TRANSLATING HORN CLAUSES FROM ENGLISH

Yeong-Ho Yu

Artificial Intelligence Laboratory
AI-TR-84-3

Computer Science Department
TR-84-29

University of Texas at Austin
Austin, Texas 78712

August 1984

TRANSLATING HORN CLAUSES FROM ENGLISH

BY

YEONG-HO YU, B.A.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF ARTS IN COMPUTER SCIENCES

THE UNIVERSITY OF TEXAS AT AUSTIN

August, 1984

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Dr. Robert F. Simmons for his persistent help and guidance throughout this work. I would also like to thank Dr. Gordon Novak for his helpful comments on my final draft.

I am grateful to my friends who encouraged me in many ways. Special thanks are due to Kyung-Sook Shin. She helped me correct my English.

I am also greatly indebted to my parents for their love and support during the years of my study.

Yeong-Ho Yu

The University of Texas at Austin
August, 1984

ABSTRACT

This work introduces a small grammar which translates a subset of English into Horn Clauses. The parallel between the syntactic structures of sentences and corresponding literals of their Procedural Logic representation facilitates the translation. An interpreter is also described which accepts English descriptions of problem solving methods and facts about an example domain, the Blocks World. The interpreter translates them into Horn Clauses by using the grammar, and interprets them as programs and data. It also carries out commands on data by using the programs, and answers queries about data. In addition, it provides a mechanism called "Consistency Rules" which maintains integrity of the database. This experiments shows that Procedural Logic provides a unified system to accomplish a wide variety of computations, and that a small grammar without any semantic or domain-specific knowledge can translate a small subset of English sentences into Horn Clauses. Further researches on several other domains are suggested to evaluate the usefulness of this translation.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

One of the merits of **Procedural Logic** as a programming language is that it is similar to one convenient form of human reasoning. Based on the theory of Clausal Logic which is a branch of logic, it can represent logical reasoning in a concise and self-explanatory way. The parallelism between the clauses of procedural logic programs and some English sentences also eases the process of converting English sentences to clauses or vice versa by using the computer. This thesis is a study of the process of converting English to Procedural Logic.

This introductory chapter explains several aspects of Procedural Logic, and discusses its parallelism with English sentences.

1.1 Procedural Logic as a Programming Language

Since the early 1970's when Procedural Logic was conceived as a programming tool [Colmerauer 78], it has been used as a major programming tool in AI research. In his book [Kowalski 79], Kowalski discussed several advantages of Procedural Logic as a programming language over the other formal languages. The advantages can be summarized as follows.

- It is very high level and human oriented. The top-down search of state space representation of a problem using Procedural Logic is one of the ways a person solves it.
- It provides a uniform representation for several different tasks such as data storage, queries, and integrity constraints in databases.
- It has the full power of standard logic and any representation of a logical statement has its counterpart in Procedural Logic.

Even though historically the disadvantage of Procedural Logic was said to lie in its slow speed in execution, current technology provides us with a powerful PROLOG compiler which can generate code with the efficiency of LISP [Warren 77]. There are several programming languages which are based on Procedural Logic such as PROLOG, QLOG, HCPRVR[1], and LOGLISP.

The syntax and semantics of those languages are principally based on Clausal Logic. The basic unit for programming in those languages is a Horn Clause which has the following form.

$$(C \quad D^{**} \quad H_1 \quad H_2 \quad \ldots \quad H_n) \quad \text{where } n=0,1,\ldots$$

** D is a deliminator and any symbol can be used.
In this paper, <, IF, and TEMP are used as D.
D can be omitted if n = 0.

The semantics of this Horn Clause is that the truth value of C(Conclusion) is *true* if all of H_i(Hypotheses) are *true*. Therefore, if n=0, C is always true. Such a Horn Clause is called an *assertion*. If n is greater than 0, it is called a *rule*. C and H_i are called *literals* and each literal may be a variable, or a list of a predicate name and any number (possibly 0) of arguments. Each argument may be a variable, a constant, or a literal. The quantifiers of variables are entirely omitted, and follow the convention that the variables in C (or C and H_i both) are universally quantified, and those which are only in H_i are existentially quantified[2].

Figure 1-1 is an example of a procedural logic program which consists of two clauses. MEMBER is a predicate name and (MEMBER X (X . Z)), (MEMBER X (Y . Z)), and (MEMBER X Z) are examples of literals. The variables, X, Y, and Z are all universally quantified.

---

[1]HCPRVR is the acronym of *Horn Clauses PRoVeR* which has been developed in the University of Texas at Austin by Daniel Chester [Chester 80]. All the procedural logic programs in this work run in HCPRVR, and other LISP functions are written in ELISP.

[2]Further explanations of the Horn Clause can be found in [Simmons 84a] [Kowalski 79]

```
((MEMBER X (X . Z)))
((MEMBER X (Y . Z)) < (MEMBER X Z))
```

Figure 1-1: Procedural Logic Program for MEMBER

The mechanism of using a procedural logic program is to prove the given task (called a theorem) by using the assertions and the rules. The task of proving a theorem in HCPRVR is achieved by tracing the subgoals. When a goal (a theorem to be proved) is given, HCPRVR tries to find an assertion or a rule whose conclusion can be matched with the theorem. If it finds a rule, it tries to prove the hypotheses or the subgoals of the rule. If it can prove all of the subgoals, the theorem is proved. If it finds that the theorem matches an assertion, the theorem is proved. Otherwise, it will seek another assertion or another rule. If it fails to find any rule or assertion whose conclusion is matched with the theorem and whose hypotheses are all proved, the theorem is false. This mechanism works recursively for subgoals, and when there is another option to make another set of subgoals and the current subgoals can not be proved, the backtracking mechanism is used. And, when two literals are compared, the Unification Algorithm is used.

For example, if we want to use the program of Figure 1-1 to check whether A is a member of the list (B A C D), we will ask HCPRVR to prove the theorem, (MEMBER A (B A C D)). At the first round, HCPRVR fails to match the theorem with the first clause(assertion), but succeeds with the second clause(rule). The match will generate a subgoal, (MEMBER A (A C D)), and it will be proved by the first clause at the second round. Because there is a proof for the theorem, (MEMBER A (B A C D)), the theorem is true and the desired task has been successfully achieved.

Also, in HCPRVR, we can call any LISP function. If the function call returns NIL, it fails. Otherwise, it is proved.

1.2 Parallelism between English and Procedural Logic

In Figure 1-1, we easily read the program. An English description of the program, which is natural and easy to understand, is as below.

X is a member of the list (X . Z).
X is a member of the list (Y . Z)
if X is a member of the list Z.

In this example, there is an apparent parallelism between the program and the English sentences which describe it. This shows not only that the program is very high level and self-explanatory, but also that it is easy to convert some English description of a problem-solving method to a procedural logic program, and vice versa. Figure 1-2 shows other examples of the parallelism between some simple English sentences and the corresponding literals in one of possible Horn Clause representations of them.

```
The block is above the table.   (ABOVE* (BLOCK DET THE)
                                       (TABLE DET THE))
The block is on the table.      (ON* (BLOCK DET THE)
                                     (TABLE DET THE))

The block is clear.             (CLEAR* (BLOCK DET THE))
The block is supportable.       (SUPPORTABLE* (BLOCK DET THE))

Pick up the block.              (PICKUP (BLOCK DET THE))
Put the block down.             (PUTDOWN (BLOCK DET THE))
```

Figure 1-2: Parallelism between English sentences and Literals

In those examples, we can see the parallel between the syntactic structures of sentences and corresponding literals of their Procedural Logic representations. This suggests the study of a mechanical translation from English to Procedural Logic or vice versa for possible applications in natural language programming and automatic documentation procedures.

It is important to notice that the parallelism exists between the syntactic structures of the source language, English, and the object language, Procedural Logic. Because of this, we can make the translation procedures simple without any semantic or domain-specific knowledge.

1.3 Motivations and Goals

This work, motivated by the parallelism which was shown in the previous section, is about one direction of the translations: the translation from English to Procedural Logic. Since Procedural Logic has already been proved to be useful in several domains, we can use English in those domains if we can translate English sentences into Procedural Logic.

But the main objective is not to translate every English sentence into Horn Clauses, because there are many English sentences which are hard or impossible to translate only with the analysis of syntactic structures. As a matter of fact, we can translate only a small set of English sentences into Procedural Logic and can make programs only from careful descriptions of problem-solving methods. Therefore, the main objectives in the research on the translation of English to Procedural Logic are to study the following:

- The characteristics of the translation procedure.
- The characteristics of the subset of the English sentences which can be translated into Procedural Logic.
- The characteristics of domains in which we can use the translation to facilitate communication with computers.

This work is one of a series of preliminary experimental works which are on the way to accomplish those objectives ultimately. Two previous works, [Wang 83] and [Simmons 84b], made experiments over two tiny domains with simple subsets of English sentences. This work investigates a more complex domain, the Blocks World, with an extended set of English sentences. First, it presents a small grammar which can translate a subset of English into Horn Clauses. Later, we will present an interpreter which will accept English descriptions of a Blocks World and some problem-solving methods in the Blocks

World, translate them into Procedural Logic by using the small grammar, and interpret them as programs and data which are stored in its database. As was mentioned earlier, Procedural Logic provides a uniform way to deal with several database and programming tasks. And, because of the close correspondence of Procedural Logic and natural language, we can communicate with the interpreter solely in English which will be translated into Procedural Logic in the interpreter as far as we represent our intentions in the subset of English carefully.

In Chapter 2, we will present a small grammar which is used in translating a subset of English into Horn Clauses. First, we will show the procedure of translating simple sentences into literals. The translation of complex sentences will be the translation of several simple sentences of which the complex sentences are made up. Later, the treatment of noun phrases will be discussed.

In Chapter 3, an interpreter for the Blocks World will be described. The interpreter accepts four types of sentences, translates them into Horn Clauses by using the grammar described in Chapter 2, and interprets them as programs and data. The chapter will show several examples of the actual translation in the Blocks World.

In Chapter 4, after discussing some points which are not covered in the previous chapters, and showing related works, we will draw the conclusions from this research and suggest directions for further research.

Chapter 2

Translating English into Horn Clauses

This chapter describes the procedure of translating a subset of English into Horn Clauses with a small grammar. The basic units of Horn Clauses are literals, and the main task of the translation is to obtain the predicate names and arguments for literals. The procedure used here for this task is chiefly based on the syntactic structure of each sentence. Usually one English sentence is translated into one Horn Clause. Sometimes, however, it may result in several or no Horn Clauses. There are four types of sentences: *fact*, *rule*, *command*, and *query*. The translation procedures for them are largely similar. The grammar used here has been made to be applied to the experiment with the Blocks World of the next chapter, not for a general purpose, even though it does not have any domain-specific knowledge.

2.1 Translating a Simple Sentence

The basic syntactic unit of Horn Clauses is a literal, and as was shown in Section 1.3, a sentence having a simple structure may be translated into a literal. A *simple sentence* is a sentence which has only one main verb and no adverbial clauses. Its examples are in Figure 2-1.

Sentences of fact, command, and query types use typical forms of simple sentences. With the combinations of simple sentences and conjunctions, we can make sentences of more complex structures. The following subsections deal with simple sentences which can be found in each type of sentence which our grammar supports.

A block is on the table.	(fact)
The red block is clear.	(fact)
Put the red block down.	(command)
Is the red block on the table ?	(query)

Figure 2-1: Examples of the Simple Sentences

2.1.1 Fact

Facts are statements about relationships among the objects, states of the objects, or attributes of the objects. The simple sentences which our grammar supports are sentences containing the "be" verb. Figure 2-2 shows the allowed syntactic structures of the simple sentences which can be used in facts, and their corresponding structures as literals of Horn Clauses.

NPi means a noun phrase which has a noun with modifiers like adjectives or other nouns, or a Noun Clause, NCi. NCi is a clause which is a simple sentence or simple sentences, but one used as if it were a noun in the main sentence. For instance, the italicized parts of the following sentences are NC's.

It is consistent *that the block is on the table.*
The fact that a block is on a pyramid is false.

NC's are again translated into literals. In the figure, parenthesized items can be omitted in some cases. Rel-name is a name indicating the relationship between a NP and an adjective or a NP and a noun. COLOR is rel-name for "a block" and "red", and SUB(Subclass) is rel-name for "a red block" and "pyramid". NP* stands for a noun phrase or several noun phrases which are connected by and's, e.g. "a red block and a blue block". A noun with modifiers is translated

1. NP1 + be + prep + NP2	(prep* NP1 NP2)
2. NP1 + be + adj	(rel-name NP1 adj) (adj* NP1)
3. NP1 + be + (art) + noun	(rel-name NP1 noun) (noun* NP1)
4. NP1 + be + (art) + noun + of + NP2	(noun* NP1 NP2)
5. NP1 + be + adj + prep + NP2	(adj* NP1 NP2)
6. it + be + adj + NC1	(adj* NC1)
7. it + be + (art) + noun + NC1	(noun* NC1)
8. there + be + NP*	(THERE-IS NP*)
9. there + be + NP1 + prep + NP2	(prep* NP1 NP2)

Figure 2-2: Syntactic Structures and Literals: Fact

into a *frame*.[1] An example of frame is (BLOCK DET A COLOR RED) for a noun phrase, "a red block."

The predicate name is a word usually with the suffix, "*", which represents the relationships of the subject and the other objects (1,4,5,9), the states of subjects (2,3,6,7), the attributes of the subjects (2,3), or existence of subjects (8).

Examples of translations of the simple sentences of Figure 2-2 are in Figure 2-3. In the syntactic structures of 2 and 3, there are two options. The reason for making these two options is to gain computational efficiency in the later use of resulting Horn Clauses. Because the predicate name is the main key to searching for a Horn Clause, we have to make the predicate name become the word which will be given most often when the clause is searched for. An

[1] See [Rich 83] for the details of the frame structure.

```
1. The  block is on the table.        (ON* (BLOCK DET THE)
                                           (TABLE DET THE))
2. The  block is large.               (SIZE (BLOCK DET THE)
                                            LARGE)
   The block is clear.                (CLEAR* (BLOCK DET THE))

3. The block is a pyramid.            (SUB (BLOCK DET THE)
                                           PYRAMID)
   The block is a support.            (SUPPORT* (BLOCK DET THE))

4. X is a member of Y.                (MEMBER* X Y)

5. X is equal to Y.                   (EQ* X Y)

6. It is consistent that X is on Y.  (CONSISTENT* (ON* X Y))

7. It is a fact that the block is clear.
                                      (FACT* (CLEAR* (BLOCK DET THE)))
8. There is a  block and a table.     (THERE-IS ((BLOCK DET THE)
                                                 (TABLE DET THE)))
9. There is a block on the table.     (ON* (BLOCK DET A)
                                           (TABLE DET THE))
```

Figure 2-3: Examples of Literals: Fact

adjective or a noun may be used as the prime key to retrieving clauses, or may be the value which is looked for through the other keys. For example, we usually want to find the color of an object, and want to know if something is clear. In the first case, the color "red" may be searched with the keys of "block" and "color". In the latter case, the search proceeds with the key, "clear". There is no specific boundary between the words of two options. Sometimes, an adjective or a noun can be categorized into the first option, but sometimes not. The strategy used here is that, if an adjective or a noun represents a static characteristic of NP, then the first option is used. Otherwise, the second form will be the choice. But, if we choose the first form, the resulting literal exists only during the translation procedure. Later it will be incorporated into the frame of NP and the literal will disappear. Also the predicate "THERE-IS" is a temporary predicate and exists only during the translation. Details of these aspects will be described in the next section.

