LEVEL I

Stanford Artificial Intelligence Laboratory
Memo AIM-316

Computer Science Department
Report No. STAN-CS-78-671

June 1978

NATURAL LANGUAGE PROCESSING IN AN AUTOMATIC PROGRAMMING DOMAIN,

by

Jerrold M. Ginsparg

Research sponsored by

Advanced Research Projects Agency

Doctoral thesis,

COMPUTER SCIENCE DEPARTMENT
Stanford University

MDA903-76-C-0206,
ARPA Order-2494

Stanford Artificial Intelligence Laboratory
Memo AIM-316

June 1978

Computer Science Department
Report No. STAN-CS-78-671

NATURAL LANGUAGE PROCESSING IN AN AUTOMATIC PROGRAMMING DOMAIN

by

Jerrold M. Ginsparg

This paper is about communicating with computers in English. In particular, it describes an interface system which allows a human user to communicate with an automatic programming system in an English dialogue.

The interface consists of two parts. The first is a parser called Reader. Reader was designed to facilitate writing English grammars which are nearly deterministic in that they consider a very small number of parse paths during the processing of a sentence. This efficiency is primarily derived from using a single parse structure to represent more than one syntactic interpretation of the input sentence.

The second part of the interface is an an interpreter which represents Reader's output in a form that can be used by a computer program without linguistic knowledge. The Interpreter is repsonsible for asking questions of the user, processing the user's replies, building a representation of the program the user's replies describe, and supplying the parser with any of the contextual information or general knowledge it needs while parsing.

This thesis was submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2494, Contract MDA903-76-C-0206. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, or any agency of the U. S. Government.

ACKNOWLEDGEMENTS

I would like to thank,

my advisor, Professor Terry Winograd,

the members of my reading committee: Professor Cordell Green and Dr. Daniel Bobrow,

the PSI group: Dave Barstow, Richard Gabriel, Elaine Kant, Juan Ludlow, Brian McCune, Jorge Phillips and Lou Steinberg,

and Martin Brooks.

for their help in the preparation of this thesis.

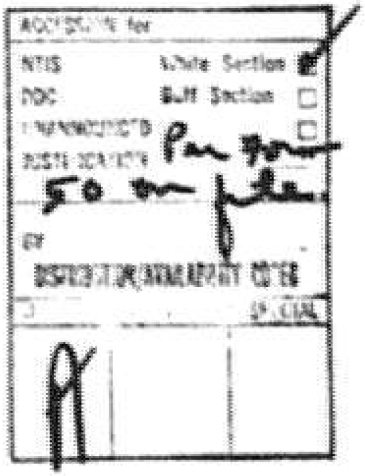


Table of Contents

Table of Contents

Table of Contents

Table of Contents

1. Introduction

This paper describes a natural language processing system. The system interacts with a human user, who describes a computer program to it in English. The output of the system is a program specification, a formal representation of the computer program the user has described. The program specification can be used as a data base for coding the user's program by computer programs without linguistic abilities.

Understanding program descriptions obtained via dialogues requires capabilities for handling almost all issues associated with natural language processing. Indeed, [Hobbs 77] mentions that even processing "well written algorithm descriptions" involves "...some of the hardest problems of linguistic analysis." Since many of the program descriptions posed by the users of the system can best be characterized as "not so well written", the system's natural language abilities must be extensive.

The system is most naturally viewed as two interrelated programs: a parser and an interpreter. Reader, the parser, provides the means of storing and utilizing the information about sentence structure (called syntax) which is necessary for the proper interpretation of the meaning of a sentence. Reader is used to transform the user's replies from strings of words into structures in which the relations between words are made explicit. The interpreter uses the structures supplied by Reader to construct the program specification.

1.1 Organization

The next section discusses the natural language abilities an automatic programming natural language system should have. The following section contains three short examples which should help to exactly clarify what is meant by the program specification, and provide some perspective on the natural language processing done by the system. The parser/interpreter can be used as part of a more complete automatic system. Section 1.4 briefly describes this system and the interpreter's interaction with it. Section 1.5 is a short overview of the operation of both Reader and the interpreter.

Chapter 2 is a general discussion of Reader. Chapters 3 and 4 continue that discussion in much more detail. Chapter 5 describes the program specification and how it is built by the interpreter. Appendix A contains several dialogues run by the system.

1.2 Capabilities

1.2.1 The parser

Reader was designed with the following criteria in mind.

The parser should be able to quickly recognize a substantial subset of English. The parsing should be done quickly, so that the parser can be used in a practical system. We mention parsing speed and grammar coverage together, because it is easy to theoretically achieve one or the other separately. Almost all parsing

schemes can parse a small set of sentences quickly, but few do as well when recognizing a large number of sentences while at the same time using a vocabulary which includes all possible syntactic uses for each word in the vocabulary. Reader achieves speed without sacrificing grammar breadth because its parsing process can combine several syntactic possibilities into a single parse path, thereby avoiding much of the backtracking or equivalently, parallel processing, which characterizes many other parsing schemes.

There should be a well defined interface between the parser and interpreter which allows the parser to interact with the interpreter and ask it to choose from among competing parses which are possible syntactic interpretations of a sentence. This is necessary because many sentences have more than one syntactic interpretation. For example, in "...find a relation in the concept marked 'possible.'", the parser must be able to ask whether the object of "find" is *"a relation whose marking is 'possible' which is in the concept."*, or *"a relation which is in the marked ('possible') concept."*

The parser should be able to use the evaluation function of the interpreter to provide parses in which most purely "function" words are eliminated. Consider the sentence, "Classify the input list on the basis of whether or not it fits the initial list". The interpreter should be asked to judge the modifications among "on the basis of", "classify" and the clause introduced by "whether". The parser should then incorporate the answers into the parse, resulting in a parse structure much closer to the meaning of the sentence than a mere syntactic structure.

```
((MP  (CLASSIFY NN
              (ARGS (LIST THE (INPUT)))
              (PROC (FIT NN
                         (ARGS (T))
                         (ARGS (LIST THE (INITIAL)))
                    ))
      )
)
```

The parse can be interpreted as,

> Perform a classification. The argument of the classification is the input list. The procedure for carrying out the classification is to test if the input list fits the initial list.

The parser's efficiency should *not* depend on using the interpreter to discontinue a possible parse of a sentence on semantic grounds. The parser-interpreter interface should only be asked to evaluate parses which are syntactically equivalent. Two partial parses are syntactically equivalent if both will lead to a successful parse on the same sentence endings, or if the end of the sentence has been reached and each is a successful parse. The reason for this decision is that in a rich environment we would expect the semantic processing required to discontinue a parse to be more expensive than the syntactic processing required to determine that the parse cannot lead to a syntactic interpretation. Woods, in [Woods 73], has experimented along these lines and found that (in his case) "...it looks as if it takes longer to do the parsing and semantic interpretation overall if the interpretation is done during the parsing than it does if the parsing is done first and the interpretation afterwards." Of course, semantic processing will have to be done to determine which syntactic parse of the sentence is most meaningful; the point is that we wish to avoid any semantic analysis whose effect could be achieved through syntactic analysis.

The assumption about the relative costs of semantics and syntactic processing cannot be proved. We can note, however, that even the simplest kinds of semantic checks can require arbitrary amounts of inference in a general system. For example, consider the decision of whether a pair of words ("street lights", for example) is a compound noun, or a noun followed by a verb. At first glance, it would seem that this could be cheaply done by simply checking a marker on the first word ("street"), which indicates whether it is a suitable subject for the proposed verb ("lights"). However, there are two problems with this approach. One is that simple markers on words are inadequate for dealing with the problems of language. Many words can be modified so that they are acceptable subjects for verbs which are not ordinarily associated with them, eg., "The glowing radioactive street lights the way for ...". The process of determining whether a modified noun is a suitable subject for an arbitrary verb seems beyond simple look-up techniques. The second problem is that even if the potential subject is unmodified, the syntax and meaning of the remainder of the sentence may constrain the behavior of the ambiguous pair to be the opposite of what one might expect. For instance, "water boils" would be predicted to be a noun-verb pair, yet in "Water boils are dangerous parasites which can be found in the Great Lakes.", it acts as a compound noun. It should also be noted that occasionally semantic analysis will be unable to act as a filter. "Set X" may be either a noun-verb pair or a noun and its appositive. The only way to tell is to know the syntactic context the words appear in. In "Set X to the empty set.", "set" acts as a verb; in "Set X is the empty set.", "set" acts a noun.

1.2.2 The Interpreter

The interpreter must be able to do the following:

1. Ask questions of the user. This enables the system to clarify actions it has taken and prompt the user for information he has omitted.

2. Understand three different types of user statements:

   User statements meant as steps in the program. These are translated into primitives in the program specification language. This is the basic method for building the program specification. "Print the greatest number in the list" must be translated into an "output" primitive with an argument representing "the greatest number".

   User statements directed as meta comments about the dialogue. These are translated into case frames which express their intent. This allows the user to control the flow of the dialogue. "Ask me about the structure of the data base first." must be interpreted as a request for a different question, rather than part of the program being written.

   Finally, some user statements should be understood as general comments about the program rather than as explicit instructions on coding it. "The program stores and retrieves data." is meant as an overall description of a program, not its first two steps.

3. Identify any objects and actions mentioned by the user with their correct referent in the program specification. If the user says "After printing it, print the

list containing it.", the interpreter must find a referent for "it", determine which "list" is meant, and match "printing it" to the appropriate operation in the program specification.

4. Use the question it has asked to aid in understanding the user's replies. In processing a description of two data structures, which are referred to as the "scene" and "concept", "The same as the concept." should be understood to mean "The scene has the same structure that the the concept has." If the question asked is "What is the structure of the scene?" However, the system must also be able to accept more information (in any order) than its question has asked for, eg.,

> What is the definition of the predicate "Reach"?
>
> A node X is connected to a node Y if there exists a pair in the graph such that X and Y are in the pair. X can be reached from Y if X is connected to Y or if X can be reached from a node which is connected to Y.

5. Learn definitions for any undefined words used by the user. If the system is to be robust, it must be able to infer certain information about words, rather than depend on knowing everything in advance. In the example above, the system inferred that "connected" is a binary predicate on nodes. If it is necessary to preprogram information of this sort, the system will fail every time an unfamiliar word is used, even though the word occurs in a context in which its meaning is apparent.

6. Incorporate implicit instructions from the user into the program specification while avoiding redundancy if the same instruction is later made explicit. Consider,

> 1. Print the result of the test, ask the user if this is correct, and read in the user's response.

versus

> 2. Print the result of the test and ask the user if this is correct.

In both 1. and 2., the next question the system should ask is "What is the structure of the user's response?". In 1., there is an explicit input operation mentioned. In 2., the system must infer the input operation because "ask" implies both an output and an input. The system must be able to supply an input for case 2., but realize that the user has already mentioned the input for case 1. This is not as trivial as just checking for an input after every output generated from "asks", since if the user says,

> "Output the result of the test and ask the user if this correct. Then read in another test item.",

the system must still ask for the structure of the user's response.

7. Use a certain amount of programming knowledge to aid in its construction of the program specification. Understanding many of the user's replies will require specific bits of programming knowledge. If the system asks, "What is the exit test of the loop", and the user replies, "Stop when 'quit' is typed", the interpreter must know that this means to test the argument of the (presumably one) input operation in the loop to see if it is "quit". If so, the loop should be exited. The same information tells the interpreter that the test should be inserted into the program after the input operation.

1.3 Three Examples

This section consists of three brief examples[1] intended to illustrate the extent of the processing done by the system.

[1] Every example in this paper was produced by the system.

WHAT IS THE NAME OF THE PROGRAM YOU WISH TO WRITE?

Write me a program called intersection which reads a set of lists of numbers and prints the numbers which are in all of them.

```
INTERSECTION
   A1 ← READ()
   A4 ← The set of all A2 in all A3 in A1 such that:
        FORALL(B1) IMPLIES(MEMBER(B1 A1)
                           MEMBER(A2 B1))
   For all A2 in A4 do:
        PRINT(A2)


B1 is a variable bound to A3.

A3 is the generic element of A1.

A1 is a set whose generic element is a list whose generic element
is a number.
```

Figure 1.1

An algorithm description and its program specification

The top section of Figure 1.1 contains a description (in answer to the system's question) of a program which finds the intersection of a set of lists of numbers. The program specification for the example is shown in Figure 1.2 on the following page. It consists of a series of interconnected nodes which represent the various components of the program. Each component type is fully described in Chapter five. For large programs, the program description is too bulky (and generally unreadable) to exhibit, so a "pretty printed" version of it will be shown instead. A simple program is used to print the specification as an Algol like control structure with data descriptions in English. The result of printing the specification in Figure 1.2 is shown beneath the algorithm description in Figure 1.1.

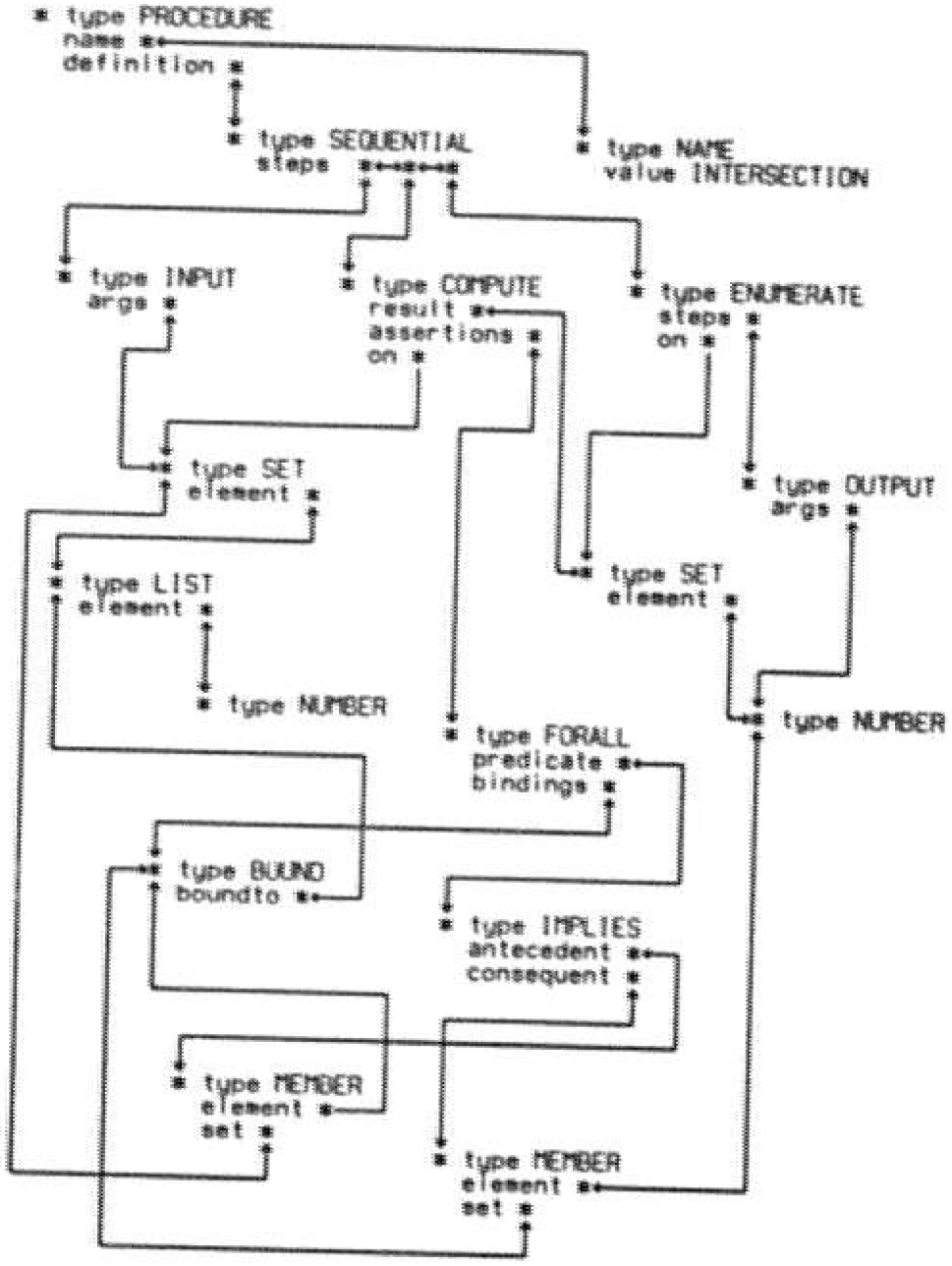


Figure L2

"Write me a program called intersection which inputs a set of lists of numbers and prints the numbers which are in all of them."

The relation between the specification and its "pretty printing" is apparent. As an example, consider the printing of the ENUMERATE component. ENUMERATEs are the specification primitive for performing an action (the STEPS slot) on each element of a set (the ON slot). To "pretty print" an ENUMERATE component, the printing program merely concatenates,

For all
⟨*the* ELEMENT *of* ⟨*the* ON *of* ENUMERATE⟩⟩
in
⟨*the* ON *of* ENUMERATE⟩
do:
⟨*the* STEPS *of* ENUMERATE⟩

One of the points this example makes is that the best method for implementing the user's program is not necessarily contained in the program specification. If the program specification were implemented straightforwardly, the resulting program would be grossly inefficient. The intersection set would be created by cycling through *every* list in the input set, rather than just the first. The reason, of course, is that the user has asked the system to write a program which finds a set satisfying some assertions, but has not given a method for determining the set. Therefore the system falls back on the method of testing each element in the set to see if it satisfies the assertions. The interpreter can accept more process oriented program descriptions. Figure 1.3 contains a very low level description of an "intersection" program.

WHAT IS THE NAME OF THE PROGRAM YOU WISH TO WRITE?

Intersection

DESCRIBE INTERSECTION.

It reads a list X.

WHAT IS THE STRUCTURE OF THE GENERIC ELEMENT OF X?

A list of of numbers.

ARE WE FINISHED?

Set First and Output to the head of X. Let Rest equal the tail of X. If First is empty, print Output and return. If Rest is empty, set First equal to the tail of First, set Rest to the tail of X, and go back to the first conditional. If the first element in First is not in the head of Rest then remove it from Output, set First to its tail, set Rest to the tail of X and go to the conditional. Otherwise, set Rest to the tail of Rest and return to the conditional.

```
        INTERSECTION
          X ← READ()
          FIRST ← HEAD(X)
          OUTPUT ← HEAD(X)
          REST ← TAIL(X)
Label1:   IF EQUAL(FIRST PHI)
            Then PRINT(OUTPUT)
                 RETURN()
          IF EQUAL(REST PHI)
            Then FIRST ← TAIL(FIRST)
                 REST ← TAIL(X)
                 GOTO Label1
          IF NOT(MEMBER(HEAD(FIRST)
                        HEAD(REST)))
            Then OUTPUT ← REMOVE(HEAD(FIRST) OUTPUT)
                 FIRST ← TAIL(FIRST)
                 REST ← TAIL(X)
                 GOTO Label1
             else REST ← TAIL(REST)
                  GOTO Label1

        REST is a list whose generic element is a list whose generic element
        is a number.

        OUTPUT is a list whose generic element is a number.

        FIRST is a list whose generic element is a number.

        X is a list whose generic element is a list whose generic element is
        a number.
```

Figure 1.3

A low level description of intersection

As a prelude to chapter five, note that even though this dialogue (unlike most the system handles) translates fairly directly into primitives in the program specification, there are still several natural language problems embedded in it.

Different ways of specifying the same action:

Set X equal to Y.	Go back to X.
Set X to Y.	Go to X.
Let X equal Y.	Return to X.

Reference problems:

...its tail.
...remove it from Output.
...go to the first conditional.
...the conditional.
...the first element in First.

Language conventions:

Otherwise, set...

The interpreter can handle more "structured" low level descriptions. Figure 1.4 contains an example.

WHAT IS THE NAME OF THE PROGRAM YOU WISH TO WRITE?

Intersection

DESCRIBE INTERSECTION.

It reads a list X. X is a list of lists of numbers. Let First equal the head of X and Output equal First. While First is not empty, set Rest to the tail of X. Then while Rest is not empty, if the head of First is not a member of the head of Rest, remove it from Output and set Rest to empty. Otherwise set Rest to the tail of Rest.

DOES THE SECOND LOOP BELONG IN THE TOP LEVEL OF INTERSECTION OR IN THE TOP LEVEL OF THE FIRST LOOP?

In the loop.

ARE WE FINISHED?

After the second loop, set First to the tail of First.

ARE WE FINISHED?

Print Output after the first loop.

```
INTERSECTION
  X ← READ()
  FIRST ← HEAD(X)
  OUTPUT ← FIRST
  While NOT(EQUAL(FIRST PHI)) do
       REST ← TAIL(X)
       While NOT(EQUAL(REST PHI)) do
            If NOT(MEMBER(HEAD(FIRST)
                          HEAD(REST)))
              Then OUTPUT ← REMOVE(HEAD(FIRST) OUTPUT)
                   REST ← PHI
              else REST ← TAIL(REST)
       FIRST ← TAIL(FIRST)
  PRINT(OUTPUT)

REST is a list whose generic element is a list whose generic element
is a number.

OUTPUT is a list whose generic element is a number.

FIRST is a list whose generic element is a number.

X is a list whose generic element is a list whose generic element is
a number.
```

Figure 1.4

A more structured intersection program

In general, the program descriptions the interpreter is asked to handle will be a cross between high level descriptions like the first dialogue and low level descriptions like the second two. The dialogues in Appendix A provide further examples of this.

In the dialogue from Figure 1.4, the interpreter had to ask the user whether the second loop was embedded in the first. More programming knowledge would have supplied the answer for the interpreter.[2] It should have been obvious that the description of the first loop was incomplete, since the exit test checked the value of variable whose value remained unchanged in the loop. Such knowledge is beyond the scope of the present parser/interpreter project. Instead, it is made available to the interpreter via the PSI system [Green 76].

1.4 PSI

The parser/interpreter has been designed to run as a part of the PSI automatic program synthesis system. The PSI system, which is being written as a group project at the Stanford University Artificial Intelligence Laboratory, consists of a number of different modules, one of which is the parser/interpreter system. Together, the parser/interpreter and the other PSI modules form a complete automatic programming system.

The most obvious addition supplied by the PSI system is the coding and efficiency module which is intended to produce optimized LISP or SAIL code from the program

[2] As we have mentioned, the interpreter has some programming knowledge; for instance, it knows enough to know it doesn't know where the loop goes.

specification. Thus the user is encouraged to use a very high level description for his program since the specification specifies the performance of the desired program, but not its implementation. [Barstow 77] and [Kant 77]

The remaining modules in PSI help support the the dialogues run by the parser/interpreter. The parser/interpreter can run independently of them, but its performance is weak (or nonexistent) in the areas these modules were designed for. The other PSI Modules are:

An English generator being developed by Richard Gabriel. The generator should not be confused with the English data description printer used in pretty printing the program specification. The data description printer uses a "fill in the blanks" paradigm (X is a Y with Z whose Q etc.), which is adequate for its purposes. The completed PSI generation system will include a program explanation module which will displace the data description printer.

A programming knowledge module. This module is responsible for checking the consistency of the program specification, suppling questions to be asked in case of inconsistencies, and answering questions whose answers can be derived from information about programming. [McCune 77]

A domain knowledge module which is being written by Jorge Phillips. This module is analogous to the programming knowledge module except that it has information about the specific type of program written, as opposed to programming in general. It might know, for instance, that in a text editing domain, when the user says "exit the file", he means "write all the changes made onto the disk and *then* exit the file."

A traces and examples module which enables the user to describe his program in terms of examples and traces as well as English. [Phillips 78]

A dialogue moderater which coordinates the various PSI modules, chooses which question to ask the user next, and processes the user's comments about the dialogue supplied to it by the parser/interpreter. [Steinberg 78]

1.5 An Overview

1.5.1 Reader

Reader can be briefly described as a left to right parser that uses a combination of top-down and bottom-up strategies. The method used at any point in a parse is determined by the grammar writer. The grammar consists of a set of Lisp programs which manipulate the data structures and data structure building primitives supplied by the parser.

Reader is able to efficiently recognize a large subset of English because it seldom needs to maintain more than one possible parse of a sentence. It should be stressed, however, that Reader is not completely deterministic[3]. Complete determinism does not seem possible when dealing with a large grammar and vocabulary in which most words can fulfill more than one syntactic role.

The characteristics which allow Reader to parse nearly deterministically are listed

[3] Almost all the nondeterminism arises from words which belong to more than one word class; eg., if a word can act as either a verb or a noun, Reader must try both possibilities separately.

below. In Section 3.2, these characteristics are divided into essentially three different categories.

1. A sentence constituent is only built when the parser knows that there is at least one other constituent that has already been built that can accept the first as a modifier.

2. A constituent is attached (ie., proposed as a modifier) to another constituent only when the attachment is forced by the syntax of the sentence. A simple example of "delayed attachment" occurs in the sentence, "The program called Intersection...". The constituent "called Intersection" is not attached to "the program" until the words following "Intersection" require that the attachment be made.

3. Because of 2., when a constituent is attached to another, the parser generally knows the reason for the attachment, and can use that reason to guide it in making the attachment. For instance, in "The program called Intersection was written by George.", "was" forces "called Intersection" to be attached to "The program". The reason for the attachment is to allow "The program" to be the subject of "was", so it is clear that "called Intersection" is to be attached as a relative clause modifying "program", since if it were attached as the main verb, there would be no place to put "was". In "The program called Intersection and returned.", when "and returned" is read, the parser knows that the clause "called Intersection" must be an active construction (as opposed to the passive construction which leads to the relative clause interpretation) so that it can be attached to "The program" as the predicate of the sentence.

4. The parser uses one syntactic structure to represent more than one possibility. In "The program called Intersection ...", the structure "called Intersection" simultaneously represents the predicate of the sentence and a relative clause. Which interpretation to use is determined after more of the sentence had been read.

5. The parser provides for local ambiguity in the parse structure that it returns. For instance, "I know that ice is dangerous" could mean either "*I know ice is dangerous.*" or "*I know that that (particular) ice is dangerous.*". The parser finds both interpretations following a single parse path, and continues following a single path after the ambiguity has been reached by preparing an output structure in which the subject of "is" is a choice between "that ice" and "ice".

As we have indicated, occasionally Reader must pursue more than one parse path at a time. To avoid analyzing the same sentence constituent each time it is

encountered on a different parse path, Reader uses a variation of the well-formed substring table idea (section 4.4). This enables a constituent which has been analyzed to be effectively shared by each parse path that can use it.

The parser-interpreter interface is only called to rate structures which are about to be attached to other structures. Structures are attached to other structures only when the syntax of the sentence forces the attachment. These two facts imply that the parser-interpreter interface will only be asked to evaluate those parses which are syntactically equivalent[4]. For a simple example of this, consider "The number in the list the program printed was ..." "Was" forces the "The number", "in the list", and "the program printed" to be attached to one another for the purpose of allowing "The number" to be the subject of "was". The parser-interpreter interface must choose from between structures which represent the meanings "*The number which was printed and in the list.*" and "*The number which was in the printed list.*" Since each structure plays the same syntactic role, namely that of a noun group, any sequence of words following "was" will lead to a parse for either both or neither of the two interpretations.

Reader's interface with its interpreter is a program called Format which rates each syntactic structure built by Reader before it is attached to another. The criteria measured by the interface are:

> 1. Does the verb of the structure (if there is one) have enough of its cases filled in to properly specify the action it represents? For example, the verb "put" requires a case which specifies *where* the object of "put" was put.
>
> 2. How appropriate are the noun groups in the structure? For instance, the noun group "water boils" would be judged inappropriate.

[4] Two parses are syntactically equivalent if and only if the end of the sentence has been reached and both are successful parses, or if both will lead to a successful parse on the same sentence endings.

3. How appropriate are the contents of the cases of the structure's verb. For instance, "street" is an inappropriate subject for "light".

The results of the rating are used to pick the most meaningful structure from among equivalent syntactic possibilities. Structures which evaluate poorly can still be included in the parse of the sentence, as long as there are no other parses which contain structures with better evaluations. The parse of "Water boils are very small." contains the "inappropriate" noun group "water boils", since there is no syntactic interpretation of the sentence which does not use "water boils" as a noun group.

1.5.2 The Interpreter

This section briefly touches on reference and concept matching, two of the subjects mentioned in section 1.2.2, as an introduction to the methods used by the interpreter. They have been singled out because they are the basis of all higher level inferences performed by the interpreter. Chapter 5 covers much more in greater detail.

The interpreter's primary means of understanding user statements is via a set of case frames and concepts. The case frames map English verbs and their modifiers into the concepts, which can then be incorporated into the program specification. For a simplified example, consider the concept of an input operation, denoted #INPUT. For now, we will assume that #INPUT takes has descriptors, its arguments (ARGS), its place in the program specification (STEPOF), and the input device (DEVICE).

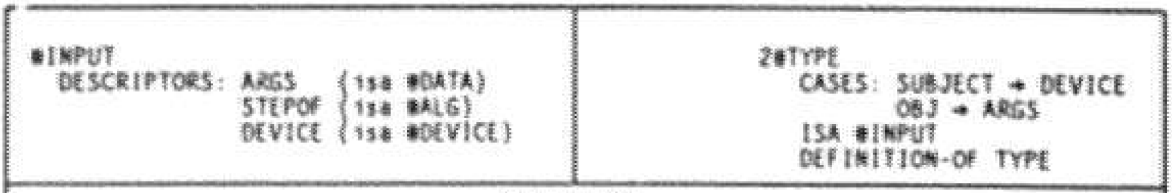


Figure 1.5

A concept and a definition which can be mapped to it.

