USING λ-CALCULUS TO REPRESENT MEANINGS IN LOGIC GRAMMARS*

David Scott Warren

Computer Science Department
SUNY at Stony Brook
Stony Brook, NY 11794

ABSTRACT

This paper describes how meanings are represented in a semantic grammar for a fragment of English in the logic programming language Prolog. The conventions of Definite Clause Grammars are used. Previous work on DCGs with a semantic component has used essentially first-order formulas for representing meanings. The system described here uses formulas of the typed λ-calculus. The first section discusses general issues concerning the use of first-order logic or the λ-calculus to represent meanings. The second section describes how λ-calculus meaning representations can be constructed and manipulated directly in Prolog. This 'programmed' representation motivates a suggestion, discussed in the third section, for an extension to Prolog so that the language itself would include a mechanism for handling the λ-formulas directly.

I λ-CALCULUS AND FOL AS MEANING REPRESENTATION LANGUAGES

The initial phase of most computer programs for processing natural language is a translation system. This phase takes the English text input and transforms it into structures in some internal meaning-representation language. Most of these systems fall into one of two groups: those that use a variant of first-order logic (FOL) as their representation language, and those that use the typed λ-calculus (LC) for their representation language. (Systems based on semantic nets or conceptual dependency structures would generally be calssified as using variants of FOL, but see [Jones and Warren, 1982] for an approach that views them as LC-based.)

The system considered here are several highly formalized grammar systems that concentrate on the translation of sentences of logical form. The first-order logic systems are exemplified by those systems that have developed around (or gravitated to) logic programming, and the Prolog language in particular. These include the systems described in [Colmerauer 1982], [Warren 1981], [Dahl 1981], [Simmons and Chester 1982], and [McCord 1982]. The systems using the λ- calculus are those that developed out of the work of Richard Montague. They include the systems described in [Montague 1973], [Gawron et al. 1982], [Rosenschein and Sheiber 1982], [Schubert and Pelletier 1982], and [Warren and Friedman 1981]. For the purposes of this paper, no distinction is made between the intensional logic of Montague grammar and the typed λ-calculus. There is a mapping from intensional logic to a subset of a typed λ-calculus [Gallin 1975], [Clifford 1981] that shows they are essentially equivalent in expressive power.

All these grammar systems construct a formula to represent the meaning of a sentence compositionally over the syntax tree for the sentence. They all use syntax directed translation. This is done by first associating a meaning structure with each word. Then phrases are constructed by syntactically combining smaller phrases together using syntactic rules. Corresponding to each syntactic rule is a semantic rule, that forms the meaning structure for a compound phrase by combinging the meaning structures of the component phrases. This is clearly and explicitly the program used in Montague grammar. It is also the program used in Prolog-based natural language grammars with a semantic component; the Prolog language itself essentially forces this methodology.

Let us consider more carefully the meaning structures for the two classes of systems of interest here: those based on FOL and those based on LC.

Each of the FOL systems, given a declarative sentence as input, produces a well-formed formula in a first-order logic to represent the meaning of the sentence. This meaning representation logic will be called the MRFOL. The MRFOL has an intended interpretation based on the real world. For example, individual variables range over objects in the world and unary predicate symbols are interpreted as properties holding of those real world objects.

As a particular recent example, consider Dahl's system [1981]. Essentially the same approach was used in the Lunar System [Woods, et al. 1972]. For the sentence 'Every man walks', Dahl's system would produce the expression:

```
for(X,and(man(X),not walk(X)),
   equal(card(X),0))
```

where X is a variable that ranges over real-world

* This material is based upon work supported by the National Science Foundation under grant #IST-80-10834

individuals. This is a formula in Dahl's MRFOL, and illustrates her meaning representation language. The formula can be paraphrased as "the X's which man is true of and walk is not true of have cardinality zero." It is essentially first-order because the variables range over individuals. (There would need to be some translation for the card function to work correctly.) This example also shows how Dahl uses a formula in her MRFOL as the meaning structure for a declarative sentence. The meaning of the English sentence is identified with the meaning that the formula has in the intended interpretations for the MRFOL.