It should be noted that the noun clause of the structure 7 is translated into a literal. As mentioned in Chapter 1, a literal can have a literal as its argument. In 8, NP* yields a list of frames or a list of literals.

2.1.2 Command

A command is an imperative statement which starts with a transitive verb. In our grammar, both forms of active and passive voices of commands are accepted. The active voice form is used in actual commanding, while the other form is used in describing the programs defining the command. Figure 2-4 represents the syntactic structure of simple sentences used in commands and their corresponding formats of literals.

Because the passive forms are only used in describing programs and the programs are given to the actor who will execute the commands later, we do not need the phrase such as "by somebody", and the structures in the figure reflect it. The predicate names are made up of the transitive verb with suffix "*", particle, or preposition. Examples of the literals of simple sentences of commands are in Figure 2-5.

2.1.3 Query

A query is a question about known facts. Queries can be divided into two types- yes/no questions and WH-questions. The former ones start with the "be" verb or an auxiliary verb, while the latter ones start with a question term which includes one of what, which, how, where, who, and whom. The type covered by our grammar is only the former one. Since the only syntactic forms available for facts are the ones with the "be" verb, the queries in our grammar should start with the "be" verb, too.

The only difference between a fact and a yes/no query about the fact is not the literal itself, but the post-processing of resulting Horn Clauses. The Horn Clause of a fact is asserted while that of a query is to be proved. In the translation procedure, a simple sentence of a query is translated into the literal

```
ACTIVE VOICE

 1. trans.verb + particle + NP1
    trans.verb + NP1 + particle       (trans.verb.particle NP1)

 2. trans.verb + NP1                  (trans.verb* NP1)

 3. trans.verb + NP1 + prep + NP2     (trans.verb.prep NP1 NP2)

PASSIVE VOICE

 4. NP1 + be + verb(p.p. form) + particle
                                      (trans.verb.particle NP1)
 5. NP1 + be + verb(p.p. form)        (trans.verb* NP1)

 6. NP1 + be + verb(p.p. form) + prep + NP2
                                      (trans.verb.prep NP1 NP2)
 7. It + is + verb(p.p. form) + NC1   (trans.verb NC1)
```

Figure 2-4: Syntactic Structures and Literals: Commands

```
 1. Pick up the block.
    Pick the  block up.               (PICKUP (BLOCK DET))

 2. Make it that
       the block is on the table.     (MAKE* (ON* (BLOCK DET THE)
                                                  (TABLE DET THE)))
 3. Put the block on the table.       (PUTON (BLOCK DET THE)
                                             (TABLE DET THE))

 4. The block is picked up.           (PICKUP (BLOCK DET THE))

 5. The fact that the block is clear (PRINT* (CLEAR*
        is printed.                                (BLOCK DET THE)))

 6. The block is put on the table. (PUTON (BLOCK DET THE)
                                          (TABLE DET THE))
 7. It is asserted that            (ASSERT* (ON* (BLOCK DET THE)
       the block is on the table.                (TABLE DET THE)))
```

Figure 2-5: Examples of Literals: Command

of the corresponding simple sentence of a fact.[2] First, a simple sentence of a query is translated into the similar simple sentence of a fact, and the fact will be translated into a literal according to Figure 2-2. The translation procedure from a query to a fact is in Figure 2-6, and examples are in Figure 2-7.

```
1. be + there + remainders --------> there + be + remainders
2. is + it + remainders    --------> It + is + remainders
3. be + NP1 + remainders   --------> NP1 + be + remainders
```

Figure 2-6: Query to Fact

```
1. Is there a pyramid ?                (THERE-IS (PYRAMID DET A))

2. Is it consistent to put
     a block on a pyramid ?            (CONSISTENT* (PUTON (BLOCK DET A)
                                                   (PYRAMID DET A)))

3. Is the block on the table ?         (ON* (BLOCK DET THE)
                                            (TABLE DET THE))
```

Figure 2-7: Queries and Literals

2.1.4 Rule

Rules are the main parts of programs for problem-solving methods and inferences. Simple sentences used in rules are those which are used in facts or commands (passive forms), and there are no special syntactic structures for simple sentences used in rules. Therefore, we will discuss the structure of the rule in the sentence level.

---

[2]There are some differences in dealing with some predicate names and noun phrases of a query and those of a fact. Usually, however, the resulting literals are the same. For the time being, we will not think about those different cases. The different cases will be discussed in the next section.

The HCPRVR program for translating simple sentences into literals is in Appendix A.

2.2 Translation of a Sentence

The translation of a sentence is done by analyzing the sentence into several simple sentences, the translation of which was discussed in the preceding section. A sentence is composed of one or several simple sentences and conjunctions such as "and" and "if". There are some allowed formats of sentence which can be made up of simple sentences and conjunctions. They depend on the types of the sentences. Among these formats, those of rules are the most complex ones, and explained first.

Rule Structure = *if-clause then-clause*
| *simple-snt if-clause*
| It is always the case that *simple-snt*
| It is not the case that *simple-snt*
if-clause = *if-word simple-snt* {and *simple-snt*
| and *if-word simple-snt*}
then-clause = then *simple-snt* | *simple-snt*
if-word = if | unless

| means alternatives and { } means unspecified number of repetitions (possibly 0).

Figure 2-8: Structures of Rules

Figure 2-8 is the allowed structures of a rule which can be made up of simple sentences in our grammar. The simple sentences should be those used in facts, or passive forms of those used in commands. A rule is mainly composed of an if-clause and a then-clause. The latter should be a simple sentence while the former may include a simple sentence or several simple sentences which are connected by and's, if's, and unless's.

The format of the Horn Clause for that type of rule is given below.

(then-literal IF if-literal$_1$ if-literal$_2$ ...)

where then-literal is the literal of then-clause
and if-literal$_i$ is the literals of if-clause.

In simple sentences of the last section, we have not dealt with a negative sentence, because negative information cannot be represented as a literal unless we use a negative predicate name and specify that a predicate is a negative one of the other by a rule. In rules, however, we can represent negative information in if-clauses. That is, by using the predicate name, UNLESS+, we can represent a negative premise for the conclusion. UNLESS+ takes any number of literals as its arguments. If any of them is false, the truth value of this literal is true. (Therefore, a logical-and is assumed among the literals in UNLESS+.) Examples of rules and their Horn Clauses are in Figure 2-9.

```
A block is picked up if the block is clear
                    and the robotarm is clear.
((PICKUP (BLOCK DET A)) IF (CLEAR* (BLOCK DET THE))
                           (CLEAR* (ROBOTARM DET THE)))

X is clear unless something is on X.
((CLEAR* X) IF (UNLESS+ (ON* SOMETHING X)))
```

Figure 2-9: Examples of Rules and Horn Clauses

There are two special types of syntactic structures for rules which are composed of only a simple sentence while others need at least two simple sentences (one for then-clause and one for if-clause). The reason for providing those special structures is that their meanings are those of rules and more than one literal is needed to represent their meanings even if they look like facts. The second special structure is used for the negative information. Their corresponding Horn Clauses are in 2-10.

It is always the case that *simple-snt*
(simple-literal IF !**)

It is not the case that *simple-snt*
(simple-literal IF ! (FAIL)***)

where simple-literal is the literal of simple-snt.

** "!" is treated as a literal in HCPRVR, and it always succeeds. But, it marks the boundary which HCPRVR can not backtrack over.
*** (FAIL) is a literal which always fails to be proved.

Figure 2-10: Horn Clauses of Two Special Rule Structures

A fact may be a simple sentence or several simple sentences which are connected by and's. In the case of several simple sentences, the literals of these simple sentences are combined into a list to avoid being confused with rules, and will be asserted separately later. A logical-and may be thought as the predicate name of the list, but it does not appear in the list. "IF" is added as a deliminator in the Horn Clauses. Figure 2-11 shows some examples of facts.

```
A red block is on the table.
((ON* (BLOCK DET A COLOR RED) (TABLE DET THE)) IF)

A white block is on the red block
 and a blue block is on the white block.
(((ON* (BLOCK DET A COLOR WHITE)(BLOCK DET THE COLOR RED))
  (ON* (BLOCK DET A COLOR BLUE)(BLOCK DET THE COLOR WHITE)))
  IF)
```

Figure 2-11: Facts and Horn Clauses

A command may be a sentence of an active form having one or more than one simple sentences whose voices are active. The allowed form of a command is the same as that of a fact but the allowed types of simple sentences are different from one another.

A yes/no question must be a simple sentence which starts with the "be" verb and ends with the question mark "?". In this work, only simple questions are covered, since the translation of WH-questions, which are indefinitely complex, is a big problem by itself, and it is not a primary interest of this thesis. General Questioning/Answering systems with Clausal Logic can be found in [LeVine 80] and [Tseng 83].

Figure 2-12 is the HCPRVR program for the sentence analysis described so far. The top-level predicate name is SNT(Sentence) and the lowest one is SIMSNT(Simple Sentence). The translation is done by proving the literal below,

(SNT (sentence) Variable)

and the result of translation will be the value of Variable. In the program, facts, commands, and yes/no questions are treated as special cases of rules, that is, rules with no if-clause. CONJSNT deals with the if-clause's and SIMCONJSNT processes simple sentences which are connected by and's.

2.3 Frames for Noun Phrases

This section discusses how to treat the noun phrases in the translation procedure. Even though the procedure described in this section is applied to each simple sentence, the necessity of the procedure comes from the interactions of several sentences which refer to the same object.

When we translate the following text according to the process discussed in the previous section,

A blue block is on the table.

```
(AXIOMS
'(((SNT (IT IS ALWAYS THE CASE THAT . X) (V IF !))
   <
   ! (SIMSNT X V NIL))
  ((SNT (IT IS NOT THE CASE THAT . X) (V IF ! (FAIL)))
   <
   !
   (SIMSNT X V NIL))
  ((SNT (IF . X) (V1 IF . V))
   <
   ! (CONJSNT (AND IF . X) V R)
   (SIMSNT R V1 NIL))
  ((SNT (UNLESS . X) (V1 IF . V))
   <
   !
   (CONJSNT (AND UNLESS . X) V R) !
   (SIMSNT R V1 NIL))
  ((SNT X (U IF . V1))
   < (SIMCONJSNT X V R) (NORM V U) (CONJSNT R V1 NIL))))

(AXIOMS
'(((CONJSNT (THEN . Y) NIL Y))
  ((CONJSNT (AND IF . Y) V2 R1)
   <
   !
   (SIMCONJSNT Y V R)
   (CONJSNT R V1 R1)
   (APPEND* V V1 V2))
  ((CONJSNT (AND UNLESS . Y) ((UNLESS+ . V) . V1) R1)
   <
   (SIMCONJSNT Y V R)
   (CONJSNT R V1 R1))
  ((CONJSNT X NIL X))))

(AXIOMS
'(((SIMCONJSNT NIL NIL NIL) < !)
  ((SIMCONJSNT (IF . X) NIL (AND IF . X)))
  ((SIMCONJSNT (UNLESS . X) NIL (AND UNLESS . X)))
  ((SIMCONJSNT (THEN . X) NIL (THEN . X)))
  ((SIMCONJSNT X (U . V) R)
   < (SIMSNT X U R1) (SIMCONJSNT R1 V R) )
  ((SIMCONJSNT X NIL X))))
```

Figure 2-12: HCPRVR program for a Sentence Analysis

A white block is on the block.
The block is large.

the resulting Horn Clauses will be the following:

```
((ON* (BLOCK DET A COLOR BLUE)(TABLE DET THE)) IF)
((ON* (BLOCK DET A COLOR WHITE)(BLOCK DET THE)) IF)
((SIZE (BLOCK DET THE) LARGE) IF)
```

The problem in the resulting Horn Clauses is that there is no knowing to which objects each noun phrase refers later, because the Horn Clauses are not saved in a serial list, but according to the predicate name. If we use proper nouns only, there will no problem like this. But, in real situations, we have to deal with common nouns, and pronominal references. To make an efficient solution of the problem, we decided to represent each common noun and all of its attributes in a frame structure with an index, and to use the index instead of the noun phrase in Horn Clauses. The reason for using the frame structure is to get the computational efficiency of the structure and the reason for using indices is to distinguish the identical objects which are not the same objects. In this scheme, the Horn Clauses above would be represented in the following way.

```
((ON* (INST 2)(INST 1)) IF)
((ON* (INST 3)(INST 2)) IF)
((INST 1 TABLE) IF)
((INST 2 BLOCK COLOR BLUE) IF)
((INST 3 BLOCK COLOR WHITE SIZE LARGE) IF)
```

The predicate name for the frame is INST and each frame has an unique number as an index. A new index is generated by the grammar whenever the grammar finds a new object. It is remarkable that the last sentence only changed the content of the frame, and disappeared completely. There is another type of simple sentence which does not yield any Horn Clause but generates frames. This is "there + be + NP*". The predicate THERE-IS is given temporarily and the whole literal disappears after the frames are made for the noun phrases in the literal. The other types of a simple sentence of facts may or may not generate the frames. But, if they produce frames, the frames

are made as a side effect. In making a frame, DET and its value are omitted for convenience.

Finding the correct index for a noun phrase is an example of anaphoric resolution. In this grammar, a simple method is used. By maintaining a list of frames in the reverse order of references [Ehrlich 81], and searching for the candidate in the list, the frame which is encountered first and fulfills all the requirements of the noun phrase will be used as the prime candidate.

There is a procedure which is very helpful in dealing with frames. It is *Relaxed Unification* [Simmons 83]. Unlike the Unification algorithm which has no restriction on the format of arguments, the algorithm should have the arguments of the following form.

$$(\text{Head} \quad \text{Arc-name}_1 \quad \text{Arc-val}_1 \quad \text{Arc-name}_2 \quad \text{Arc-val}_2 \quad \ldots)$$

While Unification algorithm requests that two arguments be exactly the same or that they be made the same by substituting appropriate values for variables in them, this algorithm thinks that the first argument is matched with the second if the head words are same and all the pairs of arc-name and arc-val of the first argument are a subset of those of the second one. The order of pairs does not matter.

Also the algorithm uses a taxonomic structure in matching two objects. For example, if the first argument is "a block" and the second one is "a pyramid", the algorithm thinks that they match because "pyramid" is a subclass of "block", and every pyramid is a block.

The HCPRVR program which compares two frames by using the concept of Relaxed Unification algorithm is shown in Figure 2-13. The program is the same as Relaxed Unification algorithm except that the first pair of arc-name and arc-val is always the index and noun for the frame and, as an argument, an index of a frame can be given instead of the frame itself. This program is useful in searching for a frame which has certain attributes or in checking if a frame has certain properties. For instance, to find a block which is red, we put (INST N BLOCK COLOR RED) as the fist argument and one of

```
(AXIOMS
'(((RELMATCH (INST . X) (INST . X)))
  ((RELMATCH (INST N X . X1) (INST N X . Y))
   <
   (RELUNIF (INST N X . X1) (INST N X . Y)))
  ((RELMATCH (INST N) (INST N X . Y))
   <
   (INST N . Z)
   (RELMATCH (INST N . Z) (INST N X . Y)))
  ((RELMATCH (INST N X . Y) (INST N))
   <
   (INST N . Z)
   (RELMATCH (INST N X . Y) (INST N . Z)))
  ((RELMATCH (INST N X . X1) (INST N Y . Y1))
   <
   (NOUN X _NF1 U1)
   (NOUN Y _NF2 U2)
   (FRAMEARC X Y SUB)
   (RELUNIF (INST N Y . X1) (INST N Y . Y1)))
  ((RELMATCH (INST N Y . X1) (INST N X . Z))
   <
   (NOUN X _NF1 U1)
   (NOUN Y _NF2 U2)
   (FRAMEARC X Y S)
   (RELUNIF (INST N X S Y . X1) (INST N X . Z)))))

(AXIOMS
'(((RELUNIF (INST N X) (INST N X . Y)))
  ((RELUNIF (INST N X U V . R) (INST N X . Y))
   <
   (MEMPAIR (U V) Y)
   (RELUNIF (INST N X . R) (INST N X . Y)))))

(AXIOMS
'(((MEMPAIR (X Y) (X Y . Z)))
  ((MEMPAIR (X Y) (U V . Z)) < (MEMPAIR (X Y) Z))))
```

Figure 2-13: Relaxed Unification

existing frames as the second argument. If the program succeeds, a red block is found and the value of variable N is the index of the frame. Otherwise, we choose another from the remaining frames and check it until either we find one or there is no frame left untried.