Figure 1.5 shows the concept and a definition of "type" which can be mapped to it. The definition says that if we have an instance of the verb "type", and its cases (as determined by the parser) can be mapped successfully (ie., the contents of the cases satisfy the criteria in the descriptors of the #INPUT), then we can view the verb and its cases as an instance of the #INPUT concept and take the appropriate action. Concepts can represent more than a single primitive in the program specification language. For instance, "request" in "I'll request a story by giving a key word." maps into an #INTERCHANGE concept which involves an INPUT and OUTPUT operation with a calculation of what should be output in between.

Noun and pronoun reference is facilitated by the context supplied by the selection criteria of the descriptors of a concept. In,

> "It reads in a trial-item, matches the *input* to the internal concept model, and prints the result of the match."

a referent must be found for the noun "input". There are two possibilities: the INPUT created by the "read", and the trial-item which is the argument of the "read". Since "match" is mapped to a concept (#PREDICATE) which requires that its ARGS descriptor be a #DATA (rather than an #ALGORITHM like the "read") the ambiguity is resolved.

When the choice among possible referents cannot be decided on the basis of the very general type checking outlined above, more situational checks are needed. Consider,

> "It reads a list of numbers and a list of strings. If X is in the list then..."

There are two referents for "the list"; the number list and the string list. Since they both satisfy the selectional criteria[5] for the second argument of the #MEMBER "is in" maps into, something more context dependent is needed. Each concept has a second layer of selectional requirements which are called when simple type checking fails to narrow down the field of choices sufficiently. For #MEMBER, the check succeeds if the first argument has the same type, or is referred to in the same way, as the generic element of the second argument. So in the example, if X were a string, "the list" would be matched to the string list, and if X were a number, "the list" would be matched to the number list.

In the event of a referent which remains ambiguous after all tests have been applied, the time honored method of falling back on the most recently mentioned possibility is used. Hopefully, the speaker has felt free to use a pronoun in an ambiguous situation because the referent he had in mind was the most recently mentioned possibility.

---

[5] They are both sets.

2. Parsing

Natural language processing begins with parsing. Determining the meaning of a sentence requires knowing the main verb of the sentence and how the rest of the words in the sentence relate to it. In this system, for example, the mapping of the sentence "Print the list." into a structure which is an OUTPUT operation whose argument is the referent of list is dependant on knowing that the main verb of the sentence is "print", the syntactic object of "print" is "the list", and the sentence is an imperative.

2.1 The Basic Algorithm

A pa er allows one to store and utilize the information about sentence structure needed to interpret sentences properly. The information that is stored is referred to as the grammar, while the methods for applying the grammar to a particular sentence are usually thought of as the parser. Reader is organized somewhat differently from most parsers[1] in that Reader is not syntax directed. Writing a grammar for Reader consists of specifying the processes which build the structure of an input sentence. Thus the grammar writer specifies how the grammar is actually applied to a sentence, as well as the grammar itself. Reader's function is to provide the data structures the grammar is intended to use, the control structure which activates the grammar, and programs for manipulating the data structures.

The two basic data structures that Reader supplies are the *modifier list* and the

[1] The parsers of Winograd and Riesbeck are also exceptions. See section 4.5.

stack. The *modifier list* is a list that the grammar writer can use to store words whose use has not yet been determined. The stack is used to store the structure built up while the parse is in progress. The next section describes the stack in detail. A stack, a modifier list, a message about what has just happened to the top of the stack, and a message concerning the entire stack constitute a partial parse. The top of the stack message is usually a Lisp atom, eg., message = NOUN, VERB, or CONJUNCTION means that a noun, verb or conjunction has just been added to the top structure in the stack. The stack message is a list of features that the stack has. Each feature is represented by an atom. Example features are "the stack contains a verb structure with a verb that can accept a clause as one of its cases" and "the stack represents a sentence which is an interrogative".

The parse is performed by adding each word in the input (going from left to right) to the partial parse formed by the addition of the previous words in the sentence. The first word in the sentence is applied to "the initial partial parse", which consists of the "initial stack" (a stack containing a single structure which will eventually hold the main verb of the input sentence), and an empty modifier list. The "top of the stack message" for the initial stack is BEGIN, and the message concerning the entire initial stack is NIL, meaning that the stack has not acquired any features yet.

The process of adding words to the partial parse is controlled by the grammar. The grammar consists of a set of programs, one for each syntactic word class[2], which contain the rules and conditions which specify when and how to add a particular word class to a partial parse in a given configuration. In general, there may be more

[2] the word classes the parser uses are VERB, PREPOSITION, NOUN, MODIFIER, ARTICLE, CONJUNCTION, and PUNCTUATION.

than one way a word class can be added to a partial parse. It is also true that many words belong to more than one word class. For instance, the word "like" can be a noun ("His likes are different than mine."), a verb ("She likes him."), a preposition ("a man like him."), a conjunction ("He pleys like Jack used to."), or a modifier ("men of like temperament."). These two facts (a word may be added to a partial parse in more than one way, and a word may belong to more than one word class) imply that the parser should be able to handle more than one partial parse of the input at a time. However, it should be kept in mind that one way to achieve an efficient parsing process is to write a grammar which minimizes the number of possible parses the parser has to follow at once, while at the same time writing a set of rules which adequately express English syntax. Section 3.2 shows some of the methods used by Reader's grammar to avoid a multiplicity of partial parses.

The partial parses are placed on a list called the "partial parse list". The parser's control structure is as follows:

1. *sentence* ← the list of words comprising the input sentence.
2. *partial-parse-list* ← a list of the initial partial parse.
3. WHILE *sentence* DO
    4. Apply the next word in *sentence* to each partial parse in *partial-parse-list*, using the program associated with each word class the word belongs to.
    5. Reset *sentence* by removing the first word in it.
    6. Reset *partial-parse-list* to a list of the partial parses formed in step 4.
7. Output *partial-parse-list*.

Step 5. does not imply that the grammar programs cannot look ahead in the input

sentence and use more than one word at a time. If a grammar program continues a partial parse *P* by applying the first *n* ($n > 1$) words in *sentence* to it, a message is left which prevents the next $n - 1$ words from being applied to *P*. This presentation of the control structure is accurate with the exception that steps 6. and 7. are a bit more complex than they have been made to appear. They will be explained in more detail in later sections.

The control structure indicates that the parallel processing is invisible to the grammar writer. This means that in writing the grammar programs, the grammar writer need only concern himself with one stack and one modifier list, since each grammar program is called on each partial parse in *partial-parse-list* in turn.

2.2 Stack structures and collapsing

The stack is the major data structure that Reader uses. Its function is to store the structures built up during the parse until it is decided how the structures should be attached to one another. This treatment allows for easy handling of a certain type of ambiguity that arises frequently in English utterances.

Consider the sentence, "I had another look at it". It can mean either *"I asked someone else to look at it"* or *"I took one more look at it"*. The ambiguity arises from the different uses of "had", "look" and "another" in each interpretation.

The sentence "John spoke to the man with Bill" is ambiguous in a different way. It might mean *"John and Bill spoke to the man."* or *"John spoke to the man who was with Bill."* In this sentence the ambiguity derives from the fact that "with Bill" can

be used to specify either who acted with John, or who was near the man. In each meaning, the words of the sentence have been used in the same fashion. Ambiguities of this sort, one constituent of an utterance being a possible modifier for more than one word in the utterance, have been referred to as "permanent predictable ambiguities" in [Sager 73].

The stack allows Reader to handle ambiguities of the second kind by allowing for the structuring of most of the constituents of the sentence before it is decided which words they will modify. The elements of the stack are called stack structures. Two different types of stack structures are employed by Reader: preposition structures and verb structures. The sentence "John lost the toy he bought in the woods on Sunday." would be parsed into the following stack:

4. [on Sunday]
3. [in the woods]
2. [he bought]
1. [John lost the toy]

1. and 2. would be represented by verb structures and 3. and 4. by preposition structures. Verb and preposition structures can be filled in as follows:

Verb structures	Preposition structures
noun3	noun
noun2	prep
noun1	adverbs
verb-group	measure
adverbs	message
cases	
function	
measure	
message	

The noun slots are filled by noun groups. A noun group consists of a list of the head noun followed by its modifiers. A verb may have one, two or three of its noun slots filled. A preposition may have its noun slot filled or not.

The verb-group slot is filled by a list of verbs. Each verb consists of a root and an ending.

The **adverbs** slot is filled by a list of modifiers of the verb group or preposition.

The **cases** slot is filled by the cases the verb has that are introduced by prepositions and conjunctions.

The **function** slot contains the function of the verb structure. MAIN is used to indicate that a verb structure holds the main verb of an utterance, RC indicates a verb structure is being used as a relative clause, etc.

The **prep** slot holds the preposition of a preposition structure.

The **message** slot contains information relevant to the stack structure. Its contents are controlled by the grammar. We will see examples of its uses when we discuss the grammar.

The **measure** slot contains the parser's rating of each structure. The rating is used to help the parser choose among competing parses. It will be defined in section 4.1.

Throughout this paper, stack structures will be printed as a collection of slot-value pairs. Empty slots will not be printed. Under this scheme, the stack for the sentence above would be printed as

```
    - - -
    PREP: ON
4.  NOUN: SUNDAY
    - - -
    PREP: IN
3.  NOUN: (WOODS THE)
    - - -
    VERB: ((BUY ED))
    NOUN1: HE
2.  FUNCTION: RC
    - - -
    VERB: ((LOSE ED))
    NOUN1: JOHN
    NOUN2: (TOY THE)
1.  FUNCTION: MAIN
    - - -
```

"John lost the toy he bought in the woods on Sunday."

The stack could be interpreted in several different ways:

a. *John lost a toy. He bought it in the woods. He bought it on Sunday.*
b. *John lost a toy. He bought it in the woods. He lost it on Sunday.*
c. *John lost a toy. The toy was lost on Sunday. It was lost in the woods. John bought the toy.*
etc.

The process of determining which of the interpretations was actually intended by the speaker is referred to as *collapsing the stack*, since finding the correct interpretation of the stack consists of reducing the stack to one stack structure. If we accept meaning c. as the proper interpretation of the above sentence, then the single stack structure that represents that meaning of the stack is

```
- - -
VERB:  ((LOSE ED))
NOUN1: (TOY THE (BUY PN [SUB HE] ))
NOUN2:  JOHN
CASES: ((WHERE (IN (WOODS THE)) (WHEN (ON SUNDAY)))
FUNCTION: MAIN
- - -
```

where "he bought" specifies which toy, "on Sunday" specifies when the toy was lost, and "in the woods" specifies where the toy was lost.

The parser must consult with its deductive system[3] during a Collapse of the stack. The reason that the third meaning seems to be right is that one is unlikely to buy a toy in the woods, since there usually aren't any stores located in the woods. The parser also needs to know that Sunday is a possible date rather than a location for the woods. There is, however, some syntactic knowledge embedded in the stack. The parser never considers,

---

[3] The deductive system for the Reader/Interpreter system is the Interpreter. In discussing the parser in general, we will use "its deductive system" to mean the program which calls the parser and is able to reason about the subject domain of the sentences being parsed.

d. A toy was lost in the woods by John. John had bought the toy.
The toy was bought on Sunday.

as a possible meaning for the sentence since *d.* requires that stack structure 4. modifies 2., while 3. modifies 1. English syntax does not allow such crossovers, so the parser never has to consider *d.* as a possible meaning.

The communication channel between the parser and the interpreter is a function named Format. Format is called to evaluate a structure just before it is attached to another structure during a Collapse.[4] The algorithm used by Collapse ensures that once a structure has been attached to another, it cannot be modified (ie., have another structure attached to it). Formatting serves the dual purpose of preparing a structure for output, and providing the deductive system with an opportunity to rate the likelihood that the speaker intended the words in the structure to be grouped with each other. The rating of a formatted structure is merged with the contents of the **measure** slot of the structure it is being attached to. Thus the **measure** slot of a structure contains the ratings of all the structures that have been attached to that structure. The measure of a structure is discussed in section 4.1.

Collapse chooses which one of the possible stack structures the stack could be collapsed to by picking the structure with the best measure. If there is more than one partial parse active at the end of the sentence, Reader returns the one(s) whose collapsed stacks have the best measure. The format of a preposition structure is its **measure** and a list of the preposition, adverbs and noun; the format of a verb structure is its **measure** and a list of the root of the main verb, the tense

[4] Format is also called evaluate the final structure obtained from the parsing process.

of the verb group, the verb's adverbs, and the verb's cases. Measure is only used to select from among syntactically equivalent parses, so if the only reading a sentence admits results in a bad measure, a parse will be found anyway.

When the stack for "John lost the toy he bought in the woods on Sunday." is collapsed, the measure of any resulting structure which includes structure 3. (in the woods) attached to structure 2. (he bought), will be worse than those that don't, since the measure of structure 2. modified by structure 3. will be "unacceptable" (see section 4.1) since the parser's deductive system would "know" that "the woods" does not satisfy the requirements that "buy" has for places where one can buy things. Section 5.5 explains how this "know" is implemented in the Reader/interpreter system.

We can now mention the complication referred to in step 7. of the control structure presented in section 2.2. Step 7. was originally "Output the list of partial parses". What really happens is that Reader collapses the stacks associated with each partial parse, each structure resulting from the collapse is formatted, and then Reader then outputs a list of the formatted structure(s) with the best measure.

There are two points about the stack which should be emphasized:

1. There are only two reasons for collapsing the stack: either the end of the sentence has been reached, in which case the stack is collapsed down to one structure, or the application of a word in the sentence to a partial parse results in that word being added to a stack structure which is not at the top of the stack. In the latter case, the stack is collapsed down to the structure that is receiving the word.

2. Any two structures resulting from the collapse of a stack are syntactically equivalent. This means that either both or neither will result in a parse of the sentence, so we are justified in using semantics to discard all but one of the structures resulting from a collapse, since syntactic information will not enable us to choose between them.

2.3 Reader's output

2.3.1 Cases

Given a sentence S, Reader's output consists of the main verb of S, together with its cases. If S is the simple sentence, "Bill hits John", then Reader's output would be the parse below:

```
(HIT NN
    [SUB BILL]
    [OBJ JOHN]
)
```

The open bracket, "(", signals the beginning of a presentation of a verb and its cases. NN is a tense marker whose meaning will be explained below. The SUB case (cases are introduced by square brackets, "[") of "hit" is "Bill" and the OBJ case is "John".

We are using "case" in a different sense than most of the current literature does. In the literature, "case" is usually used to refer to "deep case", a concept popularized by Fillmore in [Fillmore 68]. A good definition of "deep case" can be found in [Bruce 75]: "The deep cases are binary relations which specify an event regardless of the surface realization of that description as a sentence or noun phrase". To see exactly what this means, we will consider a number of sentences involving the verb (event) "hit". For this example, we will suppose that "hit" has three deep cases: the entity that is receiving the effect of the hit (OBJECT), the thing the object is being hit with (INSTRUMENT), and the entity that is instigating the hitting (AGENT). Then in

1. Bill was hit by the hammer.
2. John hit Bill with the hammer.
3. Bill was hit with the hammer by John.

4. The hammer hit Bill.
5. John hit Bill.

"Bill" is the OBJECT in all five sentences, "hammer" is the INSTRUMENT in the first four sentences, and "John" is the AGENT in sentences 2.,3. and 5. Consider the knowledge needed to choose the cases of a "hit". In sentence 5., the AGENT is distinguished from the OBJECT by their relative positions about the verb. The surface structure of the sentence, then, is one source of information in determining a verb's cases. It is obviously not the only source. Sentence 4. has the same surface structure as sentence 5., yet the noun preceding the verb is considered the INSTRUMENT, rather than the AGENT. Furthermore, if we say,

> "George went berserk. He battered John into unconsciousness, picked him up, and hurled him at Bill. John hit Bill.",

then John is the INSTRUMENT of "hit" in the last sentence. Therefore, determining cases requires the surface structure of the sentence as well as information about the objects the sentence refers to, and the context the sentence was uttered in. Reader produces a set of cases which are derived from the surface structure of the sentence. A deductive system can then use Reader's cases in combination with the information it has about the concepts mentioned in the sentence to derive its own cases.

The three primary cases used by Reader are SUB, OBJ and IOB (indirect object). In a passive sentence, one in which the verb group is a verb phrase whose last two verbs are the verb "to be" and the main verb inflected with an "ed" or "en" ending, the OBJ precedes the verb and and the SUB is introduced by "by". If the sentence is not passive, the OBJ follows immediately after the verb and the SUB precedes the verb. The IOB is a noun that can modify a verb, without needing a preposition to introduce it, only in the presence of both the SUB and OBJ.

"John" is the IOB in "Bill gives John the book." since we can not say "John gives Bill." to mean that "Bill received something from John.", but can say "Bill gives the book." to indicate that "A book was given to someone by Bill." Similarly, John is the IOB in "Bill names the cat John" since we can't say "Bill names John." to mean that "Bill has given the name JOHN to something.", but can say "Bill names the cat." to indicate that Bill given some name to the cat. Another way to look at this is that (without resorting to prepositions) you cannot say (using the verb "give") who you are giving something to without mentioning what you are giving, and similarly you can't mention what you are naming something without mentioning the thing being named. The reversal in the normal order of IOB and OBJ that verbs like "name" exhibit is considered a syntactic property of the verb. Unless a verb is tagged with this property, Reader assumes that it takes its OBJ and IOB in the normal order.

With the exception of "by" and "to", Reader does not try to assign meaningful case names to nouns introduced by prepositions, since the meaning of the modification between a verb and a prepositional phrase depends on both the verb and the object of the preposition. The deductive system is expected to supply a case name when it judges the appropriateness of the modification.

In passive sentences, "by" frequently introduces the SUB. When Reader parses such a sentence it returns the object of "by" as the SUB of the verb if the deductive system agrees that the object could serve as the SUB. Given the sentence "Bill was shot by Jack", Reader would ask the deductive system whether Jack could shoot Bill. If the answer were "yes", Jack would appear as the SUB case of "shoot". Change the sentence to "Bill was shot by the door" and the deductive system would answer "No, doors cannot shoot", enabling Reader to use "by the door" to specify the location of the shooting.

"To" is treated similarly to "by" by Reader in that Reader assumes that "to" always introduces an IOB if the syntax of the sentence permits this. Therefore,

"Bill gives John the book" and "Bill gives the book to John" parse to

```
(GIVE NN
     [SUB BILL]
     [IOB JOHN]
     [OBJ (BOOK THE)]
)
```

```
(GIVE NN
     [SUB BILL]
     [OBJ (BOOK THE)]
     [IOB JOHN]
)
```

respectively.

The parses for the five example sentences are:

Bill was hit by the hammer.

```
(HIT PN
     [OBJ BILL]
     [SUB (HAMMER THE)]
)
```

John hit Bill with the hammer.

```
(HIT PN
     [SUB JOHN]
     [OBJ BILL]
     [PREP (WITH (HAMMER THE))]
)
```

Bill was hit with the hammer by John.

```
(HIT PN
     [OBJ BILL]
     [SUB JOHN]
     [PREP (WITH (HAMMER THE))]
)
```

The hammer hit Bill.

```
(HIT PN
     [SUB (HAMMER THE)]
     [OBJ BILL]
)
```

John hit Bill.

```
(HIT PN
     [SUB JOHN]
     [OBJ BILL]
)
```

We can see that SUB corresponds to either AGENT or INSTRUMENT, and that OBJ corresponds to OBJECT in the case system we had made up for "hit".

To translate Reader's cases into the "hit" case system one would only have to decide which SUBs were INSTRUMENTs and which were AGENTs, equate OBJECT with OBJ, be aware that "with" can introduce the INSTRUMENT, and be able to distinguish when "with" refers to an instrument and when it doesn't. A non-trivial task, since we could say

"He hit John with Bill"	(accomplice)
"He hit John with vim and vigor"	(method)
"He hit John with malice"	(emotion)

Section 5.2 explains how Reader's cases are mapped into the interpreter's case system.

Reader actually uses more cases than than the primary ones mentioned above. But the other cases are essentially ad-hoc ones that Reader uses to store modifiers of the verb. Any preposition or conjunction (not top-level) defines its own case. As an example, consider "John pushed Janet into the closet because he thought Bill would see her.", which is parsed to:

```
(PUSH PN
     [SUB JOHN]
     [OBJ JANET]
     [PREP (INTO (CLOSET THE))]
     [BECAUSE (THINK PN
                    [SUB HE]
                    [WHAT (SEE (NN WOULD)
                               [SUB BILL]
                               [OBJ HER]
                          )]
              )]
)
```

John and Janet are the SUB and OBJ of push. "into the closet" is a preposition case

of "push", filling in where the OBJ was pushed to. The conjunction "because" fills in the presumed reason the event took place, and is considered a case of the verb. It contains the verb clause whose main verb is "think". "He" is the SUB of "think". "What the SUB is thinking" is stored in the WHAT case of "think" The contents of the WHAT case is the verb clause whose main verb is "see".

2.3.2 Tense markers

Many verb clauses contain verb groups rather than just single verbs. A verb group can be composed of adverbs, modals and other verbs. The information contained in a verb group that a deductive system needs is a list of adverbs and modals, the root of the main verb, and the tense of the verb group. Reader saves the modals and adverbs and returns them in appropriate slots in the parse structure. The root of the main verb of the sentence is similarly returned. This means that Reader must supply the tense of the verb as a separate piece of information. Reader uses six basic tense symbols. These are shown in Figure 2.1, together with an example of the verb group each represents.

VERB GROUP	TENSE	
I walk	NN	The present tense of the verb without any auxiliary verbs.
I walked	PN	The past tense of the verb without any auxiliary verbs.
I will walk	FN	The auxiliary "will" followed by the uninflected main verb.
I have walked	NP	The present tense of the auxiliary verb "have" followed by the main verb in past tense.
I had walked	PP	The past tense of the auxiliary verb followed by the main verb in past tense.
I will have walked	FP	The auxiliary "will", followed by the auxiliary "have" followed by the main verb in past tense.

Figure 2.1

Verb tenses

The tense markers are motivated by an analysis found in [Bruce 76]. Simplified, it says that a tense consists of a set of binary relations on a set of reference points. For instance, the tense of "had walked" consists of the relations on the three reference points: "the time of the speech" (S1), the "time of the subject" (S2), and the "time of the action" (S3). S2 is in the Past of S1, and S3 is in the Past of S2, so the tense of the verb group is Past-Past or PP. Similarly, the tense of "have walked" is Now-Past, or NP, since the "time of the subject" is the same (Now) as the "time of the speech" and the "time of the action" is in the Past of the "time of the subject". To see how this works, consider the sentences:

1. George, the club president, has walked through these halls. (NP)

2. George, the club president, walked through these halls. (PN)

in 1., the "time of the action" is in the past of the "time of the subject" so that we may not assume that George was president when he walked in these halls, but we do know that he is president now, since the time of the subject and speech are the same. In 2., the time of the action and subject are the same, so we know that George was president when he walked through these halls, but is not necessarily president now.

We get six more tense symbols by considering verb groups whose main verb ends in "ing". These tenses are represented by appending a "C" (continuing aspect) to the tenses above:

VERB GROUP	TENSE
I am walking	NNC
I was walking	PNC
I will be walking	FNC
I have been walking	NPC
I had been walking	PPC
I will have been walking	FPC

Figure 2.2

Tenses for verbs with a continuing aspect

When a verb is used as an infinitive, eg., "to hit" in "Bill wants to hit John", the tense marker returned is "INF". When a verb appears with an "ing" ending and no auxiliary verbs, as in "The man sitting on the chair...", the tense marker returned is "CC" (an arbitrary symbol). In terms of tense markers, passive constructions are indistinguishable (the order of the cases determines whether a construction is passive or not) from regular constructions, so the tense of "is walked" is equivalent

to the tense of "walks", namely NN. Verb groups consisting of the auxiliary verb "do" and an uninflected main verb (eg., "He did go...") are given the tense of the auxiliary "do".

We have left out tenses which require the verb "to go" as an auxiliary verb. The reason is that verb groups using "go" as an auxiliary are ambiguous. A verb group like "I am going to walk..." might mean either *"In the future some time, I will walk"* or *"I am actually going to some place (the beach, for example) in order to walk"*. Rather than try to resolve this ambiguity, Reader treats the infinitive as a case of the verb "go" and expects the deductive system to be aware of the possible ambiguity and to have enough information to resolve it. Therefore "I am going to walk" is parsed as

```
(GO NNC
   [SUB I]
   [INF (WALK INF
              [SUB !match_to_SUB]
         )]
)
```

The infinitive clause "to walk" is treated as a case of the verb "go" (INF). The system reading the parse must be aware that it can be interpreted as though the main verb were the verb of the INF case ("walk"), with a tense derived from the verb group "am going to walk". The SUB of "walk" is a dummy noun that should be matched to the SUB of "go" (I). The ambiguous situation is easy to recognize. It occurs whenever the main verb of clause is "go", and the clause has two cases, SUB and INF.

Some temporal information is contained in the cases of the verb rather than the tense. "I went yesterday" parses to

```
(GO PN
   [SUB I]
   [WHEN YESTERDAY]
)
```

so that the exact time in the past that the action occured in is specified by the WHEN case.

The verb "have" often occurs in verb groups as a modal. "I have to go away" essentially means "I must go away". When "have" is used as a modal, it is unambiguous. Therefore, when "..have to *verb*..." occurs as a verb group, Reader returns *verb* as the main verb, assigns it the tense of the verb "have", and places the marker "HAVE-TO" in its adverb slot. "I will have to leave" parses to:

```
(LEAVE FN (HAVE-TO)
   [SUB I]
)
```

This does not mean that every time the phrase "have to *verb*" appears in a sentence that "have to" will be treated as a modal. The noun phrase "The book I have to give" would be parsed into a three structure stack:

```
    - - -
    VERB: ((GIVE))
3.  FUNCTION: INF
    - - -
    VERB: ((HAVE))
    NOUN1: I
2.  FUNCTION: RC
    - - -
    NOUN1: (BOOK THE)
1.  FUNCTION: MAIN
    - - -
```

The stack can be interpreted in two different ways: *"The book I must give."* (3. attached to 2 attached to 1), or *"The book I have in my possession which I will give."*, (3. and 2. attached to 1. independently). Only the first interpretation treats "have to" as a modal.

The tense contains all the information in the sentence, yet leaves the decision of what to do with it for the system using the parser. For example, if the tense of a statement is NN the system can infer that a narrative is taking place, that the action described in the statement is habitual, etc.

2.3.3 Noun groups

Reader uses a different representation for noun groups than most parsers. To Reader, a noun group is a list whose first element is the head noun of the group, and whose remaining elements are the modifiers of the head noun. The difference in representation lies in the fact that Reader does not structure the modifiers that preceded the noun in the original sentence.

Therefore, "The messy green garbage can cover" is parsed as

[NOUN (COVER THE MESSY GREEN GARBAGE CAN)]

since Reader does not try to determine whether this means either

1. the cover of a can used for messy green garbage.
2. the messy cover of a can used for green garbage.
3. the messy green cover of a can used for garbage.
4. the messy cover of a green can used for garbage.
5. the cover of a messy green can used for garbage.
6. the cover of a messy can used for green garbage.

Instead, it allows the deductive system to structure the noun group when the stack entry containing the noun group is Formatted (section 4.3). This is necessary to avoid needless ambiguity. The sentence "A man people can trust is usually dangerous" can be parsed (correctly) as:

```
(BE NN (USUALLY)
   [SUB (MAN A (TRUST (NN CAN)
                      [SUB PEOPLE]
               ))]
   [DES DANGEROUS]
)
```

But unless the parser can discover from the system that there is unlikely to be "a man people can trust" (trust modified by can, people, man and the) it will also find

```
(BE NN (USUALLY)
   [SUB (TRUST A MAN PEOPLE CAN)]
   [DES DANGEROUS]
)
```

since "man", "can", and "people" are nouns, and therefore potential modifiers of "trust". The modifiers that followed the noun in the original sentence are structured by Reader, with help from the deductive system. This is necessary since Reader must know whether a sentence constituent coming after the noun modifies it, the verb the noun modifies, or some other constituent in the sentence. "The relation in the concept that is marked 'possible'." is parsed as:

```
[NOUN (RELATION THE (IN (CONCEPT THE))
                    (MARK PN
                          [OBJ THAT]
                          [IOB "POSSIBLE"]
                    ))]
```

in a context where the deductive system was able to determine that relations had markings and concepts did not, and as:

```
[NOUN (RELATION THE (IN CONCEPT THE (MARK PN
                                    [OBJ THAT]
                                    [IOB "POSSIBLE"]
                                ))))]
```

in a context where the deductive system thought that concepts were more likely to have markings than relations. The "closer" modification is also the preferred one in the absence of any information about whether concepts or relations have markings.

The point here is that each modifier (at top level in the noun group list) coming after the noun [5] modifies the noun independently.

