gerund`, `ing`, `grnd`

progressive-aspect Whether the verb group is progressive. Possible values are:

- `non-prog` Non-progressive. Default.
- `prog` Progressive. Alternatives: `progr`, `prgr`, `prg`, `progress`, `progressive`. A lexeme with `(stative . t)` in its lexical feature-list will not be put into progressive form.

perfective-aspect Whether the verb group is perfective. Possible values are:

- `non-perf` Non-perfective. Default.
- `perf` Perfective. Alternatives: `pft`, `prft`, `prfct`, `perfect`, `perfective`.

voice The voice of the verb group. Possible values are:

`active` Active voice. Default.

`pass` Passive voice. Alternative: `passive`.

quality Whether the verb group is negated. Possible values are:

`affirmative` Affirmative. Default.

`neg` Negative. Alternatives: `nega`, `negative`, `not`, `negated`.

modal A modal to be included in the verb group. Possible values are: `"will"`, `"shall"`, `"can"`, `"must"`, and `"may"`.

`(wordize` *number lexeme*`)`

Returns the singular or plural form of *lexeme*, according to *number*. *lexeme* should be a string(but a symbol or node will be coerced into a string), and is presumed to be a noun. If *number* is `nil` or `'sing` the singular form will be returned, otherwise the plural form will be returned. The singular form will be the value of the `root` feature of *lexeme* if it has one, otherwise it will be *lexeme* itself. The plural will be formed regularly unless there is a lexical entry for *lexeme* that contains a `plur` feature. Regular plural formation is sensitive to an extensive set of spelling rules.

9.5 Examples

In this section, we present an example lexicon and two example GATNs, both of which use the one lexicon. The GATN in Section 9.5.1 produces parse trees of acceptable sentences. The one in Section 9.5.2 builds SNePS representations of the information in statements, and answers questions. Both GATNs accept the same fragment of English.

9.5.1 Producing Parse Trees

Here is an example of the use of SNaLPS to parse simple sentences and return parse trees. The GATN is shown below. It is presented graphically in Figure 9.1.

```
(s (jump s1 t (setf *parse-trees* t))) ; This initial arc is used to
                                       ; set global variables and other parameters.

(s1 (cat wh t ; The only acceptable questions start with a wh word.
         (setr subj '(np \?)) (setr mood 'question) (to vp))
    (push np t ; The only acceptable statements are NP V [NP].
          (setr subj *) (setr mood 'decl) (to vp)))

(vp (cat v t (setr verb *) (to vp/v)))

(vp/v (push np t (setr obj *) (to s/final))
      (jump s/final t)) ; The predicate NP is optional.

(s/final (jump s/end (overlap embedded t)) ; The S might end with an embedded S.
         (wrd "." (overlap mood 'decl) (to s/end))
         (wrd "?" (overlap mood 'question) (to s/end)))
```

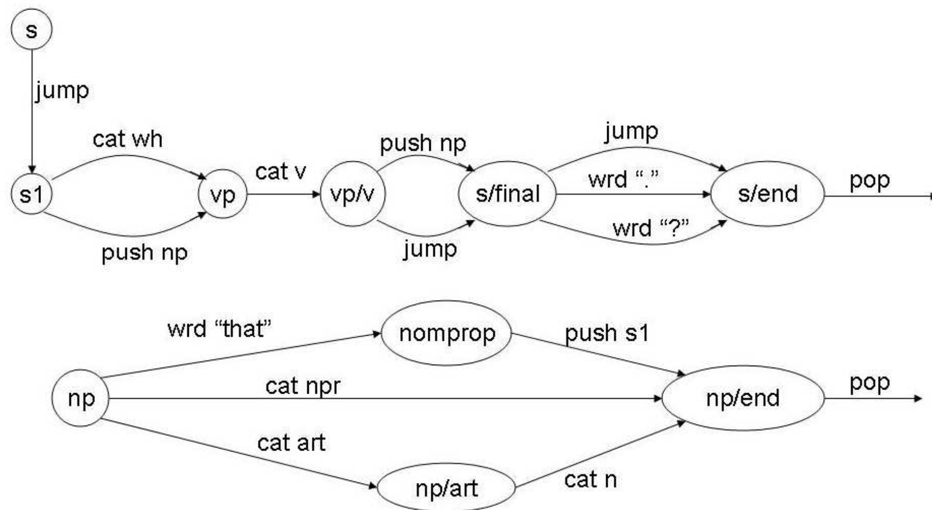

Figure 9.1: Graphical version of the example GATN.

```
(s/end (pop (buildq (s (mood +) + (vp (v +))) mood subj verb) (nullr obj))
       (pop (buildq (s (mood +) + (vp (v +) +)) mood subj verb obj) obj))

(np (wrd "that" t (to nomprop)) ; An embedded S has "that" in front of it.
    (cat npr t (setr np (buildq (npr *))) (setr def t) (to np/end))
    (cat art t (setr def (getf definite)) (to np/art)))

(np/art (cat n t (setr np (buildq (n *))) (to np/end)))

(nomprop (push s1 t (sendr embedded t) (setr def t) (setr np *) (to np/end)))

(np/end (pop (buildq (np (definite +) +) def np) t))
```

Here is the accompanying lexicon:

```
("a"                ((ctgy . art)(definite . nil)))
("the"              ((ctgy . art)(definite . t)))

("Computer Science" ((ctgy . npr)))
("John"             ((ctgy . npr)))
("Mary"             ((ctgy . npr)))

("computer"         ((ctgy . n)))
("Computer"         ((ctgy . multi-start) (multi-rest . ("Science")))
                    ((ctgy . n)(root . "computer")))
("dog"              ((ctgy . n)))
("man"              ((ctgy . n)(plur . "men")))
("men"              ((ctgy . n)(root . "man")(num . plur)))
("woman"            ((ctgy .  n)(plur . "women")))
("women"            ((ctgy . n)(root . "woman")(num . plur)))

("saw"              ((ctgy . n))
                    ((ctgy . v)(root . "see")(tense . past)))

("believe"          ((ctgy . v)(stative . t)))
("bit"              ((ctgy . v)(root . "bite")(tense . past)))
("bite"             ((ctgy . v)(num . plur)(past . "bit")))
("like"             ((ctgy . v)(num . plur)))
("see"              ((ctgy . v)(past . "saw")))
("sleep"            ((ctgy . v)(past . "slept")))
("slept"            ((ctgy . v)(root . "sleep")(tense . past)))
("study"            ((ctgy . v)))
("use"              ((ctgy . v)))

("who"              ((ctgy . wh)))
("what"             ((ctgy . wh)))
```

