13 Conceptual Analysis of Natural Language

Lawrence Birnbaum
Mallory Selfridge

INTRODUCTION

One of the crucial issues to be faced by any theory of natural language is the process of mapping natural language text into the memory structures that represent the meaning of that text. We call this process Conceptual Analysis. This chapter has two complementary goals. First, we hope to provide a clear introduction to conceptual analysis. Second, we will present the results of our own research concerning some of the unresolved problems in language analysis—of which there are many.

Research on conceptual analysis has been proceeding for several years now, starting with the work reported in Schank et al. (1970) and Riesbeck (1975). We have built on that work, concentrating on some important questions that we feel have not been sufficiently investigated:

1. What is the role of syntax in a semantics-oriented language analysis process?
2. How can we achieve a flexible balance of top-down and bottom-up processing?
3. How does short-term memory structure affect the analysis process?
4. What is the right vocabulary (formalism) for expressing the processing knowledge that the system has?
5. What are the criteria on the performance of a word-sense disambiguation process which we should strive to achieve?

Our work has provided some answers to the above questions, which have in turn enabled us to construct more flexible algorithms, capable of analyzing the kinds of complex noun groups and multiple-clause inputs found in most text.

Background

The goal of the conceptual analysis process is to map natural language input into a representation of its meaning. This goal differs from that assumed by most models of parsing, which hold that the purpose of the parsing mechanism is to perform a *syntactic* analysis of the input sentences. In other words, the goal of a conceptual analyzer is to derive the semantic and memory structures underlying an utterance, whereas the goal of a syntactic parser is to discover its (syntactic) structural description. Examples of syntactic parsers include: Thorne et al. (1968), Bobrow and Fraser (1969), Woods (1970), Winograd (1972), Kaplan (1975), and Marcus (1975, 1979). Other examples of semantics-oriented analyzers are those described by Wilks (1973, 1975a, and 1976), and by Burton (1976).

Early research in natural language processing assumed that a complete understanding system would require a syntactic analyzer as a front-end. We, on the other hand, do not believe that such prior syntactic analysis is necessary. This, of course, does *not* mean that we deny the existence of syntactic phenomena or of syntactic knowledge; rather we are arguing against the notion of a separate syntactic analysis phase.

Since the goal of conceptual analysis differs from the goal of syntactic parsing, it is understandable that the mechanisms which are employed differ as well. In particular, it is our goal that the conceptual analysis process use any and all available knowledge whenever helpful. Such knowledge may include syntactic knowledge, but is certainly not limited to that. It is a claim of our theory that non-linguistic knowledge is often crucial even at the earliest points in natural language processing.

Expectations

A language understander must connect concepts which are obtained from word meanings, and from inferences derived from word meanings, into a coherent representation of the input as a whole. Because of the possibilities of word-sense ambiguity or irrelevant inference, an understander must also be able to choose among alternative concepts. So, conceptual analysis consists mainly of connecting and disambiguating (i.e., choosing among) representational structures.

The processing knowledge which the system uses for this task is in the form of *expectations* (Schank et al., 1970; Riesbeck, 1975; and Riesbeck and Schank, 1976). When a person hears or reads words with certain meanings, he expects or predicts that words with certain other meanings may follow, or may have already been seen. People constantly predict what they are likely to see next, based on what they have read and understood so far and on what they know about language and the world. They use these expectations to disambiguate and connect incoming text.

Of course, expectations are used in syntactic analysis programs as well. The difference lies in the *origin* of the expectations. In syntactic analyzers, the

expectations are derived from a grammar. In conceptual analyzers, the expectations are governed instead by the incomplete CD structures representing the meaning of the input.

To illustrate the use of expectations in understanding, suppose that the following simple sentence were the input to an expectation-based conceptual analysis system: "Fred ate an apple." Reading from left to right, the system first finds the word "Fred". The system understands this as a reference to some male human being named Fred, and stores the reference, represented as the token FRED, in some kind of short-term memory. The next word is "ate". This is understood as an instance of the concept of eating, which in Conceptual Dependency is represented by a CD frame something like this:

(INGEST ACTOR (NIL) OBJECT (NIL)).

Also, the meaning of "ate" supplies some expectations which give hints as to how to go about filling the empty slots of this frame. One of these expectations suggests that the ACTOR of the INGEST, an animate being, may have already been mentioned. So the analyzer checks short-term memory, finds FRED there, and fills the ACTOR slot of the INGEST:

(INGEST ACTOR (FRED) OBJECT (NIL)).

There remains an unfulfilled expectation, which suggests that some mention will be made of what it is that Fred ate, that it should be some edible thing (or at least a physical object), and that it should fill the OBJECT slot. Next, the word "an" is read. This creates an expectation that some instance of a concept should follow, with the instruction that if one is found, it should be marked as an indefinite reference. (This is information which can aid memory.) Finally, "apple" is read. It is understood as an instance of the concept APPLE, representing something which is known to be food. The expectation created when "an" was read is satisfied, so APPLE is marked as an object not previously seen, which we will represent as: (APPLE REF (INDEF)). The second expectation created when "ate" was read is also satisfied, so the OBJECT slot of the INGEST is filled by (APPLE REF (INDEF)). The system's current understanding of the input is represented as

(INGEST ACTOR (FRED) OBJECT (APPLE REF (INDEF))).

There are no more words to read, so the process halts.

The above example gives an idea of the conceptual analysis process. When a word is read, the conceptual structure representing the meaning of that word is added to short-term memory. In addition, expectations are created, which provide the processing knowledge necessary to connect up these conceptual structures into a representation of the input as a whole.

Evolution of Conceptual Analysis

The working hypothesis underlying conceptual analysis is that meaning is a crucial factor at the earliest stages in the language understanding process. Two observations led to this position originally. The first was the failure of syntax-oriented approaches to facilitate the construction of natural language processing systems, particularly in the early period of machine translation research. This failure seemed to indicate that a more meaning-based approach to language analysis was necessary.

The second observation was the rather common-sense one that it is easier to read a foreign language with which one has a passing acquaintance, than it is to speak or write it, especially if one knows something about the topic being discussed. This seemed to indicate that language analysis required less knowledge of syntax than generation, so that a meaning-based approach to language analysis could be successful.

The subsequent development of Conceptual Dependency led to the recognition, in Schank et al. (1970), that the incomplete CD structures, used to represent the meaning of the input, generated expectations that could be used in language analysis. Riesbeck (1975) demonstrated the feasibility of such an approach in the MARGIE parser. He implemented the expectations by attaching small executable programs, called *requests,* to the words in the system's vocabulary. These requests were then used to connect the semantic representations of the words into a representation of the whole sentence.

The next parser written was Riesbeck's ELI (Riesbeck and Schank, 1976). ELI was an attempt to show how expectations could be used both to disambiguate words and to connect structures, by making the analyzer extremely top-down. In ELI, a conceptual structure was only accepted for processing if it satisfied a prior expectation. ELI's capabilities were expanded by the addition of Gershman's Noun Group Processor (NGP) program to handle complex noun groups and relative subclauses (Gershman, 1977 and 1979). The motivation for the work described here was the desire to make homogeneous the kinds of processing performed by ELI and by NGP.

