MASSACHUSETTS INSTITUTE OF TECHNOLOGY
A. I. LABORATORY

Artificial Intelligence
Memo No. 278

February 1973

D-SCRIPT: A COMPUTATIONAL THEORY OF DESCRIPTIONS

Robert C. Moore

Work reported herein was conducted at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported in part by the Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under Contract Number N00014-70-A-0362-0003.

Abstract

This paper describes D-SCRIPT a language for representing knowledge in artificial intelligence programs. D-SCRIPT contains a powerful formalism for descriptions, which permits the representation of statements that are problematical for other systems. Particular attention is paid to problems of opaque contexts, time contexts, and knowledge about knowledge. The design of a theorem prover for this language is also considered.

Descriptive phrases - representation of knowledge, natural language understanding, theorem proving, opaque contexts, time contexts, knowledge about knowledge.

D-SCRIPT: A Computational Theory of Descriptions

1. Introduction

1.1 Ways of Representing Knowledge

Methods advocated for representing knowledge in artificial intelligence programs have included logical statements (McCarthy, Sandewall), semantic networks (Quillian, Schank), and procedures (Hewitt, Sussman and McDermott). All these approaches share one fundamental concept, the notion of predication. That is, the basic data structure in each system is some representation of a predicate applied to objects. In this respect, the various systems are more or less equivalent. But this basic idea must be extended to handle problems of quantification and knowledge about knowledge. Here the systems do differ. We will argue, though, that these differences result from the descriptive apparatus used in the particular systems being compared, rather than from an inherent advantage of, say, procedures over declaratives or vice versa.

Advocates of PLANNER (e.g. Winograd, p. 215) have argued that the predicate calculus cannot represent <u>how</u> a piece of knowledge should be used. But this is true only of the <u>first-order</u> predicate calculus. In a higher-order or non-ordered declarative language, statements could be made which would tell a theorem prover how other statements are to be used. PLANNER, on the other hand, has no way of directly stating an existential quantification, but this does not mean that procedural languages are necessarily incapable of handling that problem.

Our belief, then, is that the type of system used to represent knowledge is unimportant, so long as it has sufficient expressive power. This paper presents an attempt at such a system, the language D-SCRIPT. As the name implies, the most interesting feature of D-SCRIPT is its powerful formalism for descriptions, which enables it to represent statements that are problematical in other systems. No position will be taken as to what kind of language D-SCRIPT is. Since it is intended to answer questions by making deductions from a data base, it can be thought of as a theorem prover. Since it operates by comparing expressions like the data-base languages of PLANNER and CONNIVER, it can be thought of as a pattern-matching language. And since it is Turing universal and, in fact, includes the lambda calculus, it can be thought of as a programming language.

1.2 Problems in Representing Knowledge

Before presenting the details of D-SCRIPT, some idea of the type of problem it is designed to solve should be given. A classic problem is that of representing opaque contexts. An opaque context is one which does not allow substitution of referentially equivalent expressions or does not allow existential quantification. For example the verb "want" creates an opaque context:

(1.1) John wants to marry the prettiest girl.

This sentence is ambiguous. It can mean either:

(1.2) John wants to marry a specific girl who also happens to be the prettiest.

or:

(1.3) John wants to marry whoever is the prettiest girl, although he may not know who that is.

Under the first interpretation we can substitute any phrase which refers to the same person for "the prettiest girl". That is, if the prettiest girl is named "Sally Sunshine", from (1.2) we can infer:

(1.4) John wants to marry a specific girl who also happens to be named Sally Sunshine.

We cannot make the corresponding inference from (1.3). It will not be true that:

(1.5) John wants to marry whoever is named Sally Sunshine, although he may not know who that is.

Because of this property, (1.2) is called the transparent reading of (1.1) and (1.3) is called the opaque reading. It is almost always the case that sentences having an opaque reading are ambiguous with the other reading being transparent.

To illustrate blocking of existential quantification, consider:

(1.6) John wants to marry a blonde.

Again the sentence is ambiguous, meaning either:

(1.7) John wants to marry a specific girl, who also happens to be a blonde.

or:

(1.8) John has no particular girl in mind, but he wants whoever he does marry to be a blonde.