and here is a test run. The output has been edited by changing some line breaks and some indentation in order to show the parse trees more clearly.

```
USER(29): (sneps)
   Welcome to SNePS-2.5 [PL:1 1999/08/19 16:38:25]
```

```
Copyright (C) 1984--1999 by Research Foundation of
State University of New York. SNePS comes with ABSOLUTELY NO WARRANTY!
Type `(copyright)' for detailed copyright information.
Type `(demo)' for a list of example applications.

   6/21/2002 9:51:59
* ^^
--> (atnin "grammar.lisp")
State S processed.
State S1 processed.
State VP processed.
State VP/V processed.
State S/OBJ processed.
State NP processed.
State NP/ART processed.
State NP/END1 processed.
State NP/END2 processed.
State S/END1 processed.
State S/END2 processed.
 Atnin read in states: (S/END2 S/END1 NP/END2 NP/END1 NP/ART NP S/OBJ VP/V VP
                        S1 S)

--> (lexin "lexicon.lisp")
undefined- (NIL)
("a" "the" "some" "dog" "man" "men" "bite" "bites" "like" "likes")
--> (parse)
 ATN parser initialization...
 Trace level = 0.
 Beginning at state 'S'.

 Input sentences in normal English orthographic convention.
 Sentences may go beyond a line by having a space followed by a <CR>
 To exit the parser, write ^end.

 : A dog bit John.
Resulting parse:
(S (MOOD DECL)
   (NP (DEFINITE NIL) (N "dog"))
   (VP (V "bite")
       (NP (DEFINITE T) (NPR "John"))))
 Time (sec.): 0.05

 : The dog slept.
Resulting parse:
(S (MOOD DECL)
   (NP (DEFINITE T) (N "dog"))
   (VP (V "sleep")))
 Time (sec.): 0.05

 : Mary believes that John likes the dog.
Resulting parse:
```

```
(S (MOOD DECL)
   (NP (DEFINITE T) (NPR "Mary"))
   (VP (V "believe")
       (NP (DEFINITE T)
           (S (MOOD DECL)
              (NP (DEFINITE T) (NPR "John"))
              (VP (V "like")
                  (NP (DEFINITE T) (N "dog")))))))
 Time (sec.): 0.117

 : Mary studies Computer Science.
Resulting parse:
(S (MOOD DECL)
   (NP (DEFINITE T) (NPR "Mary"))
   (VP (V "study")
       (NP (DEFINITE T) (NPR "Computer Science"))))
 Time (sec.): 0.05

 : Mary used a computer.
Resulting parse:
(S (MOOD DECL)
   (NP (DEFINITE T) (NPR "Mary"))
   (VP (V "use")
       (NP (DEFINITE NIL) (N "computer"))))
 Time (sec.): 0.05
```

```
 : John saw a saw.
Resulting parse:
(S (MOOD DECL)
   (NP (DEFINITE T) (NPR "John"))
   (VP (V "see")
       (NP (DEFINITE NIL) (N "saw"))))
 Time (sec.): 0.067

 : What bit John?
Resulting parse:
(S (MOOD QUESTION)
   (NP ?)
   (VP (V "bite")
       (NP (DEFINITE T) (NPR "John"))))
 Time (sec.): 0.05

 : Who sleeps?
Resulting parse:
(S (MOOD QUESTION)
   (NP ?)
   (VP (V "sleep")))
 Time (sec.): 0.034

 : Who studied?
Resulting parse:
(S (MOOD QUESTION)
   (NP ?)
   (VP (V "study")))
 Time (sec.): 0.05

 : Who uses the computer?
Resulting parse:
(S (MOOD QUESTION)
   (NP ?)
   (VP (V "use")
       (NP (DEFINITE T) (N "computer"))))
 Time (sec.): 0.067

 : Who likes a dog?
Resulting parse:
(S (MOOD QUESTION)
   (NP ?)
   (VP (V "like")
       (NP (DEFINITE NIL) (N "dog"))))
 Time (sec.): 0.067
```

```
 : Who sees a saw?
Resulting parse:
(S (MOOD QUESTION)
   (NP ?)
   (VP (V "see")
       (NP (DEFINITE NIL) (N "saw"))))
 Time (sec.): 0.067
```

9.5.2 Interacting with SNePS

The GATN in this section accepts the same fragment of English as the one in the previous section, but, instead of building and returning parse trees, it builds a SNePS network representing the information in the statements and answers the questions. The statements are echoed and the questions are answered in English generated by the generation part of this GATN.

```
;;; First, the SNePS relations used in the GATN are defined.
(^^ define agent act object propername member class lex)

;;; Next, a global variable, a global constant, and two functions are defined.
(^^ defvar *SaynBeforeVowels* nil
    "If true and the next word starts with a vowel,
     print 'n ' before that next word.")

(^^ defconstant *vowels* '(#\a #\e #\i #\o #\u)
    "A list of the vowels.")

;;; The following two functions implement a "phonological" component
;;; that can be used to output words and phrases from arcs of the GATN.
;;; In this way, the beginning of the sentence can be uttered before
;;; the rest of the sentence has been composed.

(^^ defun SayOneWord (word)
    "Prints the single WORD, which must be a string or a node.
     If the word is 'a', sets *SaynBeforeVowels*.
     If *SaynBeforeVowels* is set, then prints 'n ' before word/s
     if the first letter of word/s is a vowel."
    (check-type word (or string sneps:node))
    (when (sneps:node-p word) (setf word (format nil "~A" word)))
    (when *SaynBeforeVowels*
      (when (member (char word 0) *vowels* :test #'char=) (format t "n"))
      (setf *SaynBeforeVowels* nil))
    (when (string\= word "a") (setf *SaynBeforeVowels* t))
    (format t " ~A" word))

(^^ defun say (word/s)
    "Prints the single word or the list of words.
     If the word is 'a', sets *SaynBeforeVowels*.
     If *SaynBeforeVowels* is set, then prints 'n ' before word/s
     if the first letter of word/s is a vowel."
    (if (listp word/s) (mapc #'SayOneWord word/s)
        (SayOneWord word/s)))
```

```
;;; The initial arc is used to make two SNePSUL variables, each of
;;; which holds a SNePS variable node.  This results in a major
;;; efficiency gain over creating new SNePS variable nodes each time a
;;; question or an indefinite NP is parsed.
(s (jump s1 t
         (or (* 'wh) ($ 'wh)) ; a SNePS variable to use for Wh questions
         (or (* 'x) ($ 'x)) ; a variable for indef NP's in questions
         ))
(s1 (push ps t                ; Parse a sentence, and send results to RESPOND
          (jump respond)))
(ps (cat wh t                        ; A Wh question starts with "who" or "what".
         (setr agent (* 'wh))        ; set AGENT to a variable node.
         (setr mood 'question) (liftr mood) (to vp))
    (push np t (sendr mood 'decl) ; The only acceptable statements are NP V [NP].
                                  ; MOOD must be sent down, because an indefinite
                                  ; NP introduces a new individual in a statement,
                                  ; but must be treated as a variable to be found
                                  ; in a question.
          (setr agent *)        ; set AGENT to parse of subject.
          (setr mood 'decl) (liftr mood) ; The state RESPOND must know whether
                                         ; it is echoing a statement or answering
                                         ; a question.
          (to vp)))
(vp (cat v t                    ; Accept just a simple verb for this example,
         (setr act *) (to vp/v))) ; and ignore tense.
(vp/v (push np t (sendr mood)
            (setr object *)     ; Set OBJECT to parse of object.
            (to s/final))
      (jump s/final t))         ; If no object.

(s/final (jump s/end (overlap embedded t)) ; an embedded proposition
         (wrd "." (overlap mood 'decl) (to s/end))
         (wrd "?" (overlap mood 'question) (to s/end)))
(s/end (pop #!((assert agent ~(getr agent) ; Assert a top-level statement.
                       act   (build lex ~(getr act))
                       object ~(getr object)))
            (and (overlap mood 'decl) (nullr embedded)))
       (pop #2!((build agent ~(getr agent) ; Build an embedded statement.
                       act   (build lex ~(getr act))
                       object ~(getr object)))
            (and (getr embedded) (overlap mood 'decl)))
       (pop #!((deduce agent ~(getr agent) ; Use deduce to answer a question.
                       act   (build lex ~(getr act))
                       object ~(getr object)))
            (overlap mood 'question)))
;;; Notice in all three above arcs that if there is no object,
;;; (getr object) will evaluate to NIL,
;;; and the node will be built without an OBJECT arc.
```

```
(np (wrd "that" t (to nomprop))        ; an embedded proposition
    (cat npr t
         (setr head (or
                     ;; First try to find someone with the given name.
                     #!((find (compose object- ! propername) ~(getr *)))
                     ;; Otherwise, create one.
                     #!((find object-
                              (assert object #head propername ~(getr *))))))
         (to np/end))
    (cat art t (setr def (getf definite)) (to np/art)))

(np/art (cat n (overlap def t) ; a definite np
             (setr head ; Find the referent. (Assume there is exactly one.)
                   #!((find member-
                            (deduce member *x
                                    class (build lex ~(getr *))))))
             (to np/end))
        (cat n (and (disjoint def t) (overlap mood 'decl))
             (setr head        ; Create a new referent.
               #!((find member-
                        (assert member #hd
                          class (build lex ~(getr *))))))
             (to np/end))
        (cat n (and (disjoint def t) (overlap mood 'question))
             (setr head (* 'x)) ; a variable node.
             (to np/end)))

(nomprop (push ps t ; Return the parse of embedded sentence.
               (sendr embedded t) (setr head *) (to np/end)))

(np/end (pop head t))

;;;;;;;;;;;;;;;;;;;;;;
;;; Generation Section
;;;;;;;;;;;;;;;;;;;;;;

(respond (jump g (and (getr *) (overlap mood 'decl))
               (say "I understand that")) ; Canned beginning of echo of statement.
         (jump g (and (getr *) (overlap mood 'question))) ; Answer of question.
         (jump g/end (nullr *) (say "I don't know."))) ; Question not answered.

(g (rcall gnp (geta agent) (geta agent) ; Generate the agent as an np.
          reg (jump g/subj)))

(g/subj (jump g/v (geta act)
              (say (verbize 'past ; For this example, always use past tense.
                            (first (geta lex (geta act)))))))

(g/v (rcall gnp (geta object) (geta object) ; Generate the object.
            reg (to g/end))
     (to (g/end) (null (geta object)))) ; No object.
```

```
(g/end (pop nil t))

(gnp (to (gnp/end) (geta propername (geta object-))
         (say (geta propername (geta object-)))) ; Generate an npr.
     (to (gnp/end) (geta class (geta member-)) ; An indef np.
         (say (cons "a" #!((find (lex- class- ! member) ~(getr *))))))
     (call g * (geta act) (say "that") * ; An embedded proposition
           (to gnp/end)))

(gnp/end (pop nil t))
```