Input/Output Examples

The following input/output examples are representative of our current conceptual analyzer's abilities. The program is called CA (Birnbaum and Selfridge, 1979):

```
*(CA '(MARY TOOK BILL AN APPLE.))

FINAL ANALYSIS: STM = (CON20)

CON20 =
(ATRANS ACTOR
        (PP PPCLASS (HUMAN) NAME (MARY) GENDER (FEMALE))
        OBJECT
```

```
(PP PPCLASS (FOOD) TYPE (APPLE) REF (INDEF))
FROM
(PP PPCLASS (HUMAN) NAME (MARY) GENDER (FEMALE))
TO
(PP PPCLASS (HUMAN) NAME (BILL) GENDER (MALE))
INST
(PTRANS ACTOR
        (PP PPCLASS (HUMAN) NAME (MARY) GENDER (FEMALE))
        OBJECT
        (PP PPCLASS (FOOD) TYPE (APPLE) REF (INDEF))
        FROM
        (NIL)
        TO
        (PP PPCLASS (HUMAN) NAME (BILL) GENDER (MALE))
        TIME
        (PAST))
TIME
(PAST))
```

The above CD representation means roughly the following: A female human named Mary ATRANSed a food, type apple, from herself to a male human named Bill. This was done by means of Mary's PTRANSing the apple to Bill. The above example demonstrates the ability of the analyzer to deal with certain kinds of ambiguity. In this particular case, the problem arises because ''take'' is a highly ambiguous word. For example, the system must also handle inputs like ''Mary took the train to Boston'', where the top-level conceptualization is a PTRANS, and ''John took some aspirin'', where the top-level conceptualization is an INGEST.

```
*(CA '(WHERE DID VANCE GO LAST WEEK?))

FINAL ANALYSIS: STM = (CON11)

CON11 =
(PTRANS ACTOR
        (PP PPCLASS (HUMAN) LASTNAME (VANCE) FIRSTNAME (NIL))
        OBJECT
        (PP PPCLASS (HUMAN LASTNAME (VANCE) FIRSTNAME (NIL))
        TO
        (LOC SPEC (*?*))
        FROM
        (NIL)
        TIME
        (TIME TYPE (WEEK) NUMBER (-1.))
        MODE
        (NIL))
```

The above is simply a typical example of the sorts of questions which the program is capable of analyzing. The CD representation means: A human with

the last name Vance PTRANSed himself to some unknown place (the ''*?*'' indicates that this is the information the question is looking for) the week before the present one.

```
*(CA '(A SMALL PLANE STUFFED WITH 1500 POUNDS OF MARIJUANA CRASHED
       10 MILES SOUTH OF HERE YESTERDAY AS IT APPROACHED A LANDING
       STRIP KILLING THE PILOT.))

FINAL ANALYSIS: STM = (CON48 CON63)

CON48 =
(SAME-TIME-AS
      CON-A
      (PROPEL ACTOR
              (PP CLASS (VEHICLE)
                  TYPE (AIRPLANE)
                  SIZE (SMALL)
                  REF (INDEF)
                  REL (PTRANS ACTOR
                              (NIL)
                              OBJECT
                              (PP CLASS
                                  (PHYSOBJ)
                                  TYPE
                                  (MARIJUANA)
                                  AMOUNT
                                  (UNIT TYPE
                                        (LB)
                                        NUMBER
                                        (NUM VAL (1500))))
                              TO
                              (INSIDE PART (PREVIOUS))
                              FROM
                              (NIL)
                              TIME
                              (PAST)))
              OBJECT
              (PP CLASS (PHYSOBJ) TYPE (GROUND))
              PLACE
              (LOC PROX (HERE)
                   DIR (SOUTH)
                   DIST (UNIT TYPE
                              (MILE)
                              NUMBER
                              (NUM VAL (10))))
              TIME
              (TIME TYPE (DAY) NUMBER (-1.)))
      CON-B
      ($APPROACH VEH
                 (PP PPCLASS (NIL) REF (DEF))
                 FIELD
                 (LOC TYPE (LANDING-FIELD) REF (INDEF))))
```

```
CON63 =
(LEAD-TO ANTE
         (NIL)
         CONSE
         (HEALTH ACTOR
                 (PP PPCLASS (HUMAN) TYPE (PILOT) REF (DEF))
                 VAL
                 (-10.)))
```

This example demonstrates the analyzer's ability to parse long inputs containing several clauses. The analysis consists of two CDs. The first, CON48, which is split into CON-A and CON-B, means roughly the following: CON-A says that a vehicle, type airplane, size small, into which someone had PTRANSed 1500 pounds of marijuana at some time in the past, hit (PROPELled) the ground at a location 10 miles south of the default variable HERE. CON-B says that at the same time some vehicle was engaged in the $APPROACH scene of the $AIRPLANE script. The second CD, CON63, means that something caused the death of a human playing the role of pilot in the $AIRPLANE script. The two CDs have not been connected, and references such as "here" in the location have not been specified, because the necessary information does not exist at the lexical level. Higher level processes, such as a script applier (Cullingford, 1978), discussed in Chapter 5, are needed to complete the representation.

The Issue of Flexibility

One of the major concerns in this work has been to increase the *flexibility* of the analysis process. In order to handle noun groups and the kinds of long, multiple-clause inputs found in most text, an analyzer must be capable of operating in both a top-down and bottom-up mode with equal facility. Within the framework of conceptual analysis, this means that the analyzer must be able to use expectations when they are available. But if no helpful expectations are available, the system must also be able to accept and save intermediate concepts and processing knowledge until such time as they can be used.

This point has two immediate corollaries for language processors. The first is that in order to save intermediate conceptual structures and processing knowledge when operating in a bottom-up fashion, an analyzer must have a sufficiently flexible short-term memory in which to hold them. In particular, noun groups and multiple-clause inputs often present cases in which several constituent conceptual structures must be saved until enough information is gathered to tie them together. Our analyzer uses a simple ordered list, and this has proven sufficient.

The second corollary has to do with the problem of word-sense disambiguation. A language processor cannot always decide among competing word senses (concepts and expectations) immediately upon reading an ambiguous word, because the necessary information may simply not be available yet. For example, consider the following pair of sentences:

John broke the pot.
John smoked the pot.

In the first sentence, "broke" creates an expectation for something which can be broken. This expectation can be used immediately upon reading the word "pot" to prefer the "container" sense and reject the "marijuana" sense. In the second sentence, "smoked" creates an expectation for something which can be burned and inhaled, and this expectation can be used immediately upon reading the word "pot" to choose the "marijuana" sense.

But now consider the passive forms of these sentences:

The pot was broken by John.
The pot was smoked by John.

Unlike the corresponding active cases, the word "pot" cannot be immediately disambiguated in the passive sentences because the semantic information necessary for choosing among the possible word senses does not appear until later in the input.

Some Conclusions

Probably the major theoretical hypothesis of conceptual analysis is the claim that a separate syntactic analysis phase is unnecessary in language understanding. Previous work has demonstrated the feasibility of this approach. We believe that our success in producing a significantly improved analysis process provides further evidence for this hypothesis.

The important issue for future work in language analysis is to gain a much better understanding of how higher level memory structures can be used in the analysis process (Schank et al., 1978). The importance of this question is apparent, we believe, particularly when addressing the problem of word-sense ambiguity. Very often, when a parser thinks that a word is ambiguous, it does so because it hasn't made use of, or possessed, the knowledge that should have been provided by higher memory structures. That is, it didn't really understand well enough what it had been reading. Of course, how well we can apply memory in the parsing process will depend on how well we understand memory. That remains the most difficult, but most crucial, question.

OVERVIEW OF THE ANALYZER

What mechanism can be used to implement expectations? Basing our work on Riesbeck (1975), we chose to implement expectations by using test-action pairs known as *requests*. If the test of a request is checked and found to be true, then

the corresponding actions of the request are executed. Requests are thus a form of production (Newell, 1973).

Consider again the example used in the above section on expectations: "Fred ate an apple." We said that an expectation for an edible thing would be generated to fill the OBJECT slot of the INGEST structure built by "eat". Implemented as a request, this expectation would look something like this:

TEST: Can a concept representing an edible object be found?
ACTION: Put it in the OBJECT slot of the INGEST concept.

This request is *activated* when the word "ate" is read or heard. In the example, its test became true when the meaning of the word "apple" was understood. Its action specified that APPLE was to be taken as the OBJECT of the INGEST. (Usually, we will describe requests even more informally, as something like "If you can find an edible thing, then put it in the OBJECT slot.")

Using this approach, the conceptual analysis process can be most easily described as a special type of production system. Several questions then arise:

1. What kind of control structure is needed?
2. What kinds of tests and actions can requests perform?
3. Where are requests stored, and how are they accessed?

We will now give some preliminary answers to each of these questions in turn, by outlining a kind of "minimal" analyzer.

Control Structure and Data Structures

First of all, a conceptual analyzer needs some kind of working short-term memory (STM), a place to hold the conceptual structures being processed. Without a flexible STM, a conceptual analyzer would be unable to remember substructures which are to be integrated into the final representation of the input. Let us assume that this short-term memory is merely a list, called the *CONCEPT-LIST* or *C-LIST*. Whenever the processing knowledge of the system, in the form of requests, believes that some conceptual structure is relevant to representing the meaning of the input, that structure is added to the end of this list. This kind of flexibility in holding structures is essential if the parser is to function effectively in a bottom-up mode.

Second, a conceptual analyzer needs a data structure to hold the active requests. Without one, of course, the system would never be able to maintain any expectations. In our minimal analyzer, this is also a list, called the *REQUEST-LIST* or *R-LIST*. The system examines the requests on the R-LIST, and if the test of a request is true, its actions are executed. This process is called *request consideration*.