We can existentially quantify over the first reading but not the second. We can infer:

(1.9) There exists someone whom John wants to marry.

from (1.7), but not from (1.8).

Another problem is the occurrence of descriptive phrases in sentences involving time reference. In the sentence:

(1.10) The President has been married since 1945.

the phrase "the President" refers to an individual. In the sentence:

(1.11) The President has lived in the White House since 1800.

"the President" refers to each President in turn.

Another type of sentence where the reference of a phrase depends on time is illustrated by:

(1.12) John met the President in 1960.

This sentence is ambiguous, but unlike (1.11), each interpretation refers to only one person. The ambiguity is whether "the President" refers to the President at the time (1.12) is asserted, or the President in 1960.

Representing knowledge about knowledge raises some interesting issues. For instance, in:

(1.13) John knows Bill's phone number.

how is John's knowledge to be represented? In John's mind it might be something like:

(1.14) (PHONE-NUM BILL 987-6543)

So, (1.13) might be:

(1.15) (KNOWS JOHN (PHONE-NUM BILL 987-6543))

The trouble with (1.15) is that it includes too much information. Not only does it say what (1.13) says, it also says what the number <u>is</u>. The difficulty is to refer to a piece of information without stating it.

For all these types of sentences, D-SCRIPT provides representations which allow the correct deductions to be made. Further, it provides separate representations for each meaning of the ambiguous sentences, and these representations are related in a way that explains the ambiguity.

2. The D-SCRIPT Language

2.1 D-SCRIPT Expressions

D-SCRIPT contains the following types of expressions:

1. constants
2. variables
3. forms
4. lists

A constant is any alpha-numeric (i.e. <u>only</u> letters or numbers) character string (e.g. "FOO", "BLOCK5"). A variable is any alpha-numeric character string prefixed by "?" (e.g. "?X"). A form is any sequence of expressions enclosed in angle-brackets (e.g. "<X Y ?Z>"). A list is any sequence of expressions enclosed in parentheses (e.g. "(FOO A <BAR B C>)").

D-SCRIPT observes the convention that all functions, predicates, and operators evaluate their arguments. The rules for evaluating expressions are largely adapted from LISP. In fact, D-SCRIPT variables and forms are treated just like LISP atoms and lists, respectively. Rather than introducing "QUOTE", however, we use constants and lists to represent pre-defined items. To state our rules formally:

1. A constant evaluates to itself.

2. A variable evaluates to the expression which it has been assigned.

3. The value of a form is the result of applying its first element to the values of its remaining elements. This will not be defined in general, but only for those expressions which represent meaningful operations in D-SCRIPT. One such case is that of lambda-expressions. A lambda-expression is represented in D-SCRIPT by a form containing the constant "LAMBDA", followed by a list of variables, followed by an expression (e.g. "<LAMBDA (?X ?Y) <TIMES ?X ?Y>>"). A form whose first element is a lambda-expression is evaluated in the same way as a corresponding LISP expression. The result is the value of the body of the lambda-expression, with the values of the arguments assigned to the corresponding variables. For instance, assuming "+" has the usual meaning, "<<LAMBDA (?X) <+ 2 ?X>> 3>" has the same value as "<+ 2 3>", which is "5". We will introduce other types of forms whose value is defined when we explain the representation of statements.

4. A list evaluates to a form with identical structure, except that free variables are replaced by their values. If "?X" has previously been assigned the value "A", then "(LAMBDA (?Y) (FOO ?X ?Y)) will evaluate to "<LAMBDA (?Y) (FOO A ?Y)>".

It is worth noting that the way lambda-expressions and lists are defined makes it very easy to write functions which construct complex forms. For example, consider "<LAMBDA (?X) (FOO (BAR (GRITCH ?X)))>". The result of applying this to "Z" is "<FOO (BAR (GRITCH Z))>". A comparable LISP function would have to be built up with "CONS"'s to achieve this result.

