NETWORK: A Computer Simulation of the Construction-Integration Model[1]

Program Documentation

Suzanne Mannes
Department of Psychology
University of Delaware

Michele Roushey
Institute of Cognitive Science
University of Colorado at Boulder

ICS Technical Report #90-13

---

[11]This research is supported in part by contract IRI-8722792 from the National Science Foundation and by contract ARIMDA903-86-C0143 from the Army Research Institute and is based upon the dissertation of the first author. We would like to thank Gerhard Fischer, Walter Kintsch, Clayton Lewis and Peter Polson for their invaluable contributions to this research. Without their support the NETWORK simulation could not exist as a useable tool for others. We would also like to acknowledge Stephanie Doane for her input on the automatization of the NETWORK simulation and on the structure of this document. Her repeated editing and advising were very useful, and without her input the simulation would not be what it is today. Correspondence should be sent to Michele Roushey, Institute of Cognitive Science, University of Colorado, Campus Box 346, Boulder, CO 80309-0346.

Table of Contents

1 Overview of Documentation 4
1.1 Purpose of Document 4
1.2 Structure of Document 5
1.3 Structure of the Sections 6

2 Overview of Model 8
2.1 Purpose of NETWORK 8
2.2 Description of Model 9
2.2.1 Knowledge Representation 9
2.2.2 Long-term Memory Matrix Construction and Knowledge Activation 10
2.2.2.1 Relationships between nodes in LTM 11
2.2.2.2 Activation of world knowledge 13
2.2.3 Construction and Integration as a Means of Task Solution 13
2.2.3.1 Construction phase 14
2.2.3.2 Integration phase 14
2.3 Running Example 15

3 Construction and Knowledge Activation in Long-term Memory 18
3.1 Construction 18
3.1.1 Input 18
3.1.1.1 Input files 18
3.1.1.2 Keyboard input 22
3.1.2 Process 24
3.1.3 Output 25
3.2 Selection of Free Associates 25
3.2.1 Input 25
3.2.1.1 Input files 25
3.2.1.2 Keyboard input 26
3.2.2 Process 27
3.2.3 Output 27

4 Binding Variables in Plan-elements 31
4.1 Input 31
4.1.1 Input File 31
4.1.2 Keyboard Input 31
4.1.3 Common Errors in Plan-element Binding 34
4.2 Process 34
4.3 Output (bound plan-element file) 34

5 Constructing a Task Connectivity Matrix 36
5.1 Input 37
5.1.1 Input Files 37
5.1.1.1 Proposition input file 37
5.1.1.2 Plan-element file 39
5.1.2 Keyboard Input 40
5.2 Process 42
5.3 Output 42

6 Integration - Calculating Spreading Activation 45
6.1 Input 45
6.1.1 File Input 46
6.1.1.1 Sample vector file *include1.vec* 46
6.1.1.2 The matrix file 46
6.1.2 Keyboard Input 46
6.2 Process 47
6.3 Output 48
6.4 Decision About a Plan-element for Action 49
6.4.1 Input 49
6.4.2 Output Files 50

7 References 52

1 Overview of Documentation

1.1 Purpose of Document

The construction-integration theoretical framework (Kintsch, 1988) was originally developed to account for various phenomena in the area of text comprehension. In the same paper, Kintsch derived a model of the processes involved in arithmetic word problem solving as a particular instance of text comprehension. The framework was extended by Mannes (1989) and Mannes and Kintsch (submitted) to describe expert performance on various types of routine computing tasks that involve the combination of well known commands and functions. They developed a model of formulating and executing multi-step plans. Finally, Doane, Kintsch, and Polson (submitted) have extended this model of planning to deal with production of nonroutine, complex commands for the UNIX operating system. In addition, they characterize differences between novice, intermediate, and expert users in terms of the processes set forth in the construction-integration framework. The reader of this document, and anticipated user of NETWORK, is strongly encouraged to familiarize himself with these papers before attempting to apply the framework in his or her research program.

This document is intended to be a user manual for the computer program NETWORK.[2] The original version of this program was written by Suzanne Mannes, (with the assistance of Michele Roushey) as a computer simulation of the construction-integration model. The program was later modified to automate several steps that had previously required manual modification to data files. It is hoped that a new user of the NETWORK program will be able to use this document as an aid to developing and running their own simulation with the construction-integration framework. Some individual instruction and additional familiarity with the construction-integration model will be necessary before this can be accomplished.[3]

NETWORK was originally written in Interlisp and ran only on a Xerox 1108 dedicated lisp computer. Because of the limited use and access to Interlisp, the program was translated to Common Lisp and development was continued on a Hewlett Packard Series 9000 Model 350 computer. At the time this document was written, the NETWORK program had been kept as close to generic Common Lisp as

---

[2] In this document, NETWORK in all capital letters always refers to the simulation program. Network, written as it is usually expected, refers to semantic or associative networks as they are traditionally understood.

[3] In the event that information in addition to that contained in this document is required by the reader, the following people may be contacted. For programming and documentation questions please contact Michele Roushey, her address appears on the title page. For theoretical questions, please contact either Dr. Stephanie Doane, Department of Psychology, University of Colorado, Boulder, CO, 80309-0345 or Dr. Suzanne Mannes, Department of Psychology, University of Delaware, Newark, DE, 19716.

possible so that it can be easily ported to other computer systems. Future changes to NETWORK will include a menu driven interface and some matrix visualization tools. At this point the user interface to NETWORK is primitive at best. It is hoped that this document will help new users deal with this primitive interface.

1.2 Structure of Document

This document is divided into six sections:

Section 1. Overview of Documentation
This section which you are now reading explains the structure of rest of the documentation.

Section 2. Overview of Construction-Integration Model
This section gives a general description of the construction-integration model and its various components. Terms are defined and a running example is introduced. The flow diagram on page 7 gives a general view of the overall flow of the simulation.

Section 3. Construction and Knowledge Activation in Long Term Memory
This section describes how to run the NETWORK simulation to construct a long-term memory (LTM) matrix, the first step in solving a task. Selection of related general knowledge is also explained in this section. The flow diagram on page 18 depicts the program flow for LTM matrix construction.

Section 4. Binding Variables in Plan-elements
This section describes how to bind variables in plan-elements. Sample unbound and bound plan-element files are included. The flow diagram on page 31 describes the flow of the variable binding module.

Section 5. Constructing a Task Connectivity Matrix
This section describes how to run the NETWORK simulation to construct a task connectivity matrix. Hard coding of request relationships is also described here. The flow Diagram on page 36 describes in detail the program flow for construction of a task connectivity matrix.

Section 6. Integration - Calculating Spreading Activation
This section describes how matrix multiplication is used to accomplish the spread of activation through the task connectivity matrix described in Section 5. Also covered in this section is the decision process which selects the most active plan-element that can fire. Program flow for this section is described in the flow diagram on page 46.

Each section is divided into sub-sections which are described below. A program flow diagram is included for Sections 2 through 6. These diagrams were designed as a user aid to understanding NETWORK's overall program flow. A running example is introduced with sample input and output files. The example used is a version of one used in Suzanne Mannes' Doctoral Dissertation, Problem Solving as Text Comprehension: A Unitary Approach. If a new user of NETWORK

requires more information about the construction-integration model additional information can be obtained from the first four chapters of that dissertation.

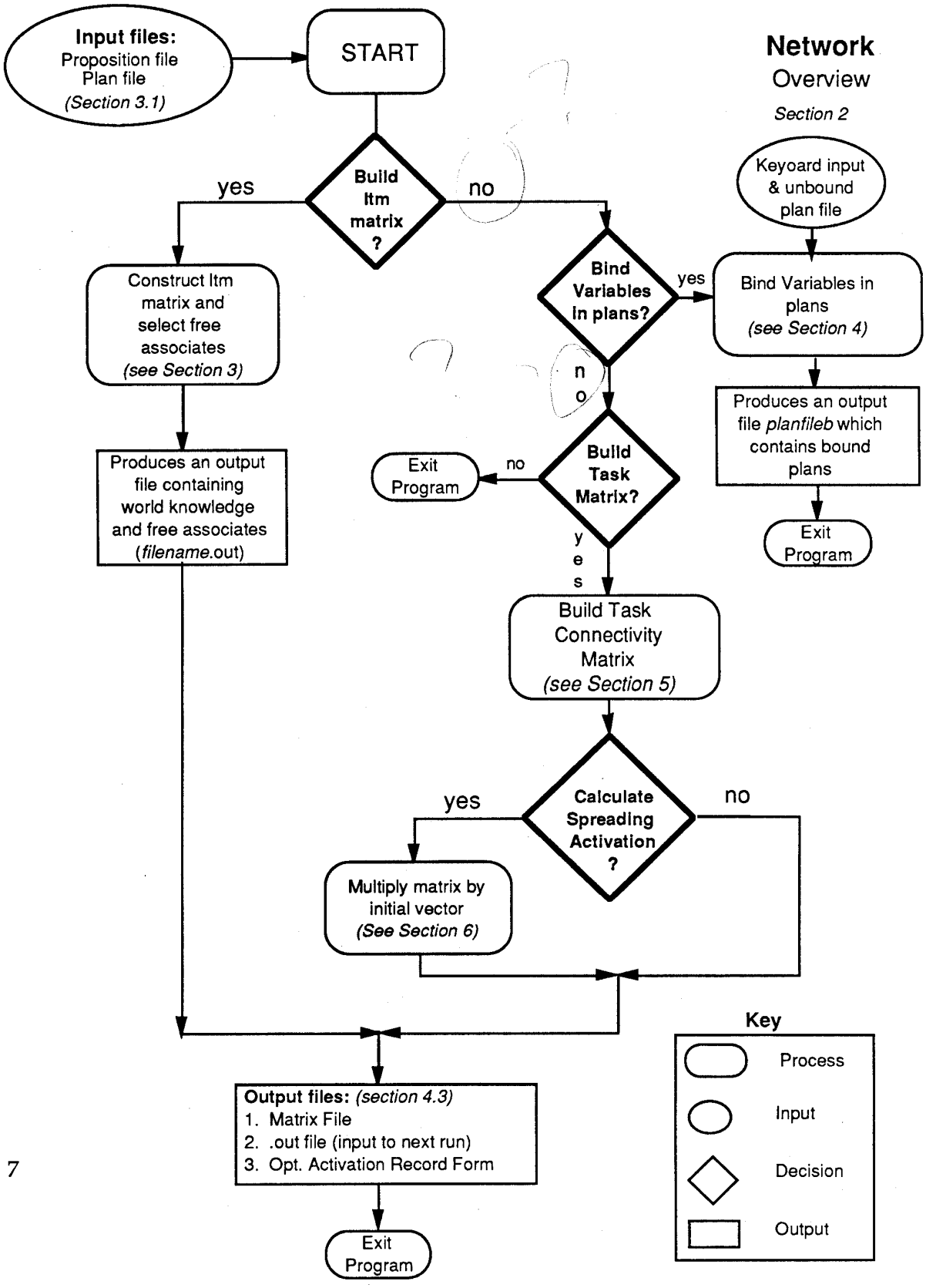
The NETWORK program is actually divided into four distinct modules: building and activating knowledge in a long-term memory (LTM) matrix, binding variables in plan-elements, constructing a task connectivity matrix, and integrating that matrix. Binding plan-element variables and calculating spreading activation can only be accomplished when a task connectivity matrix is constructed. Free associates can only be selected when a long-term memory (LTM) matrix is built. Every attempt has been made to follow a consistent pattern of presentation in this documentation.