The control structure of a minimal conceptual analysis algorithm is straightforward:

1. Get the next lexical item (word or idiomatic phrase) from the input, reading from left to right. If none, the process terminates.
2. Load (activate) the requests associated with the new item into the R-LIST.
3. Consider the active requests in the R-LIST.
4. Loop back to step 1.

Requests

It is very important that the right vocabulary be devised to express the processing knowledge the system has. Given that we are using requests to embody that processing knowledge, this problem becomes one of choosing the right set of tests and actions that requests may perform.

There are two general classes of tests which requests may perform:

1. Test for the occurrence of a particular word or phrase.
2. Test for certain semantic or ordering properties of the conceptual structures on the C-LIST.

Requests with the first kind of test are called *lexical* requests, while those with the second kind are called *conceptual* requests (or usually, just requests).

A request may perform any of the following actions should its test become true:

1. Add a conceptual structure to the C-LIST.
2. Fill a slot in a conceptual structure with some other structure.
3. Activate other requests, i.e., add them to the R-LIST.
4. De-activate requests, including itself.

There are several variants of these basic actions which will be analyzed when they are discussed in turn in the section giving a detailed view of the program.

In practice, it seems convenient for requests to be more like the LISP "conditional" than traditional productions. That is, a request should consist of an ordered set of tests which control branching to mutually exclusive actions. This is particularly useful in allowing a request to notice when it is no longer appropriate and then to remove itself from active status. We can accomplish this by making one of the tests in the request look for clues that the request is no longer appropriate. Should that test become true, the associated action would simply have the request de-activate itself.

Requests are often simply organized under the particular words for which they are useful. That is, requests stored under words form a kind of dictionary. For

example, the request created when reading "an" would simply be stored under the word "an", and activated for use whenever that word is read. This is one way of organizing requests so that only those relevant to the current situation are active in the system. Clearly, not all analysis expectations should be implemented as lexically indexed requests. For a discussion of "situationally indexed" expectations, see Gershman (1979).

An Example

Let us now examine in detail how the minimal conceptual analyzer we have described might work on another simple example, the statement "Fred gave Sally a book." (The step numbers refer back to the steps in the loop of the control structure sketched above.)

READ THE NEXT WORD (Step 1): In this case, the first word is "Fred".

ACTIVATE REQUESTS FROM THE DICTIONARY ENTRY (Step 2): There is only one in the entry for "Fred":

```
REQUEST—
    TEST:  T
    ACTIONS:  Add the structure (PP CLASS (HUMAN) NAME (FRED))
              to the C-LIST
```

The symbol "T" in the test of this request is always true, so the action should always be performed. When activating this request, the system assigns it the name REQ0.

CONSIDER THE ACTIVE REQUESTS (Step 3): There is only one, REQ0, and it has a true test, so its action is performed. Now, C-LIST = CON1, where CON1 is the internal name generated for:

(PP CLASS (HUMAN) NAME (FRED)).

READ THE NEXT WORD (Step 1): "gave".

ACTIVATE REQUESTS FROM THE DICTIONARY ENTRY (Step 2): For the purpose of this example, there is just one request in the entry for "gave":

```
REQUEST —
    TEST:  T
    ACTIONS:  Add the structure
              (ATRANS ACTOR (NIL) OBJECT (NIL)
                      TO (NIL) FROM (NIL)
                      TIME (PAST))
              to the C-LIST
```



```
Activate the request
REQUEST—
    TEST:  Can you find a human on the C-LIST
           preceding the ATRANS structure?
    ACTIONS:  Put it in the ACTOR and FROM slots
              of the ATRANS

Activate the request
REQUEST—
    TEST:  Can you find a human on the C-LIST
           following the ATRANS structure?
    ACTIONS:  Put it in the TO slot of the ATRANS

Activate the request
REQUEST—
    TEST:  Can you find a physical object on the
           C-LIST following the ATRANS structure?
    ACTIONS:  Put it in the OBJECT slot of the ATRANS
```

When activating this request, the system assigns it the name REQ1.

CONSIDER THE ACTIVE REQUESTS (Step 3): There is only one, REQ1, and its test is true, so its actions are performed. First, it adds a new structure to the end of the C-LIST. So now C-LIST = (CON1 CON2), where CON2 is the internal name generated for the ATRANS structure which represents the meaning of "gave". Then, three new requests are activated, which strive to fill the gaps in the ATRANS structure. The first of these requests strives to fill the ACTOR slot of the ATRANS. When it is activated, the system assigns it the name REQ2. The next request strives to fill the TO slot of the ATRANS. When activated, it is assigned the name REQ3. The last new request, REQ4, strives to fill the OBJECT slot of the ATRANS.

The system now considers these new requests, and REQ2 is found to have a true test. The action of REQ2 is executed, and so CON1, representing "Fred", is palced in the ACTOR slot of CON2 (the ATRANS) and removed from the C-LIST. The other two requests, REQ3 and REQ4, do not have true tests, so their actions are not executed. However, they do remain as active expectations on the R-LIST.

READ THE NEXT WORD (Step 1): "Sally".

ACTIVATE REQUESTS FROM THE DICTIONARY ENTRY (Step 2): There is a request in the entry for "Sally", which is almost exactly like the one for "Fred":

```
REQUEST—
    TEST:  T
    ACTIONS:  Add the structure (PP CLASS (HUMAN) NAME (SALLY))
              to the C-LIST
```

When activating this request, the system assigns it the name REQ5.

CONSIDER THE ACTIVE REQUESTS (Step 3): REQ5 is found to have a true test, so its action is performed. Now C-LIST = (CON2 CON3), where CON3 is the structure representing "Sally". REQ3, which is still active, now also has a true test, and so CON3 is used to fill the TO slot (recipient) of CON2, the ATRANS structure. REQ4 still does not have a true test.

READ THE NEXT WORD (Step 1): "a".

ACTIVATE REQUESTS FROM THE DICTIONARY ENTRY (Step 2): There is only one request listed under "a":

REQUEST—
TEST: Has a new structure been added to the end of the C-LIST?
ACTIONS: Mark it as an indefinite reference

When activating this request, the system assigns it the name REQ6.

CONSIDER THE ACTIVE REQUESTS (Step 3): Neither REQ4 nor REQ6, the only active requests on the R-LIST, have true tests. So, nothing happens.

READ THE NEXT WORD (Step 1): "book".

ACTIVATE REQUESTS FROM THE DICTIONARY ENTRY (Step 2): There is only one request listed under "book":

REQUEST—
TEST: T
ACTIONS: Add the structure
(PP CLASS (PHYSICAL-OBJECT) TYPE (BOOK))
to the C-LIST

When activating this request, the system assigns it the name REQ7.

CONSIDER THE ACTIVE REQUESTS (Step 3): REQ7 has a true test, so its action is performed: CON4 is added to the C-LIST, where CON4 is the structure which represents "book". REQ6 now has a true test, so CON4 is marked as an indefinite reference, like this:

(PP CLASS (PHYSICAL-OBJECT) TYPE (BOOK) REF (INDEF)).

REQ4 also has a true test, so CON4 is used to fill the object slot of CON2, the ATRANS structure.

There are no more words in the input. The analysis halts, with the following final result representing the input:

(ATRANS ACTOR (PP CLASS (HUMAN) NAME (FRED))
OBJECT (PP CLASS (PHYSICAL-OBJECT)
TYPE (BOOK) REF (INDEF))
TO (PP CLASS (HUMAN) NAME (SALLY))
FROM (PP CLASS (HUMAN) NAME (FRED))
TIME (PAST))

THEORETICAL ISSUES

The Role of Syntax

Traditional notions of syntax include ideas like "part of speech" and "phrase marker" in discussing the structure of a sentence. What we would like to claim in this section is that these notions of syntax are inappropriate when attempting to describe and utilize syntactic knowledge in a language understanding process.

To begin with, what is the purpose of syntactic knowledge? Clearly, a major use of syntactic knowledge is to direct the combination of word meanings into utterance meaning when semantic information is not sufficient or is misleading. For example, in an utterance like "Put the magazine on the plate," it is syntactic knowledge that tells the understander which object is to be placed on top of which. From the point of view of its use in a conceptual analyzer, therefore:

> A large part of syntax is knowledge of how to combine word meanings based on their positions in the utterance.

