# How a Programmer Understands a Program: A Model

by

Ruven Brooks

Technical Report #97

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

Z
699
C3
no. 97

# How A Programmer Understands A Program: A Model

Ruven Brooks

Department of Information and Computer Science

University of California - Irvine

Irvine, Ca. 92715

--------------------

A portion of this work was done using the facilities of the National Center for Software Development and Computing Techniques, Tata Institute for Fundamental Research, Bombay, India.

## Abstract

In a large variety of programming situations, a programmer is required to understand a program that someone else has written. A model has been created for the behavior seen in the verbal protocol of a programmer on a sample understanding task. The model is based on a theory of understanding which stresses the role of the programmer's apriori hypotheses or guesses about the program structure. Organization of the model is that of a production system, a structure which appears particularly well-suited to the asynchronous, non-sequential nature of the input.

Understanding how a computer program works is an important component of many different programming tasks. Modifying or debugging a program certainly requires it, and understanding how a program works may even in some cases be a prerequisite to using it. In fact, the ease with which a program can be understood is considered to be a sufficiently important property of a program that a number of new techniques, often referred to as "structured" programming and "modular" programming have been developed to enhance program understandability. A theory of how people understand programs would, therefore, be of both practical and academic interest.

A useful starting point for such a theory is the general theory of understanding task instructions that has been developed by Hayes & Simon (1974; Simon & Hayes, 1976). Essentially, their theory views the process of understanding a problem as one of building an internal representation of the basic elements of the problem. These basic elements include a set of objects along with their properties and the relations among them, operators for altering these properties and relations, a specification of the initial state of the objects, and some specification of the desired final state. Thus, to understand a chess problem, the problem-solver must know the pieces involved and the moves they can make, their starting positions on the board, and the condition, such as "checkmate," that is to describe the final relationships among the pieces. When such an internal representation has been created, the problem-solver has the necessary information to begin work on the problem. (Note that this representation of the problem need not be the one the problem poser intended; the solver may have misunderstood the problem.)

In the Hayes and Simon model, this internal representation is created by a process that first scans the text of the written instructions for the sets of objects and the relations among them. A representational structure is created to hold these objects and relations so that information on initial and goal states may be stored in it. A search is then made for potential operators that can change the relations among objects, and semantic memory is searched for semantic interpretations of the objects. Finally, the semantic interpretations are converted to a form compatible with the representational structure that holds the objects and relations.

While the theory presented here shares the same essential flavor of the Hayes & Simon work, the understanding of programs is seen as differing from the understanding of task instructions in two major respects: First, the nature of the internal represention differs significantly for a program as versus the kind of reasoning problems used by Hayes & Simon. For the reasoning problems, objects are given little or no semantic interpretation. They are treated as atomic entities with no further internal structure. For the program, however, objects have a quite complex and extensive semantic structure. The essence of this structure is seen to be a mapping between objects and operations in the real world and those in the program. Examples of real world operations are squaring a number and finding the verb in a sentence. Example of objects in a program are the expression, N**2, and the variable, VERB. The mapping between the first two would consist of the information that the value of N is the number and that **2 causes the number to be multiplied by itself. The mapping between the second pair would be the information that the value of VERB is the string of characters standing for the verb in the sentence.

Such mappings preserve only a few selected properties of the real world object. For example, the number of digits required to write a quantity is a property of real-world numbers which a program representation does not preserve. The program representation also probably does not preserve the typeface in which the number was written.

For anything but the simpliest information, mappings will usually be organized into a heirarchical structure. Thus, the meaning or interpretation given to a single variable name will play a role in the meaning given to pieces or parts of program which use the variable.

The way in which these mappings are established forms the second major point of difference between the understanding of programs and the understanding of task instructions. In the Hayes and Simon model, the information necessary for establishing the internal representations is derived almost entirely from the problem directions themselves, with only a minimal use of other knowledge by the problem solver. The authors state "the UNDERSTAND program depends primarily on a knowledge of the English language, and secondarily, upon a knowledge of the semantics of a few basic operators" (Simon & Hayes, 1976,p.278). The authors recognize that this situation may be due to the peculiarities of the puzzles that they are using, and that other kinds of problems will probably require much more extensive use of other knowledge by the programmer.