Consider now the meaning structure Dahl uses for phrases of a category other than sentence, a noun phrase, for example. For the meaning of a noun phrase, Dahl uses a structure consisting of three components: a variable, and two 'formulas'. As an example, the noun phrase 'every man' has the following triple for its meaning structure:

```
[X1,X2,for(X1,and(man(X1),not(X2)),
   equal(card(X1),0))].
```

We can understand this structure informally by thinking of the third component as representing the meaning of 'every man'. It is an object that needs a verb phrase meaning in order to become a sentence. The X2 stands for that verb-phrase meaning. For example, during construction of the meaning of a sentence containing this noun phrase as the subject, the meaning of the verb-phrase of the sentence will be bound to X2. Notice that the components of this meaning structure are not themselves formulas in the MRFOL. They look very much like FOL formulas that represent meanings, but on closer inspection of the variables, we find that they cannot be. X2 in the third component is in the position of a formula, not a term; 'not' applies to truth values, not to individuals. Thus X2 cannot be a variable in the MRFOL, because X2 would have to vary over truth values, and all FOL variables vary over individuals. So the third component is not itself a MRFOL formula that (in conjunction with the first two components) represents the meaning of the noun phrase, 'every man'.

The intuitive meaning here is clear. The third component is a formula fragment that participates in the final formula ultimately representing the meaning of the entire sentence of which this phrase is a subpart. The way this fragment participates is indicated in part by the variable X2. It is important to notice that X2 is, in fact, a syntactic variable that varies over formulas, i.e., it varies over certain terms in the MRFOL. X2 will have as its value a formula with a free variable in it: a verb-phrase waiting for a subject. The X1 in the first component indicates what the free variable must become to match this noun phrase correctly. Consider the operation of putting X1 into the verb-phrase formula and this into the noun-phrase formula when a final sentence meaning is constructed. In whatever order this is done, there must be an operation of substitution a formula with a free variable (X1) in it, into the scope of a quantifier ('for') that captures it. Semantically this is certainly a dubious operation.

The point here is not that this system is wrong or necessarily deficient. Rather the representation language used to represent meanings for subsentential components is not precisely the MRFOL. Meaning structures built for subcomponents are, in general, fragments of first-order formulas with some extra notation to be used in further formula construction. This means, in general, that the meanings of subsentential phrases are not given a semantics by first-order model theory; the meanings of intermediate phrases are (as far as traditional first-order logic is concerned) merely uninterpreted data structures.

The point is that the system is building terms, syntactic objects, that will eventually be put together to represent meanings of sentences. This works because these terms, the ones ultimately associated with sentences, always turn out to be formulas in the MRFOL in just the right way. However, some of the terms it builds on the way to a sentence, terms that correspond to subcomponents of the sentence, are not in the MRFOL, and so do not have a interpretation in its real world model.

Next let us move to a consideration of those systems which use the typed λ-calculus (LC) as their meaning representation language. Consider again the simple sentence 'Every man walks'. The grammar of [Montague 1973] associates with this sentence the meaning:

```
forall(X,implies(man(X),walk(X)))
```

(We use an extensional fragment here for simplicity.) This formula looks very much like the first-order formula given above by the Dahl system for the same sentence. This formula, also, is a formula of the typed λ-calculus (FOL is a subset of LC). Now consider a noun phrase and its associated meaning structure in the LC framework. For 'every man' the meaning structure is:

```
λ(P,forall(X,implies(man(X),P(X))))
```

This meaning structure is a formula in the λ-calculus. As such it has an interpretation in the intended model for the LC, just as any other formula in the language has. This interpretation is a function from properties to truth-values; it takes properties that hold of every man to 'true' and all other properties to 'false'. This shows that in the LC framework, sentences and subsentential phrases are given meanings in the same way, whereas in FOL systems only the sentences have meanings. Meaning structures for sentences are well-formed LC formulas of type truth-value; those for other phrases are well-formed LC terms of other types.

Consider this λ-formula for 'every man' and compare it with the three-tuple meaning structure built for it in the Dahl system. The λ-variable P plays a corresponding role to the X2 variable of the triple; its ultimate value comes from a verb-phrase meaning encountered elsewhere in the sentence.

First-order logic is not quite expressive

enough to represent directly the meanings of the categories of phrases that can be subcomponents of sentences. In systems based on first-order logic, this limitation is handled by explicitly constructing fragments of formulas, with extra notation to indicate how they must later combine with other fragments to form a true first-order formula that correctly represents the meaning of the entire sentence. In some sense the construction of the semantic representation is entirely syntactic until the full sentence meaning structure is constructed, at which point it comes to a form that does have a semantic interpretation. In contrast, in systems that use the typed λ-calculus, actual formulas of the formal language are used at each step, the language of the λ-calculus is never left, and the building of the semantic representation can actually be understood as operations on semantic objects.