How can this syntactic knowledge be characterized? We seek a specification which takes into account the fact that the point of syntax is its *use* in the understanding process. We have viewed the process of understanding as one of connecting representational structures, where a connection has been established between structures when one fills a slot in the other, or both fill a larger form. Thus syntactic knowledge is knowledge which specifies where in the utterance some word is to be found whose meaning can be connected (via slot-filling) with the meaning of another word. Of course, we must now specify the notion "position in an utterance".

Given that processing knowledge is encoded in requests, this problem reduces to the question of how positional information can be taken into account in the tests of requests. This may be done by having tests that check for:

1. Relative positions of input elements in short-term memory (the C-LIST).
2. The proper order for filling slots in structures.

These methods describe position in an utterance by use of relative positional information. Both are essentially ways of utilizing word order information. The first of these methods, using relative position in short-term memory, describes the position of a conceptual structure in direct relation to the other structures it might be connected with via slot-filling. For example, consider the following request from the example above:

REQUEST—
TEST: Can you find a human on the C-LIST preceding the ATRANS structure?
ACTIONS: Put it in the ACTOR and FROM slots of the ATRANS

The test of this request checks whether or not there is a conceptual object of a certain type (human) which has a certain position on the C-LIST relative to the ATRANS structure (precedes).

The second method, ordering of the slots to be filled, relates the position of a structure to other structures somewhat more indirectly. In particular, by constraining the order in which slots should be filled, we are relating the fillers of those slots *temporally*. For instance, a constraint that the ACTOR slot of a conceptualization be filled before the OBJECT slot reflects the fact that the ACTOR of that conceptualization should be seen before the OBJECT in an active construction.

Another important method relates the position of a conceptual structure to the position of a particular lexical item (i.e., a function word), rather than to another concept. Function words, including prepositions, post-positions, and affixes, are very important syntactic cues in conceptual language analysis.

The achievement of a flexible balance of top-down and bottom-up processing in our analyzer led to the discovery of an interesting side-effect, with implications for our notions of syntax. We found that it was not necessary for the analyzer to have any prior expectations at the start of a sentence, such as for an initial noun-group, or for a complete sentence. Most other parsers, whether syntax-based or semantics-based, do use such prior expectations at the start of a sentence.

However, if there is no prior expectation for a complete sentence, how can we explain the oddity of sentence (a) below, as opposed to sentence (b)?

(a) A plane stuffed with 1500 pounds of marijuana.
(b) A plane was stuffed with 1500 pounds of marijuana.

There is a real feeling of "incompleteness" in sentence (a) which would seem to argue that people *do* have prior expectations for complete sentences. However, consider how these two sentences would sound following this question:

(c) What crashed?

Following question (c), it is sentence (b) that sounds odd, while sentence (a) seems perfectly appropriate. Now, one could explain this phenomenon by saying that the expectations to be supplied at the beginning of a sentence can be varied, and that after question (c), a parser is no longer requiring a complete sentence. But of course, question (c) could equally well be followed by any of the following:

(d) A plane carrying 1500 pounds of marijuana crashed.
(e) Nothing crashed.
(f) What are you talking about?
(g) There wasn't any crash.
(h) I didn't hear anything.

So obviously, a parser which depends on prior expectations cannot simply throw them out in the context of a question like (c). If it did, it would then be unable to handle inputs like (d) through (h). Further, these examples make clear that the presence or absence of prior syntactic expectations does not explain why (b) is odd in the context following (c). That can only be explained by reference to higher level processes, such as memory search and question answering. However, if we have such processes, then we can also explain why (a) is odd in a null context. Therefore, this phenomenon cannot be used as evidence against our contention that prior syntactic expectations, such as those for a complete sentence, are unnecessary.

We would claim, moreover, that the explanation for the symmetrical cases of (a) in a null context and (b) in the context of (c), ought to be the same. In other words, the explanation underlying this particular type of "gramaticality" judgment does not lie in a syntactic phenomenon at all. Rather, it is a phenomenon which should be explained by reference to such higher level processes as memory search and question answering.

Word-Sense Disambiguation

In conceptual analysis, word-sense ambiguity manifests itself when one word has two or more requests adding structures to the C-LIST. How can we insure that the structure representing the correct meaning "in context" can be chosen? What we describe in this section, of course, does not constitute anything like a complete answer to the problem of word-sense ambiguity. However, we do perhaps clarify what the possibilities are. Below we present a disambiguation algorithm based on the considerations presented here.

There are essentially two ways to handle word-sense disambiguation. Either the structures which represent alternate hypotheses check the context in order to decide whether they are appropriate, or the context checks the structures in some way, or both of these methods are used. In conceptual analysis terms, this means that there are two ways to accomplish the task of request selection in order to

handle ambiguity. The first method is to have the tests of the requests check for clues as to whether or not the structure they would add to the C-LIST is appropriate. These tests could be at either a lexical level (i.e., checking for specific words or phrases), or at a conceptual level (i.e., checking other structures on the C-LIST). For example, to handle the difference between "John left the restaurant" (PTRANS) and "John left a tip" (ATRANS), the requests associated with the word "left" could check to see whether a location or an object is added to the C-LIST.

The second way to perform request selection is to have a function using the information in the context select the appropriate requests, based on the structures they build. The traditional method for doing this is to choose the structure which can be best connected with other structures the system has in its short-term memory. An early incarnation of this idea can be seen in the use of "selection restrictions" to rule out certain combinations of meanings (Katz and Fodor, 1963). In conceptual analysis, the most obvious application of this idea involves choosing that structure which can be used to fill gaps in other structures on the C-LIST, or that can use other structures on the C-LIST to fill its own gaps, or both. This is clearly not the only kind of "connectedness" or "coherence" which can cause us to favor one sense over another. In particular, the structures of the context which are used to check some potential structures need not be directly derived from the input, that is, should also include structures derived by inferential memory procedures.

Top-down selection of this sort would disambiguate "left" by choosing the ATRANS sense of "left" if "a tip" appeared, because ATRANS can use the structure associated with "tip" to fill its OBJECT slot. Similarly, it would choose the PTRANS sense of "left" if "the restaurant" appeared, because PTRANS can use the structure associated with "restaurant" to fill its FROM slot.

Neither of these two methods is new, of course. The MARGIE parser (Riesbeck, 1975) and ELI (Riesbeck and Schank, 1976) employed the first method, i.e., having requests test the context to determine their applicability, with some success (see also Small, 1978). Wilks' preference semantics scheme (Wilks, 1975a and 1976) includes probably the most highly developed mechanism for searching for connections (see also Hayes, 1977). ELI also employed a restricted form of this method.

Intelligent Error Correction

Recently, some attention has been focused on the idea of deterministic "wait and see" syntactic parsing, without backup (Marcus, 1975). Determinism is guaranteed by restricting the analysis process so that no structure which is built can later be discarded. This notion has been contrasted with the non-deterministic "guess and then backup" method typified by augmented transition network (ATN) parsers. While we are in complete agreement with the idea that it is undesirable

to use blind backup as an integral part of the analysis process, we believe that Marcus' notions of determinism have ignored the possibilities of what might be called "intelligent error correction". For example, consider the following pair of sentences:

John gave Mary a book.
John gave Mary a kiss.

In the first sentence, "gave" should add the following structure to the C-LIST:

(ATRANS ACTOR (NIL) OBJECT (NIL) TO (POSSESSION-OF (NIL))
FROM (POSSESSION-OF (NIL)).

In the second sentence, "gave" is merely a dummy verb, and this structure is completely inappropriate. However, it turns out that there are two ways to handle this problem in a conceptual analyzer. The first approach in conceptual analysis is analogous to what Marcus calls "wait and see" syntactic analysis: don't build any top-level structure until either "book" or "kiss" is read. The second approach is "intelligent error correction". Using this method, the ATRANS structure is built in both cases, and the analyzer proceeds to fill the empty gaps with appropriate substructures. However, an additional request is activated which essentially says, "If you see another complete concept, i.e., action or state, rather than something that can fill the object slot of an ATRANS, then that is the actual top-level structure. Remove the ATRANS from the C-LIST, and fill the empty gaps in the new structure with the substructures which can be found in the ATRANS." Using either of these methods, the sub-structures which have been assembled along the way, such as the representations for "John" and for "Mary", are still available, and this after all is the key to avoiding back-up. Intelligent error correction would be the better approach in those cases where the initial hypothesis is usually correct (Gershman, 1979).

