SNePS: AN INTERACTIVE APPROACH

Stuart C. Shapiro and William J. Rapaport

SNePS Research Group
Department of Computer Science and Center for Cognitive Science
State University of New York at Buffalo, Buffalo, NY 14260
{shapiro|rapaport}@cs.buffalo.edu
http://www.cs.buffalo.edu/pub/SNePS/WWW/index.html

April 12, 1995

NOTE: In all cases where this tutorial disagrees with the SNePS User's Manual (Shapiro & Group 1994; hereafter "UM"), the UM takes precedence.

1 HOW TO RUN SNePS (on the Computer Science machines at UB).

1. Run `acl`
2. Do `:ld /projects/snwiz/bin/sneps21`
3. Evaluate `(sneps)`
4. To return to Lisp, evaluate `(lisp)`. You can then evaluate `(exit)` to leave Lisp.

2 BUILDING AND FINDING NODES.

2.1 All sessions with SNePS begin by defining the relations (arc labels) that you want to use. (Some relations, for use with node-based inference, are built in; see §7, below, and UM§2.5, UM Ch. 3.) To define a set of relations, evaluate the SNePSUL command `define`, giving it as arguments the names of the relations (UM§2.5). Define the relations *isa* and *ako* by evaluating:

```
(define isa ako)
```

(Note to the reader/user: To get the full benefit from this interactive tutorial, you should do everything that is underlined. It is also useful to draw the network as you build it; you can then compare your drawing of the network with the information that SNePS gives you. You can also use the Xginseng program to see a picture of the network; see §6.2.)

2.2 If you make a mistake or change your mind, you can undefine the relations, using `undefine` (UM§2.5). Undefine the two relations we just defined, by evaluating:

```
(undefine isa ako)
```

2.3 EXERCISE: Define the relations *member*, *class*, *object*, and *ability*.

(Note: Answers to all the exercises are in §9. You should first try the exercises on your own. If you get them wrong, you can then erase your mistakes and type in the answers.)

2.4 To assert a proposition, i.e., to tell the SNePS knowledge-representation and reasoning system something that you want it to believe, evaluate the SNePSUL command `assert`. `assert` can take pairs of arguments: The first member of each pair is a relation; the second can be a node (or a list of nodes; see §2.30). `assert` builds a propositional node that has an arc that is labeled with the relation and that points to the node that is the second argument. If there are several such relation–node pairs, then the propositional node that is built has several arcs, each labeled with one of the relations, and each pointing to one of the nodes. Finally, the node that is built is asserted. SNePS automatically generates an identifier for the node that is built by the `assert` command. SNePS marks the fact that this node is asserted with an exclamation mark. For the details on `assert`, see UM§§1.4, 2.7.

Cassie is the computational cognitive agent implemented in SNePS. She can be thought of as the agent whose memory consists of the knowledge base represented in SNePS. Let's tell Cassie that Clyde is an elephant. There are several ways this can be represented. For the purposes of this tutorial, we'll represent it by a node that analyzes this proposition as: Clyde is a member of the class of elephants. To do this, evaluate:

```
(assert member Clyde class elephant)
```

which will build a node with a `member` arc pointing to a node with the identifier `Clyde` and a `class` arc pointing to a node with the identifier `elephant`. SNePS asserts node `M1` (which it represents as '`M1!`', since it *asserted* it, and didn't merely *build* it).

2.5 Suppose you accidentally typed '`ephalunt`' instead of '`elephant`'. You can erase the mistakenly-asserted node with the SNePSUL command `erase`, which takes a node as an argument (UM§2.8): `erase m2`. To practice with this, we'll do the following: First, assert a node with a misspelling. Since this will be the second node you will have asserted, SNePS will call it '`M2`'. Then erase `M2` and assert what we wanted without any misspellings (be careful!):

```
(assert member Clyde class ephalunt)
(erase m2)
(assert member Clyde class elephant)
```

SNePS should have simply returned node `M1`, since we had already built it, and, by the Uniqueness Principle, SNePS does not build or assert nodes more than once (UM§3.1.2). Thus, instead of building another representation of the proposition that Clyde is an elephant, Cassie showed us that she already believed that.