Understanding a computer program is clearly a task which will require extensive use of other knowledge. Several arguments can be advanced to indicate that this must be the case. First, in comparision with English sentences, statements in a computer program taken by themselves are relatively semantics free. Contrast the statements, "If the person is an unmarried woman, search the population for a mate," and "PENTRY[I,4]:=PENTRY[I,6] * TCARD." While the reader can probably interpret the first sentence directly and recognize the operation it is specifying, a great deal more information is required before he can guess that the program statement has to do with a payroll calculation. Moreover, the problem with the program is not just one of establishing context. The statement given above might easily occur in the first 10 statements of a program, long before any extensive context could be established. If the program has few or no comments, then the necessary knowledge to understand the statement must come from outside the program.

A second argument for the extensive use of knowledge besides the program text in understanding a program comes from timing considerations. The task directions in one of the Hayes and Simon studies contained about 10 main clauses. Subjects were able to understand the problem in about 5 minutes. Each statement in a programming language can be considered roughly equivalent to a main clause in an English sentence in terms of the information conveyed. On the same time scale, then, it should take a subject about 50 minutes to understand a 100 statement program to the point, say, where he could locate where a certain computation was performed. In fact, if a subject is given a brief description of what the program is intended to do, this level of understanding can be reached in 10-15 minutes.

This suggests that the subjects comes to the program with some store of knowledge that greatly reduces the information that must be actually extracted from the program text.

Given that understanding a program does involve extensive use of knowledge from outside of the program text, an adequate theory of understanding programs should specify the source of this knowledge. The concept put forth here is that this knowledge is derived from a set of hypotheses about the structure and organization of the program. These hypotheses take the form of the mappings described earlier between program elements and real-world objects and relations. They include guesses as to the data structures and algorithms used within the program, to the format and encoding schemes used, to the probable output, and to the program organization into modules and subprograms. An example of such a hypothesis would be that a chess playing program must do a tree search and must, therefore, use a stack to keep track of the tree.

Given these hypotheses, the process of understanding a program becomes one of verifying the hypotheses against the actual program. This verification is accomplished by scanning the program text for cues as to what the program is doing, and checking them for consistancy against the hypotheses. Should this scanning process result in some or all of these hypotheses being disconfirmed, then new hypotheses will be generated and verification attempted until a set of hypotheses is found which corresponds to the program. An important question, then, is how the hypotheses are generated.

If the assumption is made that the programmer is given an arbitrary program of arbitrary type, then the source of the hypotheses will be difficult to determine; perhaps, it will depend on the kind of program the programmer last worked on, or on what he discussed at lunch. In most real world situations in which a program must be understood, however, the programmer is usually told what the program is intended to do. For example, if the programmer is asked to modify an accounting program, he is usually told how the program currently performs. Such information is, of course, a fertile basis on which to generate hypotheses, and, if it is available, it should play the dominant role in hypothesis generation.

Summarizing the theory presented here, understanding how a computer program works is viewed as a process of establishing mappings bewteen objects in the real world and the corresponding structure within the program. This mapping is established via a process of hypothesizing and verification. If available, information about what the program is intended to do will be a primary source for these hypotheses.

## The Model

To illustrate the sufficiency of this theory for explaining a major element in how programs are understood, a model, in the form of a computer program, is being developed for an actual piece of understanding behavior. The behavior was obtained by giving an experienced programmer a program that someone else had written and asking him to tell how the program operated. As he proceeded in this task, his knowledge about the program, as revealed in his statements, gradually increased. The order and manner in which this increase occurred is the facit of the behavior which the program attempts to model.