A DETAILED VIEW OF THE PROGRAM

This section provides further details about requests. We will describe the kinds of tests and actions that we found to be necessary to accomplish conceptual analysis, and discuss why they are necessary and the situations in which they are useful. We also provide further details about the control structure of a conceptual analyzer, and describe the motivations behind this structure.

Requests

Let us reiterate the actions which a request may perform:

1. Add a conceptual structure to the C-LIST.

2. Fill a gap in a conceptual structure with some other structure.
3. Activate other requests.
4. De-activate requests, including itself.

There is no restriction that a request perform only one of these actions. However, the convention that a request may add only one structure to the C-LIST is important for the purpose of designing word-sense disambiguation algorithms. Structures are always added to the end of the C-LIST.

Gap-Filling Requests

Conceptual structures are connected when one fills a gap in the other, or both fill a gap in some larger structure. Requests which fill gaps remove embedded conceptual objects from the C-LIST.

We characterize gap-filling requests in roughly two ways. Either the request knows which gap it's trying to fill and is looking for a filler, or it knows the filler and is looking for the proper gap. Requests of the first type are looking for some structure in order to embed it in another. We have already seen numerous examples of this type of gap-filling request in the previous examples. These requests may arise in two ways:

1. They may be activated by the request which built the conceptual structure containing the gap they strive to fill.
2. They may be brought in by function words (e.g., "to").

Gap-filling requests arising from function words are particularly important in that if a request activated by a function word seeks to fill some gap with a structure, it must be allowed to do so. If the gap is already filled by some other structure, the structure should be returned to the C-LIST. For example, consider the sentence "John gave Mary to the Sheik of X" (Wilks, 1975b). After reading "John gave Mary . . ." our conceptual analyzer would assume Mary to be the recipient of the giving action, and would build:

(ATRANS ACTOR (JOHN) OBJECT (NIL) TO (MARY) FROM (JOHN)).

However, "to the Sheik" clearly marks the Sheik as the recipient, and that explicit marking has high priority. Hence, the analyzer changes its representation to

(ATRANS ACTOR (JOHN) OBJECT (NIL) TO (SHEIK) FROM (JOHN)).

The concept MARY is returned to the C-LIST, and is then picked up as the OBJECT of the ATRANS.

The second type of gap-filling request has a filler and looks for some structure in which to embed it. These usually arise from words that appear in noun groups, particularly adjectives. For example, the word "red" has an associated request which looks for the representation of a physical object, in order to fill the COLOR slot of that object with the structure RED. This kind of gap-filling request also arises from words describing time and place settings. For example, in the sentence "Yesterday, Fred bought a car," the word "yesterday" has an associated request which looks for a concept, in order to fill the concept's TIME slot.

Once a gap in some structure is filled, all other requests which seek to fill it should be removed. That can be handled in two ways. The simplest is to have requests which seek to fill some particular gap test whether it has already been filled, and if so then remove themselves. This is how our current implementation operates. The other possibility is to use two-way pointers between gaps in conceptual structures and the requests which strive to fill them, and to place in the control structure a procedure which automatically removes unnecessary requests. Riesbeck's ELI uses this approach.

Activating Other Requests

Two major purposes motivate our use of requests to activate other requests. First, requests that add some conceptual structure to short-term memory will often activate requests which strive to fill gaps in that structure.

Second, requests can change the sense of some word in some local context (Riesbeck and Schank, 1976) by activating requests that test for specific input words, and adding requests to the dictionary entries of those words. Function words are an important instance of this notion of locally changing word sense, since in many cases they function only with respect to a particular construction.

Take the example of the word "of". The relationship it denotes depends on the construction in which it appears. One such construction includes examples such as "five yards of cloth" or "ten pounds of sugar". A good way to handle this construction is to have a request which is activated by the "unit" sense of words such as "yards" and "pounds". This request tests if the next word is "of". If it is, then the request activates another request which looks for the material to be modified by the unit, and when found attaches the description created by "five yards" or "ten pounds" to it.

In general, locally changing the sense of some word in this manner does not involve throwing out whatever requests are indexed directly under the word, but rather just augments them. In this way, if the newly activated requests should fail to be applicable, then all of the usual requests associated with the word are still available to continue processing. However, the specially added request needs to be guaranteed priority over the usual requests associated with the word.

De-activating Requests

The final request action is the ability to de-activate requests. One important use of this action is to have requests de-activate themselves under certain circumstances. In this way, a request which notices that it is no longer appropriate can remove itself before fouling up the analysis. For example, consider the request for locally extending the senses of the word "of", described in the last section. If the next word is *not* "of", then this request must de-activate itself. Otherwise, if "of" occurs later in the input, the request would change its meaning in a way which is no longer appropriate.

Another situation in which this action is useful occurs when there are several requests which represent competing hypotheses about the meaning of a word. The request which represents the correct meaning (however chosen) must de-activate the other requests.

Tests of Requests

The tests of requests examine either the actual words of the input text or the objects on the C-LIST. We have at this point distinguished two variants of the latter:

1. Searching for the occurrence of some structure on the C-LIST with the correct properties.
2. Testing whether or not certain gaps in some structure are filled.

Clearly the properties with which the first type of test is concerned will often be conceptual or semantic. For instance, one might write a test like "is there a PP on the C-LIST?" or "is there a PP on the C-LIST which is a higher animate?" We have seen these sorts of tests in the examples discussed previously. But, these tests may also have constraints on *where* in the C-LIST they should look, such as, for example, an instruction to look only on the end of the C-LIST, or to search only preceding or following some other object on that list. Such constraints are one way of utilizing the structural information derived from word order. In other words, these constraints, which can be expressed by predicates in the test, embody expectations related to sentence structure rather than content.

These expectations are necessary whenever there are several gaps in a representational structure which have the same semantic requirements. For example, both the ACTOR and the TO slots of an ATRANS can appropriately be filled by a "higher animate". Syntactic knowledge must then be used to decide which of several appropriate gaps a structure should fill.

Two predicates are useful in constraining where on the C-LIST to look: (1) a predicate which tests whether or not one structure *precedes* another on the C-LIST; and (2) a predicate which tests whether or not one structure *follows*

another on the C-LIST. For example, the ATRANS sense of the word "give" activates the following request:

REQUEST—
TEST: Can you find a human on the C-LIST preceding the ATRANS structure?
ACTIONS: Put it in the ACTOR and FROM slots of the ATRANS

The other type of test for requests determines which of several empty gaps some candidate structure should fill on the basis of constraints on the order in which those gaps should be filled. These constraints are expressed by tests that check to see whether or not some gap is filled. For example, suppose we wanted to write a request to fill the TO slot of an ATRANS arising from the word "give". Because the same semantic constraints apply to the ACTOR and TO slots, we need a test to distinguish between them. Since, in an active sentence, the ACTOR will be seen *before* the recipient is seen, the following request will insure that the TO slot is not incorrectly filled with a structure that ought to fill the ACTOR slot:

REQUEST—
TEST: Can you find a human on the C-LIST, and is the ACTOR slot already filled?
ACTIONS: Put it in the TO slot of the ATRANS.

Searching the C-LIST

An ordering problem arises in conceptual analysis when the test of a conceptual-level request is searching the C-LIST for some object with certain properties. If more than one item on the C-LIST satisfies the test, how can the system choose between them? Let's examine the following fragment: "The girl Fred saw in the park . . ." The request activated by reading "saw" activates another request that looks for an animate actor. But there are two possibilities: "the girl" and "Fred". In cases like this, we use a recency rule to select from among several possibilities. That is, when scanning the C-LIST looking for some object which satisfies certain predicates, the system should look at the more recently added objects first.

Organizing Requests: The Recency Rule

A large part of the theory of conceptual analysis is the problem of request organization, which is related to controlling how and when requests get considered. Similar control questions arise in all production systems. (For a good

overview, see Davis and King, 1975.) The central problem is determining which control strategies are appropriate to which domains.

We have supposed, rather simplistically, that active requests are held in the R-LIST. Now consider the following example sentence: "Saul ate a big red apple." Our intuition is that the adjectives preceding the word "apple" modify the concept "APPLE" before the result is used to fill the INGEST frame added to the C-LIST by "ate". It appears that the requests activated by those adjectives are considered before the requests which were activated when the INGEST structure was added to the C-LIST. Examples like this, and others as well, have led us to adopt a rule of *request recency*. (Rules of this sort are common in production systems.) This rule states that requests are considered in the reverse order of their activation, from most recent to least. In addition to guaranteeing that the analysis proceeds according to our intuitions, this is a good heuristic principle, since newer requests represent newer, and so possibly better, information about what might be going on than older requests.

