Using SNePS for Mathematical Cognition: A SNeRE Based Natural Language Algorithm For Computing GCD

Albert Goldfain
CSE740 Progress Report
ag33@cse.buffalo.edu

29th April 2004

Abstract

Mathematical cognition is an emerging branch of cognitive science. Recent work in mathematical cognition has stressed the link between natural language and mathematics. SNePS (the Semantic Network Processing System) has proven to be a powerful natural language representation and reasoning tool. Motivated by the techniques used in the Contextual Vocabulary Acquisition (CVA) project, we establish an intuitive, natural language based algorithm for computing the greatest common divisor (GCD) of two natural numbers. Human protocols are provided to support the intuitiveness of our algorithm. The algorithm is implemented using the framework of SNeRE (the SNePS Rational Engine). After CASSIE, the SNePS cognitive agent, performs the SNeRE plan we have given her, we ask her a series of questions to demonstrate her understanding of GCD and related concepts. Further research ideas and improvements to our algorithm are discussed. The full source code and a transcript of the running implementation are also included.

1 Introduction

Mathematics is among the oldest of human endeavors. For as long as we have been in the company of digital computers, they have been "doing" mathematics right along with us (and often times *for* us). The natural tendency in computer science has been to optimize algorithms. The faster an algorithm runs and the less space it uses, the more efficient we

consider it to be. Intuitive algorithmic solutions to many computational problems are labeled "brute force" and computer science students are often discouraged from applying them. In this computer-as-fancy-calculator context, coding the most efficient algorithms is the right approach. However, if we are attempting a faithful cognitive simulation, this is the wrong approach to take.

Often overlooked by computer scientists is the fact that "brute force" algorithms are the first ones that come to mind. They are also the most intuitive to learn and teach. Indeed, our ability to optimize an algorithm requires that we first have some non-optimal algorithms to work with. Mathematical cognition, an emerging area of research in cognitive science, is an investigation into how mathematical concepts are learned, represented and applied by cognitive agents. Digital computers are of the utmost importance to this endeavor. In this psychological, computer-as-simulated-mind context, we must favor the natural over the efficient and the intuitive over the optimized.

The recent mathematical cognition literature has emphasized the link between natural language and math:

- Weise, Heike (2003) *Numbers, Language and the Human Mind*: Illustrates the mapping of math concepts to known linguistic concepts.

- Lakoff, George and Nunez, Rafael Nuñez (2000) *Where Mathematics Comes From* Shows how metaphors play a role in our conception of difficult math concepts, such as

infinity.

- Dehaene, Stanislas (1997) *The Number Sense: How the Mind Creates Mathematics*: Deals with the psychological and linguistic aspects of our concept of "number".

We feel these linguistic links make SNePS (the Semantic Network Processing System) an ideal representation tool for a mathematical cognition task. The SNePS based cognitive agent, CASSIE, will be responsible for carrying out a math cognition task. The task we have chosen for CASSIE is finding the greatest common divisor (GCD) of two natural numbers. We use SNeRE (the SNePS rational engine) as a framework for carrying out a "Natural Language" GCD algorithm (for more information on SNePS and SNeRE, visit `http://www.cse.buffalo.edu/sneps`).

Before we describe the motivations for our work and spell out the details of our "Natural Language" GCD algorithm, it will be useful to separate what we are doing from what we are not:

WHAT WE ARE DOING

- Developing a SNeRE plan to find the greatest common divisor of two natural numbers.
- Providing CASSIE with enough background knowledge to reason about GCD and related concepts.
- CASSIE will build an internal representation (i.e. a SNePS network) as she carries out

the plan.

- After CASSIE carries out the plan, we demonstrate that she can reason about GCD by asking her appropriate questions.

WHAT WE ARE NOT DOING

- We are not implementing a LISP function which computes GCD and simply hands the answer to CASSIE.
- We are not giving CASSIE a calculator with a GCD "button". This would be an efficient solution, but CASSIE would not be able to reason about what was involved in computing GCD.

It is important to keep these distinctions in mind. A computer doing math is *NOT* a computer understanding and reasoning about math.

2 Motivations

This project developed as a spin off of the Contextual Vocabulary Acquisition project (for more information, see `http://www.cse.buffalo.edu/~rapaport/cva.html`). The technique used in CVA is as follows:

1. CASSIE is given background knowledge about a passage containing an unknown word
2. CASSIE is allowed to read (and represent) the passage containing the unknown word
3. CASSIE attempts to define the unknown word

We use a similar technique:

1. CASSIE is given background knowledge about basic arithmetic operations, mathematical properties and natural numbers.

2. CASSIE is allowed to perform a GCD algorithm on two natural numbers.

3. CASSIE attempts to "define" GCD by answering questions about GCD related concepts. Some of these concepts are known facts from step #1 and others will be formed (and possibly revised) during step #2.

Another motivation for this project comes from (Rapaport 1990:3):

> "... consider a student in an introductory computer-science course who is given a complicated Turing-machine algorithm that, unknown to her, computes the greatest common divisor (GCD) of two integers. Suppose that the student, as an exercise in using Turing machines, is given pairs of integers and asked to follow the algorithm. Suppose further (though, no doubt, this is bad pedagogy!) that she is not told that what she is doing is computing the GCD of the inputs and that she (alas!) does not even know what a GCD is. Surely, she is computing GCDs, though she does not understand that she is doing so ... Suppose that our student who is using a Turing machine to compute GCDs learns in her math class what GCDs are and is given a different algorithm in that class for computing them. Eventually, she should be able to realize that the Turing-machine algorithm is also a GCD algorithm."

For this project, we will first give CASSIE a natural algorithm (the kind she might learn in math class) before giving her a more cryptic one (for example the Euclidean algorithm). If CASSIE is able to reason that the Euclidean Algorithm "is also a GCD algorithm", she will be demonstrating one aspect of what John Searle has called "strong AI". At the time of this writing, work has not yet begun on a SNeRE plan for the Euclidean GCD algorithm.

Finally, we are motivated by the recent interest in self-aware computer systems. In his article *Systems That Know What They Are Doing*, Ron Brachman (of DARPA) has said: "The higher-order cognitive processes of reflection and self-awareness could be the key to creating systems that are not fragile in the presence of unforeseen inputs" (Brachman 2002:69). CASSIE will demonstrate an awareness of her actions by answering questions about these actions after she has completed the GCD algorithm.

3 The Many Faces of GCD

GCD is introduced very early in the math curriculum. The GCD concept was also introduced very early in the history of mathematics. One of the most well known algorithms was introduced by Euclid before the advent of algebra. We feel that the GCD problem is an ideal starting point for CASSIE's math career. There are many GCD algorithms to choose from, each at varying levels of efficiency and intuitiveness. The concept of GCD can be represented in a variety of ways, and is used as a building block for more complex mathematical concepts. Let us consider the following two definitions of GCD:

"An integer $d \neq 0$ is said to be a common divisor of a and b if $d|a$ and $d|b$. A common divisor d of an and b is said to be the greatest common divisor if $d > 0$ and if every common divisor of a and b is also a divisor of d" (Gerstein, 1996:294).

"The greatest common divisor g of b and c is the least positive value of $bx + cy$ where x and y range over all integers" (Niven, Zuckerman, Montgomery, 1991:7).

The first definition is from a discrete math textbook and the second is from a number theory textbook. We see immediately that each definition identifies a unique set of properties that determine the GCD. Both of these texts define GCD early and use the GCD concept later in proofs and other definitions. Since these texts are meant to be read cover to cover, it is advantageous for the author to select a definition for GCD that will be useful throughout the book. Thus, the discrete math version highlights the recursive nature of GCD (every common divisor is also a divisor of d) while the number theory text expresses GCD as a linear combination.

4 Human Protocols

A natural, intuitive algorithm for finding the GCD can be extracted by performing some human protocols. The following is a typical example of such a protocol. The subject has a college level understanding of mathematics and computer science, but has been out of school for a few years.

- AG: Do you know how to find the greatest common divisor of two numbers?
- JU: I think I used to, but I don't remember what divisors are.
- AG: A "divisor" of a number x is a number you can divide x by evenly.
- JU: What do you mean by "evenly"?
- AG: I mean there is no remainder when you divide.
- JU: OK.
- AG: So what is the greatest common divisor of 12 and 16.
- JU: 4.
- AG: How did you get that answer?
- JU: I just found the biggest divisor for both.
- AG: Write down the divisors of both numbers.
- JU: [Writes 1 2 3 4 6 next to 12 and 1 2 4 8 next to 16]
- AG: Now write the divisors common to both numbers
- JU: [Circles 1 2 and 4] and 4 is the biggest.

We notice that JU only needs clarification of the term "divisor". The terms "greatest" and "common" are adjectives frequently used in everyday situations. When given larger numbers (large enough so that JU could not find the GCD in his head), JU applied the technique of writing the divisors of both given input numbers and circling the greatest one that appeared in both lists. It is also interesting to note that JU does not rewrite 12 and 16 as divisors. This indicates an implicit background knowledge of the fact that every number of is divisible by itself. In other protocols, subjects did not write down the number 1 as a divisor. This was usually the result of two even numbers being given as inputs, but at least one subject indicated that 1 was the least likely candidate to be the *greatest* common divisor.

These results suggest that the subjects are in possession of a set of division background rules.

All subjects wrote down a list of divisors for at least one of the inputs when given input numbers greater than 1000. Divisor lists were always written in either ascending or descending order. The most common optimization that human subjects applied was to answer immediately when the larger input is divisible by the smaller input (an immediate response that the smaller input is the GCD is given). As expected, no subjects asked for a clarification of what was meant by the terms "greatest" and "common". With very few exceptions, subjects asked for a clarification of the term "divisor". When given large numbers, human subjects sometimes apologize for having to write down divisor lists. We speculate that such apologies are the result of the subject knowing that there are more optimal ways of finding the GCD, but not remembering exactly what those ways are. These universal features suggest the sort of algorithm would be suitable for CASSIE's first encounter with GCD.

5 A "Natural Language" Algorithm

Mathematics scares many people off by its language alone. Simple concepts are given intimidating names. Consider the following definition from Wikipedia:

> "A *bijection* (or a bijective function) is a mathematical function that is both injective ("one-to-one") and surjective ("onto")... In simple terms, a bijective function creates a *one-to-one correspondence* between its possible input values and possible output values" (`http://www.wikipedia.org`)

The concept of a bijection, when stated in the "simple terms" of a one-to-one correspondence, is quite natural. We frequently use the idea of "correspondence" outside of mathematics. When described in "non-simple" terms, it may bring the memory of a flu shot to mind! That is to say, the word "bijection" does not carry the semantic indication that "one-to-one correspondence" does. What makes one version hard and the other simple is the language used. Some math concepts use "simple" language, but in a way that provides no extra semantic information. For example, consider the term "prime number". The word "prime" is used all of the time in natural language, but my "prime rib" is not divisible by itself and one (the definition of being a prime number)!