Reader's technique of not structuring noun groups as they are encountered allows it to parse more efficiently than a parser that gets involved in the structure of noun groups immediately. Suppose we are given a sentence beginning with "The messy green garbage can cover...". A parser that started out by trying to parse for a structured noun group would immediately get bogged down trying to determine which of the six or more possibilities the phrase represented. It would have to call in the deductive system, which would then start looking for instances of green garbage, messy cans, etc. By delaying the structuring until later, Reader can provide the deductive system with more information (information including the main verb of the clause, its cases and the case of the unknown noun group) to guide its search in determining the structure of the noun group. And, if the entire sentence happened to be "The messy green garbage can cover the earth.", no time will ever be wasted structuring the noun group.

2.3.4 Choices

Occasionally, a sentence contains an ambiguous constituent whose ambiguity can be restricted to a small segment of the parse structure. When this happens, Reader returns one parse structure, and offers a choice between the ambiguous constituents. This leads to a more efficient parse, and enables the system reading the parse to compare the different meanings of the sentence easily, since the choice clearly shows where the parses differ. Here are two examples of this idea:

[5] The non pretty-printed version of the parser output contains a marker between the modifiers which come before and after the noun.

"I knew that ice was slippery." could mean either "I knew that *that* ice was slippery" or "I knew ice was slippery". If the deductive system is unable to determine which noun group it prefers at the time it is asked to structure the noun group, Reader would return the following parse, offering a choice for the SUB of "be".

```
(KNOW PN
     [SUB I]
     [WHAT (BE PN
               [SUB (*CHOICE  ICE
                              (ICE THAT)
                    )]
               [DES SLIPPERY]
          )]
)
```

"The man hitting Janet angered Bill" could mean either "The man who was hitting Janet angered Bill" or "The man's hitting of Janet angered Bill". Reader represents this as follows:

```
(ANGER PN
      [SUB (*CHOICE (HIT CC
                         [SUB (MAN THE)]
                         [OBJ JANET]
                    )
                    (MAN THE (HIT CC
                                   [SUB !match_to_head_noun]
                                   [OBJ JANET]
                             ))
           )]
      [OBJ BILL]
)
```

The first choice is the action "hit". The second choice is "man" modified by "the" and a verb clause with a dummy SUB (!match_to_head_noun) that should be matched to the noun it is modifying ("man"). In general, a choice can be offered as the contents of any case.

Another method Reader uses for representing ambiguous sentences is prefixing the

name of a case with an asterisk. This means that the case can modify either the verb or the noun in the case directly above it. "John hits the salesman with the hammer" is parsed to

```
(HIT NN
    [SUB JOHN]
    [OBJ (SALESMAN THE)]
    [*PREP (WITH (HAMMER THE))]
)
```

The asterisk preceding the case name "PREP" indicates that the PREP case could be a case of "hit" or that it could modify the salesman. The first interpretation is "The salesman was hit by John with the hammer" and the second is "The salesman with the hammer was hit by John". Reader uses the asterisk notation when running without a deductive system, or when running with a deductive system that cannot decide which interpretation is more likely at the time Reader asks. The parse would have been

```
(HIT NN
    [SUB JOHN]
    [OBJ (SALESMAN THE (WITH (HAMMER THE)))]
)
```

if the system was able to determine the salesman had the hammer when given the choice by Reader.

2.3.6 Conventions

Reader employs several notational conventions.

Whenever a conjunction contains an implied SUB, as in "The program reads the data and prints the answer" the implicit SUB is represented by the symbol "!match_to_conjunct_SUB". eg.,

```
[CONJ AND
  (READ NN
       [SUB (PROGRAM THE)]
       [OBJ (DATA THE)]
  )
  (PRINT NN
       [SUB !match_to_conjunct_SUB]
       [OBJ (ANSWER THE)]
  )
]
```

!match_to_conjunct_SUB has the same referent as "the program".

When a noun is modified by a relative clause, the case the noun occupies in the relative clause is held by the symbol !match_to_head_noun. For example,

"The man captured by the police."

```
[NOUN (MAN THE (CAPTURE PN
                        [OBJ !match_to_head_noun]
                        [SUB (POLICE THE)]
                ))]
```

"The man the police captured"

```
[NOUN (MAN THE (CAPTURE PN
                        [SUB (POLICE THE)]
                        [OBJ !match_to_head_noun]
                ))]
```

!match_to_head_noun has the same referent as "the man", the noun the verb clause is modifying.

!match_to_head_noun is also used in sentences which contain dangling prepositions. "The man I came with" parses to:

```
[NOUN (MAN THE (COME PN
                     [SUB I]
                     [PREP (WITH !match_to_head_noun)]
                ))]
```

!match_to_head_noun has the same referent as the noun ("the man") modified by the clause which contains the dangling preposition.

When a conjunction contains an implied object, Reader uses the symbol !match_to_conjunct_OBJ[6] to mark the second occurrence. "He breeds and raises rabbits" parses to:

```
[CONJ AND
  (BREEDS NN
        [SUB HE]
        [OBJ (RABBIT !PL)]
  )
  (RAISES NN
        [SUB !match_to_conjunct_SUB]
        [OBJ !match_to_conjunct_OBJ]
  )
]
```

In conjunctions in which the verb is omitted, Reader simply repeats[7] the verb. "He gave John a pencil and Janet a pen" parses to:

```
[CONJ AND
  (GIVE PN
      [SUB HE]
      [IOB JOHN]
      [OBJ (PENCIL A)]
  )
  (GIVE PN
      [SUB !match_to_conjunct_SUB]
      [IOB JANET]
      [OBJ (PEN A)]
  )
]
```

Suffixes are removed by the parser. If a word is a plural, the symbol !PL appears in its modifier list. "The answers" parses to:

```
[NOUN (ANSWER THE !PL)]
```

If a word can be either singular or plural, and agreement constrains it to be one or

---

[6] !match_to_conjunct_PREP is used when the OBJ refers to the object of a preposition in the higher conjunct.

[7] Nouns are represented by symbols (rather than being repeated) so that the interpreter will not have to find the referent of the same noun twice.

the other, it is noted by inserting !PL or !SING into the modifier list. "The fish is dangerous." and "The fish are dangerous" parse to:

```
(BE NN                          (BE NN
   [SUB (FISH THE !SING]           [SUB (FISH THE !PL]
   [DES DANGEROUS]                 [DES DANGEROUS]
)                               )
```

In "The fish can be dangerous", The SUB case is [SUB (FISH THE)] since there is no agreement information.

3. Grammar writing

This chapter explains how to write grammars in the formalism we have been discussing. The actual grammar is written in Lisp, and consists of a set of programs, one for each word class, which explain when and how a word may be added to a partial parse. The grammar also uses several utility programs and predicates.

An example of a utility program is ADD-NOUN. It takes two arguments, a noun group (ng) and a stack structure (s), and returns the stack structure with the noun group added to it. For example, if

ng = (MAN THE) and *s* is
VERB: ((SAVE ED))
NOUN1: (BOY THE)
FUNCTION: MAIN

then (ADD-NOUN *ng s*) is
VERB: ((SAVE ED))
NOUN1: (BOY THE)
NOUN2: (MAN THE)
FUNCTION: MAIN

An example of a predicate is CAN-ACCEPT-A-NOUN. It takes one argument, which is a structure, and returns T if the structure can accept a noun, and NIL otherwise. A structure can accept a noun if it is either

1. a preposition structure without a noun.

2. a verb structure without a noun

3. a verb structure with a verb and one noun whose verb is transitive. If the verb group is passive, the main verb must take a beneficiary or indirect object.

4. a verb structure with two nouns and a main verb that takes a beneficiary or indirect object. The verb group must not be passive.

3. and 4. must also satisfy the condition that the verb has not received any cases

since it was added to the structure[1] On the surface, it would seem that this definition would rule out slightly peculiar constructions like "That he likes", (instead of "He likes that") since a verbless verb structure with one noun cannot accept another noun. However, such constructions are handled as relative clauses.

Reader has other predicates which test for legal verb groups, whether a structure has a noun which can be modified by another structure, whether the verb group of a structure is passive or active, etc. When, in describing the actions of the parser, we say that a structure satisfies some condition, we mean that the proper predicate has been applied to that structure and that the test has succeeded.

Reader also has two programs, SHIFT and SEARCH, which are useful for manipulating the stack. SEARCH is used to search the stack for structures with a certain property. The information gained from a search is usually used to determine whether a particular structure should be pushed on to the stack. For instance, it would be pointless to push a relative clause structure (section 3.1.3) onto the stack if there were no structures in the stack that contained a noun which could be modified by a relative clause. SHIFT, described more fully in section 3.1.2, is used to facilitate the addition of words to structures other than the one at the top of the stack. Basically, SHIFT searches the stack for a given structure, collapses the stack down to that structure, and then applies the input word to the resulting stack. SHIFT is important because most actions that can be applied to the top of stack, such as adding in a noun or verb, can also be applied to structures lower down in the stack. Similarly, SEARCH is important because pushing a structure onto the stack usually depends on the existence of a structure with a given property, regardless of its position in the stack.

[1] Eg., "He spent in the store the money." is incorrect.

3.1 Some beginning grammars

A series of grammars is described, each one more complicated than the previous one. An example sentence is parsed for each grammar defined. The first two examples, Grammar.1 and Grammar.2, will step through the sentence in detail, examining how each successive word is applied to the partial parses formed by the application of the previous words in the sentence. The remainder of the examples will cover only the methods used to apply words that were not handled by the previously defined grammars.

Section 3.2 shows some more efficient methods for parsing the subset of English handled by the example grammars.

The variables used in the examples are:

stack	The stack.
word	The current input word
root	The root of *word*.
ending	The ending of *word*.
ml	The unassigned modifier list.
msg	The message concerning the top of the stack.
stack-msg	The message concerning the entire stack.

3.1.1 Grammar.1

The first grammar handles sentences of the form "noun verb noun noun" or "noun verb noun". All that is needed is a NOUN program and a VERB program.

The NOUN program:

The NOUN program forms the noun group consisting of the modifiers on the modifier list and the noun. Then, if the top structure in the stack can accept a noun (eg., satisfies the predicate CAN-ACCEPT-A-NOUN, defined at the begining of the chapter), a partial parse is created with:

msg = NOUN, indicating that the last addition to the stack was a noun.
ml = NIL, the modifier list is empty.
stack-msg = *stack-msg*, the addition of a noun doesn't change *stack-msg*.

stack = (REPLACE-TOP-STACK (ADD-NOUN (MAKE-NOUN-GROUP *word mf*)
(TOP-STACK *stack*))
stack)

where MAKE-NOUN-GROUP is a predicate which returns the noun group formed by its arguments (or NIL if one cannot be formed), and TOP-STACK and REPLACE-TOP-STACK are utility programs. TOP-STACK returns the top structure of the stack that is its argument. REPLACE-TOP-STACK returns the stack which is its second argument with the top structure replaced by its first argument.

The VERB program:

The VERB program examines the stack. If the top structure in the stack is a verb structure with one noun and no verb, it creates a partial parse by adding the verb to the top structure in the stack.

Here is how this grammar parses the sentence "John drinks water."

Reader starts out with the initial partial parse.

```
msg = BEGIN, mf = NIL
        - - -
      FUNCTION: MAIN
        - - -
```

"John" is input. It belongs to only one word class (NOUN), and therefore has only one program associated with it (NOUN). The partial parse produced by applying the NOUN program is:

```
msg = NOUN, mf = NIL
        - - -
      NOUN1: JOHN
      FUNCTION: MAIN
        - - -
```

"drinks" is the next word. It can be used as either a noun or verb. The top stack structure cannot accept a noun so the application of the noun program does not result in a continuation of the parse. The verb program is then applied to the parse which causes the following partial parse to be set up:

```
msg = VERB, mf = NIL
        - - -
      VERB: ((DRINK . 5))
      NOUN1: JOHN
      FUNCTION: MAIN
        - - -
```

"Water" can also be used as a noun or verb. The verb program fails though, since the top structure already has a verb. The NOUN program succeeds in continuing the parse by adding the noun "water" to the top structure in the stack, producing,

```
msg = NOUN, ml = NIL
        - - -
      VERB: ((DRINK . 5))
      NOUN1: JOHN
      NOUN2: WATER
      FUNCTION: MAIN
        - - -
```

The input sentence is exhausted so Reader collapses the stack, (trivial since there is only one structure in it), and formats the resulting structure. This yields

```
(DRINK NN
      [SUB JOHN]
      [OBJ WATER]
)
```

as the parse.

3.1.2 Grammar.2

In order to parse more interesting sentences, it is necessary to expand the grammar. The next grammar includes prepositions, articles and modifiers.

> The MODIFIER program simply adds *word* to *ml*.
>
> The ARTICLE program adds *word* (which is an article) to *ml* if *ml* is NIL or consists of words (almost, all, etc.) which can appear before an article.
>
> The PREPOSITION program checks to see whether the preposition can be modified by the modifiers on *ml*. If so, the partial parse is continued by pushing a preposition structure with *word* as the preposition onto the stack.

As the grammar grows, the grammar programs has to be prepared to handle stacks containing more than one structure. In general, there will be two parts to every grammar program: a set of actions associated with just the top of the stack and a

set of actions that should be applied to every structure in the stack that satisfies certain conditions. For example, in parsing "He gave the man in the store the book." a noun (the book) must be added to a structure (He gave the man) which is not at the top of the stack. Adding words to structures below the top of the stack is facilitated by the program SHIFT.

(SHIFT *stack program args purpose number predicate1 predicate2*)

The idea behind SHIFT is to find a structure(s) in the stack which satisfies a given predicate, (CAN-ACCEPT-A-NOUN, for example, would be used to search down the stack for a structure to add a noun to), then collapse the stack down to that structure, and then apply a program to the collapsed stack. SHIFT enables the grammar writer to specify the *purpose* of the collapse, which is valuable in guiding the way the collapse is carried out. For instance, if SHIFT is collapsing the stack of the sentence "He gave the man in the store ...", for the purpose of finding a structure which can accept a noun, it knows not to try to attach "in the store" to "gave", since that would prevent "gave" from accepting another noun.

SHIFT works as follows: It searches down *stack* looking for a structure S that satisfies *predicate1*. *stack* is then divided into two segments, S1 starting from the top of *stack* and going down to S, and S2 consisting of the structures not in S1. S1 is then collapsed into a single structure SS. If SS satisfies *predicate2*, then *program* is applied to (STACK-PUSH SS S1) with arguments equal to *args*. *number* controls how many times the sequence is performed. If *number* is an integer *n*, SHIFT tries to find the first *n* structures that satisfy *predicate1*. *number* = T means that shift finds all the stack structures satisfying *predicate1*. *purpose* is an atom (eg., NOUN means the collapse is looking for a structure which can accept a noun) which controls how structures can be attached to one another.

Grammar.2 involves adding a SHIFT to both the NOUN and VERB programs. The SHIFT in noun searches for all structures in the stack which can accept a noun, and then adds the word to that structure. The SHIFT in verb looks down the stack for the topmost verb structure in the stack which can accept a verb.

Grammar.2 can handle sentences like "The woman from the city bank gave the man in the store the news". The parse starts out with the initial parse. After "The" is input, there is one partial parse.

```
msg = BEGIN, sl = (THE)
        - - -
      FUNCTION: MAIN
        - - -
```

"woman" is read. MAKE-NOUN-GROUP forms the noun group, (WOMAN THE).

```
msg = NOUN, sl = NIL
        - - -
      NOUN1: (WOMAN THE)
      FUNCTION: MAIN
        - - -
```

"from" is read. The preposition program causes a preposition structure to be pushed on the stack.

```
msg = PREP, sl = NIL
        - - -
      PREP: FROM
        - - -
      NOUN1: (WOMAN THE)
      FUNCTION: MAIN
        - - -
```

"the" is read and placed on the modifier list. "city" is read. All nouns are treated as both NOUNs and MODIFIERs, so there are now two partial parses:

```
1. msg = NOUN, sl = NIL
           - - -
         PREP: FROM
         NOUN (CITY THE)
           - - -
         NOUN1: (WOMAN THE)
         FUNCTION: MAIN
           - - -
```

```
2. msg = PREP, sl = (CITY THE)
           - - -
         PREP: FROM
           - - -
         NOUN1: (WOMAN THE)
         FUNCTION: MAIN
           - - -
```

"bank" is read. When "bank" is applied as a verb, partial parse 2 can not be continued since "bank" (as a verb) does not accept the modifiers, (CITY THE), on the modifier list. Partial parse 1 cannot be continued using "bank" as a verb since after SHIFT finds a structure that can accept a verb, the verb "bank" fails to agree with the noun group (WOMAN THE). The agreement is tested using a predicate which takes a verb structure as input, and returns NIL if the structure does not exhibit agreement, and the structure modified by any information supplied by agreement (eg., "He saw" agrees only when "saw" is viewed as the past tense of "see", as opposed to the present tense of "saw".) when the structure does agree. Reader then applies "bank" to both partial parses as a noun. Partial parse 1 does not contain a structure that can accept a noun, so no partial parses can be continued from it. When "bank" is applied to the partial parse 2., it accepts the modifiers on the modifier list and is added to the top preposition structure, producing

```
msg = NOUN, ml = NIL
          - - -
          PREP: FROM
          NOUN (BANK THE CITY)
          - - -
          NOUN1: (WOMAN THE)
          FUNCTION: MAIN
          - - -
```

"gave" is read. The SHIFT program searches down the stack looking for the first structure that can accept a verb. It collapses the stack down to that structure and adds in the verb, which produces,

```
msg = VERB, ML = NIL
          - - -
          VERB: ((GIVE ED))
          NOUN1: (WOMAN THE (FROM (BANK THE CITY)))
          FUNCTION: MAIN
          - - -
```

"the" and "man" are read in and handled by the MODIFIER and NOUN programs. "man" is applied as both a noun and a modifier so two partial parses result:

```
1  msg = NOUN, ML = NIL                    2  MSG = NOUN, ML = (MAN THE)
   . . .                                      . . .
VERB: ((GIVE ED))                          VERB: ((GIVE ED))
NOUN1: (WOMAN THE (FROM (BANK THE CITY)))  NOUN1: (WOMAN THE (FROM (BANK THE CITY)))
NOUN2: (MAN THE)                           FUNCTION: MAIN
FUNCTION: MAIN                              . . .
 . . .
```

"in" is reed. The preposition program causes a preposition structure to be pushed on the stack of partial parse 1. Nothing is done with partial parse 2. since the preposition does not accept the modifiers, (MAN THE), on the modifier list.

```
msg = PREP, ML = NIL
        . . .
      PREP: IN
       . . .
      VERB: ((GIVE ED))
      NOUN1: (WOMAN THE (FROM (BANK THE CITY)))
      NOUN2: (MAN THE)
      FUNCTION: MAIN
       . . .
```

"the" and "store" are read. As before, two parses are created when "store" is read in. One in which the noun group "the store" becomes the noun of the preposition structure on the top of the stack, and another in which "store" is treated as a modifier. When "store" is tried as a verb it fails since it cannot accept "the" as a modifier. "the" is read in. In the former partial parse, it is simply added to the modifier list. In the latter, it cannot be added to the modifier list, since the modifier list contains a word (store) which cannot occur before an article.

```
msg = NOUN, ML = (THE)
        . . .
      PREP: IN
      NOUN: (STORE THE)
       . . .
      VERB: ((GIVE ED))
      NOUN1: (WOMAN THE (FROM (BANK THE CITY)))
      NOUN2: (MAN THE)
      FUNCTION: MAIN
       . . .
```

"news" is read. When it is applied as a noun, SHIFT searches for a structure on the stack that can accept a noun, collapses the stack to that structure, and then adds

in the noun group (NEWS THE). When "news" is tried as a modifier it is simply added to the modifier list.

```
1  msg = NOUN, al = NIL
 - - -
VERB  ((GIVE ED))
NOUN1  (WOMAN THE (FROM (BANK THE CITY)))
NOUN2  (MAN THE (IN (STORE THE)))
NOUN3  (NEWS THE)
FUNCTION  MAIN
 - - -
```

```
2  msg = NOUN, al = (NEWS THE)
 - - -
PREP  IN
NOUN  (STORE THE)
 - - -
VERB  ((GIVE ED))
NOUN1  (WOMAN THE (FROM (BANK THE CITY)))
NOUN2  (MAN THE)
FUNCTION  MAIN
 - - -
```

There are no more input words. Partial parse *2* is discarded since its modifier list is not empty. The stack from partial parse 1. is collapsed, (once again, this is trivial since there is only one structure in the stack.) and the resulting structure is formatted and returned as the parse of the sentence.

```
(GIVE PN
     [SUB (WOMAN THE (FROM (BANK THE CITY)))]
     [IOB (MAN THE (IN (STORE THE)))]
     [OBJ (NEWS THE)]
)
```

3.1.3 Grammar.3

Grammar.3 expands Grammar.2 by the inclusion of verb groups and relative clauses.

To parse relative clauses, a test is added to NOUN that checks to see if there is a structure in the stack which has a noun that can be modified, using the predicate CAN-NOUN-BE-MODIFIED. If the test succeeds, NOUN pushes a verb structure with function equal RC on the stack and adds the noun group to it. This addition enables the grammar to parse "The mirror on the wall he broke". The parse proceedes exactly as the previous ones until "he" is reached. The partial parse[2] when "he" is

[2] There are actually two partial parses. The second uses "wall" as a modifier and is discontinued since MAKE-NOUN-GROUP fails to make a noun group from "he" and (WALL THE).

read is

```
msg =  NOUN, a1 = NIL
          . . .
          PREP  ON
          NOUN  (WALL THE)
           . . .
          NOUN1  (MIRROR THE)
          FUNCTION  MAIN
           . . .
```

CAN-NOUN-BE-MODIFIED succeeds on the preposition structure on the top of the stack. Therefore a parse is created with a verb structure pushed on to the previous stack. Only one parse results from applying NOUN to the parse since when SHIFT is called, it cannot find a structure that can accept a noun.

```
msg =  NOUN, a1 = NIL
          . . .
          NOUN1  HE
          FUNCTION  RC
           . . .
          PREP  ON
          NOUN  (WALL THE)
           . . .
          NOUN1  (MIRROR THE)
          FUNCTION  MAIN
           . . .
```

"broke" is read. SHIFT is called to find a verb structure with an open verb slot. It finds the top structure in the stack, and creates a parse with the verb added in.

```
msg =  NOUN, a1 = NIL
          . . .
          VERB  ((BREAK ED))
          NOUN1  HE
          FUNCTION  RC
           . . .
          PREP  ON
          NOUN  (WALL THE)
           . . .
          NOUN1  (MIRROR THE)
          FUNCTION  MAIN
           . . .
```

The sentence is over, and the parse is concluded by the collapse of the stack. The deductive system must decide which of "the wall" or "the mirror" was broken. If we assume that "the mirror" was broken, the collapse of the stack would be,

```
 . . .
NOUN1  (MIRROR THE (OR (WALL THE)) (BREAK PB (SUB HE)
                                              (OBJ !match_to_head_noun)))
FUNCTION: MAIN
 . . .
```

The format of such a structure is simply the noun. Reader returns

```
[NOUN (MIRROR THE (OR (WALL THE))
                  (BREAK PB
                        [SUB HE]
                        [OBJ !match_to_head_noun]
                  ))]
```

as the parse. "Mirror" is the OBJ of the verb "break". Notice that the noun CAN-NOUN-BE-MODIFIED succeeded on was not the noun that was modified by the relative clause.

Parsing verb groups requires the addition of a test to VERB which tests that *msg* equals VERB. If the test succeeds, meaning that the last thing done to the stack was the addition of a verb, VERB tries to form a verb group with *word* and the verbs already in the top structure in the stack. If a legal verb group can be formed, (this is checked by the same predicate which tenses the verbs in a structure) the parse is continued by adding the verb into the verb group slot of the top structure in the stack. As an example, consider "He was given the prize". When "given" is read, there is one partial parse:

```
msg = VERB, a! = NIL
          . . .
          VERB: ((BE ED))
          NOUN1: HE
          FUNCTION: MAIN
           . . .
```

The msg is VERB and "was given" is a legal verb group so the parse is continued as:

```
msg = verb, a! = NIL
          . . .
          VERB: ((GIVE EN))(BE ED))
          NOUN1: HE
          FUNCTION: MAIN
           . . .
```

"The" and "prize" are read in. The stack is collapsed and formatted. The result is

```
(GIVE PN
     (IOB HE)
     (OBJ (PRIZE THE))
)
```

3.1.4 Grammar.4

Grammar.4 extends Grammar.3 in two ways.

The first addition is a test for time and place referents that will be placed in the NOUN program. This will enable the grammar to handle sentences like "I saw the man downtown.", "Yesterday John was in town." etc.

NOUN is augmented with a test which checks whether the noun-group can be used as a time or place (this is considered a syntactic property of the head noun of the group). If so, a preposition structure is created with preposition equal *TIME or *PLACE. The preposition structure is pushed onto the stack and a new partial parse created.

The second addition allows the parser to parse sentences with verbs that accept other verbs as case fillers. An example of a verb with this property is "see". In "I saw John leave town", the clause "John leave town", is a case of "saw". A test is added to VERB which checks whether the main verb of a structure can accept a clause. If so, an empty verb structure with function equal WHAT is pushed onto the stack and a new partial parse created.

Grammar.4 handles sentences like "Yesterday the man knew John had returned."

"Yesterday" causes the formation of two partial parses, one in which it is treated

as a time referent, and one in which it is used as the first noun of the MAIN structure.

```
1. msg = NOUN, ml = NIL
     - - -
   NOUN1: YESTERDAY
   FUNCTION: MAIN
    - - -
```

```
2. msg = NOUN, ML = NIL
     - - -
   PREP: *TIME
   NOUN: YESTERDAY
    - - -
   FUNCTION: MAIN
    - - -
```

When "man" is input, it cannot be added to partial parse 1, since there is no structure in the stack that can accept a noun. "man" can be added to partial parse 2, by collapsing the stack down to the MAIN structure and adding "man" to the MAIN structure. This results in

```
msg = NOUN, ml = NIL
        - - -
      NOUN1: (MAN THE)
      CASES: ((WHEN YESTERDAY))
      FUNCTION: MAIN
       - - -
```

as the COLLAPSE routine knows that preposition structures whose preposition is *TIME fill the WHEN case of the verbs they modify.

"Know" can accept a clause, so the application of "knew" to the partial parse above results in two different partial parses:

```
1. msg = VERB, ml = NIL
        - - -
      VERB: ((KNOW ED))
      NOUN1: (MAN THE)
      CASES: ((WHEN YESTERDAY))
      FUNCTION: MAIN
       - - -
```

```
2. msg = NIL, ML = NIL
     - - -
   FUNCTION: WHAT
    - - -
   VERB: ((KNOW ED))
   NOUN1: (MAN THE)
   CASES: ((WHEN YESTERDAY))
   FUNCTION: MAIN
    - - -
```

"John" is added to both partial parses:

```
1. msg = NOUN, ml = NIL
          - - -
         VERB: ((KNOW ED))
         NOUN1: (MAN THE)
         NOUN2: JOHN
         CASES: ((WHEN YESTERDAY))
         FUNCTION: MAIN
          - - -
```

```
2. msg = NOUN, ML = NIL
          - - -
         NOUN1: JOHN
         FUNCTION: WHAT
          - - -
         VERB: ((KNOW ED))
         NOUN1: (MAN THE)
         CASES: ((WHEN YESTERDAY))
         FUNCTION: MAIN
          - - -
```

"had" is applied to each partial parse as verb. Partial parse 2 is continued by adding "had" to the top structure of the stack. Partial parse 1 cannot be continued.

The addition of "returned" to the stack produced by the application of "had" produces,

```
msg = VERB, ML = NIL
          - - -
         VERB: ((RETURN ED)(HAS ED))
         NOUN1: JACK
         FUNCTION: WHAT
          - - -
         VERB: ((KNOW ED))
         NOUN1: (MAN THE)
         CASES: ((WHEN YESTERDAY))
         FUNCTION: MAIN
          - - -
```

The input sentence is exhausted. The stack is collapsed and the resulting structure formatted.

```
(KNOW PN
     [WHEN YESTERDAY]
     [SUB (MAN THE)]
     [WHAT (RETURN PP
                    [SUB JACK]
           )]
)
```

3.2 Grammar efficiency

The primary objective in writing an efficient grammar is keeping the number of partial parses low. This is accomplished by minimizing the number of ways a word can be successfully applied to a partial parse. There are basically three different ways of handling this within the Reader formalism.

> R1. The use of the stack to avoid attaching sentence constituents to each other until more information is learned about the nature of the attachment.
>
> R2. The use of one stack structure to represent more than one syntactic possibility.
>
> R3. The use of bottom-up and top-down parsing techniques together.

The simplest example of the first technique is the handling of sentence constituents which can modify many different structures in the sentence (eg., prepositional phrases, relative clauses, etc.). Such constituents are placed on the stack, thereby avoiding the necessity of a different parse path for each sentence structure that can accept them as a modifier. Woods, in [Woods 73], mentions a similar feature, called "selective modifier placement". However, it seems limited to the simple application mentioned above. More powerful uses of the stack are obtained in conjunction with R2.

R2 makes use of the fact that in many cases, two or more syntactic possibilities can be combined in a single parse structure. For example, consider a sentence beginning "The boy that..." Obviously, "that" is part of a relative clause which will modify "boy". But it is not clear whether "that" is either

> 1. the subject of the relative clause ("The boy that likes ice cream...")
>
> 2. a modifier of the subject of the relative clause ("The boy that girl likes...")