Let's look at another example: "Fred told John Bill hit Mary." When "told" is read, a request is activated which adds a conceptual structure to the C-LIST, roughly something like

(MTRANS ACTOR (NIL) MOBJECT (NIL) TO (NIL)).

In addition, requests are activated which try to fill the empty slots in this frame. One of these requests is looking for a concept to put in the MOBJECT slot. Then, when "hit" is read, a request is activated, which adds the conceptual structure

(PROPEL ACTOR (NIL) OBJECT (NIL))

to the C-LIST.

This request also activates some new requests that try to fill the empty slots in the PROPEL frame. Even though the system has already commenced scanning the list of active requests, these new requests are more recent. Thus, the requests which strive to fill gaps in the PROPEL frame are considered before the requests which strive to fill gaps in the MTRANS frame. "Bill" is used to fill the ACTOR slot of the PROPEL before the resulting structure is used to fill the MOBJECT slot of the MTRANS.

In many cases, more than one request is activated at the same time, and hence these requests have the same recency. Such a set is called a *request pool*. Thus, in order to implement the recency rule, our program uses an ordered list of request pools, each pool containing one or more requests.

The main import of the recency rule is that if a conceptual structure A is going to be used to fill a gap in some other structure B, then A is given a chance to fill its own gaps, before A is embedded in B. Thus, in the example "Fred told John Bill hit Mary", before the PROPEL structure is embedded in the MTRANS, "Bill" is used to fill the ACTOR slot of the PROPEL. In processing terms, this

effect of the recency rule can be expressed as follows: if requests which assemble and modify lower level structures and requests which assemble these sub-structures into higher level structures are active at the same time, then the former have priority over the latter.

How are requests organized within pools? A certain amount of hierarchy is clearly necessary, since requests specifically added to change the sense of a word in some local context must be guaranteed priority.

In addition to this hierarchical organization, requests are organized within pools according to the actions they perform. For example, suppose that several requests in the same pool all add structures to the C-LIST, and suppose further that more than one of these has a true test. This is how word-sense ambiguity would manifest itself in a conceptual analyzer, and the problem is to select the correct request. The algorithm which controls request consideration must not only order the requests, it must sometimes choose among them, or make them all wait for more information.

Noun Groups

An analyzer implemented along the lines we have described up to now would have great difficulty processing noun groups with more than one noun correctly. The problem is that it is possible that a concept will incorrectly use some intermediate conceptual structure to fill a gap. For example, consider the sentence "George sat on the stairway handrail." The concept representing "sat" presumably has an empty slot for the object upon which the actor was sitting, and a request which tries to fill that slot (let us agree to call it the OBJECT slot). Since a stairway is a perfectly fine object to sit on, the algorithm as described up to this point would allow that request to fill its slot with "stairway", before the entire noun group had been analyzed. So, the problem is to somehow prevent any request trying to fill the OBJECT slot from firing prematurely, or correct the situation if that should happen. It is clear that when analyzing noun groups the control structure must consider requests in some manner different than so far described. Our problem is once again one of controlling request consideration.

To handle this, we use the following procedure for request consideration in noun groups: Consider only the requests associated with each incoming word, until the end of the noun group. That is, if the program is in a noun group, and a new word is input, then only the newly activated requests associated with that word are considered. Prior requests are not considered until the noun group is finished.

With this rule for processing requests during noun groups, the entire structure representing "stairway handrail" would be constructed *before* the request which strives to fill the OBJECT slot of the sitting action could be considered.

This rule for controlling request consideration has the additional effect of insuring that the head noun of the noun group is discovered before adjectives are attached. For example, consider how this algorithm would analyze the input

"blue car seat". First the request associated with the word "blue" would be activated. Such a request would strive to fill the COLOR slot of some physical object to the right. Next, the request associated with the word "car" would add a structure to the C-LIST representing the concept of automobile. However, since the analyzer is in noun group mode, the prior expectation from "blue" would *not* be considered at this time. Finally, the word "seat" is read, and another structure is added to the C-LIST. Now, the structure representing "car seat" is built, and then the request associated with "blue" is considered. It would then ascribe the attribute of being colored blue to the seat, not to the car. Under certain circumstances, of course, this could be wrong, e.g., "red stairway railing" might refer to the railing of a red stairway, instead of the red railing of a stairway. We believe that a general solution to problems of this kind will depend on better understanding of the role of memory in parsing. However, the process described here does have the virtue of allowing noun groups to be completed before the resulting analysis is used in larger structures. For example, if the phrase "blue car seat" is embedded in the sentence "George sat on the blue car seat," then the analysis would determine that George sat on the seat, not on the car.

In order to use the process we sketched out above, an analyzer must first have requests organized by recency, and second, be able to determine when it should be in noun group mode and when it should be in normal mode. We have used, with slight modification, Gershman's (1977) heuristics for determining noun group boundaries. Some of these rules depend on conceptual, and some on syntactic, knowledge.

A Disambiguation Algorithm

Using requests to check the context in order to disambiguate themselves requires no special addition to the control structure of a conceptual analyzer, since the work is done by the requests. Using context to choose among competing requests, however, requires augmenting the mechanism which controls request consideration. We will make use of only those obvious connections between structures which arise when one can fill a gap in the other. The algorithm should be flexible, which means that if there is not enough evidence to make a selection, the system must be able to wait and try again later.

We will assume that requests are organized within pools according to their actions, and will concern ourselves with those requests which strive to add some structure to the C-LIST. The mechanism for controlling these requests within each pool must be applied any time the requests are considered. This mechanism works as follows:

1. Collect those requests which have true tests.
2. If more than one has a true test, then check the structures which these add to the C-LIST, to see if they can either fill a gap in some other structure already on the C-LIST, or can use some other structures to fill one of their gaps.

At this point, three things can happen: either none of the requests succeed in this check for connections, only one succeeds, or several succeed. The first case is easy to handle:

3a. If none of the requests succeed in the check for connections (step 2), then none should be allowed to fire (i.e., postpone making a decision).

The second case is also relatively easy, although complexities arise if the choice made turns out to be a bad one:

3b. If only one of the requests succeeds in the check for connections (step 2), then perform the actions of the successful request and de-activate the rest.

The third case is quite difficult:

3c. If neither of the above apply, then don't execute any of the requests. A more complex decision needs to be made.

This third case requires postponing a decision until one or the other structure is found to be the "best connected". In Wilks' system (1976), the "best connected" is the "most connected", and we use this measure as well. However, Wilks' analyzer performs this comparison only at what are essentially clause boundaries, and does not rule out any possibilities until then, whereas our system chooses the first structure to become more connected than any other. In this third case where several requests succeed in the check for connections, it seems plausible that those which do not could be de-activated immediately, thus reducing fairly quickly the set of viable alternatives.

The use of some disambiguation process as outlined above places new requirements on the control structure of a conceptual analyzer. In particular, we have often depended on a fairly consistent correspondence between the order of input of some word, and the location in the C-LIST of any associated structure. But, since the requests which add structures to the C-LIST could be prevented from firing immediately by the disambiguation process, such a correspondence is no longer likely. The solution is to change the control and data structures a bit to guarantee the relationship between order of input and order on the C-LIST. The easiest way to do that is as follows: whenever a pool of requests is activated, a new empty node is to be added to the C-LIST. Should a request in that pool add some structure to the C-LIST, it should add it in the corresponding node, not at the end as previously described. Thus, the C-LIST consists of a list of such nodes, some with structures, and some empty. An empty node often has associated requests trying to add structures to the C-LIST in that position.

A DETAILED EXAMPLE

This section presents a detailed example of how the Conceptual Analyzer actually runs. The input we will use is: "A small plane stuffed with 1500 pounds of marijuana crashed 10 miles south of here." The dictionary entries exhibited, although not particularly general, do illustrate the variety of actions which the analyzer must perform. The example will trace the execution monitor, describing the states of the C-LIST and of the request pools as the sentence is analyzed. We assume that the C-LIST is NIL and there are no request pools at the start. (Note: The request names (REQ1,REQ2, etc.) are generated dynamically during the parse. We have added them to the sample definitions for the sake of clarity.)