### Behavioral Data

Before presenting the model, itself, a more detailed explanation of the behavioral data and the manner in which is was collected is worthwhile. The programmer who served as subject in this study was, at the time he performed the task, a user consultant for a university computing center, a post which he had held for more than a year. Prior to that, he received an undergraduate degree in computer science. The program which he was asked to understand was written in FORTRAN, a language with which he was very familiar, and was 93 statements in length. It performed a random-walk simulation of an object moving on a rectangular grid. The program had the unusual feature that, instead of generating random numbers internally, it read them in, 1Ø at a time, from an external file.

The program was printed on 8 1/2 by 11 inch sheets with 26 statements per page. In comparision with the normal 52 line page, this had the effect of somewhat restricting the amount of program that the programmer could view at a given time; consequently, he had to turn pages more frequently. Each line on the page was numbered in red on the left-hand margin.

While performing the task, the subject was seated at a table with the program listing placed in front of him. A microphone connected to a taperecorder was also placed on the table.

General instructions to the subject were that he was to be given a listing of a program and asked to understand what it did. While performing this task, he was to "talk aloud" about what he was doing and also to state the numbers of the lines he was looking at. The instructions concluded with the following statement:

> "The program does a random-walk simulation of movement. In a random-walk simulation of movement, an object is presumed to be on a rectangular grid. At each time interval, a random number is used to determine which square an object moves to next. In the program that will be given to you, the object can be considered to move for 1ØØ time intervals. At the end of these moves, the program prints out the squares which were visited in order of frequency of visit. What you are to do is to go through the listing and give a line by line description

of how the program works. Be sure to state the line numbers."

The subject spent approximately 15 minutes going over the program. At the end of this period, he was still unable to figure out what one section of program did. On being told that it was a Shell sort, he stated that he now understood what the entire program did.

The tape recording of the subjects comments was transcribed and broken into phrases; the resulting protocol consisted of 384 such phrases. A short sample of this protocol is given below:

S46 Yeah, ahh, if this second argument
S47 is greater than 10
S48 greater than or equal 10
S49 Ok, if it's less than 10 then you're
S50 just adding one to the second argument
S51 and returning
S52 You do read in the buffer
S53 and then you set that second argument equal to one.

These protocols were analyzed for two different sorts of information:

1. Indications of what part of the program the subject was attending to. These include statements of line numbers, page turnings, and the use of variable names or values which occurred only at certain locations.

2. Indications of what information about the program the subject was acquiring. These included statements about the actions of specific pieces of code as well as more

indirect information about the overall organization of the program into sections or segments.

An example of information of the first sort is that at lines 46-53 of the protocol given above, the subject was probably looking at lines 86-89 of the program. The basis for this inference is that in line 41 of the protocol the subject states that he is looking at program lines 84, and the number,1Ø, mentioned in the protocol occurs in program lines 86-89.

An example of information of the second sort is that in those same lines of protocol the subject has just discovered that the subroutine, RANSUB, reads in something from an external file, rather than generating the random numbers directly. Further, it also indicates that he realizes that the subroutine alters the value of its second argument, though he gives no indication of realizing that the second argument is a pointer into the buffer.

## Model Structure

### Overview

A simple way to view the structure of the model is that it takes as input of the first kind of information, in the same order as the subject does, and produces as output the same information of the second kind. More precisely, a description of what the subject is presumed to be looking at in the program listing is placed, piece by piece, into a database called the STM. For each piece, a set of condition action rules - a production system - is applied to the database. The conditions of each rule specify the presence in the database of information about something the subject may have seen and the context in which he may have seen it. The actions of the rule are to modify the database to indicate inferences made from this perception about the program structure and organization.

## Knowledge Structures

The STM is structured as an ordered list of discrete items. These items include both the inputs to the model and the representations of knowledge that the model has about the program. While there is no explicit limit on the size of STM, it is conceived of as being small, on the order of 10-50 items.