Here is a sample run using this grammar and the same lexicon as before:

```
--> (parse)
 ATN parser initialization...
 Trace level = 0.
 Beginning at state 'S'.

 Input sentences in normal English orthographic convention.
 Sentences may go beyond a line by having a space followed by a <CR>
 To exit the parser, write ^end.

 : A dog bit John.
 I understand that a dog bit John
 Time (sec.): 0.25

 : The dog slept.
 I understand that a dog slept
 Time (sec.): 1.884

 : Mary believes that John likes the dog.
 I understand that Mary believed that John liked a dog
 Time (sec.): 0.4

 : Mary studies Computer Science.
 I understand that Mary studied Computer Science
 Time (sec.): 0.2

 : Mary used a computer.
 I understand that Mary used a computer
 Time (sec.): 0.233

 : John saw a saw.
 I understand that John saw a saw
 Time (sec.): 0.217

 : What bit John?
 a dog bit John
 Time (sec.): 0.167
```

```
: Who sleeps?
a dog slept
Time (sec.): 0.917

: Who studied?
Mary studied
Time (sec.): 0.167

: Who uses the computer?
Mary used a computer
Time (sec.): 0.267

: Who likes a dog?
I don't know.
Time (sec.): 0.167

: Who sees a saw?
John saw a saw
Time (sec.): 0.217
```

The SNePS network built as a result of this interaction is shown in Figure 9.2. Note especially that when definite noun phrases occurred in statements, they were represented by nodes that were already in the net because of previous indefinite noun phrases.

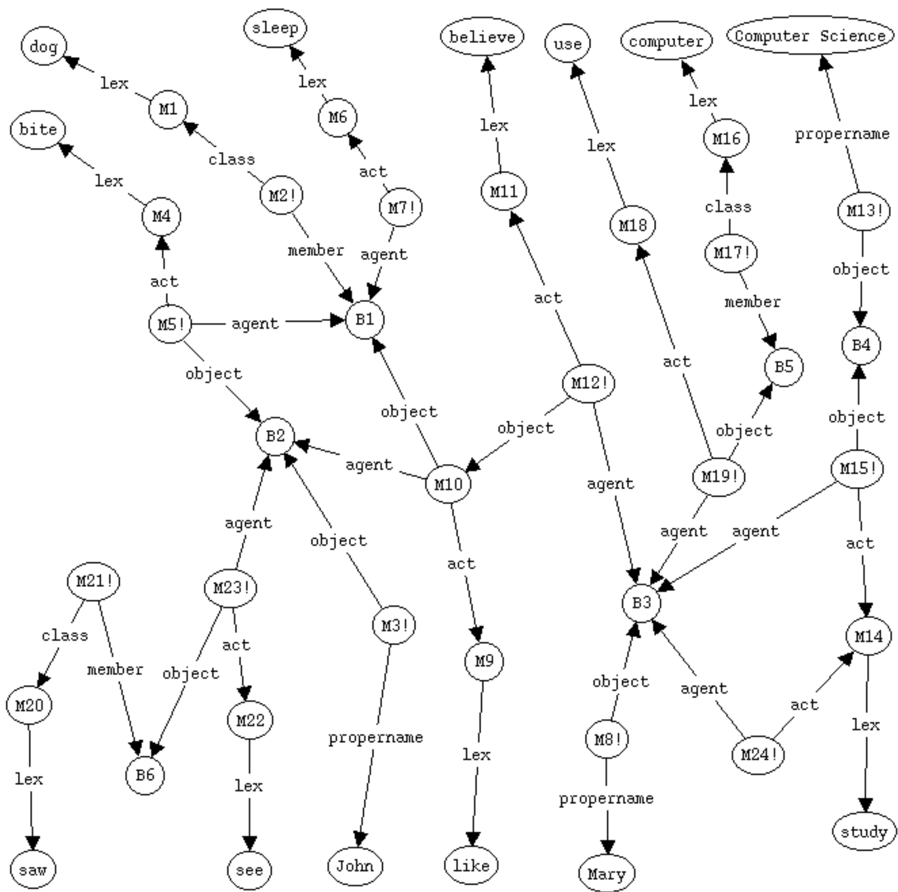

Figure 9.2: SNePS network after running the example.

Chapter 10

SNePS as a Database Management System

SNePS can be used as a network version of a relational database system in which every element of the relational database is represented by a base node, each row of each relation is represented by a molecular node, and each column label (attribute) is represented by an arc label. Whenever a row r of a relation R has an element e_i in column c_i, the molecular node representing r has an arc labeled R to the special node `relation`, and an arc labelled c_i pointing to the base node representing e_i. Table 10.1 shows two relations from the Supplier-Part-Project database of Date[1], p. 114. Figure 10.1 shows a fragment of the SNePS network

Table 10.1: From Date's Supplier-Part-Project Database

SUPPLIER

S#	SNAME	STATUS	CITY
s1	Smith	20	London
s2	Jones	10	Paris
s3	Blake	30	Paris
s4	Clark	20	London
s5	Adams	30	Athens

PROJECT

J#	JNAME	CITY
j1	sorter	Paris
j2	punch	Rome
j3	reader	Athens
j4	console	Athens
j5	collator	London
j6	terminal	Oslo
j7	tape	London

version of this database.

10.1 SNePS as a Relational Database

The three basic operations on relational databases are **select, project,** and **join.** The next three subsections show how these operations may be expressed in SNePSUL.

10.1.1 Project

Project is a database operation that, given one relation, produces another that has all the rows of the first, but only specific columns. (Actually some of the rows might collapse if the only distinguishing elements were

[1] C. J. Date, *An Introduction to Database Systems 3rd Edition* (Reading, MA: Addison-Wesley) 1981.