1.3 Structure of the Sections

There are two distinct types of sections in this document: overview sections and module description sections. Sections 1 and 2 are overview sections and are intended to provide general information about NETWORK. The general flow of the simulation, as described in Section 2, is depicted in the accompanying flow diagram on page 7. Sections 3-6 are module description sections and are designed to be a direct instructional aid to running NETWORK. Each of these sections also has a flow diagram specific to the program flow for that section.

Sections 3-6 provide specific information on running the NETWORK simulation. They are structured as followed:

1. Flow diagram for section
2. General description of module
3. Input required (files and keyboard, with a running example)
4. Short description of calculations/processes
5. Common errors (and how to avoid them)
6. Program output (with running example)


Input files:
Proposition file
Plan file
(Section 3.1)
START
Network
Overview
Section 2
Build
ltm
matrix
?
yes
no
Construct ltm
matrix and
select free
associates
(see Section 3)
Produces an output
file containing
world knowledge
and free associates
(filename.out)
Bind
Variables
in plans?
yes
Keyoard input
& unbound
plan file
Bind Variables in
plans
(see Section 4)
Produces an output
file planfileb which
contains bound
plans
Exit
Program
n
o
Exit
Program
no
Build
Task
Matrix?
y
e
s
Build Task
Connectivity
Matrix
(see Section 5)
Calculate
Spreading
Activation
?
yes
no
Multiply matrix by
initial vector
(See Section 6)
Output files: (section 4.3)
1. Matrix File
2. .out file (input to next run)
3. Opt. Activation Record Form
Exit
Program
Key
Process
Input
Decision
Output

2 Overview of Model

2.1 Purpose of NETWORK

One of the goals of psychology has been to come up with a unitary theory of cognition. If we are to understand human thinking as a whole rather than as a culmination of many discrete processes, one model which describes many behaviors is essential. The theory of text comprehension developed by van Dijk and Kintsch (1983) and Kintsch's (1988) construction-integration model provide a nice starting point. The construction-integration model shows how several cognitive tasks which appear to use top-down processing, and which have been viewed as top-down in the past (e.g., the word superiority effect in letter recognition, Wheeler, 1970; the sentence superiority effect for words, Tulving & Gold, 1963; and the skilled reading model of Perfetti, 1985), can be modeled in a bottom-up manner. That is, it shows that the postulation of large pre-compiled structures in memory is not necessary to account for many experimental findings. The construction-integration model provides a more parsimonious description of experimental data by building these structures on-line.

In this framework, knowledge about the particular domain to be modeled is represented as an associative network. The nodes in this network stand for some domain specified units and the links for the degree of relationship between all of the possible pairs of items in the net. These relationships can be positive, negative, or zero when two items are not at all related. The different ways in which the nodes become interconnected initially, comprises the construction phase of the model. When the knowledge net is activated by some outside source such as a problem to be solved or a word to be understood, it activates nodes which are related to it and the spread of activation is begun. The representation of the outside source reverberates its action throughout the network until it settles into a stable state. This comprises the integration phase of the model.

Looking at phenomena from apparently different areas of cognition within this paradigm of a cyclic construction-integration process has proven quite useful in unveiling the similarities these areas share. It has been used to model at least four different domains so far; word sense disambiguation, recognition of sentences from text, word arithmetic problem solving, and routine and complex computing tasks. Phenomena at the word, sentence and short text level have been addressed and humans performing several types of cognitive tasks have been simulated. Thus, the model begins to approach the unitary theory of cognition for which we are searching.

Kintsch (1988) has described a model in which representations are derived in an essentially bottom-up manner. Rather than postulating powerful and sophisticated rules for constructing a representation of the world, he has put forth a scheme whereby the initial construction of this representation is sloppy, perhaps including information about the world which is wrong or pieces of evidence which are contradictory. It is then, by using world knowledge, that our cognitive processes determine what parts of the representation are coherent and which parts don't fit. Hence the name of the model is telling with respect to its nature. First we construct a representation and then that representation is integrated.

2.2 Description of the Model

The NETWORK simulation takes as input a list of propositions, representing either a textual passage or some knowledge about a particular domain, and constructs from it an associative network based upon several different types of relations which the propositions in the list share. The simulation then uses the network in an integration process, and settles into a state which minimizes the activation flow between elements from cycle to cycle. This process results in a network where concepts which are related to one another have activated each other and concepts which are inconsistent have become deactivated. The final state of the network can be interpreted as a reader's final understanding of the meaning of a discourse, or a subject's understanding of a task to be completed using information from some fairly well circumscribed domain. The flow of the simulation can be followed by examining the flow diagram on pg 7.

The NETWORK simulation has four main program modules; constructing and knowledge activation in LTM, binding variables in plan-elements, task connectivity matrix construction, and integration and decision. Construction and activation of knowledge from a LTM matrix is usually done only once per task solution. However, it normally takes several iterations through both task connectivity matrix construction and integration before a task solution is achieved.

2.2.1 Knowledge Representation

In order for a task to be represented by the model it must parsed and propositionalized. This is usually based on some linguistic input (though the input need not necessarily be linguistic). It has been proposed that the unit most appropriate for describing human processing in a text related task is the **proposition** (Clark & Clark, 1977; van Dijk & Kintsch, 1983; Kintsch, 1974; Kintsch & Keenan, 1973; Norman & Rumelhart, 1975).

A proposition for NETWORK can be defined as:

(P*n*(predicate argument1 argument2 argument3 ... argument*j*))

where *n* is a unique proposition number preceded by a capital P and each predicate and argument is a valid lisp symbol. Long predicates or arguments may use a -, a . or an _ as delimiters for readability as in propositions P2-P4 below.

```
(P1(EXIST ADDRESS))
(P2(AT-LEVEL SYSTEM))
(P3(AT.LEVEL EDIT))
(P4(AT_LEVEL MAIL MAIL_MESSAGE))
```

In this simulation of the construction-integration model, nodes represent three non-exclusive types of knowledge; knowledge about the current state of the world (world knowledge), general knowledge, and knowledge about certain plan-

elements which can be combined to produce plans of action. These are all represented in a propositional form.

World knowledge has a standard propositional format and initially represents the task instructions. **General knowledge** is also represented propositionally and contains information about computers, tasks, and related things which are experienced in the computer domain. The representation of the **plan-elements** is slightly more complex. They have four fields to them, a number, a name, preconditions, and outcomes. For example, the plan-element to find a file has the following form, where

```
the plan-element number is:                (L7
the plan-element name is:                  ((FIND FILE))
the plan-element preconditions are:  ((EXIST FILE)
                                           (AT-LEVEL SYSTEM))
and the plan-element outcome is:           ((KNOW FILE LOCATION)))
```

A task's **requests** and anticipated **outcomes** are also part of world knowledge. Each task is defined by one or more requests which, when satisfied, result in the carrying out of a corresponding outcome. When all requests for a task have been satisfied and hence all outcomes completed, the simulation has solved the task. Below is an example request/outcome set that would be used by the simulation to solve a task involving sending a paragraph of text to someone.

```
(P4(RQSEND ?TEXT^PARAGRAPH))
(P5(OUTC4RECEIVE ?TEXT^PARAGRAPH))[4]
```

2.2.2 Long-term Memory Matrix Construction and Knowledge Activation

The model starts out with a LTM network of only two of these three proposition types, general knowledge and plan-element knowledge. These are associated on the basis of relations like argument overlap, and propositional embedding. (Current state of the world information comes into play only after the system has been presented with a task to be solved.) This initial network represents the system's LTM and remains the same for all task solutions. When a specific problem description is encountered, NETWORK creates a current state of the world proposition list of this description, and some of the general knowledge in LTM (which is related to the concepts mentioned in the problem) becomes activated.

Long-term memory is assumed to have its effect when a problem is encountered and not at any later time. This is analogous to what has been viewed as a person's use of general knowledge in understanding a problem she has been asked to solve (Kintsch, 1988). In the model, this is a crucial step in the solution process. Only the knowledge which is activated at this time can serve to aid in a solution

---

[4]The meaning and source of the ? and ^ is presented in Section 4. They are used to identify variables.

attempt. This means that task solution depends not only upon whether a user has a particular piece of knowledge, but also on whether or not that knowledge is activated during a solution attempt.

2.2.2.1 Relationships between nodes in LTM

When either a LTM matrix (described in Section 3) or a task connectivity matrix (described in Section 5) is constructed certain relationships between pairs of propositions must be established. Nodes representing knowledge are joined with links representing the relations between the nodes.

There are five types of relationships that are used to construct connections between the nodes in the network.[5] These relationships are described in the six cases below:

1. Argument overlap. One method of associating ideas which are represented propositionally is by argument overlap and is based directly on a principle from the Kintsch and van Dijk (1978) model of text comprehension. This principle states that propositions sharing arguments ought to be related. For example, the propositions

(P6(USE MAIL STUDENTS)) and
(P7(IN MESSAGE MAIL))

share a reference to the concept MAIL and this relationship should be reflected in the network of which they are a part. The more general question of how to approximate this relationship is left unanswered here. A numerical value, which is relative in degree to the other types of association, is assigned by the user to represent the strength of the link between the nodes. As with most computer programs, in order for any of the matching routines to work, the match must be *exact*.