3. a function word ("The boy that the girl likes...").

A single stack entry which covers all these possibilities is

```
        - - -
S =  NOUN1: THAT
     FUNCTION: RC
        - - -
```

If a verb is applied to the stack containing S before a noun is applied, S will lead to a successful parse. Now suppose a noun is applied before a verb. If a noun group can be made from "that", the modifiers on the modifier list, and the noun being added, then the sentence involves usage 2, and "that" is replaced by the noun group[3]. If a noun group cannot be constructed using "that", but can be made using just the modifier list and the noun, then "that" is replaced by the noun group (usage 3.).

R2 can be used with R1 in a slightly different way. Consider the two sentences:

1. "He saw the man running out the door."
2. "He saw the man running out the door drop the bag."

In sentence 1., "running out the door" is most likely interpreted as "what he saw the man doing". In sentence 2., "running out the door" is a relative clause which modifies "man". One structure,

```
          - - -
S = VERB ((RUN ING))
    FUNCTION: PARTICIPLE
          - - -
```

can represent both interpretations. It is decided which interpretation to use depending on the conditions under which the stack is collapsed. The relative clause

[3] If a noun group could also be made without using "that", a message is left which indicates to Format that a choice between "that *noun-group*" and *noun-group* should be offered.

interpretation is used if the stack is being collapsed to add a verb, and the "see" case filler interpretation is used otherwise. A more detailed example can be found in section 3.2.5.

Section 3.2.3 provides an example of R3. The following two sections contain examples of R2.

3.2.1 Nouns as modifiers

Virtually all English nouns can also be used as modifiers. In "The baseball bat is used to hit the baseball", the first occurrence of "baseball" is used as a modifier, while the second is used as a noun. The grammars in section 3.1.1 coped with this by applying each noun to every possible partial parse as both a noun and a modifier. The example sentence would have two partial parses after "baseball" was read.

```
1. msg = NOUN, ml = NIL          2. msg = BEGIN, ml = (BASEBALL THE)
   - - -                                 - - -
  NOUN1: (BASEBALL THE)                 FUNCTION: MAIN
  FUNCTION: MAIN                         - - -
   - - -
```

It is true that one of the two parses will always be killed rather quickly, but it would be better to avoid the overhead involved in carrying extra partial parses. As a noun cannot modify a verb, there is no advantage to be gained from putting one on the modifier list. When a noun acts as a modifier, it modifies one of the nouns that come directly after it in the sentence. The second parse can be eliminated by adding a test to the NOUN program that checks for:

> 1. *msg* = NOUN (meaning the last thing done to the stack was the addition of a noun group to the top structure)
>
> 2. the noun group consisting of *word* and the words in the last noun group added to the top structure in the stack is a legal noun group.

If the test succeeds, the last noun group added to the top structure in the stack is replaced by the noun group consisting of word with the words in the replaced noun group as its modifiers. Under this scheme, there would be only one partial parse for a sentence beginning "The baseball..." (parse 1, shown above). If the next word in the sentence were "bat", its application to parse 1 would result in

```
msg = NOUN, el = NIL
 - - -
NOUN1: (BAT THE BASEBALL)
FUNCTION: MAIN
 - - -
```

since parse 1 meets the requirement of msg = NOUN and "the baseball bat" is a legal[4] noun group.

3.2.2 Relative clauses

Grammar.3 (section 3.1.3) parses relative clauses in essentially a top down fashion. When a noun is read, and the stack contains a structure with a noun which could be modified by a relative clause, a verb structure with function equal RC is created, the noun is added to it, and the resulting structure is pushed onto the stack to await the verb of the relative clause. If a sentence began "The city people..." after "people" was read there would be two partial parses:

```
1. msg = NOUN, el = NIL
 - - -
NOUN1: (PEOPLE THE CITY)
FUNCTION: MAIN
 - - -
```

```
2. msg = NOUN, el = NIL
 - - -
NOUN1: PEOPLE
FUNCTION: RC
 - - -
NOUN1: (CITY THE)
FUNCTION: MAIN
 - - -
```

If the complete sentence were "The city people hate is Tokyo." the second partial

[4] The test would fail if the sentence were "The baseballs bat ..." since "the baseballs bat" is not a legal noun group".

parse would lead to a parse. "hate" would be the verb of the "RC" verb structure and "is" would be the verb of the "MAIN" structure. Parse 1 would use "hate" as the verb of the "MAIN" structure and the parse would be discontinued after "is" is read, since the stack would not contain a verb structure which could accept "is". If the complete sentence was "The city people favor bonds.", partial parse 1 would lead to a parse. Parse 2 would be discontinued when the end of the sentence is reached and the parser realizes that it cannot attach "people favor bonds" to "the city". If the main verb of a sentence which begins with a such a compound noun takes an indirect object, then the sentence is syntactically ambiguous. (eg., "The city people gave the bonds") The parser must not refuse to add "bonds" to "people favor" (which would kill the parse earlier) since the sentence might have been "The city people favor bonds for is Tokyo."

This splitting can be avoided by making changes in the NOUN and VERB program. In the previous section, a test was added to NOUN which determined when it was possible to replace the last noun group added to a structure with the noun group consisting of *word* and the words in the old noun group. If that test succeeds, and *word* is a legal noun group by itself, then instead of parsing for a possible relative clause in a new partial parse (by pushing a verb structure whose function is RC onto the stack), a message is inserted in the message slot of the top structure explaining that it is possible to form a relative clause with the head noun of the last noun group in the structure. In VERB, the method used to find an empty verb slot is modified so that if no structure can be found with an empty verb slot, VERB tries to find a structure whose message is "Possible RC".

These changes allow "The city people hate is Tokyo." to be parsed using only one parse path. After "hate" is read, there is one partial parse:

```
msg = VERB, al = NIL
 - - -
VERB: ((HATE))
NOUN1: (PEOPLE THE CITY)
MESSAGE: POSSIBLE-RC
FUNCTION: MAIN
 - - -
```

VERB tries to find an open verb slot to put "is" in. It can't find one, but it is able to find a stack structure whose message is POSSIBLE-RC. It removes the message, verb and head noun from the structure, forms a new verb structure, and places it in the stack just above the old one. This forms a new stack,

```
 - - -
VERB: ((HATE))
NOUN1: PEOPLE
FUNCTION: RC
 - - -
NOUN1: (CITY THE)
FUNCTION: MAIN
 - - -
```

which is has a place for the verb "is".

3.2.3 Verbs which accept clauses

Grammar.4 (section 3.1.4) showed one way of handling verbs which can accept clauses as case fillers. Like the first relative clause mechanism, it was essentially top down. When a verb that was able to accept a clause was added to a structure, a second partial parse was created with an empty verb structure whose function was WHAT pushed onto the stack. A better method is to wait for the verb of the clause to arrive before sprouting another partial parse. "I saw the man in the store steal the book." would then have one partial parse at the time "steal" was read:

```
msg = NOUN, al = NIL
        - - -
        PREP: IN
    2. NOUN: (STORE THE)
        - - -
        VERB: ((SEE-SAW))
        NOUN1: I
        NOUN2: (MAN THE)
    1. FUNCTION: MAIN
        - - -
```

"See-saw" is the verb used by Reader to represent either the past tense of "see" or the present tense of "saw". It has all the syntactic properties of both. If something in the parse resolves which verb is intended, Reader makes the change. When "steal" is read, VERB looks down the stack for a structure that can accept a verb. It finds structure 1., which has a verb, "see-saw", that can accept a clause. The stack is collapsed down to structure 1., yielding

```
        - - -
        VERB: ((SEE-SAW))
        NOUN1: I
        NOUN2: (MAN THE (IN (STORE THE)))
    1   FUNCTION: MAIN
        - - -
```

A verb structure with function equal WHAT is created to hold "steal". NOUN2 is removed from structure 1., and placed in the new structure, which is pushed onto the top of the stack. The verb "see-saw" has been changed to "see" by the program which pushed the WHAT structure onto the stack, since "saw" cannot accept a clause. The result is:

```
        - - -
        VERB: ((STEAL))
        NOUN1: (MAN THE (IN (STORE THE)))
    2.  FUNCTION: WHAT
        - - -
        VERB: ((SEE ED))
        NOUN1: I
    1.  FUNCTION: MAIN
        - - -
```

3.2.4 Conjunctions

Conjunctions are similar to other sentence constituents in that, syntactically, they usually can be attached to more than one sentence constituent. For example,

"The man in the suit and tie." (*suit* and *tie* form the conjunction.)

"The man in the suit and John." (*man* and *John* form the conjunction.)

"Bill bought the turntable John was selling because he needed the money."
("because he needed the money" specifies why "John was selling".)

"Bill bought the turntable John was selling because he liked the way it sounded."
("because he liked the way it sounded" specifies why "Bill bought".)

Ambiguities arising from which constituent the conjunction should be attached to are handled by the stack and COLLAPSE. "The man in the suit and John" would be parsed into the stack,

```
   . . .
   PREPOSITION: AND
3. NOUN: JOHN
   . . .
   PREPOSITION: IN
2. NOUN: (SUIT THE)
   . . .
   NOUN1: (MAN THE)
1. FUNCTION: MAIN
   . . .
```

"And" (when acting as a conjunction between nouns) is treated as a preposition syntactically. When the stack is collapsed, it is determined whether 3. should be attached to 1. or 2.

Conjunctions between verbs are handled by pushing a verb structure whose function is the conjunction onto the stack. "Bill bought the turntable John was selling because he needed the money." would be parsed into:

```
    - - -
    VERB: ((NEED ED))
    NOUN1: HE
    NOUN2: (MONEY THE)
3. FUNCTION: BECAUSE
    - - -
    VERB: ((SELL ING)(BE ED))
    NOUN1: JOHN
2. FUNCTION: RC
    - - -
    VERB: ((BUY ED))
    NOUN1: BILL
    NOUN2: (TURNTABLE THE)
1. FUNCTION: MAIN
    - - -
```

When the stack is Collapsed, it is determined (by the interpreter, acting through Format) whether 3. modifies 2. or 1.

At first glance, it would appear that the application of a conjunction that can conjoin nouns and verbs (or a conjunction that is also a preposition, eg., before, like) to a parse will result in two partial parses: one in which a verb clause is expected (a verb structure is pushed on the stack), and one in which just a noun is anticipated (a preposition structure is pushed on the stack). However, both expectations can be handled by pushing on a verb structure[5] whose message is POSSIBLE-PREP and modifying Format so that it formats a verb structure whose message is POSSIBLE-PREP and whose verb slot is empty as if it were a preposition structure whose preposition is function and whose noun slot is the value of the noun1 slot of the verb structure. Also, VERB has to be modified to search for empty verb structures down the stack past those verb structures whose message is POSSIBLE-PREP.

Using this method, the stack for "John likes Janet and Bill ..." would be

[5] Assuming the stack can accept a verb conjunction. The stack for the sentence beginning "John and ..." can only accept "and" as a noun conjunction. The general condition is that a stack cannot accept a verb conjunction if the top most verb structure whose message is not POSSIBLE-PREP does not contain a verb. If the stack cannot accept a verb conjunction then the parse is continued by pushing a preposition structure on the stack.

```
    . . .
    NOUN1: BILL
    MESSAGE: POSSIBLE-PREP
2. FUNCTION: AND
    . . .
    VERB: ((LIKE 5))
    NOUN1: JOHN
    NOUN2: JANET
1. FUNCTION: MAIN
    . . .
```

If the sentence continued "John likes Janet and Bill hates Jill", "hates" would be placed in the verb slot of structure 2. If the sentence was simply "John likes Janet and Bill", the stack would be collapsed and the format of structure 2. would be

```
(AND BILL)
```

the same as the format of the preposition structure,

```
PREPOSITION: AND
NOUN: BILL
```

Finally, if the sentence were "John likes Janet and Bill and George hate Jill.", "hate" would be applied to the following stack:

```
    . . .
    NOUN1: GEORGE
    MESSAGE: POSSIBLE-PREP
3. FUNCTION: AND
    . . .
    NOUN1: BILL
    MESSAGE: POSSIBLE-PREP
2. FUNCTION: AND
    . . .
    VERB: ((LIKE 5))
    NOUN1: JOHN
    NOUN2: JANET
1. FUNCTION: MAIN
    . . .
```

VERB would first try to add "hate" to structure 3. This would fail since "hate" and "George" do not agree. It would then try to add "hate" to structure 2., after having attached 3. This would succeed since "hate" and (BILL (AND GEORGE)) do agree. Note that if "hate" could have been added to structure 3. (if the sentence were "John likes Janet and Bill and the children hate Jill.", for instance) then VERB would

still have tried to attach "hate" to a structure lower down in the stack so that all the possible meanings of the sentence could be uncovered. "John likes Janet and Bill and the children hate Jill." could mean either

```
[CONJ AND
    (LIKE NN
        [SUB JOHN]
        [OBJ JANET]
    )
    (HATE NN
        [SUB (AND BILL
                  (CHILD !PL)
             )]
        [OBJ JILL]
    )
]
```

or

```
[CONJ AND
    (LIKE NN
        [SUB JOHN]
        [OBJ (AND JANET
                  BILL
             )]
    )
    (HATE NN
        [SUB (CHILD !PL)]
        [OBJ JILL]
    )
]
```

In producing the two parses above, Reader did not have to split into two parses until the word "hate" was encountered.

3.2.5 Verbs inflected with ed endings

Verbs inflected with an "ed" ending which are not preceded by auxiliary verbs can usually be applied to a parse (as verbs) in two different ways: as the main verb of a clause, "The police captured the robber.", or as a modifier following a noun, "The robber captured by the police was convicted". The grammar Reader uses combines the two possibilities into one.

When an "ed" verb is encountered, any combination of

1. There is a verb structure in the stack that has an empty verb slot.

2. There is a structure in the stack that has a noun which could be modified by a relative clause.

can be true. Suppose an "ed" verb is encountered.

If the last operation on the stack was the addition of a verb (msg = VERB), and the "ed" verb forms a legal verb group with the verb just added, it is added into the top structure in the stack as part of the verb group. VERB exits.

If 1 and 2. are true, then a verb structure is pushed on to the stack with **FUNCTION** equal REL-OR-MAIN, **VERB** equal the "ed" verb, and **NOUN1** equal !match_to_head_noun. If the verb clause is used as the predicate of the sentence, then !match_to_head_noun will be replaced by the NOUN1 of the structure it is added to.

If just 2. is true, then a verb structure is pushed on the stack with **FUNCTION** equal REL.

If just 1. is true, the stack is collapsed down to the structure with the empty verb slot, and the verb is added.

If neither 1. or 2. is true, then VERB simply exits. The parse will be continued by using the "ed" verb as a modifier.

These methods parse "The man in the photograph framed for the police was his father". as follows. The stack, before "framed" is read and after "police" is read, is shown below:

```
                                        . . .
                                        PREP  FOR
                                    4   NOUN (POLICE THE)
                                        . . .
                                        VERB  ((FRAME ED))
                                        NOUN1  !match-to-head-noun
                                    3   FUNCTION  REL-OR-MAIN
    . . .                               . . .
    PREP  IN                            PREP  IN
2   NOUN  (PHOTOGRAPH THE)          2   NOUN  (PHOTOGRAPH THE)
    . . .                               . . .
    NOUN1  (MAN THE)                    NOUN1  (MAN THE)
1   FUNCTION  MAIN                  1   FUNCTION  MAIN
    . . .                               . . .
"The man in the photograph          framed for the police. . "
```

A verb structure with **FUNCTION** equal REL-OR-MAIN has been pushed on, since the stack contains both a structure with an empty verb slot (1) and one (both 1. and 2.) with a noun which could be modified by a relative clause. If the sentence ended

with "police", the stack would be collapsed, and the deductive system would be asked to choose from among the three possible parses the stack could be collapsed to:

```
"The man in the photograph which was framed for the police."

(NOUN (MAN THE (IN (PHOTOGRAPH THE (FRAME PN
                                      [OBJ !match_to_head_noun]
                                      [FOR (FOR (POLICE THE))]
                                    )))))

"The man in the photograph who was framed for the police."

(NOUN (MAN THE (IN (PHOTOGRAPH THE))   (FRAME PN
                                         [OBJ !match_to_head_noun]
                                         [FOR (POLICE THE)]
                                       )))

"The man in the photograph did frame (photos or people) for the police."

(FRAME PN
      [SUB (THE MAN (IN (PHOTOGRAPH THE)))]
      [FOR (FOR (POLICE THE))]
)
```

The sentence continues with "was", however. The VERB program applies "was" to the stack by searching down the stack for a structure with an empty verb slot. It finds 1., and collapses the stack with the *purpose* of inserting a verb. This means that 3. cannot be attached to 1. as the main verb of the sentence, since that slot is now reserved for "was". The deductive system decides whether the man or photograph was framed (we will assume "the man"), and "was" is inserted in the resulting structure. This yields

```
 - - -
VERB:  ((BE ED))
NOUN1: (MAN THE (IN (PHOTOGRAPH THE))
                {V FRAME PN (OBJ !match_to_head_noun)
                            (FOR (FOR (POLICE THE)))))
FUNCTION: MAIN
 - - -
```

and the parse is continued. In the parse of the complete sentence, the companion

system never had to consider a meaning which used "the man" as the SUB of "frame".

4. A closer look

This chapter explains some of the algorithms mentioned earlier in greater detail.

4.1 Measure

Each stack structure has a slot set aside for its measure, which is used by Reader to help it choose among competing partial parses. The measure of a structure rates both the syntax and semantics of the structure. The deductive system (via Format) is responsible for determining the semantic component of a structure's measure. Section 5.5 explains how semantic measure is calculated in the Reader-Interpreter system.

Two measures are compared by first comparing the two semantic components. If one measure has a better semantic rating (section 4.1.1) than the other, it is preferred. If the semantic components are equal, the measure with the best syntax rating (section 4.1.2) is preferred. If both components are equal, the measures are equal. This comparison system prefers a very unusual (but legal) syntactic structure to a more common syntactic structure if the former is judged to be even slightly better semantically.

A structure is measured when it is Formatted. Format returns the format of the structure as well as its measure, which is then merged[1] with the contents of the **measure** slot of the structure receiving the formatted structure. The measure of a

[1] The merge of two measures, M1 and M2, is the measure whose semantic and syntactic components are the union of the semantic and syntactic components of M1 and M2.

structure, therefore, contains the measure of all the structures that have been attached to it.

4.1.1 The semantic component

The semantic component consists of three features. The interpreter is responsible for rating each feature. A rating can have one of 3 values.

perfect: The interpreter is perfectly satisfied with this feature.

acceptable: The interpreter would prefer something else but the feature is acceptable.

unacceptable: The feature is unacceptable.

A semantic component *A* is better than a semantic component *B* if

1. *A* has fewer unacceptable features than *B*.

or

2. *A* and *B* have the same number of unacceptable features, and *A* has fewer features which are merely acceptable.

This algorithm would prefer a semantic component with only acceptable features to a component with one unacceptable feature and a large number of perfect features. An alternative method is to allow some number of perfect features to cancel the effects of an unacceptable feature.

The following features contribute to the semantic component.

Verb Cases

Is the verb well modified? The ratings are:

perfect: The verb has all the cases it needs to be well defined.

acceptable: The verb is missing some cases which are usually found with it.

unacceptable: The verb is missing some cases which are necessary.

"Put" is an example of a verb requiring a case; namely a where-put case. One almost never says "John put the ball". Therefore a verb structure whose main verb was "put" that did not have a where-put case would be rated unacceptable. This does not prohibit the parser from parsing a sentence like "John put the ball". If that were the sentence the parser was given, then the best structure the parser would be able to find would be one whose measure contained a semantic component with at least one unacceptable rating.

An acceptable, but not perfect, case of verb modification can occur with verbs like "go". "Go" prefers a case explaining where the SUB has gone. However it is fairly common to omit that case if it is implicit from some other information.

Noun Modifications

This is an evaluation of the appropriateness of each noun group in the structure. The ratings assigned are,

perfect: The noun group is perfect. The deductive system can find an object in its representation of what has been said which the noun group refers to.

acceptable: A referent cannot be found, but all the modifications in the noun group are meaningful to the deductive system. eg., The deductive system will know how to interpret the noun group.

unacceptable: The deductive system cannot understand the proposed modifications.

Sometimes the rating given a noun group will depend on the context the sentence containing the noun group occurs in. Consider the noun group "The student George". If there were two George's and one of them was known to be a student,

one might want to disambiguate which George was being referred to by using the phrase, "the student George"; as in "The student George is always busy". However we would not want the parser to consider the phrase "the student George saw" as having a meaning other than "the student that George saw", except in such a context.

This feature is also responsible for measuring the fit of the modifiers coming after the noun. "The ball in the box" would be rated perfect if the interpreter could find a ball in the box, acceptable if not. "The store he kissed" would be rated perfect if the interpreter could locate a store that was kissed, unacceptable if not.

Appropriateness of Verb Cases

Most verbs prefer certain types to fill their cases. The interpreter should have a verb frame for each verb [Reader can operate without this frame; it just means that one more level of discrimination is lost, which might result in Reader finding more interpretations of a sentence than a person would] which it uses to evaluate how well the verb's cases fit it. The values are,

> perfect: The verb and case satisfy the interpreter's expectations.
>
> acceptable: The verb does not usually contain the case, but the interpreter is aware of idioms that would cause the verb to receive it.
>
> unacceptable: The interpreter is unable to find any role for the case to play in the verb's definition.

The verb "give" prefers a human as its SUB, a non-human as its OBJ and a human as its IOB (recipient). Using these expectations enables a person to find only one meaning for "He gave the ball Bill gave the salesman", namely

```
(GIVE PN
     [SUB HE]
     [OBJ (BALL THE (GIVE PN
                          [SUB BILL]
                          [IOB (SALESMAN THE)]
                    ))]
)
```

and not consider,

```
(GIVE PN
     [SUB HE]
     [IOB (BALL THE (GIVE PN
                          [SUB BILL]
                    ))]
     [OBJ (SALESMAN THE)]
)
```

since the second interpretation assigns "give" a non-human for its IOB case and a human for its OBJ.

A parser cannot afford to reject possible parses that contain verbs that don't accept their cases since one frequently uses verbs in ways which violate their case preferences, as in "He gave the bride away", "The noise gives him a headache" or "He gave the wall a kick".

4.1.2 The Syntactic Component

Reader tries to filter out some of the partial parses that are valid syntactically, semantically meaningful, and yet would not be selected by a person. If a structure has this property, it is marked in the syntactic component of its measure. The syntactic component with the fewest such markings is the best. A structure inherits the measure of any structure that is attached to it, so it is possible for the syntactic component of the measure of a structure to have more than one syntactic mark against it. Here is an example of this idea:

"The salesman crushed by the elevator was hurt" is understood by realizing that the verb phrase, "the salesman crushed by the elevator" is the subject of was. Using the same methods Reader finds two meanings to "I saw the salesman crushed".

The only meaning most people would consider is, M1: *"I saw the act of salesman being crushed"*,

```
(SEE PN
    (SUB I)
    (WHAT (CRUSH PN
                [OBJ (SALESMAN THE)]
          )]
)
```

Reader finds another interpretation, which is M2: *"I saw the salesman who was crushed"*

```
(SEE PN
    (SUB I)
    (OBJ (SALESMAN THE (CRUSH PN
                            [OBJ !match_to_head_noun]
                       )))]
)
```

People who want to convey the second meaning say the sentence differently, so we do not want the parser to return with two parses for "I saw the salesman crushed" since people do not find it ambiguous. The second meaning has to be considered, since the parser may be given "I saw the salesman crushed by the elevator walk away unhurt". Reader marks the syntactic component of any verb structure whose verb can accept a clause and whose OBJ is a noun modified by a verb clause with !match_to_head_noun for a dummy OBJ. Thus, if Reader were given the example sentence "I saw the salesman crushed", M1 would have a better measure than M2, so Reader would return only one parse for the sentence.

It should be noted that the rules used in determining the measure of a structure are distinct from the rules used in the grammar. The rule used in the above example ("...mark any verb structure whose verb can accept a clause, and whose OBJ is a noun modified by a verb clause with !match_to_head_noun for an OBJ") may seem somewhat ad-hoc. But this rule in no way effects the structuring of an input sentence. It is merely used to filter structures that the parser finds. Without this rule, the system working with the parser would have to decide for itself whether "I saw the salesman crushed" meant M1 or M2.

Other parsers have used variants of a "measure" concept. Robinson, [Robinson 75], uses the term *factor score* to refer to how well various syntactic features "fit" together. In theory, this seems quite similar to the syntactic component just defined. In practice, it is used quite differently, since the motivation for *factor scores* lies in the ambiguous inputs a speech parser must deal with. Reader uses the measure of a structure to help it choose from among completed parse structures, or from among structures resulting from the collapse of a stack segment. Measure is never used to determine how a word should be applied to a parse, or whether or not to continue a parse. In contrast, *factor scores* are primarily used to determine the priority of active parse paths. The *factor score* of "out" eliminates a parse path. An example of an "out" *factor score* is the combination of "foot" and "s". Presumably, the speaker intended the "s" as the first letter of the word following "foot", rather than the last letter of the incorrect plural "foots". This level of detail is unnecessary in a parser intended for written input.

In many cases, the syntactic measure can be done away with in favor of more efficient parsing methods. In the example above, syntactic measure is needed

whenever the grammar "splits" on a verb inflected with "ed" by creating a parse in which the "ed" verb is the main verb of a clause, and one in which the "ed" verb is part of an embedded clause modifying a noun. In a grammar which did not split (see section 3.2.5), "I saw the salesman crushed by the elevator" would be divided into:

```
    . . .
    PREP  BY
3   NOUN  (ELEVATOR THE)
    . . .
    VERB  ((CRUSH ED))
    NOUN1  !match_to_head_noun
2   FUNCTION  REL
    . . .
    VERB  ((SEE ED))
    NOUN1  1
    NOUN2  (MAN THE)
1   FUNCTION  MAIN
    . . .
```

When the stack is collapsed, 2. would be attached to 1. as the WHAT case of "see", and !match_to_head_noun would be replaced by "the man". If the sentence were "I saw the man crushed by the elevator walk away.", then when walk was "read", the only place to put it would be the verb slot of the WHAT case of "see". Therefore the stack would be collapsed with the purpose "VERB", meaning, "Don't fill up any verb slots." This would cause 2. to be attached to 1. as a modifier of "man", rather than as the WHAT case of "see".

4.2 Collapsing

Collapsing a stack (or stack segment) consists of converting it into a single stack structure by attaching all the structures in the stack to each other until there is only one left that has not been attached to any other. The methods used to build the stack ensure that structures will only modify structures beneath them in the

stack. There is one "syntactic" constraint the collapse must satisfy. Given a stack [Sn, Sn-1,.....S2, S1], if Sk is attached to Sj, then for all i, k > i > j, Si cannot be attached to Sm, j > m. This constraint, which may be viewed as nesting condition, reflects the syntax of English. As an illustration, the stack [D C B A] could be collapsed in five different ways:

(A B C D)	A modified independently by B, C and D.
(A B (C D))	A modified independently by B, and C modified by D.
(A (B C D))	A modified by B modified independently by C and D.
(A (B (C D)))	A modified by B modified by C modified by D.
(A (B C) D)	A modified by independently by B modified by C, and D.

It can't be collapsed so that D modifies B, which then modifies A, and C modifies A since this would violate the nesting condition.

Depending on the stack, each one of the above structures could be the meaning intended in the sentence, so the Collapse algorithm must be able to consider each possible collapse and return the one(s) with the best measure.

The following sentence illustrates the fact that any one of the five structures could be the preferred interpretation of a four structure stack. "He puts the block in the box in the carton on the table." would be divided into

D. on the table
C. in the carton
B. in the box
A. He puts the block

Depending on the circumstances the sentence occured in, it could mean either:

(A (B (C D))) -- The box is in the carton, the carton is on the table, and the block is put in the box. [When B modifies A, it can modify either the location of the block, or where the block was put. If only B modifies A directly, then it must specify where the block was put. If there is another modifier that could specify where the block was put, then B specifies the location of the block.]

(A (B C) D) -- The block is in the box, the box in the carton, and the block is put on the table.

(A B (C D)) -- The block is in the box, the carton is on the table, and the block is put in the carton.

Changing D to "on Thursday" yields

(A B C D) -- The block is in the box. It is put in the carton. The action is done on Thursday.

Changing C to "with the cover" yields

(A (B C D)) -- The box has a cover. The box is on the table. The block is put in the box.

The simplest algorithm for collapsing the stack would be to generate all legal collapses and then choose one with the best measure. This method is not used because the number of structures a stack can be collapsed to grows exponentially with the length of the stack. In fact, the sequence followed is the Catalan[2] sequence, which is (1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796...). The closed form for the Nth term of the sequence is

$$\frac{(2(N-1))!}{(N-1)!N!}$$ - The number of ways a stack of length N can be collapsed.

So it is obvious that we will want to use a more intelligent method for collapsing.

The set of structures a stack S may be reduced to is called the *collapse set*. We wish to generate the members of the *collapse set* in an order that gives us the best chance of finding the preferred structure in the set before generating the entire set.