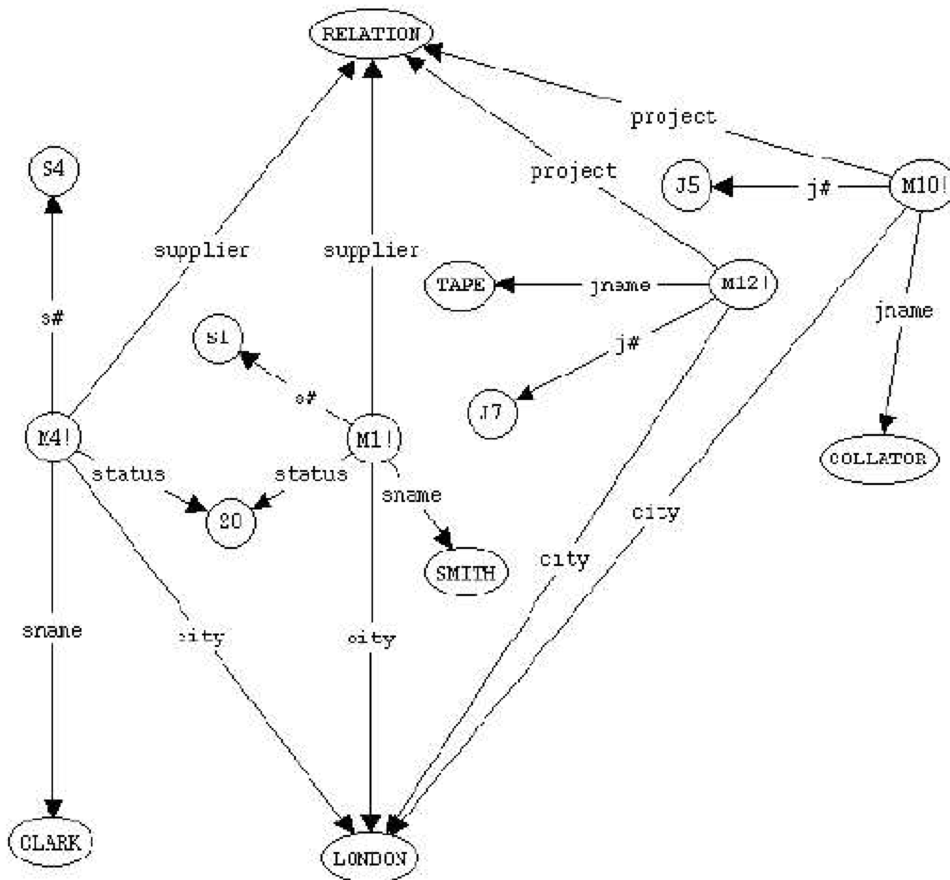

Figure 10.1: Fragment of SNePS network for the Supplier-Part-Project Database

in columns that were eliminated.) The SNePSUL `dbproject` function has been designed for this purpose. For example, to show the STATUS and CITY of all suppliers, one can do

```
* (dbproject (find supplier relation) status city)
((STATUS (20) CITY (LONDON)) (STATUS (10) CITY (PARIS))
 (STATUS (30) CITY (PARIS)) (STATUS (30) CITY (ATHENS)))
 CPU time : 0.07
```

The `dbproject` function forms and returns a *virtual* relation, which is represented as a SNePS data type called a *set of flat cable sets.* Compare the following two ways of getting complete details of the SUPPLIER relation. The first uses the SNePSUL `describe` function to print the details of the nodes that make up the relation:

```
* (describe (find supplier relation))
(M1! (CITY LONDON) (S# S1) (SNAME SMITH) (STATUS 20) (SUPPLIER RELATION))
(M2! (CITY PARIS) (S# S2) (SNAME JONES) (STATUS 10) (SUPPLIER RELATION))
(M3! (CITY PARIS) (S# S3) (SNAME BLAKE) (STATUS 30) (SUPPLIER RELATION))
(M4! (CITY LONDON) (S# S4) (SNAME CLARK) (STATUS 20) (SUPPLIER RELATION))
(M5! (CITY ATHENS) (S# S5) (SNAME ADAMS) (STATUS 30) (SUPPLIER RELATION))
(M1! M2! M3! M4! M5!)
 CPU time : 0.15
```

The second uses `dbproject` to display a virtual relation with the same information:

```
* (dbproject (find supplier relation) supplier s\# sname status city)
((SUPPLIER (RELATION) S# (S1) SNAME (SMITH) STATUS (20) CITY (LONDON))
 (SUPPLIER (RELATION) S# (S2) SNAME (JONES) STATUS (10) CITY (PARIS))
 (SUPPLIER (RELATION) S# (S3) SNAME (BLAKE) STATUS (30) CITY (PARIS))
 (SUPPLIER (RELATION) S# (S4) SNAME (CLARK) STATUS (20) CITY (LONDON))
 (SUPPLIER (RELATION) S# (S5) SNAME (ADAMS) STATUS (30) CITY (ATHENS)))
 CPU time : 0.12
```

Virtual relations are created without building any new SNePS network structure. To make these relations permanent, use the SNePSUL `dbAssertVirtual` function. For example, to create a CITYSTATUS relation that is a projection of the SUPPLIER relation down the CITY and STATUS attributes, we would first define CITYSTATUS as a new SNePS relation:

```
* (define citystatus)
(CITYSTATUS)
 CPU time : 0.03
```

Then we would do

```
* (describe (dbAssertVirtual (dbproject (find supplier relation) city status)
                             (citystatus relation)))
(M13! (CITY LONDON) (CITYSTATUS RELATION) (STATUS 20))
(M14! (CITY PARIS) (CITYSTATUS RELATION) (STATUS 10))
(M15! (CITY PARIS) (CITYSTATUS RELATION) (STATUS 30))
(M16! (CITY ATHENS) (CITYSTATUS RELATION) (STATUS 30))
(M13! M14! M15! M16!)
 CPU time : 0.28
```

10.1.2 Select

Select is an operation that is given a relation and specific values for some of its attributes, and yields the rows of the relations in which those attributes take on those values. A selection from relation R_1 in which attribute a_{1i} takes on value v_{1i} is expressed in SNePSUL as

(find R_1 relation a_{11} v_{11} ...a_{1n} v_{1n}).

For example, to select rows of the SUPPLIER relation where the CITY is Paris or Athens and the STATUS is 30, we could do:

```
* (describe (find supplier relation city (paris athens) status 30))
(M3! (CITY PARIS) (S# S3) (SNAME BLAKE) (STATUS 30) (SUPPLIER RELATION))
(M5! (CITY ATHENS) (S# S5) (SNAME ADAMS) (STATUS 30) (SUPPLIER RELATION))
(M3! M5!)
 CPU time : 0.08
```

If we want a new permanent relation, say `supplier2`, to be this selection from the SUPPLIER relation, we could do:

```
* (define supplier2)
(SUPPLIER2)
 CPU time : 0.03

* (describe
   (dbAssertVirtual
    (dbproject (find supplier relation city (paris athens) status 30)
               s\# sname status city)
    (supplier2 relation)))
(M17! (CITY PARIS) (S# S3) (SNAME BLAKE) (STATUS 30) (SUPPLIER2 RELATION))
(M18! (CITY ATHENS) (S# S5) (SNAME ADAMS) (STATUS 30) (SUPPLIER2 RELATION))
(M17! M18!)
 CPU time : 0.23
```

10.1.3 Join

Join is a database operation that, given two relations, R_1 and R_2, with attributes $a_{11}, \ldots, a_{1n}$ and $a_{21}, \ldots, a_{2m}$, respectively, and an atttibute $a = a_{1i} = a_{2j}$ produces a relation with attributes $a_{11}, \ldots, a_{1n}, a_{21}, \ldots, a_{2j-1}, a_{2j+1}, \ldots, a_{2m}$, and every row, $e_{11}, \ldots, e_{1n}, e_{21}, \ldots, e_{2j-1}, e_{2j+1}, \ldots, e_{2m}$ where $e_{11}, \ldots, e_{1n}$ was a row of R_1, and $e_{21}, \ldots, e_{2j-1}, e_{1i}, e_{2j+1}, \ldots, e_{2m}$ was a row of R_2. For example, Table 10.2 shows the join of the relations in Table 10.1 on the attribute CITY.