The general idea of how to handle the example sentence 'Every man walks' in the two systems is essentially the same. The major difference is how this idea is expressed in the available languages. The LC system can express the entire idea in its meaning representation language, because the typed λ-calculus is a more expressive language.

The obvious question to ask is whether there is any need for semantically interpretable meaning representations at the subsentential level. One important reason is that to do formal deduction on subsentential components, their meanings must be represented in a formal meaning representation language. LC provides such a language and FOL does not. And one thing the field seems to have learned from experience in natural language processing is that inferencing is useful at all levels of processing, from words to entire texts. This points us toward something like the LC. The problem, of course, is that because the LC is so expressive, deduction in the full LC is extremely difficult. Some problems which are decidable in FOL become undecidable in the λ-calculus; some problems that are semi-decidable in FOL do not even have partial decision procedures in the LC. It is certainly clear that each language has limitations; the FOL is not quite expressive enough, and the LC is much too powerful. With this in mind, we next look at some of the implications of trying to use the LC as the meaning representation language in a Prolog system.

II LC IN PROLOG

PROLOG is extremely attractive as a language for expressing grammars. Metamorphosis Grammars [Colmerauer 1978] and Definite Clause Grammars (DCGs) [Pereira and Warren 1980] are essentially conventions for representing grammars as logic programs. DCGs can perhaps most easily be understood as an improved version of the Augmented Transition Network language [Woods 1970]. Other work on natural language in the PROLOG framework has used first-order meaning representation languages. The rest of this paper explores the implications of using the λ-calculus as the meaning representation language for a system written in PROLOG using the DCG conventions.

The following paragraphs describe a system that includes a very small grammar. The point of this system is to investigate the use of PROLOG to construct meanings with the λ-calculus as the meaning representation language, and not to explore questions of linguistic coverage. The grammar is based on the grammar of [Montague 1973], but is entirely extensional. Including intensionality would present no new problems in principle.

The idea is very simple. Each nonterminal in the grammar becomes a three-place predicate in the Prolog program. The second and third places indicate locations in the input string, and are normally suppressed when DCGs are displayed. The first place is the LC formula representing the meaning of the spanned syntactic component.

Lambda-formulas are represented by Prolog terms. The crucial decision is how to represent variables in the λ-formulas. One 'pure' way is to use a Prolog function symbol, say lvar, of one argument, an integer. Then lvar(37) would represent a λ-variable. For our purposes, we need not explicitly encode the type of λ-terms, since all the formulas that are constructed are correctly typed. For other purposes it might be desirable to encode explicitly the type in a second argument of lvar. Constants could easily be represented using another function symbol, lcon. Its first argument would identify the constant. A second argument could encode its type, if desired. Application of a λ-term to another is represented using the Prolog function symbol lapply, which has two argument places, the first for the function term, the second for the argument term. Lambda abstraction is represented using a function symbol λ with two arguments: the λ-variable, and the function body. Other commonly used connectives, such as 'and' and 'or', are represented by similarly named function symbols with the appropriate number of argument places. With this encoding scheme, the λ-term:

$$\lambda P(\exists x(man(x) \ \& \ P(x))$$

would be represented by the (perhaps somewhat awkward-looking) Prolog term:

```
lambda(lvar(3),lthereis(lvar(1),land(
            lapply(lcon(man),lvar(1))
            lapply(lvar(3),lvar(1))
               )))
```

λ-reduction would be coded as a predicate lreduce (Form, Reduced), whose first argument is an arbitrary λ-formula, and second is its λ-reduced form.

This encoding requires one to generate new variables to create variants of terms in order to avoid collisions of λ-variables. The normal way to avoid collisions is with a global 'gensym' counter, to insure the same variable is never used twice. One way to do this in Prolog is to include

a place for the counter in each grammar predicate. This can be done by including a parameter which will always be of the form gensym(Left,Right), where Left is the value of the gensym counter at the left end of the phrase spanned by the predicate and Right is the value at the right end. Any use of a λ-variable in building a λ-formula uses the counter and bumps it.