The term "greatest common divisor" has a very natural meaning: It is the "greatest" of the "divisors" which are "common" to both of the given numbers. As we have seen from the human protocols, people who have not performed a GCD calculation for several years will only need clarification of the term "divisor". The common adjectives "greatest" and "common" need no such clarification. Thus, we treat the concept of GCD as an aggregate of CASSIE's background knowledge and definitions of the terms "greatest" "common" and "divisor". Such a natural language definition yields the following natural language algorithm:

1. Obtain the two natural number inputs, x and y
2. Make a list of divisors of x
3. Make a list of divisors of y
4. Make a list of divisors common to the lists from step #2 and step #3.

5. Select the greatest member of the list made in step #4.

6 SNeRE

The SNePS Rational Engine (SNeRE), is an integrated component of the SNePS system. SNeRE is CASSIE's framework for acting, planning and achieving goals. In SNeRE, complex acts are broken down into a set of primitive acts. All SNeRE acts can be iterated, sequenced and conditionally performed. Primitive acts in SNeRE are represented as LISP functions. SNeRE includes several built in primitive acts and allows for the user to designate their own primitive acts. These should be understood as the atomic units of action CASSIE will need to perform to achieve her goals.

We develop a SNeRE plan for the natural language GCD algorithm described in the previous section [1] . CASSIE will build up a network as she performs the steps of our algorithm. This serves as her "memory" of what she has done and will allow us to question her later.

7 Implementation

The full source code for our implementation is given in Appendix A. Here we give an overview of the representation and design.

[1] We actually do not need the full power of SNeRE since our GCD algorithm doesn't require nondeterminism or interaction with the real world (i.e. all of the actions are "mental" actions). We speculate that this will be true of most mathematical algorithms

7.1 Primitive Actions

Currently, the natural language GCD algorithm requires four user defined primitive actions:

1. `get-args`: Obtain input numbers x and y from the user, and assert that these are natural numbers.

2. `build-div-list`: Build a list of divisors for a given number.

3. `build-common-list`: Build a list of common divisors from the network built so far.

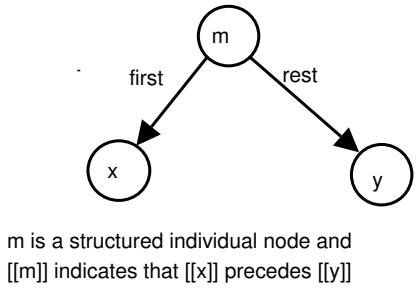
4. `get-greatest`: Retrieve the greatest element from the the common divisor list.

In addition to these, we need the following built in SNeRE primitive actions: `snsequence`, `believe`, `disbelieve`, `sniterate`, `snif` and `achieve`. Clearly the `build-div-list` action can be broken down into significantly simpler actions, but we consider it as primitive for the "first cut" of our algorithm [2] The user defined primitive actions call on external LISP functions in the file `BuildDivisorList.lisp`. These LISP functions populate CASSIE's internal network with a representation of the divisor list she is working on.

[2]Breaking down CASSIE's actions as she builds divisor lists requires her to have some background knowledge of division. One way CASSIE can obtain this knowledge is by performing another SNeRE plan where her goal is only to build divisor lists. Our attempt to teach CASSIE how to build divisor lists (which is still a work in progress) is given in Appendix A.

7.2 Representing Lists of Divisors

We handle our SNePS representation of numeric lists in a very LISP-like way, using a `first-rest` case frame.



m is a structured individual node and
[[m]] indicates that [[x]] precedes [[y]]

Figure 1: "The `first-rest` case frame"

This case frame can be built up into a list of arbitrary length. The SNePS base node `nil` indicates the end of the list. Each numeric list can be "tagged" with an assertion specifying what type of list it actually represents to CASSIE. Figure 2 shows CASSIE's SNePS representation of the divisor list of x. CASSIE builds the divisor list by checking

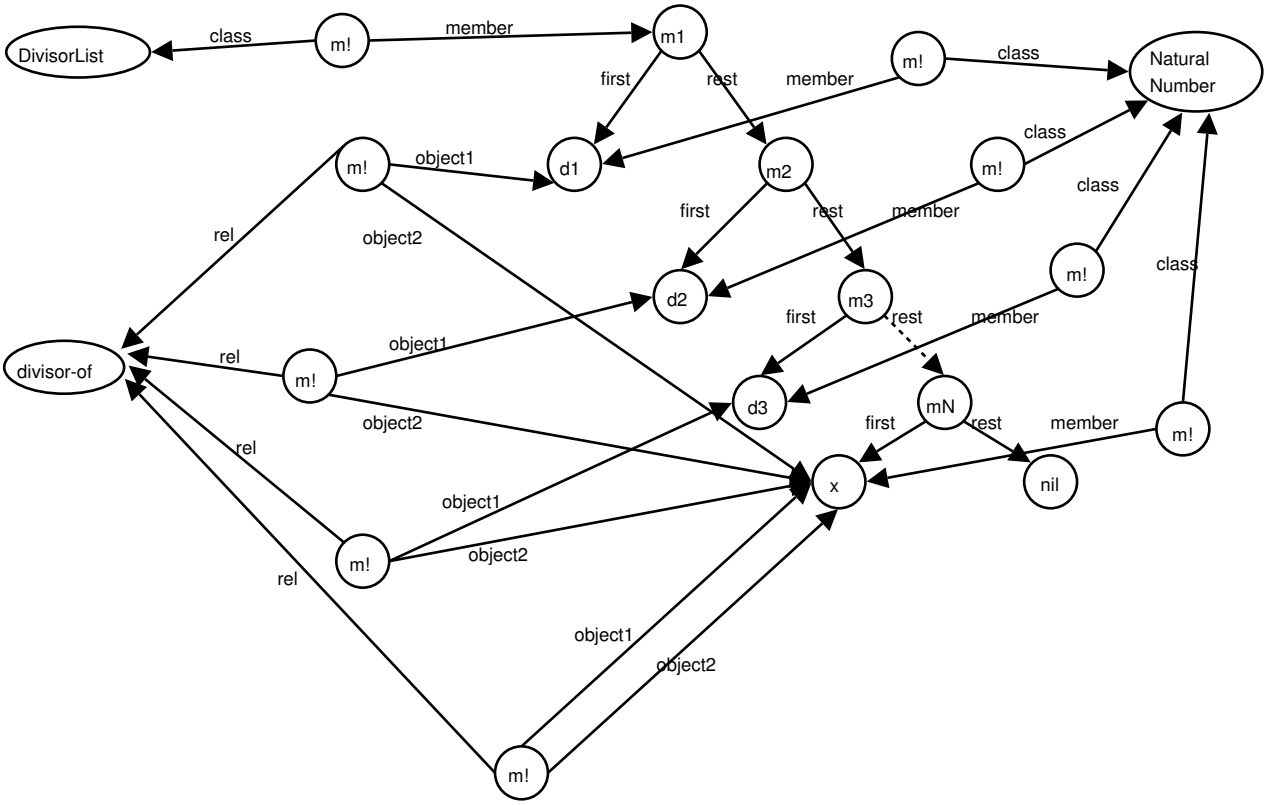


Figure 2: "CASSIE's representation of a list of divisors of x"

the remainders of $x/1, x/2 \ldots x/n$. Whenever the remainder for one of these quotients is 0, the divisor is added to the list. Our list representation explicitly represents CASSIE's knowledge that this is a divisor list (in the form of a `member-class` arc) and implicitly represents CASSIE's knowledge that it is a divisor list for x (Since x is in the `divisor-of` relation with all of the other numbers in the list). Thus CASSIE knows what she is making a list of, and the order in which elements are added to the list.

7.3 Retrieving the Common Divisors

After building the divisor lists for both of the numbers she is given, CASSIE can find the common divisors using path based reasoning. She adds the common divisors to a new list, which is designated to be a member of the class `CommonList`.

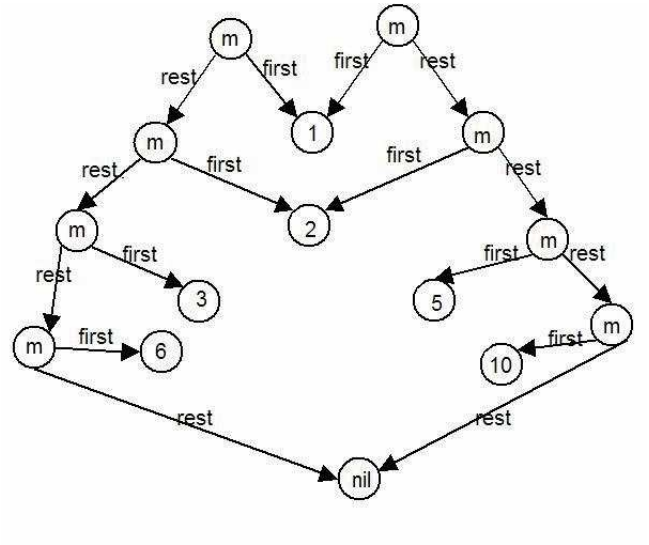


Figure 3: "The common divisors are those with two first arcs pointing to them"

7.4 Retrieving the Greatest Common Divisor

Now all that is left for CASSIE to do is to find the greatest member of the `CommonList`. Because CASSIE adds elements of the divisor lists in ascending order, the greatest member of the `CommonList` is always the last divisor added (and can thus easily be obtained using path based inference. This is fine for our present purposes. A more desirable method might be for us to tell CASSIE that she has been adding divisors to these lists in ascending order, and give her enough background knowledge to infer that the last element would therefore

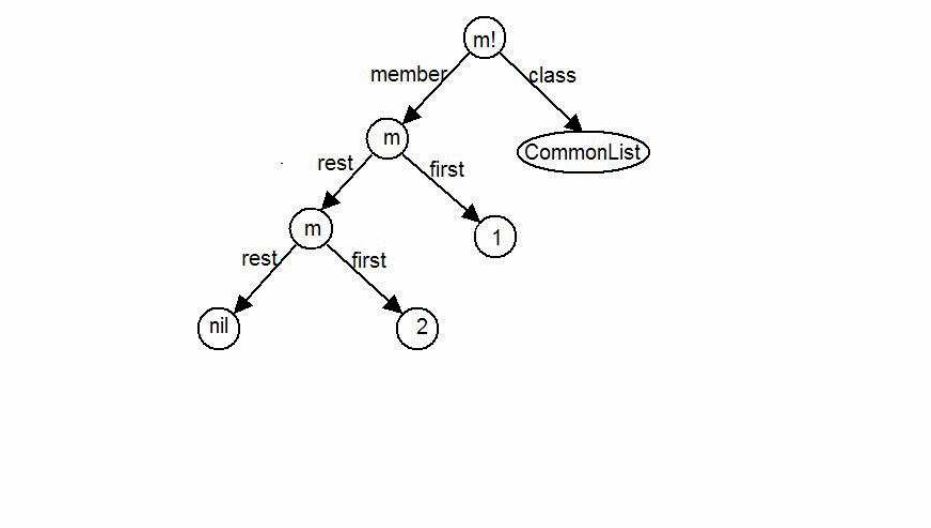


Figure 4: "The greatest common divisor is the last element in the `CommonList`"

have to be the greatest. A still more desirable method would be to "teach" CASSIE the concept of order (less than and greater than) in another SNeRE demo.