2.6 Let's start over. To do that, you can reset the network with the SNePSUL command `resetnet`, which needs no arguments. For other options, see §4.1, below, and UM§2.8. So, do `(resetnet)` now.

2.7 EXERCISE: Tell Cassie that Clyde is an elephant. Then tell Cassie that Dumbo is an elephant.

2.8 Now let's look at what you've built. One way of doing this is using `dump`, which can take any node or nodes as arguments. Try:

```
(dump m1)
(dump elephant)
```

Try to understand the information that SNePS returns. If you don't understand all of it now, be patient. Experiment with some other uses of `dump`. (See UM§2.10.)

2.9 Another way of looking at the network you've built is with `describe`, which can take any node or nodes as arguments (UM§2.10). Try:

```
(describe m1 m2)
```

2.10 `describe` can also take the argument `*nodes`, which will return *all* the nodes in the network (UM§§1.6, 2.9). Try:

```
(describe *nodes)
```

Experiment with some other uses of `describe`. And as you continue on in this tutorial, use `dump` and `describe` frequently, to see what is going on.

2.11 Since `assert` returns a node, it can be the argument of `describe`. This way, you can see what you're building while you're building it. Try:

```
(describe (assert object Dumbo ability fly))
```

This is one way of telling Cassie that Dumbo can fly. (For more information on `dump` and `describe`, see UM§2.10.)

2.12 EXERCISE: Use the `(describe (assert ... ))` technique to tell Cassie the following:

1. Tweety is a canary.
2. Tweety can fly.
3. Opus is a bird.

2.13 Now let's ask Cassie questions about what we've told her. One way to do this is by asking SNePS to *find* nodes. To do this, we need to describe the node(s) we want SNePS to find. Like `assert`, the SNePSUL command `find` can take relation–node pairs as arguments. E.g., suppose we ask Cassie who is an elephant. One way to ask this is to ask SNePS to find all *propositions* that say who is an elephant. To do this, we want to find all nodes that have a `class` arc coming out of them and pointing to the `elephant` node, since that uniquely describes all propositional nodes that represent that something is an elephant. (See UM§2.11.) Try:

```
(find class elephant)
```

2.14 Since `find` only returns node identifiers, we can get more information by using `describe`: Try:

```
(describe (find class elephant))
```

2.15 We can also describe nodes that we want to find by describing *paths* of relations (i.e., paths of arcs). E.g., another way to ask Cassie who an elephant is is to ask SNePS to find all elephants, i.e., all members of the class of elephants. To do this, we want to find all nodes that have a `member` arc coming *into* them from a node that has a `class` arc going *out* of it. This node in the middle, therefore, has a `member` arc going out of it *and* a class arc going out of it; i.e., it is a node that represents a proposition that the thing with the `member` arc coming into it is a member of the class with the `class` arc coming into it. So, to repeat, we want to find a node that (among other things) has a `member` arc coming into it. Now, another way to look at this is to say that that node has a *converse* `member` arc coming *out* of it! Such a converse relation is denoted with a '–' sign: `member-`. (See UM§2.5.)

But it's not enough to find nodes with `member-` arcs coming out of them (i.e., with `member` arcs coming into them). That `member-` arc must go into a node that has a `class` arc coming out of it. So we want to find a node that has a *path* of arcs consisting of a `member-` arc followed by a `class` arc. *Furthermore*, the `class` arc must go into the `elephant` node!

We can represent paths by a list of relations: `(member- class)` is the path we've been talking about. So, to ask Cassie who is an elephant in such a way that she will tell us who the elephants are, we ask:

```
(find (member- class) elephant)
```

Before trying this, make sure you understand what it says: It is a `find` command with a relation–node pair as its argument. The relation is the path `(member- class)` (it can be understood as the path expressing the relationship of *being a member of the class of*), and the node is `elephant`. Now try it.

