SNePS 2.6.1 USER'S MANUAL[1]

Stuart C. Shapiro
and
The SNePS Implementation Group

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

October 6, 2004

[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.; and 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.

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
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

Chapter 1

Introduction

1.1 General

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

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.

The core of SNePS 2 is a system for building nodes in the network, retrieving nodes that have a certain pattern of connectivity to other nodes, and performing certain housekeeping tasks, such as dumping a network to a file or loading a network from a file.

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.

SNeBR, the SNePS Belief Revision system, recognizes when a contradiction exists in the network, and interacts with the user whenever it detects that the user is operating in a contradictory belief space.

SNaLPS (Chapter 8), 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.

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.

XGinseng (Chapter 6) is an X Windows-based graphical editing and display environment for SNePS networks. XGinseng is the best environment to use for preparing diagrams of SNePS networks for inclusion in papers. It can also be used to build and edit SNePS networks.

SNePSLOG (Chapter 7) is a logic programming interface to SNePS, and provides direct access in a predicate logic notation to almost all the facilities provided by SNePSUL.

This manual follows the style of Guy Steele's Common Lisp book, and assumes that the reader is familiar with that book and with Common Lisp.

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 5.3); 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 7.2).

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 7.2); 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 7.3, "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`.

1.3 System Portability

SNePS 2, except for XGinseng, is written entirely in COMMON LISP (as defined in Guy Steele's CLtL-II), hence, every proper implementation of CLtL-II 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

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 echoing them.

`(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 $\ldots P_n$`)` 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 $\ldots P_n$`)` 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 *nodesets*; 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`* * `&key :file :format)`
Displays the network connected to the nodes in the *`nodeset`*s in a graphical form. The function 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`. The `dot` program, which is part of Graphviz, must be downloaded separately from `http://www.research.att.com/sw/tools/graphviz/`.

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

`(xginseng )`
X Windows based facility for creating, editing, and displaying SNePS networks. See Chapter 6.

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 the set of nodes from the nodes that would be returned by `deducetrue` that substitute for the free variables in the specification. `deducewhnot` returns the set of nodes from the nodes that would be returned by `deducefalse` that substitute for the free variables in the specification. 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:

$$\begin{array}{lll}\text{(assert forall} & (x_1,\ldots,x_n) \\ & \text{\&ant} & (R_1(x_1),\ldots,R_n(x_n)) \\ & \text{cq} & (P_1(x_1,\ldots,x_n),\ldots,P_m(x_1,\ldots,x_n)))\end{array}$$

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))
```

3.2 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:`$\{\langle$`whenever, p`$\rangle, \langle$`do, a`$\rangle\}$ or of the form `M:`$\{\langle$`when, p`$\rangle, \langle$`do, a`$\rangle\}$, 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 the negation of ***`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 {⟨condition, *p*⟩, ⟨then, *a*⟩}, or of the form {⟨else, *elseact*⟩}, 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`.

```
^^
--> (define-primaction say (object1)
      "Print the object."
```

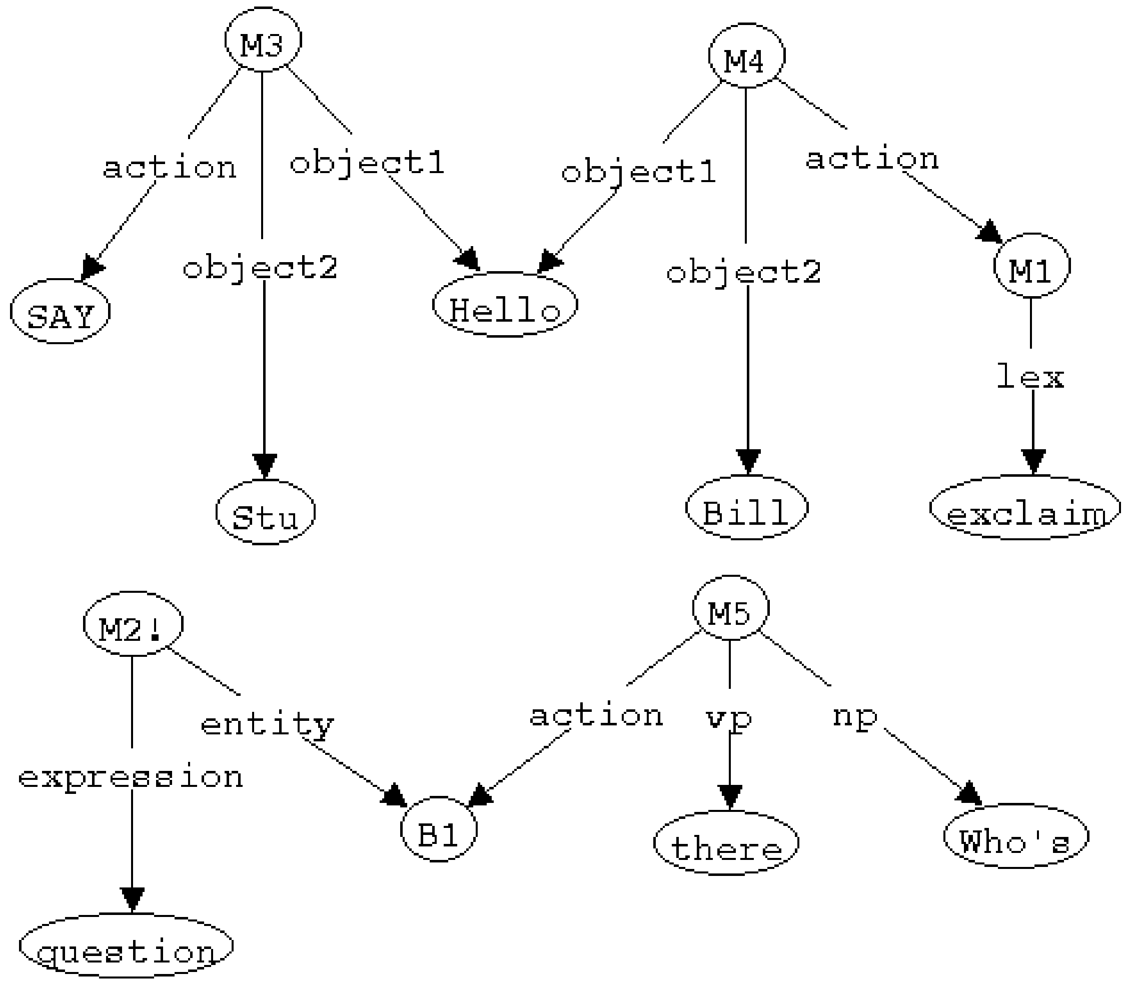

Figure 4.1: Three ways of associating action nodes with action functions. `M3`, `M4`, and `M5` are act nodes, with action nodes `SAY`, `M1`, and `B1` respectively.

```
      (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!:`$\{\langle$`act, a`$\rangle$, $\langle$`precondition, p`$\rangle\}$ 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!:`$\{\langle$`act, ` *a*$\rangle$, $\langle$`effect, ` *e*$\rangle\}$ 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 `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 The Tell-Ask Interface

The Tell-Ask interface, where appropriate, is an even easier way to interface with SNePS from Common Lisp programs than is the with-snepsul reader macro.

`(tell &rest` *strings*`)`
strings must be valid SNePSLOG inputs. (*See* Chapter 7.) `Tell` gives each *string* to the SNePSLOG interpreter; prints what SNePSLOG would print; and returns `NIL`.

`(ask` *string* `&key verbose)`
`(askifnot` *string* `&key verbose)`
`(askwh` *string* `&key verbose)`
`(askwhnot` *string* `&key verbose)`
string must be a valid SNePSLOG input. These functions all give *string* to the SNePSLOG interpreter (*See* Chapter 7.), and return what the SNePSUL interpreter would if given the SNePSUL version of the string. However, `ask` uses `deducetrue`, `askifnot` uses `deducefalse`, `askwh` uses `deducewh`, and `askwhnot` uses `deducewhnot`. If `:verbose` is `T`, the functions print the results as well

as returning them.

5.4 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-forms 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 `build`s (not `assert`s) 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 *command*s. 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

XGinseng: An X Windows Editing and Display Tool

6.1 Introduction

XGinseng is a graphic editing and display tool for SNePS networks that can be run on any system supporting X Windows. Xginseng is built upon the Garnet software package, which implements the interface between the Lisp process and the X server. Garnet is not distributed as part of SNePS, but is available under licence from the Department of Computer Science at Carnegie-Mellon University. For more information about Garnet, contact Brad Myers, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 15213, or Brad.Myers@gem.garnet.cs.cmu.edu.

6.2 Starting XGinseng

Xginseng must be run under X Windows. After starting X, start up a Lisp process and load and start a SNePS session. Once SNePS is started, at any time you may execute the SNePSUL function `(xginseng)`. This function will load all the necessary files and create four XGinseng windows. As each new window is created, its "ghost" will appear under the cursor. Place it where you like on the display screen by clicking on the left mouse button.

Once XGinseng is started, the SNePS session is suspended. Thus only one process — *either* SNePS *or* XGinseng — may be running at one time. Merely moving the cursor from one of the XGinseng windows to the SNePS window will *not* change it to the active window as you would normally expect from an X windows application. Rather, control must be explicitly relinquished by XGinseng via the Pause function (to suspend the Garnet process and return control back to SNePS). See Section 6.4.1 for directions on how to use the Pause function.

6.3 Windows

Upon executing the SNePSUL command `xginseng`, four XGinseng X window will appear on the screen in the following order:

Display Window Where nodes and arcs are displayed and created;

Input Window A window with an easy-to-use panel of buttons for node and arc creation;

Command Window A window containing a panel of buttons to execute the main XGinseng operations;

Dialogue Window A window for receiving input from the keyboard and for displaying messages to the user.