8 Asking CASSIE some questions

After CASSIE reaches the goal of her SNeRE plan, we can ask her some questions. These questions can be categorized as follows:

- SNeRE based questions. *These are motivated by the SNeRE case frames that generate CASSIE's network as she is working. Answering such questions will demonstrate that CASSIE remembers what she just did.*
  - What goals did you achieve?
  - What plans did you have for achieving those goals?
  - What actions did you perform?

- Object/Relation questions. *CASSIE can now talk about the entities she manipulated during the algorithm*
  - What natural numbers were used during the algorithm?
  - What tuples are in the divisor-of relation?
- Temporal questions. *A temporal "step-by-step" sequence was also included alongside the natural language algorithm. This will allow CASSIE to answer temporal queries*
  - Which was the first number you built a divisor list for?
- Conceptual questions. *Some of these questions may be answered in a manner that is either a declarative (e.g. "a divisor is . . . ") or procedural (e.g. "you can find a divisor by doing . . . ")*
  - What are the properties of the relation gcd-of?
  - What does it mean to be a divisor of some number?

The above questions only represent a sampling of the kinds of questions we can ask CASSIE. If we run into some question that CASSIE cannot answer, we can respond in one of three ways: (i) give CASSIE the background knowledge to answer the question (ii) change our representation/algorithm or (iii) give CASSIE a lesson in the concepts needed to answer the question. The last of these methods is preferable because it will give CASSIE a large network to reason with. This will be a network she has built by *doing*, which is what we encourage human students to do. Handing CASSIE background knowledge just "fast-forwards" this

lesson. This may need to be done as a practical matter (i.e. we may not want to generate special purpose implementations just to handle specific questions).

9 Harder Questions

CASSIE can answer the non-conceptual questions from the previous section with relative ease. That is to say, they are usually answered by doing some quick path based inference on CASSIE's existing network. To prove a deeper understanding of GCD, CASSIE might be asked some questions of a higher cognitive order. She may be asked if the GCD of two natural numbers is unique. She may be asked whether or not the GCD of any odd number will ever be 2. She may be asked to show that $GCD(x, y) = GCD(y, x)$. These sorts of questions require CASSIE to generalize beyond the natural language algorithm she has just carried out for two particular inputs.

Let us consider the question "Does $GCD(x, y) = GCD(y, x)$ for all natural numbers x and y?" How would *we* answer this question? The answer will require a formal proof since it involves "all natural numbers x and y". But, before launching into such a formal proof, a student might check that this property holds for small values of x and y. Good math students usually want to convince themselves that what they are proving is true! This is the attitude we want CASSIE to take. She performs the check for two small values in the `Symmetric.demo` file. Her background knowledge then allows her to infer that the "gcd-of" relation has the "symmetric" property (that is, GCD is symmetric with respect to the order

of its inputs). This allows CASSIE to tentatively answer the question. If some inputs x and y were found such that $GCD(x, y) \neq GCD(y, x)$ then CASSIE would have to revise her beliefs.

This is precisely the kind of information CASSIE will need if we expect her to identify that the Euclidean algorithm is "also a GCD algorithm". If CASSIE believes $NaturalLanguageGCD(x, y) = NaturalLanguageGCD(y, x)$, and she finds $EuclideanGCD(x, y) \neq EuclideanGCD(y, x)$, then she should seriously doubt that the Euclidean algorithm as an algorithm for GCD. She should still go on believing that the natural language GCD algorithm is a GCD algorithm because it is based on her natural language intuitions!

10 Natural Numbers and CASSIE's Calculator

> The kinds of concepts CASSIE can reason about (and thus, the kinds of questions she can answer) are directly determined by the actions she is allowed to take in SNeRE. Currently, CASSIE must appeal to LISP to do the basic arithmetic operations and for a representation of natural numbers. The difficulty with representing numeric information was noted early in the history of SNePS by Shapiro: "... numeric information is basically syntactic rather than semantic. This conclusion suggests that the way to provide numeric information to a semantic network is to provide it with a gracious interface to a syntactic representation" (Shapiro, 1977:284)

For our purposes, the LISP file `BuildDivisorList.lisp` is such an interface. This defines a kind of "calculator" that CASSIE is using. She sometimes needs to press a button on the calculator to obtain a result. While she may know what buttons she needs to press to get the result, and what purpose the result will serve in the overall algorithm, she will *not* know the internal workings of the calculator (i.e. LISP) "behind" that button.

Therefore, any interrogation of CASSIE must stop at the point of natural numbers:

- Albert: "What does it mean for d to be a divisor of x?"
- CASSIE: "If you divide x by d there will be no remainder."
- Albert: "What is d then?"
- CASSIE: "d is a divisor of x and d is a number."
- Albert: "What is a number?"
- CASSIE: "Something I get from LISP."

We should not be pessimistic about this restriction. Most human subjects will have a very difficult time describing what their concept of a natural number is. One could appeal to Peano's Axioms, which define the natural numbers abstractly in set theoretic terms. However, many adults can do GCD calculations, while very few adults have heard of Peano's Axioms. We therefore feel that CASSIE's final answer to the natural number question is acceptable.

It is desirable, however, to make CASSIE's calculator as simple as possible. A student

that has gone through a college level Calculus course with only a four button calculator probably understands the core concepts of calculus (as well as the background concepts from trigonometry and geometry) much better than a student who uses a modern graphing calculator. Thus, anything CASSIE can do by herself, we want to force her to do by herself. Poor CASSIE!

11 Future Work

11.1 Adding a Natural Language Frontend

As we have seen, the algorithm used in the human protocols relied on a natural language understanding of "greatest", "common" and "divisor". Since the algorithm we have given CASSIE attempts to capture this reliance, it would be quite appropriate to make CASSIE do her reasoning in natural language as well. Currently, the natural language algorithm, the background knowledge, and CASSIE's questions and answers are all given in SNePSUL (The SNePS User Language). If we add an appropriate, wide coverage grammar and GCD specific lexicon, all of these can be given in natural language. We are currently determining whether to add an ATN (Augmented Transition Network) or an LKB grammar as a frontend to our demo.

11.2 The Euclidean Algorithm

We intend to develop a SNeRE plan for the Euclidean GCD algorithm. We have already seen in section [9] how CASSIE might begin to identify this algorithm as also being a GCD algorithm, namely, checking whether the relations `nl-gcd-of` and `euclidean-gcd-of` have the same properties. General Input/Output behavior of the two algorithms is another approach CASSIE might take. A difference in input/output behavior is one of the most natural ways we identify two processes as being "different".

11.3 Teaching CASSIE Division

We have already begun breaking down the process of building a divisor list into more primitive actions. CASSIE can easily be introduced to the vocabulary of division (i.e. quotient, dividend, divisor, remainder etc.) via background knowledge, but it would be interesting to give CASSIE a task in which she must separate things evenly. This would give her a more semantic procedure with which to associate the process of division. The method of Egyptian continued fractions, in which sums of reciprocals of natural numbers are used to represent any fraction, is one such semantic interpretation for division (see `http://www.ics.uci.edu/~eppstein/numth/egypt/`).

11.4 Other Topics

Other avenues of potential research include:

- Developing a useful semantic representation of natural numbers and numerals in SNePS.

- Giving CASSIE the ability to generalize (i.e. to prove general theorems) with the SNeRE plans in hand.

- Developing a discourse system which would allow CASSIE to ask questions about concepts she is not clear about.

Appendix A: Source Code

NatLangGCD.demo

```
; ========================================================================
; FILENAME: NatLangGCD.demo
; DATE: 3/8/04
; PROGRAMMER: Albert Goldfain

; Lines beginning with a semi-colon are comments.
; Lines beginning with "^" are Lisp commands.
; All other lines are SNePS commands.
;
; To use this file: run SNePS; at the SNePS prompt (*), type:
;
; (demo "NatLangGCD.demo" :av)
;
; Make sure all necessary files are in the current working directory
; or else use full path names.
; ========================================================================

^(load "BuildDivisorList")

;TRACING OPTIONS
^(setq snip:*infertrace* nil)
^(setq snip:*plantrace* nil)
^(setq common-lisp-user::firstDone nil)


;
; NL GCD Algorithm Primitive Actions
;

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

(^ (define-primaction build-div-list(object1)
   "Build divisor list for object1"
   (setf num1 (first #!((find (member- ! class) Input1))))
   (setf num2 (first #!((find (member- ! class) Input2))))
   (if (endp common-lisp-user::firstDone)
       (setf dlist (sneps-gcd::build-divisor-list (node-to-lisp-object num1) 1 nil))
       (setf dlist (sneps-gcd::build-divisor-list (node-to-lisp-object num2) 1 nil))
   )
   (if (endp common-lisp-user::firstDone)
       (format t "~&Divisor List for ~A is ~A~%" num1 dlist)
       (format t "~&Divisor List for ~A is ~A~%" num2 dlist)
   )
   (setf common-lisp-user::firstDone '(1))
   #!((assert member ~(sneps-gcd::sneps-divisor-list dlist) class NaturalNumberList))
   )
)

(^ (define-primaction build-common-list ()
   "Build common divisor list from existing network"
   (setf clist #!((find (object1- ! rel) divisor-of (object1- ! object2)
                                                    (find (member- ! class) Input1)
                        (object1- ! rel) divisor-of (object1- ! object2)
```

```
                                                    (find (member- ! class) Input2))))
   (setf clist (ns-to-lisp-list clist))
   (setf clist (sort clist #'<))
   (format t "~&Common Divisor List is ~A.~%"  clist)
   #!((assert member ~(sneps-gcd::sneps-common-list clist) class CommonList))
   ;#!((assert member ~(sneps-gcd::sneps-common-list clist) class NaturalNumberList))
   )
)

(^ (define-primaction get-greatest ()
   "Get greatest common divisor from the common list"
   (setf gcdlist #!((find (member- ! class) CommonDivisor (first- rest) *nil)))
   (format t "~&The Greatest Common Divisor is ~A.~%" (first gcdlist))
   #!((assert object1 ~(first gcdlist)
              rel gcd-of
              object2 (find (member- ! class) Input1)
              object3 (find (member- ! class) Input2)
     ))
   )
)
;
; Mimicking Deepak Kumar's initialization of primitive actions
; Attach functions to their associated primitive action nodes:
;

(^ (attach-primaction
    ;; built-in actions:
    snsequence snsequence
    sniterate sniterate
    achieve achieve
    believe believe
    disbelieve disbelieve
    withsome withsome
    withall withall
    ;;user defined actions:
    say say
    build-div-list build-div-list
    build-common-list build-common-list
    get-greatest get-greatest))


;
; Define all the necessary rels for NL GCD Algo
;

(define rel before after time lex member class object state object3 object4 object5 object6
 object7 object8 object9 first rest dividend divisor property op arg1 arg2)

;
;LOAD Background Information
;
(intext "BGInfo")

(assert member #nil class nil)
;Temporal Sequence Definition
(assert before T1 after T2)
(assert before T2 after T3)
(assert before T3 after T4)
(assert before T4 after T5)
(assert before T5 after T6)
(assert before T6 after T7)
```

```
(assert before T7 after T8)
(assert before T8 after T9)

;-----------------------------------------------
; Complex acts and the plans for achieving them
;-----------------------------------------------

(assert forall $x
ant (build member *x class NaturalNumber)
cq (build act (build action create-divisor-list object1 *x)
plan (build action build-div-list object1 *x))
)

(assert forall ($x $y)
&ant((build member *x class NaturalNumberList)
     (build member *y class NaturalNumberList))
cq(build act (build action find-common)
plan (build action build-common-list)
  )
)

(assert forall $x
ant(build member *x class NaturalNumberList)
cq(build act (build action find-greatest)
plan (build action get-greatest)
  )
)

(assert forall ($x $y)
&ant ((build member *x class NaturalNumber)
      (build member *y class NaturalNumber))
cq (build act (build action find-gcd object1 *x object2 *y)
plan (build action snsequence
object1 (build action create-divisor-list object1 *x)
object2 (build action believe object1 (build action create-divisor-list
                                                                    object1 *x
     state complete
                                                                    time T1))
object3 (build action create-divisor-list object1 *y)
object4 (build action believe object1 (build action create-divisor-list
                                                                    object1 *y
     state complete
     time T2))
object5 (build action find-common)
                                object6 (build action believe object1 (build action find-common
                                                                    state complete
                                                                    time T3))
object7 (build action find-greatest)
object8 (build action believe object1 (build action find-greatest
     state complete
                                                                    time T4))
object9 (build action believe object1 (build object gcd state found)))
   )
)

;-------------------------------------------------------------------
; The SNeRE NL GCD Algorithm is now given as a goal-plan case frame
; ------------------------------------------------------------------

;CASSIE initially believes that all (sub)goals are not yet reached
;(assert min 0 max 0 arg (build object divisor-lists state built))
```

```
;(assert min 0 max 0 arg (build object common-divisors state found))
(assert min 0 max 0 arg (build object gcd state found))

(assert goal (build object gcd state found)
       plan (build action find-gcd object1 (find (member- ! class) Input1)
   object2 (find (member- ! class) Input2))
)

;Invoke achieve action
(perform (build action achieve object1 (build object gcd state found)))
```