2.16 To ask Cassie who can fly, we can ask her to find all nodes that have `(object- ability)` paths to the `fly` node: Evaluate:

```
(find (object- ability) fly)
```

2.17 To ask Cassie which elephants can fly, we want her to find all nodes that represent elephants *and* that represent things that can fly. I.e., we want her to find all nodes that have a `(member- class)` path to `elephant` *and* an `(object- ability)` path to `fly`. This is the intersection of the two sets of nodes she found for us before. To say this, evaluate:

```
(find (member- class) elephant (object- ability) fly)
```

2.18 EXERCISE: Find canaries that can fly.

2.19 Sometimes we might want to build nodes without asserting them. This is done with the SNePSUL command `build`, which, like `assert` and `find`, takes relation–node pairs as arguments (UM§§1.4, 2.7).

Suppose that John has some beliefs about things that can fly, and suppose that Cassie has some beliefs about what John believes. It might be the case (in fact, it will probably be the case) that Cassie's beliefs and John's beliefs will differ. So, when Cassie represents John's beliefs, she might not assert them. (She would only assert them if she herself believed them, too.) E.g., suppose John believes that Opus can fly. Note that Cassie has no beliefs one way or the other about Opus's ability to fly. Let's tell Cassie that John believes that Opus can fly. To do this, we'll first have to have a way to represent Cassie's beliefs about John's beliefs. We'll do this with an `agent-act-object` *case frame*. (For more on this, see items [41,50,52,76,82,83] in the SNeRG bibliography.)[1] So, first we need some new arcs: Evaluate:

```
(define agent act object)
```

(Note that SNePS reminds us that `object` has already been defined.)

2.20 The *agent* will be John (the believer), the *act* is the mental act of believing, and the *object* of John's act of believing will be a proposition. So, to tell Cassie that John believes that Opus can fly, evaluate:

```
(describe (assert agent John
                  act believe
                  object (build object Opus ability fly)))
```

Note that we *built* but did not *assert* that Opus can fly. We did, however, *assert* that John believes this. So, *Cassie* believes that John believes that Opus can fly, and *John* believes that Opus can fly. (Sometimes we say that the proposition that Opus can fly is now in John's "belief space" but not (directly) in Cassie's belief space, whereas the proposition that John believes that Opus can fly *is* in Cassie's belief space.)

2.21 What does Cassie believe about things that can fly? Ask her what propositions she has *stored* in her knowledge base about who can fly:

```
(describe (find ability fly))
```

2.22 We can also ask her what propositions she *believes* about who can fly, using the SNePSUL command `findassert`, which is just like `find` except that it only returns *asserted* nodes (UM§2.11): Ask Cassie:

```
(describe (findassert ability fly))
```

2.23 We can also ask Cassie which nodes in her knowledge base represent things that (someone or other believes) can fly: Evaluate:

```
(find (object- ability) fly)
```

2.24 And we can ask her which things *she* believes can fly. To do this, we want her to find all nodes that have `object` arcs coming from *asserted* nodes that have `ability` arcs also coming out of them. This is done by putting an exclamation mark in the path (UM§2.5.2): Evaluate:

```
(find (object- ! ability) fly)
```

[1] `http://www.cs.buffalo.edu/pub/SNePS/WWW/Bibliography/bibliography/node1.html`

2.25 EXERCISES: Tell Cassie the following:

1. John believes that Clyde can fly.
2. Tweety believes that Tweety can fly.

And then (3) find the elephants that Cassie believes can fly.

2.26 We can also find the individuals that John believes can fly. To see how to do this, first find the nodes that represent Cassie's beliefs about John's beliefs; this can be done using `(find agent John act believe)`. Then use this to find the nodes that are objects of John's beliefs about who can fly: `(find object- (find agent John act believe) ability fly)`. Finally, use this to find the objects that John believes can fly: Evaluate the following:

```
(find object-
      (find object- (find agent John act believe)
            ability fly))
```

Be sure you understand why there are two occurrences of `object-`; they do *not* necessarily refer to the same arcs!