2. Propositional embedding. Another type of relationship derives from the nature of the propositional representation and has to do with propositional embedding. Often, a proposition contains another proposition number as an argument. That is, one proposition is entirely contained within another. This relationship is apparent in the following pair of propositions.

(P8(REQUEST MESSAGE P9))
(P9(SEND YOU FILE))

This is a fairly direct relationship and the numerical value assigned to pairs of propositions which are embedded within each other is another parameter which may be specified by the user.

---

[5]Because the network is represented as a matrix for the simulation, network and matrix are used interchangeably.

3. Associated knowledge. Concepts which are associates of each other receive a link indicating this. Concepts such as

(P10(QUALITY LETTER FRIENDLY *P11))
(P11(SEND MAIL POSTOFFICE))

are clearly related but this association would not be found based on the previously mentioned relation types.[6] A value may also be assigned to this relationship.

4. Interplan relations. The plan-elements are related by their causal relations, with plan-elements which provide as part of their outcomes pre-conditions for other plan-elements receiving activation from those plan-elements (4a). In general, a plan-element, for example FIND FILE, which produces as an outcome a proposition, KNOW FILE LOCATION, which is a precondition for a plan-element, DELETE FILE, will receive activation from DELETE FILE, producing a type of causal chain among plan-elements. Likewise, a plan-element, DELETE FILE, which destroys a precondition for another plan-element, SEND FILE MAIL, is inhibited by the SEND plan-element (4b). Values may be specified by the user for both the causal chain and inhibitory relationships.[7]

5. Inhibition. There may exist propositions which are inconsistent or contradictory to each other. In order to allow for this relationship which in many circumstances would require semantic knowledge which NETWORK does not have, the user has available the use of a special purpose code like that just mentioned for coding associated knowledge. This special relation is signified with the $ character. The user will be prompted for a numerical value that will be applied to the association when this symbol is encountered in a proposition. For example, given the following propositions

(P12(ISA TEXT SHORT))
(P13(ISA TEXT LONG)),

we can show that these propositions are contradictory by modifying them as follows.

---

[6]In the development of this model, a study was done in which subjects were shown many phrases one at a time and asked to produce the first thing that came to their minds. Then their answers were used as examples of "associated" knowledge in constructing a LTM for NETWORK. Because NETWORK has no semantic knowledge, this relationship had to be hard coded into the proposition list that represented the contents of LTM. The * which occurs prior to the proposition number denotes the proposition to which that number refers as an associate of the one in which it is contained. This * symbol may be used by the user to denote any type of positive symmetric relationship they wish. They will be prompted for a value for the strength of relationship between propositions sharing this "associative" relationship whenever a matrix is created by NETWORK.

[7]Note that this is the only type of relationship computed between plan-element pairs. They are not linked on the basis of argument overlap, etc.

(P12(ISA TEXT SHORT $P13))
(P13(ISA TEXT LONG)).

6. Null or no relations. The user may also use the @ to signify no relationship. That is, when propositions appear to be related, but are not, and the user wants to be sure that they will not be linked.[8]

2.2.2.2 Activation of world knowledge

Once a LTM has been created to show these types of relationships, a task may be attempted. Before any actual task connectivity construction is achieved however, the propositions representing the task to be solved have the effect of activating world knowledge in LTM to which they are related. Thus, in parsing a sentence about eating ice cream, concepts of COLDNESS, SWEETNESS, and SUMMERTIME might become active. Free associate propositions are activated and added to the task description as are all of the plan-elements NETWORK knows about which can be used in a solution. This activation is hypothesized to be automatic and it is a large part of the engine which drives the model. After the activation of **associated knowledge**, relationships between the constituent propositions are determined when a task connectivity matrix is built (described in detail in Section 5).

At this time LTM has played its role by producing related knowledge to help in the solution attempt. Information which is not currently active in LTM is not considered in the task solution.

2.2.3 Construction and Integration as a Means of Task Solution

The actual task solution is accomplished in two distinct phases; the **construction phase** and the **integration phase**. Several iterations through each of these phases is usually required before a task solution is complete. The output of each iteration becomes the input to the next iteration.

Once some set of knowledge residing in LTM has been activated in response to a task description, another network, separate from LTM, is constructed for the solution of that task. This **task connectivity matrix** (network) contains nodes representing all three types of propositions; the task description or current state of the world, the activated general knowledge, and the plan-elements with links representing the relationships between them. This newly constructed task network is then integrated. Integration entails activation flowing throughout the network until a stable state is reached yielding activation values for all propositions. The

[8]It is not necessary to hard code this relationship in both propositions because this is considered to be a symmetric relationship and will be reflected as such in the network. Argument overlap, propositional embedding, inhibition, and association are also considered to be symmetric. Only the plan-element chaining is a non-symmetric relationship.

activation values for plan-elements are then used by a decision process to select a plan-element for action.

2.2.3.1 Construction phase

Variable binding. Before a specific task can be solved the concepts to which the task refers, and which are represented in LTM as variables must be bound to their specific values. For example, if knowledge of files in general is maintained in LTM (files are kept on disk for example), then to solve a task with a file called MANUSCRIPT that general knowledge, if active, must come to apply to this particular instance of a file. The particular contents or bindings of plan-elements also depends upon the task to be done. These plan-elements reside in LTM in a generic form and become activated and bound whenever a task description is encountered. If the task description contains multiple instances of texts or files, the plan-elements dealing with these automatically become duplicated so that there exists one plan-element for each text/file combination. This means that the total number of plan-element propositions will vary depending on the task to be done and might even change during the solution of a task if objects are created or destroyed.

Node relationships. The six cases of node relationship described in Section 2.2.2.1 are used to construct the task connectivity matrix or network. In addition, one further type of relationship may now exist between the current state of the world propositions and the plan-elements. This is described as case 7, plan-world inhibition.

7. Plan-world inhibition. If the outcomes of a plan-element all exist in the current state of the world already, the propositions representing these outcomes in the world list inhibit the plan-element. In this way, once the results of a plan-element exist, either because they were given as part of the task description or because a plan-element has been executed, it is discouraged from firing again. This is called plan-world inhibition.

Results of construction phase. The matrix constructed in this phase becomes input to the integration phase. The integration of the matrix produced by NETWORK starts the spread of activation from the current state of the world, and allows the pieces of the network which fit together to strengthen each other and the pieces which don't fit neatly into the network to lose their activation. This process settles on an interpretation of what should be done, what plan-element should be carried out, given the understanding of the instructions and the plan-elements available.

2.2.3.2 Integration phase

During the **integration phase**, connectionist-like methods are used to spread activation throughout the network (or matrix). The result of this is that items which are highly related strengthen (receive activation from) each other and those items

which are inconsistent or fit poorly into the representation as a whole, inhibit each other lessening their activation. When the system stabilizes and activation fails to spread further, a coherent representation of the original input has been obtained with the help of these relatively automatic processes and the use of knowledge about the world.

Vector-matrix multiplication. The spread of activation through the network is accomplished using a vector-matrix multiplication technique. **Vector multiplication** by the connectivity matrix is repeated until a stable, but arbitrary, user defined criterion has been reached. This activation-relaxation process is carried out by a vector-matrix multiplication module (originally written by Turner (1987) in the C programming language).

Carrying out a plan-element. After the network has settled, the values in the last output vector represent the degree of activation for each proposition. The values for the plan-elements are considered for a plan of action to be carried out. The plan-element with the highest activation value is selected for possible execution. If all of its preconditions, those propositions in a plan-element's third field, have matching propositions in the world proposition list, then it is allowed to fire, otherwise the next most highly activated one is considered and so on until a plan-element is found having all of its preconditions available in the world. Then the plan-element's outcomes, propositions in the fourth plan-element field, are added to the world knowledge and the process begins again at the post knowledge activation phase.

When a **plan-element is carried out** (fired) the outcomes of that plan-element are added to world knowledge. In order for a plan-element to fire all of that plan-element's preconditions must already exist in world knowledge.

Results of integration phase. After the first task connectivity matrix is integrated, its output (after some manual modification which is described in Section 5.1.1.1), including the outcome of the fired plan-element, becomes the input to the next iteration of the construction-integration cycle. This continues until the task is solved.

When is a task solved. Each iteration through task connectivity matrix construction and integration results in the selection of the most active plan-element with all of its preconditions met. This plan-element is then fired and its outcomes are added to world knowledge.

When the outcome(s) of a plan-element that has fired matches the OUTCOME specified by the task description, the task is considered solved.

2.3 Running Example

One of the tasks used to develop NETWORK was the following, "include an address that is known into a letter being written on the computer". This, the INCLUDE task, requires a short sequence of steps in order to be accomplished. Each