GET THE NEXT ITEM: The first word of the sentence is "A". The program switches from normal to noun group mode. (The current mode is determined by using Gershman's (1977) heuristics for noun group boundaries.)

CONSIDER REQUESTS LOOKING FOR SPECIFIC WORDS: (There aren't any.)

LOAD THE REQUESTS FROM THE DICTIONARY ENTRY: The entry for "a" is:

```
(DEF A
 (REQUEST REQ1
      [TEST: "find a concept following 'a' "]
  [ACTIONS: "add (REF (INDEF)) to it"]]
```

The request is named REQ1 by the system and activated in POOL-1. This request adds the gap REF, and fills it with INDEF.

CONSIDER REQUESTS IN NOUN GROUP MODE: REQ1 is considered but does not fire.

GET THE NEXT ITEM: "small".

CONSIDER REQUESTS LOOKING FOR SPECIFIC WORDS: (There aren't any.)

LOAD THE REQUESTS FROM THE DICTIONARY ENTRY: The entry for "small" is:

```
(DEF SMALL
 (REQUEST REQ2
      [TEST: "find PP on C-LIST following 'small' "]
  [ACTIONS: "add (SIZE (SMALL)) to it"]]
```

The request is named REQ2 by the system and activated in POOL-2.

CONSIDER REQUESTS IN NOUN GROUP MODE: REQ2 is considered but does not fire.

GET THE NEXT ITEM: "plane".

CONSIDER REQUESTS LOOKING FOR SPECIFIC WORDS: (There aren't any.)

LOAD THE REQUESTS FROM THE DICTIONARY ENTRY: The entry for "plane" is:

```
(DEF PLANE
 (REQUEST REQ3
      [TEST: T]
  [ACTIONS:
    "add (PP CLASS (VEHICLE) TYPE (AIRPLANE)) to C-LIST"]]
```

Recall that all structures are added on the end of the C-LIST. The request is named REQ3 by the system and activated in POOL-3.

CONSIDER REQUESTS IN NOUN GROUP MODE: REQ3 is considered and fires. The result is that POOL-3 is now empty, and the C-LIST = (CON1) where CON1 = (PP CLASS (VEHICLE) TYPE (AIRPLANE)).

GET THE NEXT ITEM: "stuffed". The program switches from noun group to normal mode. Recall that normal mode requires the execution monitor to immediately consider all active requests. So, the monitor checks all of the request pools, from most recent to least. REQ2 and REQ1 both fire, and the end result is that all the request pools are empty, with CON1 = (PP CLASS (VEHICLE) TYPE (AIRPLANE) SIZE (SMALL) REF (INDEF)).

CONSIDER REQUESTS LOOKING FOR SPECIFIC WORDS: (There aren't any.)

LOAD THE REQUESTS FROM THE DICTIONARY ENTRY: The entry for "stuffed" is:

```
(DEF STUFFED
 (REQUEST REQ4
      [TEST: T]
  [ACTIONS:
   "Set STR0 to (PTRANS ACTOR (NIL)
                        OBJECT (NIL)
                        TO (INSIDE PART (NIL)))
    Add STR0 to C-LIST"
```

```
''activate
 (REQUEST REQ5
      [TEST: ''the next word is WITH'']
 [ACTIONS:
  ''activate in the next pool, marked high priority:
   (REQUEST REQ6
        [TEST: ''find PP on C-LIST and (TO PART) slot of
                 STR0 is filled'']
    [ACTIONS: ''put it in the OBJECT slot of STR0''])
   (REQUEST REQ7
        [TEST: ''find PP on C-LIST, preceding STR0'']
    [ACTIONS: ''add (REL (STR0)) to it and
                 put it in (TO PART) slot of STR0'']]]
```

The local variable STR0 (which is replaced by a system-unique symbol) is set to the PTRANS when the request is executed. STR0 is used to save the structure for other requests to reference. The request is named REQ4 by the system and activated in POOL-4.

CONSIDER REQUESTS IN NORMAL MODE: REQ4 is triggered, so now C-LIST = (CON1 CON2), where CON2 is the PTRANS. Another request is activated as REQ5, which is put in the special pool looking for particular words.

GET THE NEXT ITEM: ''with''.

CONSIDER REQUESTS LOOKING FOR SPECIFIC WORDS: REQ5 is triggered. Because of the nature of the action of REQ5, the execution monitor proceeds, noting it must add two new requests, REQ6 and REQ7, to the next request pool.

LOAD THE REQUESTS UNDER THE DICTIONARY ENTRY: In this instance, it doesn't really matter what the requests listed under ''with'' are, since the requests added by REQ5 are the proper sense of the word in this local context. Hence, when POOL-5 is built, REQ6 and REQ7 are added to it. They have a higher priority than the standard requests listed under ''with'' in the dictionary.

CONSIDER REQUESTS IN NORMAL MODE: REQ7, which builds the REL structure, is triggered, and so now C-LIST = (CON1), where CON1 =

```
(PP CLASS (VEHICLE)
    TYPE (AIRPLANE)
    SIZE (SMALL)
    REL (PTRANS ACTOR (NIL)
                OBJECT (NIL)
                TO (INSIDE PART (CON1))))
```


The only remaining active request is REQ6, which strives to fill the OBJECT slot of the PTRANS.

GET THE NEXT ITEM: ''1500''. The execution monitor changes to noun group mode.

CONSIDER REQUESTS LOOKING FOR SPECIFIC WORDS: (There aren't any.)

LOAD THE REQUESTS FROM THE DICTIONARY ENTRY: The program recognizes that ''1500'' is a number, and for the sake of convenience generates a corresponding literal atom *1500*, with a definition like this:

```
(DEF *1500*
 (REQUEST REQ8
      [TEST: T]
  [ACTIONS: ''add (NUM VAL (1500)) to C-LIST'']]
```

This is named REQ8 by the system, and activated in POOL-6.

CONSIDER REQUESTS IN NOUN GROUP MODE: REQ8 is triggered, and the C-LIST = (CON1 CON3) where CON3 = (NUM VAL (1500)).

GET THE NEXT ITEM: ''pounds''.

CONSIDER REQUESTS LOOKING FOR SPECIFIC WORDS: (There aren't any.)

LOAD THE REQUESTS FROM THE DICTIONARY ENTRY: the entry for ''pound'' is:

```
(DEF POUND
 (REQUEST REQ9
      [TEST: T]
  [ACTIONS:
   ''Set STR0 to (UNIT TYPE (LB)
                      NUMBER (NIL))
     Add STR0 to C-LIST''
   ''activate
    (REQUEST REQ10
         [TEST: ''find a number preceding STR0
                  on the C-LIST'']
     [ACTIONS: ''put it in the NUMBER slot of STR0''])
    (REQUEST REQ11
         [TEST: ''next word is OF'']
```

```
[ACTIONS:
 "activate in the next pool, marked high priority:
  (REQUEST REQ12
       [TEST: "if you find a PP on C-LIST following STR0"]
   [ACTIONS: "add (AMOUNT STR0) to it"]]]
```

This is named REQ9 and activated in POOL-7 by the system.

CONSIDER REQUESTS IN NOUN GROUP MODE: REQ9 is triggered, so that the C-LIST = (CON1 CON3 CON4) where CON4 is the representation for "pounds". Also, the request looking for a number is activated as REQ10 in POOL-8, which should be considered in noun group mode. The other request is activated in the pool for requests testing for particular words as REQ11. REQ10 triggers, and C-LIST = (CON1 CON4), where CON4 = (UNIT TYPE (LB) NUMBER (NUM VAL (1500))).

GET THE NEXT ITEM: "of".

CONSIDER REQUESTS LOOKING FOR SPECIFIC WORDS: REQ11 is triggered, and because of the nature of its action, the execution monitor proceeds, noting it must add a new request, REQ12, to the next request pool.

LOAD THE REQUESTS FROM THE DICTIONARY ENTRY: Again, since the sense of this word is locally changed, it doesn't matter what they are. However, POOL-9 is built, and REQ12 is activated in it.

CONSIDER REQUESTS IN NOUN GROUP MODE: REQ12 is considered, but fails to fire.

GET THE NEXT ITEM: "marijuana".

CONSIDER REQUESTS LOOKING FOR SPECIFIC WORDS: (There aren't any.)