2.27 We can also ask Cassie to find nodes that are described a bit more vaguely. E.g., we could ask her to find all nodes that have a `class` arc to `elephant` *and* a `member` arc to *anything*. This is done by using SNePSUL variables that range over nodes, indicated by a question mark (UM§2.11): Evaluate:

```
(find member ?x class elephant)
```

2.28 The nodes found this way are stored in the variable `x`; we can retrieve the values in `x` by prefixing it with an asterisk (UM§2.9): Evaluate:

```
*x
```

2.29 EXERCISE: Find individuals who believe that they, themselves, can fly.

2.30 Instead of telling Cassie that Clyde is an elephant *and* that Dumbo is an elephant, we can tell Cassie that *Clyde and Dumbo* are elephants, using *sets* of nodes. Sets of nodes are denoted by lists of nodes; e.g., the set consisting of Clyde and Dumbo is represented in SNePSUL by `(Clyde Dumbo)` (UM§2.7). So let's reset the network, and tell Cassie what we had told her earlier: Evaluate each of the following:

```
(resetnet)
(describe (assert member (Clyde Dumbo) class elephant))
(describe (assert member Tweety class canary))
(describe (assert member Opus class bird))
```

2.31 EXERCISE: Using one sentence, tell Cassie that Tweety and Dumbo can fly.

2.32 Ask Cassie whom she believes to be flying elephants; evaluate:

```
(find (member- ! class) elephant (object- ! ability) fly)
```

2.33 EXERCISES:

1. Using one sentence, tell Cassie that John believes that Opus and Clyde can fly.
2. Find the elephants that John believes can fly.

3 REDUCTION INFERENCE (UM§2.5.1).

3.1 Besides merely asking Cassie to tell us what's in her knowledge base, by using the `find` command, we can ask her to *infer* information. One way to do this is with the `deduce` command. Let's first recall what Cassie believes; evaluate:

```
(describe *nodes)
```

3.2 Just like `assert`, `build`, and `find`, `deduce` can take as arguments relation–node pairs, and it can use SNePSUL variables in place of nodes (UM§2.11). However, the variables are denoted with dollar signs (UM§2.7): `$x`. E.g., to find out who is a canary, *not* just by asking Cassie whom she explicitly believes to be a canary, but by asking her to infer all the canaries she explicitly or implicitly is aware of, evaluate:

```
(describe (deduce member $x class canary))
```

3.3 To infer whether Tweety is a bird, evaluate:

```
(describe (deduce member Tweety class bird))
```

Note that Cassie's silent reply is tantamount to her saying "I don't know".

3.4 To infer whether Dumbo is an elephant, evaluate:

```
(describe (deduce member Dumbo class elephant))
```

3.5 To infer whether John believes that Clyde can fly, evaluate:

```
(describe (deduce agent John act believe
                  object (build object Clyde ability fly)))
```

3.6 EXERCISES:

1. Tell Cassie that John believes that Orville and Wilbur can fly.
2. Does John believe that Orville can fly?

4 DIFFERENT NODE TYPOGRAPHY.

4.1 Sometimes, you need to refer to an already-built node, but you don't know what identifier SNePS has given it. To handle cases like this, you can create a pointer to the node when you build it that has a mnemonic name (a SNePSUL variable) that you give it, so that you'll remember what it is. This can be done by using the infix '`=`' operator, as shown in the following examples (UM§2.9). First, reset the network *and undefine all user-defined relations* (UM§2.8); this is done by giving `resetnet` the argument `t`. So, evaluate:

```
(resetnet t)
```

4.2 Now, define some new relations:

```
(define department division number name credits prerequisites)
```

4.3 Tell Cassie some information about a course in the CS department:

```
(describe (assert department CS division undergrad number 113
                  name "Introduction to Computer Science I"
                  credits 4) = cs113)
```

4.4 Note that the node that was asserted about CS 113 now has a SNePSUL variable, `cs113`, assigned to it. So we can always access that node using the SNePSUL variable `cs113`. To access it, prefix it with an asterisk (UM§2.9): `*cs113`. E.g., to say that CS 113 is a prerequisite for CS 114 without having to know which node represents CS 113, evaluate:

```
(describe (assert department CS division undergrad number 114
                  name "Introduction to Computer Science II"
                  credits 4 prerequisites *cs113) = cs114)
```

4.5 To find CS 113's prerequisites, evaluate

```
(find prerequisites- *cs113)
```

Note that Cassie doesn't know about any.

