SNePS 3 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

March 25, 2003

[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
Bharat Bhushan
Jong S. Byoun
Alistair E. Campbell
Scott S. Campbell
Hans Chalupsky
Chung M. Chan
Robert G. Church
Joongmin Choi
Chi C. Choy
Soon Ae Chun
Maria R. Cravo
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
Sijun Kang
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
Anthony S. Petre
David R. Pierce
Carlos Pinto-Ferreira
William J. Rapaport
Victor H. Saks
John F. Santore
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 **v**

1 SNePS 3 Concepts **1**
1.1 Terms/Cablesets/Nodes . . . 1
1.2 Syntactic Types . . . 2
1.3 Semantic Types . . . 2
1.4 Relations . . . 3
1.5 Case frames . . . 4
1.6 Contexts . . . 4
1.7 Arbitrary and Indefinite Entities . . . 4
1.8 Categorizations . . . 4
1.9 Inference . . . 4
1.9.1 Wire-Based Inference . . . 4
1.9.2 Path-Based Inference . . . 4
1.9.3 Subsumption Inference . . . 4
1.9.4 Node-Based Inference . . . 4

2 SNePSUL: The SNePS User Language **5**
2.1 Entering and Leaving SNePS . . . 5
2.2 Using Auxiliary Files . . . 5
2.3 Semantic Types . . . 6
2.3.1 Contextually-Determined Types . . . 6
2.4 Relations . . . 7
2.4.1 Path-Based Inference Rules . . . 7
2.5 Operating on Contexts . . . 9
2.6 Case Frames . . . 10
2.7 Building Networks . . . 11
2.7.1 Nodesetspec Syntax . . . 12
2.8 Deleting Information . . . 12
2.9 Functions on Sets of Nodes . . . 13
2.10 Displaying the Network . . . 14
2.11 Retrieving Information . . . 15

3 SNIP: The SNePS Inference Package **17**
3.1 Introduction . . . 17
3.2 Wire-Based Inference . . . 17
3.3 Path-Based Inference . . . 18
3.4 Subsumption Inference . . . 21

4 Transformers **25**

5 SNePS as a Database Management System **27**
5.1 Database Functions 27

GNU GENERAL PUBLIC LICENSE **29**

Bibliography **35**

Index **37**

List of Figures

1.1 SNePS terms shown as a network. 2
1.2 Initial syntactic hierarchy of SNePS 3 nodes. 2
1.3 Initial semantic hierarchy of SNePS 3 nodes. 2

3.1 Path-based inference: `M4!` is inferred from the paths rooted in `M1!` with the help of `M2!` and `M3!`. 18
3.2 Path-based inference: `M47!` is inferred from the paths rooted in `M19!` with the help of `M1!` and `M12!`. 20

Chapter 1

SNePS 3 Concepts

1.1 Terms/Cablesets/Nodes

SNePS 3 is a logic- and network-based knowledge representation, reasoning, and acting system. As a logic-based KR system, SNePS 3 manages a knowledge base that consists of a collection of logical terms. Terms may represent individuals, properties of individuals, categories of individuals, acts (that may be performed), propositions (that may be believed) (Shapiro, 1993), rules, and any other mental entities that agents may conceive of. A term may be atomic or molecular. Atomic terms correspond to individual constants and variables in standard logics, while molecular terms correspond to functional terms.

In standard logics, a functional term is written as a function symbol, followed by its arguments, in linear order, separated by commas, such as *flight*(*US*, *BUF*, *ORD*, *1304*) The role each argument plays in the functional term is determined by the function symbol and the linear position of the argument. So we might provide the intensional semantics of *flight* by saying that *flight*(w, x, y, z) represents the flight of airline w that flies from airport x to airport y and departs at time z. In SNePS, a functional term is written as a cableset, which is a set of cables, where a cable is a pair consisting of a relation and a set of terms, for example

$$\{\langle \texttt{airline, } \{\texttt{US}\}\rangle\ \langle \texttt{from, } \{\texttt{BUF}\}\rangle\ \langle \texttt{to, } \{\texttt{ORD}\}\rangle\ \langle \texttt{departing, } \{\texttt{1304}\}\rangle\}$$

and we would provide the intensional semantics of this construction by saying that $\{\langle \texttt{airline, } \{w\}\rangle\ \langle \texttt{from, } \{x\}\rangle\ \langle \texttt{to, } \{y\}\rangle\ \langle \texttt{departing, } \{z\}\rangle\}$ represents the flight of airline w that flies from airport x to airport y and departs at time z. In SNePS, molecular terms may also represent propositions, so we might say that $\{\langle \texttt{member}, \{\texttt{Fido Rover}\}\rangle\ \langle \texttt{class}, \{\texttt{dog pet}\}\rangle\}$ represents the proposition that Fido and Rover are dogs and pets.

SNePS molecular terms are also assigned identifiers of the form `M`n, where n is an integer. When written as a cableset, the identifier is written first, as `M1:`$\{\langle \texttt{airline, } \{\texttt{US}\}\rangle\ \langle \texttt{from, } \{\texttt{BUF}\}\rangle\ \langle \texttt{to, } \{\texttt{ORD}\}\rangle\ \langle \texttt{departing, } \{\texttt{1304}\}\rangle\}$, or `M2:`$\{\langle \texttt{member, } \{\texttt{Fido Rover}\}\rangle\ \langle \texttt{class, } \{\texttt{dog pet}\}\rangle\}$. The identifier may also be written by itself, as "`M1` represents the US Airlines flight that departs Buffalo Niagara airport at 1304 for O'Hare airport."

As a network-based KR system, a SNePS 3 knowledge base is a network of nodes and labeled, directed arcs. The nodes are the terms, and molecular terms are nodes with outgoing labeled, directed arcs to their argument terms, labeled by their relations. For example, Figure 1.1 shows the terms `M1, M2, US, BUF, ORD, 1304, Fido, Rover, dog`, and `pet` as a network.

Every SNePS 3 node is implemented as a CLOS object whose class is a subclass of a SNePS 3 syntactic type and a SNePS 3 semantic type. Syntactic types are discussed in Section 1.2. Semantic types are discussed in Section 1.3.

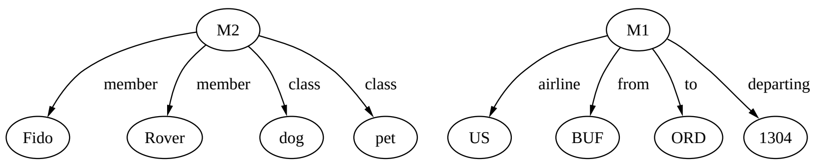

Figure 1.1: SNePS terms shown as a network.

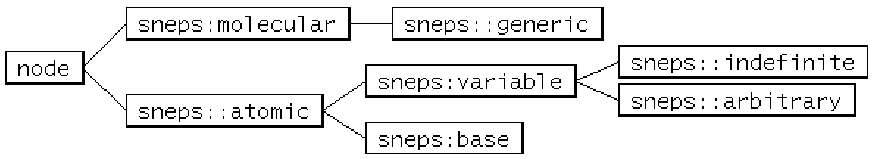

Figure 1.2: Initial syntactic hierarchy of SNePS 3 nodes.