In English usage, sentence constituents have a tendency to modify the constituents that are closest to them in the sentence. In a stack, this translates as "a stack

[2] Which, among other things, counts the number of ways a convex polygon of N sides can be triangulated [Gardner 76].

structure is most likely to modify the one directly beneath it in the stack." Our heuristic is to generate the members of *collapse set* that have the "closest modifications" first[3], and stop as soon as we generate a structure with perfect measure.

We define a metric to measure how well a member of the *collapse set* fits the "close modification" criteria. The metric counts the number of structures in the stack that modify structures *n* structures beneath them. S(N1,N2,...,NA) is the subset of *collapse set* whose members contains N1 structures that jump over one structure to find the structure they modify, N2 structures that jump over 2 structures to find the structure they modify, etc. The members of S(N1,N2,...NA) are more closely modified than the members of S(M1,M2...,MA) if and only if the sum of the N*i* (*i*=1,A) is less then the sum of the M*i* (*i*=1,A), or the sums are equal and there exists *j* ($1 \le j \le A$) such that N*j* > M*j* and N*i* = M*i* for all *i* less than A. eg., For a stack of five structures, the structure with the closest modifications is S(0,0,0) the structures that are in S(1,0,0) are the next most likely interpretation of the stack, and the structures in S(2,0,0,) are preferred over those in S(1,1,0). The Collapse routine generates the structures with the closest modifications first, with one important exception. Suppose the modification of structure N by structure M leads to a bad measure. Then every final structure in which M modifies some other structure with a better measure than it does N is generated before those containing N modified by M, even though the latter may be more closely modified.

Here is how this works on the sentence,

---

[3] There are certain exceptions: for example, if a verb structure in the stack has a passive verb group, and there is a preposition structure whose preposition is "by" above it, then the collapse routine tries to attach the "by" preposition structure to the verb structure first.

"Write me a program called intersection which prints a set of lists of numbers and outputs the numbers which are in all of them."

The stack to be collapsed is,

```
        PREP: OF
    9.  NOUN: THEM
        - - -
        PREP: IN
    8.  NOUN: ALL
        - - -
        VERB: ((BE))
        NOUN1: (WHICH !PL)
    7.  FUNCTION: WHICH
        - - -
        VERB: ((PRINT S))
        NOUN1: !match_to_conjunct_sub
        NOUN2: (NUMBER !PL THE)
    6.  FUNCTION: AND
        - - -
        PREP: OF
    5.  NOUN: (NUMBER !PL)
        - - -
        PREP: OF
    4.  NOUN: (LIST !PL)
        - - -
        VERB: ((READ   S))
        NOUN2: (SET A)
        NOUN1: (WHICH !SING)
    3.  FUNCTION: WHICH
        - - -
        VERB: ((CALL . ED))
        NOUN2: !#INTERSECTION
        NOUN1: !match_to_head_noun
    2.  FUNCTION: PASS
        - - -
        VERB: ((WRITE))
        NOUN3: (PROGRAM A)
        NOUN2: ME
        NOUN1: YOU*
        MSG: (IMP)
    1.  FUNCTION: MAIN
        - - -
```

or more simply,

9. of them
8. in all
7. which are
6. and prints the numbers
5. of numbers
4. of lists
3. which reads a set
2 called Intersection
1. write me a program

Collapse begins by trying to generate (1 (2 (3 (4 (5 (6 (7 (8 9)))))))), the only member of S(0,0,0,0,0,0,0,0). It successfully forms (6 (7 (8 9))) and tries to attach it to 5. It cannot since 6. must be attached to verb structure. An illegal attachment and an attachment with bad measure are handled similarly[4]. Collapse now looks down the stack for the closest structure which will accept 6. with a perfect measure. It finds 3. which means it now has to collapse the stack segment from 5. to 3. It calls itself recursively on the stack consisting of 5.,4. and 3. which results in the structure (3 (4 5)). The structure (6 (7 (8 9))) is attached to it, and Collapse goes back to work on the stack consisting of 1., 2. and (3 (4 5) (6 (7 (8 9)))). The result is,

[4] eg., if the attachment were legal but had a bad measure, Collapse would immediately start looking for a better place to put it. If none were found, it would settle for the bad measure.

```
(IMP (2#WRITE NN
       [ARG1 YOU*]
       [ARG3 ME]
       [ARG2 (PROGRAM A (1#CALL PN
                             [ARG1 !match_to_head_noun]
                             [ARG2 1#INTERSECTION]
                        )
                        [CONJ AND

                           (1#READ NN
                                [STEPOF !match_to_head_noun]
                                [ARGS (SET A (OF (LIST !PL (OF (NUMBER !PL)))))]
                           )

                           (1#OUTPUT NN
                              [STEPOF !match_to_conjunct_sub]
                              [ARGS (NUMBER THE !PL (2#BE NN
                                                      [ARG1 (!match_to_head_noun)]
                                                      [ARG2 (ALL (OF THEM))]
                                                    ))]
                           )
                        ])]
     )
)
```

4.3 Formatting

Format is the algorithm which prepares a structure for output. It is responsible for calling the deductive system to measure the structure.

4.3.1 Noun groups

The noun group of an unformatted structure is a list of the head noun and its modifiers. This list is handed to the deductive system which structures it and returns a measure of the appropriateness of the noun group. The representation used for the noun group's structure is dependent on the needs of the deductive system. Suppose Format were given a structure containing the noun group,

NOUN: (PROGRAM THEORY FORMATION THE)

The deductive system would be asked to structure it. The structure returned by the interpreter (chapter 5) would be:

```
NOUN: PROGRAM
        program-type:  THEORY-FORMATION
        definite: T
MEASURE: PERFECT
```

where "THEORY-FORMATION" is an atom denoting a certain kind of program.

The noun group representation used by the the deductive system does not matter to Reader, since once a structure is formatted, the parser no longer accesses it. The important piece of information, as far as Reader is concerned, is the measure of the noun group. It is not unreasonable to expect the deductive system to be capable of supplying such a measure. A system's ability to represent a noun group in a useful fashion implies that it has a measure on how well the noun group fits the representation.

The structured noun group is returned in the proper slot of Format's output. The measure of the noun group is added into the structure's **measure**, which will be returned along with the formatted structure.

4.3.2 Conjunctions

Format is responsible for bringing conjunctions up to their proper level in the sentence. "He reads books and writes poetry and music" would be parsed into

```
      NOUN: MUSIC
   3. FUNCTION: AND
      - - -
      VERB: ((WRITE S))
      NOUN1: !match_to_conjunct_SUB
      NOUN2: POETRY
   2. FUNCTION AND
      - - -
      VERB: ((READ S))
      NOUN1: HE
      NOUN2: (BOOK !PL)
   1. FUNCTION: MAIN
      - - -
```

When the stack is collapsed, 3. would be attached to 2., yielding

```
      VERB: ((WRITE S))
      NOUN1: !match_to_conjunct_SUB
      NOUN2: (POETRY (AND MUSIC))
   2. FUNCTION AND
```

When 2. is formatted, the conjunction (which until now has been treated just like a preposition) in NOUN2 is brought up to toplevel, producing (AND POETRY MUSIC). When the format of 2. is attached to 1., it is placed in the cases slot:

```
   - - -
   VERB:   ((READ S))
   NOUN1:  HE
   NOUN2: (BOOK PL!)
   CASES: ((AND (WRITE NN ((SUB !match_to_conjunct_SUB)
                           (OBJ (AND POETRY MUSIC))))))
1. FUNCTION: MAIN
   - - -
```

Format brings it up to top level so that the result of the parse is easily seen to be a conjunction:

```
[CONJ AND

  (READ NN
       [SUB HE]
       [OBJ (BOOK !PL)]
  )
  (WRITE NN
        [SUB !match_to_conjunct_SUB]
        [OBJ (AND POETRY
                  MUSIC)]
  )
]
```

The symbol "!match_to_conjunct_SUB" (section 2.4.6) refers to the SUB of the first conjunct ("he").

4.3.3 Filling in extra cases

Format provides a channel for the deductive system to determine if there are any missing cases in the verb that can be filled in from the rest of the sentence. Consider the sentence "John drove through and destroyed the plate glass window.", taken from [Woods 73]. Syntactically, it is possible for the object of the preposition "through" to be "the plate glass window." Reader asks the deductive system if this would make sense. If the answer is affirmative, Format would return

```
[CONJ AND
  (DRIVE PN
        [SUB JOHN]
        [WHERE (THROUGH (WINDOW THE PLATE GLASS))]
  )

  (DESTROY PN
         [SUB !match_to_conjunct_SUB]
         [OBJ !match_to_conjunct_PREP]
  )
]
```

where "match_to_conjunct_PREP" is to be matched to "the plate glass window". Notice that Reader cannot add cases to a verb without consulting the deductive system. In the sentence "John drove through and destroyed her confidence in him.", the object of "through" is not "her confidence in him".

4.3.4 Choices

Any choices in the parse structure (section 2.3.4) are generated in Format. Consider the choice offered for the SUB of "be" in

```
(KNOW PN                "I know that ice is slippery."
      (SUB ())
      (WHAT (BE PN
                (SUB (*CHOICE ICE
                              (ICE THAT)
                     ))
                (DES SLIPPERY)
            ))
)
```

Just before Format asks the deductive system to structure a noun it examines it to see if a choice can be made from it. In this case, the test that succeeds is that the noun is modified by "that" and is the SUB of a verb which belongs to a structure whose function is WHAT. The consequence of the test is that a choice of noun groups should be offered, one with "that" as a modifier, and one without "that". If the original sentence had been "I know that that ice is slippery", the second "that" would not have been added to the Modifier List. Instead, a message would have been left in the message slot of the verb structure which would have signalled Format not to test for this particular choice being present.

4.4 Parallel processing

Reader is designed to follow partial parses in parallel. If this were implemented straightforwardly, it would lead to an unfortunate amount of duplicated effort. Consider the parsing of the sentence "He had another look at the man in the trench coat who had been following him for the last hour." When "at" is read there are two partial parses:

```
1. msg = NOUN, ml = NIL
    - - -
   VERB: ((HAVE ED))
   NOUN1: HE
   NOUN2: (LOOK ANOTHER)
   FUNCTION: MAIN
    - - -
```

```
2. msg = VERB, ml = NIL
    - - -
   VERB: ((LOOK))
   NOUN1: ANOTHER
   FUNCTION: WHAT
    - - -
   VERB: ((HAVE ED))
   NOUN1: HE
   FUNCTION: MAIN
    - - -
```

If reader used simple parallel processing, "at" would be added to both partial parses, producing

```
1. msg = PREP, ml = NIL
    - - -
   PREP: AT
    - - -
   VERB: ((HAVE ED))
   NOUN1: HE
   NOUN2: (LOOK ANOTHER)
   FUNCTION: MAIN
    - - -
```

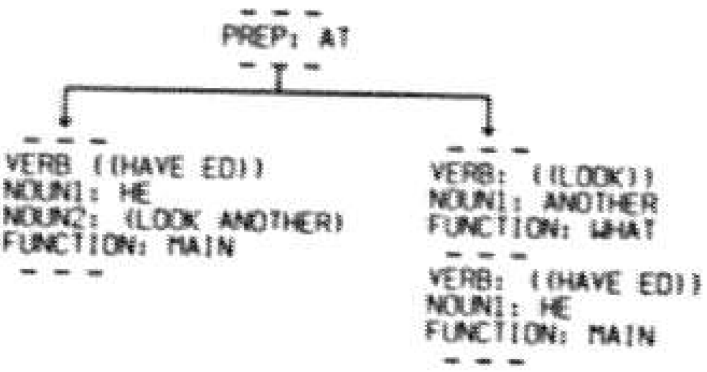
```
2. msg = PREP, ml = NIL
    - - -
   PREP: AT
    - - -
   VERB: ((LOOK))
   NOUN1: ANOTHER
   FUNCTION: WHAT
    - - -
   VERB: ((HAVE ED))
   NOUN1: HE
   FUNCTION: MAIN
    - - -
```

At this point, both stacks have the same top structure. The rest of the sentence, consisting of the noun group "the man in the trench coat who had been following him for the last hour" is going to be parsed twice, once for each partial parse. The different partial parses arose because words were applied to a single partial parse in different ways. This necessitated two different parses, because each could accept words differently. Parse 2. was able to accept "look" as a verb and parse 1. was able to accept it as a noun. But now that the stacks of each partial parse have the same top structure, most words will be added to the stacks in the same fashion. We can take advantage of this fact to avoid parsing the object of "at" twice.

In general, whenever two (or more) partial parses have identical top structures,

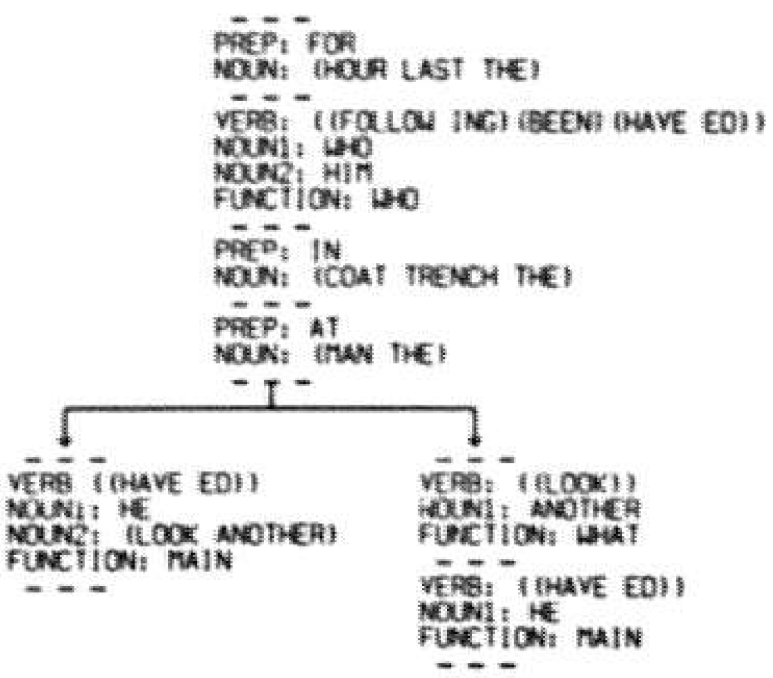
they are merged into one partial parse with a branching stack. The two partial parses above would be merged to:

msg = PREP, m1 = NIL



The stack branching is invisible to the grammar programs. When a SHIFT is called on a branched stack, it automatically follows down all the branches and separates the branched stack as required. In this case, the merge of the two partial parses cuts the parsing time for the rest of the sentence in half. The succeeding words in the sentence are applied to one partial parse, instead of two. Since none of the words in the remainder of the sentence are attached to structures below the current top of the stack, the two partial parses remain merged until the end of the sentence. After the last word in the sentence has been read, the stack looks like:

msg = NOUN, ml = NIL



Collapsing this stack produces two different parses:

```
(HAVE PN
     [SUB HE]
     [OBJ (LOOK ANOTHER (AT (MAN THE (IN (COAT THE TRENCH))
                                     (FOLLOW PPC
                                           [SUB !match_to_head_noun]
                                           [OBJ HIM]
                                           [FOR (FOR (HOUR THE LAST))]
                                     )))))]
)

and

(HAVE PN
     [SUB HE]
     [WHAT (LOOK NN
                [SUB ANOTHER]
                [AT (AT (MAN THE (IN (COAT THE TRENCH))
                                 (FOLLOW PPC
                                       [SUB !match_to_head_noun]
                                       [OBJ HIM]
                                       [FOR (FOR (HOUR THE LAST))]
                                 )))]
```

Merging partial parses is the other complication mentioned in the general control structure presented in section 2.2. Step 6 was "Reset *partial-parse-list* to a list of the partial parses formed in step 4." What actually occurs, is that Reader examines the list of partial parses formed in step 4, and modifies it by merging any partial parses whose stacks have the same top structure. *partial-parse-list* is then reset to the modified list.

The merging of partial parses is similar (in effect) to the use of a well-formed substring table (WFST) by parsers which use backup to achieve non-determinism rather than parallel processing. A well-formed substring table, [Kuno 63], is a collection of parsed sentence constituents. When a parser using a WFST backs up, it avoids reparsing sentences constituents by picking constituents it has already parsed out of the WFST. Similarly, in a parallel processing environment, the merging of partial parses avoids the reparsing of constituents by allowing each parsed constituent to be shared by every active partial parse which can use it.

4.5 Other parsers

A considerable amount of the work has been done in the field of natural language parsing. Much of this work has concentrated on syntax based parsers. These have evolved from simple systems implementing context free grammars, to rather complex systems motivated by transformational grammar considerations. Such parsers have grammars which consist of a context free grammar, along with a set of rules for modifying the parse tree built by the context free component. The parse tree may be modified while it is being constructed [Woods 73], or after it has been

completed [Sager 73]. This section examines the differences between some of these systems and Reader.

Reader's organization is similar to these systems in that we can view Format as the transformational component, and the grammar programs as the context free component. The differences in the systems lie primarily in the "context free" component. The first difference is that the grammar programs are more powerful than a context free grammar. Consider the sentence "Only one man was found who could speak English." In this sentence, "who could speak English" modifies "man". Reader parses the sentence by dividing it into a stack of two structures. When the stack is Collapsed, the top structure is attached to the bottom structure, which results in the proper modification. This modification cannot be expressed in a strictly context free grammar.

A more important difference lies in the way the "context free" component operates. The grammars for most syntax based parsers consist of a description of legal sentence structures. The grammar's application to a sentence results in a series of choices about which kind of constituent should be built at a particular point in the parse. Each system makes some effort to diminish the number of unsuccessful guesses. For example, Woods allows the grammar writer to "recommend" what guess to make at any point in the parse. Winograd's grammar[5] attempts to use the information gained from a failed guess at a decision point to allow it to choose intelligently from the remaining choices at the decision point.

---

[5] The grammar in Winograd's parser also consists of a set of programs. However, the programs deal solely with the construction of a parse tree, and are not oriented towards building structures that can represent more than one parse tree at a time.

Reader's grammar consists of a set of programs which determine the different ways a word may be added to a parse in a given configuration. The two methods are similar in that the guesses the older parsers make correspond to the guesses Reader must make in deciding which way to add a word to a partial parse. The difference in the methods is that Reader provides a framework (the stack) and a means (the grammar programs) for writing grammars that diminish the number of ways a word can be applied to a partial parse while still maintaining a substantial grammar. In most cases the grammar programs will apply a word class to a parse in only one way. However, a word which belongs to more than one word class will generally[6] be applied to a parse once for each word class it belongs to.

It can be argued that since all the more recent systems have the power of Turing machines, they can perform any algorithm, including those that Reader carries out. A simple answer to this is "Ah, but they don't". The reason they don't is that in many of the systems the "full power of a Turing machine" is used only to modify, as opposed to help build, the parse trees generated by the context free component. In other words, the Turing machine comes in *after* all the guessing has been done.

The methods used by Reader to avoid nondeterminism include a mechanism used in the ATN parser described in [Woods 1970]. Wood's parser is partially based on a finite state machine, and the method referred to involved the technique of making an arbitrary nondeterministic finite state machine deterministic by introducing several new states. Some of Reader's strategies can be viewed in this light, but most cannot, since they are involved with eliminating nondeterminism from situations which involve pushdown operations in the ATN formalism.

---

[6] Exceptions are single applications for words which are both conjunctions and prepositions, and words which are both nouns and modifiers.

Here is a concrete example. Section 3.2.2 explains how Reader parses simple relative clauses deterministically, using the example sentence "The city people hate is Tokyo". A nondeterministic ATN would begin parsing the sentence by attempting to find a noun phrase. It would have to guess whether to find "the city people" or "the city people hate". The guess consists of deciding when to "pop" up from the "push" of finding a noun phrase; exactly the kind of guess that a finite state machine transformation cannot help.

Another advantage listed for ATNs is the use of registers to make "...tentative decisions about the sentence structure and then change one's mind later in the sentence without backtracking." This is obviously a good feature for a parser to have, and seems equivalent to Reader's method of representing both sides of a decision while reserving the right to chose one or the other (without backtracking) later in the sentence. In Reader, this allows one to parse relative clauses and conjunctions deterministically, delay attaching various parse structures until more information is gathered about the reason for the attachment (thereby reducing the combinatorics of the attachment), combine different word class usages of a single word into one parse, etc. In contrast, [Woods 1970] contains two examples of the tentative decision method at work, which occur in the parsing of the sentence "John was believed to have been shot." The first decision is that *was* is the main verb of the sentence, which is later revised to *believe* is the main verb and *was* is an auxiliary verb. The second is the decision that *John* is the subject of *was*, revised later to *John* is the object of *believe*, and revised still later to *John* is the object of *shot*. In Reader's formalism, all these "decisions" are made and revised trivially. The final stack to collapse is:

```
      . . .
   VERB  ((SHOOT ED)(BEEN)(HAVE))
   NOUN1  !match_to_sub
2  FUNCTION  INF
      . . .
   VERB  ((BELIEVE ED)(BE BE3SP))
   NOUN1  JOHN
1  FUNCTION  MAIN
      . . .
```

The decision to make *was* a helping verb is accomplished by simply adding *believed* to structure 1. There is no need to assume what case *John* fills until the structure it is in is Formatted. Attaching an INF structure whose VERB is passive to a structure with a passive verb which accepts a clause entails removing the first noun in the latter structure, installing it as the first noun of the INF structure, and then attaching the INF structure as the clause case. When the INF structure is Formatted, "John" is made the object of "shot". The parse is,

```
(BELIEVE PN
        [WHAT (SHOOT NP
                     [OBJ JOHN]
              )]
)
```

There is at least one other parser under development that also tries to avoid needless guessing. It is being written by Marcus [Marcus 75] and is based in the belief that "...the structure of natural language provides enough and the right information to determine exactly what to do next at each point of the parse." The claim is that the parser will be able to avoid guessing what to do at a decision point because there is really only one acceptable choice. The system is still being written, so it is too early to comment on it. However, it seems that this approach will encounter problems when working with a sufficiently large grammar and words that can assume more than one syntactic category.

Some more recent parsing systems have been developed which deemphasize the role that syntax plays in the parsing process. Naturally, such parsers do not produce a "classical" parse tree, but instead produce a structure which is said to represent the "meaning" of the sentence being parsed. Examples of this type of work may be found in [Riesbeck 74] and [Wilks 73]. As this work has come after the more syntax oriented parsers discussed above, we should explain why we have rejected this approach.

The main reason is our belief that most semantic processing will be more expensive than syntactic processing in a rich environment. Therefore, it is desirable to use syntax to minimize the number of semantic interactions that need be considered. This contrasts with (for example) Riesbeck's work, in which he says "the functions of the analyzer to be described here ask questions about the relationship of words and concepts." Here, the process has been reversed; semantics and deduction are used to determine which words interact, and syntax is used only later, if at all, to ensure that a proposed modification between words is permitted. If one limits oneself to simple sentences, the added expense of using semantics instead of syntax to decide whether two words interact will not be overwhelming, since the possible interactions in a simple sentence will be few in number. However, the number of possible interactions to be examined semantically grows exponentially with the complexity of the sentence, so it seems that these methods will not be practical in a rich environment (in which there are many possible relationships between almost all words and concepts) which has to deal with complicated sentences.

5. The Interpreter

A brief overview of the interpreter is given in sections 1.2.2 and 1.5.2. Essentially, it is a computer program which attempts to understand natural language. There are many other computer systems which would make the same claim. The points of interest in all programs of this type are:

1. The representation used for the information contained in the natural language. For the interpreter, this is the program specification.

2. The representation(s) used for the knowledge base needed to understand the natural language.

3. The methods used for activating parts of the knowledge base to bear on a particular task.

The first point is covered in Section 5.1. Examples of different types of program specification types are given, along with an example which illustrates how several components fit together to describe a computer program. The section also discusses the representation of user's replies which are not incorporated into the program specification.

Section 5.2 introduces "concepts" and "definitions", the two representation units in the interpreter's knowledge base. The simplest type of concepts are those which are abstractions of components in the specification. An example of such a concept is #ADD, which refers to the concept of adding up several numbers. Information included in the #ADD concept is,

#ADD can be instantiated as a step in the program specification.

#ADD takes two or more arguments.

The arguments should be numbers. But an exception occurs when there is one argument which is a set of numbers. In that case, the numbers in the set should be considered the arguments of the #ADD.

Definitions provide instructions for mapping English word strings into concepts. The definition of "sum" contains information which allows the interpreter to map "The program sums up the last three numbers." into an #ADD which is a step of "the program" and whose arguments are "the last three numbers".

The task of relating a phrase like "the last three numbers" to a specific component (or components) in the program specification is referred to as matching. Section 5.4 covers the matching process, explaining how the information contained in concepts and definitions is used during matching.

The primary goals of the processing performed by the interpreter are conceptually very simple, and sections 5.2, 5.4 and 5.3 (which explains the interpreter's processing cycle to provide background for section 5.4) should be read with them in mind. The goals, upon receiving a parse structure, are:

1. Determine which definitions can be applied to the parse structure, and therefore which concepts the parse structure is invoking.

2. Find or create referents in the program specification for the descriptor slots of the concepts the parse has been reduced to.

3. Incorporate the appropriate concepts into the program specification.

Section 5.5 explains how definitions and concepts are used to provide the measure information necessary for the interface between Reader and the Interpreter. The final section mentions some work remaining to be done.

5.1 The results of interpretation

5.1.1 The program specification

The program specification contains a record of everything the user has said (and the interpreter has inferred) which is relevant to the description of the program being written. The parser/interpreter uses it as a data base for matching, the parser/interpreter interface etc. This section describes the format of the specification. Later sections will show how it is utilized by the parser/interpreter.

The principal result of the interpreter is the program specification. The program specification[1] represents a computer program, and can be viewed as a high level programming program language. It consists of a connected set of components. Such a data structure has been labeled a "entity-attribute-value data structure" in [Heidorn 74], and a "set of conceptual entities with associated descriptions" in [Bobrow 76].

The description of a component is a collection of descriptor/value pairs which specify the actions and structure of the component. For example, a component may have as its description,

```
A0358
  class: ALG
  type: OUTPUT
  args: "Ready"
  step-of: A0367
```

which means that it is an Algorithm component that should be mapped into an "output" operation in the target language (eg., WRITE in Fortran, PRINT in Lisp, etc.).

---

[1] The program specification semantics were developed with Jorge Phillips.

The argument of the output is the string "Ready". The **step-of** descriptor indicates the position of the component in the specification; it is one of the steps of an ALGorithm component denoted by A0367.

Each descriptor has an inverse associated with it. For example, if a component X is in the **steps** descriptor of a component Y, this fact can be derived by examining either X or Y.

A component belongs to one of two classes: ALGorithm or DATA. Each class is subdivided into several types. Figure 5.1 shows some control structure ALGORITHM types.

PROCEDURE
ARGS: a list of DATA components whose type is BOUND.
DEFINITION: An ALG component.

SEQ
STEPS: a list of ALGs to be executed in sequential order.

CASE
CONDITION:
an ALG with a RESULT slot, or a DATA which is the RESULT of an ALG.
STEPS: a list of ALGS to be executed if the CONDITION is TRUE.

COND
CASES:
a list of ALGS whose type is CASE. The first CASE whose condition is TRUE is executed. The rest are ignored.

ENUMERATE
ON: a DATA whose type is SET.
STEPS:
a list of ALGS to be executed sequentially for each element in the ON set. The iteration element is represented by the generic element of the ON set.

LOOP
EXITS: a list of ALGS whose type is CASE.
COUNTER:
a DATA of type INTEGER whose value is the number of times the LOOP has been executed.
STEPS:
a list of ALGs which includes every CASE in EXITS. The ALGs in STEPS are repeatedly executed until the condition of a CASE in EXITS is satisfied.

CALL
PROCEDURE: an ALG of type PROCEDURE.
ARGS: a list of DATAs which are bound to the args of PROCEDURE.

Figure 5.1

Control structure ALGorithm types

The remaining ALGorithm types can be divided into predicates and primitive

operations. The number of these is essentially unlimited, since anything the PSI coding module can code can without instructions from the user can be considered primitive. Figure 5.2 provides some samples of the primitive operations and predicates used by the current system.

MAP
  ARG1: a DATA component whose type is MAPPING.
  ARG2: a DATA component
  ARG2: a DATA component
MAP is the program specification primitive for associating one DATA component (ARG2) with another (ARG3) via the mapping ARG1. It is a generalization of the Lisp PUTPROP command. IMAP corresponds to GETPROP.

IMAP
  ARG1: a DATA component whose type is MAPPING.
  ARG2: a DATA component
  RESULT: the DATA component that ARG1 maps ARG2 to.

COMPUTE
  ON: a DATA component which is a set.
  RESULT: a DATA component which is a set.
  QUANTIFY: either ALL, SOME or DATA component which is an integer.
  ASSERTIONS:
        a list of ALGs which are assertions involving the generic
        element of the RESULT set.
The RESULT set is a subset of the ON set which consists of all, some or any n (depending on the value of QUANTIFY: ALL, SOME or a number n) of the of the elements in the ON set which satisfy the ASSERTION list.

INPUT
  ARGS: a list of the DATAs being read in.
  PROMPT: a DATA of type STRING which is output to herald the INPUT.