An alternative and more efficient way to encode λ-terms as Prolog terms involves using Prolog variables for λ-variables. This makes the substitution trival, essentially using Prolog's built-in facility for manipulating variables. It does, however, require the use of Prolog's meta-logical predicate var to test whether a Prolog variable is currently instantiated to a variable. This is necessary to prevent the λ-variables from being used by Prolog as Prolog variables. In the example below, we use Prolog variables for λ-variables and also modify the lcon function encoding of constants, and let constants stand for themselves. This results in a need to use the meta-logical predicate atom. This encoding scheme might best be considered as an efficiency hack to use Prolog's built-in variable-handling facilities to speed the λ-reduction.

We give below the Prolog program that represents a small example grammar with a few rules. This shows how meaning structures can be represented as λ-formulas and manipulated in Prolog. Notice the simple, regular structure of the rules. Each consists of a sequence of grammar predicates that constructs the meanings of the subcomponents, followed by an instance of the lreduce predicate that constructs the compound meaning from the component meanings and λ-reduces the result. The syntactic manipulation of the formulas, which results for example in the relatively simple formula for the sentence 'Every man walks' shown above, is done in the λ-reduction performed by the lreduce predicate.

```
/*
   */
    ts(M,X,Y) :-
         te(M1,X,Z).
         iv(M2,Z,Y),
         lreduce(lapply(M1,M2),M).

    te(M,X,Y) :-
         det(M1,X,Z),
         cn(M2,Z,Y),
         lreduce(lapply(M1,M2),M).

     te(lambda(P,lapply(P,j)),[john|X],X).

     cn(man,[man|X],X).
     cn(woman,[woman|X],X).

     det(lambda(P,lambda(Q,lforall(Z,
          limplies(lapply(P,Z),lapply(Q,Z))))),
         [every|X],X).

     iv(M,X,Y) :-
         tv(M1,X,Z),
         te(M2,Z,Y),
         lreduce(lapply(M1,M2),M).

     iv(walk,[walks|X],X).

     tv(lambda(P,lambda(Q,lapply(P,
          lambda(Y,lapply(lapply(love,Y),Q))))),
         [loves|X],X).

    /*

*/
```

III λ-CALCULUS IN THE PROLOG INTERPRETER

There are several deficiencies in this Prolog implementation of grammars using the λ-calculus as a meaning representation language.

First, neither of the suggested implementations of λ-reduction in Prolog are particularly attractive. The first, which uses first-order constants to represent variables, requires the addition of a messy gensym argument place to every predicate to simulate the global counter. This seems both inelegant and a duplication of effort, since the Prolog interpreter has a similar kind of variable-handling mechanism built into it. The second approach takes advantage of Prolog's built-in variable facilities, but requires the use of Prolog's meta-logical facilities to do so. This is because Prolog variables are serving two functions, as Prolog variables and as λ-variables. The two kinds of variables function differently and must be differentiated.

Second, there is a problem with invertibility. Many Prolog programs are invertible and may be run 'backwards'. We should be able, for example, to evaluate the sentence grammar predicate giving the meaning of a sentence and have the system produce the sentence itself. This ability to go from a meaning formula back to an English phrase that would produce it is one of the attractive properties of logic grammars. The grammar presented here can also be run this way. However, a careful look at this computation process reveals that with this implementation the Prolog interpreter performs essentially an exhaustive search. It generates every subphrase, λ-reduces it and checks to see if it has the desired meaning. Aside from being theoretically unsatisfactory, for a grammar much larger than a trivially small one, this approach would not be computationally feasible.

So the question arises as to whether the Prolog interpreter might be enhanced to know about λ-formulas and manipulate them directly. Then the Prolog interpreter itself would handle the λ-reduction and would be responsible for avoiding variable collisions. The logic grammars would look even simpler because the lreduce predicate would not need to be explicitly included in each grammar rule. For example, the ts clause in the grammar in the figure above would become:

```
ts(lapply(M1,M2),X,Y) :-
     te(M1,X,Z),
     iv(M2,Z,Y).
```

Declarations to the Prolog interpreter could be included to indicate the predicate argument places that contain λ-terms. Consider what would be involved in this modification to the Prolog system. It might seem that all that is required is just the addition of a λ-reduction operator applied to λ-arguments. And indeed when executing in the forward direction, this is essentially all that is involved.