This join may be created and displayed by the SNePSUL `dbjoin` command, which, like `dbproject` creates a virtual relation.

```
* (dbjoin city
          (find supplier relation) (s\# sname status city)
          (find project relation) (j\# jname))
(S# (S1) SNAME (SMITH) STATUS (20) CITY (LONDON) J# (J7) JNAME (TAPE))
(S# (S1) SNAME (SMITH) STATUS (20) CITY (LONDON) J# (J5) JNAME (COLLATOR))
(S# (S2) SNAME (JONES) STATUS (10) CITY (PARIS) J# (J1) JNAME (SORTER))
(S# (S3) SNAME (BLAKE) STATUS (30) CITY (PARIS) J# (J1) JNAME (SORTER))
(S# (S4) SNAME (CLARK) STATUS (20) CITY (LONDON) J# (J7) JNAME (TAPE))
(S# (S4) SNAME (CLARK) STATUS (20) CITY (LONDON) J# (J5) JNAME (COLLATOR))
(S# (S5) SNAME (ADAMS) STATUS (30) CITY (ATHENS) J# (J4) JNAME (CONSOLE))
(S# (S5) SNAME (ADAMS) STATUS (30) CITY (ATHENS) J# (J3) JNAME (READER))
 CPU time : 0.35
```

Table 10.2: The join of SUPPLIER and PROJECT on CITY

S#	SNAME	STATUS	CITY	J#	JNAME
s1	Smith	20	London	j5	collator
s1	Smith	20	London	j7	tape
s2	Jones	10	Paris	j1	sorter
s3	Blake	30	Paris	j1	sorter
s4	Clark	20	London	j5	collator
s4	Clark	20	London	j7	tape
s5	Adams	30	Athens	j3	reader
s5	Adams	30	Athens	j4	console

Again, to make the virtual relation permanent, `dbAssertVirtual` is used:

```
* (define supplierproject)
(SUPPLIERPROJECT)
 CPU time : 0.03

* (describe
   (dbAssertVirtual
    (dbjoin city
            (find supplier relation) (s\# sname status city)
            (find project relation) (j\# jname))
    (supplierproject relation)))
(M19! (CITY LONDON) (J# J7) (JNAME TAPE) (S# S1) (SNAME SMITH) (STATUS 20)
      (SUPPLIERPROJECT RELATION))
(M20! (CITY LONDON) (J# J5) (JNAME COLLATOR) (S# S1) (SNAME SMITH) (STATUS 20)
      (SUPPLIERPROJECT RELATION))
(M21! (CITY PARIS) (J# J1) (JNAME SORTER) (S# S2) (SNAME JONES) (STATUS 10)
      (SUPPLIERPROJECT RELATION))
(M22! (CITY PARIS) (J# J1) (JNAME SORTER) (S# S3) (SNAME BLAKE) (STATUS 30)
      (SUPPLIERPROJECT RELATION))
(M23! (CITY LONDON) (J# J7) (JNAME TAPE) (S# S4) (SNAME CLARK) (STATUS 20)
      (SUPPLIERPROJECT RELATION))
(M24! (CITY LONDON) (J# J5) (JNAME COLLATOR) (S# S4) (SNAME CLARK) (STATUS 20)
      (SUPPLIERPROJECT RELATION))
(M25! (CITY ATHENS) (J# J4) (JNAME CONSOLE) (S# S5) (SNAME ADAMS) (STATUS 30)
      (SUPPLIERPROJECT RELATION))
(M26! (CITY ATHENS) (J# J3) (JNAME READER) (S# S5) (SNAME ADAMS) (STATUS 30)
      (SUPPLIERPROJECT RELATION))
(M19! M20! M21! M22! M23! M24! M25! M26!)
 CPU time : 0.92
```

10.2 SNePS as a Network Database

Although SNePS can be treated as a relational database, as shown in the previous section, it is more naturally a network database. For example, to find the names of suppliers with the same status as suppliers in the same city as the sorter project using relational database techniques, one would join the SUPPLIER and PROJECT

relations on CITY, join the result with SUPPLIER again on STATUS, select rows where PROJECT is sorter, and project the result on the SNAME attribute.

However, in SNePSUL, one could just do

```
* (find (sname- status status- city city- jname) sorter)
(ADAMS BLAKE JONES)
 CPU time : 0.02
```

Additional examples of these techniques may be found in the SNePS DBMS demonstration.

10.3 Database Functions

Functions specifically supplied for treating SNePS as a Database Management System are documented in this section. Additional ones may be created using the functions documented in Chapter 5. Note also `innet` and `outnet`, documented in Section 2.4, for saving the database across runs.

`(dbAssertVirtual` *virtualexp* `[([`*relation nodeset*`]*)])`
Evaluates *virtualexp*, which must return a virtual relation (set of flat cable sets), appends the list `([`*relation nodeset*`]*)` to each flat cable set, asserts each resulting flat cable set as a SNePS molecular node, and returns the set of asserted nodes.

`(dbcount` *nodesetexp*`)`
Evaluates the SNePSUL nodeset expression, *nodesetexp*, and returns a node whose identifier looks like the number which is the number of nodes in the resulting set.

`(dbjoin` *relation nodesetexp1 relations1 nodesetexp2 relations2*`)`
A virtual relation (a set of flat cable sets) is created and returned. The virtual relation is formed by taking the nodes returned by the SNePSUL node set expression, *nodesetexp1* and the nodes returned by the SNePSUL node set expression, *nodesetexp2,* joining these two relations on the attribute *relation,* and then projecting the result down the *relations1* attributes from the first nodeset and the *relations2* attributes from the second nodeset. Note that `relations1` and `relations2` is each a list of relations.

`(dbmax` *nodesetexp*`)`
Evaluates the SNePSUL nodeset expression, *nodesetexp*, which must evaluate to a set of nodes all of whose identifiers look like numbers, and returns the node whose identifier looks like the biggest of the numbers.

`(dbmin` *nodesetexp*`)`
Evaluates the SNePSUL nodeset expression, *nodesetexp*, which must evaluate to a set of nodes all of whose identifiers look like numbers, and returns the node whose identifier looks like the smallest of the numbers.

`(dbproject` *nodesetexp relations*`)`
A virtual relation (a set of flat cable sets) is created and returned. The virtual relation is formed by taking the nodes returned by the SNePSUL node set expression, *nodesetexp,* and projecting down the SNePSUL relations included in the sequence, *relations*.

`(dbtot` *nodesetexp*`)`
Evaluates the SNePSUL nodeset expression, *nodesetexp*, which must evaluate to a set of nodes all of whose identifiers look like numbers, and returns a node whose identifier looks like the sum of the numbers.

University at Buffalo Public License ("UBPL") Version 1.0

1. Definitions.

1.0.1. "Commercial Use"
means distribution or otherwise making the Covered Code available to a third party.

1.1. "Contributor"
means each entity that creates or contributes to the creation of Modifications.

1.2. "Contributor Version"
means the combination of the Original Code, prior Modifications used by a Contributor, and the Modifications made by that particular Contributor.

1.3. "Covered Code"
means the Original Code or Modifications or the combination of the Original Code and Modifications, in each case including portions thereof.

1.4. "Electronic Distribution Mechanism"
means a mechanism generally accepted in the software development community for the electronic transfer of data.

1.5. "Executable"
means Covered Code in any form other than Source Code.

1.6. "Initial Developer"
means the individual or entity identified as the Initial Developer in the Source Code notice required by Exhibit A.

1.7. "Larger Work"
means a work which combines Covered Code or portions thereof with code not governed by the terms of this License.

1.8. "License"
means this document.

1.8.1. "Licensable"
means having the right to grant, to the maximum extent possible, whether at the time of the initial grant or subsequently acquired, any and all of the rights conveyed herein.

1.9. "Modifications"
means any addition to or deletion from the substance or structure of either the Original Code or any previous Modifications. When Covered Code is released as a series of files, a Modification is:

a. Any addition to or deletion from the contents of a file containing Original Code or previous Modifications.

b. Any new file that contains any part of the Original Code or previous Modifications.

1.10. "Original Code"
means Source Code of computer software code which is described in the Source Code notice required by Exhibit A as Original Code, and which, at the time of its release under this License is not already Covered Code governed by this License.