4.6 EXERCISE: Find CS 114's prerequisites.

4.7 To find the courses that have CS 113 as a prerequisite, evaluate:

```
(find (name- prerequisites) *cs113)
```

4.8 EXERCISE: Tell Cassie about two graduate courses:

1. CS 572, Introduction to Artificial Intelligence, 3 credits, no prerequisites
2. CS 676, Knowledge Representation, 3 credits, prerequisite: CS 572

And ask about their prerequisites, as before.

5 BASE NODES.

5.1 If we want to tell Cassie about individuals that satisfy certain descriptions without giving those individuals a name, we can use *base nodes* (UM§§1.4, 2.7). These are nodes that have no outgoing arcs; i.e., they have no structural information. All that we know about them is what is asserted about them. (On the difference between structural and assertional information, see Woods 1975.) To get started, evaluate:

```
(resetnet t)
(define agent act object member class)
```

5.2 Suppose we want to tell Cassie that a dog bit John, without telling her which dog (it might not matter which, or we might not know). We'll do this in two steps. First, we'll tell Cassie that something is a dog; then we'll tell her that the dog bit John. The first can be done by creating a Skolem constant representing a dog. The way to do this in SNePSUL is by prefixing a `#` to a SNePSUL variable (UM§2.7), e.g., `#dog`. Evaluate:

```
(describe (assert member #dog class dog))
```

5.3 Second, we tell Cassie that the dog bit John. This is done with the `*` operator (UM§2.9): Evaluate:

```
(describe (assert agent *dog act bite object John))
```

5.4 EXERCISE: Tell Cassie that John saw an elephant.

6 PATH-BASED INFERENCE (UM§2.5.2).

6.1 Another way for Cassie to infer information is via *path-based inference*. This is a generalization of the notion of *inheritance* found in other semantic-network formalisms. Begin by setting up a knowledge base about various animals:

```
(resetnet t)
(define object isa has)
(describe (assert object elephant isa animal))
(describe (assert object circus\ elephant isa elephant))
(describe (assert object Dumbo isa circus\ elephant))
(describe (assert object Clyde isa elephant))
(describe (assert object bird isa animal))
(describe (assert object Tweety isa bird))
(describe (assert object animal has head))
(describe (assert object head has mouth))
(describe (assert object elephant has trunk))
(describe (assert object trunk isa appendage))
```

6.2 Digression. Now might be a good time to try the Xginseng graphical interface to SNePS (UM Ch. 5). You must be on an X-windows terminal, preferably using a fast machine. See the manual for details on how to use Xginseng. You may need to do the following: Suppose you are sitting at machine A, but you are executing SNePS on machine B.

1. On machine A, type `xhost B` at the system prompt.
2. On machine B, type `setenv DISPLAY A:0` (that's "A-colon-zero") at the system prompt..

Then evaluate: `(xginseng)` in SNePS.
end digression.

6.3 Recall that we have a knowledge base about animals. To see it, evaluate:

```
(describe *nodes)
```

6.4 Ask Cassie to tell you who she believes to be an animal:

```
(find (object- ! isa) animal)
```

6.5 Now, this isn't all the animals. What we did was asked Cassie whom she explicitly believed to be an animal. What we probably would rather know is whom she can infer to be an animal. To do this, we can tell her that certain paths of relations are equivalent to certain relations. For instance, we can tell her that the `isa` relation is transitive. This is done with the `define-path` command, which takes a relation as its first argument and a path as its second. To define the path, we can use the `compose` command, which can be thought of as creating a new relation consisting of a path of relations. The path of relations might consist of zero or more (i.e., `kstar`) occurrences of certain relations, and we might want to restrict the path to go through only asserted nodes. (See UM§2.5.2 for a full description of the syntax and semantics of paths.) E.g., to tell Cassie that `isa` is transitive, evaluate:

```
(define-path isa (compose isa (kstar (compose object- ! isa))))
```

6.6 Now, ask Cassie whom she believes to be an animal. She will use the inheritance information we just gave her to `find` the answer:

```
(find (object- ! isa) animal)
```

6.7 An alternative way to ask a similar question is to use `findassert`:

```
(find object- (findassert isa animal))
```

Note the different—but related—information returned by 6.6 and 6.7.