Consider what happens, however, if we wish to execute the grammar in the reverse direction, i.e., give a λ-term that is a meaning, and have the Prolog system find the English phrase that has that meaning. Now we find the need for a 'λ-expansion' ability.

Consider the situation in which we present Prolog with the following goal:

```
ts(forall(X,implies(man(X),walk(X))),S,[]).
```

Prolog would first try to match it with the head of the ts clause given above. This would require matching the first terms, i.e.,

```
forall(X,implies(lapply(man,X),lapply(walk,X)))
   and
lapply(M1,M2)
```

(using our encoding of λ-terms as Prolog terms.) The matcher would have available the types of the variables and terms. We would like it to be able to discover that by substituting the right terms for the variables, in particular substituting

```
lambda(P,forall(X,implies(
   lapply(man,X),lapply(P,X))))      for M1
   and
walk for M2
```

in the second term, it becomes the same as the first term (after reduction). These M1 and M2 values would then be passed on to the te and iv predicates. The iv predicate, for example, can easily find in the facts the word to express the meaning of the term, walk; it is the work 'walks' and is expressed by the fact iv(walk,[walks|X],X), shown above. For the predicate te, given the value of M1, the system would have to match it against the head of the te clause and then do further computation to eventually construct the sentence.

What we require is a general algorithm for matching λ-terms. Just as Prolog uses unification of first-order terms for its parameter mechanism, to enhance Prolog to include λ-terms, we need general unification of λ-terms. The problem is that λ-unification is much more complicated than first-order unification. For a unifiable pair of first-order terms, there exists a unique (up to change of bound variable) most general unifier (mgu) for them. In the case of λ-terms, this is not true; there may be many unifiers, which are not generalizations of one another. Furthermore unification of λ-terms is, in general, undecidable.

These facts in themselves, while perhaps discouraging, need not force us to abandon hope. The fact that there is no unique mgu just contributes another place for nondeterminism to the Prolog interpreter. And all interpreters which have the power of a universal Turing machine have undecidable properties. Perhaps another source of undecidability can be accommodated. Huet [1975] has given a semi-decision procedure for unification in the typed λ-calculus. The question of whether this approach is feasible really comes down to the finer properties of the unification procedure. It seems not unreasonable to hope that in the relatively simple cases we seem to have in our grammars, this procedure can be made to perform adequately. Notice that, for parsing in the forward direction, the system will always be unifying a λ-term with a variable, in which case the unification problem is trivial. We are in the process of programming Huet's algorithm to include it in a simple Prolog-like interpreter. We intend to experiment with it to see how it performs on the λ-terms used to represent meanings of natural language expressions.

Warren [1982] points out how some suggestions for incorporating λ-calculus into Prolog are motivated by needs that can easily and naturally be met in Prolog itself, unextended. Following his suggestions for how to represent λ-expressions in in Prolog directly, we would represent the meaning of a sentence by a set of asserted Prolog clauses and an encoding atomic name, which would have to be generated. While this might be an interesting alternate approach to meaning representations, it is quite different from the ones discussed here.

IV CONCLUSIONS

We have discussed two alternatives for meaning representation languages for use in the context of logic grammars. We pointed out how one advantage of the typed λ-calculus over first-order logic is its ability to represent directly meanings of phrases of all syntactic categories. We then showed how we could implement in Prolog a logic grammar using the λ-calculus as the meaning representation language. Finally we discussed the possibility and some of the implications of trying to include part of the λ-calculus in the logic programming system itself. We suggested how such an integration might allow grammars to be executed backwards, generating English sentences from input logical forms. We intend to explore this further in future work. If the λ-calculus can be smoothly incorporated in the way suggested, then natural language grammar writers will find themselves 'programming' in two languages, the first-order language (e.g. Prolog) for syntax, and the typed λ-calculus (e.g. typed LISP) for semantics.

As a final note regarding meaning representation languages: we are still left with the feeling that the first-order languages are too weak to express the meanings of phrases of all categories, and that the λ-calculus is too expressive to be

computationally tractable. There is a third class of languages that holds promise of solving both these difficulties, the function-level languages that have recently been developed in the area of programming languages [Backus 1978] [Shultis 1982]. These languages represent functions of various types and thus can be used to represent the meanings of subsentential phrases in a way similar to the λ-calculus. Deduction in these languages is currently an active area of research and much is beginning to be known about their algebraic properties. Term rewriting systems seem to be a powerful tool for reasoning in these languages. I would not be surprised if these function-level languages were to strongly influence the formal meaning representation languages of the future.