1.10.1. "Patent Claims"
means any patent claim(s), now owned or hereafter acquired, including without limitation, method, process, and apparatus claims, in any patent Licensable by grantor.

1.11. "Source Code"
means the preferred form of the Covered Code for making modifications to it, including all modules it contains, plus any associated interface definition files, scripts used to control compilation and installation of an Executable, or source code differential comparisons against either the Original Code or another well known, available Covered Code of the Contributor's choice. The Source Code can be in a compressed or archival form, provided the appropriate decompression or de-archiving software is widely available for no charge.

1.12. "You" (or "Your")
means an individual or a legal entity exercising rights under, and complying with all of the terms of, this License or a future version of this License issued under Section 6.1. For legal entities, "You" includes any entity which controls, is controlled by, or is under common control with You. For purposes of this definition, "control" means (a) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (b) ownership of more than fifty percent (50%) of the outstanding shares or beneficial ownership of such entity.

2. Source Code License.

2.1. The Initial Developer Grant.

The Initial Developer hereby grants You a world-wide, royalty-free, non-exclusive license, subject to third party intellectual property claims:

a. under intellectual property rights (other than patent or trademark) Licensable by Initial Developer to use, reproduce, modify, display, perform, sublicense and distribute the Original Code (or portions thereof) with or without Modifications, and/or as part of a Larger Work; and

b. under Patents Claims infringed by the making, using or selling of Original Code, to make, have made, use, practice, sell, and offer for sale, and/or otherwise dispose of the Original Code (or portions thereof).

c. the licenses granted in this Section 2.1 (a) and (b) are effective on the date Initial Developer first distributes Original Code under the terms of this License.

d. Notwithstanding Section 2.1 (b) above, no patent license is granted: 1) for code that You delete from the Original Code; 2) separate from the Original Code; or 3) for infringements caused by: i) the modification of the Original Code or ii) the combination of the Original Code with other software or devices.

2.2. Contributor Grant.

Subject to third party intellectual property claims, each Contributor hereby grants You a world-wide, royalty-free, non-exclusive license

a. under intellectual property rights (other than patent or trademark) Licensable by Contributor, to use, reproduce, modify, display, perform, sublicense and distribute the Modifications created by such Contributor (or portions thereof) either on an unmodified basis, with other Modifications, as Covered Code and/or as part of a Larger Work; and

b. under Patent Claims infringed by the making, using, or selling of Modifications made by that Contributor either alone and/or in combination with its Contributor Version (or portions of such combination), to make, use, sell, offer for sale, have made, and/or otherwise dispose of: 1) Modifications made by that Contributor (or portions thereof); and 2) the combination of Modifications made by that Contributor with its Contributor Version (or portions of such combination).

c. the licenses granted in Sections 2.2 (a) and 2.2 (b) are effective on the date Contributor first makes Commercial Use of the Covered Code.

d. Notwithstanding Section 2.2 (b) above, no patent license is granted: 1) for any code that Contributor has deleted from the Contributor Version; 2) separate from the Contributor Version; 3) for infringements caused by: i) third party modifications of Contributor Version or ii) the combination of Modifications made by that Contributor with other software (except as part of the Contributor Version) or other devices; or 4) under Patent Claims infringed by Covered Code in the absence of Modifications made by that Contributor.

3. Distribution Obligations.

3.1. Application of License.

The Modifications which You create or to which You contribute are governed by the terms of this License, including without limitation Section 2.2. The Source Code version of Covered Code may be distributed only under the terms of this License or a future version of this License released under Section 6.1, and You must include a copy of this License with every copy of the Source Code You distribute. You may not offer or impose any terms on any Source Code version that alters or restricts the applicable version of this License or the recipients' rights hereunder. However, You may include an additional document offering the additional rights described in Section 3.5.

3.2. Availability of Source Code.

Any Modification which You create or to which You contribute must be made available in Source Code form under the terms of this License either on the same media as an Executable version or via an accepted Electronic Distribution Mechanism to anyone to whom you made an Executable version available; and if made available via Electronic Distribution Mechanism, must remain available for at least twelve (12) months after the date it initially became available, or at least six (6) months after a subsequent version of that particular Modification has been made available to such recipients. You are responsible for ensuring that the Source Code version remains available even if the Electronic Distribution Mechanism is maintained by a third party.

3.3. Description of Modifications.

You must cause all Covered Code to which You contribute to contain a file documenting the changes You made to create that Covered Code and the date of any change. You must include a prominent statement that the Modification is derived, directly or indirectly, from Original Code provided by the Initial Developer and

including the name of the Initial Developer in (a) the Source Code, and (b) in any notice in an Executable version or related documentation in which You describe the origin or ownership of the Covered Code.

3.4. Intellectual Property Matters

(a) Third Party Claims

If Contributor has knowledge that a license under a third party’s intellectual property rights is required to exercise the rights granted by such Contributor under Sections 2.1 or 2.2, Contributor must include a text file with the Source Code distribution titled “LEGAL” which describes the claim and the party making the claim in sufficient detail that a recipient will know whom to contact. If Contributor obtains such knowledge after the Modification is made available as described in Section 3.2, Contributor shall promptly modify the LEGAL file in all copies Contributor makes available thereafter and shall take other steps (such as notifying appropriate mailing lists or newsgroups) reasonably calculated to inform those who received the Covered Code that new knowledge has been obtained.

(b) Contributor APIs

If Contributor’s Modifications include an application programming interface and Contributor has knowledge of patent licenses which are reasonably necessary to implement that API, Contributor must also include this information in the LEGAL file.

(c) Representations.

Contributor represents that, except as disclosed pursuant to Section 3.4 (a) above, Contributor believes that Contributor’s Modifications are Contributor’s original creation(s) and/or Contributor has sufficient rights to grant the rights conveyed by this License.

3.5. Required Notices.

You must duplicate the notice in Exhibit A in each file of the Source Code. If it is not possible to put such notice in a particular Source Code file due to its structure, then You must include such notice in a location (such as a relevant directory) where a user would be likely to look for such a notice. If You created one or more Modification(s) You may add your name as a Contributor to the notice described in Exhibit A. You must also duplicate this License in any documentation for the Source Code where You describe recipients’ rights or ownership rights relating to Covered Code. You may choose to offer, and to charge a fee for, warranty, support, indemnity or liability obligations to one or more recipients of Covered Code. However, You may do so only on Your own behalf, and not on behalf of the Initial Developer or any Contributor. You must make it absolutely clear than any such warranty, support, indemnity or liability obligation is offered by You alone, and You hereby agree to indemnify the Initial Developer and every Contributor for any liability incurred by the Initial Developer or such Contributor as a result of warranty, support, indemnity or liability terms You offer.

3.6. Distribution of Executable Versions.

You may distribute Covered Code in Executable form only if the requirements of Sections 3.1, 3.2, 3.3, 3.4 and 3.5 have been met for that Covered Code, and if You include a notice stating that the Source Code version of the Covered Code is available under the terms of this License, including a description of how and where You have fulfilled the obligations of Section 3.2. The notice must be conspicuously included in any notice in an Executable version, related documentation or collateral in which You describe recipients’ rights relating to the Covered Code. You may distribute the Executable version of Covered Code or ownership rights under a license of Your choice, which may contain terms different from this License, provided that You are in

compliance with the terms of this License and that the license for the Executable version does not attempt to limit or alter the recipient's rights in the Source Code version from the rights set forth in this License. If You distribute the Executable version under a different license You must make it absolutely clear that any terms which differ from this License are offered by You alone, not by the Initial Developer or any Contributor. You hereby agree to indemnify the Initial Developer and every Contributor for any liability incurred by the Initial Developer or such Contributor as a result of any such terms You offer.

3.7. Larger Works.

You may create a Larger Work by combining Covered Code with other code not governed by the terms of this License and distribute the Larger Work as a single product. In such a case, You must make sure the requirements of this License are fulfilled for the Covered Code.