InputArgs.demo

```
^(setq snip:*infertrace* nil)
^(setq snip:*plantrace* nil)

(resetnet t)

(^ (define-primaction getArgs ()
   "Obtain NLGCD algorithm arguments from the user"
(format t "~%I will perform the Natural Language GCD Algorithm on two inputs you give me~%")
(format t "~%Please enter the first input number~%")
(setf i1 (read))
(format t "~%Please enter the second input number~%")
(setf i2 (read))
#!((assert member ~i1 class Input1))
#!((assert member ~i2 class Input2))
(if (and (numberp i1)(numberp i2))
    #!((assert member (~i1 ~i2) class NaturalNumber))
    (format t "~%Sorry.  I am expecting two natural numbers.~%")
)
   )
)

(^ (attach-primaction
getArgs getArgs))

(define member class)
(perform (build action getArgs))
```

BuildDivisorList.lisp

```
(defpackage sneps-gcd
  (:shadow build-divisor-list sneps-divisor-list sneps-common-list)
  (:use common-lisp)
  (:export build-divisor-list sneps-divisor-list sneps-common-list)
)
(in-package sneps-gcd)

(defun build-divisor-list (dividend divisor divisor-list)
  (if (and(numberp dividend)(numberp divisor))
    (if (= dividend divisor)
(setf divisor-list (cons divisor divisor-list))
        (if (zerop (rem dividend divisor))
    (setf divisor-list (cons divisor (build-divisor-list dividend (+ 1 divisor) divisor-list)))
    (setf divisor-list (build-divisor-list dividend (+ 1 divisor) divisor-list))
)
    )
    (format t "~%Error: build-divisor-list needs numeric arguments~%")
```

```
  )
  divisor-list
)

(defun sneps-divisor-list-base (x)
  #!((assert object1 ~(first x) rel divisor-of object2 ~(last x)))
  #4!((build first ~(first x) rest *nil))
)

(defun sneps-divisor-list-recursive (x)
  #!((assert object1 ~(first x) rel divisor-of object2 ~(last x)))
  #4!((build first ~(first x) rest ~(sneps-divisor-list (rest x))))
)


(defun sneps-divisor-list (x)
  #!((assert member ~(first x) class NaturalNumber))
  (cond ((= 0 (length x)) #4!((build rest *nil)))
        ((= 1 (length x)) (sneps-divisor-list-base x))
        (t (sneps-divisor-list-recursive x))
  )
)

(defun sneps-common-list (x)
 #!((assert member ~(first x) class CommonDivisor))
 (if (= 1 (length x))
     #4!((build first ~(first x) rest *nil))
     #4!((build first ~(first x) rest ~(sneps-common-list (rest x))))
 )
)
```

BGInfo

```
;Some background knowledge about divisors and division
;-------------------------------------------------------------
; If x is divided by y then a quotient and a remainder are
; generated.
;-------------------------------------------------------------
(assert forall ($x $y)
ant (build action divide object1 *x object2 *y)
cq (build object1 #remainder rel remainder-of object2 *x object3 *y)
)

(assert forall ($x $y)
ant (build action divide object1 *x object2 *y)
cq (build object1 #quotient rel quotient-of object2 *x object3 *y)
)

;-------------------------------------------------------------
; If x is a divisor of y then y is divisible by x
;-------------------------------------------------------------
(assert forall ($x $y)
ant (build object1 *x rel divisor-of object2 *y)
cq (build object1 *y rel divisible-by object2 *x)
)


;-------------------------------------------------------------
; If x is a divisor of y then y mod x = 0
;-------------------------------------------------------------
(assert forall ($x $y)
```

```
ant (build object1 *x rel divisor-of object2 *y)
cq (build object1 0 rel mod-of object2 *y object3 *x)
)

;---------------------------------------------------------------
; If y mod z = x then x is the remainder of y divided by z
;---------------------------------------------------------------
(assert forall ($x $y $z)
ant (build object1 *x rel mod-of object2 *y object3 *z)
cq (build object1 *x rel remainder-of object2 *y object3 *z ))
)
;---------------------------------------------------------------
; Symmetric Property of Relations
;---------------------------------------------------------------
(assert forall ($w $x $y $z)
&ant ((build object1 *w rel *x object2 *y object3 *z)
              (build object1 *w rel *x object2 *z object3 *y))
cq (build rel *x property symmetric)
)
```

AskCassieNLGCD.demo

```
;DEDUCE SOME THINGS FROM BG KNOWLEDGE
(deduce (object1) $x (rel) divisible-by (object2) $y)
(deduce (object1) $x (rel) mod-of (object2) $y (object3) $z)
;(deduce (object1) $x (rel) remainder-of (object2) $y (object3) $z)


;NOW ASK CASSIE SOME QUESTIONS.
;"What were your goals?"
(describe (deduce (goal) $x))
;"What plans did you have for finding the gcd?"
(describe (find (plan- goal) (build object gcd state found)))
;"What actions did you complete?"
(find (action- state) complete)
;"What numbers did you find divisors for?"
(findbase (object1- action) create-divisor-list)
;"When did you build the divisor list for the first input?"
(find (time-) (find (action) create-divisor-list (object1) (find (member- ! class) Input1)))
;"When did you build the divisor list for the second input?"
(find (time-) (find (action) create-divisor-list (object1) (find (member- ! class) Input2)))
;"What was the input for which you first built a divisor list?"
(findbase (object1-) (find (action) create-divisor-list
                       (time) (find (before- after) (find (time- action) create-divisor-list))))
;"Is 2 a divisor of the first input?"
(find (object1) 2 (rel) divisor-of (object2) (find (member- ! class) Input1))
;"What natural numbers do you know?"
(findbase
 (member- class) NaturalNumber)
;"What tuples are in the divisor-of relation?
(describe (deduce (object1) $x (rel) divisor-of (object2) $y))
```

Symmetric.demo

```
(resetnet t)
(define member class)
(assert member (6 8) class NaturalNumber)
(assert member 6 class Input1)
(assert member 8 class Input2)
(demo "NatLangGCD.demo")
```

```
(perform (build action disbelieve object1 (build object gcd state found)))
(erase (deduce member 6 class Input1))
(erase (deduce member 8 class Input2))
(assert member 8 class Input1)
(assert member 6 class Input2)
(demo "NatLangGCD.demo")
(deduce (rel) gcd-of (property) $x)
(findbase (rel- property) symmetric)
```