To check if a block is large, we use the program proving the following Horn Clause.

```
(RELMATCH (INST X BLOCK SIZE LARGE)(INST n))
where (INST n) is the index of the frame of the block.
```

If it succeeds, the block is large. Otherwise, it is not.

While a noun phrase of a fact indicates a specific object, a noun phrase with an indefinite article of a rule or a query does not indicate a specific object. The noun phrase "a block" of the sentence below, for example, does not specify any particular object, and it is rather a variable.

> A block is clear unless something is on the block.
> Is a block on the red block ?

But we cannot simply substitute it with a variable like X or (INST X). It is required that the value which is bound to variable must be a block. Therefore, a noun phrase having an indefinite article may be thought as a variable with some qualifiers and the qualifiers restrict the possible values of the variable. In our grammar, each noun clause with an indefinite article in a rule or a query is substituted with a variable, and a literal which restricts the values for the variable is inserted as Figure 2-14.

Several English words are the same as the variables in their original meanings. Those words are treated as the actual variables in this grammar. "Something", "everything", and "anything" are their examples. The ordinary variables supported in HCPRVR are letters from "M" through "Z" optionally followed by a number from 1 to 4, or any atom which starts with the underline "_", e.g. _VARIABLE.

Also, in queries and rules, literals about some attributes of NP's such

```
((CLEAR* (INST N)) IF
          (RELMATCH (INST N BLOCK)(INST N))
          (UNLESS+ (ON* SOMETHING (INST N))))

(((ON* (INST N)(INST 2))
  (RELMATCH (INST N BLOCK)(INST N))) IF)
```

Figure 2-14: Horn Clauses of Noun Phrase Variables

as (SIZE (INST 3) LARGE) or (COLOR (INST 2) BLUE) are substituted with (RELMATCH (INST V1 V2 SIZE LARGE)(INST 3)) or (RELMATCH (INST V1 V2 COLOR BLUE) (INST 2)).

The procedure described in this section will be invoked after the procedure of the previous section has been accomplished successfully. The remaining question is how the Horn Clauses which are generated by this grammar should be processed in order to be interpreted as programs and data. This will be discussed in the next chapter with the examples of the Blocks World.

Chapter 3

HCTRANS and Blocks World

This chapter describes an interpreter which has been developed to translate several types of English sentences about the Blocks World as an example domain. The interpreter is equipped with the small grammar described in the previous chapter. Because neither the interpreter nor the grammar includes any domain-specific knowledge about the Blocks World, the interpreter can be experimented with in other domains with little modification.

3.1 An Interpreter: HCTRANS

The interpreter which has been developed in this research is called HCTRANS(Horn Clause TRANSlator). Its major functions can be summarized as below.

1. It accepts English descriptions of objects, relationships among them, and problem-solving methods. It translates them into Horn Clauses, and stores the Horn Clauses in the database(DB).
2. It interprets the resulting Horn Clauses as programs and data, and uses them to accomplish several tasks such as answering questions and executing commands.
3. It maintains integrity of the DB.

The top level executive of HCTRANS is written in ELISP, whose code is given in Appendix B.1. The executive reads a sentence at a time and pre-processes it, and calls the HCPRVR programs which translate the sentence into Horn Clauses, and finally post-processes the clauses according to the type of the sentence.

Through the pre-processing, it changes variables into dummy variables to prevent them from being substituted with anything in the grammar and to make them remain as variables in the resulting Horn Clause, and decides the type of a given sentence. The decision of the type of the sentence is made only with syntactic structure of the sentence. If it ends with "?", it is a query. If it has if-clause or is one of two special structures for rules, it is regarded as a rule. And if it starts with a transitive verb, it is a command. Otherwise, it is a fact.

After the grammar, which is also represented in HCPRVR programs, translates a sentence into a Horn Clause or several Horn Clauses, the HCPRVR programs which make the following post-processing are called. The post-processing is based on the type of the sentence which was decided in the pre-processing.

Fact	It checks the resulting Horn Clauses according to Consistency Rules(See Section 3.5) for integrity of the DB and stores them in the DB. Later, they are used mainly as data.
Rule	It stores them in the DB directly. They are used as programs. In the DB, there is no difference between a fact and a rule. Every Horn Clause is stored with the Clauses having the same predicate name.
Command	After checking it with consistency rules to see if the command is reasonable, it finds a program of the command, and executes it to accomplish the required task. The program should be given in advance.
Query	It attempts to prove the resulting Horn Clauses in the DB. If it succeeds, it answers "Yes". Otherwise, it says "No". As for the DB, we assume the closed-world DB. [Clark 78] [Jaffer 83] In the closed-world DB, failing to prove a theorem in DB means that the theorem is false in DB.

The details of post-processing can be seen in the HCPRVR program of Figure 3-1. The complete list of the program is in Appendix B.2.

For a sentence of fact, the interpreter first parses it into a Horn Clause

```
(AXIOMS
'(((FACT X) < (SNT X (Y IF)) (MULTI-FACTS Y))))

(AXIOMS
'(((MULTI-FACTS NIL) < !)
  ((MULTI-FACTS ((U . V) . Y))
   <
   ! (SING-FACT (U . V)) ! (MULTI-FACTS Y))
  ((MULTI-FACTS X) < (SING-FACT X))))

(AXIOMS
'(((SING-FACT X)
   <
   (CHK-OBJS X Y) (CONSISTENT Y) (ASSERT-FACT Y))))

(AXIOMS
'(((COMMAND X) < (SNT X (Y IF)) (MULTI-CMDS Y))))

(AXIOMS
'(((MULTI-CMDS NIL) < !)
  ((MULTI-CMDS ((U . V) . Y))
   <
  (SING-CMD (U . V)) ! (MULTI-CMD Y))
  ((MULTI-CMDS X) < (SING-CMD X))))

(AXIOMS
'(((SING-CMD X) < (CHK-OBJS X Y) (CONSISTENT Y) (EXECUTE Y))))

(AXIOMS
'(((RULE X) < (SNT X Y) (TRANS-OBJS Y Z) (ASSERT-RULE Z))))

(AXIOMS
'(((QUERY X)
   <
   (SNT X (Y IF))
   (TRANS-OBJS Y Z)
   (SETV* W (SUBLIS (PAIRVARS VARLIST) 'Z))
   (ASKS W))))
```

Figure 3-1: HCPRVR program of HCTRANS

as described in Section 2.2 (SNT). Because a sentence may result in several literals in case that several simple sentences are connected by and's (MULTI-FACTS), it checks the result of SNT, and if it is a list of literals, it processes them as if they were separate sentences(SING-FACT). For each literal, it substitutes a noun phrase with a frame index, eliminates THERE-IS and rel-name predicates (CHK-OBJS), and adjusts the list of frames. After it checks the result with the consistency rules (CONSISTENT), it asserts the literal as a Horn Clause (ASSERT-FACT).

The processing of a sentence of command is the same as that of a sentence of fact except that HCTRANS executes the program instead of storing it in the DB. The execution of command is the same as proving the literal of the command. HCPRVR will automatically retrieve the program of the command, and execute it.

The post-processing of rules and queries is simpler. There are no consistency rules for them. The major difference of rules and queries from facts and commands is in the grammar as mentioned in Chapter 2; the grammar changes a noun phrase, which has an indefinite article, e.g. "a", "an", and "another" for a variable and a qualification literal which constrains the possible values of the variable. It also changes the literals of THERE-IS and rel-name predicates by using RELMATCH (TRANS-OBJS). The Horn Clause of a rule is asserted and that of a query is asked.

There are three predicate names whose evaluations are made by the interpreter program. They are TRUE*, FALSE*, and LIST*. Because the evaluations of them are too primitive or dependent on the implementation of the interpreter, the interpreter provides the programs for them instead of asking users to do so. The meanings of them are explained as below. For each entry, the first word is the one used in English and the second one is the predicate name.

True --> TRUE* A literal with this name is true if a literal or a list of literals which is given as the argument is true, that is, if the literal or all the literals in the list can be proved in the current DB.

False --> FALSE* The value of the literal with this predicate name is the reverse of TRUE* and it is equivalent to (UNLESS+ (TRUE* X)).

List --> LIST* The value of the literal with this predicate name is true if the argument is a list of literals or a list of indices.

3.2 Blocks World: An Example Domain

The Blocks World is an example domain which has been used in many researches in the AI field [Winograd 72] [Fikes 71]. The reasons it has been chosen as an example domain here are as follows:

- The domain is simple and well defined. The Blocks World is like a simplified real world.
- Its characteristics are well exploited in the several researches.
- Even though it is simple, it is not a domain which can be attacked easily. For example, it has the characteristic of non-monotonicity.
- We can extend the domain easily.

The discussions in the remaining part of this chapter will proceed with the test sessions of HCTRANS whose recorded listings are presented throughout the section. In those recorded sessions, user's inputs are those which are typed in when the prompts have come ("**" in LISP mode, and "-" in HCTRANS mode). Comments are in the brackets ("{" and "}") and italicized. When it prints Horn Clauses, it will substitute the indices with actual frames for readability.

The basic assumptions on the blocks world which is described here are as follows.

- There is only one table, and only one robotarm. The size of the table is infinite and there is always room for more blocks on the table.
- The robotarm can hold one block at a time.

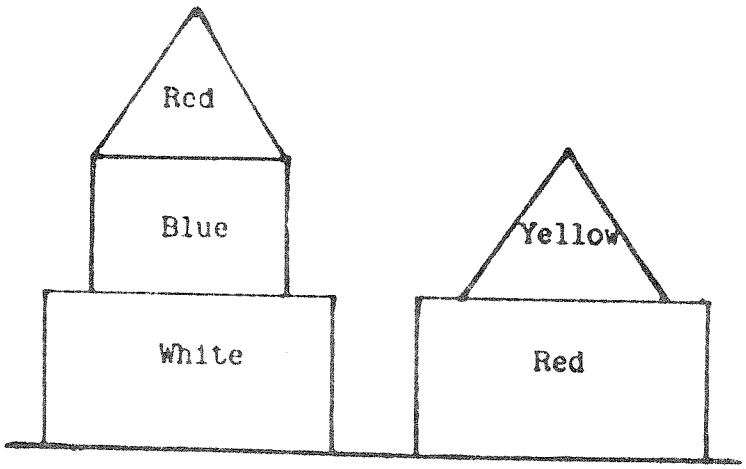
- Only one block can be on another block at a time.
- A pyramid cannot support anything.
- The table and the robotarm can not be put on anything.

The purpose of the test sessions is to show the ability of the grammar and the interpreter. Several types of English sentences with several purposes are all translated into Horn Clauses and interpreted as programs and data.

3.3 Examples of Facts and Rules

We will start the test run with a simple example of Blocks World in Figure 3-2. We will describe the example to the interpreter in English, and the interpreter will make the DB for it. We will also add some inference rules, and ask questions about the example.

```
[PHOTO:  Recording initiated  Sun 8-Jul-84 12:49AM]

[Link from CS.YU, TTY36]

 Tops-20 Command processor 5.1(121700)
2@ELISP                    {Getting into the ELISP mode.}
Elisp, 4 1 84


A version of recursion-checking HCPRVR.FLAP is loaded, and
  functions and HCPRVR program for HCTRANS are loaded, too

To use HCTRANS, type (HCTRANS)


*(HCTRANS)                 {Getting into the interpreter.}
Welcome to HCTRANS, a translator from English to Horn Clauses

  I can accept facts, rules, queries, and commands , and
    make inferences with them.  To exit, type ^G at any time.

  You can type in any LISP expression, too.

{Starting to describe the Blocks World to the interpreter.}
```



Rule: If X is large and Y is small
then X is larger than Y.
It is always the case
that the table is larger than anything.

If X is on Y and Y is on Z
then X is above Z.

If X is on Y then Y is under X.

Figure 3-2: Description of a Blocks World

```
-THERE IS A TABLE AND A ROBOTARM .

(Asserted: (INST 1 TABLE))
(Asserted: (INST 2 ROBOTARM))

{Whenever a new object is introduced,
 the interpreter creates a frame for it.}
{Note that there is no predicate name for
 "there + be + NP*" statement.}

-A WHITE BLOCK IS ON THE TABLE
```

```
-AND A BLUE BLOCK IS ON THE WHITE BLOCK .
```

{Two simple sentences are connected by "and" to make a fact.}
{This sentence not only introduces new objects
but addresses the relationships among objects.}

```
(Asserted: (INST 3 BLOCK COLOR WHITE))
(Asserted: (ON* (INST 3 BLOCK COLOR WHITE) (INST 1 TABLE)))
(Asserted: (INST 4 BLOCK COLOR BLUE))
(Asserted: (ON* (INST 4 BLOCK COLOR BLUE)
                (INST 3 BLOCK COLOR WHITE)))
```

```
-THE WHITE BLOCK IS LARGE AND THE BLUE BLOCK IS SMALL .
```

{This sentence does not introduce any new object
nor does it address any relationship among objects.
But, it describes some attributes of known objects
and it changes the frames of objects.}

```
(Deleted: (INST 3 BLOCK COLOR WHITE))
(Asserted: (INST 3 BLOCK SIZE LARGE COLOR WHITE))
(Deleted: (INST 4 BLOCK COLOR BLUE))
(Asserted: (INST 4 BLOCK SIZE SMALL COLOR BLUE))
```

```
-THERE IS A RED BLOCK ON THE BLUE BLOCK .

(Asserted: (INST 5 BLOCK COLOR RED))
(Asserted: (ON* (INST 5 BLOCK COLOR RED)
                (INST 4 BLOCK SIZE SMALL COLOR BLUE)))
```

```
-THE BLOCK IS A PYRAMID .

(Deleted: (INST 5 BLOCK COLOR RED))
(Asserted: (INST 5 BLOCK SUB PYRAMID COLOR RED))
```

```
-A LARGE RED BLOCK IS ON THE TABLE
-AND A YELLOW PYRAMID IS ON THE BLOCK .

(Asserted: (INST 6 BLOCK SIZE LARGE COLOR RED))
(Asserted: (ON* (INST 6 BLOCK SIZE LARGE COLOR RED)
                (INST 1 TABLE)))
(Asserted: (INST 7 PYRAMID COLOR YELLOW))
(Asserted: (ON* (INST 7 PYRAMID COLOR YELLOW)
                (INST 6 BLOCK SIZE LARGE COLOR RED)))
```

{*Starting to input the rules to the interpreter.*}

```
-IF X IS LARGE AND Y IS SMALL THEN X IS LARGER THAN Y .
```

{*This rule is valid on the assumption that
there are only two sizes in the world.* }

```
(Asserted: ((LARGER* X Y)
            IF
            (RELMATCH (INST _VAR1 _VAR2 SIZE LARGE) X)
            (RELMATCH (INST _VAR3 _VAR4 SIZE SMALL) Y)))
```

{*The simple sentences about the attributes
of objects are changed into RELMATCH's.*}

```
-IT IS ALWAYS THE CASE
-THAT THE TABLE IS LARGER THAN ANYTHING .
```

{*A special format for a rule.*}

```
(Asserted: ((LARGER* (INST 1 TABLE) ANYTHING) IF !))
```

```
-IF X IS ON Y THEN Y IS UNDER X .
```

{*Antonym. Because the interpreter does not have the knowledge
about the relationships among words, we have to specify them
by rules.*}

```
(Asserted: ((UNDER* Y X) IF (ON* X Y)))
```

```
-IF X IS ON Y AND Y IS ON Z
-THEN X IS ABOVE Z .
```

```
(Asserted: ((ABOVE* X Z) IF (ON* X Y) (ON* Y Z)))
```

{*Question and Answering Session.*}

```
-IS THERE A RED PYRAMID ?
```

{*Question about the existence of any red pyramid.*}

```
(YES (RELMATCH (INST 5 PYRAMID COLOR RED)
               (INST 5 BLOCK SUB PYRAMID COLOR RED)))
```

{*It finds a red pyramid.*}

-IS THE YELLOW PYRAMID ON THE RED BLOCK ?