LOAD THE REQUESTS FROM THE DICTIONARY ENTRY: the entry for "marijuana" is:

```
(DEF MARIJUANA
 (REQUEST REQ13
      [TEST: T]
  [ACTIONS: "add (PP CLASS (PHYSOBJ) TYPE (MARIJUANA)) to C-LIST"]]
```

This is named REQ13 by the system and activated in POOL-10.

CONSIDER REQUESTS IN NOUN GROUP MODE: REQ13 is triggered, so now C-LIST = (CON1 CON4 CON5), where CON5 is the representation for "marijuana".

GET THE NEXT ITEM: "crashed". The execution monitor switches from noun group mode to normal mode, and so immediately considers all active requests. By recency, REQ12 is considered first, and triggers. Now C-LIST = (CON1 CON5) where CON5 =

```
(PP CLASS (PHYSOBJ)
    TYPE (MARIJUANA)
    AMOUNT (UNIT TYPE (LB)
                 NUMBER (NUMBER VAL (1500))))
```

REQ6, which was activated to fill the OBJECT slot of the subordinate PTRANS, is triggered; CON5 fills the OBJECT slot, and C-LIST = (CON1) again. There are no more active requests.

CONSIDER REQUESTS LOOKING FOR SPECIFIC WORDS: (There aren't any.)

LOAD THE REQUESTS FROM THE DICTIONARY ENTRY: The entry for "crash" is:

```
(DEF CRASH
 (REQUEST REQ14
      [TEST: T]
  [ACTIONS:

  "Set STR0 to (PROPEL ACTOR (NIL)
                       OBJECT (PP CLASS (PHYSOBJ) TYPE (GROUND))
                       PLACE (NIL))
   Add STR0 to C-LIST"

  "activate
   (REQUEST REQ15
        [TEST: "find a PP which is a physical object
                preceding STR0 on the C-LIST"]
    [ACTIONS: "put it in the ACTOR slot of STR0"])
   (REQUEST REQ16
        [TEST: "find a PP which is a location on the C-LIST"]
    [ACTIONS: "put it in the PLACE slot of STR0"]]]
```

This is REQ14, activated in POOL-11. (The filler of the object slot of the PROPEL (ground) is a default that can be overwritten.)

CONSIDER REQUESTS IN NORMAL MODE: REQ14 is triggered, and the C-LIST = (CON1 CON6) where CON6 is the representation for "crash". Also, two new requests, REQ15 and REQ16 are activated in POOL-12. These are considered, and REQ15 is triggered, which places CON1 in the ACTOR slot of CON6, so that C-LIST = (CON6). REQ16 is not triggered.

GET THE NEXT ITEM: ''10''. Switch to noun group mode.

CONSIDER REQUESTS LOOKING FOR SPECIFIC WORDS: (There aren't any.)

LOAD THE REQUESTS FROM THE DICTIONARY ENTRY: Again, the literal atom *10* is used in place of the number 10, and the following entry is constructed:

```
(DEF *10*
 (REQUEST REQ17
  [TEST: T]
  [ACTIONS: ''add (NUM VAL (10)) to C-LIST'']]
```

This is REQ17, activated in POOL-13.

CONSIDER REQUESTS IN NOUN GROUP MODE: REQ17 is triggered, so C-LIST = (CON6 CON7) where CON7 = (NUM VAL (10)).

GET THE NEXT ITEM: ''miles''.

CONSIDER REQUESTS LOOKING FOR SPECIFIC WORDS: (There aren't any.)

LOAD THE REQUESTS FROM THE DICTIONARY ENTRY: The entry for ''mile'' is:

```
(DEF MILE
 (REQUEST REQ18
      [TEST: T]
  [ACTIONS:
   ''Set STR0 to (UNIT TYPE (MILE)
                  NUMBER (NIL))
     Add STR0 to C-LIST''
   ''activate
     (REQUEST REQ19
          [TEST: ''find a number preceding STR0 on C-LIST'']
      [ACTIONS: ''put it in the NUMBER slot of STR0'']]]
```

This is REQ18, activated in POOL-14.

CONSIDER REQUESTS IN NOUN GROUP MODE: REQ18 is triggered, so C-LIST = (CON6 CON7 CON8) where CON8 is the representation for ''mile''. Also, REQ19 is activated in POOL-15, and then also considered. It fires, so that the C-LIST = (CON6 CON8) where CON8 = (UNIT TYPE (MILE) NUMBER (NUM VAL (10))).

GET THE NEXT ITEM: ''south''.

CONSIDER REQUESTS LOOKING FOR SPECIFIC WORDS: (There aren't any.)

LOAD THE REQUESTS FROM THE DICTIONARY ENTRY: The entry for ''south'' is:

```
(DEF SOUTH
 (REQUEST REQ20
      [TEST: T]
  [ACTIONS:
   ''Set STR0 to (LOC PROX (NIL)
                      DIR (SOUTH)
                      DIST (NIL))
     Add STR0 to C-LIST''
   ''activate
     (REQUEST REQ21
          [TEST: ''find a distance unit preceding
                   STR0 on the C-LIST'']
      [ACTIONS: ''put it in the DIST slot of STR0''])
     (REQUEST REQ22
          [TEST: ''the next word is OF'']
      [ACTIONS:
       ''activate in the next pool, marked as high priority:
        (REQUEST REQ23
             [TEST: ''find a structure representing a location
                      following STR0 on the C-LIST'']
         [ACTIONS: ''put it in the PROX slot of STR0'']]]
```

This is REQ20, activated in POOl-16.

CONSIDER REQUESTS IN NOUN GROUP MODE: REQ20 is triggered, and now C-LIST = (CON6 CON8 CON9) where CON9 is the structure representing ''south''. Also, REQ21 is activated in POOL-17, and REQ22 in the special pool of requests looking for specific words. REQ21 fires, and so now C-LIST = (CON6 CON9) where CON9 =

```
(LOC PROX (NIL)
     DIR (SOUTH)
     DIST (UNIT TYPE (MILE) NUMBER (NUM VAL (10))))
```

GET THE NEXT ITEM: ''of''.

CONSIDER REQUESTS LOOKING FOR SPECIFIC WORDS: REQ22 is triggered, but waits until the next request pool is activated to add a request, REQ23.

LOAD THE REQUESTS FROM THE DICTIONARY ENTRY: Again, since the meaning of "of" will be locally changed, it doesn't matter what the dictionary request for "of" has in it. However, REQ23 is activated in POOL-18.

CONSIDER REQUESTS IN NOUN GROUP MODE: REQ23 is considered but does not fire.

GET THE NEXT ITEM: "here".

CONSIDER REQUESTS LOOKING FOR SPECIFIC WORDS: (There aren't any.)

LOAD THE REQUESTS FROM THE DICTIONARY ENTRY: In this case, the entry is:

```
(DEF HERE
 (REQUEST REQ24
      [TEST: T]
  [ACTIONS: "add (HERE) to C-LIST"]]
```

This becomes REQ24 in POOL-19. (The representation for "here" is simply a place-holder. In a newspaper story understander, it would be replaced by information from the dateline.)

CONSIDER REQUESTS IN NOUN GROUP MODE: REQ24 is triggered, and changes the C-LIST to (CON6 CON9 CON10).

GET THE NEXT ITEM: "period". Switch to normal mode of the execution monitor. This means checking all requests. REQ24 is triggered and C-LIST = (CON6 CON9) where CON9 =

```
(LOC PROX (HERE)
      DIR (SOUTH)
      DIST (UNIT TYPE (MILE) NUMBER (NUM VAL (10))))
```

Then REQ17 is considered and triggers, filling the PLACE slot of CON6 with CON9. Since period signals the end of a sentence, there are no more active requests. The analysis has constructed one complete conceptualization, and we are done. C-LIST = (CON6) where

```
CON6 =
(PROPEL ACTOR (CON1)
        OBJECT (PP CLASS (PHYSOBJ) TYPE (GROUND))
        PLACE (CON9))
```



```
CON1 =
(PP CLASS (VEHICLE)
   TYPE (AIRPLANE)
   SIZE (SMALL)
   REF (INDEF)
   REL (PTRANS ACTOR (NIL)
               OBJECT
               (PP CLASS (PHYSOBJ)
                  TYPE (MARIJUANA)
                  AMOUNT
                  (UNIT TYPE (LB)
                        NUMBER (NUM VAL (1500))))
               TO (INSIDE PART (CON1))))
CON9 =
(LOC PROX (HERE)
     DIR (SOUTH)
     DIST (UNIT TYPE (MILE)
                NUMBER (NUM VAL (10))))
```