DivisorList.demo (IN PROGRESS)

```
; ============================================================================
; FILENAME: DivisorList.demo
; DATE: 4/15/04
; PROGRAMMER: Albert Goldfain

; Lines beginning with a semi-colon are comments.
; Lines beginning with "^" are Lisp commands.
; All other lines are SNePS commands.
;
; To use this file: run SNePS; at the SNePS prompt (*), type:
;
; (demo "DivisorList.demo" :av)
;
; Make sure all necessary files are in the current working directory
; or else use full path names.
; ============================================================================
^(load "Calculator.lisp")


;TRACING OPTIONS
^(setq snip:*infertrace* nil)
^(setq snip:*plantrace* t)


;=============================================================
; PRIMITIVE ACTIONS
;=============================================================

(^ (define-primaction calc-add (object1 object2)
    "adds object1 and object2"
    (setf x (node-to-lisp-object (sneps:choose.ns object1)))
    (setf y (node-to-lisp-object (sneps:choose.ns object2)))
(format t "~&Entering ~A + ~A on my calculator.~%" x y)
    (setf sum (sneps-calculator::plus-button x y))
(format t "~&CALCULATOR READS: ~A~%" sum)
    #!((assert result ~(lisp-object-to-node sum)
               op calc-add
               arg1 ~(sneps:choose.ns object1)
               arg2 ~(sneps:choose.ns object2)))
#!((assert member ~(lisp-object-to-node sum) class NaturalNumber))
   )
)

(^ (define-primaction calc-mod (object1 object2)
    "finds the remainder of object1 divided by object2"
    (setf modarg1 (node-to-lisp-object (sneps:choose.ns object1)))
    (setf modarg2 (node-to-lisp-object (sneps:choose.ns object2)))
(format t "~&Entering ~A MOD ~A on my calculator.~%" modarg1 modarg2)
    (setf theMod (sneps-calculator::mod-button modarg1 modarg2))
(format t "~&CALCULATOR READS: ~A~%" theMod)
```

```
    #!((assert result ~(lisp-object-to-node theMod)
               op calc-mod
               arg1 ~(sneps:choose.ns object1)
               arg2 ~(sneps:choose.ns object2)))
#!((assert member ~(lisp-object-to-node theMod) class NaturalNumber))
#!((assert member ~(lisp-object-to-node theMod) class CurrentRemainder))
   )
)

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

;(^ (define-primaction start-list(object1 object2)
; "creates a new object2 list for object1"
; (format t "~&Creating a ~A list for ~A"
; (sneps:choose.ns object2)
; (sneps:choose.ns object1)
; )
; #!((assert object1 ~(sneps:choose.ns object1)
;                    rel ~(sneps:choose.ns object2)
;                    object2 (build rest *nil)
;          ))
;   )
;)



;(^ (define-primaction add-to-list(object1 object2)
;    "adds object1 to the list of object2"
;    (setf x (node-to-lisp-object (sneps:choose.ns object1)))
;    (setf y (node-to-lisp-object (sneps:choose.ns object2)))
; (format t "~&Finding the mod of ~A and ~A~%" x y)
;    (setf theMod (sneps-calculator::mod-button x y))
;    #!((assert result ~(lisp-object-to-node theMod)
;               op calc-mod
;               arg1 ~(sneps:choose.ns object1)
;               arg2 ~(sneps:choose.ns object2)))
;   )
;)



(^ (attach-primaction
    ;; built-in actions:
    snsequence snsequence
    sniterate sniterate
    snif snif
    achieve achieve
    believe believe
    disbelieve disbelieve
    withsome withsome
    withall withall
    ;;user defined actions:
    calc-add calc-add
    calc-mod calc-mod
    say say))
```

```
(define rel before after time lex member class object state object3 object4 object5 object6
 object7 object8 object9 first rest dividend divisor property op arg1 arg2 result)

(assert member #nil class nil)

;=============================================================
; update-num-var
;=============================================================
(assert forall ($theVar $oldValue $newValue)
ant (build member *newValue class NaturalNumber)
cq (build act (build action update-num-var object1 *theVar object2 *oldValue object3 *newValue)
plan (build action snsequence
         object1 (build action disbelieve
   object1 (build member *oldValue
  class *theVar))
    object2 (build action believe
   object1 (build member *newValue
  class *theVar))
     )
   )
)

;=============================================================
; update-divisor-candidate
;=============================================================
(assert forall ($oldC $newC)
&ant ((build member *oldC class DivisorCandidate)
      (build member *newC class NewDivisorCandidate))
cq (build act (build action update-divisor-candidate)
plan (build action snsequence
         object1 (build action disbelieve
   object1 (build member *oldC
  class DivisorCandidate))
    object2 (build action believe
   object1 (build member *newC
  class DivisorCandidate))
    object3 (build action disbelieve
   object1 (build member *newC
  class NewDivisorCandidate))
     )
   )
)
;=============================================================
; increment
;=============================================================
(assert forall $x
ant (build member *x class NaturalNumber)
cq (build act (build action increment object1 *x)
plan (build action calc-add object1 1 object2 *x)
   )
)

;=============================================================
; store-remainder
;=============================================================
(assert forall ($x $y $z)
&ant ((build member *x class NaturalNumber)
      (build member *y class DivisorCandidate)
      (build member *z class NaturalNumber))  ;TEMP
cq (build act (build action store-remainder)
plan  (build action believe object1 (build member *z class CurrentRemainder))
```

```
   )
)
;=============================================================
; check-divisible
;=============================================================
(assert forall ($x $y)
&ant ((build member *x class NaturalNumber)
      (build member *y class DivisorCandidate))
cq (build act (build action check-divisible object1 *x object2 *y)
 plan (build action snsequence
     object1 (build action calc-mod object1 *x object2 *y)
     object2 (build action store-remainder)
     object3 (build action snif
    object1 ((build condition (build member 0 class CurrentRemainder)
    then (build action snsequence
object1 (build action say object1 "Found a divisor")
object2 (build action believe
object1 (build object1 *y
               rel divisor-of
                 object2 *x))
                                                                    )
     )
    (build else (build action believe
object1 (build min 0 max 0
 arg (build object1 *y
    rel divisor-of
    object2 *x))
                                                                    )
                                                          ))
                                        )
     object4 (build action disbelieve
    object1 (build member (find (result- ! op) calc-mod
                                                                          (result- ! arg1) *x
                                                                          (result- ! arg2) *y)

   class CurrentRemainder)
             )
      )
    )
)


;=============================================================
; set-new-divisor-candidate
;=============================================================
(assert forall $x
ant (build member *x class DivisorCandidate)
cq (build act (build action set-new-divisor-candidate)
 plan (build action believe
       object1 (build member (find (result- ! op) calc-add
      (result- ! arg1) 1
      (result- ! arg2) *x)
      class NewDivisorCandidate
       )
                     )
   )
)



;=============================================================
; build-divisor-list
```

```
;=============================================================
(assert forall ($x $y)
&ant ((build member *x class NaturalNumber)
      (build member *y class DivisorCandidate))
cq (build act (build action build-divisor-list object1 *x)
 plan (build action sniterate
     object1 ((build
                                     condition (build min 0 max 0 arg (build object divisor-list
                                                                          state built))
     then (build action snsequence
   object1 (build action check-divisible object1 *x object2 *y)
   object2 (build action increment object1 *y)
   object3 (build action set-new-divisor-candidate)
   object4 (build action update-divisor-candidate))

     ))

                )
            )
)

;=============================================================
; build-divisor-lists
;=============================================================
(assert forall ($x $y $z)
&ant ((build member *x class NaturalNumber)
      (build member *y class NaturalNumber))
cq (build act (build action build-divisor-lists object1 *x object2 *y)
 plan (build action snsequence
object1 (build action believe
                                              object1 (build min 0 max 0 arg
                                                            (build object divisor-list
                                                                         state built))
                                         )
object2 (build action believe
       object1 (build member 1 class DivisorCandidate)
                                         )
object3 (build action build-divisor-list object1 *x)
object4 (build action believe
       object1 (build member 1 class DivisorCandidate)
                                         )
object5 (build action build-divisor-list object1 *y)
object6 (build action believe object1 (build object divisor-lists state found))
      )
   )
)

(assert goal (build object divisor-lists state found)
plan (build action build-divisor-lists object1 (find (member- ! class) Input1)
       object2 (find (member- ! class) Input2))
)

(assert member 1 class NaturalNumber)
;(assert member 9999 class CurrentRemainder)
(assert min 0 max 0 arg (build object divisor-lists state found))
(perform (build action achieve object1 (build object divisor-lists state found)))
```

Calculator.lisp (IN PROGRESS)

```
(defpackage sneps-calculator
  (:shadow plus-button minus-button times-button divide-button mod-button)
  (:use common-lisp)
  (:export plus-button minus-button times-button divide-button mod-button)
)
(in-package sneps-calculator)

(defun plus-button (x y)
  (+ x y)
)

(defun minus-button (x y)
  (- x y)
)

(defun times-button (x y)
  (* x y)
)

(defun divide-button (x y)
  (/ x y)
)

(defun mod-button (x y)
  (mod x y)
)
```