Inputs to the model appear as new items in the STM. These inputs take the form of deep-structure representations of what the subject is assumed to be seeing or hearing at a given time. The adjective, "deep-structure," is used to indicate that that the information has already been processed perceptually, and that what is entered into the model is the result of this processing. Thus, when the subject is assumed to be looking at the program statement, "INTEGER RANBUF(10), PPTR," at line 86, the input appears in the model as:

```
SEGMENT 86 86
        DECLARATION:
                INTEGER (RANBUF 10), PPTR
```

Note that the perceptual processing is presumed to have supplied the information that an INTEGER statement is a form of declaration statement.

Perceptions in the model can be of variable size. While the perception given above is only one line long, perceptions of several lines in length are also possible.

The main action that takes place in the model is the conversion of these perceptions into representations of knowledge about the program. These representations are presumed to be organized heirarchically, with representations for individual expressions forming representations for lines which form segments, which, in turn, form still larger units. Each unit at any level is assigned a meaning, and the meanings of units at a lower level combine to form the higher level meanings. No apriori structure is imposed on any given amount of code so that, in one place, a line of code may be treated as a single unit while, in another, one line can be broken down into several levels of sub-expressions.

Inside the model, these knowledge representations appear as elements in the STM. These elements can be of two types: OBJECTs represent atomic objects such as variables or labels; PARTs represent segments of code. Each OBJECT or PART consists of a collection of attributes and values, such as its name, its function in the program, and whether it has any subparts.

An example of a PART is:

```
   Name: SUBROUTINE
   Number: 6
   Function: Generates random
numbers.
   Location: 82-89
   Subparts: DECLARATION 83
       READ 84 84
       .
       .
       .
```

In addition to the PARTs and OBJECTs themselves, the STM also contains the model's hypotheses about what PARTs and OBJECTs might possibly exist on the basis of a priori knowledge about what the program is intended to do. These take the form of HYPOTHESIZE elements which have the same form as PARTs and OBJECTs except that they are preceeded by the token, HYPOTHESIZE. Only the guessed-at information is present in the item; missing information, to be filled in if the hypothesis is confirmed, is indicated by question marks. An example of a HYPOTHESIZE element is:

```
HYPOTHESIZE
   PART
     Name: ?
     FUNCTION RANDOM-NUMBER-GENERATOR
```

Location: ?
Subparts: ?

As the model runs, the effect of the input is to cause changes in the PARTs and OBJECTs and the hypothesized PARTs and OBJECTs present in STM. Among the forms these changes can take are the addition of new PARTs and OBJECTs, the elaboration of existing PARTs and OBJECTs, and the confirmation or disconfirmation of hypotheses. At any given point in time, however, the particular structure of PARTs and OBJECTs (including hypothesized ones) which is contained in STM represents the model's entire knowledge about the program.

## Control Structure

The changes in the STM as a result of the input are made by a production system, the sole control structure in the model. The particular type of production system used here and the language used to express it are derived from the PSG system (Newell & McDermott,1975). The production system consists of an ordered set of condition- action rules. On each cycle of the system, the conditions of each rule are scanned in order. When a rule is found whose conditions are met, the action portion of the rule is carried out, and the scan begins again from the beginning of the list.

The conditions of a rule consist of a set of items, each of which must be present in STM for the condition to be met. Items needn´t be specified exactly to meet a condition; instead, a pattern language may be used to describe an item in general terms. Thus, to specify a PART which is a random number generator, the condition specification may contain:

```
PART *ANY* (FUNCTION RANDOM-NUMBER GENERATOR) *REST*
```

The *ANY* and *REST* tokens are used to indicate that a match may be made against any PART in STM which contains any single item occurring at the position of the *ANY* and one or more items occuring after the position of the *REST*. The condition would be met by the following STM item:

```
PART
        17 SUBROUTINE
        FUNCTION: RANDOM-NUMBER-GENERATOR
        LOCATION: 83 92
        PARTS: ?
```

The complete invoking conditions for a single production may consist of a whole sequence of such pattern items. The normal assumption is then that all the patterns must be matched for the rule to be invoked. Moreover, the matching is presumed to take place from left to right across STM, without replacement (i.e., each STM item can be used to match one and only one pattern). Additionally, it is possible to include as part of a condition a specification of a set of alternative patterns, at least one of which must be matched by an element in STM. Last, part of a condition may be the specification of the absence in STM of items matching a particular pattern.