{*It asks about the relationship between two objects.*}

```
(YES (ON* (INST 7 PYRAMID COLOR YELLOW)
          (INST 6 BLOCK SIZE LARGE COLOR RED)))
```

-IS THERE A WHITE PYRAMID ?

{*Again, it asks about the existence of an object,*
but the interpreter fails to find any.}

(NO)

-IS THE RED BLOCK ON THE TABLE LARGE ?

{*It asks about an attribute of an object.*}
{*Because there are two red blocks, we specify*
one by stating its relationship with another object.
The grammar uses the prepositional phrase
to locate the correct object.}

```
(YES (RELMATCH (INST 6 BLOCK SIZE LARGE)
               (INST 6 BLOCK SIZE LARGE COLOR RED)))
```

{*Note that RELMATCH is used for checking the*
attribute of the object, and the grammar locates
the correct object.}

-IS THE BLUE BLOCK LARGE ?

{*The blue block is small.*}

(NO)

-IS THE RED BLOCK ON THE BLUE BLOCK ABOVE THE WHITE BLOCK ?

{*This sentence can be interpreted in two ways, that is,*
"Is the red block on the blue block which is on
the white block?", and "Is the red block, which is on
the blue block, above the white block ?"
The interpreter tries to find any possible interpretation

which can make the answer "yes", and finds one of two interpretations makes it true.}

{This query also uses the rule about "above".}

```
(YES (ABOVE* (INST 5 BLOCK SUB PYRAMID COLOR RED)
             (INST 3 BLOCK SIZE LARGE COLOR WHITE)))
```

```
-IS THE WHITE BLOCK LARGER
-THAN THE RED BLOCK UNDER THE YELLOW BLOCK ?
```

{This query uses the rule about "larger-than".}
{The search of the rule is done by HCPRVR.}

```
(NO)
```

```
-IS THE BLOCK LARGER THAN BLUE BLOCK ?
```

{In this question, a simple anaphoric reference is resolved. The list of frames is maintained in the reverse order of references.}

```
(YES (LARGER* (INST 3 BLOCK SIZE LARGE COLOR WHITE)
              (INST 4 BLOCK SIZE SMALL COLOR BLUE)))
```

{To be continued in the next section.}

3.4 Examples of Commands

In this section, we will add the programs for three commands: PICKUP, PUTDOWN, and PUTON. It should be made clear that, since this is a kind of simulation, the real operations are primitive operations of ASSERT and DELETE which are primitive functions of the interpreter. There are some commands for which HCTRANS provides the programs, because they are related with the dealing with the memory unit or printers. They are ASSERT, DELETE, and PRINT. The first two are asserting or deleting some facts into or from the DB, and the last one prints out the argument in the screen.

Some special cases of PUTON are the same as other commands. "Put

something on the robotarm" is the same as PICKUP, and "put something on the robotarm on the table" is the same as PUTDOWN.

{Continued from the last record session.}

{Starting to give programs of commands.}

```
-IT IS ALWAYS THE CASE THAT THE TABLE IS CLEAR .
```

*{This and the next rules are about the predicate name
"clear", which will be used in other programs."
"clear" means that there is room for another block."}*

```
(Asserted: ((CLEAR* (INST 1 TABLE)) IF !))
```

```
-X IS CLEAR UNLESS SOMETHING IS ON X
-        AND UNLESS X IS ON THE ROBOTARM .
```

```
(Asserted: ((CLEAR* X)
            IF
            (UNLESS+ (ON* SOMETHING X))
            (UNLESS+ (ON* X (INST 2 ROBOTARM)))))
```

{There are three commands: Pick up, Put down , and Put on.}

```
-A BLOCK IS PICKED UP IF THE BLOCK IS CLEAR
-        AND IF THE ROBOTARM IS CLEAR
-        AND IF IT IS DELETED THAT THE BLOCK IS ON SOMETHING
-        AND IF IT IS ASSERTED THAT THE BLOCK IS ON THE ROBOTARM .
```

*{The program uses two primitive commands: Assert and Delete.
It first checks the pre-condition of the command.}*

```
(Asserted: ((PICKUP (INST _VAR1))
            IF
            (RELMATCH (INST _VAR1 BLOCK) (INST _VAR1))
            (CLEAR* (INST _VAR1))
            (CLEAR* (INST 2 ROBOTARM))
            (DELETE (ON* (INST _VAR1) SOMETHING))
            (ASSERT* (ON* (INST _VAR1) (INST 2 ROBOTARM)))))
```

*{Note that a noun phrase "a block" is substituted with a
variable and a literal which limits the possible values of*

the variable is inserted.}
{Also the word "something" is treated as a variable.}

```
-A BLOCK IS PUT DOWN IF IT IS ON THE ROBOTARM
-        AND IF IT IS DELETED THAT THE BLOCK IS ON SOMETHING
-        AND IF IT IS ASSERTED THAT THE BLOCK IS ON THE TABLE .
```

{The passive form of commands is used in rules, and it makes the rules look more natural.}

```
(Asserted: ((PUTDOWN (INST _VAR1))
            IF
            (RELMATCH (INST _VAR1 BLOCK) (INST _VAR1))
            (ON* (INST _VAR1) (INST 2 ROBOTARM))
            (DELETE (ON* (INST _VAR1) SOMETHING))
            (ASSERT* (ON* (INST _VAR1) (INST 1 TABLE)))))


-A BLOCK IS PUT ON THE ROBOTARM
-        IF THE BLOCK IS PICKED UP .

(Asserted: ((PUTON (INST _VAR1) (INST 2 ROBOTARM))
            IF
            (RELMATCH (INST _VAR1 BLOCK) (INST _VAR1))
            (PICKUP (INST _VAR1))))
```

{There are three rules for PUTON. The first two specify the operation is some special cases, and the last one is for general cases. The third rule uses PICKUP and another rule of PUTON as primitive operations.}

```
-A BLOCK IS PUT ON X IF THE BLOCK IS ON THE ROBOTARM
-        AND IF X IS CLEAR
-        AND IF IT IS DELETED THAT THE BLOCK IS ON THE ROBOTARM
-        AND IF IT IS ASSERTED THAT THE BLOCK IS ON X .

(Asserted: ((PUTON (INST _VAR1) X)
            IF
            (RELMATCH (INST _VAR1 BLOCK) (INST _VAR1))
            (ON* (INST _VAR1) (INST 2 ROBOTARM))
            (CLEAR* X)
            (DELETE (ON* (INST _VAR1) (INST 2 ROBOTARM)))
            (ASSERT* (ON* (INST _VAR1) X))))


-A BLOCK IS PUT ON X IF THE BLOCK IS CLEAR
-          AND IF X IS CLEAR
```

```
-           AND IF THE ROBOTARM IS CLEAR
-           AND THE BLOCK IS PICKED UP
-           AND THE BLOCK IS PUT ON X .
```

{*In this case, the whole operation is divided into two steps: Picking up a block and putting it on the desired place.*}

```
(Asserted: ((PUTON (INST _VAR1) X)
            IF
            (RELMATCH (INST _VAR1 BLOCK) (INST _VAR1))
            (CLEAR* (INST _VAR1))
            (CLEAR* X)
            (CLEAR* (INST 2 ROBOTARM))
            (PICKUP (INST _VAR1))
            (PUTON (INST _VAR1) X)))
```

{*Test runs of the command programs.*}

```
-PICK UP THE RED BLOCK ON THE BLUE BLOCK .

(Deleted: (ON* (INST 5 BLOCK SUB PYRAMID COLOR RED)
               (INST 4 BLOCK SIZE SMALL COLOR BLUE)))
(Asserted: (ON* (INST 5 BLOCK SUB PYRAMID COLOR RED)
                (INST 2 ROBOTARM)))
```

{*The actual operation is the deletion and assertion of facts.*}

```
-PUT IT DOWN .
```

{*A pronoun is used.*}

```
(Deleted: (ON* (INST 5 BLOCK SUB PYRAMID COLOR RED)
               (INST 2 ROBOTARM)))
(Asserted: (ON* (INST 5 BLOCK SUB PYRAMID COLOR RED)
                (INST 1 TABLE)))

-PUT THE YELLOW BLOCK ON THE TABLE .
```

{*The third rule of PUTON is used in this case.*}

```
(Deleted: (ON* (INST 7 PYRAMID COLOR YELLOW)
               (INST 6 BLOCK SIZE LARGE COLOR RED)))
(Asserted: (ON* (INST 7 PYRAMID COLOR YELLOW)
                (INST 2 ROBOTARM)))
```

```
(Deleted: (ON* (INST 7 PYRAMID COLOR YELLOW)
               (INST 2 ROBOTARM)))
(Asserted: (ON* (INST 7 PYRAMID COLOR YELLOW)
                (INST 1 TABLE)))
```

```
-PICK UP THE BLUE BLOCK ON THE WHITE BLOCK .

(Deleted: (ON* (INST 4 BLOCK SIZE SMALL COLOR BLUE)
               (INST 3 BLOCK SIZE LARGE COLOR WHITE)))
(Asserted: (ON* (INST 4 BLOCK SIZE SMALL COLOR BLUE)
                (INST 2 ROBOTARM)))
```

```
-PUT IT ON THE RED BLOCK .
```

{The second rule is used in this case.}

```
(Deleted: (ON* (INST 4 BLOCK SIZE SMALL COLOR BLUE)
               (INST 2 ROBOTARM)))
(Asserted: (ON* (INST 4 BLOCK SIZE SMALL COLOR BLUE)
                (INST 6 BLOCK SIZE LARGE COLOR RED)))
```

```
-PUT THE LARGE YELLOW BLOCK ON THE TABLE .
```

{This command is aborted because there is a unknown object in the command.}

```
(Unidentifiable Object: (BLOCK DET THE SIZE LARGE COLOR YELLOW))
(Unacceptable Command :
    (PUTON (BLOCK DET THE SIZE LARGE COLOR YELLOW)
           (TABLE DET THE)))
(Processing of input aborted)
```

```
-PUT THE LARGE RED BLOCK ON THE WHITE BLOCK .
```

{This command is aborted because there is no rule which can accomplish the task. Later, in Section 3.6, we will modify the rules to accomplish this task.}

```
(Unacceptable Command :
   (PUTON (BLOCK DET THE SIZE LARGE COLOR RED)
          (BLOCK DET THE COLOR WHITE)))
(Processing of input aborted)

-PUT THE BLUE BLOCK ON THE WHITE BLOCK ON THE TABLE .
```

{*This command can be interpreted in two ways. The one is "put the blue block which is on the white block on the table", the other is "put the blue block on the white block which is on the table."*}

```
(Unacceptable Command :
 (PUTON (BLOCK DET THE COLOR BLUE ON* (BLOCK DET THE COLOR WHITE))
        (TABLE DET THE)))
```

{*The first interpretation can not be accomplished, because there is no blue block on the white block.*}

```
(Deleted: (ON* (INST 4 BLOCK SIZE SMALL COLOR BLUE)
               (INST 6 BLOCK SIZE LARGE COLOR RED)))
(Asserted: (ON* (INST 4 BLOCK SIZE SMALL COLOR BLUE)
                (INST 2 ROBOTARM)))
(Deleted: (ON* (INST 4 BLOCK SIZE SMALL COLOR BLUE)
               (INST 2 ROBOTARM)))
(Asserted: (ON* (INST 4 BLOCK SIZE SMALL COLOR BLUE)
                (INST 3 BLOCK SIZE LARGE COLOR WHITE)))
```

{*But the second interpretation succeeded.*}
{*HCTRANS tries to find the interpretation with which it can accomplish the command.*}

{*To be continued in the next section.*}

3.5 Consistency Rules

{*Continued from the last recorded session.*}

{*A description and a command are given.*}

```
-ANOTHER WHITE BLOCK IS ON THE YELLOW BLOCK .
```

{*This description is inconsistent with the restriction on the Blocks World: No block can be on a pyramid, but the yellow block is a pyramid.*}

```
(Asserted: (INST 8 BLOCK COLOR WHITE))
(Asserted: (ON* (INST 8 BLOCK COLOR WHITE)
                (INST 7 PYRAMID COLOR YELLOW)))
```

{*But the interpreter accepts it.*}

```
-IS THE YELLOW PYRAMID UNDER A BLOCK ?

(YES (UNDER* (INST 7 PYRAMID COLOR YELLOW)
             (INST 8 BLOCK COLOR WHITE)))


-PUT THE WHITE BLOCK ON THE RED PYRAMID .
```

{This command makes a state which is not allowed.}

```
(Deleted: (ON* (INST 8 BLOCK COLOR WHITE)
               (INST 7 PYRAMID COLOR YELLOW)))
(Asserted: (ON* (INST 8 BLOCK COLOR WHITE)
                (INST 2 ROBOTARM)))
(Deleted: (ON* (INST 8 BLOCK COLOR WHITE)
               (INST 2 ROBOTARM)))
(Asserted: (ON* (INST 8 BLOCK COLOR WHITE)
                (INST 5 BLOCK SUB PYRAMID COLOR RED))
```

{But, the command is processed.}

```
-IS THE RED PYRAMID UNDER THE A BLOCK ?

(YES (UNDER* (INST 5 BLOCK SUB PYRAMID COLOR RED)
             (INST 8 BLOCK COLOR WHITE)))
```

In the recorded session above, there were some mistakes in descriptions and commands. But, the interpreter, having not detected it, stored the description and processed the command which resulted in an inconsistent DB. We could see the inconsistency by observing inconsistent answers to a set of questions.

To preserve integrity of the Blocks World whose possible semantics is described in Section 3.2, we need a way to state the constraints of the Blocks World to HCTRANS. Because HCTRANS stores everything in the DB, the problem can be reduced to the problem of preserving integrity of the DB [Nicholas 78a] [Nicholas 78b] [Kowalski 78] [Wiederhold 84]. That is, to preserve integrity of the Blocks World, we only need to check if the changes of the DB are consistent and allowed according to the semantics of the Blocks World.