MEMBER
  ARG1: a DATA component.
  ARG2: a DATA component which is a SET.
  RESULT: a DATA of type BOOLEAN which reflects whether ARG1 is in ARG2.

FORALL
  BINDINGS: a list of DATAs whose type is BOUND.
  PREDICATE: an ALG with a RESULT.
  RESULT: a BOOLEAN which is the truth value of universal quantification.

Figure 5.2

Primitive operations and predicates

Data structures, like primitive operations, come in any form that the coder is able to handle. Figure 5.3 shows some DATA types and example DATAs.

SET
ELEMENT: a DATA which is the generic element of the set.

RECORD
FIELDS: a list of DATA components whose type is FIELD.

FIELD
DATA: a DATA component which the contents of NAME filed of a RECORD.
NAME: the name of the FIELD.
QUANTIFY: either ALL, SOME or DATA component which is an integer.

```
class  DATA
type   SET                  [the empty set]
value  PHI

class DATA
type  BOOLEAN               [the boolean value TRUE]
value TRUE

class DATA
type  RECORD
rep  GRAPH
instanceof A0001
assertions (A0002 A0003)
```

The DATA above illustrates the three descriptors any DATA may have. The REP descriptor indicates that the program designer is referring to this component by the word "graph". ALG components may also have REP descriptors. The INSTANCEOF descriptor indicates that the structure of this component is the same as the structure of the component which A0001 points to. The ASSERTIONS descriptor contains a list of assertions about the component.

Figure 5.3

Data structure types and examples

5.1.2 An example and comparison

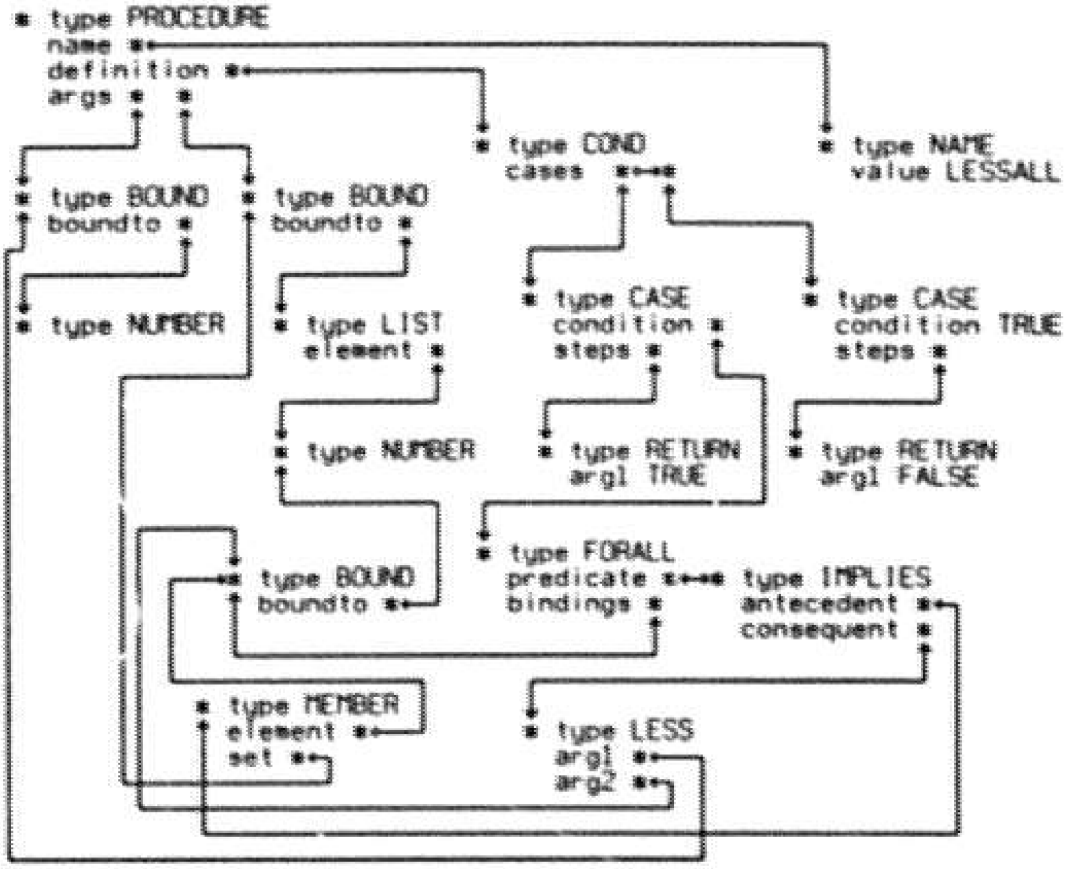
This section illustrates how these pieces are combined in a program description. Figure 5.4 contains a short dialogue, the program specification the interpreter has built from it, and the pretty printed version of the specification.

WHAT IS THE NAME OF THE PROGRAM YOU WISH TO WRITE?

Lessall.

DESCRIBE LESSALL.

Lessall takes a number and a list of numbers as arguments. It returns True if the number is less than every number in the list. Otherwise it returns False.




```
LESSALL (B1 B2)
  If FORALL(B3) IMPLIES(MEMBER(B3 B2)
                        LESS(B1 B3))
    Then RETURN(TRUE)
    else RETURN(FALSE)
```

B3 is a variable bound to A1. B2 is a variable bound to A2.

B1 is a variable bound to A3. A3 is a number. A1 is the generic element of A2.

A2 is a list whose generic element is a number.

Figure 5.4

Lessall and its program specification

The top node in the program specification is always a PROCEDURE component. In this case, it has two arguments, which are bound to a number and list of numbers respectively. This structure information is required by the coder, as it enables it choose its algorithm based on the data structures the algorithm is meant to manipulate. The body of the procedure is a COND with two cases. If the condition ("∀X X ∈ B2 ↔ B1 < X", where B1 is the number and B2 the number list) is True, then True[2] is returned. If not, the STEPS slot is ignored and the second CASE is tried. The condition of the second case is True, so anytime the first condition does not obtain, False will be returned. The control structure and data descriptions beneath the specification diagram are the distillation (as obtained by the specification pretty printer) of the program description information contained in the diagram.

The LESSALL program was taken from a paper on the Dedalus system, [MANNA 77]. Dedalus is an automatic program synthesis which uses a formal specification language as its input, rather than English. Since the interpreter's output corresponds to the input of such a system, a comparison between the two is a useful measure of the effectiveness of the interpreter. In this case, the two are virtually identical: the Dedalus input for LESSALL is

LESSALL(X L) <== **compute** X < *all*(L)
where X is a number and L is a list of numbers.

The expression X < *all*(L) means that "...X is less than every member of the list L."

[2] To save space, TRUE and FALSE have been used to represent the BOOLEAN components whose values are TRUE and FALSE.

5.1.3 Mets-comments

Some of the program designer's instructions to the system do not describe the program, but instead are intended towards directing the course of the dialogue. Comments like,

> I don't understand.
> What we were talking about?
> What did you mean by "the predicate fits"?
> Forget about prompts.

do not fit into the program specification, but are meaningful nonetheless. Such statements are sent to the dialogue expert as a filled in case frame. The case frame is actually a concept (next section) and it is filled in in exactly the same way that concepts are instantiated. The only difference is that instead of being added to the program specification, the instantiated concept is sent to the PSI dialogue module for processing.

As an example, we will examine the concept of #USER-QUESTION-REQUEST. Statements like,

> Ask about the scene before the concept.
> Let's talk about the scene.
> Ask me about prompts before asking me about the scene.
> Ask me about the structure of the scene first.,

which are addressed to wh. and which questions should be asked are mapped to #USER-QUESTION-REQUESTs. A #USER-QUESTION-REQUEST is specified by three descriptors:

QUESTION: either a question type (eg., STRUCTURE), a question (eg., (STRUCTURE A0012)) or a component (eg., A0012).

TIME: either BEFORE, AFTER (in which case REFERENT must be present) or LATER or NOW.

REFERENT: takes the same values as question.

The interpretation of a #USER-QUESTION-REQUEST is ask
  (one of)
    all questions of type QUESTION
    this particular QUESTION
    any questions about the component which is QUESTION

either NOW or LATER, or
BEFORE or AFTER asking

  (one of)
    all questions of type REFERENT
    this particular question which is REFERENT
    any questions about the component which is REFERENT

Then if A0001 points to the scene, and A0002 to the concept, we have,

Ask about the scene before the concept.
  [#USER-QUESTION-REQUEST Question: A0001 Time: BEFORE Referent: A0002]

Let's talk about the scene.
  [#USER-QUESTION-REQUEST Question: A0001 Time: NOW]

Ask me about prompts before asking me about the scene.
  [#USER-QUESTION-REQUEST Question: PROMPT Time: BEFORE Referent: A0001]

Ask me about the structure of the scene first.
  [#USER-QUESTION-REQUEST Question: (STRUCTURE A0001) Time: NOW]

5.2 The knowledge base

The knowledge base used by the interpreter consists of two declarative blocks of knowledge, and a set of programs which make use of the information in them. The programs are used to construct the specification, using the descriptions contained in Concepts and Definitions, the two declarative blocks. There is no formal definition of what constitutes a concept; a concept is anything which the interpreter can reason about. Hence there is a concept behind every ALGorithm and DATA type in the specification, as well as several higher order concepts. A definition is a means of mapping a sequence of English words into a concept.

5.2.1 Concepts

Concepts express many things, but are oriented towards supplying the information needed to instantiate and reason about components. Instantiation refers to the process of creating a component and filling in its descriptors with other components in the specification, so that it too becomes part of the specification.

The information contained in a concept is

> Descriptors. What descriptors the concept can take, the type checking constraints the descriptors must obey, questions to ask if the concept is presented without a necessary descriptor, and default descriptor values.
>
> Postconditions: what is true after the concept has been executed
>
> Side effects: what changes to make to the program specification when the concept has been recognized

For an example, consider the concept #MAP. #MAP represents the primitive operation in the specification which allows the user to associate one data with another. Figure 5.5 contains the #MAP concept.

```
#MAP

  DESCRIPTORS:
     STEPOF
          CHECK1:     ISA #ALG
          QUESTIONS:  Where does the #MAP belong?
     ARG1
          CHECK1:     ISA #MAPPING
     ARG2
          CHECK1:     ISA #DATA
          CHECK2:     #MAP-CHECK2(ARG1 ARG2 ARG3)
          QUESTIONS:  What is being #MAPped?
     ARG3
          CHECK1:     ISA #DATA
          CHECK2:     #MAP-CHECK2(ARG1 ARG2 ARG3)
          QUESTIONS:  What is ARG2 being #MAPped to?

  POST-CONDITIONS:  (#EQUAL (#MAP ARG1 ARG2) ARG3)

  SIDE-EFFECTS:     MAPPING-UPDATE(ARG1 ARG2 ARG3)
```

Figure 5.5

The #MAP concept

Figure 5.5 shows that a #MAP is specified by four descriptors. Each descriptor has information associated with it which assists the interpreter in filling in the descriptor slot. For instance, ARG2 must be a DATA component (#DATA refers to the concept of a DATA component). The second check provides a more contextual type checking which is used during matching and the parser/interpreter interface. Since the check is more complicated than a simple type check (eg., ISA #DATA), a program (MAP-CHECK2) is called which returns True or False, depending on whether ARG1 is a MAPPING which maps components of type ARG2 into ARG3. If a #MAP is to be instantiated and ARG2 is not present, then the question "What is the second argument of the map?", represented by (ARG2 A0001) where A0001 points to the

instantiated MAP, is asked. SIDE-EFFECTS consists of things which should be done whenever a component is instantiated. In the case of #MAP, SIDE-EFFECTS consists of a program (MAPPING-UPDATE) which updates the range and domain of ARG1 if necessary. The POST-CONDITIONS are what is true after the concept has been executed. Section 5.3 on matching explains how the POST-CONDITIONS and CHECKS are used.

Figure 5.6 shows the interpreter's concept of #DATA and #SET.

```
#DATA
  DESCRIPTORS:
     INITIAL-VALUE
          CHECK1:     ISA #DATA
          QUESTIONS:  What is the initial value of the DATA?
     VALUE
          CHECK1:     ISA #DATA
  PREPOSITIONS:
     WITH
          CHECK1:     ISA #DATA
          MEANING:    (#ASSOCIATE data object)
     IN
          CHECK1:     ISA #SET
          MEANING:    (#MEMBER data object)
```

```
#SET
  DESCRIPTORS:
     ELEMENT
          CHECK1:     ISA #DATA
          DEFAULT:    instantiation of a DATA whose REP is ELEMENT.
     SIZE
          CHECK1:     ISA #INTEGER
  CLASSIFIERS:  ELEMENT
  PREPOSITIONS:
     OF
          CHECK1:     GENERIC-ELEMENT()
          MEANING:    ELEMENT
```

Figure 5.6

The #DATA and #SET concepts

The concepts in Figure 5.6 both have information about prepositional modifiers. Such information is usually associated with individual word definitions, but when the

modification is standard for the concept, regardless of how it is expressed in English, the information is tied to the concept itself. The "in" modification for #DATA means that every time a word which maps to a #DATA is modified by a prepositional phrase whose preposition is "in" and whose object is a #SET, the meaning of the modification is that the component the word matches to is a member (represented by the interpreter concept #MEMBER) of the component the preposition object matches to. The "of" modification for #SET is slightly different in that the meaning of the modification is a descriptor of #SET rather than a concept. This means that the object of the preposition fills that slot in the #SET description. The check for "of" is a program which makes sure that the preposition object is a plural noun which is a #DATA.

The CLASSIFIERS slot is similar to PREPOSITIONS in that it appears in definitons, rather than concepts, except in cases in which the meaning of the classifier is the same for all nouns mapping to the concept. For #SET, the CLASSIFIERS slot says that if a noun modifies a noun mapping to set, and the noun satisfies the checks for ELEMENT, then it fills the ELEMENT descriptor of the #SET. eg., in "the integer list", "integer" is a classifier of "list" which maps to #SET. Since "integer" is a #DATA, it is assumed to be the generic element of the list.

To avoid needless duplication of information, the concepts are arranged in a refinement tree in which every concept shares all the information associated with its parent in the tree. #SET is a refinement of #DATA. Thus when checking #SET for information, all the information connected to #DATA applies. eg. If A0424 has just been instantiated as a set, the question "What is the initial value of A04247" will be pending. Of course, if the system can answer the question (perhaps A0424 is the argument of an INPUT), it will never be asked of the program designer.

Concepts are also used to capture regularities in language. English provides many different ways to express the same thought. For example, X is a function of Y can be stated as,

X depends on Y.
X is calculated from Y.
X is determined from Y.
X is calculated on the basis of Y.
X can be found from Y.
X is based on Y.
X is obtained from Y.
X is related to Y.
X is found by examining Y.

As an aid in writing definitions, it is useful to have all these phrases map into a single manipulable entity, namely the concept of #CALCULATION. #CALCULATION has two descriptors, ARG1, which is a #DATA, and ARG2 which is #PREDICATE. Methods for using concepts like #CALCULATION are explained in the following section on definitions.

5.2.2 Definitions

Definitions are used to map from English words to concepts. At the same time, they provide the parser with measure information it needs.

The information contained in a definition is,

Concept: What concept the definition maps to.

Word: what word the definition is a definition of.

Case-Descriptor relationships: Which verb cases can be used to fill the descriptor slots of the concept. Which cases must be, or are preferred to be, present for the definition to succeed.

Prepositions: Which descriptors prepositions can fill.

Conjunctions: Which descriptors conjunctions can fill.

Defaults: Default values for some descriptor slots.

Clauses: Which descriptors can be filled by clauses not introduced by conjunctions.

Figure 6.7 contains an example.

```
1#MARK
  DEFINITION-OF:  MARK
  ISA:  #MAP
  CASES: (SUB STEPOF)(OBJ ARG2 Must)(IOB ARG3 Preferred)
  PREPOSITIONS:
    AS
      CHECK1:    ISA #DATA
      MEANING:  ARG3
  DEFAULTS:
    ARG1: GET-MAPPING(MARK)
```

```
1#COLLECTION
  DEFINITION-OF COLLECTION
  ISA #SET
```

Figure 6.7

A definitions of mark and collection

Suppose that the interpreter receives the sentence "Mark the scene 'necessary'". The parse is

```
(MARK NN
     (SUB YOU*)
     (OBJ (SCENE THE))
     (IOB "necessary")
)
```

The definition will successfully map the sentence into the concept if all the requirements for the concept descriptors are met. Following the CASEs slot, YOU* is matched to an ALGorithm component as the STEPOF descriptor, and "the scene" and "necessary" are matched to #DATAs as the ARG2 and ARG3 of the #MAP to be instantiated. The "Must" in the OBJ mapping indicates that the OBJ case must be

present for the definition to succeed. Similarly, the "Preferred" in the IOB case means that IOB case is strongly preferred to be present, but not necessary. This means that using the verb "mark", something can be marked without specifying what the marking is, but a marking cannot be specified without mentioning what is being marked. ARG1 of the #MAP comes from the default slot of the definition; the value of a program (GET-MAPPING) which finds the MAPPING component be used for "mark", or creates one if this is the first instance of "mark" in the program specification.

Nouns are defined similarly to verbs, with the exception that the case information is missing (it is usually replaced by classifier information). Figure 5.7 contains the interpreter's definition of "collection".

Figure 5.8 contains two definitons which utilize the #CALCULATION concept.

```
1#CLASSIFY
  DEFINITION-OF:  CLASSIFY
  ISA:  #CALL
  CASES:  (SUB STEPOF) (OBJ ARGS)
  CLAUSES:
     CHECK1:     ISA #CALCULATION
     MEANING:    PROCEDURE (extract ARG2)
  DEFAULTS:
     RESULT:  instantiation of a DATA whose REP is CLASSIFICATION.

1#BASE
  DEFINITION-OF:  BASE
  ISA:  #CALCULATION
  CASES:  (OBJ ARG1)
  PREPOSITIONS:
    ON
     CHECK1:     ISA #PREDICATE
     MEANING:    ARG2
```

Figure 5.8

Definitions for "classify" and "base"

Consider the processing of the sentence "It classifies the scene based on whether

it fits the concept." "based on whether it fits the concept" is mapped to a #CALCULATION whose ARG2 is the predicate "it fits the concept". It is also a clause which modifies "classify" (anticipating section 5.5 on the parser/interpreter interface, we note that the reason the parser knows "based" modifies "classify" rather than "scene" is precisely because one modification is meaningful (all the words -> definitions -> concepts maps succeed) and the other is not). According to the definition, a clause can modify "classify" if it is a #CALCULATION. If it is, the modification instructions are to fill the PROCEDURE slot of the "classify" #CALL with ARG2 of the #CALCULATION. This work is done during Formatting, so the parse for the sentence is,

```
(IMP  (CLASSIFY NN
              [STEPOF YOU*]
              [ARGS (SCENE THE)]
              [PROC (FIT NN
                         [ARGS IT]
                         [ARGS (CONCEPT THE)]
                    )]
      )
)
```

Had the sentence been,

> Classify the scene on the basis of whether it fits the concept.
> Classify the scene as a function of whether it fits the concept.
> Classify the scene depending on if it fits the concept.
> etc.

the result would have been the same.

Many times, unknown words are used to refer to undefined predicates or supparts of the program being described. Since it would be unreasonable to expect all words to be included in the system, and often, the definitons of such words are inferable from context, the interpreter uses a "template" definition to try to create a definition for any unknown words which are used in the dialogue.

Here is an example:

> The program reads a graph and a node. A graph is a set of pairs. Each pair consists of two nodes, which are primitive. The program prints a list of all the nodes which can be reached from the input node.

When the interpreter encounters the last sentence, it has no information about "reach" other than that it is a verb. Because it is being used as the main verb of a clause which modifies a noun, the interpreter assumes that it represents a predicate which the program designer has yet to define. The "template" predicate definition and its instantiation for "reach" is shown in Figure 5.9.

```
PREDICATE-TEMPLATE
  DEFINITION-OF:  ---
  ISA:  #PROCEDURE
  CASES: (SUB ARGS) (OBJ ARGS)
  PREPOSITIONS:
    match
      CHECK1:     ISA #DATA
      MEANING:    ARGS
```

```
1#REACH
  DEFINITION-OF:  REACH
  ISA:  #PROCEDURE
  CASES: (SUB ARGS) (OBJ ARGS)
  PREPOSITIONS:
    FROM
      CHECK1:     ISA #DATA
      MEANING:    ARGS
```

Figure 5.9

A template definition and its instantiation

The template definition maps to a #PROCEDURE. The "match" in its PREPOSITIONS slot matches to any preposition that the interpreter cannot attach to anything else. The resulting definition of "reach" asserts that "reach" is a PROCEDURE, and that the preposition "from" can be used to introduce one of its arguments.

5.2.3 Procedural embedding

Most of the interpreter's knowledge about programming is represented by procedures. This information is necessary in order to incorporate what the program designer has said in the program specification without asking questions which the designer would feel his statements have implicitly addressed. It is not intended to help the interpreter from a problem solving (eg., writing efficient algorithms from inefficient descriptions) standpoint. The information was modelled procedurally since this seemed to be provide the easiest way to encode and apply it. The disadvantages of the procedural approach (primarily opacity) do not apply, as the information encoded in the procedures is not needed elsewhere in the system.

The information is organized into several modules which are expert in building various constructions in the program specification. There are modules which build CONDs from a series of CASES, construct COMPUTEs, note scoping ambiguities, build quantified expressions from phrases like "all relations in the concept not in the scene...", etc. As an example, we will consider the EXIT-TEST module.

The EXIT-TEST module is responsible for setting up the exit conditions of loops. Its arguments are the loop and the phrase which indicates the exit condition. The method for building a loop from each of the phrases it knows about is simply programmed out. Here is an example.

Figure 5.10 contains a fragment of a program specification.

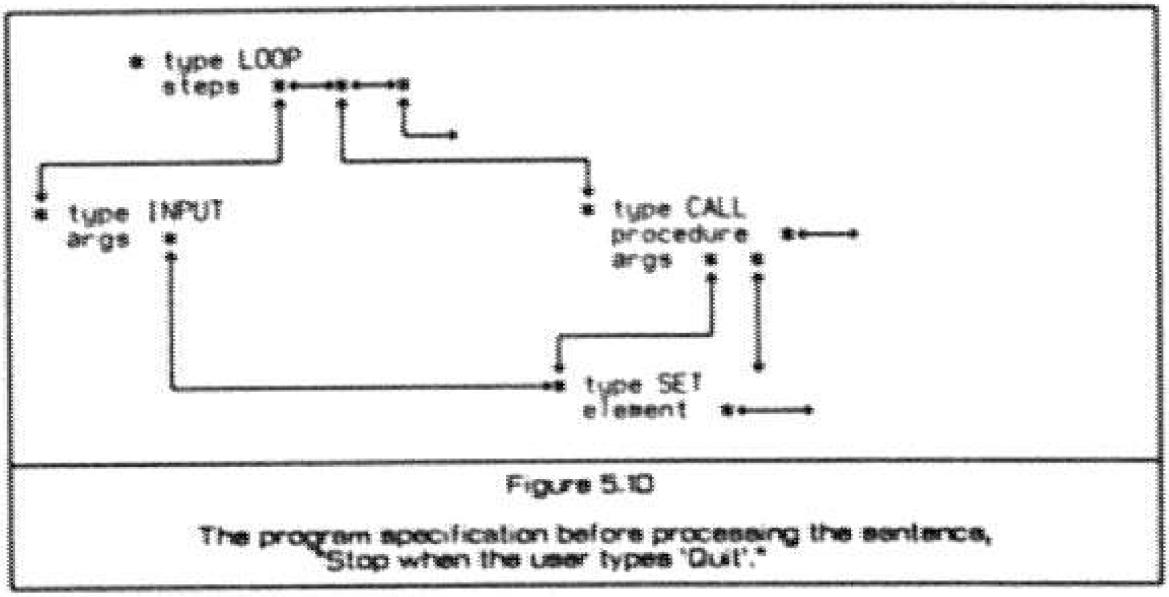


Figure 5.10

The program specification before processing the sentence, "Stop when the user types 'Quit'."

The interpreter is about to process the response the sentence "Stop when the user types 'Quit'." which was in response to the question "Should there be an exit test for the loop?"

EXIT-TEST receives [#INPUT (ARGS "Quit")] and the LOOP as its input. When the phrase is an #INPUT concept, EXIT-TEST finds an INPUT in the loop and places a test for the ARGS of the #INPUT concept after it. The result is shown in Figure 5.11.

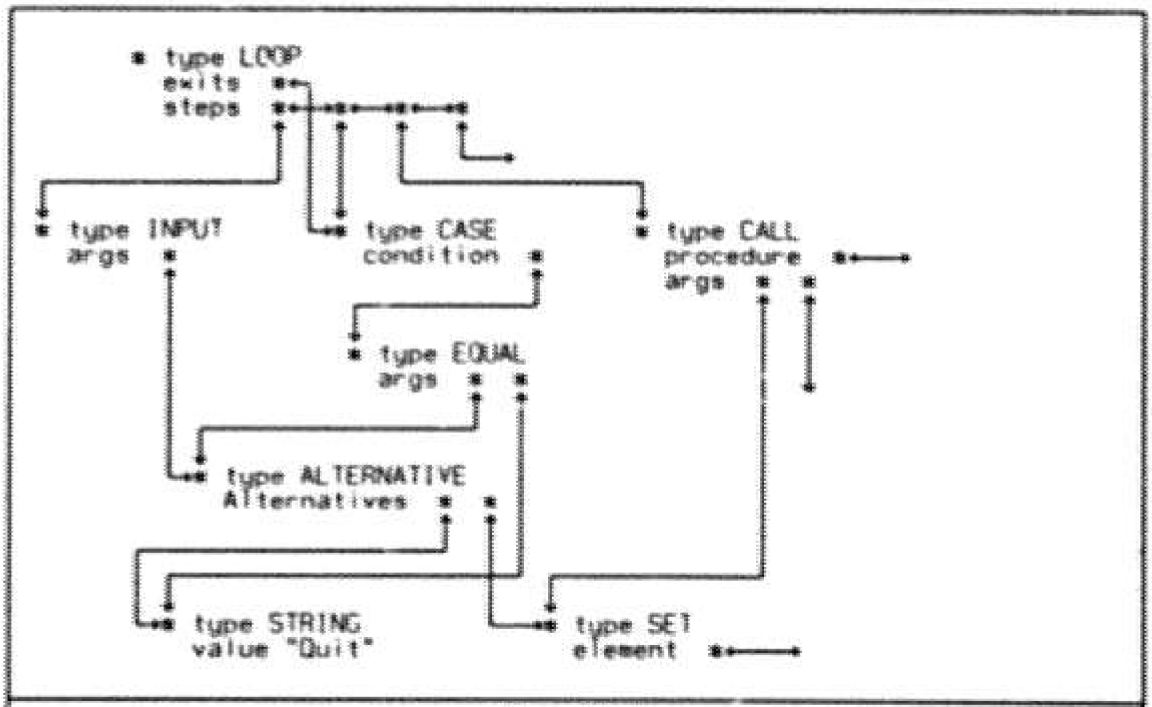


Figure 5.11

The program specification after processing the sentence, "Stop when the user types 'Quit'."

The exit test building program has added four new components: the CASE component which is the exit test, an EQUAL component which is the condition of the exit test, and a STRING and ALTERNATIVE component. The ALTERNATIVE component, which replaced the SET as the argument to the INPUT, reflects the fact the arguments to the INPUT may now be either the SET or a STRING whose value is "Quit". The ALTERNATIVE has been installed as one of the arguments of the exit test, while the SET remains as one of the arguments to the CALL following the test.

5.3 The processing cycle

The processing cycle refers to the sequence of actions taken by the interpreter during the processing of a user reply. The cycle begins with the receipt of a question and user reply from the PSI dialogue module. The reply may be a phrase or any number of sentences. The question typically consists of a descriptor slot and a component (the question object) which is missing information for the slot. (eg., (ARGS X) means "What are the arguments for X".

The first action taken by the interpreter is to update the *Focus* to the object of the question. Section 5.4 explains the use of the *Focus* and its companion, the *Data Focus*.

Then each sentence in the reply is parsed and the result is analyzed. The analysis consists of determining which concepts the sentence invokes, finding (or creating) components to fill in the descriptor slots of these concepts, and instantiating the concepts found into components in the program specification. Analysis has several side effects besides the building of the specification.

Throughout analysis, the *Focus* and *Data Focus* are constantly updated to reflect the components the program designer is talking about.

Another important side effect is the questions are posed by the instantiation of incomplete concepts. For instance, the reply,

> "It reads a scene, tests whether it fits the concept, verifies the result of this test with the user, and updates the concept. Then it repeats the process."

causes the questions,

> What is the structure of the scene?

What is the structure of the concept?
What is the initial value of the concept?
Describe verifying the test result.
Describe updating the concept.
Describe the test of whether the scene fits the concept?
What is the exit test of the loop?

to be placed on the question queue.