4. Inability to Comply Due to Statute or Regulation.

If it is impossible for You to comply with any of the terms of this License with respect to some or all of the Covered Code due to statute, judicial order, or regulation then You must: (a) comply with the terms of this License to the maximum extent possible; and (b) describe the limitations and the code they affect. Such description must be included in the LEGAL file described in Section 3.4 and must be included with all distributions of the Source Code. Except to the extent prohibited by statute or regulation, such description must be sufficiently detailed for a recipient of ordinary skill to be able to understand it.

5. Application of this License.

This License applies to code to which the Initial Developer has attached the notice in Exhibit A and to related Covered Code.

6. Versions of the License.

6.1. New Versions

University at Buffalo ("UB") may publish revised and/or new versions of the License from time to time. Each version will be given a distinguishing version number.

6.2. Effect of New Versions

Once Covered Code has been published under a particular version of the License, You may always continue to use it under the terms of that version. You may also choose to use such Covered Code under the terms of any subsequent version of the License published by UB. No one other than UB has the right to modify the terms applicable to Covered Code created under this License.

6.3. Derivative Works

If You create or use a modified version of this License (which you may only do in order to apply it to code which is not already Covered Code governed by this License), You must (a) rename Your license so that the phrases "University at Buffalo", "University at BuffaloPL", "UBPL" or any confusingly similar phrase do not appear in your license (except to note that your license differs from this License) and (b) otherwise make it clear that Your version of the license contains terms which differ from the University at Buffalo Public License. (Filling in the name of the Initial Developer, Original Code or Contributor in the notice described in Exhibit A shall not of themselves be deemed to be modifications of this License.)

6.4. Origin of License

This License is derived from the familiar Mozilla Public License Version 1.1 ("MPL") and differs only in that 1) the title now refers to UB to indicate that UB is the licensor; 2) UB retains the sole right to publish revised versions of this license (See 6.1 and 6.2); 3) Section 6.3 now refers to the phrases "University at Buffalo", "University at BuffaloPL", "UBPL"; 4) the License shall be governed by law provisions of the state of New York and any litigation relating to this License shall be subject to the jurisdiction of the state and federal courts of the State of New York and all parties consent to the exclusive personal jurisdiction of those courts (See 11); and 5) Research Foundation of State University of New York, on behalf of University at Buffalo is cited as the copyright owner of the original code (See Exhibit A).

7. DISCLAIMER OF WARRANTY

COVERED CODE IS PROVIDED UNDER THIS LICENSE ON AN "AS IS" BASIS, WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, WITHOUT LIMITATION, WARRANTIES THAT THE COVERED CODE IS FREE OF DEFECTS, MERCHANTABLE, FIT FOR A PARTICULAR PURPOSE OR NON-INFRINGING. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE COVERED CODE IS WITH YOU. SHOULD ANY COVERED CODE PROVE DEFECTIVE IN ANY RESPECT, YOU (NOT THE INITIAL DEVELOPER OR ANY OTHER CONTRIBUTOR) ASSUME THE COST OF ANY NECESSARY SERVICING, REPAIR OR CORRECTION. THIS DISCLAIMER OF WARRANTY CONSTITUTES AN ESSENTIAL PART OF THIS LICENSE. NO USE OF ANY COVERED CODE IS AUTHORIZED HEREUNDER EXCEPT UNDER THIS DISCLAIMER.

8. Termination

8.1. This License and the rights granted hereunder will terminate automatically if You fail to comply with terms herein and fail to cure such breach within 30 days of becoming aware of the breach. All sublicenses to the Covered Code which are properly granted shall survive any termination of this License. Provisions which, by their nature, must remain in effect beyond the termination of this License shall survive.

8.2. If You initiate litigation by asserting a patent infringement claim (excluding declatory judgment actions) against Initial Developer or a Contributor (the Initial Developer or Contributor against whom You file such action is referred to as "Participant") alleging that:

(a) such Participant's Contributor Version directly or indirectly infringes any patent, then any and all rights granted by such Participant to You under Sections 2.1 and/or 2.2 of this License shall, upon 60 days notice from Participant terminate prospectively, unless if within 60 days after receipt of notice You either: (i) agree in writing to pay Participant a mutually agreeable reasonable royalty for Your past and future use of Modifications made by such Participant, or (ii) withdraw Your litigation claim with respect to the Contributor Version against such Participant. If within 60 days of notice, a reasonable royalty and payment arrangement are not mutually agreed upon in writing by the parties or the litigation claim is not withdrawn, the rights granted by Participant to You under Sections 2.1 and/or 2.2 automatically terminate at the expiration of the 60 day notice period specified above.

(b) any software, hardware, or device, other than such Participant's Contributor Version, directly or indirectly infringes any patent, then any rights granted to You by such Participant under Sections 2.1(b) and 2.2(b) are revoked effective as of the date You first made, used, sold, distributed, or had made, Modifications made by that Participant.

8.3. If You assert a patent infringement claim against Participant alleging that such Participant's Contributor Version directly or indirectly infringes any patent where such claim is resolved (such as by license or settlement) prior to the initiation of patent infringement litigation, then the reasonable value of the licenses granted by such Participant under Sections 2.1 or 2.2 shall be taken into account in determining the amount or value of any payment or license.

8.4. In the event of termination under Sections 8.1 or 8.2 above, all end user license agreements (excluding distributors and resellers) which have been validly granted by You or any distributor hereunder prior to termination shall survive termination.

9. LIMITATION OF LIABILITY

UNDER NO CIRCUMSTANCES AND UNDER NO LEGAL THEORY, WHETHER TORT (INCLUDING NEGLIGENCE), CONTRACT, OR OTHERWISE, SHALL YOU, THE INITIAL DEVELOPER, ANY OTHER CONTRIBUTOR, OR ANY DISTRIBUTOR OF COVERED CODE, OR ANY SUPPLIER OF ANY OF SUCH PARTIES, BE LIABLE TO ANY PERSON FOR ANY INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY CHARACTER INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF GOODWILL, WORK STOPPAGE, COMPUTER FAILURE OR MALFUNCTION, OR ANY AND ALL OTHER COMMERCIAL DAMAGES OR LOSSES, EVEN IF SUCH PARTY SHALL HAVE BEEN INFORMED OF THE POSSIBILITY OF SUCH DAMAGES. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO LIABILITY FOR DEATH OR PERSONAL INJURY RESULTING FROM SUCH PARTY'S NEGLIGENCE TO THE EXTENT APPLICABLE LAW PROHIBITS SUCH LIMITATION. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THIS EXCLUSION AND LIMITATION MAY NOT APPLY TO YOU.

10. U.S. government end users

The Covered Code is a "commercial item," as that term is defined in 48 C.F.R. 2.101 (Oct. 1995), consisting of "commercial computer software" and "commercial computer software documentation," as such terms are used in 48 C.F.R. 12.212 (Sept. 1995). Consistent with 48 C.F.R. 12.212 and 48 C.F.R. 227.7202-1 through 227.7202-4 (June 1995), all U.S. Government End Users acquire Covered Code with only those rights set forth herein.

11. Miscellaneous

This License represents the complete agreement concerning subject matter hereof. If any provision of this License is held to be unenforceable, such provision shall be reformed only to the extent necessary to make it enforceable. This License shall be governed by law provisions of the state of New York (except to the extent applicable law, if any, provides otherwise), excluding its conflict-of-law provisions. With respect to disputes in which at least one party is a citizen of, or an entity chartered or registered to do business in the United States of America, any litigation relating to this License shall be subject to the jurisdiction of the state and federal courts of the State of New York and all parties consent to the exclusive personal jurisdiction of those courts, with the losing party responsible for costs, including without limitation, court costs and reasonable attorneys' fees and expenses. The application of the United Nations Convention on Contracts for the International Sale of Goods is expressly excluded. Any law or regulation which provides that the language of a contract shall be construed against the drafter shall not apply to this License.