6.8 EXERCISES:

1. Extend the relation `has` so that it "distributes" over `isa`. I.e., define `has` so that X `has` Y if X `isa` A and A `has` B and B `isa` Y. In general, we want this:

   ```
   has <= isa* has+ isa*
   ```

   I.e., the `has` path is defined as 0 or more `isa` relations, followed by 1 or more `has` relations, followed by 0 or more `isa` relations. (This is tricky. You'll need to remember to use asserted nodes in your path, and you'll want to start out at an asserted node!)

2. What has a mouth?

3. What does Dumbo have?

7 NODE-BASED INFERENCE (UM Ch. 3).

7.1 The third way to do inference in SNePS is to use *rules*. Rules in SNePS are not kept in a separate location from the rest of the knowledge base, as they are in many production systems. Rather, rules are propositions that Cassie believes. As such, they can themselves be reasoned about. But in what follows, we'll just use them as inference rules in a predicate logic, using an example from Rich & Knight 1991: 134–135. First, reset the network:

```
(resetnet t)
```

And define some relations:

```
(define member class)
```

7.2 Tell Cassie the following information (note, by the way, that we are ignoring tense):

1. Marcus was a man.
   ```
   (describe (assert member Marcus class man))
   ```
2. Marcus was a Pompeian.
   ```
   (describe (assert member Marcus class Pompeian))
   ```
3. All Pompeians were Romans.

   To assert this, we need some relations that are built into SNePS for use in node-based inference. These include:

`forall`	which is used to represent the universal quantifier (UM§3.1.2)
`ant`	which is used to represent the antecedent of an if–then rule (UM§3.1.1)
`cq`	which is used to represent the consequent of the rule (UM§3.1.1)

   The node that SNePS builds at the head of a `forall` arc is a *variable* node, in this case representing a universally quantified variable (UM§1.4). The nodes that SNePS builds at the heads of `ant` and `cq` arcs can be *pattern* nodes that SNIP, the SNePS Inference Engine, uses to *match* other nodes in the network as part of the inference process (UM§1.4). Note in the following SNePSUL command the use of `$` SNePSUL variables and the `*` operator.

   ```
   (describe (assert forall $p
                   ant (build member *p class Pompeian)
                   cq  (build member *p class Roman)))
   ```

   This says: For all p, if p is a member of the class Pompeian, then p is a member of the class Roman; i.e., for all p, if p is Pompeian, then p is Roman; i.e., all Pompeians are Romans.
4. Caesar was a ruler.
   ```
   (describe (assert member Caesar class ruler))
   ```

5. All Romans were either loyal to Caesar or hated him.

   Here, we need some more logical notation, and some more relations. First, define the relations:

   ```
   (define arg1 rel arg2)
   ```

   Next, the new notation. We need to express disjunction. Let's use exclusive disjunction. In ordinary first-order logic, exclusive disjunction is a connective that takes only 2 propositions. In SNePS's logic, there is a "connective" that is a generalization of conjunction, disjunction, and several others, which takes an arbitrary number of propositions. It is called *and-or*. The built-in relations used to represent it are `min` and `max`. In general, `min X max Y arg (P1 ... Pn)` is interpreted as: at least X and at most Y of the propositions P1, ... , Pn are true. Thus, e.g., if X = 0 and Y = 0, we have a generalized negation. If X = n and Y = n, we have a generalized conjunction. If X = 1, Y = 2, and n = 2, we have inclusive disjunction. If X = 1, Y = 1, and n = 2, we have exclusive disjunction, which is what we want for "All Romans were either loyal to Caesar or hated him." So, evaluate:

   ```
   (describe (assert forall $r
                    ant (build member *r class Roman)
                    cq  (build min 1 max 1
                               arg ((build arg1 *r
                                           rel loyal\ to
                                           arg2 Caesar)
                                    (build arg1 *r
                                           rel hate
                                           arg2 Caesar)))))
   ```

   This says: For all r, if r is Roman, then at least 1 and at most 1 of the following two propositions is the case: r is loyal to Caesar, r hates Caesar.