The action part of a production consists of a list of actions to be performed. These actions all affect the contents of STM, and they can be of four different types: addition of new elements, replacement of old ones, removal of old elements, and movement to the front of STM of elements which are already in the STM. (This later action will affect the order in which elements in STM are matched.) Zero or more of each of these kinds of action make up the list of actions of a production.

## Model Operation

At the beginning of model operation, the STM is considered to be empty of information relevant to the task. In response to this condition a production fires off which brings the first perception into STM. Other productions respond to this perception and convert it into PARTs and OBJECTs. This converstion process may be completable with just the perception at hand; alternately, more information may be needed. In the former case, perception is a relatively passive processes, and the model "sees" just the next input in sequence, as would be the case if the subject´s eye were just scanning down the page. In the later case, however, perception is treated as an action, and the model sets up a goal, in the form of a LOOK-FOR element, to "see" a particular thing. Figure 1 illustrates the operation of these perceptual mechanisms by showing the partial contents of STM before and after the firing several productions.

- insert Figure 1 about here -

The way a particular perception is processed is highly dependent on the context in which it occurs. This is true in two respects: it affects which information is extracted from a perception and it affects how the information is incorporated into other structures. As an example of the first effect, when an assignment statement occurs in the context of a group of initializations, only the information that an assignment has occurred to a variable is retained; the particular value assigned is ignored. An example of the second effect is when a declaration statement occurs after a subroutine heading; then, the information about the declaration is presumed to belong to the subroutine, rather than to the main program. Figure 2 shows both of these effects:

- insert Figure 2 about here -

## Major Characteristics of the Model

### _Perception and Perceptual Units_

This model does not specifically provide for perceptual processes. Instead, it assumes that only the outcome of perceptual processing is important in understanding. This assumption also characterizes the Hayes and Simon model as well as many other cognitive models; it needs little, additional defense here. What is more worthy of comment is the size of units that make up a single perception. In this model, perceptions are always of a line or several lines at a time. If each symbol in a FORTRAN statement is considered equivalent to a word, then perception in this model occurs as a whole sentence or several sentences at a time.

No claim is made that this size unit is in any way atomic or primitive; indeed, it is willingly conceeded that there are perceptual units and processes operating at much lower levels, including those for individual symbols and parts of symbols. What is asserted, however, is that the larger units are the ones which are important in understanding programs. The main argument in favor of this assertion is that the larger units are the only ones that are mentioned in this protocol or in other, similar protocols. As an example, lines 3 to 16 of this protocol are claimed to be one perceptual unit because, in scanning these lines, the subject makes only one comment on the entire section. Additionally, when he needs to use information from individual lines in this range, he reads the lines again from the listing. On the basis of this argument, the claim is made that the size of perceptual unit used is sufficient for modeling program understanding behavior.

## Memory Capacity

The database for the production system is called the STM, a commonly used abbreviation for "short-term memory." Since this STM differs in two significant respects from the characteristics conventionally associated with a short-term memory, a word of explanation is in order. In contrast to the conventional short-term memory, the STM here is much larger in capacity than the traditional 7-9 chunks (Miller, 1956), and the contents are not lost over time. Both of these differences are related. Specifying a fixed capacity for the STM requires establishment of a standard for what constitutes a "chunk" of memory. Since subjects performing problem solving tasks tend to adopt strategies which keep memory requirements within manageable sizes (Newell & Simon, 1972), it is difficult to infer from the protocols the amount of information that forms one chunk for the subject. Similarly, it is difficult to tell when information has been overwritten or "pushed off the end" of the short-term memory. Rather than arbitrarily try to force the information in STM into some fixed number of chunks (of arbitrary size), the decision was made to put no bounds on

STM size, but, instead, to let it grow as needed. While this does mean that the STM in the model will contain more information than would be present in the short-term memory of the human subject, the effect on the overall model should be minimized since the model is constructed under the assumption that immediate context is most important in determining behavior. Thus, since the additional information will appear towards the end of STM, it should play a minimal role.