1.2 Syntactic Types

The initial hierarchy of SNePS 3 syntactic types is shown in Figure 1.2. Atomic nodes and molecular nodes are SNePS atomic terms and SNePS molecular terms, respectively, as discussed in Section 1.1. Base nodes serve as individual constants. Variable nodes and generic nodes will be discussed in Section 1.7.

1.3 Semantic Types

The initial hierarchy of SNePS 3 semantic types is shown in Figure 1.3. Proposition nodes are those that represent propositions, as discussed in Section 1.1. Act nodes represent acts that can be performed by the SNePS 3 acting system, which is not yet implemented. Individual nodes are all nodes that are neither proposition nodes nor act nodes. Quantifier and closure nodes will be discussed in Section 1.7. Category and categorization nodes will be discussed in Section 1.8.

The SNePS 3 user may add additional semantic types, as discussed in Section 2.3.

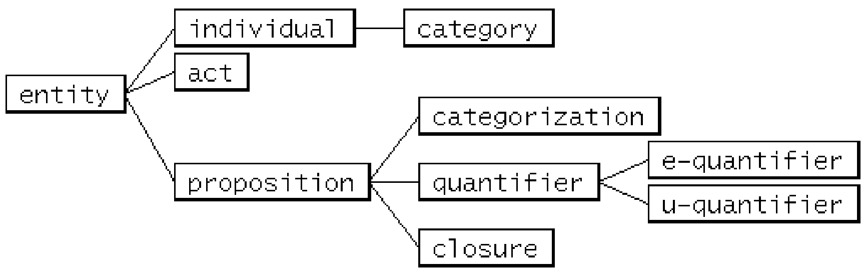

Figure 1.3: Initial semantic hierarchy of SNePS 3 nodes.

1.4 Relations

A SNePS 3 *relation* is a six-tuple, $\langle name, type, adjust, limit, docstring, path\rangle$, where the six constituents are:

***name*:** A symbolic name of the relation. No two relations may have the same name. The relation's name is used in displaying cablesets and networks.

***type*:** The SNePS semantic type of the nodes in the range of this relation, *i.e.,* of the nodes pointed to by arcs labeled with this relation.

***adjust*:** Either `expand`, `reduce`, or `none`. This specifies how wire-based inference (see Section 1.9.1) treats instances of this relation. A value of `none` means that this relation is not subject to wire-based inference.

***limit*:** The minimal size of a nodeset that can be paired with this relation in a cable. If *adjust* is `reduce`, then *limit* is the minimal allowed reduction. If *adjust* is `expand`, then *limit* is the minimal nodeset-size of a cable that allows more wires to be added.

***docstring*:** A documentation string for the relation.

path: A path, whose presence in the net from node n to node m implies the presence of an arc labeled with this relation from n to m.

The initial SNePS 3 relations are:

- For arbitrary and indefinite entities (Section 1.7):

```
<any :type entity :adjust none :limit 1
 "Points from a quantifier node to an arbitrary node.">
<some :type entity :adjust none :limit 1
 "Points from a quantifier node to an indefinite node.">
<restrict :type proposition :adjust none :limit 1
 "Points from a quantifier node to a set of restrictions.">
<depends :type entity :adjust none :limit 0
 "Points from an e-quantifier node to a set of arbitrary nodes.">

<quantifier :type quantifier :adjust none :limit 1
 "Points from a closure node to a quantifier node.">
<scope :type proposition :adjust none :limit 1
 "Points from a closure node to a set of proposition nodes.">
```

- For categorizations (Section 1.8):

```
<member :type entity :adjust reduce :limit 1
 "Points to the entity that is a member of some category."
 :path (compose member (xpath subsumees))>
<class :type category :adjust reduce :limit 1
 "Points to a category that some entity is a member of.">
```

- For subsumption inference (Section 1.9.3):

```
<subsumer :type entity :adjust reduce :limit 1
 "Points to the subsumer of some other node."
 :path (compose subsumer (kstar (compose subsumee- ! subsumer)))>
<subsumee :type entity :adjust reduce :limit 1
 "Points to a node subsumed by some other node."
 :path (compose subsumee (kstar (compose subsumer- ! subsumee)))>

<subsumers :type entity :adjust none :limit 1
 "Virtual arc from a node to its subsumers."
 :path (kstar (or strSubsumer (compose subsumee- ! subsumer)))>
<subsumees :type entity :adjust none :limit 1
 "Virtual arc from a node to its subsumees."
 :path (kstar (or strSubsumee (compose subsumer- ! subsumee)))>
```

1.5 Case frames

1.6 Contexts

1.7 Arbitrary and Indefinite Entities

1.8 Categorizations

1.9 Inference

1.9.1 Wire-Based Inference

1.9.2 Path-Based Inference

1.9.3 Subsumption Inference

1.9.4 Node-Based Inference

Chapter 2

SNePSUL: The SNePS User Language

2.1 Entering and Leaving SNePS

SNePS 3 uses the standard COMMON LISP read-eval-print loop. However, it does use a special readtable. The SNePSUL readtable extends the standard readtable with the following macros:

`#?`*nodeset*	⇒ `(deduce` *nodeset*`)`
`#@`*form*	⇒ `(@` *form*`)`
`#!`*nodeset*	⇒ `(assert` *nodeset*`)`

`#>`*var*
`#>(`*var*`)`
`#>(`*var buildit?*`)`
	⇒ `(=` *var* `(node nil t))`
	⇒ `(=` *var* `(node nil t))`
	⇒ `(=` *var* `(node nil '`*buildit?*`))`

`#C`*cname*	⇒ `(context '`*cname*`)`
`#D`*nodeset*	⇒ `(describe` *nodeset*`)`

`#N`*name*
`#N(`*name*`)`
`#N(`*name buildit?*`)`
	⇒ `(node '`*name*`)`
	⇒ `(node '`*name*`)`
	⇒ `(node '`*name* `'`*buildit?*`)`

`#C`, `#D`, and `#N` may also be written as `#c`, `#d`, and `#n`, respectively.

`(user::sneps )` [*Function*]
Makes the SNePSUL readtable the one to be used, and changes the top level Lisp listener to the `snepsul` package. Unless explicitly noted otherwise, all functions and macros listed in this manual are named by symbols in the `snepsul` package.

`(lisp )` [*Function*]
SNePSUL function that restores the original readtable, and returns the user to the `common-lisp-user` package.

2.2 Using Auxiliary Files

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

`(load file)` [*Function*]
This is the standard COMMON LISP `load` function. It evaluates the forms in the *file*, without echoing them or printing their values.

`(intext file)` [*Function*]
Reads a sequence of forms from the *file*, evaluates them, and prints their values, but doesn't echo them.