The windows will appear one by one as grid-like "ghosts" with their upper-left corners attached to the mouse. Moving the mouse allows you to move the ghost around on the display screen. Once the ghost is where you whould like the window, click on the left mouse button and the window will be created at that location. Once a window is placed the next one will appear as a ghost under the mouse until all four are on the display. Wait for all four to appear before attempting any actions. Each of these is a standard X window that can be moved, iconified, and resized like any X window. Refer to an X windows manual for help with these operations. *Note: It is very likely that immediately following the creation of the four windows the Lisp process will start garbage collection. This shouldn't take more than a minute or so.*

6.3.1 The Display Window

The Display window, entitled "XGinseng Display", is the window in which XGinseng nodes and arcs are displayed and created. The Display window is really two windows, a small one that is visible on the display screen and a much larger one that is visible only through the smaller one. This window arrangement is rather like using a magnifying glass to read a newspaper. The small window is like the glass and the SNePS networks are displayed on the newspaper. The large window is actually much larger than the display screen, and the small window moves over the large display space revealing the part of the larger space underneath it.

Consequently, the SNePS network you display can be much larger than the X window in which you are viewing it. Recall too, that the small visible window is an X window, which can be resized as you like. Thus you can control how much and what part of the SNePS network is visible at any given time.

Locating the inner window within the outer window

The two scrollbars on the left and bottom of the Display window control the position of the small window relative to the large window. By manipulating the scrollbars you can control the location of the inner window over the large window that contains the network. The left scrollbar controls the vertical position of the inner window while the bottom scrollbar controls its horizontal position. The current location for the inner window is indicated by the position of the white square on the grey strip in the middle of the scrollbar. The grey area represents the height and width of the outer window while the white square indicates where the inner window currently lies within this area. Note that the size of the white square does *not* reflect the relative size of the inner window to the outer.

Manipulating the Scrollbars

The inner window is moved over the large window through manipulation of the two scrollbars. There are three different ways to effect a change to the window. The first two ways use arrows (one a single arrow and the other a double arrow) located at both ends of either scrollbar, as in Figure 6.1. By clicking with the left mouse button over the single arrow you can move the window by a small amount in the indicated direction. For instance, if you click left once over the single arrow at the top of the left scrollbar, the inner window will move *up* a small amount relative to the outer window. Thus the contents of the inner window will move *down* on the screen. You will note that the white square indicator will move upwards on the left scrollbar. The bottom scrollbar will not change. Likewise, clicking left over the double arrow at either end of the scrollbar will move the window by a *page* i.e., the width or height of the inner window. The "pagesize" in this respect will increase or decrease as you resize the window.

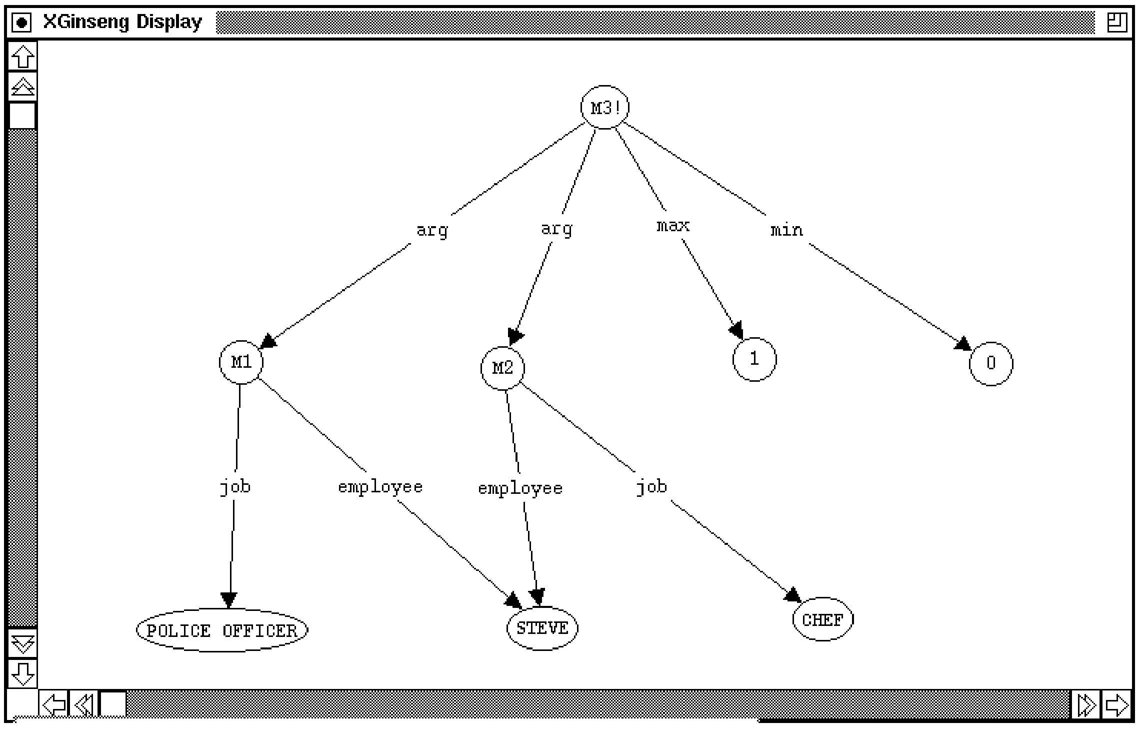

Figure 6.1: The XGinseng Display window. The scrollbars indicate that the inner window is in the top left corner of the large window.

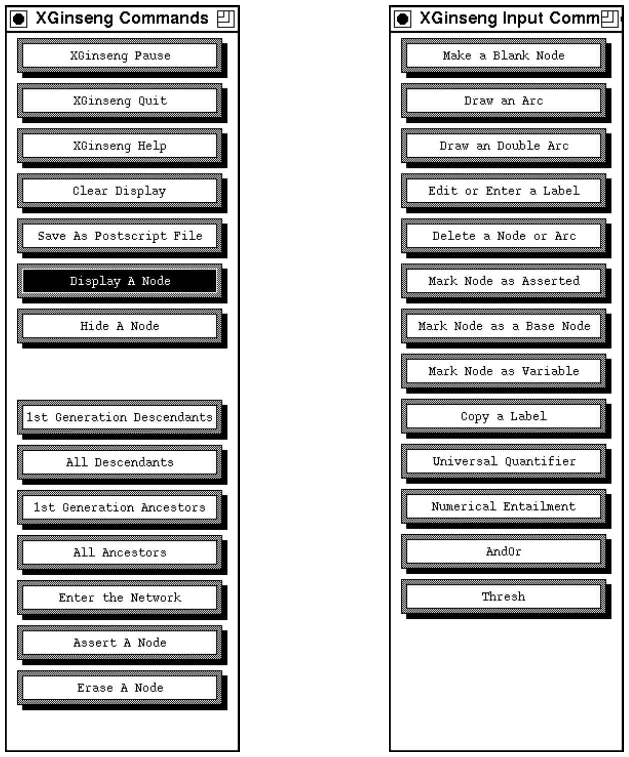

Figure 6.2: The XGinseng Command window (left) and the XGinseng Input Commands window (right). The reverse video indicates the command "Display A Node" is being executed.

The third way to move the window is by dragging the white square indicator along the scrollbar. Click with the left mouse button *and hold* it down while over the white square. Then, as you move the mouse cursor along the scrollbar, you'll notice a black square attached to the cursor. This black outline indicates where the inner window would be placed if you released the button. The white square remains stationary over the small window's current position. The contents of the Display window will not change until you release the left mouse button. Then the inner window will be moved to that position and the scrollbar will be updated.

6.3.2 The Command and Dialogue Box windows

The Command window holds a panel of text buttons, each for a different XGinseng operation. The details of each operation and how to perform it are given below in Section 6.4.1 and also on-line in the Help window. The Dialogue Box displays prompts and messages concerning the success of the attempted operation. Also, when you input a node or file name, your keyboard input is displayed here. Initially, the Dialogue Box is empty. Both the Command and Dialogue Box windows are standard X windows and my be moved, iconified, and resized as desired.

6.4 XGinseng Command Button Operations

There are three types of XGinseng operations available in the Command window button panel:

Operations on XGinseng itself — Functions in this category are Pause, Quit, Help, Clear Display, and Save As Postscript File.

Operations that affect the display of existing SNePS node — These functions are
Display A Node, Hide A Node, 1st Generation Descendants, All Descendants,
1st Generation Ancestors, and All Ancestors

Operations that affect the underlying SNePS network — These functions are
Enter the Network, Assert A Node, and Erase A Node,

Choosing an operation: The XGinseng buttons in the Command window will execute their respective functions when you click with the left mouse button while over the button's rectangle. When a button is chosen it will be displayed in reverse video (white text on a black background) while it is active. Once the operation finishes it will return to its initial display coloring of black text on a white background.

6.4.1 XGinseng Operations

Pause

The Pause command suspends the XGinseng process and returns control to the Lisp process from which XGinseng was called. No windows, nodes, or arcs are destroyed when Pause is called. To restart the XGinseng process execute the SNePSUL function `xginseng` just the way you started the process to begin with. No new windows will be created as long a previous invocation of XGinseng exists.

This function is useful for stopping XGinseng temporarily if you need to re-enter SNePS for any reason.

Quit

The Quit command terminates the XGinseng process and returns control to the Lisp process that originally called it. All windows, including the Help window if it was created, are destroyed.

Help

On-line help is available by clicking with the left mouse button over the Help button in the XGinseng Command window. A fifth X window will appear under the mouse cursor as a "ghost". Place it wherever you like by clicking the the left mouse button.
Organization of the Help window: The Help window is similar to the Display window in that only a part of the Help information is visible at a time. Unlike the Display window, however, the Help window can only be moved up and down with the scrollbar on the left. Refer to the section "Manipulating the Scrollbars" (Section 6.3.1 above) for instruction on how to use the scrollbar to move up and down in the Help text.