<u>Specificity</u> <u>of</u> <u>Productions</u>

In comparision of other problem-solving systems which have rather general purpose routines such as "MATCH-INPUT" ( ), the productions in this model are quite specific; for example, one of the productions has as its working conditions the following pattern:

```
1. SEE SEGMENT *ANY* *ANY* ( READ *ANY* *ANY*
     ( IMPLICIT-DO *ANY*=V3 *ANY* *ANY* 1 *ANY* ))
2. PART *ANY* *ANY* *ANY* *ANY* ( PARTS *ANY* )
3. OBJECT $V3 *ANY* *ANY* ( TYPE *ANY* ( ARRAY *ANY*)
 *REST*)
```

The "=" sign in the first pattern means that whatever matched the *ANY* token is to be assigned to the variable, V3. The "$" sign in the third pattern means that the value of V3, not the symbol, V3, is to be used at that point in the match. Paraphrased, the conditions can be written as:

1. A perception of a READ statement with an implicit DO operation. 2. A PART with subparts. 3. An object of TYPE, array, which is the same as the variable used in the implicit DO construction.

At the present time, the model contains some 29 productions which attempt to account for behavior in the first 56 of 384 utterances. At this rate, 150-250 productions will be required to model just one protocol. Since the model was created using the protocol, the possibility is raised that the productions have too much of an ad hoc character, and that a better though out, more general system might require fewer productions.

While this model is quite crude and could certainly be improved upon, an argument can be made that any recasting will not significantly reduce the total number of productions without a compensatory increase in the size and complexity of each production. The starting point for the argument is that the subject does display particular behaviors; he infers the specific purpose for which a specific variable is being used at a specific location. To achieve generality (or the appearence of generality), the common practice in model building is to partition the information involved in a specific behavior into a set of general mechanisms and a collection of specific data on which the mechanisms operate. Then, it is possible for the

[illegible]

D-9027
5-10

same mechanisms to operate on several sets of data or for the same data to be operated on by the same mechanisms. Since the total amount of information involved is far smaller than if each mechanism -data combination were treated separately, a measure of parsimony is achieved.

The drawback to such a partition is that the separation into procedure and data is usually an artificial one, a point well made by users of systems such as PLANNER (Hewitt, 1970; Winograd, 1972). The production rules used here do not distinguish between data and program, but, instead, achieve parsimony through the use of patterns in their invoking conditions. Nevertheless, they must still produce specific behaviors, involving specific information. The more specific information the program must model, the more specific information it must contain. This information can be incorporated into the system either by increasing the complexity of the patterns used by including more alternatives, binding more local variables, etc., or it can be accomplished by increasing the number of productions. Regardless of which route is followed, however, the net size of the system must grow at a rate which will be at least linear as it asked to account for more behavior.

## Production Systems and Input Driven Processes

A number of general considerations have been put forth in favor of the production system as a formalism for expressing models of human behavior (Newell & Simon,1972), and these considerations alone might well lead to the adoption of a production system structure for this particular model. Of interest, however, is whether there is some characteristic of this behavior under study which is particularly suited for a production system representation. The one that comes closest to filling this requirement is the role that external perceptions play in the model.

External perceptions are the "input" to the model. In a normal program, the interpretation given to a piece of input is almost entirely dependent on the variable or data structure to which it is assigned; for example, the interpretation of the input, 123, to a FORTRAN program will depend on whether the variable it is being read into is an employee i.d. or the X coordinate of a point to be plotted. In the model, on the other hand, the inputs, the perceptions, are self-identifying; what they mean is clear without reference to a variable or data structure.