The instantiation of an incomplete concept may also lead to a job being put on the background job queue. The background job queue consists of questions which the interpreter cannot answer immediately, but expects to be able to answer after more information has come in. If the information never arrives, the interpreter assumes that the program designer was leaving the implementation to the PSI coding module. These questions are placed on the background job queue (rather than the question queue) queue to ensure that they will never be asked of the user. The background job queue is implemented as a list of procedures and their arguments, which are run at the end of every processing cycle. Those that succeed in answering their questions are removed from the cycle. An example of a background is the one associated with the #ASSOCIATE concept. #ASSOCIATE is used by the interpreter as an intermediate representation of the fact that two DATAs are somehow being associated. For instance, in

"Cookbook reads a recipe list, and then repeatedly reads a name
and prints the recipe with that name"

"with that name" maps into an #ASSOCIATE whose args are "the recipe" and "the name". At this point, there is no way to tell how the program designer expects "names" and "recipes" to be associated, so a background job is set up. A background job is used rather than a question since if an answer is never found,

the PSI coder will be able to choose an efficient implementation, and in fact, the user may be too unsophisticated to answer such a question. The background job remains active until the program designer says,

"A recipe has a name, an ingredient-list, and directions."

This defines "recipe" as record structure with three fields, one of which is a name. One of the situations the #ASSOCIATE background job knows how to resolve is the case where one of the associated DATAs is a field of the other. It changes THE #ASSOCIATE assertion from

[ASSOCIATE arg1: A1 arg2: A2]

to

[EQUAL args: ([FETCH arg1: A1 label: NAME] A2)]

where A1 and A2 point to the recipe and name, and FETCH is the interpreter primitive which gets the DATA of the label FIELD of its ARG1.

When each sentence in the program designer's reply has been analyzed, the background jobs are run and the question list is examined to see if any of the questions have been answered by subsequent analysis. The revised question list is sent to the PSI dialogue module, which selects a question, gets a reply from the program designer, and gives the question chosen and the designer's response to the interpreter to start another cycle.

5.4 Matching

This section is concerned with the identification of English noun phrases, which occurs during the filling in of a concept's descriptor slots, and consists of finding the component, or creating the component if none exists, which is the contents of the descriptor slot being filled, based on the English presentation of the component (eg., the noun phrase).

The system's handling of pronouns and nouns is virtually the same. The only difference lies in the possible match set. A pronoun may match any component in the specification which has been mentioned and meets the syntactic requirements (eg., plural, animate etc.) of the pronoun. A noun may match any component in the specification which has been referred to in the same (or a synonymous) way. The key to the matching process is the context supplied by the concept whose slot is being filled.

5.4.1 Nouns

The first time a noun is used, the system creates a component which is indexed under the noun's definition. Thus, "It reads in a scene." would cause the component:

```
A1
class DATA
rep 1#SCENE
```

to be created, where 1#SCENE is a definition the interpreter creates for "scene". 1#SCENE is assumed to be a #DATA so that it satisfies the type constraints of the ARGS of an #INPUT. Associated with 1#SCENE is the fact that A1 is an instantiation of "scene". The situation we have outlined leads to the simplest kind

of matching. If the user says, "Print the scene.", "the scene" is matched to A1 because the "the" implies that the referent should be found in the specification, A1 is the only instantiation of "scene" in the specification, and it satisfies the type constraints of the ARGS of #OUTPUT.

Now consider a slightly more complicated situation. Suppose we have scenes and concepts, each of which are sets of relations. Further, the relations in the concept are marked either "possible" or "necessary". Figure 5.12 shows how this would be represented in the program specification.

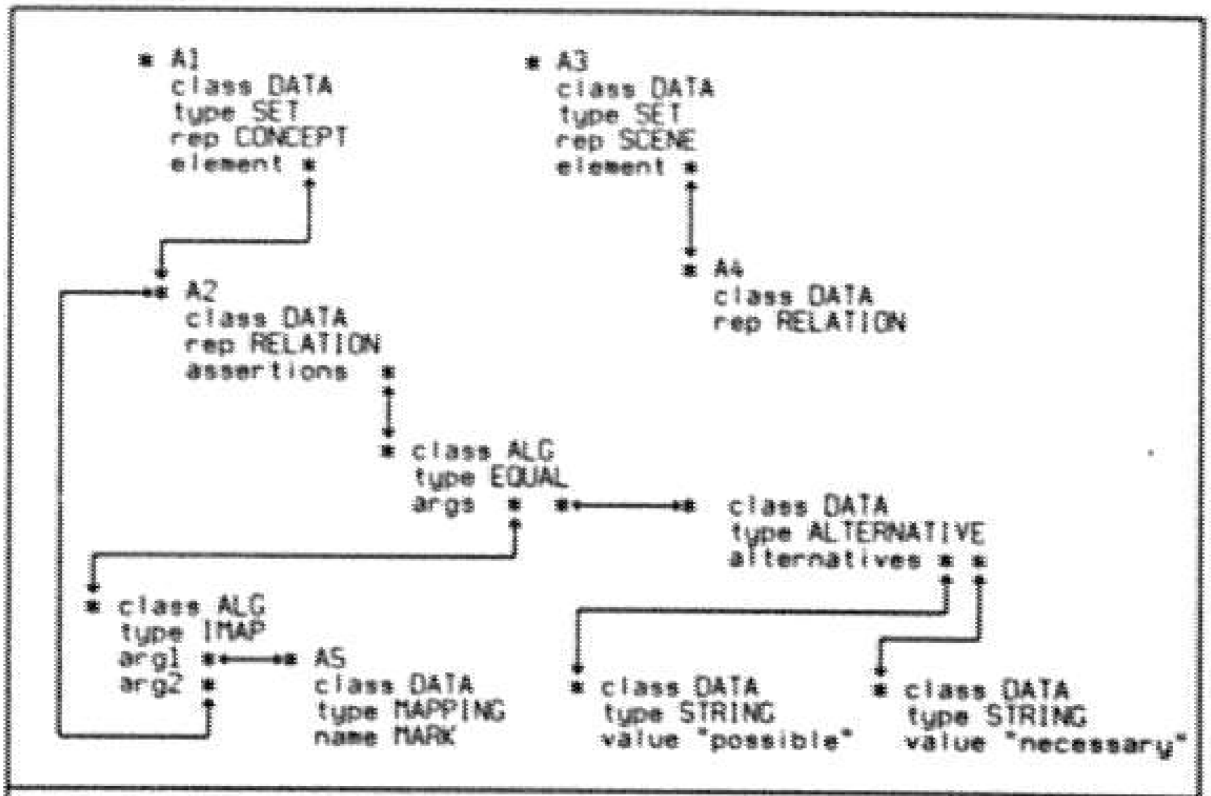


Figure 5.12

Scenes, concepts and relations

The user says, "Print the relations in the concept which are marked 'possible'", which is parsed to,

```
(1#PRINT NN
        [STEPOF YOU*]
        [ARGS (RELATION IPL THE (IN (CONCEPT THE))
                                (1#MARK PN
                                        [ARG2 !match_to_head_noun]
                                        [ARG3 "possible"]
                                ))]
)
```

The interpreter must find (or create) a component which can be used as the ARGS of the #OUTPUT 1#PRINT maps to. If the noun group were simply "the relations", the interpreter would match it to A1 or A3, whichever was mentioned last. But in this case, there are modifiers which will presumably narrow down the choice.

The first modifier is the prepositional phrase "in the concept". The #DATA concept (Figure 5.6) is used to determine the meaning of the modification. It is (#MEMBER A6 A1) where "the concept" has been matched to A1 and A6 is being used to represent the DATA which will be the final answer to the match. #MEMBER is treated as a special case in the matching process. The first #MEMBER in the modifier list which is not negated[3], and whose ARG1 is the noun in question, is transformed to the descriptor-slot/value pair of (ELEMENTOF X) where X is the ARG2 of the #MEMBER. So in this case, the #MEMBER is resolved to (ELEMENTOF A1). Following the ELEMENT slot of A1 leads to A2 which becomes the only match possibility. If there were no more modifiers, the match process would return A2 as the "relations in the concept".

The next modifier is a #MAP. The post condition of #MAP (Figure 5.5) is filled in with the #MAP descriptors, yielding, (#EQUAL (#IMAP A5 A6) "possible"). If this did not contradict the assertion list of A2, then A2 would be returned as the meaning of

[3] In "The relations which are not in the concept", the meaning of the prepositional modification is (#NOT (#MEMBER A6 A1)), which is inserted in the assertions list.

the noun phrase. It does, though, since the the assertion list of A2 asserts that a relation in the concept may be marked either "possible" or "necessary". Therefore a new component must be created, one which is the generic element of a subset of A1 which consists of all relations marked "possible". This is accomplished via the SUBSET module, which is another example of a small bit of knowledge being bound up in a procedure. The SUBSET module takes a set and an assertion list and creates a COMPUTE component which builds the subset. The COMPUTE created is shown in Figure 5.12

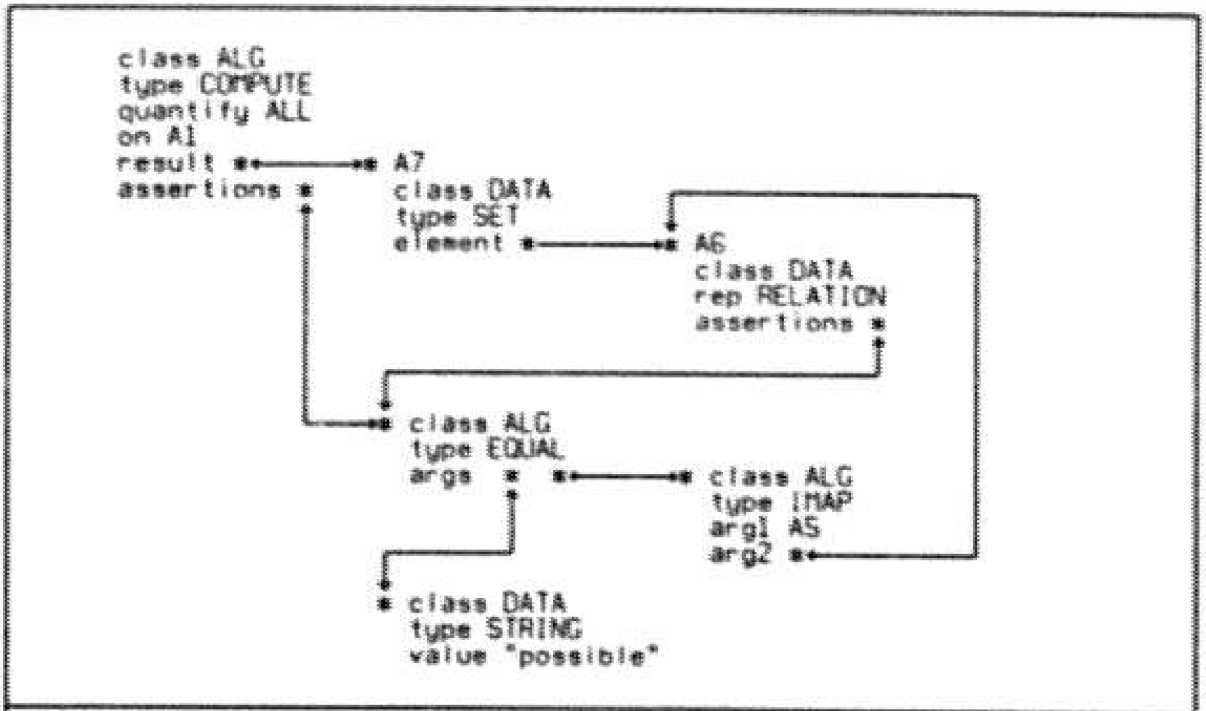


Figure 5.12

The COMPUTE for "The relations in the concept marked 'possible'."

A6 is the result of the matching process. The COMPUTE is inserted into the program specification when the "print" OUTPUT component is.

5.4.2 Pronouns

As we have indicated, the difference between pronoun reference and noun reference is in the possible match set. The interpreter keeps track of two special components, the *Focus* and *Data Focus*, which are used to help reduce the number of pronoun match possibilities.

When the program designer begins his reply, the *Focus* refers to the object of the question. During the processing of the program designer's reply, the *Focus* changes, so that it always points to the last component modified by the interpreter. We are making a distinction between "modifying" and "creating" a component. For example, the phrase, "it tests the concept", will cause a CALL component to be created with ARGS "concept"; we do not consider the CALL component to have been modified until some of its other descriptors (eg., PROCEDURE) have been filled. The *Data Focus* is the last DATA component which has been modified, described as a part of another DATA, or used as the ARGS or ARG1 of an ALGorithm component. The rules for the *Focus* and the *Data Focus* have been selected so that they are the most likely referents for any pronouns used by the program designer. Of course, they still must satisfy the requirements of the descriptor they are being proposed for. If they don't, the interpreter falls back on searching for a referent from the pronoun reference list, which is a list of each component that has been mentioned by the program designer.

We can see how this works on the following question/reply pair:

PSI: Describe the program.

USER: It reads a scene, tests whether it fits the concept, verifies the result of this test with the user, and updates the concept. Then it repeats the process.

The question sets the *Focus* to "program". The first "it" is matched to the *Focus* since "input" requires that it's SUB be an ALGorithm. The *Data Focus* is set to the "scene" because "scene" is the ARGS of the most recently created ALGorithm component (the INPUT). The second "it" is matched to the *Data Focus*, since the *Focus* is not a DATA (as is required by the ARGS of "fit"). The third "it" is matched to the *Focus*, since the STEPOF of "repeat" must be an ALGorithm. Note that none of "test", "verify", or "update" were proposed as referents for the third "it", even though they are all ALGorithm components. If there is no reason not to use the *Focus* or *Data Focus* as the referent, no other possibilities are checked.

When the *Data Focus* and the *Focus* both refer to DATAs, the preference checks given in the concepts are used to choose from between the two. Consider the dialogue fragment below:

> The two major data structures in the program are the concept and the scene. The concept is a set, which is read at the start of the program. The scene has two parts. The first part is a name. The second part is a list.
>
> 1. It should be read in after the concept.
> 2. It consists of three elements.

Either sentence 1. or 2. can logically follow the preceding paragraph, yet the "it" in 1. refers to the "scene", which is the *Focus*, and the "it" in 2. refers to the "list", which is the *Data Focus*. In 1., the choice between the two is resolved by the CHECK2 of #INPUT. The check prefers that the ARGS of #INPUT should not be parts of other components, or ARGS of an already instantiated #INPUT. Since the "list" is part of the scene, the "scene" is preferred as the referent. A similar process is used to find "list" as the proper match in 2. The definition of "consists" that succeeds is one that assigns the structure of the OBJ to the SUB. Naturally, it

prefers that its SUB have either no structure, or a structure which does not conflict with the OBJ. Since "scene" is known to be a RECORD with two fields, "list" is preferred for the match.

The methods we use for resolving reference amount to a heuristic filtering of possible referents (the *Focus* and *Data Focus*) followed by type checking on the surviving candidates. It works because the objects in our domain are easily classifiable, as are the effects (represented by which slots the objects have filled) of various actions upon them. Furthermore, the fact the we are talking about programming severely limits the different number of contexts things can be said in, which means that the preference checks associated with each component are likely to be consistently correct. Also, a conscientious program designer will probably find himself not using pronouns when he is intentionally violating these preferences. For instance, if one really wanted to write a program in which the "it" in 1. referred to the "name", he would find himself saying, "The name should be input after the concept".

For difficult reference problems, the Interpreter relies on the power of the situational checks associated with each concept's descriptors. Section 1.5.2 provided an example of their use in noun reference. In some respects, the situational checks are equivalent to methods proposed in other systems. [Hobbs 77] presents a system in which some pronoun reference is achieved by "detecting intersentence relations". One such relation is,

> A sentence asserts a change, and the following sentence presupposes the final state of that change.

When there is a reference problem, it is resolved in a way which realizes an

intersentence relation. The relation above helps match the "it" in 1., 2. and 3. below,

1. Decrease N by 1. If it is 0, reset it to MAX.
2. Decrease N by J. If it is 0, reset it to MAX.
3. Subtract J from N. If it has thereby gone down to 0, reset it to MAX.,

since N was changed in the first sentence and the second sentence has assumed (via the "if") the final state of "it". If "it" is matched to "N", the pattern holds, if it is matched to either "1" or "J", it does not.

The interpreter achieves the same effect by associating a situational check with the ARGS of #EQUAL which prefers that one of the ARGS be a variable whose value has been changed. Advocating such rules lays one open to charges of "ad hockery", but the situational checks are used for both noun and pronoun reference, as well as the parser/interpreter interface. When an individual check seems obscure, it is only because it reflects something which people rarely think consciously about. It is true, of course, that the situational checks currently associated with each concept are not now complete enough to handle all the reference problems one might encounter. However, the system's heuristics enable it to cope nicely with reference problems it must handle without complete information. For instance, even though the three sentences from [Hobbs *77*] were chosen to break the usual pronoun heuristics (the first introduces the problem, the second refutes the "0 shouldn't equal 1" method, and the third disproves the "positional" hypothesis), the interpreter would have found the correct referent in each case with the #EQUAL situational check omitted. The *Data focus* in all three sentences is "N", since it is the ARG1 of the most recently created component (the

SUBTRACT), and in the absence of any other information, it would be chosen as the referent of "it".

5.4.3 Matching to implicitly mentioned components

Often, the interpreter will have to match to a component which has been implicitly mentioned by the user. A simple example of this can be seen in the phrase,

"...classify the scene and print the result."

"Result" refers to the result of the classification. The methods described above would simply look for a component indexed by result, and not finding one, would create a new component as the result of the match. The solution is to do a little preprocessing before the matching process begins. Whenever a component is created which has a result, (in the example sentence, the CALL component created by "classify") a DATA component is instantiated, and then indexed through "result" and its synonyms, as well as any default indexing set up by the verb's definition (eg., "classification" for "classify", as shown in Figure 5.8)

A more subtle example occurs during proposed interchanges between the desired program and its user. Consider what might follow the sentence,

"I'll request a story by typing a key word".

The program designer might say nothing, in which case the system should ask how the request should be answered. Or, the user might follow immediately with a description of how the request should be handled. And finally, the user might just say what the "reply" should be. In that case, it is up to the system to realize that "reply" refers to the answering process, and that the "reply" should be printed out.

Verbs which imply an interchange of data between the program (eg., ask, request, answer, etc.) are mapped into #INTERCHANGE concepts. #INTERCHANGE concepts are represented in the specification by a SEQ with the appropriate steps. The SEQ is set up by a procedure associated with #INTERCHANGE. When the program is asking something of the user, the procedure's execution results in a SEQ whose first step is an OUTPUT component. A data is created which is indexed to "reply" (and "reply" synonyms) and a background job is set up to complete the SEQ if the user says nothing further. Completing the SEQ consists of setting up an INPUT component whose ARGS is the "reply" data set up by the #INTERCHANGE procedure. If the program is responding to a user query, the #INTERCHANGE procedure sets up a SEQ whose first step is an INPUT along with a "reply" DATA. A slightly different background program is used, however, which sets up a SEQ which takes care of the processing required to answer the user's query. The #INTERCHANGE background job does nothing if the "reply" data has been used as the ARGS of a last INPUT or OUTPUT of the #INTERCHANGE SEQ. This machinery allows the interpreter to handle the following examples:

> "Output the result of the test, ask the user if this is correct, and read in the user's response."

In this example, the designer has followed the #INTERCHANGE ("ask") with a description of the remainder of the #INTERCHANGE. "Response" matches to the "reply" DATA set up by the #INTERCHANGE procedure and the dialogue continues. The #INTERCHANGE background does nothing since the "reply" data is in the ARGS of an INPUT (the "read"). If the user had said only, "...and ask the user if this correct.", the background job would have been called to create an INPUT with the "reply" DATA as ARGS.

An example of a user initiated #INTERCHANGE is,

> PSI: Describe the program.
>
> USER: It has a data base of news stories. Each story has a set of key words associated with it. I'll request a story by giving a key word. The response should be all the stories with that key word.

"Request" sets up an #INTERCHANGE. "Response" is matched to the "reply" DATA and the background program sets up an OUTPUT to print the "response" (as defined by the program designer) to the user.

5.4.4 Coercion

The type restrictions implemented in the definitions and concepts are too strict to account for casual language usage. People often refer to an object by one of its parts, to a part of an object by the entire object, to an attribute of an object by the object, etc. The interpreter must be able to "coerce" the component the user has specified into the one he really meant, eg., the one which satisfies the type constraints of the descriptor slot being filled.

For instance, suppose the user defines a graph as "a set of nodes and a mapping which maps a pair of nodes into an edge." The interpreter assumes that a graph is a record with two fields, a set and a mapping. Then if the user mentions "the nodes in the graph", the interpreter, if using a strict interpretation of type restrictions, will fail to understand, since the meaning of "in" leading to #MEMBER requires that its object be a #SET. This is just a specific case of the more general "If X is a record and fails to satisfy a type check, the speaker may have intended one of the fields of X".

The Interpreter's type checking is implemented through the function ISA and the more complex secondary checks. ISA returns False if its object fails to satisfy the check, and a component if the object satisfies the check. The component may be the original object, or, if the object fails to satisfy the type but can be coerced into it, the component resulting from the coercion. Thus if (ISA X #SET) is evaluated and X is record structure with a field whose DATA is the set Y, then the result of the evaluation will be Y and Y will be used to fill the descriptor slot.

This type of matching allows the interpreter's matching rules to be written with a great deal of flexibility. In section 1.5.2, we used,

> "It reads in a trial-item, matches the *input* to the internal concept model, and prints the result of the match."

to illustrate how *input* is matched to "trial-item" rather than "the read "input" operation" because of the requirement that the ARGS of "match" be a #DATA. It is actually implemented through the coercion feature. In the absence of a component being explicitly referred to as an "input", the matching process looks for an #INPUT operation. When an INPUT is found, and is required to be a #DATA, ISA returns the ARGS of the INPUT.

5.5 The Reader/Interpreter interface

The Reader function Format is the interface between Reader and the Interpreter. Section 4.1 listed the criteria used by Format to supply each parse structure with a measure. Reader uses the measures to choose from among competing parse structures. The information required for measuring is,

1. Does the verb have all its required cases?
2. Are the case contents of the verb understandable?

3. Do the case contents satisfy the case requirements?

The Interpreter supplies the measure information through its concepts and definitions. Whether a verb has all its cases can be read directly from the definition. If it is missing cases the definition has marked "Must", the rating is unacceptable. If it has all the Must cases, but is missing cases marked "Prefered", the rating is acceptable. Otherwise it is perfect.

Determining whether the case contents are understandable consists of checking that the meaning of all modifications in the case contents are covered by definitons. If they are not all covered the rating is unacceptable. If they are covered, but not all contextual checks in the relevant definitons are satisfied, the rating is acceptable. Otherwise it is perfect.

Checking that the case contents of a verb satisfy the verb's case requirements makes use of the descriptor checks in the concept the verb is being mapped to. If the case satisfies the first check it is acceptable. If it satisfies the the second check, then it is perfect. Otherwise, the case is unacceptable.

The remainder of this section consists of three examples illustrating how the three different measure parts are used to affect the parsing process.

In the sentence "The program stores and retrieves data.", "data" should be viewed as the object of "store" as well as "retrieves". As we noted in 4.3.3, this depends on the meanings of "store" and "data", and is not true for all sentences with this syntax. The parser decides whether to use "data" as the OBJ of "store" depending on which is better, the measure of "The program stores", or the measure

of "The program stores data." The measure of the latter is better since the definition of "store" states that the OBJ case is preferred, and "data" does not violate the case preferences of "store".

For an example of the case preferences at work, consider the sentence, "if the scene fit and the user said the guess was 'correct, then every....". The clause introduced by "if" has two syntactic readings, namely

```
[IF (CONJ AND                          or      [IF (SAY PN
      (FIT PN                                        [SUB AND (FIT THE SCENE)
          [SUB (SCENE THE)]                                (USER THE)]
      )                                              [WHAT (BE PN
                                                             [SUB (GUESS THE)]
     (SAY PN                                                 [OBJ "Correct"]]
         [SUB (USER THE)]                          )]
         [WHAT (BE PN
                  [SUB (GUESS THE)]
                  [OBJ "Correct"]]]
     )
    )]
```

the definition of "say" which maps to #INPUT requires that the SUB case satisfies the check (ISA #IO-DEVICE). This gives the first parse a better measure than the second, since the SUB of the second includes "fit" as part of its compound SUB, and "fit" cannot be viewed as #IO-DEVICE.

The noungroup "each relation in the concept which is in the scene." provides an example of the "understandability" criteria. There is no a priori reason for it to mean

```
[NOUN (RELATION EACH (IN (CONCEPT THE))
                     (!=BE_NN
                          [ARG1 !match_to_head_noun]
                          [ARG2 (SCENE THE)]
                     ))]
```

rather than

```
[NOUN (RELATION EACH (IN (CONCEPT THE (!=BE_NN
                                           [ARG1 !match_to_head_noun]
                                           [ARG2 (SCENE THE)]
                                      ))))]
```

But if scenes, concepts and relations had been defined as shown in Figure 5.12, the first parse would obviously be correct. The first modification in each is perfect. The reason is that "relation" is a #DATA (Figure 5.6), hence there is a meaning for it to be modified by a prepositional phrase whose preposition is "in". The meaning of the modification is #MEMBER, and "concept" satisfies both #MEMBER checks; it is a set, and its generic element is a "relation". The second modification in the first parse is also perfect. 1#BE maps to #MEMBER, and "scene" satisfies both checks. The second modification of the second parse is only acceptable, however, since it fails the second #MEMBER check since "concepts" cannot be viewed as the generic element of the scene.

5.6 Future work

5.6.1 Tense evaluation

The interpreter makes almost no use of the tense information returned by the parser. This does not affect its performance greatly, as the dialogues it has handled have all been straightforward (with no skipping about into the future or past) linear algorithm descriptions.

But it is easy to see how the proper interpretation of tense information is necessary for understanding even the types of dialogues we have been considering.

In "Set X to the tail of X. If the head of X is/was 5, then ..." the use of "is" or "was" determines whether the program designer means the first or second element of the original X.

Similarly, in

> "Test if the scene fit the concept and print "fits" if it does. Then modify the concept. If the scene fits/fit the concept..."

the use of "fit" or "fits" determines whether the "fit" predicate should be recalculated for the new modified concept, or whether the old value should be accessed.

5.6.2 More domain and general programming support

Programming and domain knowledge is necessary for several reasons. A system well versed in programming and domain knowledge will ask fewer unnecessary questions of of the user, thereby making for a more practical system. A well informed system will also be able to follow the program designer that much more easily.

For instance, if the designer says,

> "Write me a program which sorts a list of words. The comparison function should be alphabetical order.",

understanding the second sentence requires knowing something about sorting programs. Information like this will be forthcoming from the two PSI modules concerned with domain and general programing support. The modules and the interface between them and the interpreter are being developed.

5.6.3 Building up more concepts and definitions

Expanding the interpreter's collection of concepts and definitons is the most obvious improvement that can be made to the system. It is impossible for the interpreter to understand a primitive idea unless it has a concept to represent that thought. Thus a simple sentence like "Print the greatest number in the list" cannot be understood unless the system has the concepts #GREAT and #SUPERLATIVE. And if it can understand that sentence, the interpreter still won't be able to understand, "Print the number in the list which is larger than any other number in the list" unless it has definitons which map "larger" into #GREAT and "any other number" into a #SUPERLATIVE.

However, with the proper concepts and definitions, which are easy to write, the interpreter can understand these sentences and many more. By having people exercise the system, and then teaching the system any unknown concepts and definitions which have been used, we hope to build up a collection of concepts and definitions which will be comprehensive enough to support most reasonable dialogues. Appendix A contains dialogues illustrative of the system's current capabilities.

6. References

[Balzer 75]
Balzer, R., *Imprecise Program Specification*, Technical Report RR-75-36, USC/Information Science Institute, Marina Del Rey, Califoria, 1975.

[Barstow 77]
Barstow, D., *A Knowledge-based System for Automatic Program Construction*, **Proceedings of the Fifth International Joint Conference on Artificial Intelligence**, 1977.

[Bobrow 76]
Bobrow, D. and Winograd, T., *An Overview of KRL, a Knowledge Representation Language*, Memo 293, Stanford A. I. Project, Stanford University, 1976.

[Brooks 74]
Brooks, M., *Another Approach to English*, Working Paper 73, MIT Artificial Intelligence Labratory, 1974.

[Bruce 72]
Bruce, B., *A Model for Temporal References and its Application in a Question Answering Program*, **Artificial Intelligence**, Volume 3: Number 1, 1972.

[Bruce 75]
Bruce, B., *Case Systems for Natural Language*, **Artificial Intelligence**, Volume 6: Number 4, 1975.

[Fillmore 68]
Fillmore, C., *The Case for Case*, in **Universals in Linguistic Theory**, Eds. Bach, E. and Harms, R., Holt, Rineheart and Winston, New York, 1968.

[Gardner 76]
Gardner, M., **Scientific American**, June 1976, pages 120-125.

[Green 76]
Green, C., *The Design of the PSI Program Synthesis System*, in **Second International Conference on Software Engineering**, San Francisco, CA., October, 1976.

[Green 77]
Green, C., *The Design of the PSI Program Synthesis System*, **Proceedings of the Fifth International Joint Conference on Artificial Intelligence**, 1977.

[Grishman 76]
Grishman, R., *A Survey of Syntactic Analysis Procedures*, **American Journal of Linguistics**, Microfiche 47, 1976.