`(demo &optional file pause)` [*Function*]
Reads a sequence of forms from the *file*, echoes them, and prints their values. If *pause* is True (the default is `nil`), `demo` will pause after echoing each form, and before evaluating it. If *file* is omitted or `nil` (default), the user will be given a menu of available demos to choose from. The following commands are available at pause points:

`h,?`	Print this help message
`l,^`	Enter Lisp read/eval/print loop
`c`	Continue without pausing
`q`	Quit the demo
`RETURN`	Continue the demo

2.3 Semantic Types

Every SNePS 3 node has a semantic type recognized by the system. The initial taxonomy of semantic types is shown in Figure 1.3. Proposition nodes have an assertional status that is recognized by the inference mechanism. Act nodes may be performed by the SNeRE acting executive. Individual nodes are nodes that are neither proposition nodes nor act nodes. Additional semantic types may be introduced via the macro `defineType`.

`(defineType newtype &rest oldtypes documentation-string)` [*Macro*]
Declares *newtype* to be a new semantic type, and a subtype of all the *oldtypes*. If there are no *oldtypes* given, *newtype* is made a subtype of `individual`. If the *documentation-string* is included, it is put in the documentation slot of *newtype*, and may be retrieved by evaluating `(documentation 'newtype 'type)`.

`(showTypes )` [*Function*]
Displays all the defined semantic types. If `composer` is available, it displays a graph of the semantic types, otherwise, it prints a list of the defined semantic types, top-down, breadth-first.

2.3.1 Contextually-Determined Types

When a node is built, its type is determined both internally and contextually. When a molecular node is built by `build`, its type is determined internally by the type of its case frame (see Sec. 2.6). If the node is constructed within a call to `build`, then build is supposed to construct a wire with some *relation* to the new node, and the type of that node is determined contextually by the type of the *relation* (see Sec. 2.4). If the node is being constructed outside of a `build` call, its contextually determined type defaults to `entity`.

Given an internally determined type t_i and a different contextually determined type t_c for a node about to be built, the actual type of the node is whichever of t_i and t_c is a subtype of the other. If t_i and t_c are the same, that is the type of the new node. However, if t_i and t_c are not the same, and neither is a subtype of the other, an error is raised.

If a node already exists with a semantic type of t_i, and due to being included in the arguments to `build`, its contextually determined type is t_c, then:

- if t_c is a supertype of t_i, or they are the same, then the building continues, and nothing special happens;
- if t_c is a subtype of t_i, then the node's semantic type is *changed* to be t_c;
- otherwise, an error is raised.

2.4 Relations

`(defineRelation` ***name*** `&key :type :adjust :limit :docstring :path)` [*Macro*]
A macro that defines a relation with the given constituents, as long as ***name*** has not already been used for a relation name. The defaults for the keyword arguments are:

`:type`	`entity`
`:adjust`	`reduce`
`:limit`	`1`
`:docstring`	`nil`
`:path`	`nil`

`(list-relations )` [*Function*]
Prints a list of all the SNePS relations.

2.4.1 Path-Based Inference Rules

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.

`(definePath` {***relation path***}*`)` [*Macro*]
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.

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*.

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 ::= `strSubsumee`
If node x structurally subsumes node y, then `strSubsumee` is a path from x to y.

path ::= `strSubsumer`
If node x is structurally subsumed by node y, then `strSubsumer` is a path from x to y.

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 ::= `(xpath` *unitpath*`)`
Note that *unitpath* can be either *relation* or *relation*-. There are four cases:

1. if *unitpath* = *relation* and *relation* has a defined path, p, then `(xpath` *unitpath*`)` = p;
2. if *unitpath* = *relation*− and *relation* has a defined path, p, then `(xpath` *unitpath*`)` = `(converse` p`)`;
3. if *unitpath* = *relation* and *relation* does not have a defined path, then `(xpath` *unitpath*`)` = *relation*;
4. if *unitpath* = *relation*− and *relation* does not have a defined path, then `(xpath` *unitpath*`)` = *relation*-.

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. **Example:** 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.

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 ::= `(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 ::= `(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, and if P is a path from node x to node y and Q is a path from x to any node, then `(domain-restrict (`Q `?)` 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, and if P is a path from node x to node y and Q is a path from y to any node, then `(range-restrict` P `(`Q `?))` is a path from x to y.

path ::= `(`*path**`)`
If P_1 is not one of the path constructors defined above, then `(`P_1 `...` P_{n-1}`)` is equivalent to `(compose` $xp(P_1)$ `...` $xp(P_{n-1})$`)`, where $xp(P_i)$ is `(xpath` P_i`)` if P_i is a unitpath, and P_i otherwise.

2.5 Operating on Contexts

`(allContexts )` [*Function*]
Returns a list of all the contexts that have been created. When SNePS is initialized, there are two contexts, `#CBaseCT` and `#CDefaultCT`, with the former the parent of the latter.

`(assertions &optional (`*context* `(currentContext)))` [*Function*]
Returns the set of nodes which are asserted in the *context* and all its ancestor contexts, namely the union of `(hyps c)` with `(ders c)` for all `c` that are ancestors of *context*, including itself. If *context* is omitted, it defaults to the current context. Note: if *context* is not the current context, the nodes may not be printed with the `!` suffix, even though they are asserted in *context*.

`(context` *cname*`)` [*Function*]
Returns the context whose name is *cname*, or `nil` if there is no context with that name. May be abbreviated as `#C`*cname* or `#c`*cname*.

`(currentContext )` [*Function*]
Returns the current context.

`(ders` *context*`)` [*Function*]
Returns the set of nodes that have been derived in the *context*. Note: if *context* is not the current context, the nodes may not be printed with the `!` suffix, even though they are asserted in *context*.

`(hyps` *context*`)` [*Function*]
Returns the set of nodes that have been asserted as hypotheses in the *context*. Note: if *context* is not the current context, the nodes may not be printed with the `!` suffix, even though they are asserted in *context*.

`(newContext &key` *name parents hyps*`)` [*Function*]
Creates and returns a new context named *name*, whose parent contexts are those whose names are

in the list ***parents***, and with ***hyps*** as its initial set of hypotheses. It is an error if the ***name*** is already the name of a context, or if any member of ***parents*** is not the name of a context. If ***name*** is omitted, a name of the form `C`*n* will be created automatically. If ***hyps*** is omitted it will default to the empty set. If any of the ***hyps*** are not propositions, a warning is issued, and they are not included in the `hyps` of the new context. If there is to be exactly one parent context, ***parents*** may be an atomic name, instead of a list of that name. If ***parents*** is omitted or `nil`, `#CBaseCT` is made the parent of the new context.

`(setCurrentContext` *context* `)` *[Function]*
Changes the current context to be ***context***, and returns it. When SNePS is initialized, the current context is `#CDefaultCT`.

`(showContexts &optional` *path* `)` *[Function]*
Displays the context hierarchy in a graphical form. The function saves a specification of the graph in the DOT language to a file, from which a postscript file is produced via the `dot` compiler. The postscript file is displayed via `gv`. The ***path*** optional argument specifies the base name of the `.dot` and `.ps` files (by default a temporary file).

2.6 Case Frames

A *case frame* is a pair, $\langle c, \rho\rangle$, where c is a SNePS semantic type and ρ is set of relations. Informally, c is the type of all nodes that have the given set of arcs emanating from them. A case frame is to SNePS 3 what a function symbol or predicate symbol is to a more classical logic.