2.2 Representing Knowledge in D-SCRIPT

The most basic statements are those which express simple predication. A statement of this kind is represented in D-SCRIPT by a form whose first element is a constant representing the predicate and whose other elements are constants representing the objects of the predicate. For example:

```
(2.1) The sun is a star.
(2.2) BlockA is on BlockB.
```

could be represented as:

```
(2.3) <STAR SUN>
(2.4) <ON BLOCKA BLOCKB>
```

A simple statement about a statement, such as:

```
(2.5) John believes the sun is a star.
```

would be:

```
(2.6) <BELIEVE JOHN (STAR SUN)>
```

The important thing to notice about (2.6) is that the embedded statement is represented by a list. This is because we need an expression whose <u>value</u> is (2.3) to be consistent with the convention that predicates (in this case, "believe") evaluate their arguments.

To represent more complex statements, two types of extensions are needed. The simpler of these is the addition of logical connectives. D-SCRIPT uses "OR", "AND", "NOT", and "IMPLIES" to stand for the obvious logical operations. As in (2.6) the embedded statements are expressed as lists. So:

(2.7) If the sun is a star, then BlockA is on
      BlockB.

would be represented by:

```
(2.8) <IMPLIES (STAR SUN) (ON BLOCKA BLOCKB)>
```

This notation reflects the fact that in D-SCRIPT, logical connectives operate on the statements themselves rather than on their truth-values. "IMPLIES", then, is not computed as a Boolean function, but rather is computed by asserting that its first argument is true, and attempting to prove its second argument.

The other extension required for complex statements, and the one that is most important to our theory, is the use of descriptions. There are three types of descriptions in D-SCRIPT; existential descriptions, universal descriptions and definite descriptions. A description is a form whose first element is "SOME" (existential), "EVERY" (universal), or "THE" (definite); whose second element is a list containing a variable; and whose third element is an expression whose value is a statement. Descriptions represent the corresponding types of natural language descriptive phrases:

```
     (2.9) a block            <SOME (?X) (BLOCK ?X)>
           every number       <EVERY (?Y) (NUM ?Y)>
           the table          <THE (?X) (TABLE ?X)>
```

Some examples of sentences containing descriptive phrases and their representations are:

```
     (2.10) The father of the bride is happy.
            <HAPPY <THE (?X) (FATHER ?X <THE (?Y) (BRIDE ?Y)>)>>

     (2.11) John owns a dog.
            <OWN JOHN <SOME (?X) (DOG ?X)>>

     (2.12) Every boy likes Santa Claus.
            <LIKE <EVERY (?X) (BOY ?X)> SANTA>
```

Notice that when descriptions appear in statements, they are left as forms. This is because, unlike embedded statements, we are talking about the objects to which the descriptions refer (i.e. their values) rather than the descriptions themselves.

The notation we have used so far is not sufficient to express statements containing more than one occurrence of the same description. In the sentence:

(2.13) Every boy either loves Santa Claus or hates him.

the phrase "every boy" is the subject of both "loves" and "hates". We cannot use the following representation though:

```
     (2.14) <OR (LOVE <EVERY (?X) (BOY ?X)> SANTA)
                (HATE <EVERY (?X) (BOY ?X)> SANTA)>
```

because this means:

(2.15) Either every boy loves Santa Claus or every
boy hates Santa Claus.

which, of course, is quite different. We can overcome this difficulty by using lambda-expressions. We will represent (2.13) by:

```
(2.16) <<LAMBDA (?X) (OR (LOVE ?X SANTA) (HATE ?X SANTA))>
        <EVERY (?Y) (BOY ?Y)>>
```

This can be read as something like "the predicate X is true of every boy," where the predicate X is "loves Santa Claus or hates him."

We have a similar situation with respect to the scope of quantifiers. It is not clear whether:

```
(2.17) <GREATER <SOME (?X) (NUM ?X)> <EVERY (?Y) (NUM ?Y)>>
```

represents:

(2.18) For every number there is some larger
number.

or:

(2.18) There is some number which is larger than
every number.

We will have to arbitrarily choose a rule to disambiguate (2.17), but by using lambda-expressions we can avoid the difficulty.

(2.18) can be represented by:

```
     (2.20) <<LAMBDA (?X) (GREATER <SOME (?Y) (NUM ?Y)> ?X)>
             <EVERY (?Z) (NUM ?Z)>>
```

and (2.19) can be represented by:

```
     (2.21) <<LAMBDA (?X) (GREATER ?X <EVERY (?Y) (NUM ?Y)>)>
             <SOME (?Z) (NUM ?Z)>>
```

Analyzing these expressions in the same way as (2.16) will show that they have the correct meaning.