Processing of input of this kind is most reasonably handled by a pattern matching approach (as versus, say, a succession of tests and branches), since the set of inputs is so large. Moreover, the range over which the match must be made will usually be this entire range of inputs. Such as system will, therefore, involve attempting to match the input against against a large set of patterns, and taking different actions depending of which pattern was matched. Whether such an organization is implemented as an augmented transition net, a PLANNER-like language, or in a special producton system language, the resultant structure will, therefore, still resemble a set of independent condition-action rules.

## Incomplete Work

As has been mentioned previously, the model currently accounts for behavior in a small segment of protocol. After enough additional production rules are added so that the model accounts for a larger segment of protocol, it will be possible to evaluate the "fit" between model and protocol. This evaluation will be accomplished in two parts. First, judges will be asked to rate at selected points the extent to which the knowledge about the program in the protocol overlaps the knowledge displayed by the model. Second, the judges will be asked to estimate the extent to which knowledge generated from hypotheses within the model could have been directly deduced from the surrounding context of that same knowledge in the protocol, thus giving an indication of the power of hypotheses in the model.

The demonstration of a model-protocol fit, while an important step, only validates the model for the behavior of a single subject on a single task. Further work, involving other subjects working at other tasks, will be mandatory to demonstrate the model´s wider generality. Of necessity, the specific productions will have to be altered for each subject. A few of the characteristics of the model, however, appear to be applicable to a wide range of subjects and tasks. These include:

1. The size and level of perceptual units.

2. The generation of hypotheses about program structure and their use in understanding the program.

3. The condition- action structure for processing input.

Future work will be directed at assessing the extent to which these features are, indeed, universally integral to models of the program understanding task.

## Bibliography

Brooks, R. Experiments with a production system language. Technical report. Department of Information and Computer Science. University of California at Irvine. Irvine, Ca. 1976.

Hayes, J.R. & Simon, H.A. Understanding written problem instructions. in Gregg, L.W. (Ed.) Knowledge and Cognition. Lawrence Erlbaum Associates. Potomac, Maryland. 1974.

Hewitt, C. Description and theorectical analysis (Using Schemata of PLANNER: A language for proving theorems and manipulating models in a robot. Unpublished doctorcal dissertation. Department of Mathematics. Massachusetts Institute of Technology. 1972.

Miller, G.A. The magical number seven, plus or minus two: Some limits on our capacity for processing information. Psychological Review 1956. 63 81-97.

Newell, A. & McDermott, J. PSG Manual. Department of Computer Science. Carnegie-Mellon University. September, 1975

Newell, A. & Simon, H.A. Human problem solving. Prentice-Hall. New York. 1972.

Simon, H.A. and Hayes, J.R. Understanding complex task instructons. in D. Klahr (Ed.) Cognition and instruction. Lawrence Erlbaum Associates: Ptomac, Maryland. 1976.

Sussman, G.J& McDermott, D.V. Why CONNIVING is better than PLANNING. (AI Memo No. 255A) Artificial Intelligence Laboratory. Massachusetts Institute of Technology. 1972.

Winograd, T. Understanding Natural Language. Cognitive Psychology 1972. 3 1-191.

```
1. SEE SEGMENT 18 19
       (LABEL 15) SUBROUTINE-CALL
                   (RANSUB RANS IPT)
                   ASSIGNMENT INUM (RANS IPT)
2. PART 1
                  PROGRAM
                  FUNCTION RANDOM-WALK OF THING 3 ON GRID 2
                  LOCATION 1 ?
                  PARTS
            INTEGER-DECLARATIONS 1 2
                INITIALIZATIONS 3 16
3. PART 5 INITIALIZATIONS
          FUNCTION INITIALIZATION
          LOCATION 3 16
          ?
4.  LOOK
                  .
                  .
                  .
                  .
                  .
```