Case frames are partially ordered by the relation *adjustable* ($\sqsupseteq_{adj}$). Case frame $\langle c_1, \rho_1\rangle$ is adjustable to case frame $\langle c_2, \rho_2\rangle$ ($\langle c_1, \rho_1\rangle \sqsupseteq_{adj} \langle c_2, \rho_2\rangle$) iff

1. c_1 is the same, or a subtype of c_2;
2. every relation in $\rho_1 - \rho_2$ is reducible and has *limit*=0;
3. every relation in $\rho_2 - \rho_1$ is expandable and has *limit*=0.

`(defineCaseframe :type` *type* `:docstring` *string* `:arcs` *arc*$^+$`)` *[Macro]*
Defines the case frame $\langle$***type***, $\{$***arc***$^+\}\rangle$, and gives it the documentation string ***string***. Each ***arc*** may be

- the name of a relation that has already been defined.
- a symbol that has not yet been used as the name of a relation, in which case a relation is defined with that name and default constituents.
- a list of the form
(*name* `:type` *type* `:adjust` *adjust* `:limit` *integer* `:docstring` *string* `:path` *path*), in which case a new relation is defined with the given constituents, any of which may be omitted and given its default value.

`(list-caseframes )` *[Function]*
Prints a list of all the defined case frames, top-down, breadth-first.

When SNePS is initialized, either when it is first begun, or after doing `(resetnet t)`, the following caseframes are automatically defined:

```
e-quantifier:{"some is an indefinite node with the given dependents and  restrictions."
              <some :type entity :adjust none :limit 1
               "Points from a quantifier node to an indefinite node.">
              <depends :type entity :adjust none :limit 0
               "Points from an e-quantifier node to a set of arbitrary nodes.">
              <restrict :type proposition :adjust none :limit 1
               "Points from a quantifier node to a set of restrictions.">}
u-quantifier:{"any is an arbitrary node with the given restrictions."
              <any :type entity :adjust none :limit 1
               "Points from a quantifier node to an arbitrary node.">
              <restrict :type proposition :adjust none :limit 1
               "Points from a quantifier node to a set of restrictions.">}
closure:{"The closed proposition that the scope is true
          of the arbitrary or indefinite node described by the quantifier."
         <quantifier :type quantifier :adjust none :limit 1
          "Points from a closure node to a quantifier node.">
         <scope :type proposition :adjust none :limit 1
          "Points from a closure node to a set of proposition nodes.">}
```

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.

We will use the term *wire* to mean a labelled arc and the node it points to. A molecular node *is* a cableset and may be thought of as a set of wires. More informally, a molecular node may be thought of as an object having a set of wires coming out of it.

(any *nodevar* &rest *restrictions*) [*Macro*]
Builds or finds an arbitrary node, *x* satisfying the given *restrictions*, and returns a singleton set containing *x*. By the time $\{x\}$ is returned, the following node will have been built:

$$\{\langle\ \texttt{any},\ \{x\}\rangle\langle\ \texttt{restrict},\ \{\mathit{restrictions}\{x/\mathit{nodevar}\}\}\rangle\}$$

(assert *nodeset* &optional (*context* (currentContext))) [*Function*]
Adds the nodes in *nodeset* as hypotheses in the given intensional context (defaults to the current context), thereby causing them to be asserted in that context. *context* may also be a list of contexts, in which case the nodes in *nodeset* are added as hypotheses in each context in the list.

(build $\{$*relation nodesetspec*$\}^*$) [*Macro*]
A macro that puts a node in the network with an arc labelled *relation* to each node specified by the following *nodesetspec*, and returns a singleton set containing the built node. 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, unless a node with a superset of the node's set of wires has already been built, and that node wb-implies (see Section 3.2) this node in the current context, in which case the built node is asserted. **Note:** where *relation* is specified in the syntax, neither a converse relation nor a non-unit path is allowed.

(close *node*) [*Function*]
Closes the *node* by creating a `closure` node with *node* as the `scope` and the `quantifier` nodes

dominating all variable nodes in `node` as the `quantifier`. If `node` is a set of nodes, closes each of the nodes in the set independently. Returns a set of the `closure` nodes.

`(node` *`name`* `&optional (`*`buildit?`* `nil))` *[Function]*
Finds or creates a node with the name *name*, and returns a set containing the node. However, if *buildit?* is `nil` (default), and the node is not found, returns the empty set.

`(some` *`nodevar depends`* `&rest` *`restrictions`*`)` *[Macro]*
Builds a new indefinite node , x, depending on the arbitrary nodes in the set *depends*, and satisfying the given *restrictions*, and returns a singleton set containing x. By the time $\{x\}$ is returned, the following node will have been built:

$$\{\langle \texttt{ some, } \{x\}\rangle\langle \texttt{ restrict, } \{\textit{restrictions}\{x/\textit{nodevar}\}\}\rangle\langle \texttt{ depends, } \textit{depends}\rangle\}$$

2.7.1 Nodesetspec Syntax

The *nodesetspec* argument in `build` and `find` may be any of the following:

1. `nil`: specifies the empty set;
2. `?`: the don't care symbol specifies any node whatever within a call to `find` (Note: a node with the identifier of `?` can be built—it can later be used in a `find` by referring to it as `#N?`);
3. a symbol, number, or string: specifies a node whose name is the symbol or a symbol looking like the number or string;
4. a list whose first member is `build`, `find`, `assert`, `node`, `=`, or `@`: specifies the set of nodes returned by the function or macro call;
5. a list of *nodesetspec* s, as specified, recursively, by this list: specifies the union of the sets specified by each *nodesetspec* in the list.

2.8 Deleting Information

The commands of this section delete information from the network.

`(erase` *`nodeset`* `&key (silent nil))` *[Function]*
Removes all garbage nodes in the *nodeset* from the network. A node is garbage if it has no arcs coming into it, and is not otherwise needed by other nodes. If `silent` is not `nil`, `erase` will do no printing, and will silently ignore non-garbage nodes in *nodeset*. If `silent` is `nil` (default), `erase` will print a message for every node it does erase, and will call a continuable error for every non-garbage node. Within the continuable error break, the user may choose to skip the non-garbage node, or may make other changes to the net to make the node garbage and then try again to erase it. `erase` is mainly intended for use after mistakes or when debugging.

`(user::kill-sneps )` *[Function]*
Kills all SNePS related packages and global variables. This allows proper reloading of SNePS without having to restart or reboot the lisp environment. This function is intended for SNePS developers and implementers during testing.

`(resetnet` *`[reset-relations?]`*`)` *[Function]*
Reinitializes the network to the state in which no nodes have been built. If *reset-relations?* is `t`,

the set of SNePS relations and caseframes is reset to the pre-defined ones; If ***reset-relations?*** is `nil` (default), the defined relations and caseframes remain as is. Note that even if ***reset-relations?*** is `t`, the SNePS semantic types are not reset.

`(unassert` ***nodeset*** `&optional (`*context* `(currentContext)))` *[Function]*
Removes every node in the ***nodeset*** from being a hypothesis in the given ***context***, or list of contexts. Returns the ***nodeset***. A warning will be issued for each node not a hypothesis in one of the given contexts, and that node will then be ignored.