The first section of the text in the help window describes the operations controlled by the mouse buttons. The section below describes the functions executed by the buttons in the Command and Input Commands windows. Included are step-by-step instructions on using these functions.
Leaving the Help window: In the top left corner of the Help window is a Quit Help button. Clicking with the left mouse button over this button will destroy the Help window. However, once a Help window is created you may want to keep it around for future use. Remember that the Help window may be iconified or buried like any X window to get it out of the way. Only one Help window may exist at one time. If you try to create a second by clicking on the XGinseng Help button in the Command window, the original iconified window will be deiconified. *Note: At this time the Garnet implementation of the deiconification function does not work with Lucid.* The Help window will be automatically destroyed when you exit XGinseng.

Clear Display

The Clear Display operation destroys all nodes and arcs in the Display window but does not destroy any windows. All nodes and arcs must be recreated from scratch using the Display A Node operation or the various node creation functions (to be described in detail in Section 6.5).

Save As Postscript File

The Save As Postscript File operation allows you to dump the Display window into a Postscript file which may then be included in a document. Only the visible part of the network inside the inner Display window will be included in the file. Since the *entire* window (minus the scrollbars, window title, and border) will be included, be sure to resize the Display window so that only the desired nodes and arcs are visible and that the surrounding white space is minimized. Remember that compact window dumps are much easier to included as Postscript files than files with extraneous space.
Performing a Dump: First, resize the Display window to include only the desired nodes and move aside any obscuring windows. Then click the left mouse button on the Save As Postscript File button in the XGinseng Command window. The following prompt will appear in the dialogue Box:

```
Type the file's name:|
```

The vertical bar is the cursor for the text editor. Type the name of the Postscript file using commands specified in Section 6.4.2 below titled "Displaying New Nodes". When you press the carriage return the mouse cursor will change from its usual appearance to a thin "plus" sign. This cursor is the prompt for the X program `xwd` (for **X** **w**indow **d**ump). Move the cursor over the Display window and click once with the left mouse button. The `xwd` process will beep once when it starts the dump and twice when it finishes. Wait for the Save As Postscript File button to return to its normal coloring before proceeding.
Aborting: To abort the operation, click with the right mouse button over the Dialogue Box at anytime *before* you press the carriage return. The prompt will be removed and the button will be redisplayed in its original coloring.
Including the dump in a LaTeX document: Your Postscript file may be inserted in a LaTeX document by using the `\special` macro. Usually, a Postscript file is included within a `figure` environment. The keywords `hoffset` and `voffset` control the offset of the figure on the page while `hscale` and `vscale` specify scaling factors for the image. Refer to the manual page for `texprint` for complete details.

For example, the following code was used to generate Figure 6.2:

```
        \begin{figure}
        \vskip4in
   \special{psfile=fig2.ps hoffset=36 voffset=-11 hscale=40 vscale=40}
%         \special{psfile=fig2.pdf hoffset=36 voffset=-11 hscale=40 vscale=40}
    \special{psfile=fig2b.ps hoffset=180 voffset=-11 hscale=40 vscale=40}
%    \special{psfile=fig2b.pdf hoffset=180 voffset=-11 hscale=40 vscale=40}
        \caption{{\sf The XGinseng Command window (left) and the XGinseng
       Input Commands  window (right).  The reverse video indicates the command
        ``Display A Node'' is being executed''}\label{fig2}}
        \end{figure}
```

The filename that you typed in the Dialogue Box shoulld be inserted as the argument to `psfile`.
Advisories: The filname that you type in may be any UNIX pathname. If no directory path is specified the file will be created in the directory where the Lisp process was created. Beware that you may inadvertently create window dumps of *any* X window you happen to click on with the `xwd`

"plus" cursor. Take care to click only on the Display window. You may click while the "plus" cursor is on any part of the window, but the cursor will *not* be included in the dump.

6.4.2 Node Operations

Moving Nodes

To move a node, click with the left mouse button over a node and *hold* the button down. As you now move the mouse about the node will stay attached to the mouse wherever you move it over the Display window. All arcs attached to the node will be redrawn as the node moves.

Displaying New Nodes

Nodes can be displayed by clicking left on the Display A Node button in the Command window. Move the mouse cursor to the Dialogue Box window. Remember that this Dialogue Box is an X window so that input from the keyboard will appear only while the mouse cursor is over the window. If you realize that the characters being typed at the keyboard are *not* appearing in the Dialogue Box window, check first that the mouse cursor is over the Dialogue Box.

After the button is selected, the prompt:

```
Type the node's name:|
```

will appear in the Dialogue Box window. The vertical bar is the cursor for the text editor. Type the name of the SNePS node you wish to display and then press the carriage return. For a node named SNEPSNODE you may use any combination of case you'd like, but for a node named Snepsnode (i.e., normally created in SNePSUL by a command like (assert arg "Snepsnode") or (assert arg |Snepsnode|)), you should type "Snepsnode" at the prompt. Note also, that when displaying asserted molecular nodes you don't need to include the trailing "!" at the prompt. It will automatically be displayed when the XGinseng node is created. The carriage return stops the input and sends the current string off to be matched to the set of current node names. *Once you press the carriage return you cannot edit the input string.* If no SNePS node with that name exists a message will be printed in the Dialogue Box informing you that no such SNePS node was found in the current SNePS session.

The standard EMACS editing commands are available before the carriage return is pressed:

`backspace, delete`: delete character before cursor;

`^D`: delete character after cursor;

`^U`: erase the `entire` string;

`^F, right arrow`: move forward a character;

`^B, left arrow`: move back a character;

`^E, end`: move to the end of the string;

`^A, home`: move to the beginning of the string.

If there is a SNePS node with the name you typed both the prompt and the string that you typed will be removed from the Dialogue Box and you will be prompted (by a new message appearing in the Dialogue Box) to choose a point in the Display window where you'd like the node to appear by simultaneously clicking the *left* mouse button down along with the *Shift* key. The new node will then appear in the Display window.
Aborting: To abort the operation, click with the right mouse button over the Dialogue Box at any time *before* you press the carriage return. The prompt will be removed and the button will be redisplayed in its original coloring.

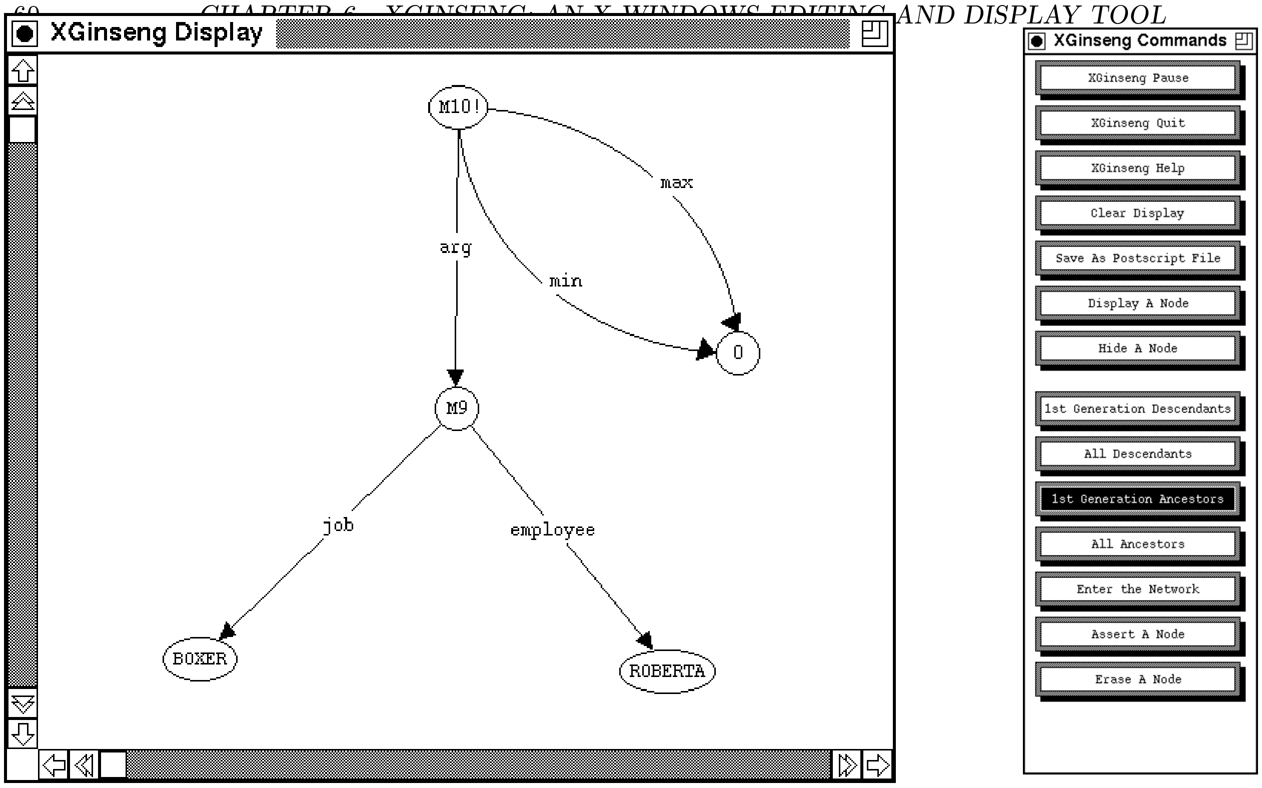

Figure 6.3: Node m9 after having its 1st generation ancestors and descendants displayed.

Describing Nodes

The process of displaying the ancestors or descendants of a given node in the XGinseng display is called "describing the node". Nodes may be described in four different ways, each corresponding to one of the description buttons — 1st Generation Descendants, 1st Generation Ancestors, All Descendants, and All Ancestors. Describing a node's descendants means displaying those nodes dominated by the selected node and drawing all the arcs between them. Similarly, describing a node's ancestors means displaying those nodes that dominate the selected node and drawing the arcs.