It should be apparent that existential and universal descriptions in D-SCRIPT serve exactly the same function as the quantifiers of the predicate calculus. In view of this, it may be asked why we have used a different notation. One reason is that our notation makes it possible to write expressions whose structure more closely resembles the sentences they represent. Hopefully this makes them more intelligible. The more important reason, though, is that having a single expression for a description makes it easier for an interpreter to manipulate them.

2.3 Formal Semantics of D-SCRIPT

The previous two sections outlined the syntax and informal semantics of D-SCRIPT. This section attempts to show how a program could be written that would interpret D-SCRIPT statements in accord with their intuitive meaning. The details of this will be somewhat sketchy. One reason for this is that choosing proof strategies and using heuristic information are complicated problems that we cannot claim to have solved. Secondly, creating a theorem prover is not our main goal. What we are trying to do is to show the sort of descriptive system necessary to represent the information contained in natural language statements. The purpose of this section is to establish that our notation for that system is "well-founded".

The program we have in mind would take a statement as its input and determine from its data base whether the statement is true. For statements which are simple predications, the program looks for another statement in the data base which matches the first statement. The statement whose truth is being determined will be called the "test statement"; the statement in the data base to which it is being compared will be called the "target statement". To prove a complex statement, the program would break it down into its components and process them according to the semantics of the operators involved. Similarly, a complex target statement must be broken down to its components for processing, but the rules are different. So, in explaining the semantics of complex expressions, analyses will be given for their use both in test statements and in target statements.

Two basic statements match if their corresponding elements match. In general, expressions which are not statements match whenever their values are identical. A variable which has not been assigned a value matches any expression, and is assigned that expression's value. These rules apply to both test statements and target statements. As an example, suppose "5" has been assigned to "?X", "?Y" is unassigned, and "+" has its usual meaning. Then "<FOO 5 ?Y>" will match "<FOO ?X <+ 3 4>>" and "7" will be assigned to "?Y".

We will not give a complete proof procedure for logical connectives. It is a well understood problem and is not of primary importance in the phenomena we wish to explain. But to suggest the kind of procedure we have in mind, consider "AND" and "IMPLIES". In handling these expressions the distinction between test statements and target statements comes through. To prove "<AND x y>" both x and y must be proved; but in matching something against "<AND x y>", the match succeeds if either x or y matches. "<IMPLIES x y>" is true if in a hypothetical state where x is asserted, y can be proved. A test statement will match a target stat ment "<IMPLIES x y>" if the test statement matches y and x can be proved. "OR" and "NOT" are somewhat more complicated but can be handled in much the same way.

The really important part of our proof procedure is the treatment of descriptions. Definite descriptions are the simplest. "<THE (?X) (...?X...)>" evaluates to the constant which when assigned to "?X" makes "<...?X...>" true. If there is not such a constant or if there is more than one, the value of the description is undefined. For example, if "LESS" means "arithmetically less than", then "<FOO 3>" matches:

(2.22) <FOO <THE (?X) (AND (LESS ?X 4) (LESS 2 ?X))>>

This rule for evaluating definite descriptions applies to both test statements and target statements.

For existential and universal descriptions, there is again a difference between test statements and target statements. In a test statement, an existential description matches anything that makes the body of the description true. That is, "<FOO <SOME (?X) (BAR ?X)>>" matches "<FOO A>" if "<BAR ?X>" is true when "?X" is assigned "A". For the case of a target statement, the evaluation is more difficult. If we know that "Some bar is foo," we could simply give it a name and continue. But giving a name would imply that we know which bar is foo, which is not true. Instead we can create a name and say that if the new name were the name of the object that is asserted to exist, then anything which we can prove about the new name is true of the object. We do this by creating a hypothetical state of the data base in which, if the new name is "G999", we assert "<BAR G999>". The target statement then becomes "<FOO G999>". Another way of putting this is that "<SOME (?X) (BAR ?X)>" evaluates to "G999", with the side effect of creating a hypothetical state of the data base in which "<BAR G999>" is asserted. When the hypothesis is discharged, the new name becomes undefined, and we are not in danger of supposing that we know what the name of the object is.