2.9 Functions on Sets of Nodes

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

For efficiency, nodesets are lazily evaluated. Thus, nodesets may appear in any of the following formats:

$\{\mathtt{n}_1 \ldots \mathtt{n}_k\}$: The set of nodes, $\{\mathtt{n}_1 \ldots \mathtt{n}_k\}$.
$\{\mathtt{U}\ \mathtt{s}_1 \ldots \mathtt{s}_k\}$: The union of the sets, $\mathtt{s}_1, \ldots, \mathtt{s}_k$.
$\{\mathtt{I}\ \mathtt{s}_1 \ldots \mathtt{s}_k\}$: The intersection of the sets, $\mathtt{s}_1, \ldots, \mathtt{s}_k$.
$\{-\ \mathtt{s}_1 \mathtt{s}_2\}$: The set difference, $\mathtt{s}_1 - \mathtt{s}_2$.

To fully evaluate a nodeset, so that it is easier to read, use the function `clean`, documented below.

`(=` *symbol nodesetspec* `)` *[Function]*
A macro that assigns the nodes specified by ***nodesetspec*** (see Sec. 2.7.1) to be the value of the Lisp variable ***symbol***, and returns that set of nodes.

`(@` *form* `)` *[Function]*
Gets the Lisp ***form*** evaluated, coerces the result to a node or set of nodes, and returns that coerced result.

`(+` *nodesets* `)` *[Function]*
Returns the union of the ***nodeset*** s.

`(*` *nodesets* `)` *[Function]*
Returns the intersection of the ***nodeset*** s.

`(-` *nodeset nodeset* `)` *[Function]*
Returns the set of nodes in the first ***nodeset*** but not in the second ***nodeset*** .

`(clean` *nodeset* `)` *[Function]*
Fully evaluates and returns the ***nodeset*** , so that it prints as a simple set.

`(doset (`*var nodeset* `&optional (`*resultform* `nil)) &body` *body* `)` *[Macro]*
Successively binds ***var*** to each node in ***nodeset*** , and executes the forms in ***body*** . If ***resultform*** is provided, its value is returned by `doset`, otherwise `doset` returns `nil`. Any SNePSUL macros, such as `build` or `find`, that occur in ***body*** may refer to the local value of ***var*** with either `(@` *var* `)` or `#@`*var* .

`(empty?` *nodeset* `)` *[Function]*
Returns `t` if the ***nodeset*** is empty; otherwise returns `nil`.

`(everyMember predicate set)` *[Function]*
Returns `t` if every member of the given ***set*** satisfies the given ***predicate***; else returns `nil`.

`(mapset` ***fn nodeset*** `&optional (`***cf*** `#'+))` *[Function]*
Applies the function ***fn*** to every element of ***nodeset***, assuming ***fn*** returns a set, applies the function ***cf*** to the list of the resulting sets, and returns that result.

`(new-set )` *[Function]*
Returns an empty set.

`(singleton` ***node*** `)` *[Function]*
Returns a set of just the single ***node***.

`(someMember predicate set)` *[Function]*
Returns `t` if some member of the given ***set*** satisfies the given ***predicate***; else returns `nil`.

`(subset-suchthat` ***s test*** `)` *[Function]*
Returns the subset of the set ***s*** consisting of every element of ***s*** that satisfies the predicate function ***test***.

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*** `&key (up 1) (down 1))` *[Function]*
Prints each node in the ***nodeset*** with **print-up** set to the value of `up` and **print-down** set to the value of `down`. The entire network can be dumped with `(dump (allNodes))`.

`(describe` ***nodeset*** `)` *[Function]*
Prints the nodes in ***nodeset***, with `*print-down*` set to `t`, and `*print-up*` set to `nil`. ***nodeset*** may be a set of nodes, but may also be a single node, a symbol, or a list. `(describe` ***nodeset*** `)` may also be written as `#D`***nodeset*** or `#d`***nodeset***.

`(list-caseframes )` *[Function]*
Prints a list of all the defined case frames, top-down, breadth-first.

`(list-relations )` *[Function]*
Prints a list of all the SNePS relations.

`*print-down*` *[Variable]*
Controls printing for nodes. If `*print-down*` is `nil` (default), only the node name, possibly with `!` appended, will be printed. If `*print-down*` is `t` the node's name is followed by its down-cableset, and all the nodes therein are printed with their down-cablesets, etc., recursively. If `*print-down*` is a positive integer, i, all nodes within a path length of i below the original node are shown.

`*print-up*` *[Variable]*
If `*print-up*` is `nil` (default), only the node name, possibly with `!` appended, will be printed. If `*print-up*` is `t` the node's name is followed by its up-cableset, and all the nodes therein are printed

with their up-cablesets, etc., recursively. If `*print-up*` is a positive integer, i, all nodes within a path length of i above the original node are shown.

`(show nodes &key up (down t) depth file format)` *[Function]*

Displays the network connected to the *nodes* 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 keyword arguments `:up` and `:down` establish bindings of the SNePS dynamic printer control variables `*print-up*` and `*print-down*`, respectively. The keyword `:depth` can be used to bind both variables to the same value.

`(showContexts &key file format)` *[Function]*
Displays the context hierarchy in a graphical form. The function saves a specification of the graph 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`.

`(showSubsumption &key file format)` *[Function]*
Displays the structural subsumption hierarchy among nodes in a graphical form. The function saves a specification of the graph 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`.

`(showSupports nodeset &key file format)` *[Function]*
Displays the the support structure of a set of nodes in a graphical form. The function saves a specification of the graph 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`.

`(showTypes )` *[Function]*
Displays all the defined semantic types. If `composer` is available, it displays a graph of the semantic types, otherwise, it prints a list of the defined semantic types, top-down, breadth-first.

`(showWithSupports nodes &key up (down t) depth file format)` *[Function]*
Displays the graphical information created by `show` and `showSupports` in one graph.

2.11 Retrieving Information

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

`(allNodes )` *[Function]*
Returns a set of all the nodes in the network.

`(assertions &optional (context (defaultCT)))` *[Function]*

Returns the set of nodes which are asserted in the *context*, namely the union of `(hyps` *context*`)` with `(ders` *context*`)`. If *context* is omitted, it defaults to the current context. Note: if *context* is not the current context, the nodes may not be printed with the `!` suffix, even though they are asserted in *context*.

`(ders` *context*`)` *[Function]*
Returns the set of nodes that have been derived in the *context*. Note: if *context* is not the current context, the nodes may not be printed with the `!` suffix, even though they are asserted in *context*.

`(find` {*path nodesetspec*}* `)` *[Macro]*
A macro that returns the set of nodes such that each node in the set has every specified *path* going from it to at least one node specified by the accompanying *nodesetspec* (see Sec. 2.7.1).

`(hyps` *context*`)` *[Function]*
Returns the set of nodes that have been asserted as hypotheses in the *context*. Note: if *context* is not the current context, the nodes may not be printed with the `!` suffix, even though they are asserted in *context*.