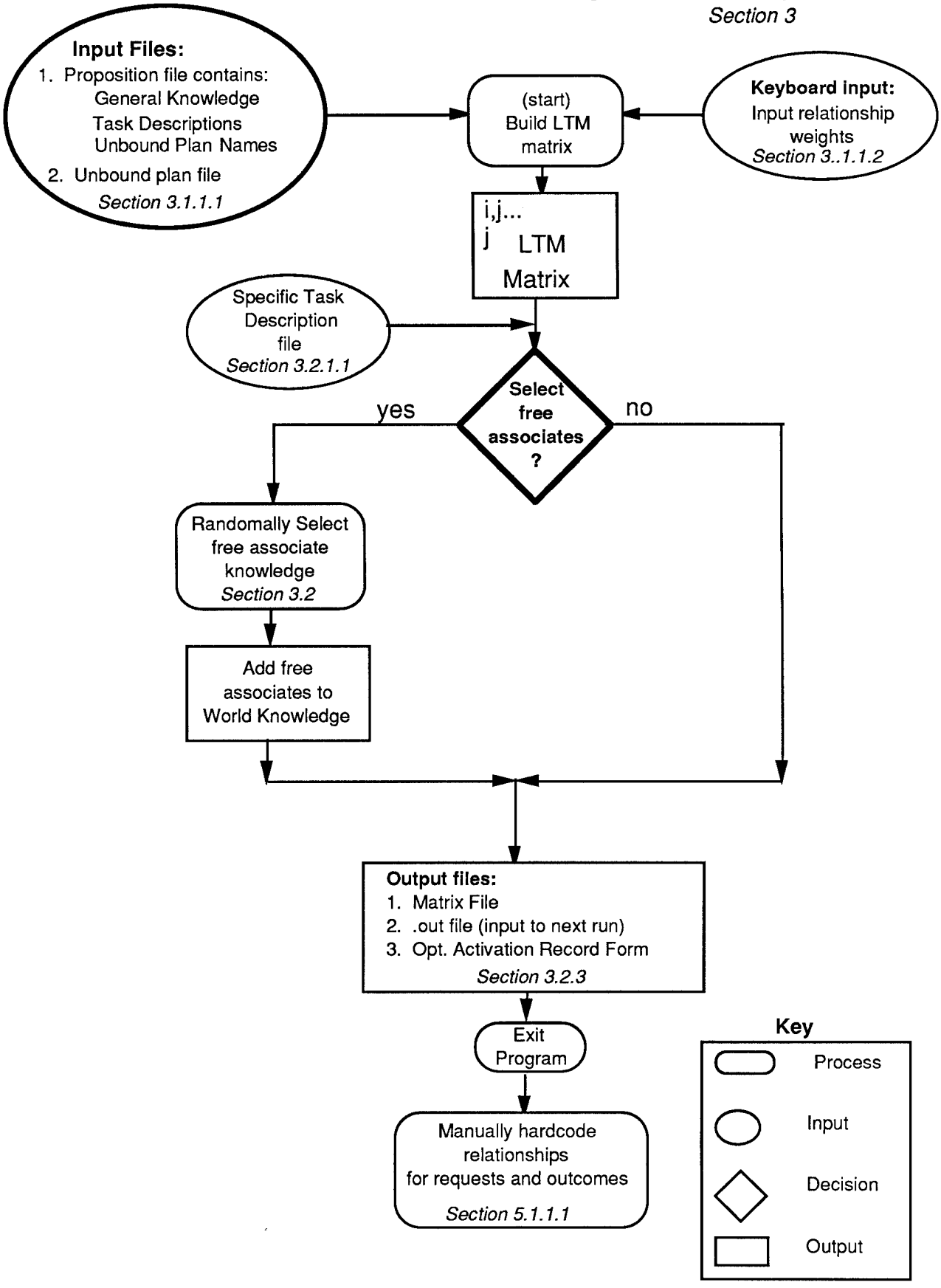
one of these steps requires a separate task connectivity construction-integration cycle. This task is represented in its propositionalized form below:

```
((P1(EXIST ADDRESS))
 (P2(EXIST LETTER))
 (P3(EXIST ?FILE^LETTER))
 (P4(EXIST ?TEXT^LETTER))
 (P5(EXIST ?TEXT^ADDRESS))
 (P6(AT-LEVEL SYSTEM ?FILE^LETTER))
 (P7(ISA ?TEXT^LETTER ?FILE^LETTER))
 (P8(RQINCLUDE ?TEXT^ADDRESS ?TEXT^LETTER))
 (P9(RQINCLUDE ?TEXT^ADDRESS ?FILE^LETTER))
 (P10(OUTC8IN ?TEXT^ADDRESS ?TEXT^FILE))
 (P11(OUTC9IN ?TEXT^ADDRESS ?FILE^LETTER))
 (P41(AT-LEVEL SYSTEM)))
```

A general description of the construction-integration theoretical framework and the NETWORK simulation has just been presented. The remainder of this paper describes in detail how to use NETWORK to carry out the four main processes; construction and knowledge activation in LTM, plan-element variable binding, task connectivity construction, and integration and decision.

Long-term Memory Matrix Build

Section 3



3 Construction and Knowledge Activation in Long-term Memory

In this section of the documentation you will be instructed on how to use the NETWORK simulation to construct a long-term memory (LTM) matrix. There is a complete description of the input files and keyboard input that are necessary to get NETWORK started on a solution to your own specific task, using the INCLUDE task as an example. You learn how NETWORK constructs the relationships between nodes in the associative network. You are instructed on the selection of free associate knowledge for the task and the output files from this first module of the program are described and discussed.

Before a task can be solved, a LTM matrix must be constructed so that it is available for knowledge activation whenever a specific problem description is encountered. NETWORK uses the task description, represented as a proposition list, to activate related information in LTM. This process is known as the selection of free associate knowledge. The LTM matrix is constructed only once for each task that is solved and is the first step in the solution process.

The task used in this example of LTM matrix construction is the one introduced in Section 2.3, "include an address that is known into a letter being written on the computer". This section includes all the information needed to build a LTM matrix for the example.

For the first, LTM construction phase, NETWORK takes as input a list of propositions containing general and plan-element knowledge and creates an associative network which represents the degree to which pairs of propositions share particular types of relationships (presented in Section 2.2.2.1) in LTM. These propositions represent a subset of the general knowledge a user has about the computer domain, and the network which is created represents the LTM or knowledge base for that domain (only a subset of knowledge is used here for ease of computation). Included in the network are potential plan-elements or operators for accomplishing tasks which might be present as part of LTM, in addition to the other knowledge a user possesses.

3.1 Construction

3.1.1 Input

3.1.1.1 Input files

Three input files are required to build a LTM matrix and activate knowledge in LTM. Two of these, the LTM proposition file and the generic plan-element file, are required for construction and an additional file, the specific task description, is necessary for knowledge activation. The specifications for these three files are given below along with example files.

Proposition file. This file consists of a list of two types of propositions; general knowledge including what is known about general task descriptions from the past,

and the unbound plan-element names. This file is designed to represent the contents of LTM, but it has been simplified (for programming representation) to include only knowledge related to the domain being researched (in these examples this domain is computer related tasks).[9]

The task description propositions were constructed such that the request of any problem solved in the past is present in LTM, and is identifiable by a RQ prefacing the predicate of the proposition, e.g., (P14(RQSEND ?TEXT)). This information makes it possible for NETWORK to select an appropriate goal or outcome for a task in the subsequent, knowledge activation, stage of the simulation. The outcome of each previously encountered task is identified by an OUTC**n** where **n** is the proposition number of the request to which the outcome corresponds, e.g. (P15(OUTC14 RECEIVE ?TEXT)).

It is suggested that the proposition filename have the extension .ltm for the LTM matrix module. In our example this file is called *include.ltm* because it is being used to create a LTM for use in solving the INCLUDE task.

> NOTE: You may use any proposition numbers in your own input file except P9000-P9999. These numbers are used by the plan firing module (section 6.4) of the simulation and may result in a proposition number conflict

Example LTM proposition file *include.ltm*:

```
((P1(EXIST ADDRESS))
 (P2(EXIST LETTER))
 (P3(EXIST ?FILE^LETTER))
 (P4(EXIST ?TEXT^LETTER))
 (P5(EXIST ?TEXT^ADDRESS))
 (P6(AT-LEVEL SYSTEM ?FILE^LETTER))
 (P7(IN ?TEXT^LETTER ?FILE^LETTER))
 (P8(RQINCLUDE ?TEXT^ADDRESS ?TEXT^LETTER))
 (P9(RQINCLUDE ?TEXT^ADDRESS ?FILE^LETTER))
 (P10(OUTC8IN ?TEXT^ADDRESS ?TEXT^LETTER))
 (P11(OUTC9IN ?TEXT^ADDRESS ?FILE^LETTER))
        ... (more omitted for brevity)
 (P18(READ MESSAGE))
 (P21(RQSEND ?TEXT^PARAGRAPH))
 (P22(OUTC21RECEIVE ?TEXT^PARAGRAPH))
 (P30(ISA PARAGRAPH TEXT))
 (P39(SEND LETTER POSTOFFICE))
 (P40(WRITE LETTER FRIEND))
 (P41(AT-LEVEL SYSTEM))
 (P44(IS ADDRESS SHORT))
 (P45(KNOW ?FILE LOCATION))
 (P48(IN ?TEXT ?FILE))
 (L1(EDIT ?FILE))
```

---

[9]In the work done by Mannes (1989), these contents were derived from the verbal protocols of expert users solving computer tasks. It is left to the user to decide what to include in their version of LTM and to decide on a principled way of making this decision.

(L2(CUT ?TEXT ?FILE))
(L3(PASTE ?TEXT ?FILE))
(L4(PASTE ?TEXT ?BUFFER))
(L5(TYPE ?TEXT))
(L6(EXIT ?FILE))
(L7(FIND ?FILE))
(L8(READ ?FILE))
(L9(ENTER SYSTEM))...)

NOTE: Unbound plan-element names (Section 2.2.3.1) are included in this file at the end of the list and are used to build the LTM matrix. Please note that these are only the plan-element numbers and names (including the unique plan-element number preceded by a capital L and the first sublist of a full plan-element).

Unbound plan-element file. A plan-element file contains a list of plan-elements in their generic, unbound, form. Each plan-element consists of a list of sublists in this order: Unique plan-element number preceded by an L (which is not a sublist and is always the first element (car) of the list), plan-element name, plan-element preconditions and plan-element outcomes. Each sublist of each plan-element consists of an unlimited number of valid lisp lists with an unlimited number of symbols as arguments (long symbols should be concatenated with a -, a . or _). At this stage the variables in the plan-elements should not be bound. That is, the information contained in this file describes what is known about files, mail systems, and tasks, *in general.* The binding of these variables can be accomplished relatively automatically before a task connectivity matrix is built, by executing the variable binding module described in section 4. It is suggested that the filename of an unbound plan-element file contain the word *plan* as in our example file listed below called *includeplan1*. A detailed description of the binding process appears in Section 4.

Example plan-element:

(L1((EDIT ?FILE))	*Plan-element number and name*
((AT-LEVEL SYSTEM) (KNOW ?FILE LOCATION))	*Plan-element preconditions*
((AT-LEVEL EDIT ?BUFFER)))	*Plan-element outcome*

Note: Programming restrictions require that the maximum number of unique variables to be bound in each plan-element file be limited to 4. In the example plan-element file below, the variables ?FILE, ?BUFFER and ?TEXT are unbound. The maximum number of bindings per variable is limited to 3. For details on variable binding see Section 4.