The two 1st Generation operations display those nodes that directly dominate (or are dominated by) the selected node. That is, these nodes are at one end of an arc that has the selected node at the other end. For completeness, ancestor nodes are fully described: All of the first generation descendants of an ancestor node are displayed, which ususally means that nodes that do not directly dominate the selected node are also displayed. Suppose that, in Figure 6.3 you selected 1st Generation Ancestors on node `m9`. Then the ancestor node `m10!` is created and displayed, but the node `0` is also displayed for much of the meaning of `m10!` depends on node `0` being visible.

The two All operations recursively call the selected display operation on the nodes they've built. Thus, All Descendants continues downward in the network displaying the descendants of the selected node's descendants, then their descendants, and so on until base nodes (nodes with no descendants) are displayed. Likewise, All Ancestors continues upward in the network describing the ancestors of the ancestor nodes, then their ancestors, until undominated nodes are displayed.

Executing a Describe Command. The four describe commands are invoked in exactly the same way: First you pick the function you want to perform, then you pick the node on which you want to perform it. Thus, first click with the left mouse button on one of the four describe commands

– 1st Generation Descendants, 1st Generation Ancestors, All Descendants, or All Ancestors. The selected button will be displayed in reverse video to highlight your selection. Then click with the *middle* mouse button over the node you want to describe. Wait for the nodes to appear in the Display window — if a large number of nodes need to built this process may take some time. When the selected button returns to its normal coloring the process is complete. If the selected node cannot be described in the chosen manner, then a message will be displayed in the Dialogue Box. For example, in Figure 6.3 node `m10!` happens to be undominated in the current SNePS network. If either 1st Generation Ancestor or All Ancestors is called on `m10!`, the message will indicate that `m10!` has no ancestors.
Aborting a Describe operation. If you click on a describe button (causing it to be displayed in reverse video) and then decide not to describe any node in this fashion, you can abort the operation by clicking with the *right* mouse button over *any* node in the Display window. The button will return to its normal coloring and you may continue with no change in the display.

Hiding Nodes

If a node is not dominated it may be removed from the Display window. In this case the node and all arcs leading out of it are removed from view. Furthermore, if the hiding of an arc leaves a node on the display with no arc pointing to it, that node is hidden as well.

These nodes and arcs are not destroyed; they are merely removed from view, yet they remain in the Display window. They may be redisplayed by executing Display A Node and entering their names at the prompt. When XGinseng realizes that the desired node is already in the Display window, it will again make it visible along with any arcs pointing to or from it, and those nodes at the ends of those arcs. The node and arcs will appear at their last location in the Display window (i.e., their position when Hide A Node was invoked on them). A message will be sent to the Dialogue Box to inform you that the node has been made visible. Since the inner window may not happen to be over the spot where then node was hidden (and so, the node may not appear in the Display window), you may need to scroll the inner window about to find that node. The message is intended to inform you that the node is now visible, but will not tell where it is.
Executing and Aborting a Hide operation: The Hide A Node operation is executed like a describe operation: First click on the Hide A Node button with the left mouse button. Then click on the node that you wish to hide with the *middle* mouse button. If the node cannot be hidden a message will inform you of that fact. Likewise, to abort the operation, click with the *right* mouse button over *any* node. The Hide A Node button will then return to its normal coloring and you may proceed.

6.5 Creating and Editing SNePS Networks with XGinseng

In addition to displaying SNePS nodes that you've created using SNePSUL, you may also create and edit a fresh network using XGinseng. In a nutshell, this process works as follows:

1. You draw/edit a network in the Display window that is just like the SNePS network you want;
2. You signal XGinseng that the network is as you would like it;
3. XGinseng generates a set of SNePSUL commands that correspond to the network, and then executes them;
4. All molecular nodes are relabeled to reflect the naming of their SNePS counterparts. See Section 6.6 for a "before and after" example.

To draw and edit networks you'll need the following buttons in the XGinseng Commands window: Enter the Network, Assert A Node, Erase A Node. You'll also be using all or most of the functions provided in the XGinseng Input Commands window.

6.5.1 Building a Network

The following is a description of the functions available in the XGinseng Input Commands window and instructions on how they can be used to create a network similar to the one in Figure 6.9.

In each case you choose the object(s) you want created or the function you want performed on an object by pressing *left* down over the proper button in the Input Commands window. You then press *right* down over the point in the Display window where you want the action to take place. You'll notice that the button in the Input Commands window remains in reverse video even though the action has been performed. The interface was designed to enable you to perform the same type of action repeatedly without continually having to re-enable the function. Each of these functions can be disabled simply by invoking another function in the button window.

You'll also notice that when you press one of these buttons for the first time a small help message appears in the Dialogue Box telling you what your next step should be, along with a "note" describing a quicker way to call the function without hitting the button. For example, hitting the Make a Blank Node function enables you to produce multiple blank nodes on the display screen until you disable it by calling another function. But you may also create blank nodes on the screen without calling the Make a Blank Node function first—you can simply set the mouse cursor over the point in the Display window where you want the new node to be created and press the *left* mouse button and *Control* key down *simultaneously.* All of the functions available in the XGinseng Input Commands window can be bypassed in a similar manner. Once you become familiar with these alternative ways of invoking these functions you can iconify the XGinseng Input Commands window to keep it out of your way.

All of these functions operate on networks that are not yet associated with an existing SNePS network and have no effect on those networks that have already been submitted to SNePSUL, or those that have been displayed using Display A Node.

Make a Blank Node

This is the function that allows you to create a new blank node anywhere in the Display window. Once this function is enabled by pressing the *left* mouse button over the function box (and the button reverting to the now-familiar reverse video) you may make new nodes all over Display window by pressing the *right* mouse button over the point where you'd like the new node to appear. Within each node is a label that may be edited as you like.

This function may also be called by setting the mouse cursor over the point in the Display window where you want the new node to be created and pressing the *left* mouse button and the *Control* key down *simultaneously.*

The nine nodes shown in Figure 6.8 were created this way.

Draw an Arc

After this function is chosen you can draw arcs between any two existing nodes — be they nodes just created with Make a Blank Node or nodes that are already part of a SNePS network — by clicking *right* down over the *source* node, dragging the mouse (without releasing the button) to the destination node, and releasing. An arc will then appear between the two nodes. You'll notice that the arc has a "break" in the middle of it. This is a label that may be edited as you like.

This function may also be invoked by simultaneously pressing *middle* down and *Control* over the source node, dragging the mouse cursor to the destination node, and releasing.

Draw a Double Arc

This function is very similar to Draw an Arc, but it draws a double arc instead of a regular one. Double arcs consist of two arcs that have the same source and destination nodes.

This function may also be invoked by simultaneously pressing *middle* down, *Shift*, and *Control* over the source node, dragging the mouse cursor to the destination node, and releasing.

Edit or Enter a Label

This function enables you to label the arcs and the nodes in a network that you've just created using the node and arc creation functions described above. Just position the mouse cursor over the label you want to edit and press the *right* mouse button down. A vertical bar will appear indicating where the cursor is located. At this point you you may use simple EMACS commands (see page 59) to enter/edit your text. You may also reposition the cursor in the label by pointing to the point in the label where you'd like it and pressing *left* down.

After you're satisfied with the label, just hit RETURN and editing will be disabled.

This function may also be invoked by simultaneously pressing *right* down and *Control* over the label.

Delete a Node or Arc

If you find that you've created some substructure that you no longer want, you can delete it by calling this function. By pressing *right* down over an arc you'll delete it; if you press it over a node, it and all arcs emanating from it or impinging on it will also be deleted.

This function may also be invoked by simultaneously pressing *right* down, *Shift*, and *Control* over the object to be deleted.

Mark Node as Asserted

If you want a node in the network you're drawing to be an asserted node after the corresponding SNePS network is built, invoke this function on the node. You'll then see a solitary "!" appear in its label. XGinseng will later use ASSERT instead of BUILD when generating the SNePSUL command for this node. Should you decide against your decision of marking a node as asserted you can undo the action by calling the same function again on the node. The displayed "!" will disappear. In fact, you may toggle this function on a node as many times as you wish.

It may also be called by placing the mouse cursor over the node and pressing the `F3` function key on your keyboard.

This function is different from the Assert a Node *found in the* XGinseng Commands button panel. *This function deals solely with nodes that are part of networks that are currently being constructed, and that that have no SNePS counterparts. The other function, in contrast, deals only with nodes that have SNePS counterparts and may effect the underlying SNePS network.*

Mark Node as a Base Node

This function will create a numbered base node and relabel the node to reflect this action.

This function also be called by placing the mouse cursor over the node and pressing the `F2` function key on your keyboard.

If you create a base node and then decide that you'd like it to be an asserted molecular node instead, **delete** *the node and redraw it instead of calling* Mark Node as Asserted *on it. Should you do the latter, the displayed node will "look" asserted, but internally will still be marked as a base node and cause problems.*

Mark Node as a Variable

This function is similar to Mark Node as a Base Node, and the alternative method of invoking it is by placing the mouse cursor over the node and pressing the `F4` function key on your keyboard.

If you create a variable node and then decide that you'd like it to be an asserted molecular node instead, **delete** *the node and redraw it instead of calling* Mark Node as Asserted *on it. Should you do the latter, the displayed node will "look" asserted, but internally will still be marked as a variable node and cause problems.*

Copy a Label

This is a function designed to be purely a convenience to the user rather than being indispensible to the creation of networks. Since a given relation is typed in a number of times during a SNePSUL session—and normally would also have to be typed in more than once during an XGinseng session—a method has provided so the user has to only label one arc with a given relation name and copy the text of this label into any other arcs that are to be labeled with the same relation. In fact, this function may also be used to significantly speed up the labeling of a number of arcs with only slightly different arc labels. For example, if you are to have n relations named REL1 ... RELn, you can label one arc REL, copy the text into the remaining labels, and then use XGinseng's editing facility to append the necessary numerals.