`(deduce` *nodeset* `&optional (`*context* `(currentContext)))` *[Function]*
Uses SNIP and all varieties of inference (see Chapter 3) to back-chain and determine which nodes in the *nodeset* are asserted in the given *context* (defaults to the current context). Returns a set of those nodes.

Chapter 3

SNIP: The SNePS Inference Package

3.1 Introduction

Inference is the process of giving a node that has already been built a status of asserted-as-a-derived-node in some context c. That is, inference is the process of adding a node to the set `(ders c)`, for some context `c`. It is not considered inference to add a node to the set `(hyps c)` via the function `assert`. Neither is it considered inference to find a node with the `find` function, even using path-based inference (see Section 2.4.1), because the assertion status of the found node is not changed by `find`ing it.

There are four varieties of inference in SNePS 3: wire-based inference; path-based inference; node-based inference; and subsumption inference. These are the topics of the following sections.

3.2 Wire-Based Inference

Wire-based inference is an inference method whereby a molecular proposition node m_1 implies a molecular proposition node m_2 with a subset or a superset of m_1's wires. To be more precise, recall that every molecular node is a cableset of the form $\{\langle r_1, ns_1\rangle, \ldots, \langle r_k, ns_k\rangle\}$. So if $m_1 = \{\langle r_{11}, ns_{11}\rangle, \ldots, \langle r_{1k}, ns_{1k}\rangle\}$ and $m_2 = \{\langle r_{21}, ns_{21}\rangle, \ldots, \langle r_{2k}, ns_{2k}\rangle\}$, then $m_1 \vdash_w m_2$ if and only if:

1. the case frame of m_1 is adjustable to the case frame of m_2 (see Section 2.6); and
2. for every relation r_{ij} in both m_1 and m_2, either
   (a) r_{ij} is reducible (Section 2.4) and $ns_{1j} \supseteq ns_{2j}$, or
   (b) r_{ij} is expandable and $ns_{1j} \subseteq ns_{2j}$, or
   (c) $ns_{1j} = ns_{2j}$

Let m_1 and m_2 be molecular proposition nodes such that $m_1 \vdash_w m_2$ and the set of wires of one is a superset of the set of wires of the other. Let m_L be the one whose set of wires is the superset, and m_S be the other. Then, as long as m_L is built before m_S, wire-based inference will be performed automatically. That is, whenever m_1 is asserted in any context (whether before or after m_2 is built), m_2 will also be asserted in the same context (once it is built). Note that automatic wire-based inference will not be done if m_S is built first, nor will it be done if neither set of wires is a superset of the other.

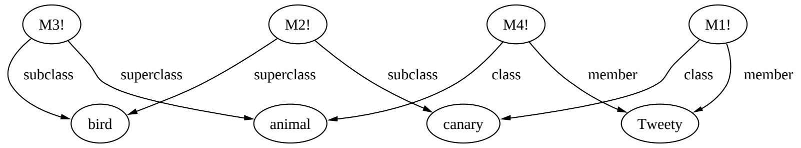

Figure 3.1: Path-based inference: `M4!` is inferred from the paths rooted in `M1!` with the help of `M2!` and `M3!`.

If $m_1 \vdash_w m_2$, but the inference is not carried out automatically, it will still be done by `deduce` (Section 2.11). In fact, `deduce` is sound and complete for wire-based inference.

Note that it is not the case that m_1 wire-base implies m_2 just because m_1 dominates a node, n_1 say, that is adjustable by rules 1–2 to a node, n_2 say, that is dominated by m_2. For example, even though *"John gives a book to Mary."* might wb-imply *"John gives a book."*, it would not therefore be the case that *"'John gives a book to Mary.' is about Mary."* should wb-imply *"'John gives a book.' is about Mary."*.

3.3 Path-Based Inference

Path-based inference is an inference method whereby a molecular proposition node m_1 is inferred from a molecular proposition node m_2 and some paths that emerge from m_2. Each wire descending from m_1 must be a wire actually descending from m_2 or a wire inferred as descending from m_2 according to a path-based inference rule, as described in Section 2.4.1. A path-based inference rule that includes one or more "`!`" symbols requires that the nodes at the corresponding path positions be asserted in order for m_1 to be inferred. If $m_3, \ldots, m_k$ are such nodes for all the paths descending from m_2, it is proper to say that m_1 is inferred from all the nodes $m_2, m_3, \ldots, m_k$. To distinguish m_2 from $m_3, \ldots, m_k$, we will say that m_1 is inferred from the paths **rooted** in m_2 with the **help** of $m_3, \ldots, m_k$, and formally write this as $m_2; m_3, \ldots, m_k \vdash_p m_1$. If the paths used to infer m_1 have no "`!`" symbols, so that m_2 needs no help, we will simple write $m_2 \vdash_p m_1$

Consider the following example, illustrated in Figure 3.1:

```
(resetnet t)
Nodes, Contexts, Relations, and Caseframes reset

(defineType category individual)
category defined as a subtype of individual

;;; Define a subclass,superclass caseframe.
(defineCaseframe :type proposition
  :arcs (subclass :type category)
        (superclass :type category))
(<proposition, [ subclass :type category :adjust reduce :limit 1 :path nil]
               [ superclass :type category :adjust reduce :limit 1 :path nil]>)

;;; Define a member,class caseframe
;;; with a path-based inference rule for the class relation.
(defineCaseframe :type proposition
  :arcs (member :type individual)
        (class :type category
```

```
               :path (compose class (kstar (subclass- ! superclass)))))
(<proposition, [ member :type individual :adjust reduce :limit 1 :path nil]
               [ class :type category :adjust reduce :limit 1
                       :path (compose-path class
                                 (kstar-path (compose-path subclass-
                                                          ! superclass)))]>)

;;; Assert some facts.
#D#!(build member Tweety class canary)
M1!:{<member, {Tweety }> <class, {canary }> }

#D#!(build subclass canary superclass bird)
M2!:{<subclass, {canary }> <superclass, {bird }> }

#D#!(build subclass bird superclass animal)
M3!:{<subclass, {bird }> <superclass, {animal }> }

;;; What is asserted?
#D(assertions)
M1!:{<member, {Tweety }> <class, {canary }> }
M2!:{<subclass, {canary }> <superclass, {bird }> }
M3!:{<subclass, {bird }> <superclass, {animal }> }

;;; See if Tweety is an animal.
#D(find member Tweety class animal)
M1!:{<member, {Tweety }> <class, {canary }> }
;;; Note the node that's returned.

;;; It looks better if we use dbproject.
(dbproject (find member Tweety class animal) member class)
{<member, {Tweety}> <class, {animal bird canary}> }

;;; But still, no new nodes have been built.
#D(assertions)
M1!:{<member, {Tweety }> <class, {canary }> }
M2!:{<subclass, {canary }> <superclass, {bird }> }
M3!:{<subclass, {bird }> <superclass, {animal }> }

;;; On the other hand, we can build a new node and deduce it.
#D(deduce (build member Tweety class animal))
M4!:{<member, {Tweety }> <class, {animal }> }

;;; Now see what's asserted.
#D(assertions)
M1!:{<member, {Tweety }> <class, {canary }> }
M2!:{<subclass, {canary }> <superclass, {bird }> }
M3!:{<subclass, {bird }> <superclass, {animal }> }
M4!:{<member, {Tweety }> <class, {animal }> }
```