In the interpreter, three of the four types of sentences can change the DB. Both facts and rules can add Horn Clauses into the DB, while a command adds some clauses and deletes some others. But, the determination of inconsistency of a rule with a set of rules is a complicated logic problem that has been discussed extensively and, has not been solved completely. Even though we can solve the problem in some cases, it usually demands lots of computing power, as well as a great deal of time.

The interpreter provides a mechanism for consistency constraints on facts and commands. That is, by stating the consistency constraints as rules of the form below, and by making the interpreter examine any new facts and commands with the rules, the DB integrity can be preserved. The rules are called as *Consistency Rules.*

It is consistent that <fact>
if <constraints>
It is consistent to <command>
if <constraints>

When we described the post-processing of facts and commands, we mentioned that the Horn Clauses of those sentences are checked with the consistency rules. If there is no rule to show that a clause of a sentence is consistent, the sentence will be rejected. (Actually, there will be backtracking for other possible interpretations of the sentence. If there is no interpretation which proves to be consistent according to any consistency rule, the sentence will be rejected.) For instance, if the consistency rules of Figure 3-3 had been provided, the inconsistent fact and command in the recorded session above would have been rejected.

In case of commands, we can put the consistency constraints in the programs directly. But, since there may be several rules for a command and this will make us state the constraints repeatedly, it is better to put the constraints separately. There are two occasions in which a command cannot accomplish the given task. The first one is the case that there is no program for the command currently, even though the command is consistent with the

It is consistent that a block is on X
if X is clear
and unless X is equal to a pyramid.

It is consistent to put a block on X
unless X is equal to a pyramid.

Figure 3-3: Examples of Consistency Rules

semantics of the Blocks World. In this case, by augmenting the program, we can make the interpreter accomplish the task. The second one is the case of an impossible command which is inconsistent with the semantics of the Blocks World, and which is prohibited by the consistency rules. In the latter case, augmenting the program does not change the situation.

When we make consistency rules for some commands, it is important to specify all the conditions in which the commands can be executed, even if the current programs for the commands can not accomplish the task. For example, in the case of the consistency rule for PUTON command, we may add the condition "unless X is clear", because the current programs can not execute the "put a block on X" if X is not clear. But, if we do so, then we will be in trouble when we augment the programs by using planning (See Section 3.6), because, by planning, it will be possible to execute the commands in the case of "X is not clear" while the consistency rule makes the command unacceptable. Therefore, we have to put all the possible conditions in consistency rules, even if some of them can not be accomplished currently.

By convention of HCTRANS, if there is no consistency rule for a predicate name at all, every fact or command of the predicate name will be accepted. The following recorded session is one with consistency rules for ON* and PUTON.

{*Continued from the record session of the last section.*}

{*We type in the consistency rules of ON* and PUTON.*}

```
-IT IS CONSISTENT THAT A BLOCK IS ON X
-       IF X IS CLEAR
-       AND X IS NOT EQUAL TO A PYRAMID .

(Asserted: ((CONSISTENT* (ON* (INST _VAR1) X))
            IF
            (RELMATCH (INST _VAR1 BLOCK) (INST _VAR1))
            (CLEAR* X)
            (UNLESS+ (EQ* X (INST _VAR2))
                     (RELMATCH (INST _VAR2 PYRAMID)
                               (INST _VAR2)))))


-IT IS CONSISTENT TO PUT A BLOCK ON X
-       UNLESS X IS EQUAL TO A PYRAMID .

(Asserted: ((CONSISTENT* (PUTON (INST _VAR1) X))
            IF
            (RELMATCH (INST _VAR1 BLOCK) (INST _VAR1))
            (UNLESS+ (EQ* X (INST _VAR2))
                     (RELMATCH (INST _VAR2 PYRAMID)
                               (INST _VAR2)))))


-ANOTHER WHITE BLOCK IS ON THE YELLOW BLOCK .
```

{*With the consistency rules, this fact can not be accepted, because the yellow block is a pyramid. The fact is rejected.*}

```
(Unacceptable Fact: (ON* (BLOCK DET ANOTHER COLOR WHITE)
                         (BLOCK DET THE COLOR YELLOW)))
(Processing of input aborted)


-PUT THE WHITE BLOCK ON THE RED PYRAMID .
```

{*Also this command is rejected according to the consistency rules.*}

```
(Unacceptable Command : (PUTON (BLOCK DET THE COLOR WHITE)
                               (PYRAMID DET THE COLOR RED)))
(Processing of input aborted)
```

{To be continued in the next session.}

3.6 Planning

So far, the possible commands are atomic: they do only one operation (actually two or four primitive operations) at a time. This section gives the implementation of planning in the Blocks World. By planning, the interpreter will make a plan to accomplish the task which cannot be made directly, but can be made if some preparatory tasks are accomplished.

The most prominent planning scheme in the Blocks World is SCRIPT. There are several versions of it. Because SCRIPT itself is very complex, we will use a simple version of it. The program is described under the command "make" in Figure 3-4. Its argument(X) is a desired state or a set of desired states which must be accomplished at the same time. The algorithm tries to accomplish one state at a time. If an operation for one state undoes another operation which was done for another state, it tries to re-accomplish the broken state from the current state. The algorithm does not yield an optimal solution to some cases. But, it is enough to show that a complex problem-solving method like planning can be accomplished by translating English descriptions into rules.

"Print" commands are inserted to show the atomic operations. The consistent rule of this command requires that there be no circularity like "A is on B, B is on C and C is on A." Also the programs for the commands, PUTON and PICKUP, are augmented by the planning program.

The following session is one with the operator, "make." All the rules added are listed in the Appendix C.

{Continued from the last session.}

{Rules for "make" operator are typed in.}

X is made if X is true.

The fact that X is on Y is made
if it is made that X is clear and Y is clear
and the robotarm is clear
and if it is printed that X is put on Y
and if X is put on Y.

The fact that the robotarm is clear is made
if X is on the robotarm
and if X is put down.

The fact that the X is clear is made
if Y is on X
and if it is made that Y is clear
and the robotarm is clear
and if it is printed that Y is put on the table
and if Y is put on the table.

X is made
if X is equal to list (X1 . R)
and X1 is made
and R is made
and X is checked.

X is checked if X is made.

Figure 3-4: Operator: Make

```
-X IS MADE IF X IS TRUE .

(Asserted: ((MAKE* X) IF (TRUE* X)))

{********  Other rules are not shown here *********}
{********        See Appendix C          *********}

{We will test the operator.  The next two commands
```

make the state of the Blocks World to be the initial state of Section 3.3.}

```
-PUT THE YELLOW BLOCK ON THE RED BLOCK .

(Deleted: (ON* (INST 7 PYRAMID COLOR YELLOW)
               (INST 1 TABLE)))
(Asserted: (ON* (INST 7 PYRAMID COLOR YELLOW)
                (INST 2 ROBOTARM)))
(Deleted: (ON* (INST 7 PYRAMID COLOR YELLOW)
               (INST 2 ROBOTARM)))
(Asserted: (ON* (INST 7 PYRAMID COLOR YELLOW)
                (INST 6 BLOCK SIZE LARGE COLOR RED)))


-PUT THE RED PYRAMID ON THE BLUE BLOCK .

(Deleted: (ON* (INST 5 BLOCK SUB PYRAMID COLOR RED)
               (INST 1 TABLE)))
(Asserted: (ON* (INST 5 BLOCK SUB PYRAMID COLOR RED)
                (INST 2 ROBOTARM)))
(Deleted: (ON* (INST 5 BLOCK SUB PYRAMID COLOR RED)
               (INST 2 ROBOTARM)))
(Asserted: (ON* (INST 5 BLOCK SUB PYRAMID COLOR RED)
                (INST 4 BLOCK SIZE SMALL COLOR BLUE)))
```

{We are ready to test the new operator.}
{The information about the asserted facts and deleted facts will be suppressed.}

```
-MAKE IT THAT THE BLUE BLOCK IS ON THE RED BLOCK
-                          UNDER THE YELLOW BLOCK .
```

{Because the blue block is under the red pyramid and the yellow block is on the red pyramid, we can not put the red block on the red block directly.}

```
(PUTON (INST 5 BLOCK SUB PYRAMID COLOR RED)
       (INST 1 TABLE))  {This clears the blue block.}

(PUTON (INST 7 PYRAMID COLOR YELLOW)
       (INST 1 TABLE))  {This clears the red block.}

(PUTON (INST 4 BLOCK SIZE SMALL COLOR BLUE)
       (INST 6 BLOCK SIZE LARGE COLOR RED))
```

{Another example.}

```
-MAKE IT THAT THE WHITE BLOCK IS ON THE LARGE RED BLOCK
-      AND THE YELLOW BLOCK IS ON THE WHITE BLOCK .

(PUTON (INST 4 BLOCK SIZE SMALL COLOR BLUE)
       (INST 1 TABLE))    {This clears the red block.}

(PUTON (INST 3 BLOCK SIZE LARGE COLOR WHITE)
       (INST 6 BLOCK SIZE LARGE COLOR RED))

(PUTON (INST 7 PYRAMID COLOR YELLOW)
       (INST 3 BLOCK SIZE LARGE COLOR WHITE))

{An example of inconsistent command of "make".}
{The next fact is asserted because we need one more
 large block.}


-THERE IS ANOTHER LARGE YELLOW BLOCK ON THE TABLE .

(Asserted: (INST 8 BLOCK SIZE LARGE COLOR YELLOW))
(Asserted: (ON* (INST 8 BLOCK SIZE LARGE COLOR YELLOW)
                (INST 1 TABLE)))

-MAKE IT THAT THE LARGE YELLOW BLOCK IS ON THE WHITE BLOCK
-      AND THE WHITE BLOCK IS ON THE LARGE RED BLOCK
-      AND THE LARGE RED BLOCK IS ON THE LARGE YELLOW BLOCK .


(Unacceptable Command :
 (MAKE* ((ON* (BLOCK DET THE SIZE LARGE COLOR YELLOW)
              (BLOCK DET THE COLOR WHITE))
         (ON* (BLOCK DET THE COLOR WHITE)
              (BLOCK DET THE SIZE LARGE COLOR RED))
         (ON* (BLOCK DET THE SIZE LARGE COLOR RED)
              (BLOCK DET THE SIZE LARGE COLOR YELLOW)))))

(Processing of input aborted)

{Because there is a circularity in the required states and they
 can not be made at the same time, the command is rejected.}

{The programs of PUTON and PICKUP are augmented.}

-A BLOCK IS PICKED UP
-   IF IT IS MADE THAT THE BLOCK IS ON THE ROBOTARM .

(Asserted: ((PICKUP (INST _VAR1))
```

```
            IF
            (RELMATCH (INST _VAR1 BLOCK) (INST _VAR1))
            (MAKE* (ON* (INST _VAR1) (INST 2 ROBOTARM)))))

-A BLOCK IS PUT ON X
-    IF IT IS MADE THAT THE BLOCK IS ON X .

(Asserted: ((PUTON (INST _VAR1) X)
            IF
            (RELMATCH (INST _VAR1 BLOCK) (INST _VAR1))
            (MAKE* (ON* (INST _VAR1) X))))
```

{By planning, the robotarm can pick up the white block on which the yellow pyramid is.}

```
-PICK THE WHITE BLOCK UP .
```

{Because there is a yellow block on the white block, the robotarm can not pick it up directly.}

```
(PUTON (INST 7 PYRAMID COLOR YELLOW)
       (INST 1 TABLE))  {It clears the white block.}

(PUTON (INST 3 BLOCK SIZE LARGE COLOR WHITE)
       (INST 2 ROBOTARM))


-PUT THE WHITE BLOCK ON THE LARGE YELLOW BLOCK .
```

{This can be done directly.}

```
(Deleted: (ON* (INST 3 BLOCK SIZE LARGE COLOR WHITE)
               (INST 2 ROBOTARM)))
(Asserted: (ON* (INST 3 BLOCK SIZE LARGE COLOR WHITE)
                (INST 8 BLOCK SIZE LARGE COLOR YELLOW)))


-PICK UP THE YELLOW PYRAMID .

(Deleted: (ON* (INST 7 PYRAMID COLOR YELLOW)
               (INST 1 TABLE)))
(Asserted: (ON* (INST 7 PYRAMID COLOR YELLOW)
                (INST 2 ROBOTARM)))
```

{Because currently the robotarm is holding a block, it has to put it down to move another block.}

```
-PUT THE BLUE BLOCK ON THE LARGE RED BLOCK .

(PUTDOWN (INST 7 PYRAMID COLOR YELLOW))

(PUTON (INST 4 BLOCK SIZE SMALL COLOR BLUE)
       (INST 6 BLOCK SIZE LARGE COLOR RED))

-PUT THE BLUE BLOCK ON THE TABLE .

(Deleted: (ON* (INST 4 BLOCK SIZE SMALL COLOR BLUE)
               (INST 6 BLOCK SIZE LARGE COLOR RED)))
(Asserted: (ON* (INST 4 BLOCK SIZE SMALL COLOR BLUE)
                (INST 2 ROBOTARM)))
(Deleted: (ON* (INST 4 BLOCK SIZE SMALL COLOR BLUE)
               (INST 2 ROBOTARM)))
(Asserted: (ON* (INST 4 BLOCK SIZE SMALL COLOR BLUE)
                (INST 1 TABLE)))
```

{*An example which does not yield the best plan.*}

```
-MAKE IT THAT THE RED BLOCK IS ON THE WHITE BLOCK AND
-             THE YELLOW BLOCK IS ON THE RED BLOCK .

(PUTON (INST 6 BLOCK SIZE LARGE COLOR RED)
       (INST 3 BLOCK SIZE LARGE COLOR WHITE))

(PUTON (INST 6 BLOCK SIZE LARGE COLOR RED) (INST 1 TABLE))
```

{*This is undoing the first movement above.*}

```
(PUTON (INST 3 BLOCK SIZE LARGE COLOR WHITE) (INST 1 TABLE))

(PUTON (INST 8 BLOCK SIZE LARGE COLOR YELLOW)
       (INST 6 BLOCK SIZE LARGE COLOR RED))

(PUTON (INST 8 BLOCK SIZE LARGE COLOR YELLOW) (INST 1 TABLE))
```

{*This is undoing the movement above.*}

```
(PUTON (INST 6 BLOCK SIZE LARGE COLOR RED)
       (INST 3 BLOCK SIZE LARGE COLOR WHITE))

(PUTON (INST 8 BLOCK SIZE LARGE COLOR YELLOW)
       (INST 6 BLOCK SIZE LARGE COLOR RED))
```

{*The planning mechanism used in this experiment is a simple one. More complicated planning scheme which yields a better solution can be found in* [Rich 83].}

{*End of record session.*}

Chapter 4

Discussion and Conclusion

In this research, we have developed a small grammar which can translate a subset of English into Horn Clauses, and experimented with it over the Blocks World domain. We have also developed a translator for the experimental domain which can accept several tasks in English, translate them into Horn Clauses by using the grammar, and interpret them as programs and data.

In this chapter, we will discuss some points which have not been covered yet, and then summarize the related works. In addition, with the conclusions drawn from this research, we will discuss the directions for further research.

4.1 Further Discussion

In the examples of the last chapter, we could see that some of the descriptions of problem-solving methods were not very natural. There are some reasons for this unnaturalness. One of the reasons is the use of variables such as X and Y. Even though, by using variables, we can state our intentions clearly, readability is reduced, since variables do not represent any semantic information. If we use a noun phrase as a variable, e.g. "a block", it will increase the readability. However, this will cause some pronominal references and will increase the ambiguity. Another cause which makes some rules unnatural is the restricted types of formats for rules. Virtually, only one form of rule, i.e. if-then structure, is allowed. To represent some ideas naturally, we may need more structures like "repeat" and "do while". Also, to represent the complex problem-solving methods easily, we need some kinds of data structure,

e.g. array or record. In HCPRVR, the dominant data structure is *list* and, in the examples of the last chapter, we used it frequently.