Production SEE-3 fires off.

```
1. PART 1
                  PROGRAM
                  FUNCTION RANDOM-WALK OF THING 3 ON GRID 2
                  LOCATION 1 ?
                  PARTS
               INTEGER-DECLARATIONS 1 2
               INITIALIZATIONS 3 16
               SUBROUTINE-CALL 18 19
2. PART 6 SUBROUTINE-CALL
          FUNCTION SUBROUTINE-CALL RANSUB
          LOCATION 18 19
          ?
1Ø. HYPOTHESIZE
              PART
              ?
              ?
              FUNCTION RANDOM-NUMBER-GENERATOR
              ?
              ?
                  .
                  .
                  .
                  .
```

At this point, CONCLUDE-1 is fired off.

```
1. LOOK-FOR SUBROUTINE RANDOM-NUMBER-GENERATOR
2. PART 1
                  PROGRAM
```

```
                FUNCTION RANDOM-WALK OF THING 3 ON GRID 2
                LOCATION 1 ?
                PARTS
              INTEGER-DECLARATIONS 1 2
              INITIALIZATIONS 3 16
              SUBROUTINE-CALL 18 19
3. PART 6 SUBROUTINE-CALL
          FUNCTION SUBROUTINE-CALL RANSUB
          LOCATION 18 19
          ?
                .
                .
                .
                .
```

Figure 1

Operation of Perceptual Mechanisms

The figure shows the partial contents of STM before and after the firing of several productions. The "?" is used as a place holder to indicate information not yet known to the model. Note the HYPOTHESIZE element; this serves as part of the invoking conditions for CONCLUDE-1 which places the LOOK-FOR element into STM.

```
1. SEE SEGMENT 85 85
       DECLARATION INTEGER RANBUF 1Ø PPTR
2. PART 7 SUBROUTINE
           FUNCTION RANDOM-NUMBER-GENERATOR
                    GENERATES NUMBERS RANDOM
                    LOCATION 84  ?
                    ?
3. PART 1
         PROGRAM
         FUNCTION RANDOM-WALK OF THING 3 ON GRID 2
         LOCATION 1 ?
         PARTS
                INTEGER-DECLARATIONS 1 2
                INITIALIZATIONS 3 16
                SUBROUTINE-CALL 18 19
        .
        .
        .
        .

Production SEE-5 fires off.

1. PART 7 SUBROUTINE
           FUNCTION RANDOM-NUMBER-GENERATOR
                    GENERATES NUMBERS RANDOM
           LOCATION 84  ?
           ?
2. OBJECT RANBUF 9 (SUBROUTINE 7)
           TYPE INTEGER ARRAY 1Ø
           MEANING BUFFER
3. OBJECT PPTR 8 (SUBROUTINE 7)
           TYPE POINTER
           MEANING BUFFER-POINTER
                  .
                  .
                  .
```

Figure 2a.

Context-dependent Information Extraction

Note that the two new OBJECTs that are created by SEE-5 belong to the subroutine, not to the main program.

```
1. SEE SEGMENT 3 3
      ASSIGNMENT NS 500
2. PART 1
                 PROGRAM
                 FUNCTION RANDOM-WALK OF THING 3 ON GRID 2
                 LOCATION 1 ?
                 PARTS
          INTEGER-DECLARATIONS 1 2
          INITIALIZATIONS 3 16
          SUBROUTINE-CALL 18 19
                .
                .
                .
```

The producion, SEE-12, is fired.

```
1. PART 5 INITIALIZATIONS
          FUNCTION INITIALIZATION
          LOCATION 3 16
          PARTS ASSIGNMENT 3 3
2. OBJECT NS 15 (INITIALIZATIONS 5)
                VALUE ?
3. PART 14 ASSIGNMENT
           FUNCTION ASSIGNMENT
             ASSIGN NS ?
           LOCATION 3 3
           ?
                .
                .
                .
```

Figure 2b.

Context-dependent Information Incorporation

The SEE-12 production recognizes the assignment statement as being part of some initializations. In creating the NS object, therefore, only the fact of the assignment, not the specific value, is retained.