Even though, according to Section 2.4.1, path-based inference inferred a `class` relation from node `M1!` to node `animal`, that is not strictly correct. SNePS does not permit a new relation to

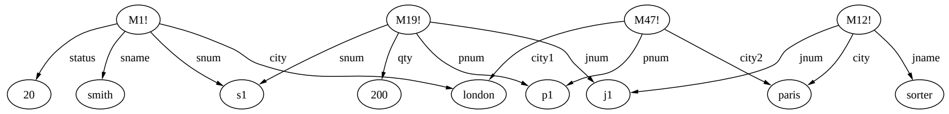

Figure 3.2: Path-based inference: `M47!` is inferred from the paths rooted in `M19!` with the help of `M1!` and `M12!`.

be added to an already existing node. So it is more correct to say that the path-based inference rule sanctioned the inference of a node of the description `<member, {Tweety}> <class, {canary animal}>`. This latter node, however, was never built. Instead, the deduce command derived node `M4!` from it by wire-based inference, without actually building the intermediate node.

Path-based inference is technically a multi-step process. Node m_1 is inferred by paths rooted in node m_2 with the help of $m_3, \ldots, m_k$ $(m_2; m_3, \ldots, m_k \vdash_p m_1)$, if there is some node m_v, that need never be built, such that

1. The case frame of m_2 is adjustable to the case frame of m_1.
2. The set of wires of m_v is a superset of the set of wires of m_2;
3. The extra wires of m_v are sanctioned by some path-based inference rules from those of m_2 which require that $m_3, \ldots, m_k$ be asserted;
4. $m_v \vdash_w m_1$.

Condition (1) above is very strict, and there are times when one wants to use path-based inference even though it does not hold. Consider the example illustrated in Figure 3.2. and shown in the following fragment:

```
;;; Input only for defining relations and caseframes.
(defineRelation snum :type individual :adjust none :limit 1)
(defineRelation sname :type individual :adjust none :limit 1)
(defineRelation status :type individual :adjust none :limit 1)
(defineRelation city :type individual :adjust none :limit 1)

(defineRelation pnum :type individual :adjust none :limit 1)
(defineRelation pname :type individual :adjust none :limit 1)
(defineRelation color :type individual :adjust none :limit 1)
(defineRelation weight :type individual :adjust none :limit 1)

(defineRelation jnum :type individual :adjust none :limit 1)
(defineRelation jname :type individual :adjust none :limit 1)
(defineRelation qty :type individual :adjust none :limit 1)

(defineCaseframe :type proposition :arcs snum sname status city)
(defineCaseframe :type proposition :arcs pnum pname color weight city)
(defineCaseframe :type proposition :arcs jnum jname city)
(defineCaseframe :type proposition :arcs snum pnum jnum qty)

;;; The relevant hypotheses.
M1!:{<snum, {s1}> <sname, {smith}> <status, {20}> <city, {london}>}
M12!:{<city, {paris}> <jnum, {j1}> <jname, {sorter}>}
```

```
M19!:{<snum, {s1}> <pnum, {p1}> <jnum, {j1}> <qty, {200}>}

;;; Definitions of new city1 and city2 relations, and a caseframe using them.
(defineRelation city1 :type individual :adjust none :limit 1
                :path (compose snum snum- city))
(defineRelation city2 :type individual :adjust none :limit 1
                :path (compose jnum jnum- city))
(defineCaseframe :type proposition :arcs city1 pnum city2)

;;; The inferring of a new node.
#D(deduce (build pnum p1 city1 london city2 paris))
M47!:{<pnum, {p1}> <city1, {london}> <city2, {paris}>}
```

In this second variety of path-based inference, node m_1 is inferred by paths rooted in node m_2 with the help of $m_3, \ldots, m_k$ ($m_2; m_3, \ldots, m_k \vdash_p m_1$), if

1. The case frame of m_2 is not adjustable to the case frame of m_1.

2. The wires of m_1 are either wires of m_2, or are sanctioned by some path-based inference rules from those of m_2 which require that $m_3, \ldots, m_k$ be asserted.

The function `deduce` is sound and complete for path-based inference.

3.4 Subsumption Inference

Subsumption inference is an inference method whereby a proposition about some node y is inferred from a proposition about a node x that subsumes y. Subsumption inference has been implemented only for cases in which x and y are arbitrary or indefinite nodes.

The implemented subsumption inference rules include

1. $\frac{P((any\ \ x\ \ \Phi(x))), \Psi(y) \supseteq \Phi(x)\{y/x\}}{P((any\ \ y\ \ \Psi(y)))}$
   Example: *If every elephant is large, then every albino elephant is large.*

   ```
   input:
   #D#!(build object (any x (build member x class elephant))
              property large)
   output:
   M3!:{<object, {(any Arb1 {M1!:{<member, {Arb1}> <class, {elephant}>}})}>
        <property, {large}>}

   input:
   #D(deduce (build object (any y (build member y class elephant)
                                  (build object y property albino))
                    property large))
   output:
   M7!:{<object,
         {(any Arb2
           {M4:{<member, {Arb2}> <class, {elephant}>}
            M5:{<object, {Arb2}> <property, {albino}>}})}>
        <property, {large}>}
   ```

2. $\frac{P((any\ x\ \Phi(x))), \Psi(y) \supseteq \Phi(x)\{y/x\}}{P((some\ y\ \alpha\ \Psi(y)))}$

   Example: *If every elephant is large, then some albino elephant is large.*

```
input:
#D#!(build object (any x (build member x class elephant))
          property large)
output:
M3!:{<object, {(any Arb1 {M1!:{<member, {Arb1}> <class, {elephant}>}})}>
     <property, {large}>}

input:
#D(deduce (build object (some y () (build member y class elephant)
                                   (build object y property albino))
                property large))
output:
M11!:{<object,
       {(some Ind1
         ({})
         {M8:{<member, {Ind1}> <class, {elephant}>}
          M9:{<object, {Ind1}> <property, {albino}>}})}>
      <property, {large}>}
```

3. $\frac{P((any\ x\ \Phi(x))), \Psi(y) \subseteq \Phi(x)\{y/x\}}{P((some\ y\ \alpha\ \Psi(y)))}$

   Example: *If every albino elephant is valuable, then some elephant is valuable.*

```
input:
#D#!(build object (any x (build member x class elephant)
                         (build object x property albino))
                property valuable)
output:
M12!:{<object,
       {(any Arb2
         {M4!:{<member, {Arb2}> <class, {elephant}>}
          M5!:{<object, {Arb2}> <property, {albino}>}})}>
      <property, {valuable}>}

input:
#D#!(deduce (build object (some y () (build member y class elephant))
                  property valuable))
output:
M15!:{<object,
       {(some Ind2 ({}) {M13!:{<member, {Ind2}> <class, {elephant}>}})}>
      <property, {valuable}>}
```