The treatment of universal descriptions is the exact dual of that for existential descriptions. In a test statement, we know that whatever we can prove about an arbitrarily selected member of a class is true of every member of the class. So just as we did for existential target statements, we set up a hypothetical state, produce an arbitrary unique name, and assert that it is a member of the class. Analogously to what we said before, "<EVERY (?X) (FOO ?X)>" evaluates to, say, "G111" with the side effect of creating a hypothetical state in which "<FOO G111>" is asserted. Also in duality with existential descriptions, in a target statement a universal description matches anything which makes its body true. For example, "<FOO A>" matches "<FOO <EVERY (?X) (BAR ?X)>>" if "<BAR ?X>" is true when "?X" is assigned "A".

Now we can see why lambda-expressions are important for representing information in D-SCRIPT. Evaluating existential and universal descriptions sometimes has the side effect of changing the data base. Later we will introduce other expressions which also do this. If we have other descriptions in the statement, we need to be able to control whether they are evaluated in the old data base or the new. By "lambda-fying" a statement we can bring one or another description to the outside and force it to be evaluated first. In this way we can control the order in which expressions are evaluated. A detailed example of this will be given in section 3.3.

In this brief summary, we have given the barest outlines of a proof procedure. We have not discussed any of the complex interactions among these logical operators. But hopefully we have laid a sufficient foundation to talk about the issues that are the real point of this paper.

3. Solution to Representation Problems Using D-SCRIPT

3.1 Descriptions in Opaque Contexts

In general, descriptive phrases in opaque contexts are subject to more than one interpretation. Furthermore, at least one of the interpretations seems not to behave according to normal rules of logical manipulation. Looking more closely, opaque contexts primarily occur in the complement constructions of verbs like "want", "believe", "know", etc. These verbs all have the property of describing somebody's model of the world. When we say:

(3.1) John wants to marry Sally.

what we mean is that in John's model of the world, the state:

(3.2) John is married to Sally.

is considered desirable. The ambiguity of descriptive phrases arises from the question of whether the descriptive phrase is to be evaluated in our model of the world or the model of the subject of the sentence. To illustrate this, recall the sentence:

(3.3) John wants to marry the prettiest girl.

In D-SCRIPT, the opaque reading is represented by:

```
(3.4) <WANTS JOHN (MARRY JOHN <THE (?X) (PRETTIEST ?X)>)>
```

The reason that there are restrictions on substituting other expressions for "<THE (?X) (PRETTIEST ?X)>" is that the statement which actually contains this description, i.e.:

```
(3.5) <MARRY JOHN <THE (?X) (PRETTIEST ?X)>>
```

is part of John's world model. If in our program we represent John's world model by a separate data base, then the expressions which may be substituted are those which are equivalent in that data base, not in the main data base which represents our world model.

To represent the transparent reading of (3.3), we must take the description outside the scope of John's model. We can do this with a lambda-expression:

```
(3.6) <<LAMBDA (?X) (WANTS JOHN (MARRY JOHN ?X))>
        <THE (?Y) (PRETTIEST ?Y)>>
```

This says that the statement we get by evaluating the description in our model and substituting that value for "?X" in:

(3.7) <MARRY JOHN ?X>

is marked as a desirable state in John's world model.

The analysis is analogous for existential descriptions. The two readings of:

(3.8) John wants to marry a blonde.

can be represented by:

```
(3.9) <WANT JOHN (MARRY JOHN <SOME (?X) (BLONDE ?X)>)>
```

for the opaque reading, and by:

```
(3.10) <<LAMBDA (?X) (WANT JOHN (MARRY JOHN ?X))>
         <SOME (?Y) (BLONDE ?Y)>>
```

for the transparent reading. (3.9) means:

(3.11) John wants there to be a blonde that he marries.

and (3.10) means:

(3.12) There is a blonde that John wants to marry.

So the reason we can't make a "there is..." paraphrase of (3.9) is that rather than being an existential statement, it is an assertion <u>about</u> an existential statement.