In the first chapter, we observed the parallelism between Procedural Logic and English which eases the process of translation of English to Procedural Logic or vice versa. This research is about only one direction of the translation. But, the other direction seems to be feasible, too. In some senses, the translation from Horn clause programs to English seems to be more promising for automatic documentation of programs. We can produce an English description of a Horn Clause program with the reverse procedure of the translation discussed in this paper. One merit of this translation is that the grammar for the subset can be small. The system can explain the behavior of the program by using only a few combining roles for simple English sentences.

As for the problem of "consistency", we discussed the problem only when we asserted something. But, we have to consider the problem also when we delete something. It indicates that DB is non-monotonic. If we delete some facts (or rules) without checking, it may make DB inconsistent. For example, if we delete the fact "a red block is on the table" from DB of Figure 4-1, the resulting DB will be inconsistent. Furthermore, we have the fact "a blue block is on the red block," which means that we have a block which is on a non-existent block.

```
A red block is on the table.
A blue block is on the red block.

((INST 1 BLOCK COLOR RED) IF)
((INST 2 TABLE DET THE) IF)
((INST 3 BLOCK COLOR BLUE) IF)
((ON* (INST 1) (INST 2)) IF)
((ON* (INST 3) (INST 1)) IF)
```

Figure 4-1: Deletion from DB

A solution to this problem would be to resort on the consistency rules

again. Rules can specify the conditions which must be met in order to delete something. In the example above, the following consistency rule can be used to prevent the inconsistency shown above.

It is consistent to delete the fact that X is on Y
if X is clear.

More general solutions to the problem of consistency of the non-monotonic DB are discussed in [Doyle 79] [McCarthy 80] [Petrie 84],and the rule-based control of DB state change is discussed in [Nicholas 78b].

4.2 Related Works

This work deals with the translation of English program descriptions into Horn Clause programs by using a small grammar, and can be categorized as *Natural Language Computation*(NLC) or *Natural Language Processing*(NLP).

There have been lots of pros and cons about programming in natural language, and it is still controversial. A summary of arguments can be found in [Petrick 76]. The claims of proponents are that many potential users do not want to learn and use a formal language, and that, in some applications, natural language is a better tool for communication with the computer than formal languages. Also they claim that the current state of art in NLC is sufficiently advanced and many remaining problems can be solved. But, opponents argue that natural language is so ambiguous and vague that NLC system can make errors because of misunderstanding the input in natural language. They also argue that it is very hard to provide a large subset of natural language for computation in the near future. Recently, [Rosenschein 83] has stressed the importance of studying NLC in current environment. That is, we have to study NLC, because NLC is getting practical utility in some domains, e.g. natural data base query language, and the study interlinks related fields such as linguistics, philosophy, and psychology. In A.I., the study of NLC accelerates the study of "explicit models of mind", which is one of this field's ultimate goals, and contributes to progress in knowledge representation and acquisition.

[Biermann 76] compared the efforts and time needed to make programs using a subset of English and a formal language(PL/C). In this work, experimentation was done by asking a group of students to program in both languages about problems of arrays. The result from the statistics of time measurements and also from interviewing students showed that natural language fared better than PL/C.

Among the reports on individual projects related with this work, [Van Baalen 83] developed a system which constructs a recursive LISP program out of English sentences. This system accepts declarative sentences which are programs and imperative sentences which are function calls. The declarative sentence uses the "be" verb, and the basic translation mechanism is rather syntactic. For example, a sentence of the following structure

noun + of + NP1 + qualifier + is + NP2

will be translated into the following LISP function.

```
(noun (LAMBDA (X)
       (COND ((NP1* X)
              (COND ((qualifier* X) NP2*))))))

where NP1* is a predicate function to check if X is NP1,
      qualifier* is predicate function to check if X
      has the required qualification,
      and NP2* is a function which returns the value
      which is specified by NP2.
```

The system also has a list of words and their corresponding LISP functions(user-defined and system-defined), and uses it to choose the right function for each word used in input.

About translating English to Procedural Logic, there have been two previous works. [Wang 83] proposed the original idea of this kind of translation, and experimented with the idea on a domain of family hierarchy with a small grammar. [Simmons 84b] extended the idea to the extent that the grammar understands comparative statements. Their works mainly dealt with facts and rules which can make inferences from the given facts.

4.3 Conclusion

In chapter 1, it was argued that Procedural Logic is not only an excellent programming language, but also a good symbolism for representing some human problem solving. In addition, by showing parallelism between Procedural Logic and a small subset of English, we suggested that the mechanical translation from English program descriptions to Horn Clause programs was worth study.

In the next chapter, we introduced a small grammar which can translate a subset of English sentences into Horn Clauses. The translation is mainly based on the syntactic structures of simple sentences from which sentences of complex structures can be built. Also, to make it easy to identify the references to several objects, noun phrases were stored as frames. This made changes in some literals. A program which is based on Relaxed Unification algorithm was introduced to compare frames effectively.

In Chapter 3, an interpreter was described which uses the grammar discussed in Chapter 2, and is able to accept English definitions of Blocks World tasks. We experimented with the interpreter on an example domain, Blocks World, to show its power. Also, in this chapter, we discussed the mechanism which maintains DB consistency by using Consistency Rules. Throughout many examples on the domain, we showed how several tasks such as programming, querying, describing the Blocks World, and commanding the robot could be done in English in a unified way.

As mentioned in the first chapter, this research was one of a series of experiments relating to the automatic translation between English and Horn Clauses. This research, even though limited to a small example domain, has provided the following conclusions.

- Procedural Logic provides a unified system to accomplish a wide variety of computations, and, because procedural descriptions in English can closely parallel Procedural Logic descriptions, translations between them are facilitated.
- The grammar used in the translation can be made small because the

translation procedure does not need any domain-specific or semantic knowledge. The semantics of each predicate name and domain-specific knowledge should be defined and given by users in the form of rules and assertions whenever they are needed.

4.4 Further Research

Because we experimented with translation only in a simple domain, we need more experiments on other domains to get insights into the problems in this kind of translation. And, in further experiments on other domains, the following two points should be considered seriously.

- The characteristics of English descriptions of problem-solving methods should be studied to provide a subset of English which is small but which offers a reasonable and comfortable way of describing the problem-solving methods.
- The practical domains, in which this kind of translation is helpful, should be sought. One potential practical domain is expert systems such as Emycin. It has been already known that Procedural Logic can provide a good tool for implementing expert systems [Clark 80], because many expert systems use rules as the main structure of domain knowledge, and Procedural Logic can provide several facilities which an expert system has to have. The remaining problem is to determine if it is possible to convert English rules of an expert system into Horn Clauses by the translation procedure described in this thesis.

It is too early to evaluate the usefulness of this research. The accurate evaluation will be possible only after we get more experiences from further research.

Appendix A.

HCPRVR Program for Simple Sentences

```
(AXIOMS
'(((SIMSNT X V R)
  < (SIMSNT1 X V R1) (OR* (EQ* R1 (AND . R)) (EQ* R1 R)))))

(AXIOMS
'(((SIMSNT1 (X THERE . Y) V R)
   <
   (VBE X)
   !
   (SIMSNT1 (THERE X . Y) V R))
  ((SIMSNT1 (THERE X . Y) (_PREP U V) R)
   <
   (VBE X)
   (NP Y _NF U (Z . R1))
   (PREP Z)
   !
   (NP R1 _NF2 V R)
   (PREPARC NF Z NF2 _PREP))
  ((SIMSNT1 (THERE X . Y) (THERE-IS V) R)
   <
   (VBE X)
   !
   (NP* Y _NF V R))
  ((SIMSNT1 (IS IT X . Y) (_PRED V) R)
   <
   ! (SIMSNT1 (IT IS X . Y) (_PRED V) R))
  ((SIMSNT1 (IT IS X . Y) (_PRED V) R)
   <
   (OR* (ADJ X X1 _IDIOM _PRED)
        (AND* (PASTP X Z) (VERB Z Z1 Z2 _PRED)))
   ! (NP Y CLAUSES V R))
  ((SIMSNT1 (X . Y) V R1)
   < (VERB X X1 X2 X3) ! (VP NIL NIL (X . Y) V R1))
  ((SIMSNT1 (X . Y) V R1)
   <
```

```
   (VBE X)
   (MOOD QUERY)
   !
   (NP Y _NF V1 R)
   (VP V1 _NF (X . R) V R1))
  ((SIMSNT1 X V R1) < (NP X _NF V1 R) (VP V1 _NF R V R1))))

(AXIOMS
'(((NP (LIST X . R) LIST X R) < !)
  ((NP (THE X THAT . Y) CLAUSES V R)
   <
   (NOUN X _NF X1)
   !
   (SIMCONJSNT Y V2 R)
   (NORM V2 V))
  ((NP (X THAT . Y) CLAUSES V R)
   <
   (OR* (NOUN X _NF X1) (PRON X X1))
   !
   (SIMCONJSNT Y V2 R)
   (NORM V2 V))
  ((NP (X . Y) PRON* X1 Y) < (PRON X X1))
  ((NP (X . Y) VAR* X Y) < (VAR* X) !)
  ((NP (THAT . X) CLAUSES V R)
   < ! (SIMCONJSNT X V1 R) (NORM V1 V))
  ((NP (TO . X) CLAUSES V R) < ! (VP NIL NIL X V R))
  ((NP (X . Y) _NF (U DET X . V) R)
   < (ART X) ! (NP1 Y _NF (U . V) R))
  ((NP X _NF V R) < (NP1 X _NF V R))))

(AXIOMS
'(((NP* (AND . X) _NF (U . V) R)
   < (NP X _NF U R1) (NP* R1 _NF V R))
  ((NP* NIL _NF NIL NIL) < !)
  ((NP* (AND . X) _NF NIL (AND . X)) < !)
  ((NP* X _NF Y R)
   <
   (NP X _NF U (AND . R1))
   (NP* (AND . R1) _NF V R)
   (NORM (U . V) Y))
  ((NP* X _NF V R) < (NP X _NF V R))))

(AXIOMS
'(((NP1 (X . Y) _NF (U S X . V) R)
   <
   (ADJ X _FA _IDIOM _PRED)
   (NP1 Y _NF (U . V) R)
   (FRAMEARC _FA _NF S))
  ((NP1 (X . Y) _NF (U S X . V) R)
```

```
   <
   (NOUN X _NF1 _PRED)
   (NP1 Y _NF2 (U . V) R)
   (FRAMEARC X U S))
  ((NP1 (X Y . Z) _NF1 (X _ARC V) R)
   <
   (NOUN X _NF1 _PRED)
   (PREP Y)
   (NP Z _NF2 V R)
   (PREPARC _NF1 Y _NF2 _ARC))
  ((NP1 (X . Y) _NF (X) Y) < (NOUN X _NF W1))))

(AXIOMS
'(((VP _NP _NF (X NOT . Z) (UNLESS+ V) R)
   < (VBE X) ! (VP _NP _NF (X . Z) V R))
  ((VP _NP _NF (X NOT . Z) (UNLESS+ V) R)
   < ! (VP _NP _NF Z Y R))
  ((VP NIL NIL (X . Y) (_PRED U V) R)
   <
   (VERB X _VF _IDIOM _PRED)
   (NEQ* _IDIOM NIL)
   (NP Y _NF U R1)
   (REMB _IDIOM R1 R2)
   (NP R2 _NF2 V R))
  ((VP NIL NIL (X . Y) (_PRED V) R)
   <
   (VERB X _VF _IDIOM _PRED)
   (REMB _IDIOM Y R1)
   (NP R1 _NF V R))
  ((VP NIL NIL (X . Y) (_PRED V) R)
   <
   !
   (VERB X _VF _IDIOM _PRED)
   (NP Y _NF V R1)
   (REMB _IDIOM R1 R))
  ((VP _NP _NF (X Y . R) (_ARC _NP Y) R)
   <
   (VBE X)
   (ADJ Y _FA NIL _PRED)
   (FRAMEARC _FA _NF _ARC))
  ((VP _NP _NF (X Y . R) (_PRED _NP) R)
   < (VBE X) (ADJ Y _FA NIL _PRED))
  ((VP _NP _NF (X Y . Z) (_PRED _NP _NP2) R)
   <
   (VBE X)
   (ADJ Y _FA _IDIOM _PRED)
   (REMB _IDIOM Z R1)
   (NP R1 _NF2 _NP2 R))
  ((VP _NP _NF1 (X Y . Z) (_PRED _NP _NP2) R)
```

```
   <
   (VBE X)
   (PREP Y)
   !
   (NP Z _NF2 _NP2 R)
   (PREPARC _NF1 Y _NF2 _PRED))
  ((VP (V . V1) _NF1 (X . Y) (_ARC (V . V1) U) R)
   <
   (VBE X)
   (NP Y _NF2 (U . U1) R)
   (FRAMEARC V U _ARC))
  ((VP _NP _NF1 (X . Y) (_PRED _NP W) R)
   <
   (VBE X)
   (NP Y _NF2 V R)
   (REMDET V (U OF* W))
   (NOUN U X1 _PRED))
  ((VP _NP _NF1 (X . Y) (_PRED _NP) R)
   <
   (VBE X)
   (NP Y _NF2 (U . V) R)
   (NOUN U X1 _PRED))
  ((VP _NP _NF (X Y . Z) (_PRED _NP U) R)
   <
   (VBE X)
   (PASTP Y Y1)
   (VERB Y1 _VF _IDIOM _PRED)
   (REMB _IDIOM Z R1)
   (NP R1 _NF2 U R))
  ((VP _NP _NF (X Y . Z) (_PRED _NP) R)
   <
   (VBE X)
   (PASTP Y Y1)
   (VERB Y1 _VF _IDIOM _PRED)
   (REMB _IDIOM Z R))))

(AXIOMS
'(((OR*) < ! (FAIL))
  ((OR* X . Y) < X !)
  ((OR* X . Y) < (OR* . Y))))

(AXIOMS
'(((NORM (X) X) < !) ((NORM X X))))

(AXIOMS
'(((REMB NIL X X)) ((REMB (X . Y) (X . Z) R) < (REMB Y Z R))))

(AXIOMS
'(((NOUN LIST TYPE LIST*))
```

```
  ((NOUN BLOCK OBJ BLOCK*))
  ((NOUN CUBE OBJ CUBE*))
  ((NOUN MEMBER STATE MEMBER+))
  ((NOUN PYRAMID OBJ PYRAMID*))
  ((NOUN ROBOTARM OBJ ROBOTARM*))
  ((NOUN RULES TEXT NIL))
  ((NOUN TABLE OBJ TABLE*))
  ((NOUN FACT TEXT FACT*))
  ((NOUN INFERENCE TEXT INFERENCE*))))

(AXIOMS
'(((VERB ASSERT ASSERT NIL ASSERT*))
  ((VERB DELETE DELETE NIL DELETE))
  ((VERB DUMP PRINT NIL DUMP*))
  ((VERB MAKE CREATE NIL MAKE*))
  ((VERB PICK MOVE (UP) PICKUP))
  ((VERB PRINT PRINT NIL RPRINT))
  ((VERB PUT MOVE (DOWN) PUTDOWN))
  ((VERB PUT MOVE (ON) PUTON))
  ((VERB CHECK TEST NIL CHECK*))))

(AXIOMS
'(((VBE IS)) ((VBE ARE))))

(AXIOMS
'(((PRON NIL NIL)) ((PRON IT IT))))

(AXIOMS
'(((ADJ LARGER COMPAT (THAN) LARGER*))
  ((ADJ YELLOW COLOR NIL YELLOW*))
  ((ADJ BLUE COLOR NIL BLUE*))
  ((ADJ CLEAR PHYSTATE NIL CLEAR*))
  ((ADJ CONSISTENT STATE NIL CONSISTENT*))
  ((ADJ EQUAL RELATION (TO) EQ*))
  ((ADJ LARGE SIZE NIL LARGE*))
  ((ADJ RED COLOR NIL RED*))
  ((ADJ SMALL SIZE NIL SMALL*))
  ((ADJ WHITE COLOR NIL WHITE*))
  ((ADJ FALSE VALUE NIL FALSE*))
  ((ADJ SUPPORTABLE REL NIL SUPPORTABLE*))
  ((ADJ TRUE VALUE NIL TRUE*))))

(AXIOMS
'(((ART A)) ((ART AN)) ((ART ANOTHER)) ((ART THE))))

(AXIOMS
'(((PASTP ASSERTED ASSERT))
  ((PASTP DELETED DELETE))
  ((PASTP MADE MAKE))
```

```
  ((PASTP PICKED PICK))
  ((PASTP PUT PUT))
  ((PASTP CHECKED CHECK))
  ((PASTP PRINTED PRINT))))

(AXIOMS
'(((PREP ON)) ((PREP OF)) ((PREP ABOVE)) ((PREP UNDER))))

(AXIOMS
'(((FRAMEARC BLOCK PYRAMID SUB))
  ((FRAMEARC VAR* PYRAMID SUB))
  ((FRAMEARC COLOR X COLOR))
  ((FRAMEARC SIZE X SIZE))
  ((FRAMEARC PRON* PYRAMID SUB))))

(AXIOMS
'(((PREPARC X ON Y ON*))
  ((PREPARC X OF Y OF*))
  ((PREPARC X ABOVE Y ABOVE*))
  ((PREPARC X UNDER Y UNDER*))))
```

Appendix B.