Appendix B: Running Demo

```
Script started on Thu Apr 29 12:04:45 2004
pollux {~/cse740/GCD_project} > acl
International Allegro CL Enterprise Edition
6.2 [Solaris] (Oct 28, 2003 9:00)
Copyright (C) 1985-2002, Franz Inc., Berkeley, CA, USA.  All Rights Reserved.

This development copy of Allegro CL is licensed to:
   [4549] SUNY/Buffalo, N. Campus

;; Optimization settings: safety 1, space 1, speed 1, debug 2.
;; For a complete description of all compiler switches given the
;; current optimization settings evaluate (explain-compiler-settings).
;;---
;; Current reader case mode: :case-sensitive-lower
cl-user(1): (load "/projects/snwiz/bin/sneps")
; Loading /projects/snwiz/bin/sneps.lisp
Loading system SNePS...10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
SNePS-2.6 [PL:0a 2002/09/30 22:37:46] loaded.
Type `(sneps)' or `(snepslog)' to get started.
t
cl-user(2): (sneps)


   Welcome to SNePS-2.6 [PL:0a 2002/09/30 22:37:46]

Copyright (C) 1984--2002 by Research Foundation of
State University of New York. SNePS comes with ABSOLUTELY NO WARRANTY!
Type `(copyright)' for detailed copyright information.
Type `(demo)' for a list of example applications.

   4/29/2004 12:05:48

* (demo "NLGCDALL.demo")

File /home/csgrad/ag33/cse740/GCD_project/NLGCDALL.demo is now the source of input.


 CPU time : 0.00

* (demo "InputArgs.demo")

File /home/csgrad/ag33/cse740/GCD_project/InputArgs.demo is now the source of input.


 CPU time : 0.00

*
^(
--> setq snip:*infertrace* nil)
nil



 CPU time : 0.00

* ^(
--> setq snip:*plantrace* nil)
nil
```

```
 CPU time : 0.00

*
(resetnet t)

Net reset


 CPU time : 0.00

*
(^ (define-primaction getArgs ()
   "Obtain NLGCD algorithm arguments from the user"
        (format t "~%I will perform the Natural Language GCD Algorithm on two inputs you give me~%")
        (format t "~%Please enter the first input number~%")
        (setf i1 (read))
        (format t "~%Please enter the second input number~%")
        (setf i2 (read))
        #!((assert member ~i1 class Input1))
        #!((assert member ~i2 class Input2))
        (if (and (numberp i1)(numberp i2))
            #!((assert member (~i1 ~i2) class NaturalNumber))
            (format t "~%Sorry.  I am expecting two natural numbers.~%")
        )
   )
)

(getArgs)

 CPU time : 0.00

*
(^ (attach-primaction
        getArgs getArgs))

(t)

 CPU time : 0.00

*
(define member class)

(member class)

 CPU time : 0.00

* (perform (build action getArgs))

I will perform the Natural Language GCD Algorithm on two inputs you give me

Please enter the first input number
8

Please enter the second input number
6


 CPU time : 0.00
```

```
*

End of /home/csgrad/ag33/cse740/GCD_project/InputArgs.demo demonstration.


 CPU time : 0.02

* (demo "NatLangGCD.demo")

File /home/csgrad/ag33/cse740/GCD_project/NatLangGCD.demo is now the source of input.


 CPU time : 0.00

* ; ====================================================================
; FILENAME:     NatLangGCD.demo
; DATE:         3/8/04
; PROGRAMMER:   Albert Goldfain

; Lines beginning with a semi-colon are comments.
; Lines beginning with "^" are Lisp commands.
; All other lines are SNePS commands.
;
; To use this file: run SNePS; at the SNePS prompt (*), type:
;
;       (demo "NatLangGCD.demo" :av)
;
; Make sure all necessary files are in the current working directory
; or else use full path names.
; ====================================================================

^(
--> load "BuildDivisorList")
; Loading /home/csgrad/ag33/cse740/GCD_project/BuildDivisorList.lisp
t



 CPU time : 0.01

*
;TRACING OPTIONS
^(
--> setq snip:*infertrace* nil)
nil



 CPU time : 0.00

* ^(
--> setq snip:*plantrace* nil)
nil



 CPU time : 0.00

* ^(
--> setq common-lisp-user::firstDone nil)
nil
```

```
 CPU time : 0.03

*

;
; NL GCD Algorithm Primitive Actions
;

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

(say)

 CPU time : 0.00

*
(^ (define-primaction build-div-list(object1)
   "Build divisor list for object1"
   (setf num1 (first #!((find (member- ! class) Input1))))
   (setf num2 (first #!((find (member- ! class) Input2))))
   (if (endp common-lisp-user::firstDone)
       (setf dlist (sneps-gcd::build-divisor-list (node-to-lisp-object num1) 1 nil))
       (setf dlist (sneps-gcd::build-divisor-list (node-to-lisp-object num2) 1 nil))
   )
   (if (endp common-lisp-user::firstDone)
       (format t "~&Divisor List for ~A is ~A~%" num1 dlist)
       (format t "~&Divisor List for ~A is ~A~%" num2 dlist)
   )
   (setf common-lisp-user::firstDone '(1))
   #!((assert member ~(sneps-gcd::sneps-divisor-list dlist) class NaturalNumberList))
   )
)

(build-div-list)

 CPU time : 0.00

*
(^ (define-primaction build-common-list ()
   "Build common divisor list from existing network"
   (setf clist #!((find (object1- ! rel) divisor-of (object1- ! object2)
                                                    (find (member- ! class) Input1)
                        (object1- ! rel) divisor-of (object1- ! object2)
                                                    (find (member- ! class) Input2))))
   (setf clist (ns-to-lisp-list clist))
   (setf clist (reverse clist))
   (format t "~&Common Divisor List is ~A.~%"  clist)
   #!((assert member ~(sneps-gcd::sneps-common-list clist) class CommonList))
   ;#!((assert member ~(sneps-gcd::sneps-common-list clist) class NaturalNumberList))
   )
)

(build-common-list)

 CPU time : 0.00
```

```
*
(^ (define-primaction get-greatest ()
   "Get greatest common divisor from the common list"
   (setf gcdlist #!((find (member- ! class) CommonDivisor (first- rest) *nil)))
   (format t "~&The Greatest Common Divisor is ~A.~%" (first gcdlist))
   #!((assert object1 ~(first gcdlist)
              rel gcd-of
              object2 (find (member- ! class) Input1)
              object3 (find (member- ! class) Input2)
     ))
   )
)

(get-greatest)

 CPU time : 0.00

* ;
; Mimicking Deepak Kumar's initialization of primitive actions
; Attach functions to their associated primitive action nodes:
;

(^ (attach-primaction
    ;; built-in actions:
    snsequence snsequence
    sniterate sniterate
    achieve achieve
    believe believe
    disbelieve disbelieve
    withsome withsome
    withall withall
    ;;user defined actions:
    say say
    build-div-list build-div-list
    build-common-list build-common-list
    get-greatest get-greatest))

(t)

 CPU time : 0.00

*

;
; Define all the necessary rels for NL GCD Algo
;

(define rel before after time lex member class object state object3 object4 object5 object6
 object7 object8 object9 first rest dividend divisor property)
member is already defined.
class is already defined.

(rel before after time lex member class object state object3 object4
 object5 object6 object7 object8 object9 first rest dividend divisor
 property)

 CPU time : 0.01

*
;
```

```
;LOAD Background Information
;
(intext "BGInfo")
File BGInfo is now the source of input.


 CPU time : 0.00

*

(m5!)

 CPU time : 0.00

*

(m6!)

 CPU time : 0.01

*

(m7!)

 CPU time : 0.00

*

(m8!)

 CPU time : 0.00

*

(m9!)

 CPU time : 0.00

*
Warning: ignoring extra right parenthesis on
         #<file-simple-stream #p"BGInfo" for input pos 1477>

(m10!)

 CPU time : 0.00

*

End of file BGInfo


 CPU time : 0.02

*
(assert member #nil class nil)

(m11!)

 CPU time : 0.00

* ;Temporal Sequence Definition
```

```
(assert before T1 after T2)

(m12!)

 CPU time : 0.00

* (assert before T2 after T3)

(m13!)

 CPU time : 0.00

* (assert before T3 after T4)

(m14!)

 CPU time : 0.00

* (assert before T4 after T5)

(m15!)

 CPU time : 0.00

* (assert before T5 after T6)

(m16!)

 CPU time : 0.00

* (assert before T6 after T7)

(m17!)

 CPU time : 0.00

* (assert before T7 after T8)

(m18!)

 CPU time : 0.01

* (assert before T8 after T9)

(m19!)

 CPU time : 0.00

*
;-----------------------------------------------
; Complex acts and the plans for achieving them
;-----------------------------------------------

(assert forall $x
        ant (build member *x class NaturalNumber)
        cq (build act (build action create-divisor-list object1 *x)
                plan (build action build-div-list object1 *x))
)

(m20!)
```

```
 CPU time : 0.00

*

(assert forall ($x $y)
        &ant((build member *x class NaturalNumberList)
             (build member *y class NaturalNumberList))
        cq(build act (build action find-common)
                plan (build action build-common-list)
          )
)

(m24!)

 CPU time : 0.00

*
(assert forall $x
        ant(build member *x class NaturalNumberList)
        cq(build act (build action find-greatest)
                plan (build action get-greatest)
          )
)

(m28!)

 CPU time : 0.01

*
(assert forall ($x $y)
        &ant ((build member *x class NaturalNumber)
              (build member *y class NaturalNumber))
        cq (build act (build action find-gcd object1 *x object2 *y)
                plan (build action snsequence
                                object1 (build action create-divisor-list object1 *x)
                                object2 (build action believe object1 (build action create-divisor-list
                                                                        object1 *x
                                                                        state complete
                                                                        time T1))
                                object3 (build action create-divisor-list object1 *y)
                                object4 (build action believe object1 (build action create-divisor-list
                                                                        object1 *y
                                                                        state complete
                                                                        time T2))
                                object5 (build action find-common)
                                object6 (build action believe object1 (build action find-common
                                                                        state complete
                                                                        time T3))
                                object7 (build action find-greatest)
                                object8 (build action believe object1 (build action find-greatest
                                                                        state complete
                                                                        time T4))
                                object9 (build action believe object1 (build object gcd state found)))
           )
)

(m35!)

 CPU time : 0.00

*
```

```
;-------------------------------------------------------------------
; The SNeRE NL GCD Algorithm is now given as a goal-plan case frame
; -------------------------------------------------------------------

;CASSIE initially believes that all (sub)goals are not yet reached
;(assert min 0 max 0 arg (build object divisor-lists state built))
;(assert min 0 max 0 arg (build object common-divisors state found))
(assert min 0 max 0 arg (build object gcd state found))

(m36!)

 CPU time : 0.01

*
(assert goal (build object gcd state found)
       plan (build action find-gcd object1 (find (member- ! class) Input1)
                                   object2 (find (member- ! class) Input2))
)

(m38!)

 CPU time : 0.00

*
;Invoke achieve action
(perform (build action achieve object1 (build object gcd state found)))
Divisor List for 8 is (1 2 4 8)
Divisor List for 6 is (1 2 3 6)
Common Divisor List is (1 2).
The Greatest Common Divisor is 2.


 CPU time : 0.45

*



End of /home/csgrad/ag33/cse740/GCD_project/NatLangGCD.demo demonstration.


 CPU time : 0.57

* (demo "AskCassieNLGCD.demo" :av)

File /home/csgrad/ag33/cse740/GCD_project/AskCassieNLGCD.demo is now the source of input.

  The demo will pause between commands, at that time press
  RETURN to continue, or ? to see a list of available commands


 CPU time : 0.01

* ;DEDUCE SOME THINGS FROM BG KNOWLEDGE
(deduce (object1) $x (rel) divisible-by (object2) $y)
--- pause ---

(m93! m92! m91! m90! m89! m88!)

 CPU time : 0.01
```

```
* (deduce (object1) $x (rel) mod-of (object2) $y (object3) $z)
--- pause ---

(m99! m98! m97! m96! m95! m94!)

 CPU time : 0.02

* ;(deduce (object1) $x (rel) remainder-of (object2) $y (object3) $z)


;NOW ASK CASSIE SOME QUESTIONS.
;"What were your goals?"
(describe (deduce (goal) $x))
--- pause ---

(m100! (goal (m33! (object gcd) (state found))))

(m100!)

 CPU time : 0.02

*

;"What plans did you have for finding the gcd?"
;OLD: (describe (find (plan- goal) (build object gcd state found)))
(describe (deduce (plan) $x (goal) (build object gcd state found)))
--- pause ---

(m38! (goal (m33! (object gcd) (state found)))
 (plan (m37 (action find-gcd) (object1 8) (object2 6))))

(m38!)

 CPU time : 0.01

*
;"What actions did you complete?"
;OLD: (find (action- state) complete)
;WANT: (describe (deduce (action) $x (state) complete)) THROWS AN INTERNAL ERROR
(find (action- ! state) complete)
--- pause ---

(find-greatest find-common create-divisor-list)

 CPU time : 0.00

* ;"What numbers did you find divisors for?"
(findbase (object1- action) create-divisor-list)
--- pause ---

(6 8)

 CPU time : 0.00

* ;"When did you build the divisor list for the first input?"
(find (time-) (find (action) create-divisor-list (object1) (find (member- ! class) Input1)))
--- pause ---

(T1)
```

```
 CPU time : 0.00

* ;"When did you build the divisor list for the second input?"
(find (time-) (find (action) create-divisor-list (object1) (find (member- ! class) Input2)))
--- pause ---

(T2)

 CPU time : 0.00

* ;"What was the input for which you first built a divisor list?"
(findbase (object1-) (find (action) create-divisor-list
                      (time) (find (before- after) (find (time- action) create-divisor-list))))
--- pause ---

(8)

 CPU time : 0.00

* ;"Is 2 a divisor of the first input?"
;OLD: (find (object1) 2 (rel) divisor-of (object2) (find (member- ! class) Input1))
(describe (deduce (object1) 2 (rel) divisor-of (object2) (find (member- ! class) Input1)))
--- pause ---

(m58! (object1 2) (object2 8) (rel divisor-of))

(m58!)

 CPU time : 0.00

* ;"What natural numbers do you know?"
;OLD: (findbase (member- ! class) NaturalNumber)
(describe (deduce (member) $x (class) NaturalNumber))
--- pause ---

(m72! (class NaturalNumber) (member 3))
(m59! (class NaturalNumber) (member 4))
(m57! (class NaturalNumber) (member 2))
(m55! (class NaturalNumber) (member 1))
(m50! (class NaturalNumber) (member 8))
(m49! (class NaturalNumber) (member 6))

(m72! m59! m57! m55! m50! m49!)

 CPU time : 0.01

*
;"What tuples are in the divisor-of relation?
(describe (deduce (object1) $x (rel) divisor-of (object2) $y))
--- pause ---

(m73! (object1 3) (object2 6) (rel divisor-of))
(m71! (object1 2) (object2 6) (rel divisor-of))
(m70! (object1 1) (object2 6) (rel divisor-of))
(m60! (object1 4) (object2 8) (rel divisor-of))
(m58! (object1 2) (object2 8) (rel divisor-of))
(m56! (object1 1) (object2 8) (rel divisor-of))

(m73! m71! m70! m60! m58! m56!)

 CPU time : 0.03
```

```
*

End of /home/csgrad/ag33/cse740/GCD_project/AskCassieNLGCD.demo demonstration.

 CPU time : 0.16

*

End of /home/csgrad/ag33/cse740/GCD_project/NLGCDALL.demo demonstration.

 CPU time : 0.76

* (demo "Symmetric.demo")

File /home/csgrad/ag33/cse740/GCD_project/Symmetric.demo is now the source of input.

 CPU time : 0.00

* (resetnet t)

Net reset

 CPU time : 0.01

* (define member class)

(member class)

 CPU time : 0.00

* (assert member (6 8) class NaturalNumber)

(m1!)

 CPU time : 0.01

* (assert member 6 class Input1)

(m2!)

 CPU time : 0.00

* (assert member 8 class Input2)

(m3!)

 CPU time : 0.00

* (demo "NatLangGCD.demo")

File /home/csgrad/ag33/cse740/GCD_project/NatLangGCD.demo is now the source of input.

 CPU time : 0.00
```

```
* ; ============================================================================
; FILENAME:     NatLangGCD.demo
; DATE:         3/8/04
; PROGRAMMER:   Albert Goldfain

; Lines beginning with a semi-colon are comments.
; Lines beginning with "^" are Lisp commands.
; All other lines are SNePS commands.
;
; To use this file: run SNePS; at the SNePS prompt (*), type:
;
;       (demo "NatLangGCD.demo" :av)
;
; Make sure all necessary files are in the current working directory
; or else use full path names.
; ============================================================================

^(
--> load "BuildDivisorList")
; Loading /home/csgrad/ag33/cse740/GCD_project/BuildDivisorList.lisp
t



 CPU time : 0.01

*
;TRACING OPTIONS
^(
--> setq snip:*infertrace* nil)
nil



 CPU time : 0.00

* ^(
--> setq snip:*plantrace* nil)
nil



 CPU time : 0.00

* ^(
--> setq common-lisp-user::firstDone nil)
nil



 CPU time : 0.00

*

;
; NL GCD Algorithm Primitive Actions
;

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