[Heidorn 74]
Heidorn, G., *English as a Very High Level Language for Simulation Programming*, **Proceedings of a Symposium on Very High Level Languages, Sigplan Notices**, Vol. 9, No. 4, 1974.

[Heidorn 76]
Heidorn, G., *Automatic Programming Through Natural Language: A Survey*, **IBM Journal of Research and Development**, Vol. 20, No. 4, 1976.

[Hobbs 77]
Hobbs, J., *From "Well-written" Algorithm Descriptions into Code*, Research Report #77-1, Department of Computer Sciences, City University of New York, July, 1977.

[Kant 77]
Kant, E., *The Selection of Efficient Implementations for a High Level Language*, **Proceedings of Symposium on Artificial Intelligence and Programming Languages**, SIGPLAN Notices, Volume 12, Number 8, SIGART Newsletter, Number 64, August 1977.

[Kuno 63]
Kuno, S., and Oettinger, A., *Multiple Path Syntactic Analyzer*, in **Information Processing**, North-Holland Publishing Co., Amsterdam, 1963.

[McCune 77]
McCune, B., *The PSI Program Model Builder: Synthesis of Very High-level Programs*, **Proceedings of Symposium on Artificial Intelligence and Programming Languages**, SIGPLAN Notices, Volume 12, Number 8, SIGART Newsletter, Number 64, August 1977.

[McCune 78]
McCune, B., *Building Program Models Incrementally from Informal Descriptions*, Ph.D. thesis, AI Memo, CS Report, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, to appear.

[Malhotra 75]
Malhotra, A., *Design Criteria for a Knowledge-Based English Language System for Management: An Experimental Analysis*, Technical Report TR-146, Project MAC, MIT, Cambridge Massachusetts, 1975.

[Manna 77]
Manna, Z., and Waldinger, R., *Synthesis: Dreams => Programs* Memo 302, Stanford A. I. Project, Stanford University, 1977.

[Marcus 75]
Marcus, M., *Diagnosis as a Notion of Grammar*, in **Proceedings of a Workshop on Theoretical Issues in Natural Language Processing**, Eds. Schank, R. and Nash-Weber, B., Cambridge, Mass., June, 1975.

[Phillips 78]
Phillips, J., *The use of inference in automatic programming systems*, AI Memo, CS Report, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, to appear.

[Riesbeck 74]
Riesbeck, C., *Computer Analysis of Natural Language in Context*, Memo 238, Stanford A. I. Project, Stanford University, 1974.

[Rieger 74]
Rieger, C., *Conceptual Understanding: A Theory and Computer Program for Processing the Meaning Content of Natural Language Utterances*, Memo 233, Stanford A. I. Project, Stanford University, 1974.

[Robinson 75]
Robinson, J., *A Tuneable Performance Grammar*, SRI Artificial Intelligence Center Technical Note 112, 1975.

[Sager 73]
Sager, N., *The String Parser for Scientific Literature*, in Rustin, R., Ed., **Natural Language Processing**, Algorithmics Press, 1973.

[Steinberg 78]
Steinberg L., *A Dialogue Moderator for Program Specification Dialogues in the PSI System*, Ph.D. thesis, AI Memo, CS Report, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, to appear.

[Stockwell 73]
Stockwell, R., Schachter, P. and Partee, B., *The Major Syntactic Structures of English*, Holt, Rinehart and Winston, INC., 1973.

[Wilks 73]
Wilks, Y., *Preference Semantics*, Memo 206, Stanford A. I. Project, Stanford University, 1973.

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

[Winston 75]
Winston, P., *Learning Structural Descriptions from Examples*, in Winston, P., Ed. **The Psychology of Computer Vision**, McGraw-Hill Book Company, Inc., 1975.

[Woods 70]
Woods, W., *Network Grammars for Language Analysis*, **Communications of the ACM**, Voulume 13, Number 10, October 1970.

[Woods 72]
Woods, W., Kaplan R. and Nash-Weber B., *The Lunar Sciences Natural Language Information System*, BBN Report No. 2378, 1972.

[Woods 73]
Woods, W. *An Experimental Parsing System for Transition Network Grammars*, in Rustin, R., Ed. **Natural Language Processing**, Algorithmics Press, 1973.

Appendix A. Example Dialogues

Here we present a number of dialogues handled by the system. Some are target dialogues for PSI system and others were written to illustrate a point in natural language processing. Each dialogue is followed by the program specification the interpreter produced from it.

The first dialogue is one of the original PSI target dialogues. In it, the user wishes to write a program similar to one described in [Winston 75]. The program, known as a concept formation program, is "...a type of simple learning program that takes as input several objects that are instances of some concept and also several objects that are not instances of the concept. As output, it produces an abstracted description of the concept so that given a new instance, it can classify that instance as representing the concept or not" [GREEN 76]. The second dialogue is another version of the same program. The next three dialogues are PSI target dialogues which are simpler versions (from a coding standpoint) of concept formation programs.

Dialogues six and seven are two different ways of specifying a graph traversal algorithm. The first is a description of what the algorithm is to accomplish. The second describes how the algorithm should be implemented.

The last four dialogues are all variants of information retrieval programs.

-DIALOGUE 1-

WHAT IS THE NAME OF THE PROGRAM YOU WISH TO WRITE? TF

DOES TF HAVE A SPECIAL PROGRAM TYPE? Theory formation

DESCRIBE TF
It inputs a scene, tests whether it fits the concept, verifies the result of this test with the user, and updates the concept. Then it repeats the process.

WHAT IS THE STRUCTURE OF THE SCENE?
An input scene will just be a collection of relations, each of which is a property name followed by the names of the objects over which the property holds.

WHAT IS THE STRUCTURE OF THE CONCEPT?
It should be a set of relations, each marked either "possible" or "necessary".

WHAT IS THE INITIAL VALUE OF THE CONCEPT?
Initially the concept has no descriptions at all.

PLEASE DEFINE WHAT IT MEANS FOR A SCENE TO FIT A CONCEPT.
The scene fits the concept if all of the relations in the concept that are marked "necessary" are part of the scene.

DESCRIBE VERIFYING THE CONCEPT.
Output the result of the test, ask the user if this is correct, and input the user's response.

WHAT IS THE ARGS OF THE ASK?
"correct or wrong"*

WHAT IS THE STRUCTURE OF THE RESPONSE?
The response should be "correct" or "wrong".

DESCRIBE UPDATING THE CONCEPT.
There are four cases. If the scene fit and the user said the guess was correct, then every relation that is part of the scene but not part of the concept should be added to the concept and marked "possible"

DESCRIBE THE NEXT CASE.
The next case is if the scene fit and the user said "wrong", then find a relation in the concept which is marked "possible" and which is not in the scene. Change its marking to "necessary".

DESCRIBE THE NEXT CASE.
If the scene did not fit, and the user said "correct", then do nothing.

DESCRIBE THE NEXT CASE.
If the scene did not fit, and the user said "wrong", then for each relation in the concept marked "necessary" that is not in the scene, re-mark it "possible".

WHAT IS THE FORMAT OF THE OUTPUT?
Output "fit" if the scene fit, "didn't fit" if it didn't.

WHAT IS THE EXIT TEST OF THE LOOP?
Stop when the user types "quit" instead of an input scene.

ARE WE FINISHED? yes

```
TF
  R1 ← PHI
  LOOP1:
     PRINT("Ready for the SCENE")
     R2 ← READ()
     IF EQUAL(R2 "quit")
       Then GO EXIT1
     R4 ← F1T(R2 R1)
     Cases:  IF R4
               Then PRINT("I.I")
        else IF NOT(R4)
               Then PRINT("I don't I.")
     PRINT("correct or wrong?")
     R5 ← READ()
     Cases:  IF AND(R4 EQUAL(R5 "correct"))
               Then R11 ← The set of all R1B in R2 such that:
                         NOT(MEMBER(R1B R1))
                    For all R1B in R11 do:
                         R1 ← INSERT(R1B R1)
                         MAP(R3 R1B "possible")
        else IF AND(R4 EQUAL(R5 "wrong"))
               Then R7 ← The set of any 1 R6 in R1 such that:
                         AND(NOT(MEMBER(R6 R2))
                             EQUAL(MAP(R3 R6) "possible"))
                    For all R6 in R7 do:
                         MAP(R3 R6 "necessary")
        else IF AND(NOT(R4)
                    EQUAL(R5 "correct"))
               Then NIL
        else IF AND(NOT(R4)
                    EQUAL(R5 "wrong"))
               Then R9 ← The set of all R8 in R1 such that:
                         AND(EQUAL(MAP(R3 R8) "necessary")
                             NOT(MEMBER(R8 R2)))
                    For all R8 in R9 do:
                         MAP(R3 R8 "possible")
     Goto LOOP1
  EXIT1:

F1T(B1 B2)
  FORALL(B3) IMPLIES(AND(EQUAL(MAP(B3 F1) "necessary")
                         MEMBER(B3 B2))
                     MEMBER(B3 B1))
```

R5 is either a string whose value is "wrong" or a string whose value is "correct".

R4 is either TRUE or FALSE. B1 is a variable bound to R2.

R3 is a mapping from the elements of R1 to either a string whose value is "necessary" or a string whose value is "possible".

B3 is a variable bound to R12. B2 is a variable bound to R1.

R2 is either a set whose generic element is a record whose fields are PROPERTY: a primitive name and NAME-SET: a set whose generic element is a primitive name or a string whose value is "quit".

R1 is a set whose generic element is a record whose fields are PROPERTY: a primitive name and NAME-SET: a set whose generic element is a primitive name with assertions:
EQUAL(MAP(R3 R12) R13)

R13 is either a string whose value is "necessary" or a string whose value is "possible".

R12 is the generic element of R1.

-DIALOGUE 2-

WHAT IS THE NAME OF THE PROGRAM YOU WISH TO WRITE?
Arch.

DOES ARCH HAVE A SPECIAL PROGRAM TYPE?
Theory formation.

DESCRIBE ARCH
The program is a loop. On each iteration, it reads in a trial-item, matches the input to the internal concept model, and prints the result of the match. The user must then inform the program whether or not the guess is correct, and the program modifies its model of the concept accordingly.

WHAT IS THE STRUCTURE OF THE TRIAL-ITEM?
A trial-item is a set of relations.

WHAT IS THE STRUCTURE OF THE MODEL?
The model is a set of relations, each marked "necessary" or "possible"

WHAT IS THE STRUCTURE OF THE RESPONSE?
"correct" or "wrong"

WHAT IS THE STRUCTURE OF THE RELATIONS IN THE TRIAL-ITEM?
a relation is a property name and a list of atoms.

WHAT IS THE INITIAL VALUE OF THE MODEL?
Nil.

PLEASE DEFINE WHAT IT MEANS FOR A SCENE TO MATCH A CONCEPT.
A trial-item matches the model if all the relations in the model that are marked "necessary" are elements of the trial-item.

WHAT IS THE FORMAT OF THE PRINT?
Type "the trial-item matches" if the trial-item matched and "the trial-item does not match" if it didn't.

DESCRIBE MODIFYING THE MODEL
If the trial-item matched and the user said the guess was correct, then every relation in the trial-item which is not in the model should be put in the model and marked "possible". If the trial-item matched and the user said "wrong" , then change the marking of a relation marked "possible" which is in the model and not in the trial-item to "necessary". If the trial-item didn't match, and the user said "correct" , then do nothing. If the trial-item did not match, and the user said "wrong" , then re-mark each relation in the model marked "necessary" that is not in the trial-item "possible".

WHAT IS THE EXIT TEST OF THE LOOP?
stop when the user types "quit" instead of a trial-item.

ARE WE FINISHED? yes.

```
SEEK
  R1 ← PH1
  LOOP1:
     PRINT("Ready for the TRIAL-ITEM")
     R2 ← READ()
     IF EQUAL(R2 'quit')
       Then GO EXIT1
     R4 ← MATCH(R2 R1)
     Cases:  IF R4
               Then PRINT("the trial-item matches")
       else IF NOT(R4)
               Then PRINT("the trial-item does not match")
     R5 ← READ()
     Cases:  IF AND(R4 EQUAL(R5 'correct'))
               Then R11 ← The set of all R10 in R2 such that:
                          NOT(MEMBER(R10 R1))
                    For all R10 in R11 do:
                          R1 ← INSERT(R10 R1)
                          MAP(R3 R10 'possible')
       else IF AND(R4 EQUAL(R5 'wrong'))
               Then R7 ← The set of any 1 R6 in R1 such that:
                          AND(NOT(MEMBER(R6 R2))
                              EQUAL(MAP(R3 R6) 'possible'))
                    For all R6 in R7 do:
                          MAP(R3 R6 'necessary')
       else IF AND(NOT(R4)
                   EQUAL(R5 'correct'))
               Then NIL
       else IF AND(NOT(R4)
                   EQUAL(R5 'wrong'))
               Then R8 ← The set of all R6 in R1 such that:
                          AND(EQUAL(MAP(R3 R6) 'necessary')
                              NOT(MEMBER(R6 R2)))
                    For all R6 in R8 do:
                          MAP(R3 R6 'possible')
     Goto LOOP1
  EXIT1:

MATCH(B1 B2)
  FORALL(B3) IMPLIES(AND(EQUAL(MAP(R3 B3) 'necessary')
                         MEMBER(B3 B2))
                     MEMBER(B3 B1))

R5 is either a string whose value is "wrong" or a string whose value is "correct".

R4 is either TRUE or FALSE.  B3 is a variable bound to R12.

R3 is a mapping from the elements of R1 to either a string whose value
is "possible" or a string whose value is "necessary".

B2 is a variable bound to R1.  B1 is a variable bound to R2.

R2 is either a set whose generic element is a record whose fields are
PROPERTY: a primitive name and ATOM-LIST: a list whose generic element
is a primitive or a string whose value is "quit".

R1 is a set whose generic element is a record whose fields are PROPERTY:
a primitive name and ATOM-LIST: a list whose generic element is a
primitive with assertions:
      EXISTS(B4) EQUAL(MAP(R3 B4) R13)

R13 is either a string whose value is "possible" or a string whose value is "necessary".

B4 is a variable bound to R12.  R12 is the generic element of R1.
```

-DIALOGUE 3-

WHAT IS THE NAME OF THE PROGRAM YOU WISH TO WRITE? CLASS

DOES CLASS HAVE A SPECIAL PROGRAM TYPE? No

DESCRIBE CLASS.
CLASS first inputs a concept. Then it repeatedly accepts an input scene from a user, classifies it based on whether or not it fits the concept, and outputs this classification to the user.

WHAT IS THE STRUCTURE OF THE CONCEPT?
The concept will just be a collection of relations, each of which is a property name followed by the names of the objects over which the property holds.

WHAT IS THE STRUCTURE OF THE SCENE?
The scene has the same structure as the concept.

PLEASE DESCRIBE WHAT IT MEANS FOR A SCENE TO FIT A CONCEPT.
The scene fits the concept if all of the relations in the concept are part of the scene.

WHAT IS THE FORMAT OF THE OUTPUT?
Output "Fit" if the scene fit, "Didn't fit" if it didn't.

WHAT IS THE EXIT TEST OF THE LOOP?
Stop when the user types "Quit" instead of an input scene.

ARE WE FINISHED? yes

```
CLASS
  PRINT("Ready for the CONCEPT")
  R1 ← READ()
  LOOP1:
     PRINT("Ready for the SCENE")
     R2 ← READ()
     If EQUAL(R2 "Quit")
       Then GO EXIT1
     R3 ← FIT(R2 R1)
     Cases:  If R3
               Then PRINT("Fit")
        else If NOT(R3)
               Then PRINT("Didn't fit")
     Goto LOOP1
  EXIT1:

FIT(B1 B2)
  FORALL(B3) IMPLIES(MEMBER(B3 B2)
                     MEMBER(B3 B1))
```

R3 is either TRUE or FALSE. B1 is a variable bound to R2.

B3 is a variable bound to R4. B2 is a variable bound to R1.

R2 is either a set whose generic element is a record whose fields are PROPERTY: a primitive name and NAME-SET: a set whose generic element is a primitive name or a string whose value is "Quit".

R1 is a set whose generic element is a record whose fields are PROPERTY: a primitive name and NAME-SET: a set whose generic element is a primitive name.

R4 is the generic element of R1.

-DIALOGUE 4-

WHAT IS THE NAME OF THE PROGRAM YOU WISH TO WRITE?

Class1

DOES CLASS1 HAVE A SPECIAL PROGRAM TYPE?

no

DESCRIBE CLASS1.

It reads a concept. Then it repeatedly reads a string and tests if it is in the concept. If it is, it types "its in". Otherwise, it types "not in". The concept is a collection of strings.

WHAT IS THE EXIT TEST OF THE LOOP?

Stop when the user types "quit"

ARE WE FINISHED?

Yes.

```
CLASS1
  PRINT("Ready for the CONCEPT")
  R1 ← READ()
  LOOP1:
     PRINT("Ready for the STRING")
     R2 ← READ()
     IF EQUAL(R2 "quit")
       Then GO EXIT1
     R3 ← MEMBER(R2 R1)
     IF R3
       Then PRINT("its in")
       else PRINT("not in")
     Goto LOOP1
  EXIT1:
```

R3 is either TRUE or FALSE.

R2 is either a string or a string whose value is "quit".

R1 is a set whose generic element is a string.

-DIALOGUE 5-

WHAT IS THE NAME OF THE PROGRAM YOU WISH TO WRITE?

Class0.

DOES CLASS0 HAVE A SPECIAL PROGRAM TYPE?

no.

DESCRIBE CLASS0.

It reads a concept.

WHAT IS THE STRUCTURE OF THE CONCEPT?

a collection of strings.

ARE WE FINISHED?

Then it repeatedly reads a set of strings and prints "its a subset" if all the elements in the set are in the concept. Otherwise it prints "not a subset".

WHAT IS THE EXIT TEST OF THE LOOP?

Stop when the user types "quit".

ARE WE FINISHED?

Yes.

```
CLASS0
  PRINT("Ready for the CONCEPT")
  R1 ← READ()
  LOOP1:
     PRINT("Ready for the ELEMENT-SET")
     R2 ← READ()
     IF EQUAL(R2 "quit")
       Then GO EXIT1
     IF FORALL(B1) IMPLIES(MEMBER(B1 R2)
                           MEMBER(B1 R1))
       Then PRINT("its a subset")
       else PRINT("not a subset")
     Goto LOOP1
  EXIT1:
```

B1 is a variable bound to R3.

R2 is either a set whose generic element is a string or a string whose value is "quit".

R1 is a set whose generic element is a string.

R3 is the generic element of R2.

-DIALOGUE 6-

WHAT IS THE NAME OF THE PROGRAM YOU WISH TO WRITE? **Findnodes**

DOES FINDNODES HAVE A SPECIAL PROGRAM TYPE? **no**

DESCRIBE FINDNODES.

The program should input a graph and a node. A graph is a set of pairs. Each pair consists of two nodes, which are primitives. The program outputs a list of all the nodes which can be reached from the input node.

PLEASE DESCRIBE WHAT IT MEANS FOR A NODE TO BE REACHED FROM ANOTHER NODE.

A node X is connected to a node Y if there exists a pair in the graph such that X and Y are in the pair. X can be reached from Y if X is connected to Y or if X can be reached from a node which is connected to Y.

ARE WE FINISHED? **Yes**

```
FINDNODES
  PRINT("Ready for the GRAPH and the NODE")
  R1 ← READ()
  R2 ← READ()
  R5 ← The set of all R3 in all R4 in R1 such that:
       REACH(R3 R2)
  PRINT(R5)

REACH(B1 B2)
  OR(CONNECT(B1 B2)
     EXISTS(B6) AND(CONNECT(B6 B2)
                    REACH(B1 B6)))

CONNECT(B3 B4)
  EXISTS(B5) AND(MEMBER(B5 R1)
                 MEMBER(B3 B5)
                 MEMBER(B4 B5))
```

B6 is a variable bound to R6. B5 is a variable bound to R7.

B4 is a variable bound to Y. B3 is a variable bound to X .

B2 is a variable bound to Y. B1 is a variable bound to X

R4 is the generic element of R1. R2 is a primitive.

R1 is a set whose generic element is a set of size 2 whose generic element is a primitive.

X is a primitive. Y is a primitive.

R7 is a set of size 2 whose generic element is a primitive.

R6 is a primitive.

-DIALOGUE 7-

WHAT IS THE NAME OF THE PROGRAM YOU WISH TO WRITE? Reach.

DOES REACH HAVE A SPECIAL PROGRAM TYPE? No.

DESCRIBE REACH.
It reads a graph and a node. The graph is a set of nodes and a mapping which maps a pair of nodes into an edge. Nodes and edges are primitives. The program puts the input node into a set Nodes. Then it repeatedly finds a node in Nodes which is not marked "done", marks it "done", and places all the graph nodes connected to it which are not in Nodes into Nodes.

WHAT IS THE INITIAL VALUE OF NODES? NIL.

PLEASE DEFINE WHAT IT MEANS FOR A NODE TO BE CONNECTED TO ANOTHER NODE.
Two nodes are connected if they are mapped into an edge.

WHAT IS THE EXIT TEST OF THE LOOP?
Stop when there are no nodes in Nodes which are not marked "done".

ARE WE FINISHED? Output Nodes.

WHICH ONE OF EITHER REACH OR THE LOOP DOES THE OUTPUT BELONG IN? Reach.

```
REACH
   NODES ← PHI
   PRINT("Ready for the GRAPH and the NODE")
   R1 ← READ()
   R2 ← READ()
   NODES ← INSERT(R2 NODES)
   LOOP1:
      R4 ← The set of any 1 R3 in NODES such that:
          NOT(EQUAL(MAP(R5 R3) "done"))
      IF EQUAL(R4 PHI)
        Then GO EXIT1
      For all R3 in R4 do:
          MAP(R5 R3 "done")
      R8 ← The set of all R6 in R7 such that:
          AND(FORALL(B1) IMPLIES(MEMBER(B1 R4)
                                 CONNECT(R6 B1))
              NOT(MEMBER(R6 NODES)))
      For all R6 in R8 do:
          NODES ← INSERT(R6 NODES)
      Goto LOOP1
   EXIT1:
   PRINT(NODES)
```

```
CONNECT(B2 B3) EXISTS(B4) EQUAL(MAP(R5 (B2 B3) ) B4)
```

R9 is the DATA of the MAPPING field of R1. B4 is a variable bound to R18. R2 is a primitive.

B2 is a variable bound to R12. B1 is a variable bound to R3. B3 is a variable bound to R11.

R7 is the DATA of the NODE-SET field of R1. NODES is a set whose generic element is a primitive.

R5 is a mapping from the elements of R4 to a string whose value is "done"

R1 is a record whose fields are NODE-SET: a set whose generic element is a primitive and MAPPING: a mapping from a set of size 2 whose generic element is a primitive to a primitive.

R12 is a primitive. R11 is a primitive. R18 is a primitive.

-DIALOGUE 8-

WHAT IS THE NAME OF THE PROGRAM YOU WISH TO WRITE?

COOKBOOK

DOES COOKBOOK HAVE A SPECIAL PROGRAM TYPE?

no.

DESCRIBE COOKBOOK.

Cookbook inputs a recipe list, and then repeatedly inputs a name and prints the recipe with that name.

WHAT IS THE STRUCTURE OF THE RECIPES IN THE LIST?

A recipe has a name, an ingredient-list, and directions. The name and the directions are strings of characters. An ingredient-list is a list whose elements have an ingredient and an amount.

WHAT IS THE STRUCTURE OF THE INGREDIENT?

An ingredient and an amount are both strings of characters.

WHAT IS THE EXIT TEST OF THE LOOP?

Stop when the user types "quit" as a recipe name.

ARE WE FINISHED? Yes.

```
(COOKBOOK  NIL
   PRINT("Ready for the RECIPE-LIST")
   R1 ← READ()
   LOOP1:
      PRINT("Ready for the NAME")
      R2 ← READ()
      IF EQUAL(R2 "quit")
        Then GO EXIT1
      R4 ← The set of all R3 in R1 such that:
          EQUAL(R2 FETCH(R3 NAME))
      For all R3 in R4 do:
          PRINT(R3)
      Goto LOOP1
   EXIT1:
)
```

R2 is either a primitive name or a string whose value is "quit"

R1 is a list whose generic element is a record whose fields are NAME: a string , INGREDIENT-LIST: a list whose generic element is a record whose fields are INGREDIENT: a string and AMOUNT: a string , and DIRECTION-SET: a set whose generic element is a string.

-DIALOGUE 6-

WHAT IS THE NAME OF THE PROGRAM YOU WISH TO WRITE?

Recipe.

DOES RECIPE HAVE A SPECIAL PROGRAM TYPE?

no.

DESCRIBE RECIPE.

It reads in a recipe list. A recipe consists of a name and a string.
Then it enters a loop which reads a recipe name and prints the recipe
with that name. If there is no recipe with that name, print "no such recipe".

WHAT IS THE EXIT TEST OF THE LOOP?

Stop when the user types "quit"

ARE WE FINISHED?

Yes.

```
RECIPE
  PRINT("Ready for the RECIPE-LIST")
  R1 ← READ()
  LOOP1:
     PRINT("Ready for the NAME")
     R2 ← READ()
     If EQUAL(R2 "quit")
       Then GO EXIT1
     R4 ← The set of all R3 in R1 such that:
         EQUAL(FETCH(R3 NAME) R2)
     If EQUAL(R4 PHI)
       Then PRINT("no such recipe")
       else for all R3 in R4 do:
                PRINT(R3)
     Goto LOOP1
  EXIT1:
```

R2 is either a primitive name or a string whose value is "quit".

R1 is a list whose generic element is a record whose fields are NAME: a primitive name and STRING: a string.

-DIALOGUE 10-

WHAT IS THE NAME OF THE PROGRAM YOU WISH TO WRITE? **News.**

DOES NEWS HAVE A SPECIAL PROGRAM TYPE? **News retrieval.**

DESCRIBE NEWS.
It answers questions about a data base.

WHAT IS THE STRUCTURE OF THE GENERIC ELEMENTOF THE DATA-BASE?
The data base is a list of descriptors, each of which describes one newspaper story.

WHAT IS THE STRUCTURE OF THE DESCRIPTOR?
A descriptor consists of a set of features.

WHAT IS THE STRUCTURE OF THE QUESTION?
A feature and a flag.

WHAT IS THE STRUCTURE OF THE FEATURES IN THE DESCRIPTOR?
Each feature consists of a property and a value. A property is a name which is a character string. A value is either a number or a string of characters.

WHAT IS THE STRUCTURE OF THE FLAG?
The flag will be either "descriptors" or "number".

WHAT IS THE INITIAL VALUE OF THE DATA-BASE? **Unnecessary.**

DESCRIBE THE ANSWER.
If the flag is "descriptors", then the program should print each descriptor that contains the feature.

ARE WE FINISHED?
If the flag is "number", then it should just print the total number of such descriptors.

ARE WE FINISHED? **Yes.**

```
NEWS
  PRINT("Ready for the QUESTION")
  R1 ← READ()
  R6 ← The set of all R2 in R3 such that:
       MEMBER(R5, R2)
  If EQUAL(R6, "descriptors")
    Then For all R2 in R6 do:
           PRINT(R2)
  If EQUAL(R6, "number")
    Then PRINT(R7)
```

R6 is the DATA of the FLAG field of R1. R5 is the DATA of the FEATURE field of R1.

R7 is a number which is the size of R6.

R3 is a list whose generic element is a set whose generic element is a record whose fields are PROPERTY: a string and VALUE: either a string or a number.

R1 is a record whose fields are FEATURE: a record whose fields are PROPERTY: a string and VALUE: either a string or a number and FLAG: either a string whose value is "number" or a string whose value is "descriptors".

-DIALOGUE 11-

WHAT IS THE NAME OF THE PROGRAM YOU WISH TO WRITE?

Write me a news retrieval program

WHAT IS THE NAME OF THE PROGRAM YOU WISH TO WRITE? News

DESCRIBE NEWS.

It has a data base of news stories. Each story has a set of key words associated with it. I'll request a story by giving a key word. The response should be all the stories with that key word. If the command is "number" followed by a key word, then the number of stories having that key word should be printed.

WHAT IS THE STRUCTURE OF THE STORYS IN THE DATA-BASE?

they are strings.

WHAT IS THE STRUCTURE OF THE KEY-WORDS IN THE SET?

primitive.

WHAT IS THE INITIAL VALUE OF THE DATA-BASE?

unnecessary.

WHAT IS THE INITIAL VALUE OF THE KEY WORD SET?

unnecessary.

ARE WE FINISHED? yes

```
NEWS
  PRINT("Ready for the KEY-WORD")
  R1 ← READ()
  Cases:  If EQUAL(R1 R11)
            Then R1# ← The set of all R8 in R3 such that:
                    MEMBER(R11 INSPIRE R8))
                 PRINT(R1#)
     else If EQUAL(R1 R6)
            Then R4 ← The set of all R2 in R3 such that:
                    MEMBER(R5 INSPIRE R2))
                 PRINT(R7)
```

R11 is a primitive.

R6 is a record whose fields are STRING: a string whose value is "number" and KEY-WORD: a primitive.

R7 is a number which is the size of R4.

R8 is a mapping from the elements of R3 to a set whose generic element is a primitive.

R5 is the DATA of the KEY-WORD field of R8.

R3 is a set whose generic element is a string. R1 is either R11 or R6.