3.2 Descriptions in time contexts

In order to discuss the next set of examples, we need a way to represent time. The basic fact here is that any predicate can be made to vary with time. Even those that we choose to consider eternal can be alleged to depend on time, e.g.:

(3.13) Two used to be greater than three.

To account for this in first-order logic, we would have to make time an explicit parameter of every predicate symbol. Instead, we will represent time by a <u>context-structured</u> data base (McDermott). By this we mean that the data base will be broken down into a series of sub-data bases, or contexts, each of which represents the state of the world at some particular time. This can be efficiently implemented, as it is in CONNIVER (Sussman and McDermott) by specifying each context by recording the differences between it and its predecessor.

To use this kind of data base, we need a special predicate "T-A-T" which takes as its parameters a statement and the name of a time context. "<T-A-T <u>s</u> <u>t</u>>" means statement <u>s</u> is True At Time <u>t</u>. The formal semantics of "T-A-T" are that it attempts to prove <u>s</u> in the time context named by <u>t</u>. We also need to be able to generate references to time contexts. For instance, the phrase:

(3.14) when Washington was President

would be represented by the description:

(3.15) <THE (?T) (T-A-T (PRES WASHINGTON) ?T)>

Finally we need the one-place predicate "TIME" to make quantified statements about time. We would represent:

(3.16) Three is always greater than two.

by:

(3.17) <T-A-T (GREATER 3 2) <EVERY (?T) (TIME ?T)>>

Given this notation for time, we can solve the associated problems which we raised earlier. As in the case of opaque contexts, the solution depends on whether a description is evaluated in the context in which a statement is made or the context which the statement is about. Recalling the previous examples:

(3.18) The President has been married since 1945.

is represented by:

```
(3.19) <<LAMBDA (?X) (T-A-T (MARRIED ?X)
                            <EVERY (?T) (AFTER ?T 1945)>)>
          <THE (?Y) (PRES ?Y)>>
```

In (3.19) the use of the lambda-expression puts the description "<THE (?Y) (PRES ?Y)>" outside the time construction, so it is evaluated in the context in which the statement is made. On the other hand:

(3.20) The President has lived in the White House
       since 1800.

is represented by:

```
(3.21) <T-A-T (LIVE-IN <THE (?X) (PRES ?X)> W-H)
              <EVERY (?T) (AFTER ?T 1800)>>
```

Here the description is inside the time construction and is not evaluated until the time description has been instantiated. The analysis is the same for:

(3.22) John met the President in 1960.

except that in this case the time reference is definite. One interpretation is given by:

```
(3.23) <T-A-T (MEET JOHN <THE (?X) (PRES ?X)> 1960>
```

and the other is given by:

```
(3.24) <<LAMBDA (?X) (T-A-T (MEET JOHN ?X) 1960)>
        <THE (?Y) (PRES ?Y)>>
```

3.3 Knowledge about Knowledge

One of the questions we raised in the beginning was how to represent:

(3.25) John knows Bill's phone number.

If we knew the number we could represent (3.25) by:

```
(3.26) <KNOW JOHN (PHONE-NUM BILL xxx)>
```

where xxx is the number. We do know one description of the number, namely "Bill's phone number". If we substitute this into (3.26), however, we get a trivial statement:

```
(3.27) <KNOW JOHN (PHONE-NUM BILL
                     <THE (?X) (PHONE-NUM BILL ?X)>)>
```

which means:

```
(3.28) John knows that Bill's phone number is
       Bill's phone number.
```

What we need to do is to remove the occurrence of the description from John's world model into our world model. Once again, we can do this with a lambda-expression:

```
(3.29) <<LAMBDA (?X) (KNOW JOHN (PHONE-NUM BILL ?X))>
        <THE (?X) (PHONE-NUM BILL ?X)>>
```

This says that if we were to evaluate the description "Bill's phone number" and stick the result in (3.26), we would correctly describe John's knowledge.