```
        (sneps:choose.ns object1)
        (sneps:choose.ns object2))))

(say)

 CPU time : 0.00

*
(^ (define-primaction build-div-list(object1)
   "Build divisor list for object1"
   (setf num1 (first #!((find (member- ! class) Input1))))
   (setf num2 (first #!((find (member- ! class) Input2))))
   (if (endp common-lisp-user::firstDone)
       (setf dlist (sneps-gcd::build-divisor-list (node-to-lisp-object num1) 1 nil))
       (setf dlist (sneps-gcd::build-divisor-list (node-to-lisp-object num2) 1 nil))
   )
   (if (endp common-lisp-user::firstDone)
       (format t "~&Divisor List for ~A is ~A~%" num1 dlist)
       (format t "~&Divisor List for ~A is ~A~%" num2 dlist)
   )
   (setf common-lisp-user::firstDone '(1))
   #!((assert member ~(sneps-gcd::sneps-divisor-list dlist) class NaturalNumberList))
   )
)

(build-div-list)

 CPU time : 0.00

*
(^ (define-primaction build-common-list ()
   "Build common divisor list from existing network"
   (setf clist #!((find (object1- ! rel) divisor-of (object1- ! object2)
                                                    (find (member- ! class) Input1)
                        (object1- ! rel) divisor-of (object1- ! object2)
                                                    (find (member- ! class) Input2))))
   (setf clist (ns-to-lisp-list clist))
   (setf clist (reverse clist))
   (format t "~&Common Divisor List is ~A.~%"  clist)
   #!((assert member ~(sneps-gcd::sneps-common-list clist) class CommonList))
   ;#!((assert member ~(sneps-gcd::sneps-common-list clist) class NaturalNumberList))
   )
)

(build-common-list)

 CPU time : 0.00

*
(^ (define-primaction get-greatest ()
   "Get greatest common divisor from the common list"
   (setf gcdlist #!((find (member- ! class) CommonDivisor (first- rest) *nil)))
   (format t "~&The Greatest Common Divisor is ~A.~%" (first gcdlist))
   #!((assert object1 ~(first gcdlist)
              rel gcd-of
              object2 (find (member- ! class) Input1)
              object3 (find (member- ! class) Input2)
     ))
   )
)
```

```
(get-greatest)

 CPU time : 0.00

* ;
; Mimicking Deepak Kumar's initialization of primitive actions
; Attach functions to their associated primitive action nodes:
;

(^ (attach-primaction
    ;; built-in actions:
    snsequence snsequence
    sniterate sniterate
    achieve achieve
    believe believe
    disbelieve disbelieve
    withsome withsome
    withall withall
    ;;user defined actions:
    say say
    build-div-list build-div-list
    build-common-list build-common-list
    get-greatest get-greatest))

(t)

 CPU time : 0.01

*

;
; Define all the necessary rels for NL GCD Algo
;

(define rel before after time lex member class object state object3 object4 object5 object6
 object7 object8 object9 first rest dividend divisor property)
member is already defined.
class is already defined.

(rel before after time lex member class object state object3 object4
 object5 object6 object7 object8 object9 first rest dividend divisor
 property)

 CPU time : 0.01

*
;
;LOAD Background Information
;
(intext "BGInfo")
File BGInfo is now the source of input.


 CPU time : 0.00

*

(m4!)

 CPU time : 0.00
```

```
*

(m5!)

 CPU time : 0.00

*

(m6!)

 CPU time : 0.00

*

(m7!)

 CPU time : 0.00

*

(m8!)

 CPU time : 0.00

*
Warning: ignoring extra right parenthesis on
         #<file-simple-stream #p"BGInfo" for input pos 1477>

(m9!)

 CPU time : 0.00

*

End of file BGInfo


 CPU time : 0.02

*
(assert member #nil class nil)

(m10!)

 CPU time : 0.00

* ;Temporal Sequence Definition
(assert before T1 after T2)

(m11!)

 CPU time : 0.01

* (assert before T2 after T3)

(m12!)

 CPU time : 0.00

* (assert before T3 after T4)
```

```
(m13!)

 CPU time : 0.00

* (assert before T4 after T5)

(m14!)

 CPU time : 0.00

* (assert before T5 after T6)

(m15!)

 CPU time : 0.00

* (assert before T6 after T7)

(m16!)

 CPU time : 0.00

* (assert before T7 after T8)

(m17!)

 CPU time : 0.01

* (assert before T8 after T9)

(m18!)

 CPU time : 0.00

*
;-----------------------------------------------
; Complex acts and the plans for achieving them
;-----------------------------------------------

(assert forall $x
        ant (build member *x class NaturalNumber)
        cq (build act (build action create-divisor-list object1 *x)
                plan (build action build-div-list object1 *x))
)

(m19!)

 CPU time : 0.00

*

(assert forall ($x $y)
        &ant((build member *x class NaturalNumberList)
             (build member *y class NaturalNumberList))
        cq(build act (build action find-common)
                plan (build action build-common-list)
          )
)

(m23!)
```

```
 CPU time : 0.00

*
(assert forall $x
        ant(build member *x class NaturalNumberList)
        cq(build act (build action find-greatest)
                plan (build action get-greatest)
          )
)

(m27!)

 CPU time : 0.00

*
(assert forall ($x $y)
        &ant ((build member *x class NaturalNumber)
              (build member *y class NaturalNumber))
        cq (build act (build action find-gcd object1 *x object2 *y)
                plan (build action snsequence
                                object1 (build action create-divisor-list object1 *x)
                                object2 (build action believe
                                                object1 (build action create-divisor-list
                                                                object1 *x
                                                                state complete
                                                                time T1))
                                object3 (build action create-divisor-list object1 *y)
                                object4 (build action believe
                                                object1 (build action create-divisor-list
                                                                object1 *y
                                                                state complete
                                                                time T2))
                                object5 (build action find-common)
                                object6 (build action believe
                                                       object1 (build action find-common
                                                                       state complete
                                                                       time T3))
                                object7 (build action find-greatest)
                                object8 (build action believe
                                                object1 (build action find-greatest
                                                                state complete
                                                                time T4))
                                object9 (build action believe
                                                object1 (build object gcd state found)))
           )
)

(m34!)

 CPU time : 0.00

*
;-------------------------------------------------------------------
; The SNeRE NL GCD Algorithm is now given as a goal-plan case frame
; -------------------------------------------------------------------

;CASSIE initially believes that all (sub)goals are not yet reached
;(assert min 0 max 0 arg (build object divisor-lists state built))
;(assert min 0 max 0 arg (build object common-divisors state found))
(assert min 0 max 0 arg (build object gcd state found))
```

```
(m35!)

 CPU time : 0.00

*
(assert goal (build object gcd state found)
       plan (build action find-gcd object1 (find (member- ! class) Input1)
                                   object2 (find (member- ! class) Input2))
)

(m37!)

 CPU time : 0.00

*
;Invoke achieve action
(perform (build action achieve object1 (build object gcd state found)))
Divisor List for 6 is (1 2 3 6)
Divisor List for 8 is (1 2 4 8)
Common Divisor List is (1 2).
The Greatest Common Divisor is 2.


 CPU time : 0.25

*



End of /home/csgrad/ag33/cse740/GCD_project/NatLangGCD.demo demonstration.


 CPU time : 0.38

* (perform (build action disbelieve object1 (build object gcd state found)))


 CPU time : 0.02

* (erase (deduce member 6 class Input1))

(m2 class (Input1) member (6))
node dismantled.


 CPU time : 0.01

* (erase (deduce member 8 class Input2))

(m3 class (Input2) member (8))
node dismantled.