Here's how it's done: After choosing Copy a Label from the panel, position the mouse cursor over the text you want copied and press the *right* mouse button down. This will copy the string into an internal buffer. (Note: you may copy *from* any label, even if the label is located in a node). Then place the mouse cursor over a label *into* which you want the text copied and press the *middle* button down. You may fill in the labels of as many arcs as you want in this manner.

You may also copy text into the internal buffer by placing the mouse cursor over the text and simultaneously pressing the *right* mouse button down along with the *Shift* key. You then copy text from the internal buffer by placing the mouse cursor over the target label and simultaneously pressing the *right* mouse button along with the *Ctrl*, and *Shift* keys.

6.5.2 Case Frames

XGinseng provides four case frames that may be created in the Display window as easily as one creates a node.

Universal Quantifier

By clicking on this button and then pressing *right* down over a point in the Display window you can create a Universal Quantifier case frame at that point (See Figure 6.4). Please be aware that the *bottom* of the frame will be positioned at the point where you clicked. The frame will possess only four nodes and arcs, but you may easily create additional nodes and arcs, and copy labels (Section 6.5.1).

This function may also be called by setting the mouse cursor over a point in the Display window and hitting `F5` *TWICE.*

Numerical Entailment

By clicking on this button and then pressing *right* down over a point in the Display window you can create a Numerical Entailment case frame at that point (See Figure6.5). Please be aware that the *bottom* of the frame will be positioned at the point where you clicked. The frame will possess only four nodes and arcs, but you may easily create additional nodes and arcs, and copy labels (Section 6.5.1).

This function may also be called by setting the mouse cursor over a point in the Display window and hitting `F6` *TWICE.*

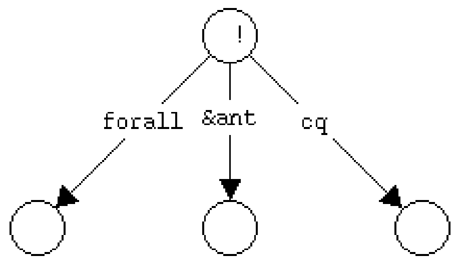

Figure 6.4: A Universal Quantifier case frame.

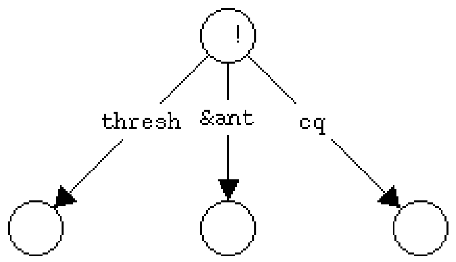

Figure 6.5: A Numerical Entailment case frame.

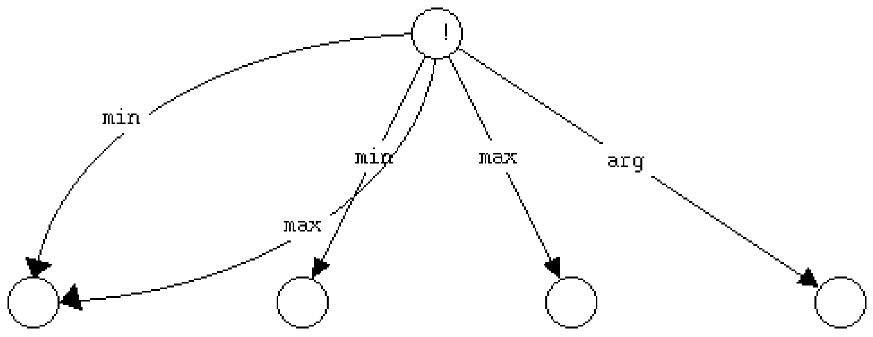

Figure 6.6: An AndOr case frame.

AndOr

By clicking on this button and then pressing *right* down over a point in the Display window you can create a AndOr case frame at that point (See Figure 6.6). Please be aware that the *bottom* of the frame will be positioned at the point where you clicked. The frame will possess only five nodes and arcs, but you may easily create additional nodes and arcs, and copy labels (Section 6.5.1). Note that there are two min-max pairs: This is to allow for the possibility that min and max may be pointing to the same value. Should this be the case in your network, just delete whichever pair is not useful to you.

This function may also be called by setting the mouse cursor over a point in the Display window and hitting `F7` *TWICE.*

Thresh

By clicking on this button and then pressing *right* down over a point in the Display window you can create a Thresh case frame at that point (See Fig. 6.7). Please be aware that the *bottom* of the frame will be positioned at the point where you clicked. The frame will possess only five nodes and arcs, but you may easily create additional nodes and arcs, and copy labels (Section 6.5.1). Note that there are two thresh-threshmax pairs: This is to allow for the possibility that thresh and thresh max may be pointing to the same value. Should this be the case in your network, just delete whichever pair is not useful to you.

This function may also be called by setting the mouse cursor over a point in the Display window and hitting `F8` *TWICE.*

Enter the Network

This is the main function involved in taking a network you've just drawn in the Display window and creating a SNePS network that corresponds to it. When you decide that the network (or subnetwork) you've drawn is complete, call this function by pressing its button on the panel. This function takes no arguments just as XGinseng Quit or XGinseng Pause take no arguments. XGinseng will then examine your network for the following syntax violations: unlabeled nodes or arcs, unasserted top-level nodes, or solitary nodes. If any of these exist, XGinseng will stop at each violation and flash the offending object (and sound a beep as well) to alert the user of the object's whereabouts. The function will at this time abort to allow the user to edit the network and correct the deficiencies. If Enter the Network is called again the same checks are performed, and, assuming the network is in order, an examination of all the arc labels in the networks is begun. Each label is then examined to see if it has already been defined in SNePSUL, and, if it hasn't, the user is asked (in the Lisp

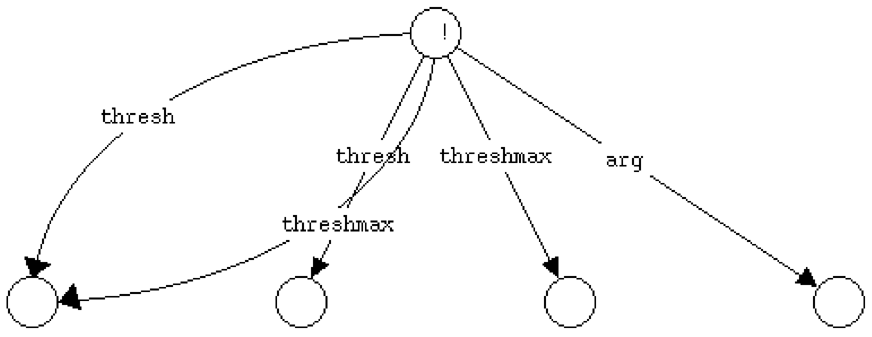

Figure 6.7: A Thresh case frame.

window) if it should. If it is decided that a relation name should at this point be discarded the user should reply `N`. Should this happen the function is once again disabled to allow the user to re-edit the network. If these two tests are passed by the network the next time the function is called, a set of SNePSUL commands is generated and executed by XGinseng and a SNePS network is created as a result. All molecular nodes displayed in the Display window are then relabeled to reflect their proper naming.

Assert a Node

If a SNePS molecular node is unasserted, and you wish to assert it, you may do so by calling this function and then pressing the *middle* button down while the mouse cursor is over the node. The SNePSUL macro **!** will then be called with the node as its argument and any changes to the network will be reflected in the Display window. Note that you may not do anything to a network or a node through XGinseng that you normally cannot do while in the SNePSUL environment since XGinseng executes top-level SNePSUL commands.
Aborting: You may abort the execution of this function by pressing *right* down over any node in the Display window.

This function is different from the Mark Node as Asserted *function found in the* XGinseng Input Commands *button panel. This function deals only with nodes that have SNePS counterparts.*

Erase a Node

This function is called in exactly the same way as Assert a Node described above and it applies SNePSUL's ERASE function to its argument node. Any changes to the underlying SNePS network are reflected in the Display window. If no change is shown in the Display window then the node was not eligible for erasing.
Aborting: You may abort the execution of this function by pressing *right* down over any node in the Display window.

6.6 Example: Creating a SNePS Network using XGinseng

To illustrate how network creation works, Figures 6.8 and 6.9 provide a "before and after" look. There are three things to note about these two figures:

All molecular nodes in Figure 6.8. with the exception of the asserted nodes, are unlabeled. These cannot be labeled until corresponding SNePS nodes are created – as shown in Figure 6.9.

The word `student` in Figure 6.8. is surrounded by double quotes while it appears in lowercase without double quotes in Figure 6.9. This is to allow you to preserve the case of a symbol name in XGinseng just as you would in SNePSUL.