6. Everyone is loyal to someone.

   Here, we might say: For all m, if m is a man, then there is a y such that m is loyal to y. (Of course, not everyone is a man; we can fix this later.) This might be done with an existential quantifier arc. Instead, however, we use the technique of Skolem functions. We will have a Skolem function that takes as its argument a person and returns the individual that that person is loyal to. First, define a new relation:

   ```
   (define function)
   ```

   Next, we need a way to describe the individual to whom someone is loyal; so, call this individual the "liege of" the man who is loyal. Putting this altogether, we have:

   ```
   (describe (assert forall $m
                    ant (build member *m class man)
                    cq  (build arg1 *m
                               rel loyal\ to
                               arg2 (build function liege\ of
                                           arg1 *m))))
   ```

   This says: For all m, if m is a man, then m is loyal to the liege of m.

7. People only try to assassinate rulers they are not loyal to.

   Here, we need negation, for which we'll use `min 0 max 0`, as described above. We'll also need to have a conjunction in the antecedent: For all ppl and all r, if ppl is a person, *and* r is a ruler *and* ppl tries to assassinate r, then ppl is not loyal to r. This is done using the built-in SNePSUL relation `&ant`. Note also that we *re-use* the SNePSUL variable `*r`:

```
(describe (assert forall ($ppl *r)
                 &ant ((build member *ppl class person)
                       (build member *r class ruler)
                       (build arg1 *ppl
                              rel try\ to\ assassinate
                              arg2 *r))
                 cq (build min 0 max 0
                           arg (build arg1 *ppl
                                      rel loyal\ to
                                      arg2 *r))))
```

8. Marcus tried to assassinate Caesar.

```
(describe (assert arg1 Marcus
                 rel try\ to\ assassinate
                 arg2 Caesar))
```

9. Was Marcus loyal to Caesar?

```
(describe (deduce arg1 Marcus rel loyal\ to arg2 Caesar))
```

10. Oops! We forgot to tell Cassie that all men are persons. We can use *forward* inference to do this. I.e., we can tell Cassie a new rule, that all men are persons, and then we can immediately have Cassie perform forward inference using this new rule. This is done with the SNePSUL command `add`, which is syntactically just like `assert`:

```
(describe (add forall *m
               ant (build member *m class man)
               cq  (build member *m class person)))
```

7.3 EXERCISE (based on Ginsberg 1993: 126ff):

1. Barbara is a lawyer.
2. All lawyers are rich.
3. Everyone who is rich owns a big house.
4. Big houses are a lot of work.
5. What is a lot of work?

8 PROGRAMMING PROJECT.

1. Start a new SNePS session. Represent the information:

   (a) Elephants are animals.

   (b) A circus elephant (i.e., any/every circus elephant) is an elephant.

   (c) Dumbo is a circus elephant.

   (d) Clyde is an elephant

   (e) A trunk is an appendage.

   (f) Alex is an AI course instructor.

   (g) Tweety is a bird.

   Show the following questions being answered correctly:

   (a) Is Dumbo an animal? (Should be "yes".)

   (b) Is Alex an animal? (Should be "no"/"I don't know".)

2. Represent the following information:

   (a) Elephants have trunks.

   (b) Animals have heads.

   (c) A head has a mouth.

   (d) Circus performers have colorful costumes.

   Show the following questions being answered correctly:

   (a) Does Clyde have a mouth? (Should be "yes".)

   (b) Do elephants have appendages? (Should be "yes".)

3. Represent the following information:

   (a) Elephants are gray.

   (b) AI course instructors are insane.

   Show the following questions being answered correctly:

   (a) Are circus elephants gray? (Should be "yes".)

   (b) Is Alex insane? (Should be "yes".)

4. Represent the following information:

   (a) Birds fly.