HCTRANS

B.1 ELISP programs

```
(DE HCTRANS NIL
 (PROG (SENT SAVENUM)
       (SETQ QFLAG)
       (SETQ TFLAG)
       (SETQ LIMIT 3000.)
       (VARIABLES SOMETHING ANYTHING)
     A (SETQ VARLIST NIL)
       (SETQ SAVENUM OLDINSTNUM)
       (SETQ IVARNUM 0.)
       (TERPRI)
       (SETQ SENT (VARCHECK (READ-SENT)))
       (COND [(EQ MOOD '?)
              (ASSERT (MOOD QUERY) TEMP)
              (TRY (LIST 'QUERY SENT))
              (DELETE* (MOOD QUERY) TEMP)]
             [(RULE? SENT)
              (ASSERT (MOOD RULE) TEMP)
              (TRY (LIST 'RULE SENT))
              (DELETE* (MOOD RULE) TEMP)]
             [(COMMAND? SENT)
              (ASSERT (MOOD COMMAND) TEMP)
              (TRY (LIST 'COMMAND SENT))
              (DELETE* (MOOD COMMAND) TEMP)]
             [T (ASSERT (MOOD FACT) TEMP)
                (TRY (LIST 'FACT SENT))
                (DELETE* (MOOD FACT) TEMP)])
       (COND [(NULL VAL)
              (MSG T "(Processing of input aborted)" T)
              (CLEAR TEMP)
              (PUTPROP 'TEMP NIL 'AXIOMS)
              (SETQ OLDINSTNUM SAVENUM)])
       (TERPRI)
       (GO A)))
```

```
(DE READ-SENT NIL
 (PROG (SENT BUF TYPE)
       (TERPRI)
       (PROMPT 45.)
     A (SETQ BUF (READL))
       (COND [(AND [NOT (ATOM (CAR BUF))]
                   [NULL (CDR BUF)])
              (PRINT (EVAL (CAR BUF)))
              (TERPRI)
              (GO A)])
       (SETQ MOOD (CAR (LAST BUF)))
       (COND [(MEMQ MOOD (LIST '/. '?))
              (SETQ BUF (REVERSE (CDR (REVERSE BUF))))
              (PROMPT 42.)
              (RETURN (APPEND SENT BUF))]
             [(SETQ SENT (APPEND SENT BUF)) (GO A)])))

(DE VARCHECK (SNT)
 (COND [(NULL SNT) NIL]
       [(VAR SNT) (CAR (VARCHECK (LIST SNT)))]
       [(NOT (ATOM (CAR SNT)))
        (CONS (CONS (VARCHECK (CAAR SNT))
                    (VARCHECK (CDAR SNT)))
              (VARCHECK (CDR SNT)))]
       [(VAR (CAR SNT))
        (SETQ JJ (READLIST (APPEND '(V A R -)
                 (EXPLODE (CAR SNT)))))
        (AXIOMS (LIST (LIST (LIST 'VAR* JJ) 'LEXI)))
        (COND [(MEMBER JJ VARLIST) NIL]
              [T (SETQ VARLIST (CONS JJ VARLIST))])
        (CONS JJ (VARCHECK (CDR SNT)))]
       [T (CONS (CAR SNT) (VARCHECK (CDR SNT)))]))

(DE RULE? (SENT)
 (COND [(NOT (NULL VARLIST)) T]
       [(OR [MEMBER 'IF SENT] [MEMBER 'UNLESS SENT]) T]
       [(HEAD '(IT IS ALWAYS THE CASE) SENT) T]
       [(HEAD '(IT IS NOT THE CASE) SENT) T]
       [T NIL]))

(DE HEAD (X Y)
 (COND [(NULL X) T]
       [(EQ (CAR X) (CAR Y)) (HEAD (CDR X) (CDR Y))]
       [T NIL]))

(DE COMMAND? (SENT)
 (COND [(TRY (LIST 'VERB (CAR SENT) 'X 'Y 'Z)) T] [T NIL]))
```

```
{The following functions used in HCPRVR programs}

(DF VAR? (X) (OR [VAR (CAR X)] [HEAD '(V A R) (EXPLODE (CAR X))]))

(DE GETAVAR (V N)
 (PROG (VAR VAR1)
       (SETQ VAR1 (READLIST (CONS '_ (APPEND (EXPLODE V)
                                             (EXPLODE N)))))
       (PUTPROP VAR1 T 'VARIABLE)
       (SETQ VAR
        (READLIST (APPEND '(V A R - _) (APPEND (EXPLODE V)
                                               (EXPLODE N)))))
       (SETQ VARLIST (CONS VAR VARLIST))
       (RETURN VAR)))

(DF MAKEAVAR (X) (READLIST (CDDDDR (EXPLODE (CAR X)))))

(DE PAIRVARS (LST)
 (MAPCAR (F:L (XX) (CONS XX (READLIST (CDDDDR (EXPLODE XX)))))
                   LST))

(DF NOCONSRULE (X)
 (PROG (AXIOMS)
       (SETQ AXIOMS (GET 'CONSISTENT* 'AXIOMS))
       (SETQ X (CAR X))
    A (COND [(NULL AXIOMS) (RETURN T)]
            [(EQ (CAADR (CAAR AXIOMS)) X) (RETURN)])
       (SETQ AXIOMS (CDR AXIOMS))
       (GO A)))
```