 CPU time : 0.04

* (assert member 8 class Input1)

(m88!)

 CPU time : 0.01
```

```
* (assert member 6 class Input2)

(m89!)

 CPU time : 0.00

* (demo "NatLangGCD.demo")

File /home/csgrad/ag33/cse740/GCD_project/NatLangGCD.demo is now the source of input.


 CPU time : 0.00

* ; ============================================================================
; FILENAME:     NatLangGCD.demo
; DATE:         3/8/04
; PROGRAMMER:   Albert Goldfain

; Lines beginning with a semi-colon are comments.
; Lines beginning with "^" are Lisp commands.
; All other lines are SNePS commands.
;
; To use this file: run SNePS; at the SNePS prompt (*), type:
;
;       (demo "NatLangGCD.demo" :av)
;
; Make sure all necessary files are in the current working directory
; or else use full path names.
; ============================================================================

^(
--> load "BuildDivisorList")
; Loading /home/csgrad/ag33/cse740/GCD_project/BuildDivisorList.lisp
t



 CPU time : 0.01

*
;TRACING OPTIONS
^(
--> setq snip:*infertrace* nil)
nil



 CPU time : 0.00

* ^(
--> setq snip:*plantrace* nil)
nil



 CPU time : 0.00

* ^(
--> setq common-lisp-user::firstDone nil)
nil
```

```
 CPU time : 0.01

*

;
; NL GCD Algorithm Primitive Actions
;

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

(say)

 CPU time : 0.00

*
(^ (define-primaction build-div-list(object1)
   "Build divisor list for object1"
   (setf num1 (first #!((find (member- ! class) Input1))))
   (setf num2 (first #!((find (member- ! class) Input2))))
   (if (endp common-lisp-user::firstDone)
       (setf dlist (sneps-gcd::build-divisor-list (node-to-lisp-object num1) 1 nil))
       (setf dlist (sneps-gcd::build-divisor-list (node-to-lisp-object num2) 1 nil))
   )
   (if (endp common-lisp-user::firstDone)
       (format t "~&Divisor List for ~A is ~A~%" num1 dlist)
       (format t "~&Divisor List for ~A is ~A~%" num2 dlist)
   )
   (setf common-lisp-user::firstDone '(1))
   #!((assert member ~(sneps-gcd::sneps-divisor-list dlist) class NaturalNumberList))
   )
)

(build-div-list)

 CPU time : 0.00

*
(^ (define-primaction build-common-list ()
   "Build common divisor list from existing network"
   (setf clist #!((find (object1- ! rel) divisor-of (object1- ! object2)
                                                    (find (member- ! class) Input1)
                        (object1- ! rel) divisor-of (object1- ! object2)
                                                    (find (member- ! class) Input2))))
   (setf clist (ns-to-lisp-list clist))
   (setf clist (reverse clist))
   (format t "~&Common Divisor List is ~A.~%"  clist)
   #!((assert member ~(sneps-gcd::sneps-common-list clist) class CommonList))
   ;#!((assert member ~(sneps-gcd::sneps-common-list clist) class NaturalNumberList))
   )
)

(build-common-list)

 CPU time : 0.00
```

```
*
(^ (define-primaction get-greatest ()
   "Get greatest common divisor from the common list"
   (setf gcdlist #!((find (member- ! class) CommonDivisor (first- rest) *nil)))
   (format t "~&The Greatest Common Divisor is ~A.~%" (first gcdlist))
   #!((assert object1 ~(first gcdlist)
              rel gcd-of
              object2 (find (member- ! class) Input1)
              object3 (find (member- ! class) Input2)
     ))
   )
)

(get-greatest)

 CPU time : 0.00

* ;
; Mimicking Deepak Kumar's initialization of primitive actions
; Attach functions to their associated primitive action nodes:
;

(^ (attach-primaction
    ;; built-in actions:
    snsequence snsequence
    sniterate sniterate
    achieve achieve
    believe believe
    disbelieve disbelieve
    withsome withsome
    withall withall
    ;;user defined actions:
    say say
    build-div-list build-div-list
    build-common-list build-common-list
    get-greatest get-greatest))

(t)

 CPU time : 0.00

*

;
; Define all the necessary rels for NL GCD Algo
;

(define rel before after time lex member class object state object3 object4 object5 object6
 object7 object8 object9 first rest dividend divisor property)
rel is already defined.
before is already defined.
after is already defined.
time is already defined.
lex is already defined.
member is already defined.
class is already defined.
object is already defined.
state is already defined.
object3 is already defined.
object4 is already defined.
```

```
object5 is already defined.
object6 is already defined.
object7 is already defined.
object8 is already defined.
object9 is already defined.
first is already defined.
rest is already defined.
dividend is already defined.
divisor is already defined.
property is already defined.

(rel before after time lex member class object state object3 object4
 object5 object6 object7 object8 object9 first rest dividend divisor
 property)

 CPU time : 0.00

*
;
;LOAD Background Information
;
(intext "BGInfo")
File BGInfo is now the source of input.


 CPU time : 0.01

*

(m90!)

 CPU time : 0.00

*

(m91!)

 CPU time : 0.00

*

(m92!)

 CPU time : 0.01

*

(m93!)

 CPU time : 0.00

*

(m94!)

 CPU time : 0.00

*
Warning: ignoring extra right parenthesis on
         #<file-simple-stream #p"BGInfo" for input pos 1477>
```

```
(m95!)

 CPU time : 0.00

*

End of file BGInfo


 CPU time : 0.03

*
(assert member #nil class nil)

(m96!)

 CPU time : 0.00

* ;Temporal Sequence Definition
(assert before T1 after T2)

(m11!)

 CPU time : 0.00

* (assert before T2 after T3)

(m12!)

 CPU time : 0.00

* (assert before T3 after T4)

(m13!)

 CPU time : 0.00

* (assert before T4 after T5)

(m14!)

 CPU time : 0.00

* (assert before T5 after T6)

(m15!)

 CPU time : 0.00

* (assert before T6 after T7)

(m16!)

 CPU time : 0.00

* (assert before T7 after T8)

(m17!)

 CPU time : 0.00
```

```
* (assert before T8 after T9)

(m18!)

 CPU time : 0.00

*
;-----------------------------------------------
; Complex acts and the plans for achieving them
;-----------------------------------------------

(assert forall $x
        ant (build member *x class NaturalNumber)
        cq (build act (build action create-divisor-list object1 *x)
                plan (build action build-div-list object1 *x))
)

(m97!)

 CPU time : 0.00

*

(assert forall ($x $y)
        &ant((build member *x class NaturalNumberList)
             (build member *y class NaturalNumberList))
        cq(build act (build action find-common)
                plan (build action build-common-list)
          )
)

(m98!)

 CPU time : 0.01

*
(assert forall $x
        ant(build member *x class NaturalNumberList)
        cq(build act (build action find-greatest)
                plan (build action get-greatest)
          )
)

(m99!)

 CPU time : 0.00

*
(assert forall ($x $y)
        &ant ((build member *x class NaturalNumber)
              (build member *y class NaturalNumber))
        cq (build act (build action find-gcd object1 *x object2 *y)
                plan (build action snsequence
                                object1 (build action create-divisor-list object1 *x)
                                object2 (build action believe
                                                object1 (build action create-divisor-list
                                                                object1 *x
                                                                state complete
                                                                time T1))
                                object3 (build action create-divisor-list object1 *y)
                                object4 (build action believe
```

```
                                        object1 (build action create-divisor-list
                                                        object1 *y
                                                        state complete
                                                        time T2))
                                object5 (build action find-common)
                                object6 (build action believe
                                                object1 (build action find-common
                                                        state complete
                                                        time T3))
                                object7 (build action find-greatest)
                                object8 (build action believe
                                                object1 (build action find-greatest
                                                        state complete
                                                        time T4))
                                object9 (build action believe
                                                        object1 (build object gcd state found)))
          )
)

(m100!)

 CPU time : 0.01

*
;---------------------------------------------------------------------
; The SNeRE NL GCD Algorithm is now given as a goal-plan case frame
; ---------------------------------------------------------------------

;CASSIE initially believes that all (sub)goals are not yet reached
;(assert min 0 max 0 arg (build object divisor-lists state built))
;(assert min 0 max 0 arg (build object common-divisors state found))
(assert min 0 max 0 arg (build object gcd state found))

(m35!)

 CPU time : 0.01

*
(assert goal (build object gcd state found)
       plan (build action find-gcd object1 (find (member- ! class) Input1)
                                   object2 (find (member- ! class) Input2))
)

(m102!)

 CPU time : 0.00

*
;Invoke achieve action
(perform (build action achieve object1 (build object gcd state found)))
Divisor List for 8 is (1 2 4 8)
Divisor List for 6 is (1 2 3 6)
Common Divisor List is (1 2).
The Greatest Common Divisor is 2.


 CPU time : 0.37

*
```

```
End of /home/csgrad/ag33/cse740/GCD_project/NatLangGCD.demo demonstration.


 CPU time : 0.47

* (deduce (rel) gcd-of (property) $x)

(m130!)

 CPU time : 0.09

* (findbase (rel- property) symmetric)

(gcd-of)

 CPU time : 0.00

*

End of /home/csgrad/ag33/cse740/GCD_project/Symmetric.demo demonstration.


 CPU time : 1.04

* (lisp)
"End of SNePS"
cl-user(3): :exit
; Exiting Lisp
pollux {~/cse740/GCD_project} > exit
exit

script done on Thu Apr 29 12:06:54 2004
```

Appendix C: Open SNePS Issues

- CASSIE cannot use `deduce` when given the SNeRE relation `action`. The current workaround is to use `find` instead.
- CASSIE evaluates each `deduce` and `find` in a SNeRE plan before executing that plan. This "early binding" of variables causes CASSIE to ignore late in an `snsequence` nodes which are built earlier in that same `snsequence`.

References

1. Brachman, Ronald J. (2002), "Systems That Know What They're Doing", *IEEE Intelligent Systems* November/December 2002: 67-71.

2. Epstein, David (2004), "Egyptian Fractions", [`http://www.ics.uci.edu/~eppstein/numth/egypt/`]

3. Gerstein, Larry J. (1996) *Introduction to Mathematical Structures and Proofs*, (New York:Springer-Verlag).

4. Niven,Iven; Zuckerman, Herbert S.; Montgomery, Hugh L.(1991) *An Introduction to the Theory of Numbers, Fifth Edition*, (New York:John Wiley & Sons, Inc.).

5. Rapaport, William (1990) *Computer Processes and Virtual Persons: Comments on Cole's Artificial Intelligence and Personal Identity*, Technical Report 90-13, Buffalo: SUNY Buffalo Department of Computer Science.

6. Shapiro, S.C. (1977) *Representing numbers in semantic networks: prolegomena.* In Proceedings of the 5th International Joint Conference on Artificial Intelligence, (Los Altos: Morgan Kaufmann)