Example unbound plan-element file *includeplan1*:[10]

[10]This plan-element file also has as part of its name a number. This is to show that this is the plan-element file to be used for the first step of the task. Often, during the solution of a task, a new file will come into being or an old file will be removed due to a delete or rename, etc. When this happens a

```
(L1((EDIT ?FILE))
     ((AT-LEVEL SYSTEM) (KNOW ?FILE LOCATION))
     ((AT-LEVEL EDIT ?BUFFER)))
(L2((CUT ?TEXT ?FILE))
     ((AT-LEVEL EDIT ?BUFFER) (IN ?TEXT ?BUFFER))
     ((IN ?TEXT ?BUFFER) (~IN ?TEXT ?BUFFER) (KNOW TEXT CONTENTS)))
(L3((PASTE ?TEXT ?FILE))
     ((AT-LEVEL EDIT ?BUFFER) (IN ?TEXT ?BUFFER))
     ((IN ?TEXT ?BUFFER ADDRESS) (KNOW ?TEXT CONTENTS)))
(L4((PASTE ?TEXT ?BUFFER))
     ((AT-LEVEL EDIT ?BUFFER) (IN ?TEXT ?BUFFER))
     ((IN ?TEXT ?BUFFER) (KNOW ?TEXT CONTENTS)))
(L5((TYPE ?TEXT))
     ((AT-LEVEL EDIT ?BUFFER) (KNOW ?TEXT CONTENTS) (~EXIST ?FILE))
     ((IN ?TEXT ?BUFFER ADDRESS)))
(L6((EXIT ?FILE))
     ((AT-LEVEL EDIT ?BUFFER))
     ((~AT-LEVEL EDIT)))
(L7((FIND ?FILE))
     ((AT-LEVEL SYSTEM) (EXIST ?FILE))
     ((KNOW ?FILE LOCATION)))
(L8((READ ?FILE))
     ((AT-LEVEL SYSTEM) (EXIST ?FILE) (KNOW ?FILE LOCATION))
     ((READ ?FILE) (KNOW ?FILE CONTENTS)))
(L9((ENTER SYSTEM))
     ((AT-LEVEL MAIL))
     ((AT-LEVEL SYSTEM)))...)
```

Common errors found in input files. The most common errors found in any of the input files described are mismatched or missing parentheses. It is very easy to create an input file with this error and it is often difficult to detect. When NETWORK reads in a file with a missing right parenthesis error you will receive the following message:

```
!!!!!Error: Unexpected end-of-file encountered.
Condition signalled in : LISP::FILE-IN
```

When there are too many right parentheses, you may not get an error message but your entire list of propositions may not have been read in. It is important that you know how many propositions you have and check this number against NETWORK's idea of how many propositions there are after it reads in the list. A good way to avoid parentheses errors is to create your file in an editor that will do automatic checking of right and left parentheses. Sloppy typing, preventing an exact match of lisp symbols, can also be a problem.

---

modified plan-element file must be created to take this change in object status into account. Thus, it is recommended that if your domain allows for the creation and deletion of objects, the step for which the current plan-element file is relevant be represented in its name.

Another error that is occasionally encountered is duplicate numbering of propositions or plan-elements. Each proposition must have a unique number prefixed with a P, such as P105. Each plan-element must also have a unique letter-number combination, but uses L as a prefix, such as L33. If you have a file that contains a duplicate proposition or plan-element number please renumber your propositions or plan-elements so that no duplicates exist, and recheck all propositional embeddings, associations, etc. that are achieved through the use of a proposition number and one of the special codes.

3.1.1.2 Keyboard input

NETWORK has been written in generic Common LISP. It should compile and load in your favorite LISP environment without any changes. After you have loaded the LISP code call the top level function (start) to start the simulation.

The LTM matrix construction module will prompt you for the following keyboard input. Please type the appropriate response and hit the enter key, assuming the required files now exist.

The prompts from NETWORK are shown here in **bold face type** and example responses are in *italic*. NETWORK's replies to your keyboard input that do not require additional response from you are in ***bold italic.*** Hints and comments on what should be typed are in plain text.

(start)
This top level LISP function call starts the simulation.

Enter a run trace filename: *nsf/work/ltm.trace*
In response to this prompt enter an output file name (full path is required) where NETWORK can store a run trace of the program. This should be a unique name of 16 or fewer alphanumeric characters. By convention, this should be named <filename>.trace. Many files are needed and created during the simulation so systematic naming will aid organization and reduce confusion. When you are done, this trace file will contain the filename from which the proposition list was taken, a list of the propositions, and a list of all the parameter values which were used in deriving the link strengths.

Enter proposition filename (q to quit): *nsf/work/include.ltm*
Enter the full path and name of the LTM proposition file you created. This file contains general knowledge including knowledge of previously encountered tasks, and unbound plan-element names.

This file contains 50 propositions.
NETWORK counts the propositions in this file and lets you know how many there are. Where there are N propositions the matrix that is built is of size N x N.

Enter plan file name (q to quit): *nsf/work/includeplan1*
Enter the name of an unbound plan-element file (see example on pg. 22).

Enter matrix output filename: *nsf/work/includeltm.mat*
Enter the name of the file in which you want the LTM matrix stored. By convention, include the name of the task for which it was constructed and the letters .mat in the filename.

Do you want to build a ltm matrix? (y or n) *y*
Type y in response to this question. NETWORK will now invoke the module that is used to construct the LTM matrix.[11]

This portion of the program will construct a long-term memory (ltm) matrix.
NETWORK lets you know that you are in the correct module of the program.

Number between 0.0 and 1.0 for the argument overlap weight: [0.4] *0.4*
Enter any real number between 0.0 and 1.0 inclusive in response to this prompt. This number will be used as the weight to represent the connection between propositions having argument overlap (see Section 2.2.2.1 for description of node relationships). 0.4 is the default weight for argument overlap and is the one used in the INCLUDE example. You have the choice of typing <enter> to use the default weight or typing in a value.

Number between 0.0 and 1.0 for the weight for proposition embedding: [0.5] *0.5*
Again enter a real number between 0.0 and 1.0 inclusive. The default for this weight is 0.5.

Number between -15.0 and 0.0 for the inhibition weight ($): [-10.0] *-10.0*
Enter a number between -15.0 and 0.0 inclusive. The default for this weight is -10.0.[12]

Number between 0.0 and 5.0 for the free association weight (*): [.5] *.5*
Enter a number between 0.0 and 5.0 inclusive. The default for this weight is .5.

Number between 0.0 and 5.0 for the next association weight ($*): [2.0] *2.0*
Enter a positive real number between 0.0 and 5.0 inclusive. The default for the association weight is 2.0.

Number between 0.0 and 1.0 for the plan relation weight: [0.5] *0.6*

---

[11]Note that the same program is used to construct both the LTM and task connectivity matrices. Which will be done in any given situation is determined by your responses to the questions NETWORK asks.

[12]There are two types of inhibition that may exist. One is the inhibition present in the causal chain of plan-elements. The value entered here will be used for that relationship. Additionally, there may be items which are inconsistent with each other, the link for which the user also wants to have an inhibitory value. The user may use the hard code $ preceding a proposition number as an argument of a proposition to signify this. Regardless of how it is derived an inhibitory link will always have this value.

Enter a real number between 0.0 and 1.0 inclusive. The default for this weight is 0.5. This value is used in computing the positive causal chaining among plan-elements.

Number between 0.0 and 1.0 for the no relationship weight (@): [0.5] *0.5*
Enter a real number between 0.0 and 1.0 inclusive. The default for the no relationship (or null relationship) weight is 0.5.[13]

A ltm matrix is being built...Please wait.
NETWORK will now proceed to build a LTM matrix. Please be patient, this sometimes takes awhile.

Common errors. There are several errors that can happen during keyboard input for this section. An easy one to make is to misspell or leave out the full path to one of your input files. If an input file cannot be found you will receive the following message:

ERROR! Can't find file *file/path/name*
Please re-enter correct filename.

To correct the error just retype the full path to the file correctly.

When the relationship weights are being input, a common error is to type a number out of range. When this occurs NETWORK will respond with the following message:

Bad Input: *10236*

To correct the error just retype the correct number within the ranges described above.[14]

3.1.2 Process

Once provided with all this information, NETWORK goes through the list of propositions recursively, finding any and all relationships that exist between all pairs of propositions. NETWORK then represents the degree of relationship, as defined by the parameter values the user has entered, in a matrix of values - one matrix for each relationship type. All relationships EXCEPT CAUSAL CHAINING are assumed to be symmetric so that entry i,j and j,i should be the same unless they involve an index to a plan-element proposition. These matrices are then combined according to the

---

[13]Although this may seem odd, that the no relationship weight is in fact a .5, the user must remember that these codes can be used for any of their own purposes. In the first author's dissertation this code signified no relationship, but it need not always.

[14]No theoretical reason exists for these limits. They were imposed for programming purposes and thus, the code which enforces these limits is changeable by the user without violation of the construction-integration theoretical framework.

following rules to produce a single matrix representing the system's LTM network. The matrix combination is guided by the following logic. All positive relationships are additive. All negative relations and zero relationships that were introduced by the use of the @ take precedence over positive relationships.

3.1.3 Output

This results in a matrix where there exists a row for each proposition, and the values in that row represent the degree of relationship that proposition has to each other proposition in LTM. The matrix contains N X N nodes where N is the number of propositions in LTM. This resulting matrix is then normalized so that knowledge activation may take place. The normalization process consists of adding all of the weights for any particular proposition (row) and dividing each of its individual connection weights to other propositions by the total of the weights it possesses. This produces a matrix where the weights in each row, representing a concepts relationship to all the others in the user's LTM, add to one. Normalization is necessary for the knowledge activation step, which is described below, because this activation is simulated as a probabilistic process.

At this point the LTM network has been constructed. It contains all that the system knows about the computer domain and is general so that it can be used for all tasks.

3.2 Selection of Free Associates

The second step in task solution, that of knowledge activation, is described next. It starts by a task description activating some portion of LTM which is relevant to a task, and then incorporating that content into the description of the given task. This represents how in a situation a user might bring to bear their past, general, knowledge to aid in task solution. Note that this may only be done during this module of the simulation and occurs after a LTM matrix has been constructed and before the actual task solution begins.

3.2.1 Input

3.2.1.1 Input files

To begin the process, NETWORK asks for the name of a file containing a task description, which has already been put into propositional format, and it treats the propositions as instructions. The specific task description file contains the task which the NETWORK simulation will solve. This instruction list, along with the LTM matrix just constructed, comprises the input for knowledge activation. In this example the input file shown below, which was mentioned in Section 2.3, is called *include.txt* and is for the task "include an address that is known into a letter being written on the computer".

The suggested name for this file should include the word txt such as *include.txt* because it represents the initial text from which the instructions have

been derived. The propositions in this file are used to select free associate knowledge from the LTM matrix so the user will be prompted for its name after LTM has been constructed.