   (b) Dumbo can fly.

   (c) Elephants can play.

   Show the following questions being answered correctly:

   (a) Can Tweety fly? (Should be "yes".)

   (b) Can elephants fly? (Should be "yes".)

   (c) Can Clyde fly? (Should be "no"/"I don't know".)

   (d) Can Clyde play? (Should be "yes".)

9 ANSWERS TO EXERCISES.

2.3
```
(define member class object ability)
```

2.7
```
(assert member Clyde class elephant)
(assert member Dumbo class elephant)
```

2.12
```
(describe (assert member Tweety class canary))
(describe (assert object Tweety ability fly))
(describe (assert member Opus class bird))
```

2.18
```
(find (member- class) canary (object- ability) fly)
```

2.25
```
(describe (assert agent John act believe
                  object (build object Clyde ability fly)))
(describe (assert agent Tweety act believe
                  object (build object Tweety ability fly)))
(find (member- ! class) elephant (object- ! ability) fly)
```

2.29
```
(find agent ?x act believe object (find object ?x ability fly))
*x
```

2.31
```
(describe (assert object (Tweety Dumbo) ability fly))
```

2.33
```
(describe (assert agent John act believe
                  object (build object (Opus Clyde)
                                ability fly)))

(find (member- ! class) elephant
       object- (find object- (find agent John act believe)
                     ability fly))
```

3.6
```
(describe (assert agent John act believe
                  object (build object (Orville Wilbur)
                                ability fly)))

(describe (deduce agent John act believe
                  object (build object Orville ability fly)))
```

4.6 `(find prerequisites- *cs114)`

4.8

```
(describe (assert department CS division grad
                  number 572
                  name "Introduction to Artificial Intelligence"
                  credits 3) = cs572)

(describe (assert department CS division grad
                  number 676
                  name "Knowledge Representation"
                  credits 3
                  prerequisites *cs572))

(find prerequisites- *cs572)

(find prerequisites- *cs676)

(find (name- prerequisites) *cs572)
```

5.4

```
(describe (assert member #elephant class elephant))
(describe (assert agent John act see object *elephant))
```

6.8

```
(define-path has (compose ! (kstar (compose isa object- !))
                          has (kstar (compose object- ! has))
                          (kstar (compose object- ! isa))))

(find object- (findassert has mouth))

(find has- (findassert object Dumbo))
```

7.3

```
(describe (assert member Barbara class lawyer))

(define object property)

(describe (assert forall $l
                  ant (build member *l class lawyer)
                  cq  (build object *l property rich)))

(describe (assert forall $r
                  ant (build object *r property rich)
                  cq ((build arg1 *r
                              rel owns
                              arg2 (build function house\ of arg1 *r) = house)
                      (build member *house class house)
                      (build object *house property big))))

(describe (assert forall $h
                  &ant ((build member *h class house)
                        (build object *h property big))
                  cq (build object *h property lots\ of\ work)))

(describe (deduce object $x property lots\ of\ work))
```

10 ACKNOWLEDGMENTS

Thanks to Gerald J. Erion for pointing out an error.

11 REFERENCES

1. Ginsberg, Matt (1993), *Essentials of Artificial Intelligence* (San Mateo, CA: Morgan Kaufman).

2. Rich, Elaine, & Knight, Kevin (1991), *Artificial Intelligence, 2nd edition* (New York: McGraw-Hill).

3. Shapiro, Stuart C., & the SNePS Research Group (15 August 1994), "SNePS 2.1 User's Manual" (Buffalo: SUNY Buffalo Department of Computer Science); available on-line at URL `http://www.cs.buffalo.edu/pub/SNePS/WWW/Manuals/index.html`.

4. Woods, William A. (1975), "What's in a Link: Foundations for Semantic Networks," in Bobrow, D. G., & Collins, A. (eds.), *Representation and Understanding: Studies in Cognitive Science* (New York: Academic Press): 35–82; reprinted in Brachman, Ronald J., & Levesque, Hector J. (eds.), *Readings in Knowledge Representation* (Los Altos, CA: Morgan Kaufmann, 1985): 217–241.