The arc label `expreseed` was intended to be `expressed` instead. As soon as Enter the Network was executed, the following exchange took place in the *Lisp* invocation window:

```
Do you want to define BSTR? (Y or N): y
Do you want to define WORD? (Y or N): y
Do you want to define STRC? (Y or N): y
Do you want to define CAT? (Y or N): y
Do you want to define EXPRESSION? (Y or N): y
Do you want to define EXPRESEED? (Y or N): n
Do you want to define LEXI? (Y or N): y
Do you want to define BEG? (Y or N): y
Do you want to define END? (Y or N): y
Please re-edit your network.
```

6.6

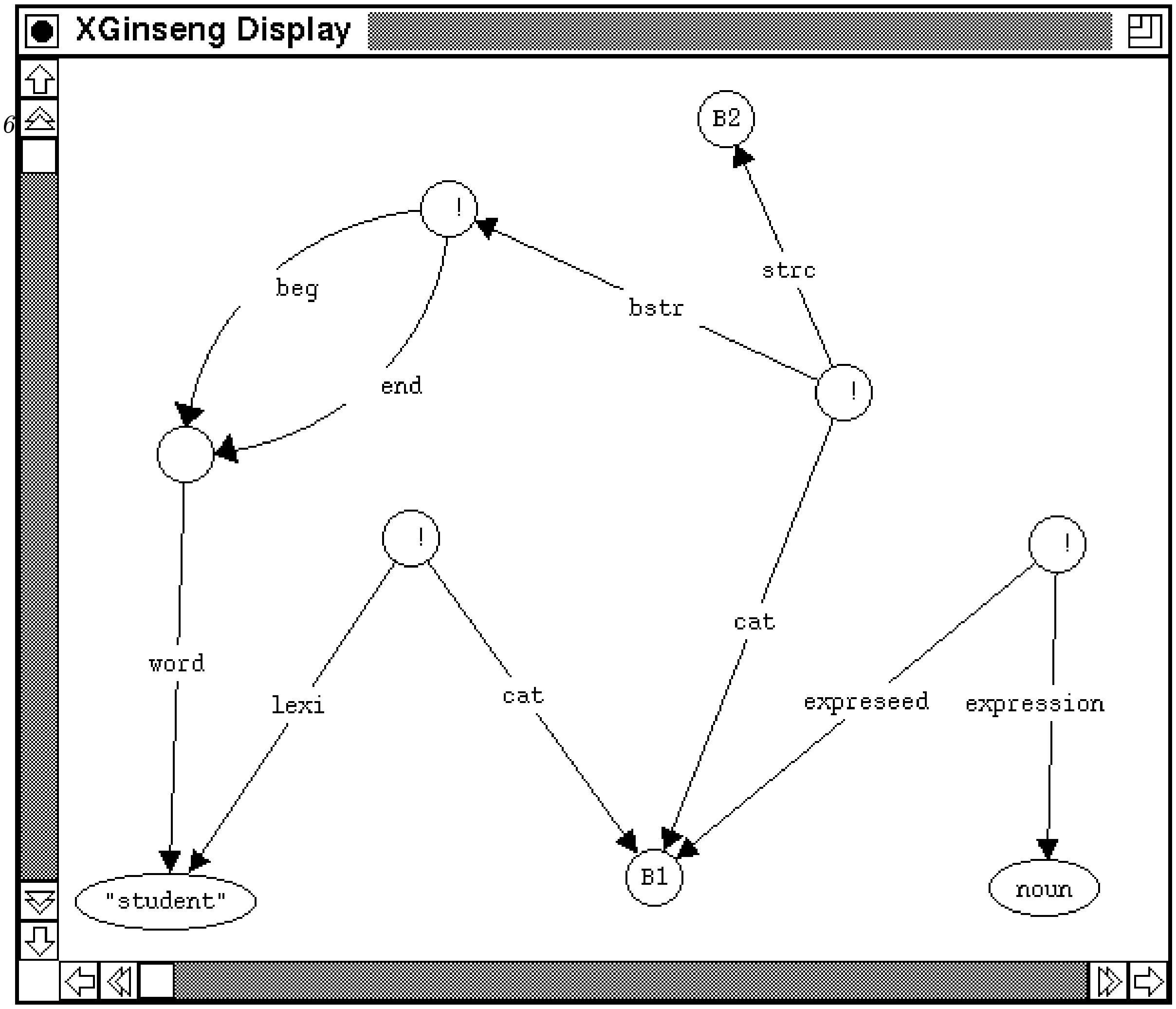

Figure 6.8: An illustration of a newly created network before its corresponding SNePS network has been built.

At this point the offending label was edited using Edit or Enter a Label and Enter the Network was once again executed — yielding the following exchange:

```
Do you want to define EXPRESSED? (Y or N): y
Creating a SNePS network...

(M1! (EXPRESSED B1)
     (EXPRESSION NOUN))

(M4! (BSTR (M3! (BEG (M2 (WORD student)))
                (END (M2))))
     (CAT B1)
     (STRC B2))

(M5! (CAT B1) (LEXI student))
```

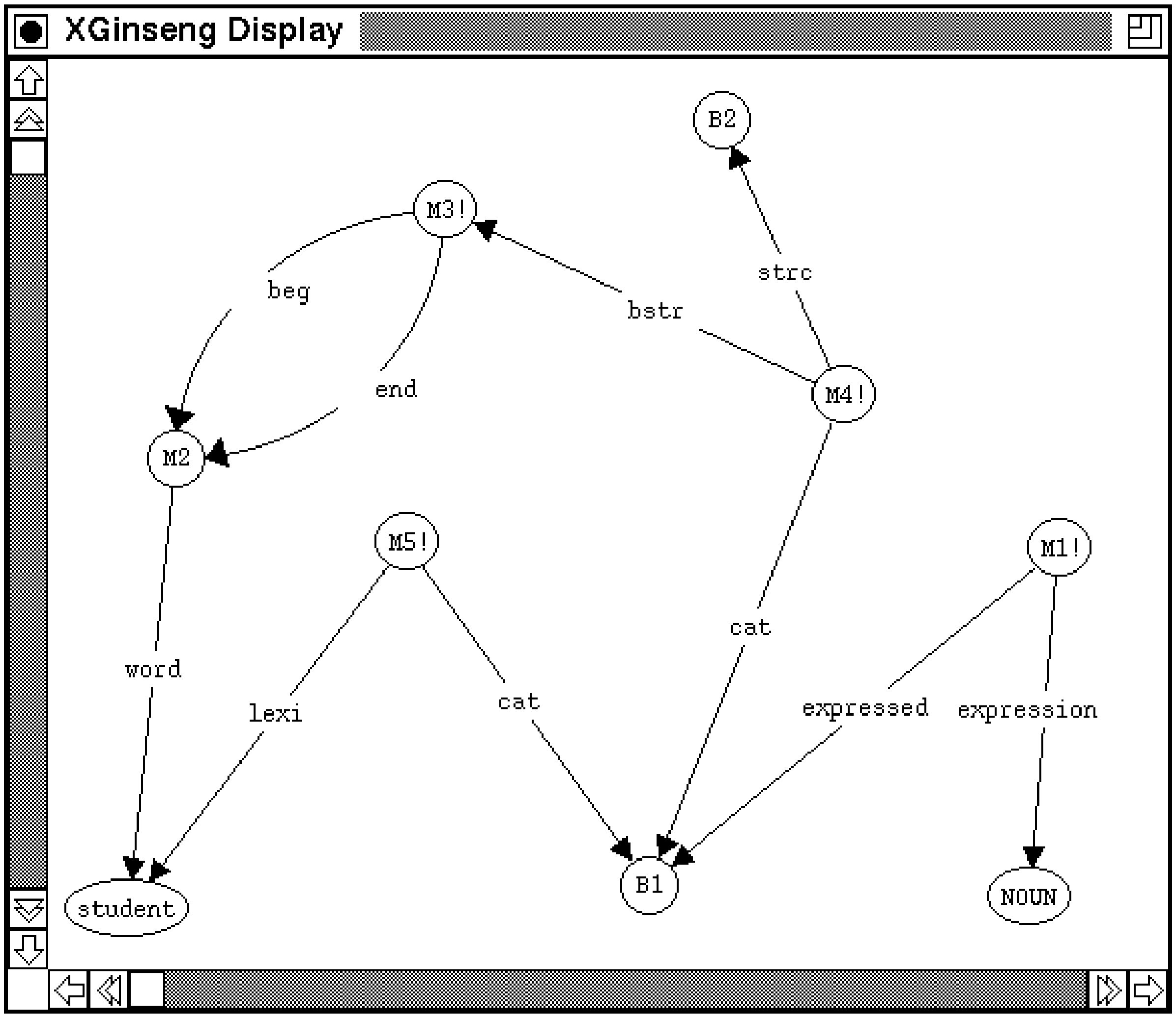

Figure 6.9: The network in Fig 4 after "Enter the Network" has been successfully executed.

6.7 Emergency Exits

Since Garnet and XGinseng are still fairly new, you may run into trouble at times. If your XGinseng display malfunctions so that you cannot execute the Quit operation by clicking on the Command button, you may still exit cleanly by escaping out to the calling Lisp process and executing the XGinseng function call `(xginseng:do-quit nil nil)`. This call will execute the Quit function from the Lisp interpreter.

Chapter 7

SNePSLOG

SNePSLOG is a logic programming interface to SNePS. That is, everything that can be done interactively using SNePSUL can be done interactively using SNePSLOG, just with a different syntax.

To enter SNePSLOG, load SNePS and evaluate

```
(snepslog)
```

To leave SNePSLOG, execute the SNePSLOG command

```
lisp
```

Once in SNePSLOG, the basic commands of `assert`, `add`, `find` and `deduce` are produced by typing a <wff> followed by the proper punctuation. The proper punctuation is:

`assert`: a period (`.`) or nothing.

`add`: an exclamation mark (`!`).

`find`: two question marks (`??`).

`deduce`: one question mark (`?`).

For `find` and `deduce`, free variables are preceded by a question mark (`?`), as they are for `find` in SNePSUL syntax.

The question mark terminating a `deduce` may optionally be followed by a list of one or two integers, `(`*i*`)` or `(`*i j*`)`. If the list is omitted, then deduce continues until no more answers can be derived. If the list is `(0)`, no inference is done—only answers already in the network are returned. If the list is `(`*i*`)`, for a positive *i*, then deduction terminates as soon as possible after *i* positive answers have been found. Otherwise, the list must contain two non-negative integers, `(`*npos nneg*`)`, and deduction terminates as soon as possible after at least *npos* positive and *nneg* negative instances are derived.