To see the difference between (3.27) and (3.29), suppose we know that Bill has a phone number, and we know that John knows that Bill has a phone number. These facts are represented by:

```
(3.30) <PHONE-NUM BILL <SOME (?X) (NUM ?X)>
```

(3.31) <KNOW JOHN (PHONE-NUM BILL <SOME (?X) (NUM ?X)>)>

Given this, we can prove (3.29) from itself. Notice that in D-SCRIPT this is non-trivial. Complex statements are never proved by simply looking to see if they are in the data base. Rather, they are broken down to their basic components and these components are processed according to the semantics of the operators combining them. In the case of "KNOW" the semantics are to shift the proof to the data base of the person doing the knowing. So even to prove a statement from itself, the semantics really have to work.

In trying to prove (3.29) the lambda-expression makes us first evaluate "<THE (?X) (PHONE-NUM BILL ?X)>". We do this by trying to find a match for "<PHONE-NUM BILL ?X>". If we don't know Bill's phone number we can't do this directly. (3.30), however, entitles us to create a hypothetical state in which some arbitrary constant, say "G777" is asserted to be Bill's number. So to prove (3.29), we attempt to prove:

(3.32) <KNOW JOHN (PHONE-NUM BILL G777)>

with the hypothesis:

(3.33) <PHONE-NUM BILL G777>

To prove (3.32) from (3.29) we process (3.29) much the same as before. This time, however, we already have (3.33) in the data base; so "<THE (?X) (PHONE-NUM BILL ?X)>" evaluates to "G777" directly. Our proof then reduces to proving (3.32) from itself, which reduces again to proving (3.33) from itself in the data base which represents John's world model. (3.33) is a basic statement, so it can be inferred from itself immediately, and the entire proof succeeds.

Now suppose instead that we were trying to prove (3.29) from (3.27). The proof would be the same down to the point where we generated the subgoal of proving (3.32). To prove this from (3.27), we have to prove (3.33) from:

(3.34) <PHONE-NUM BILL <THE (?X) (PHONE-NUM BILL ?X)>>

in the context of John's world model. But this time we cannot use (3.33) when we evaluate the description, because (3.33) is asserted only in our world model, and the evaluation is taking place in John's. What will happen is that (3.31) will be used to generate another arbitrary constant (e.g. "G888") in John's world model. We will then try to prove (3.33) from:

(3.35) <PHONE-NUM BILL G888>

Since "G777" does not match "G888", the proof fails.

4. Future Work

In this paper we have presented a formal language for the representation of knowledge. We have shown how information which is difficult to express in other formalisms can be expressed in our language. And we have suggested how a theorem prover could be designed to make deductions in our language. Clearly, the next step in this research is to build that theorem prover.

Even as a first-order theorem prover, it would seem to have advantages over existing theorem provers. For one thing, since it would handle quantification at any level, it would be able to deal with statements in the form in which they are most naturally expressed. Also, the type of procedure we have discussed is completely semantic in the way it handles logical operators, which according to at least one current A.I. doctrine is A Good Thing.

It will probably be more difficult to work out how predicates like "T-A-T" and "KNOW", which refer to other data bases, interact with the traditional logical operations. But the pay-off here is greater. If we can program a theorem prover to treat "KNOW" in the way we have proposed, we will have taken a first step towards creating programs which can think about thinking.

Bibliography

Hewitt, C., "Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot," Report AI TR-258, M.I.T. A.I. Laboratory, 1972.

McCarthy, J., "Programs with Common Sense," in _Semantic Information Processing_, Marvin Minsky, ed., pp. 403-418. Cambridge, Mass.: M.I.T. Press, 1968.

McDermott, D. V., "Assimilation of New Information by a Natural Language-Understanding System," unpublished S.M. thesis, M.I.T., 1973.

Quillian, M. R., "Semantic Memory," in _Semantic Information Processing_, pp. 227-270.

Sandewall, E., "Formal Methods in the Design of Question-Answering Systems," _Artificial Intelligence_, Vol. 2 (1971), pp. 129-145.

Schank, R. C., "A Conceptual Dependency Representation for a Computer-Oriented Semantics," Memo AI-83, Stanford A.I. Project, 1969.

Sussman, G. J. and D. V. McDermott, "From PLANNER to CONNIVER - A genetic approach," _Proc. FJCC_ 41 (1972), pp. 1171-1179.

Winograd, T., "Procedures as a Representation for Data in a Program for Understanding Natural Language," Report AI TR-17, M.I.T. A.I. Laboratory, 1971.