4. $\frac{P((any\ x\ \Phi(x) \wedge \Psi(x)))}{\Psi((some\ y\ \alpha\ \Phi(y)))}$

   Example: *If every albino elephant is valuable, then some elephant is albino.*

```
input:
#D#!(build object (any x (build member x class elephant)
                         (build object x property albino))
                property valuable)
```

```
output:
M12!:{<object,
       {(any Arb2
         {M4!:{<member, {Arb2}> <class, {elephant}>}
          M5!:{<object, {Arb2}> <property, {albino}>}})}>
      <property, {valuable}>}

input:
#D#!(deduce (build object (some y () (build member y class elephant))
                   property albino))
output:
M18!:{<object,
       {(some Ind3 ({}) {M16!:{<member, {Ind3}> <class, {elephant}>}})}>
      <property, {albino}>}
```

5. $\frac{P((some\ x\ \alpha\ \Phi(x) \wedge \Psi(x)))}{\Psi((some\ y\ \beta\ \Phi(y)))}$
Example: *If some old elephant is wrinkled, then some elephant is old.*

```
input:
#D#!(build object (some x () (build member x class elephant)
                             (build object x property old))
           property wrinkled)
output:
M22!:{<object,
       {(some Ind4
         ({})
         {M19!:{<member, {Ind4}> <class, {elephant}>}
          M20!:{<object, {Ind4}> <property, {old}>}})}>
      <property, {wrinkled}>}

input:
#D(deduce (build object (some y () (build member y class elephant))
                 property old))
output:
M25!:{<object,
       {(some Ind5 ({}) {M23:{<member, {Ind5}> <class, {elephant}>}})}>
      <property, {old}>}
```

Chapter 4

Transformers

These functions convert Lisp objects to SNePS nodes, and vice versa. To convert a single Lisp object to a node, see the function

`(@ form)` *[Function]*
Gets the Lisp *form* evaluated, coerces the result to a node or set of nodes, and returns that coerced result.

`(apply-function-to-ns fn ns)` *[Function]*
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 or node set, and returns that.

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

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

`(node-to-lisp-object nde)` *[Function]*
Returns a Lisp object corresponding to the SNePS node *nde*. The Lisp object will be either a number or a symbol.
`node` (page 12).

Chapter 5

SNePS as a Database Management System

5.1 Database Functions

Functions specifically supplied for treating SNePS as a Database Management System are documented in this section.

`(dbcount` *nodesetexp* `)` *[Function]*
Evaluates the SNePSUL nodeset expression, ***nodesetexp***, and returns a singleton nodeset of the node whose identifier looks like the number which is the number of nodes in the resulting set.

`(dbmax` *nodesetexp* `)` *[Function]*
Evaluates the SNePSUL nodeset expression, ***nodesetexp***, which must evaluate to a set of nodes all of whose identifiers look like numbers, and returns a singleton nodeset of the node whose identifier looks like the largest of the numbers.

`(dbmin` *nodesetexp* `)` *[Function]*
Evaluates the SNePSUL nodeset expression, ***nodesetexp***, which must evaluate to a set of nodes all of whose identifiers look like numbers, and returns a singleton nodeset of the node whose identifier looks like the smallest of the numbers.

`(dbproject` *nodesetexp relations* `)` *[Function]*
For each node, n_i in the ***nodesetexp***, and for each $relname_j$ in the ***relations***, prints a virtual cableset of the form {...<$relname_j$, ns_{ij}>...} such that a relation named $relname_j$ goes from n_i to the nodeset ns_{ij}. Does not print two virtual cablesets that look alike.

`(dbtot` *nodesetexp* `)` *[Function]*
Evaluates the SNePSUL nodeset expression, ***nodesetexp***, which must evaluate to a set of nodes all of whose identifiers look like numbers, and returns a singleton nodeset of the 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.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

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

LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

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.

Bibliography

Shapiro, S. C. (1993). Belief spaces as sets of propositions. *Journal of Experimental and Theoretical Artificial Intelligence (JETAI)*, 5(2&3):225–235.

Index

*, 13
+, 13
-, 13
=, 12, 13
?, 9, 12
@, 12, 13, 25
#!, 5
#>, 5
#?, 5
#@, 5
#C, 5, 9
#CBaseCT, 9, 10
#CDefaultCT, 9, 10
#D, 5, 14
#N, 5
#c, 5, 9
#d, 5, 14
#n, 5

allContexts, 9
allNodes, 14, 15
and, 8
any, 3, 11
apply-function-to-ns, 25
arbitrary node, 11, 21
assert, 11, 12
assertions, 9, 15

BaseCT, 9, 10
build, 6, 7, 11, 12

case frames, 6, 10
class, 3
clean, 13
close, 11
closure, 11, 12
common-lisp-user package, 5
compose, 8
context, 9
contexts
  current, 9, 11, 16
  operating on, 9
converse, 8
currentContext, 9

dbcount, 27
dbmax, 27
dbmin, 27
dbproject, 27
dbtot, 27
deduce, 16, 18, 21
DefaultCT, 9, 10
defineCaseframe, 10
definePath, 7
defineRelation, 7
defineType, 6
demo, 6
depends, 3
ders, 9, 16
describe, 14
documentation, 6
documentation string, 3, 6, 10
domain-restrict, 9
doset, 13
dump, 14

e-quantifier, 11
empty?, 13
erase, 12
everyMember, 14

find, 12, 16

hyps, 9, 16

indefinite node, 11, 12, 21
inference, 17
  path-based, 7, 18
  subsumption, 21
  wire-based, 11, 17
intext, 6
irreflexive-restrict, 9

kill-sneps, 12
kplus, 8
kstar, 8

lisp, 5
lisp-list-to-ns, 25
list-caseframes, 10, 14
list-relations, 7, 14
load, 6

mapset, 14
member, 3

new-set, 14
newContext, 9
node
    arbitrary, 11, 21
    indefinite, 11, 12, 21
    molecular, 6
node, 12
node-to-lisp-object, 25
nodesets, 13
nodesetspec, 11–13
ns-to-lisp-list, 25

or, 8

package
    common-lisp-user, 5
    snepsul, 5
path-based inference, 18
path-based inference rules, 7
paths
    syntax and semantics, 7
print-down, 14, 15
print-up, 14, 15

quantifier, 3, 11, 12

range-restrict, 9
relations, 7
resetnet, 10, 12
restrict, 3

scope, 3, 11
semantic types, 6
setCurrentContext, 10
show, 15
showContexts, 10, 15
showSubsumption, 15
showSupports, 15
showTypes, 6, 15
showWithSupports, 15
singleton, 14
sneps, 5
snepsul package, 5
SNePSUL readtable, 5
SNIP, 17
some, 3, 12
someMember, 14
strSubsumee, 8
strSubsumer, 8
subset-suchthat, 14
subsumee, 4
subsumees, 4
subsumer, 4
subsumers, 4
subsumption
    structural, 8
subsumption inference, 21

u-quantifier, 11
unassert, 13
unitpath, 7
user::kill-sneps, 12
user::sneps, 5

variables
    arbitrary, 12

wire-based inference, 17

xpath, 8, 9