The rest of the details of SNePSLOG syntax is in the next section. The following section lists and describes other SNePSLOG commands.

7.1 SNePSLOG syntax

In this section we introduce SNePSLOG syntax. SNePSLOG expressions use the following alphabet:

- *Letters*: All letters from `a` to `z` and `A` to `Z` and digits from `0` to `9`
- *Logical Symbols*: `v=> &=> => <=> and or ~ all exists`

- *Punctuation Symbols*: ( ) { } ,

SNePSLOG syntax is described using the Extended BNF notation.

```
<string> ::= <letter>[<letter>]*

<constant> ::= <string>
<variable> ::= <string>
<function symbol> ::= <string>
<predicate symbol> ::= <string>

<function application> ::= <function symbol> ( <set of terms> [, <set of terms>]*
                           )
<predicate application> ::= <predicate symbol> ( <set of terms or wffs>
                            [, <set of terms or wffs>]* )
<term> ::= <constant> | <variable> | <function application>

<and-entailment> ::= <set of wffs> \&=> <set of wffs>
<or-entailment> ::= <wffs> => <wff>
<or-entailment> ::= <set of wffs> v=> <set of wffs>
<and-or> ::= andor (<integer> , <integer>) <set of wffs>
<and> ::= <wff> and <wff>
<or> ::= <wff> or <wff>
<not> ::= ~ <wff>
<thresh> ::= thresh ( <integer> ) <set of wffs>
<equivalence> ::= <wff> <=> <wff>
<numerically quantified wff> ::= nexists <nexists-parameters> <list of variables> ( <set of wffs> :
                                 <set of wffs> )
<nexists-parameters> ::= (<integer> , <integer> , <integer>) | (<integer> , _ , <integer>) |
                         ( _ , <integer> , _ )


<term or wff> ::= <term> | <wff>
<set of terms or wffs> ::= { <term or wff> [, <term or wff>]*} |
                           <term or wff>
<set of terms> ::= <term> | { <term> [, <term>]*}
<set of wffs> ::= { <wff> [, <wff>]*}
<set of wffs'> ::= <set of wffs> |
                   <wff>
<list of variables> ::= ( <variable> [, <variable>]*)

<wff> ::= <predicate application> |
          <and-entailment> | <or-entailment> |
          <and-or> | <and> | <or> |
          <thresh> | <equivalence> |
          <not> |
          <numerically quantified wff> |
          all <list of variables> (<wff>) |
          exists<list of variables> (<wff>) |
          (<wff>)
```

7.2 SNePSLOG commands

Here we present a list of the SNePSLOG commands, with a description of what they do.

- `%` <SNePSUL expression>
  Evaluates the <SNePSUL expression>, and prints the result.

- `^` &optional <lisp form>
  If <lisp form> is given, prints the result of evaluating <lisp form>. Otherwise, enters a Lisp read-eval-print loop. To leave the loop, type `end` or `^^`.

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

- `activate` <term or wff>
  Builds the <term or wff> and performs the `activate` function on it. That is, forward inference is performed on all asserted propositions that dominate the <term or wff>.

- `activate!` <wff>
  Builds the <wff>, asserts it, and performs the `activate` function on it. That is, forward inference is performed on <wff> and on all asserted propositions that dominate it.

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

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

- `askwh` <wff>
  Performs backward inference on the <wff> and prints the individuals in inferred positive instances that are substituted for the free variables in it.

- `askwhnot` <wff>
  Performs backward inference on the <wff> and prints the individuals in inferred negative instances that are substituted for the free variables in it.

- `add-to-context` <name> <list of hyps>
  Adds the wffs in <list of hyps> to the context named <name>.

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

- `clearkb`
  Clears the knowledge base.

- `define-frame` <P> <(R0 R1 $\cdots$ Rn)>
  If SNePSLOG is in Mode 3, this declares that every proposition of the form $\mathtt{P}(\mathtt{x}_1, \ldots, \mathtt{x}_n)$ is to be represented by a node of the form

$$\{\langle \mathtt{R0}, \{\mathtt{P}\}\rangle, \langle \mathtt{R1}, \{\mathtt{x}_1\}\rangle, \ldots, \langle \mathtt{R}_n, \{\mathtt{x}_n\}\rangle\}$$

  If `R0` is `NIL`, then the node will have no arc pointing to `P`. One predicate name may be associated with at most one set of relations, and one set of relations may be associated with at most one predicate name.

- `demo` &optional <filename> <pause control>
  The input comes from the file specified by <filename>, until the end of file is reached. The input is then reset to the previous input stream. Notice that embebbed demos are allowed. If <filename> is ? or omitted, a menu of possible demonstrations will be printed, and you will be able to choose one of them. If <filename> is an integer, and the menu lists at least that many demonstrations, the one with that number will be run. For the options for <pause control>, see page 8.

- `describe-context` &optional <name>
  Lists the context named <name>. If no argument is given, list the default context.

- `expert`
  SNePSLOG describes the proposition using the full description, including the wff name, `wff`*n*, for some integer *n*. Note that `wff`*n* is the same node referred to in SNePSUL as `m`*n*.

- `lisp`
  Returns to lisp

- `list-asserted-wffs` &optional <name>
  Lists all wffs that are asserted in the context named <name>. If no argument is given, the default context is used.

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

- `normal`
  SNePSLOG prints the propositions using only the SNePSLOG representation of a proposition.

- `perform` <term>
  If SNePSLOG is in Mode 3, the act represented by term is performed. (*See* Chapter 4.)

- `remove-from-context` <name> <list of hyps>
  Removes the wffs in <list of hyps> from the context named <name>.

- `set-context` <name> <list of hyps>
  Defines the context <name> as the <list of hyps>.

- `set-default-context` <context name>
  Makes the context named <context name> to be the default context.

- `set-mode-1`
  The knowledge base is cleared, and SNePSLOG is put into Mode 1. In this mode, every proposition 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 should be used when you want to make deductions using deduction variables in the predicate of a relation. Mode 1 is less efficient than the other modes because more nodes match any given pattern. This is the default mode.

- `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 may only be used when you are not going to use deduction variables in the predicate of a relation. SNePSLOG is more efficient in this mode than in Mode 1, although some flexibility is lost.

- `set-mode-3`
  The knowledge base is cleared, and SNePSLOG is put into Mode 3. In this mode, the user may decide how propositions are represented by using `define-frame`. In this mode, SNePSLOG syntax may be used to build SNePS networks using as flexible a choice of relations as may be done using SNePSUL syntax.

- `trace`
  Traces the `sneps:topsneval` function.

- `untrace`
  Untraces the `sneps:topsneval` function.

7.3 SNeRE in SNePSLOG

7.3.1 SNePSLOG Versions of SNeRE Constructs

With the introduction of Mode 3 in SNePSLOG (*see* p. 77), almost every structure that can be built *via* SNePSUL can be build *via* SNePSLOG. (For some restrictions, *see* §7.3.2.) 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 any proposition that directly contradicts `p` (*see* p. 27) 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.

7.3.2 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
```

Atomic Propositions and Zero-Argument Functions

SNePSLOG syntax does not accept atomic propositions, zero-argument functions, nor zero-argument predicates. One implication of this is that every action must take an argument. For example, the following can be done in SNePSUL:

```
* (resetnet t)
Net reset
 CPU time : 0.00

* ^^
--> ;;; Define a zero-argument action
(define-primaction reportAction ()
   (princ "I'm here."))
reportAction
--> ;;; Attach it
(attach-primaction report reportAction)
t
--> ^^
 CPU time : 0.00

* ;;; Test it
(perform (build action report))
I'm here.
```

A way to perform zero-argument actions In SNePSLOG is to define a caseframe for a dummy act, as shown here:

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

```
...

: ^^
Entering a read/eval/print loop, type ^^ or end to exit

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

: ;;; Define a frame for a dummy do act.
define-frame do (nil action)
do(x1) will be represented by {<action, x1>}

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

Chapter 8

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.

8.1 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 8.3 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 8.4.1 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.

8.2 The Top-Level SNaLPS Loop

8.2.1 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.

8.2.2 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.

8.3 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.

8.3.1 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.

8.3.2 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.

8.3.3 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.

8.3.4 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*.

8.3.5 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.

8.3.6 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.

8.3.7 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.*

8.4 Morphological Analysis and Synthesis

8.4.1 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 8.5 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"`).

8.4.2 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.

8.4.3 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.

8.5 Examples

In this section, we present an example lexicon and two example GATNs, both of which use the one lexicon. The GATN in Section 8.5.1 produces parse trees of acceptable sentences. The one in Section 8.5.2 builds SNePS representations of the information in statements, and answers questions. Both GATNs accept the same fragment of English.