Example specific task description file *include.txt*

```
((P1(EXIST ADDRESS))
 (P2(EXIST LETTER))
 (P3(EXIST ?FILE^LETTER))
 (P4(EXIST ?TEXT^LETTER))
 (P5(EXIST ?TEXT^ADDRESS))
 (P6(AT-LEVEL SYSTEM ?FILE^LETTER))
 (P7(ISA ?TEXT^LETTER ?FILE^LETTER))
 (P8(RQINCLUDE ?TEXT^ADDRESS ?TEXT^LETTER))
 (P9(RQINCLUDE ?TEXT^ADDRESS ?FILE^LETTER))
 (P41(AT-LEVEL SYSTEM)))
```

3.2.1.2 Keyboard input

The keyboard input for free associate selection is described below. You will be prompted for this input only after a LTM matrix has been built. Please type the appropriate response and hit the enter key. The prompts from NETWORK are **bold face type** and example responses are in *italic*. NETWORK's replies to your keyboard input that do not require additional response from you are in ***bold italic.*** Hints and comments on what should be typed are in plain text.

Enter the specific task description filename: */nsf/work/include.txt*
Enter the name of the file that contains your specific task description in propositional form. Be sure to include the entire path name.

Enter proposition output filename: *include1.out*
The file name entered here will become the main output file of the knowledge activation. This file, with manual modifications, will be used as input to NETWORK to construct the task connectivity matrix for the first step in the solution of the INCLUDE task(see Section 5).

Number between 0 and 10000 for the number of associates: [2] *2*
The integer entered here will determine how many free associates are chosen from general knowledge for each proposition in the task description.

Do you want an activation record sheet? (y or n) *y*
Reply yes if you would like an activation record sheet. The activation record sheet is a worksheet that can be used to manually record activations for each element in the task connectivity network after integration has been completed.

Enter an activation sheet filename: *include1.act*

3.2.2 Process

These instructions are used to address the matrix and activate several pieces of general knowledge which are related to the instructional propositions. NETWORK does this by sampling the LTM with associated probabilities. For each proposition in the task description, a random probability is generated for however many associates have been requested by the user. NETWORK then considers that proposition's relationship to each other proposition, incrementing a counter by the value it finds in each element of the proposition's row, until the random value is exceeded by the counter value. The proposition associated with the column entry that caused the counter value to exceed the random value is thus chosen as an associate. This allows for a probabilistic selection of associated knowledge, where items which are more related to the proposition under consideration have larger matrix entries and thus have a larger chance of being chosen.

The number of associates produced or selected for each proposition (N) in the instruction list is specifiable and a proposition can never be chosen as an associate for itself. Once these propositions have been found, arguments are added to the task description propositions to show which propositions were selected as their associates.

Associates for the request proposition(s) (those having their predicate prefaced by RQ) are not selected probabilistically, but rather a search for an appropriate goal proposition takes place using a composite cue much as in the memory retrieval model of Raaijmaker and Shiffrin (1981). The composite cue is comprised of the request proposition and a context cue of the form (GOAL-OF REQUEST $). An intersecting search takes place with this composite cue producing an item from LTM which is related to both propositions; an appropriate goal for the task request.

At this time, all of the generic plan-elements in LTM become active as well. Their names are added to the proposition list.

3.2.3 Output

This process produces two output files. The first, called *include1.out*, contains a list to be used for task connectivity construction (see Section 5) for the first step of the INCLUDE task. The list contains the current state of the world (original instruction propositions), along with a certain amount of general knowledge, and all potential plan-elements the user knows about. This output file with manual modification (See Section 5.1.1.1) will be used as input to the first task connectivity matrix construction.

The second output file contains the LTM matrix. This file is called *includeltm.mat* in our example and can now be examined to make sure that relationships exist where the user thinks they ought to and don't exist in places which the user hadn't anticipated. Since matrix output files produced by the simulation are usually very large, all matrix elements are truncated to the tenths position before this file is written.

Sample output file *include1.out*

```
((P1(EXIST ADDRESS *P44 INTHEWORLD))
(P2(EXIST LETTER *P39 INTHEWORLD))
(P3(EXIST ?FILE^LETTER *P9 *P11 INTHEWORLD))
(P4(EXIST ?TEXT^LETTER *P8 *P10 INTHEWORLD))
(P5(EXIST ?TEXT^ADDRESS *P48 INTHEWORLD))
(P6(AT-LEVEL SYSTEM ?FILE^LETTER *P11 *P48 INTHEWORLD))
(P7(ISA ?TEXT^LETTER ?FILE^LETTER *P11 *P6 INTHEWORLD))
(P8(RQINCLUDE ?TEXT^ADDRESS ?TEXT^LETTER *P4 *P10 INTHEWORLD))
(P9(RQINCLUDE ?TEXT^ADDRESS ?FILE^LETTER *P11 *P8 INTHEWORLD))
(P10(OUTC8IN ?TEXT^ADDRESS ?TEXT^LETTER *P4 *P7 INTHEWORLD))
(P11(OUTC9IN ?TEXT^ADDRESS ?FILE^LETTER *P48 *P10 INTHEWORLD))
(P41(AT-LEVEL SYSTEM INTHEWORLD))
(P39(SEND LETTER POSTOFFICE))
(P44(IS ADDRESS SHORT))
(P48(IN ?TEXT^ADDRESS ?FILE^LETTER))
(L1(EDIT ?FILE))
(L2(CUT ?TEXT ?FILE))
(L3(PASTE ?TEXT ?FILE))
(L4(PASTE ?TEXT ?BUFFER))
(L5(TYPE ?TEXT))
(L6(EXIT ?FILE))
(L7(FIND ?FILE))
(L8(READ ?FILE))
(L9(ENTER SYSTEM))...)
```

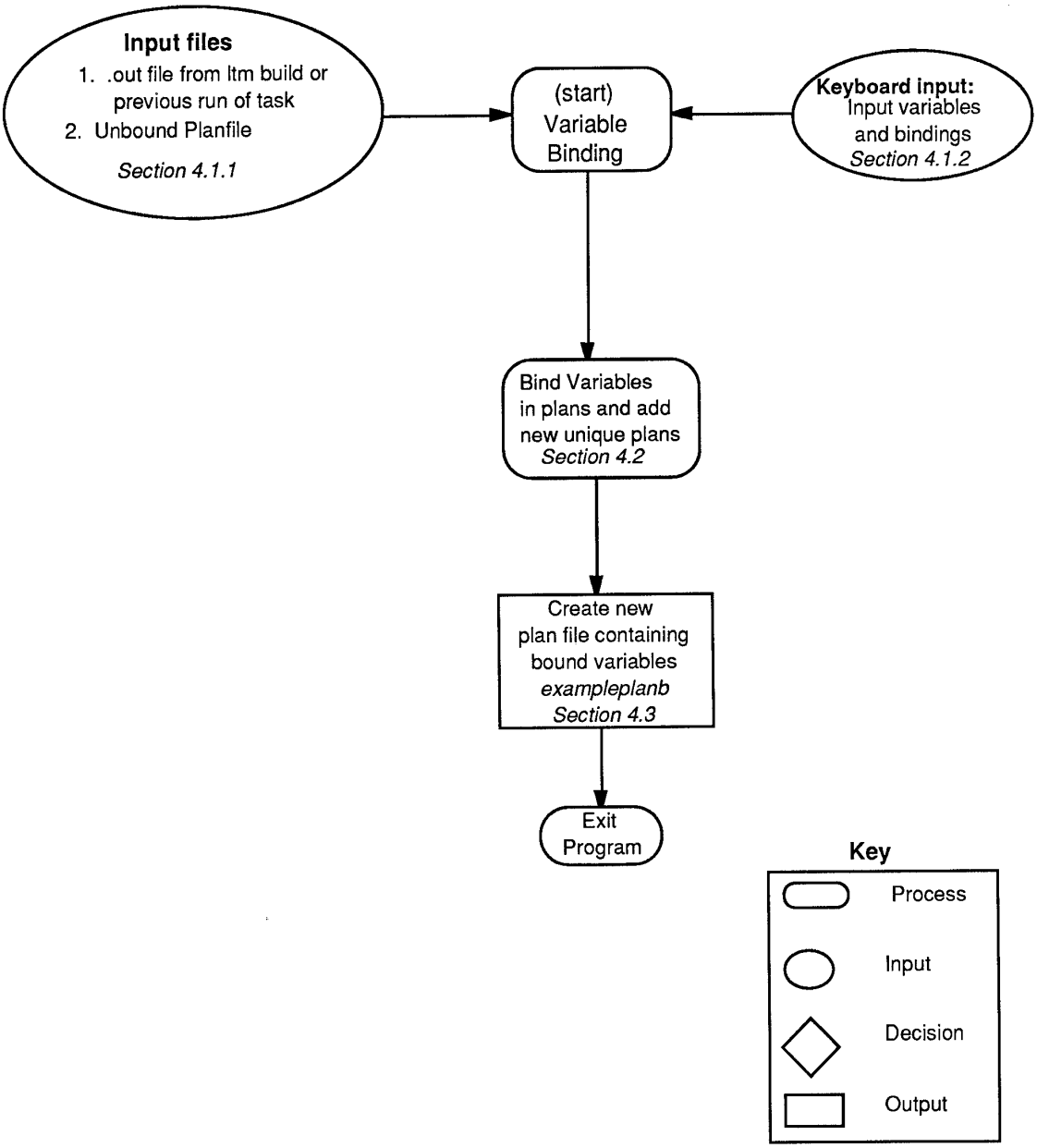
NOTE: Propositions containing variables may be chosen from LTM as free associates, e.g. (IN ?TEXT ?FILE). Variables must be manually bound in the free association section of the proposition file. NETWORK will not look for variables in propositions other than plan-elements so the user needs to be sure that all others have been bound manually. There can be a maximum of 4 different variables with a maximum of 3 bindings each. Both bound and unbound variables are separate symbols and are preceded with a ?. The binding is then separated from the variable name by a ^. Each variable name must be unique. For more details on variable binding representation please see Section 4.