V REFERENCES

Backus, J. [1978] Can Programming Be liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, Communications of the ACM, Vol 21, No 8, (Aug 1978), 613-641.

Clark, K.L and S.-A. Tärnlund (eds.) [1982] Logic Programming, Academic Press, New York, 366 pp.

Clifford, J. [1981] ILs: A formulation of Montague's intensional logic that includes variables and constants over indices. TR#81-029, Department of Computer Science, SUNY, Stony Brook, New York.

Colmerauer, A. [1978] Metamorphosis Grammars, in Natural Language Communication with Computers, Vol 1, Springer Verlag, 1978, 133-189.

Colmerauer, A. [1982] An Interesting Subset of Natural Language, in Logic Programming, Clark, K.L and S.-A Tärnlund (eds.), 45-66.

Dahl, Veronica [1981] Translating Spanish into Logic through Logic, American Journal of Computational Linguistics, Vol 7, No 3, (Jul-Sep 1981), 149-164.

Gallin, D. [1975] Intensional and Higher-order Modal Logic, North-Holland Publishing Company, Amsterdam.

Gawron, J.M., et.al. [1982] The GPSG Linguistic System, Proceedings 20th Annual Meeting of the Association for Computational Linguistics, 74-81.

Huet, G.P. [1975] A Unification Algorithm for Typed λ-Calculus, Theoretical Computer Science, Vol 1, No 1, 22-57.

Jones, M.A., and Warren, D.S. [1982] Conceptual Dependency and Montague Grammar: A step toward conciliation, Proceedings of the National Conference on Artificial Intelligence, AAAI-82, 79-83.

McCord, M. [1982] Using Slots and Modifiers in Logic Grammars for Natural Language, Artifical Intelligence, Vol 18, 327-367.

Montague, Richard [1973] The proper treatment of quantification in ordinary English, (PTQ), reprinted in Montague [1974], 246-270.

Montague, Richard [1974] Formal Philosophy: Selected Paper of Richard Montague, edited and with an introduction by R. Thomason, Yale University Press, New Haven.

Pereira, F.C.N. and Warren, D.H.D. [1980] Definite Clause Grammars for Language Analysis - A survey of the formalism and a Comparison with Augmented Transition Networks. Artificial Intelligence 13,3 (May 1980) 231-278.

Rosenschein, S.J. and Shieber, S.M. [1982] Translating English into Logical Form, Proceedings of the 20th Annual Meeting of the Association for Computational Linguistics, June 1982, Toronto, 1-8.

Schubert L.K. and Pelletier F.J. [1982] From English to Logic: Context-free Computation of 'Conventional' Logical Translation, American Journal of Computational Linguistics, Vol 8, No 1, (Jan-Mar 1982), 27-44.

Shultis, J. [1982] Hierarchical Semantics, Reasoning, and Translation, Ph.D. Thesis, Department of Computer Science, SUNY, Stony Brook, New York.

Simmons, R.F. and Chester, D. [1982] Relating Sentences and Semantic Networks with Procedural Logic, Communications of the ACM, Vol 25, Num 8, (August, 1982), 527-546.

Warren, D.H.D. [1981] Efficient processing of interactive relational database queries expressed in logic, Proceedings of the 7th Conference on Very Large Data Bases, Cannes, 272-281.

Warren, D.H.D. [1982] Higher-order extensions to PROLOG: are they needed? Machine Intelligence 10, Hayes, Michie, Pao, eds. Ellis Horwood Ltd., Chichester.

Warren, D.S. and Friedman, J. [1981] Using Semantics in Noncontext-free Parsing of Montague Grammar, TR#81-027, Department of Computer Science, SUNY, Stony Brook, New York, (to appear).

Woods, W.A. [1970] Transition Network Grammars for Natural Language Analysis, Communications of the ACM, Vol 1, No 10, (Oct 1970).

Woods, W.A., Kaplan, R.M., and Nash-Webber, B. [1972] The Lunar Science Natural Language Information System: Final Report, BBN Report No. 2378, Bolt Baranek and Newman, Cambridge, MA.