8.5.1 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 8.1.

```
(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.
```

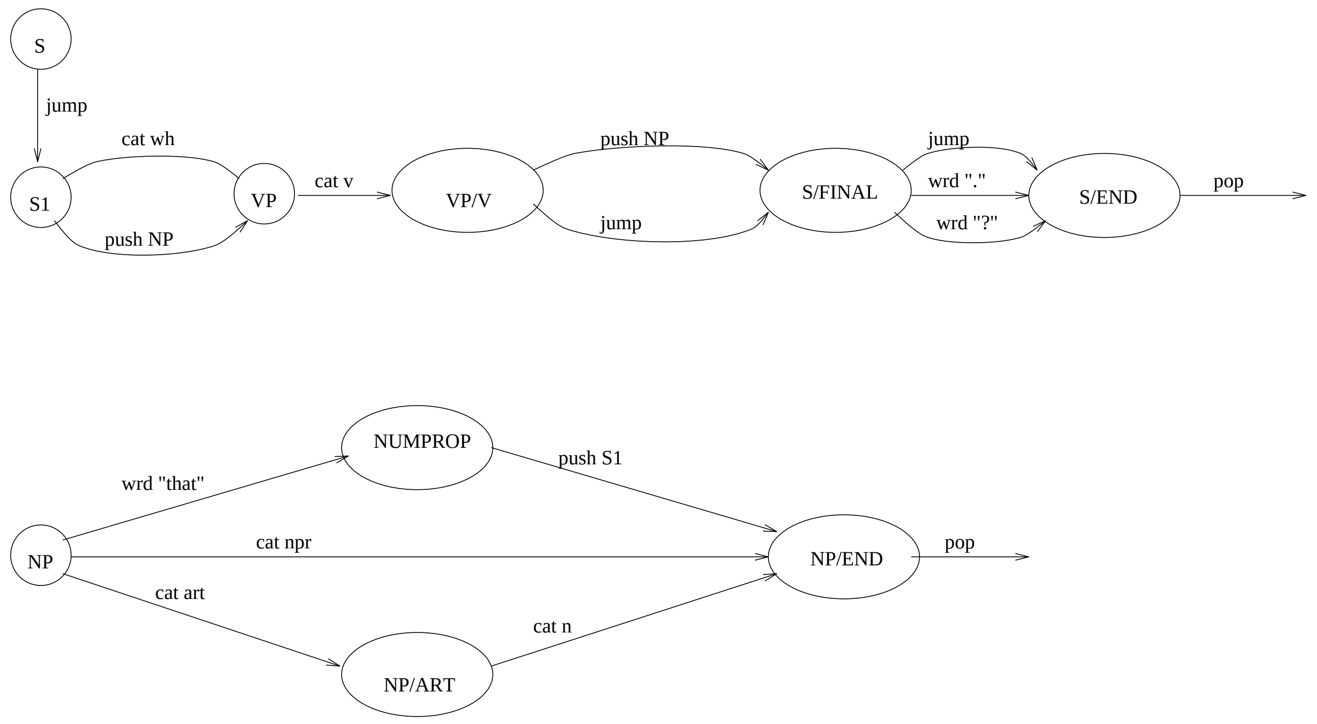

Figure 8.1: Graphical version of the example GATN.

```
(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)))

(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
```

8.5.2 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 8.2. 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.

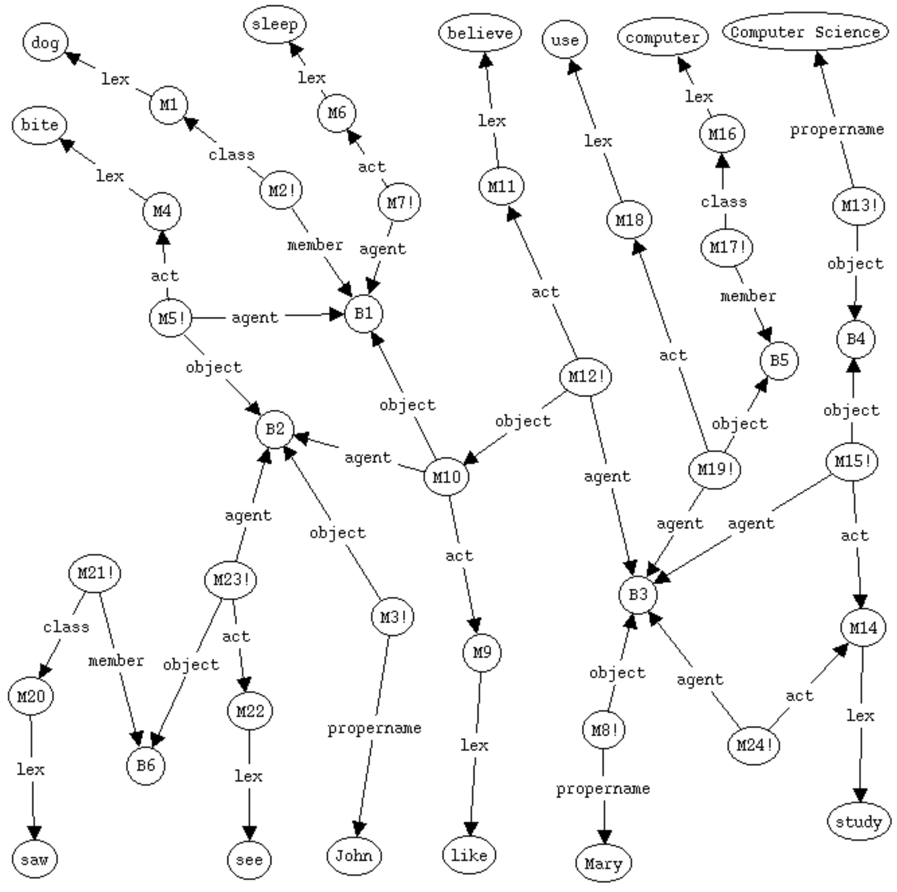

Figure 8.2: SNePS network after running the example.

Chapter 9

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 9.1 shows two relations from the Supplier-Part-Project database of Date[1], p. 114. Figure 9.1

Table 9.1: 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

shows a fragment of the SNePS network version of this database.

9.1 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.

9.1.1 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

[1]C. J. Date, *An Introduction to Database Systems 3rd Edition* (Reading, MA: Addison-Wesley) 1981.

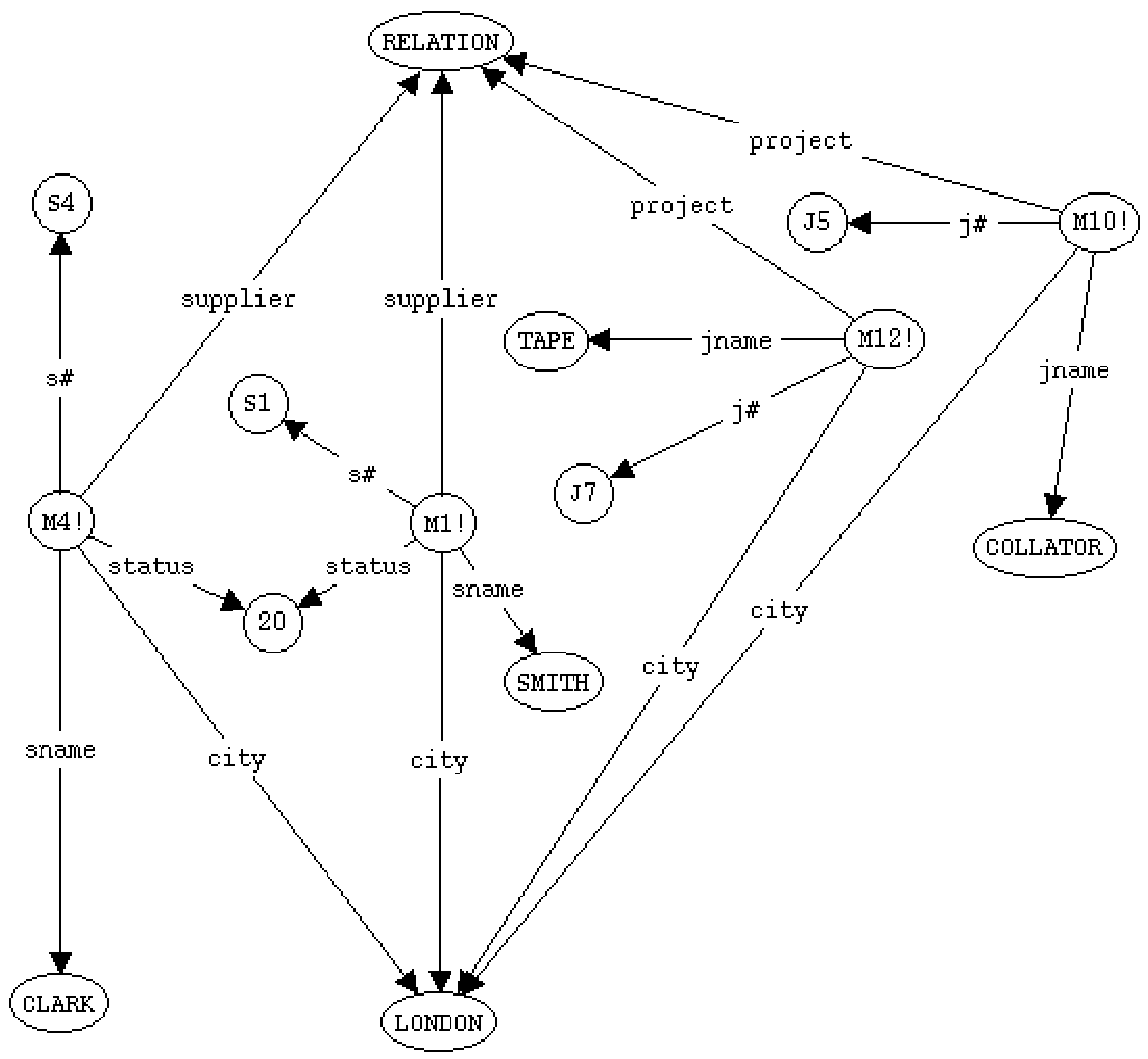

Figure 9.1: Fragment of SNePS network for the Supplier-Part-Project Database

elements were 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
```

9.1.2 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

$$\texttt{(find } R_1 \texttt{ relation } a_{11}\ v_{11}\ \ldots a_{1n}\ v_{1n}\texttt{)}.$$

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
```

9.1.3 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 9.2 shows the join of the relations in Table 9.1 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))
```

Table 9.2: 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

```
(S# (S5) SNAME (ADAMS) STATUS (30) CITY (ATHENS) J# (J3) JNAME (READER))
 CPU time : 0.35
```

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
```

9.2 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.

9.3 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.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and an idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type 'show w'.  This is free software, and you are welcome
to redistribute it under certain conditions; type 'show c'
for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w` and `show c`; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program 'Gnomovision'
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Index