Sample output file *includeltm.mat*

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P18	P21	P22	P30	P39	P40	P41	P44	P45	P48	L1	L2	L3	L4	L5	L6	L7	L8	L9	
P1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	P1
P2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	P2
P3	0.0	0.0	0.0	0.0	0.0	0.2	0.2	0.0	0.2	0.0	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	P3
P4	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.3	0.0	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	P4
P5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.2	0.2	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	P5
P6	0.0	0.0	0.1	0.0	0.0	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.1	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	P6
P7	0.0	0.0	0.1	0.1	0.0	0.1	0.0	0.1	0.1	0.1	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	P7
P8	0.0	0.0	0.0	0.1	0.1	0.0	0.1	0.0	0.1	0.2	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	P8
P9	0.0	0.0	0.1	0.0	0.1	0.1	0.1	0.1	0.0	0.1	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	P9
P10	0.0	0.0	0.0	0.1	0.1	0.0	0.1	0.2	0.1	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	P10
P11	0.0	0.0	0.1	0.0	0.1	0.1	0.1	0.1	0.2	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	P11
P18	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	P18
P21	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	P21
P22	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	P22
P30	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	P30
P39	0.0	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	P39
P40	0.0	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	P40
P41	0.0	0.0	0.0	0.0	0.0	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	P41
P44	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	P44
P45	0.0	0.0	0.2	0.0	0.0	0.2	0.2	0.0	0.2	0.0	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	P45
P48	0.0	0.0	0.1	0.0	0.1	0.1	0.1	0.1	0.2	0.1	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	P48
L1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.0	0.5	L1
L2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.0	0.0	0.5	0.0	0.0	0.0	0.0	0.0	L2
L3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	-10.0	0.0	0.5	0.0	0.0	0.0	0.0	0.0	L3
L4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	-10.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	L4
L5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.0	0.3	0.3	0.0	0.0	0.0	0.0	0.0	L5
L6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	L6
L7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	L7
L8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.0	0.5	L8
L9	0.0	0.0	0.0	0.0	0.0	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	L9
	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P18	P21	P22	P30	P39	P40	P41	P44	P45	P48	L1	L2	L3	L4	L5	L6	L7	L8	L9	

Variable Binding

Section 4



4 Binding Variables in Plan-elements

All plan-elements exist in an unbound state in LTM. In order for them to be appropriate for a task, the variables they contain must take on the values of the objects which are relevant in the task description. This is known as binding variables.

4.1 Input

4.1.1 Input file

An unbound plan-element file is the input to this section. The file name is entered at the prompt "Enter plan file name:". For an example unbound plan-element file see Section 3.1.1.1.

There can be a maximum of 4 different variables with a maximum of 3 bindings each. Variables are separate symbols preceded by a ?. The variable binding module will then separate the binding from the variable name by inserting a ^ between them.

This module of NETWORK will create a plan-element for each unique instance of every **?variable^binding** combination. In order to prevent combinatorial explosion in the plan-element file, care should be taken to use only necessary variables with a minimum number of bindings.

Example 1. Unbound plan-element to find the differences between two files:

(L32(DIFFERENCES ?FILE ?FILE2))
(in this example FILE and FILE2 are unbound)

Example 2. After binding for a task with two files, letter and manuscript:

(L32(DIFFERENCES ?FILE^LETTER ?FILE2^MANUSCRIPT))

4.1.2 Keyboard input

This module requires initial keyboard input that is very similar to the keyboard input for the LTM matrix construction (see Section 3). The prompts from NETWORK are **bold face type** and example responses are in *italic*. NETWORK's replies to your keyboard input that do not require additional response from you are in ***bold italic.*** Hints and comments on what should be typed are in plain text.

NETWORK has been written in generic Common LISP. It should compile and load in your favorite LISP environment without any changes. After you have loaded the LISP code call the top level function (start) to start the simulation.

(start)
This top level LISP function call starts the simulation.

Enter a run trace filename: *nsf/work/include1.trace*
In response to this prompt enter a trace file name (full path is required) where NETWORK can store a run trace of the program. This should be a unique name of 16 or fewer alphanumeric characters.

Enter proposition filename (q to quit): *nsf/work/include1.out*
Enter the full path and name of the output file created by the LTM knowledge activation procedure, by hand, or a previous iteration of NETWORK. Note the 1 in the input file name to denote that this is the input file for the first step of the INCLUDE task.

This file contains 28 propositions.
NETWORK counts the propositions in this file and lets you know how many there are. Where there are N propositions the matrix that is built is of size N x N.

Enter plan file name (q to quit): *nsf/work/includeplan1*
Enter the name of an unbound plan-element file for the first step of INCLUDE.

Enter matrix output filename: *nsf/work/include1.mat*
Enter the name of the file in which you want the task connectivity matrix for step 1 of INCLUDE stored.

Do you want to build a ltm matrix? (y or n) *n*
Type n in response to this question. Remember that a LTM matrix is only built once for each task and this should have already been done.

Do you want to bind variables in the plan-elements? (y or n) *y*
You should respond y (yes) to this prompt to bind variables in the unbound plan-element file. NETWORK searches through all the plan-elements in the plan-element file looking for any atoms preceded with a ?

(?FILE ?BUFFER ?TEXT)
NETWORK has found the three unbound variable listed above. You will now be prompted for values (bindings) for those variables.

Do you want to bind ?FILE? (y or n) *y*
Enter a y here if you want to bind the variables.

Enter binding for ?FILE. *LETTER*
Enter a binding for the variable **?FILE.** In the INCLUDE example, ?FILE has the value LETTER.

Do you have another binding for this variable? (y or n) *n*
Remember that each variable can have up to 3 different bindings. In the INCLUDE example ?FILE is bound only to LETTER so in this case reply n to this question.

Do you want to bind ?BUFFER? (y or n) *y*
Enter a y here if you want to bind the variable ?BUFFER.

Enter binding for ?BUFFER. *LETTER*
Enter a binding for the variable ?BUFFER. In the INCLUDE example BUFFER has no explicit binding so it must be given all of the possible bindings that can be taken on. .

Do you have another binding for this variable? (y or n) *y*
Remember that each variable can have up to 3 different bindings. In the INCLUDE task description example ?BUFFER is not bound so it must be assigned all possible bindings from the objects mentioned.

Enter binding for ?BUFFER. *ADDRESS*
Enter a binding for the variable ?BUFFER. In the INCLUDE example the other object named is ADDRESS.

Do you have another binding for this variable? (y or n) *n*
Remember that each variable can have up to 3 different bindings. In the INCLUDE example ?BUFFER is bound only to LETTER and ADDRESS so in this case reply n to this question.

Do you want to bind ?TEXT? (y or n) *y*
Reply y to this question since the variable ?TEXT needs to be bound.

Enter binding for ?TEXT. *ADDRESS*
Enter the first binding for the variable ?TEXT. In this example it is ADDRESS.

Do you have another binding for this variable? (y or n) *y*
Reply y to this question since there is another binding for ?TEXT.

Enter binding for ?TEXT. *LETTER*
Enter the second binding for ?TEXT, in this case LETTER.

Do you have another binding for this variable? (y or n) *n*
Reply n if you have no more bindings.

A new file called "nsf/work/includeplan1.b" which contains your bound plans has been created for you.
NETWORK creates a new file containing the bound plan-elements for you to use with task connectivity construction iterations.

> NOTE: Once plan-elements are bound, if new plan-elements are created, the plan numbers are changed by the variable binding section of the simulation. Therefore you must check the manual hard coding of plan-element relations discussed in Section 5.1.1.1 before going on to task connectivity matrix construction.

4.1.3 Common Errors in Plan-element Binding

The most common error in this module would be to have more than 4 variables that need to be bound or more than 3 bindings for each variable. These limitations exist because NETWORK creates a set of plan-elements to represent each **?variable^binding** combination, duplicating generic plan-elements as necessary. The matrix resulting from this plan-element file could become so large that most computers could not access all the array elements. If you find that your task description requires more than 4 unique variables you may need to simplify your task description or separate it into several smaller subtasks. This is obviously a constraint imposed by the implementation and not a limitation of the theory.

As always, a missing or mismatched parentheses can cause problems. (See Section 3.1.1.1 for more details on this error and solutions.)

4.2 Process

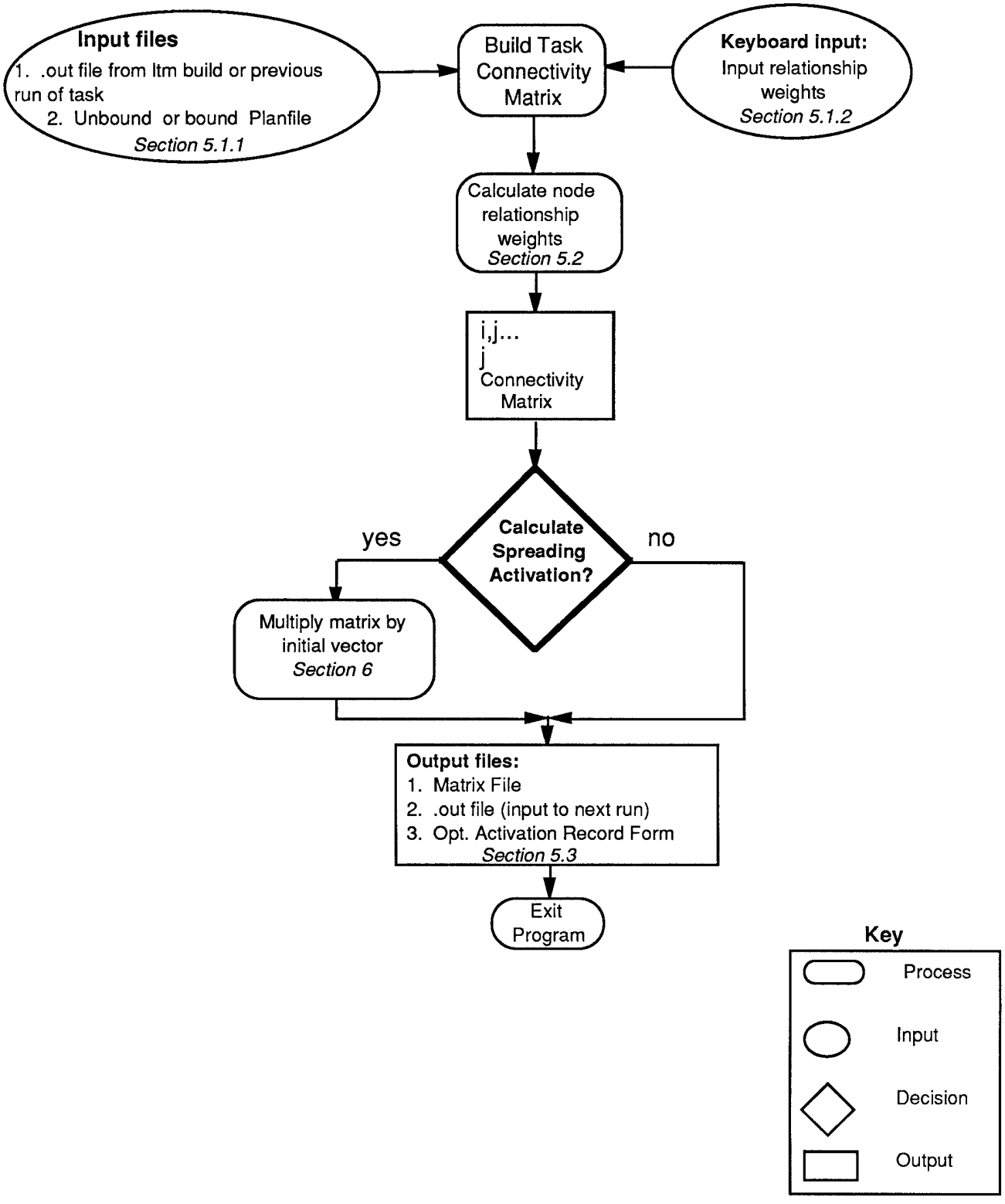
The unbound plan-element file is searched for all occurrences of ?variable. The user is then prompted (see section 4.1.2) for bindings for each ?variable. After the variables are bound each ?variable in the plan-element file is replaced with the new ?variable^binding. A new set of plan-elements are created for each ?variable^binding combination. The plan-elements are renumbered to include the ones that were newly created. The new plan-element names are added to the end of the proposition file (in this example *include1.out)* that will be used as input into the task connectivity matrix construction (described in section 5).