12. Responsibility for claims

As between Initial Developer and the Contributors, each party is responsible for claims and damages arising, directly or indirectly, out of its utilization of rights under this License and You agree to work with Initial Developer and Contributors to distribute such responsibility on an equitable basis. Nothing herein is intended or shall be deemed to constitute any admission of liability.

13. Multiple-licensed code

Initial Developer may designate portions of the Covered Code as “Multiple-Licensed”. “Multiple-Licensed” means that the Initial Developer permits you to utilize portions of the Covered Code under Your choice of the UBPL or the alternative licenses, if any, specified by the Initial Developer in the file described in Exhibit A.

Exhibit A - University at Buffalo Public License.

The contents of this file are subject to the University at Buffalo Public License Version 1.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at `http://www.cse.buffalo.edu/sneps/Downloads/ubpl.pdf`.

Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License.

The Original Code is SNePS 2.7.

The Initial Developer of the Original Code is Research Foundation of State University of New York, on behalf of University at Buffalo.

Portions created by the Initial Developer are Copyright (C) 2007 Research Foundation of State University of New York, on behalf of University at Buffalo. All Rights Reserved.

Contributor(s):____________________.

NOTE: The text of this Exhibit A may differ slightly from the text of the notices in the Source Code files of the Original Code. You should use the text of this Exhibit A rather than the text found in the Original Code Source Code for Your Modifications.

Index

!, 5, 6, 14
*, 6, 15
+, 2, 15
-, 10, 15
=, 6, 15
>, 16
?, 6, 17
#, 5, 6, 14
#!, 45
$, 5, 6, 14
%, 56
&, 2, 15
_, 15
^ , 8, 15, 56
^ ^ , 8, 56

achieve, 37, 65
act-effect, 41
act-plan, 34
act-precondition, 38
activate, 2, 14, 56
activate!, 2, 56
active connection graph, 56, 60
ActPlan, 66
add, 14
add-to-context, 13, 56
all-hyps, 7
and, 12
apply-function-to-ns, 45
arc labels, 10
ask, 2, 56, 67, 68
askifnot, 2, 56, 67, 68
askwh, 2, 56, 67, 68
askwhnot, 2, 56, 67, 68
assert, 5, 6, 14
assertions
    SNEPSUL variable, 6
parser::atn-read-sentence, 9
atnin, 75
attach-function, 70
attach-primaction, 33
attachedfunction, 69

belief revision, 71
belief space, 71
believe, 2, 27, 65, 71
blieve, 71
bnscommands
    SNEPSUL variable, 6
break-arc, 75
build, 4, 5, 11, 14

snip::*choose-randomly*, 28
cl-user:*use-gui-show*, 3
clear-infer, 15, 56
clear-infer-all, 15
clearkb, 56
commands
    infix, 4
    macro, 5
    postfix, 4
    prefix, 4
    SNEPSUL, 4
commands
    SNEPSUL variable, 6
compose, 12
context, 71
    current, 6
    default, 6
:context, 7
context-specifier, 7
contexts, 5
    extensionally defined, 5
    intensionally defined, 5
    names, 5
    nodes in, 6
    operating on, 13
contradictions, 5
converse, 12
current context, 71
current-configuration, 76

dbAssertVirtual, 102
dbcount, 102
dbjoin, 102

dbmax, 102
dbmin, 102
dbproject, 102
dbtot, 102
dc-lisp, 9
dc-no-pause, 9
dc-pause-help, 9
dc-quit, 9
dc-quit-all, 9
dc-read-pause, 9
dc-set-pause, 9
dc-sneps, 9
dc-snepslog, 9
deduce, 10, 17
deducefalse, 2, 17
deducetrue, 2, 17
deducewh, 2, 17
deducewhnot, 2, 17
default-defaultct, 6
defaultct
  SNEPSUL variable, 6, 7, 13
define, 10
define-attachedfunction, 69
define-frame, 56
define-path, 11, 56
define-primaction, 26
defsnepscom, 49
demo, 8, 57
depthCutoffBack, 23, 64
depthCutoffForward, 23, 64
derived belief, 71
describe, 16
describe-context, 13, 57
describe-terms, 57
disbelieve, 2, 27, 65
do-all, 28, 43, 65
do-one, 28, 43, 65
domain-restrict, 13
dot, 3, 16
dump, 16

Effect, 66
endLispConnection, 68
erase, 14
ev-trace, 23
exception, 12
expert, 57

files
  auxiliary, 8
  compatibility, 9
find, 4, 17
findassert, 4, 17
findbase, 17
findconstant, 17
findpattern, 17
findvariable, 17
fnscommands
  SNEPSUL variable, 6
Franz ACL, 3, 16, 68
full-describe, 16
function, SNEPSUL, 4

getValFromVar, 68
goal-plan, 37
GoalPlan, 66

hypotheses, 5, 6
hypothesis, 71

if-do, 26
ifdo, 65
in-trace, 23
inference
  path-based, 11
  reduction, 10, 11
infertrace, 23
innet, 8
intext, 8
irreflexive-restrict, 12
isConnected, 68

Java-SNePS API, 3, 68
JavaSnepsAPI, 68
JIMI, 3, 16
JUNG, 3, 16

kplus, 12
kstar, 12

lexin, 75
lisp, 7, 57
lisp-list-to-ns, 45
lisp-object-to-node, 45
list-asserted-wffs, 57
list-context-names, 13
list-contexts, 57
list-hypotheses, 13
list-nodes, 15
list-terms, 57
list-wffs, 2, 57
load, 57

multi::print-regs, 24

node-to-lisp-object, 45
nodes
  asserted, 5, 6
  base, 5
  in contexts, 6
  molecular, 5
  pattern, 5
  proposition, 6
  types of, 5
  unasserted, 5, 6
  variable, 5, 59
nodes
  SNEPSUL variable, 6
normal, 58
not, 12
ns-to-lisp-list, 45
nscommands
  SNEPSUL variable, 6

or, 12
outnet, 8

parse, 75
path-based inference, 11
paths
  syntax and semantics, 11
patterns
  SNEPSUL variable, 6
perform, 25, 38, 58
plan-act, 34
plan-goal, 37
plantrace, 38
Precondition, 66
primitive action functions, 26
  table of, 34
primitive actions, 26
primitive acts, 26
procedural attachment, 69
procedure, SNEPSUL, 4

range-restrict, 13
reduction inference, 10, 11
relations, 10
  converse, 10
  initial, 10
relations
  SNEPSUL variable, 6, 10
relative-complement, 12
remove-from-context, 13, 58
resetnet, 15
restriction set, 5
rscommands
  SNEPSUL variable, 6

set-context, 13, 58
set-default-context, 13, 58
set-mode-1, 58
set-mode-2, 58
set-mode-3, 58
show, 2, 3, 16, 58
silent-erase, 14
SNeBR, 71
sneps, 7
snepslog:ask, 67
snepslog:init-java-sneps-connection, 68
snepslog:tell, 67
SNeRE, 25, 65
snif, 28, 65
SNIP, 19, 62
sniterate, 29, 65
snsequence, 28, 65
support set, 59
support set, 57, 72
surface, 16

tell, 67, 68
terminalPunctuation, 59, 60
topcommands
  SNEPSUL variable, 6
trace, 59

unbreak-arc, 76
undefine, 10
undefine-path, 11, 59
undefsnepscoms, 51
unev-trace, 23
unin-trace, 24
Unique Variable Binding Rule, 62
unitpath, 10, 11
unlabeled, 59
untrace, 59

variables
  SNEPSUL, 6
variables
  SNEPSUL variable, 6
varnodes
  SNEPSUL variable, 6
verbize, 84

wff, 60, 61
wffCommand, 59, 60
wffName, 59

unassigned, 60
wffNameCommand, 59
`when-do`, 25
`whendo`, 65
`whenever-do`, 25
`wheneverdo`, 65
`with-snepsul-standard-eval`, 46
`with-snepsul-toplevel-style-eval`, 46
`with-snepsul-trace-eval`, 46
`withall`, 30, 65
`withsome`, 31, 65
`wordize`, 85