B.2 HCPRVR programs

```
(AXIOMS
'(((ADDFRAME (INST N) _ARC Y)
   <
   (INST N X . X1)
   (DELETE (INST N X . X1))
   (ASSERT* (INST N X _ARC Y . X1)))))

(AXIOMS
'(((ALLTRUE* NIL) < !)
  ((ALLTRUE* (X . Y)) < X (ALLTRUE* Y) !)))

(AXIOMS
```

```
'(((AND*)) ((AND* X . Y) < X (AND* . Y))))

(AXIOMS
'(((APPEND* NIL X X))
  ((APPEND* (X . Y) Z (X . V1)) < (APPEND* Y Z V1))))

(AXIOMS
'(((ART A)) ((ART AN)) ((ART ANOTHER)) ((ART THE))))

(AXIOMS
'(((ASKS ((U . V) . W))
   < (ALLTRUE* ((U . V) . W)) (RPRINT* (YES (U . V))))
  ((ASKS X) < X (RPRINT* (YES X)) !)
  ((ASKS X) < (RPRINT (NO)))))

(AXIOMS
'(((ASS-TEMP)
   <
   (TEMP X)
   (DELETE* (TEMP X))
   (DELETE* X TEMP)
   (ASSERT* X)
   (FAIL))
  ((ASS-TEMP))))

(AXIOMS
'(((ASSERT* X) < (RPRINT* (Asserted: X)) (ASSERT X IF))))

(AXIOMS
'(((ASSERT-FACT (THERE-IS X)) < (ASS-TEMP) (REFERENCES X))
  ((ASSERT-FACT (X Y Z))
   < (FRAMEARC U V X) (REFERENCES (Z Y)) (ADDFRAME Y X Z))
  ((ASSERT-FACT X)
   <
   (ASS-TEMP)
   (REVERSE* X X1)
   (REFERENCES X1)
   (SETV* Y (SUBLIS (PAIRVARS VARLIST) 'X))
   (ASSERT* Y))))

(AXIOMS
'(((ASSERT-RULE X)
   <
   (RPRINT* (ASSERTED: X))
   (SETV* Y (SUBLIS (PAIRVARS VARLIST) 'X))
   (ASSERTB . Y)
   !)))

(AXIOMS
```

```
'(((CHANGEVAR (U DET U1 . U2)
        (RELMATCH (INST Z1 U . U2) (INST Z1)) NIL)
   <
   (MEMBER* U1 (A AN ANOTHER))
   !
   (NEWIVAR N)
   (SETV* Z1 (GETAVAR 'VAR N))
   (TEMP-ASSERT (INST Z1 U . U2)))
  ((CHANGEVAR (X . Y) V W) < (CHANGEVAR1 (X . Y) V W))
  ((CHANGEVAR X X NIL))))

(AXIOMS
'(((CHANGEVAR1 (U DET U1 . U2)
               (INST Z1)
               ((RELMATCH (INST Z1 U . U2) (INST Z1))))
   <
   (MEMBER* U1 (A AN ANOTHER))
   (NEWIVAR N)
   (SETV* Z1 (GETAVAR 'VAR N))
   (TEMP-ASSERT (INST Z1 U . U2)))
  ((CHANGEVAR1 (X . X1) Y Z) < (CHANGEVAR2 (X . X1) Y Z))
  ((CHANGEVAR1 X X NIL))))

(AXIOMS
'(((CHANGEVAR2 NIL NIL NIL))
  ((CHANGEVAR2 (X . Y) (U . V) W)
   <
   (CHANGEVAR1 X U W1)
   (CHANGEVAR2 Y V W2)
   (APPEND* W1 W2 W))
  ((CHANGEVAR2 X X NIL))))

(AXIOMS
'(((CHECK* X) IF (MAKE* X))))

(AXIOMS
'(((CHK-OBJS (X . Y) U) < (NOUN X X1 X2) ! (CHKIN (X . Y) U))
  ((CHK-OBJS (X . Y) V) < ! (CHK-OBJS1 (X . Y) V))
  ((CHK-OBJS X Y) < (PRON X X1) ! (CHKIN X Y))
  ((CHK-OBJS X X))))

(AXIOMS
'(((CHK-OBJS1 NIL NIL) < !)
  ((CHK-OBJS1 (X . Y) (U . V)) < ! (CHK-OBJS X U) (CHK-OBJS1 Y V))
  ((CHK-OBJS1 X X))))

(AXIOMS
'(((CHKIN X (INST N))
   <
```

```
   (REVERSE* X ((R1 . R2) X1 . R))
   !
   (CHKIN (R1 . R2) W)
   (REVERSE* R X2)
   (REMDET X2 X3)
   (RELMATCH (INST N . X3) (INST N))
   (X1 (INST N) W))
  ((CHKIN X (INST N)) < (PRON X X1) (INST N . Y))
  ((CHKIN (W DET THE . R1) (INST N))
   < (RELMATCH (INST N W . R1) (INST N)))
  ((CHKIN (W DET THE . R1) (INST N))
   <
   (RELMATCH (INST N W . R1) (INST N))
   !
   (FAIL))
  ((CHKIN (W DET Y . R1) (INST N))
   <
   (MOOD FACT)
   (MEMBER* Y (A AN ANOTHER))
   (NEWINSTNUM N)
   !
   (TEMP-ASSERT (INST N W . R1)))
  ((CHKIN (W DET THE . R1) (INST N))
   <
   (OR* (MOOD FACT) (MOOD RULE))
   (NEWINSTNUM N)
   !
   (TEMP-ASSERT (INST N W DET THE . R1)))
  ((CHKIN (X . X1) (INST N))
   <
   (UNLESS* (MEMBER* DET (X . X1)))
   (OR* (MOOD FACT) (MOOD RULE))
   (NEWINSTNUM N)
   !
   (TEMP-ASSERT (INST N X DET NIL . X1)))
  ((CHKIN (X DET Y . R) (X DET Y . R))
   <
   (OR* (MOOD RULE) (MOOD QUERY))
   (MEMBER* Y (A AN ANOTHER)))
  ((CHKIN X NIL)
   < (RPRINT (Uidentifiable Object: X)) ! (FAIL))))

(AXIOMS
'(((CLEAR-TEMP)
   < (TEMP X) (DELETE* X TEMP) (DELETE* (TEMP X)) (FAIL))
  ((CLEAR-TEMP))))

(AXIOMS
'(((COMMAND X) < (SNT X (Y IF)) (MULTI-CMDS Y))))
```

```
(AXIOMS
'(((CONJSNT (THEN . Y) NIL Y))
  ((CONJSNT (AND IF . Y) V2 R1)
   <
   !
   (SIMCONJSNT Y V R)
   (CONJSNT R V1 R1)
   (APPEND* V V1 V2))
  ((CONJSNT (AND UNLESS . Y) ((UNLESS+ . V) . V1) R1)
   <
   (SIMCONJSNT Y V R)
   (CONJSNT R V1 R1))
  ((CONJSNT X NIL X))))

(AXIOMS
'(((CONSISTENT (U . V)) < (NOCONSRULE U) !)
  ((CONSISTENT X) < (CONSISTENT* X))))

(AXIOMS
'(((DELETE X) < X (RPRINT* (Deleted: X)) (DELETE* X IF))))

(AXIOMS
'(((EQ* X X))))

(AXIOMS
'(((EXECUTE X) < X (REVERSE* X Y) (REFERENCES Y))))

(AXIOMS
'(((FACT X) < (SNT X (Y IF)) (MULTI-FACTS Y))))

(AXIOMS
'(((LIST* ((X . Y) . Z)))))

(AXIOMS
'(((MEMBER* X (X . Y)) < !)
  ((MEMBER* X (Y . Z)) < (MEMBER* X Z))))

(AXIOMS
'(((MEMPAIR (X Y) (X Y . Z)))
  ((MEMPAIR (X Y) (U V . Z)) < (MEMPAIR (X Y) Z))))

(AXIOMS
'(((MULTI-CMDS NIL) < !)
  ((MULTI-CMDS ((U . V) . Y))
   < ! (SING-CMD (U . V)) ! (MULTI-CMD Y))
  ((MULTI-CMDS X) < (SING-CMD X))))

(AXIOMS
```

```
'(((MULTI-FACTS NIL) < !)
  ((MULTI-FACTS ((U . V) . Y))
   < ! (SING-FACT (U . V)) ! (MULTI-FACTS Y))
  ((MULTI-FACTS X) < (SING-FACT X))))

(AXIOMS
'(((NEQ* X Y) < (UNLESS* (EQ* X Y)))))

(AXIOMS
'(((NEWINSTNUM N)
  < (SETV X (ADD1 OLDINSTNUM))
    (SETQ OLDINSTNUM X) (EQ* N X) !)))

(AXIOMS
'(((NEWIVAR N)
  <
  (SETV X (ADD1 IVARNUM)) (SETQ IVARNUM X) (EQ* N X) !)))

(AXIOMS
'(((OR*) < ! (FAIL))
  ((OR* X . Y) < X !) ((OR* X . Y) < (OR* . Y))))

(AXIOMS
'(((POINTERS X Y) < (VAR? X) (UNVAR X Y))
  ((POINTERS (INST N) (INST Y)) < (VAR? N) ! (UNVAR N Y))
  ((POINTERS (INST N) (INST N . X)) < (INST N . X) !)
  ((POINTERS (X . Y) (U . V))
   < (POINTERS X U) (POINTERS Y V) !)
  ((POINTERS X X) < !)))

(AXIOMS
'(((QUERY X)
   <
   (SNT X (Y IF))
   (TRANS-OBJS Y Z)
   (SETV* W (SUBLIS (PAIRVARS VARLIST) 'Z))
   (ASKS W))))

(AXIOMS
'(((REFERENCES (INST N))
   <
   (INST N . X)
   (DELETE* (INST N . X) IF)
   (ASSERT (INST N . X) IF))
  ((REFERENCES (X . Y))
   < ! (REFERENCES X) (REFERENCES Y))
  ((REFERENCES X))))

(AXIOMS
```

```
'(((RELMATCH (INST . X) (INST . X)))
  ((RELMATCH (INST N X . X1) (INST N X . Y))
   <
   (RELUNIF (INST N X . X1) (INST N X . Y)))
  ((RELMATCH (INST N) (INST N X . Y))
   <
   (INST N . Z)
   (RELMATCH (INST N . Z) (INST N X . Y)))
  ((RELMATCH (INST N X . Y) (INST N))
   <
   (INST N . Z)
   (RELMATCH (INST N X . Y) (INST N . Z)))
  ((RELMATCH (INST N X . X1) (INST N Y . Y1))
   <
   (NOUN X _NF1 U1)
   (NOUN Y _NF2 U2)
   (FRAMEARC X Y SUB)
   (RELUNIF (INST N Y . X1) (INST N Y . Y1)))
  ((RELMATCH (INST N Y . X1) (INST N X . Z))
   <
   (NOUN X _NF1 U1)
   (NOUN Y _NF2 U2)
   (FRAMEARC X Y S)
   (RELUNIF (INST N X S Y . X1) (INST N X . Z)))))

(AXIOMS
'(((RELUNIF (INST N X) (INST N X . Y)))
  ((RELUNIF (INST N X U V . R) (INST N X . Y))
   <
   (MEMPAIR (U V) Y)
   (RELUNIF (INST N X . R) (INST N X . Y)))))

(AXIOMS
'(((REMDET (X DET Y . R1) (X . R1))) ((REMDET X X))))

(AXIOMS
'(((REVERSE* NIL NIL))
  ((REVERSE* (X . Y) V)
   < (REVERSE* Y Y1) (APPEND* Y1 (X) V))))

(AXIOMS
'(((RPRINT* X) < (POINTERS X Y) (RPRINT Y) ')))

(AXIOMS
'(((RREAD) X) < (SETV* X (READ)))))

(AXIOMS
'(((RULE X) < (SNT X Y) (TRANS-OBJS Y Z) (ASSERT-RULE Z))))
```

```
(AXIOMS
'(((SETV* X Y) < (SETV Z Y) (EQ* X Z) !)))

(AXIOMS
'(((SIMCONJSNT NIL NIL NIL) < !)
  ((SIMCONJSNT (IF . X) NIL (AND IF . X)))
  ((SIMCONJSNT (UNLESS . X) NIL (AND UNLESS . X)))
  ((SIMCONJSNT (THEN . X) NIL (THEN . X)))
  ((SIMCONJSNT X (U . V) R)
   < (SIMSNT X U R1) (SIMCONJSNT R1 V R))
  ((SIMCONJSNT X NIL X))))

(AXIOMS
'(((SING-CMD X) < (CHK-OBJS X Y) (CONSISTENT Y) (EXECUTE Y))
  ((SING-CMD X)
   < (RPRINT* (Unacceptable Command : X)) ! (FAIL))))

(AXIOMS
'(((SING-FACT X)
   < (CHK-OBJS X Y) (CONSISTENT Y) (ASSERT-FACT Y))
  ((SING-FACT X)
   < (RPRINT* (Unacceptable Fact: X)) ! (FAIL))))

(AXIOMS
'(((SNT (IT IS ALWAYS THE CASE THAT . X) (V IF !))
   < ! (SIMSNT X V NIL))
  ((SNT (IT IS NOT THE CASE THAT . X) (V IF ! (FAIL)))
   <
   !
   (SIMSNT X V NIL))
  ((SNT (IF . X) (V1 IF . V))
   < ! (CONJSNT (AND IF . X) V R) (SIMSNT R V1 NIL))
  ((SNT (UNLESS . X) (V1 IF . V))
   <
   !
   (CONJSNT (AND UNLESS . X) V R)
   !
   (SIMSNT R V1 NIL))
  ((SNT X (U IF . V1))
   < (SIMCONJSNT X V R) (NORM V U) (CONJSNT R V1 NIL))))

(AXIOMS
'(((TEMP-ASSERT X) < (ASSERTB (TEMP X)) (ASSERTB X TEMP))))

(AXIOMS
'(((TRANS-OBJS X Y) < (MOOD RULE) (TRANSFORM X Y) (CLEAR-TEMP))
  ((TRANS-OBJS X Y)
   < (MOOD QUERY) (TRANSFORM1 X Y) (CLEAR-TEMP))))
```

```
(AXIOMS
'(((TRANSFORM (X Y . Z) (U Y . V))
   <
   (TRANSFORM1 X (U . R))
   (TRANSFORMN Z U1)
   (APPEND* R U1 V))))

(AXIOMS
'(((TRANSFORM1 (FAIL) ((FAIL))))
  ((TRANSFORM1 ! (!)))
  ((TRANSFORM1 (UNLESS+ . X) ((UNLESS+ . Y)))
   < ! (TRANSFORMN X Y))
  ((TRANSFORM1 (THERE-IS ((U . V) . Y)) Z)
   < (TRANSFORMN ((U . V) . Y) Z))
  ((TRANSFORM1 (THERE-IS X) Y) < (TRANSFORM1 X Y))
  ((TRANSFORM1 (X Y Z) ((RELMATCH (INST V1 V2 X Z) Y1)))
   <
   (FRAMEARC X1 X2 X)
   (NEWIVAR N)
   (SETV* V1 (GETAVAR 'VAR N))
   (NEWIVAR N1)
   (SETV* V2 (GETAVAR 'VAR N1))
   (CHK-OBJS Y Y1))
  ((TRANSFORM1 X (X2 . V))
   <
   (CHANGEVAR X X1 V)
   !
   (CHK-OBJS X1 X2)
   (REVERSE* X2 Y)
   !
   (REFERENCES Y))))

(AXIOMS
'(((TRANSFORMN NIL NIL))
  ((TRANSFORMN (X . Y) W)
   <
   (TRANSFORM1 X U) (TRANSFORMN Y V) (APPEND* U V W))))

(AXIOMS
'(((TRUE* NIL)) ((TRUE* X) < X)
  ((TRUE* ((X . X1) . Y)) < (X . X1) (TRUE* Y))))

(AXIOMS
'(((UNLESS* X) < X ! (FAIL)) ((UNLESS* X))))

(AXIOMS
'(((UNLESS+) < ! (FAIL))
  ((UNLESS+ X . Y) < X ! (UNLESS+ . Y))
  ((UNLESS+ X . Y) < !)))
```

```
(AXIOMS
'(((UNVAR U V) < (SETV* V (MAKEAVAR U)))))
```

Appendix C.

Planning

MAKE*

```
X is made if X is true.

The fact that X is on Y is made
        if it is made that X is clear and Y is clear
             and the robotarm is clear
    and if it is printed that X is put on Y
    and if X is put on Y.

The fact that the robotarm is clear is made
        if X is on the robotarm
    and if X is put down.

The fact that the X is clear is made
        if Y is on X
    and if it is made that Y is clear
           and the robotarm is clear
    and if it is printed that Y is put on the table
    and if Y is put on the table.

 X is made
        if X is equal to list (X1 . R)
    and if X1 is made
    and if R is made
    and if X is checked.

 ((MAKE* X) IF (TRUE* X))
 ((MAKE* (ON* X Y))
  IF
  (MAKE* ((CLEAR* X) (CLEAR* Y) (CLEAR* (INST 2 ROBOTARM))))
  (RPRINT* (PUTON X Y))
  (PUTON X Y))
 ((MAKE* (CLEAR* (INST 2 ROBOTARM)))
  IF
```

```
 (ON* X (INST 2 ROBOTARM))
 (RPRINT* (PUTDOWN X))
 (PUTDOWN X))
((MAKE* (CLEAR* X))
 IF
 (ON* Y X)
 (MAKE* (CLEAR* Y))
 (RPRINT* (PUTON Y (INST 1 TABLE)))
 (PUTON Y (INST 1 TABLE)))
((MAKE* X)
 IF
 (EQ* X (X1 . R)) (MAKE* X1) (MAKE* R) (CHECK* X))
```

CHECK*

X is checked if X is made.

```
((CHECK* X) IF (MAKE* X))
```

MEMBER+

It is always the case that X is a member of list (X . Y).

X is a member of list (Y . Z) if X is a member of Z.

```
((MEMBER+ X (X . Y)) IF !)
((MEMBER+ X (Y . Z)) IF (MEMBER+ X Z))
```

INFERENCE*

The fact that U is above V is an inference of X
if the fact that U is on W is a member of X
and the fact that W is on V is another member of X.

The fact that U is above V is an inference of X
if the fact that U is on W is a member of X
and the fact that W is above V is an inference of X.

```
((INFERENCE* (ABOVE* U V) X)
 IF
 (MEMBER+ (ON* U W) X) (MEMBER+ (ON* W V) X))
((INFERENCE* (ABOVE* U V) X)
 IF
 (MEMBER+ (ON* U W) X)
 (INFERENCE* (ABOVE* W V) X))
```

CONSISTENT*

It is consistent to make X

unless U is above U is an inference of X.

```
((CONSISTENT* (MAKE* X))
 IF
 (UNLESS+ (INFERENCE* (ABOVE* U U) X)))
```

Bibliography

[Biermann 76] Biermann, A. W.
Approaches to Automatic Programming.
In Morris Rubinoff and Marshall C. Yovits (editor), *Advances in Computer, vol 15*, . Academic Press, New York, 1976.

[Chester 80] Chester, D.
Using HCPRVR.
Technical Report, Department of Computer Science, University of Texas, 1980.

[Clark 78] Clark, K. L.
Negations as Failure.
In Gallaire, H., and Minker, J. (editor), *Logic and Data Bases*, . Plenum Press, New York, 1978.

[Clark 80] Clark, K. L., McCabe, F. G.
PROLOG: A Language for Implementing Expert Systems.
In Hayes, J. and Michie, D. J. (editor), *Machine Intelligence 10*, . Ellis and Horwood, 1980.

[Colmerauer 78] Colmeraruer, A.
Metamorphosis Grammars.
In Bolc, L. (editor), *Natural Language Communication with Computers*, . Springer-Verlag, New York, 1978.

[Doyle 79] Doyle, J.
A Truth Maintenance System.
Artificial Intelligence 12(3), 1979.

[Ehrlich 81] Ehrich, K.
Search and Inference Strategies in Pronoun Resolution: An Experimental Study .
In *Proceedings of the Conference: 19th Annual Meeting of the Association for Computational Linguistics.* Association for Computational Linguistics, 1981.

[Fikes 71] Fikes, R. E. and Nilsson, N. J.
Strips: A New Approach to the Application for Theorem Proving to Problem Solving.
Artificial Intelligence 2, 1971.

[Jaffer 83] Jaffer, J., Lassez, J., Lloyd, J.
Completeness of the Negation as Failure Rule.
In *Proceedings of the 8th International Conference on Artificial Intelligence.* Karlsruhe, West Germany, August, 1983.

[Kowalski 78] Kowalski, R.
Logic for Data Description.
In Gallaire, H., and Minker, J. (editor), *Logic and Data Bases*, . Plenum Press, New York, 1978.

[Kowalski 79] Kowalski, R.
Logics for Problem Solving.
North Holand, Limerick, Ireland, 1979.

[LeVine 80] LeVine, S. H.
Questioning English Text with Clausal Logic.
Master's thesis, University of Texas, December, 1980.

[McCarthy 80] McCarthy, J.
Circumscription - A Form of Non-Monotonic Reasoning.
Artificial Intelligence 13(1,2), 1980.

[Nicholas 78a] Nicholas, J. M. and Gallaire, H.
Data Base: Theory vs. Interpretation.
In Gallaire, H., and Minker, J. (editor), *Logic and Data Bases*, . Plenum Press, New York, 1978.

[Nicholas 78b] Nicholas, J. M. and Yazdanian, K.
Integrity Checking in Deductive Data Bases.
In Gallaire, H., and Minker, J. (editor), *Logic and Data Bases*, . Plenum Press, New York, 1978.

[Petrick 76] Petrick, S. R.
On Natural Language Based Computer Systems.
IBM Journal of Research Development , July, 1976.

[Petrie 84] Petrie, C.
The Problem of Consistency in Non-Monotonic Logics.
1984.
Submitted to AAAI-84.

[Rich 83] Rich, E.
Artificial Intelligence.
McGraw Hill, New York, 1983.

[Rosenschein 83] Rosenchein, S.
Natural Language Processing:Crucial for Computational Theories of Cognition.
In *Proceedings of the 8th International Conference on Artificial Intelligence.* Karlsruhe, West Germany, August, 1983.

[Simmons 83] Simmons, R. F.
A Text Knowledge Base for the AI Handbook.
Technical Report TR-83-24, Department of Computer Science, University of Texas, 1983.

[Simmons 84a] Simmons, R. F.
Computations from the English.
Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[Simmons 84b] Simmons, R. F.
Simple.Sim.
1984.
Lecture Material.

[Tseng 83] Tseng, S. H.
Questioning English Test with Clausal Logic: Case Studies of Three Texts.
Master's thesis, University of Texas, December, 1983.

[Van Baalen 83] Van Baalen, J. .
Using Natural Language to Solve Recursive Programming Problems .
Technical Report No. 2, Department of Computer Science, University of Wyoming, August, 1983.

[Wang 83] Wang, T. C.
Understanding a Subset of English.
1983.
Draft.

[Warren 77] Warren, D. H. D, Pereira, L. M., and Pereira, F.
PROLOG - The Language and its Implementation Compared with LISP.
SIGPLAN Notices 12(8), August, 1977.

[Wiederhold 84] Wiederhold, G.
Knowledge and Database Management.
Software, Computer Society, IEEE 1(1), February, 1984.

[Winograd 72] Winograd, T.
Understanding Natural Language.
Academic Press, New York, 1972.