4.3 Output (bound plan-element file)

The plan-element file created by the variable binding module contains a set of plan-elements with a unique plan-element for each **?variable^binding** combination. See Section 5.1.1.2 for a listing of a bound plan-element file. The proposition file (*include1.out*) is also modified if necessary.

Task Connectivity Matrix Build

Section 5



5 Constructing a Task Connectivity Matrix

In Section 3 you learned how to construct a LTM and to sample related general information from LTM with the INCLUDE instructions to produce a list of propositions containing the original task description, the general knowledge which has been activated, and a list of plan-elements. Aside from the influence of this recently activated material in solving the task, LTM now plays no role in future computations. The explanation which follows, then, is concerned with the solution of a specific task, the INCLUDE task, and not with the development, use, or contents of LTM.

A task is considered solved when the initial request (see Section 2.2.1) has been carried out and the selected goal has been met. This usually requires several iterations through both construction and integration phases of NETWORK. Each iteration through the program requires the running of both the construction phase, which adds the results of the previous step to the task connectivity matrix, and the integration phase, which allows a new plan-element to be chosen for the current step.

The text of the INCLUDE task, "include an address that is known into a letter being written on the computer" was propositionalized and used as part of the input file for the LTM matrix build (see Section 3). The output proposition file from LTM construction and knowledge activation (include1.out) described in Section 3.2.3, with the manual modifications described in Section 5.1.1.1 becomes the proposition input file used in building the first task connectivity matrix. In subsequent iterations through this module of the program the output file created in Section 6.4.2 with manual modifications becomes the proposition input file for the next step.

Plan-elements are also needed to construct a task connectivity matrix. If an unbound plan-element file is used the plan-elements must be bound using variable binding module described in Section 4 before the user can go on.

After the task connectivity matrix has been constructed the matrix is ready to be used as input to the integration phase where activation is spread through the network (see Section 6).

In this section of the documentation you learn how to use the NETWORK simulation to construct a task connectivity matrix. There is a complete description of the input files and keyboard input that are necessary to get NETWORK started on a solution to your own specific task, using INCLUDE as an example. You learn how NETWORK produces link values for the associative network based upon the relations between nodes and the parameter values you supply. The output files from this second module of the program are described and discussed.

The flow of this section of the program is described on the flow diagram on page 36.

5.1 Input

5.1.1 Input Files

There are two input files needed to construct a task connectivity matrix. The first is the proposition output file (.out file) from LTM construction and knowledge activation, or a proposition file resulting from a previous construction-integration cycle. The second is a bound plan-element file appropriate for the task.

The first file (described below) is the proposition file produced by a LTM run of NETWORK (see Section 3) and is used to produce a task connectivity matrix representing an understanding of the INCLUDE (or any other) task, a situation model built up from the provided instructions. The input list contains all three types of propositions; those true in the world including the task to be done, (these propositions have as their last argument INTHEWORLD), those representing the influence of general knowledge from LTM, and plan-element names.

5.1.1.1 Proposition input file

The .out file from LTM construction and knowledge activation processes contains the specific task description (or current state of the world), related general knowledge (including the expected outcome for the request), and all bound or unbound plan-element names. NETWORK has added the argument INTHEWORLD to each task description proposition to distinguish it from related information which has been activated in the system's LTM. It has added a *P*n* of the free associate propositions which were chosen from LTM during knowledge activation to their argument lists. The *n* in this case refers to the proposition number of the proposition which was chosen as an associate. For example, if (P16(IN MESSAGE MAIL)) activates (P17(USE MAIL STUDENT)) in LTM, NETWORK will represent this by adding *P17 to the argument list of P16, thus (P16(IN MESSAGE MAIL *P17)). All of this information is in propositional form.

Manual hard coding of relationships. There are three special cases of relationships between the request and goal propositions and the plan-elements which must be coded manually. This is accomplished by extending the argument list for all request and goal propositions to include an argument for each of the plan-elements. These arguments specify the relationships the request or goal shares with the plan-elements. The algorithm for assignment of these special relations is as follows.

The request proposition contents are matched to all of the possible plan-element names, their first field. Whenever a plan-element name is found that specifies the request, (an exact match is not necessary only a semantic one) a hard coded positive relationship between that plan-element and the request is specified by using the $* special character. All other request to plan-element connections are set to zero using the @. Thus, the request proposition (P14(RQSEND ?TEXT)) becomes (P14'(RQSEND ?TEXT @L1... $*Li...@Ln INTHEWORLD)), where **i** is the plan-element number of any plan(s) which would accomplish SEND and **n** is the total number of plan-elements.

The connections between the goal propositions and the operators is slightly more complex. Here, matches are based on the plan-elements' outcome fields and can be of three types. One, the outcome of a plan-element can match the goal proposition and a positive hard code is specified with $*. In this case, the outcome of a plan-element must deal with the same objects mentioned in the goal statement and must produce an effect consistent with the goal. Two, if the outcome of a plan-element matches the goal because it deals with the same objects as the goal but produces an effect inconsistent with that goal, a hard coded inhibitory connection is specified using $. Three, all other plan-elements which deal with objects other than those specified in the request and goal statements receive a zero connection strength to the goal statement by using @L*n*.

The identification of semantic relationships between requests and plan-elements would require NETWORK to do some sophisticated natural language processing which is beyond the current scope of the simulation. Therefore, the above algorithm is applied manually to the request and outcome propositions in the proposition output file (see Section 3.2.3). After these manual modifications are added to this file, it is ready to be used as input to task connectivity matrix construction.

Example file *include1.out* with manual entries (manual entries were added in P8-P11):

```
((P1(EXIST ADDRESS *P44 INTHEWORLD))
(P2(EXIST LETTER *P39 INTHEWORLD))
(P3(EXIST ?FILE^LETTER *P9 *P11 INTHEWORLD))
(P4(EXIST ?TEXT^LETTER *P8 *P10 INTHEWORLD))
(P5(EXIST ?TEXT^ADDRESS *P48 INTHEWORLD))
(P6(AT-LEVEL SYSTEM ?FILE^LETTER *P11 *P48 INTHEWORLD))
(P7(ISA ?TEXT^LETTER ?FILE^LETTER *P11 *P6 INTHEWORLD))
(P8(RQINCLUDE ?TEXT^ADDRESS ?TEXT^LETTER *P4 *P10 @L1 @L2 @L3 @L4 $*L5 @L6 @L7
    @L8 @L9 @L10 $*L11 @L12 $*L13 INTHEWORLD))
(P9(RQINCLUDE ?TEXT^ADDRESS ?FILE^LETTER *P11 *P8 @L1 @L2 @L3 @L4 $*L5 @L6 @L7
    @L8 @L9 @L10 $*L11 @L12 $*L13 INTHEWORLD))
(P10(OUTC8IN ?TEXT^ADDRESS ?TEXT^LETTER *P4 *P7 @L1 $L2 @L3 $L4 @L5 @L6 @L7 @L8
    @L9 @L10 $*L11 @L12 $*L13 INTHEWORLD))
(P11(OUTC9IN ?TEXT^ADDRESS ?FILE^LETTER *P48 *P10 @L1 $L2 @L3 @L4 $*L5 @L6 @L7 @L8
    @L9 @L10 $*L11 @L12 $*L13 INTHEWORLD))
(P41(AT-LEVEL SYSTEM INTHEWORLD))
(P39(SEND LETTER POSTOFFICE))
(P44(IS ADDRESS SHORT))
(P48(IN ?TEXT^ADDRESS ?FILE^LETTER))
(L1(EDIT ?FILE^LETTER))
(L2(CUT ?TEXT^LETTER ?FILE^LETTER))
(L3(PASTE ?TEXT^LETTER ?FILE^LETTER))
(L4(PASTE ?TEXT^LETTER ?BUFFER^ADDRESS))
(L5(TYPE ?TEXT^LETTER))
(L6(EXIT ?FILE^LETTER))
(L7(FIND ?FILE^LETTER))
(L8(READ ?FILE^LETTER))
(L9(ENTER SYSTEM))
```

(L10(CUT ?TEXT^ADDRESS ?FILE^LETTER))
(L11(PASTE ?TEXT^ADDRESS ?FILE^LETTER))
(L12(PASTE ?TEXT^ADDRESS ?BUFFER^ADDRESS))
(L13(TYPE ?TEXT^ADDRESS))...)

5.1.1.2 Plan-element file

The second file needed to build a task connectivity matrix is a bound plan-element file. Remember that plan-elements are comprised of a plan-element number and three sublists; a plan-element name, pre-conditions for that particular plan-element to be executable, and outcomes, things which become true in the world when a plan-element is executed (see Section 2.2.1). You may run NETWORK for the sole purpose of binding the variables in your plan-elements for you before a task connectivity matrix is built. When the variables are bound a new set of plan-elements for each unique **?variable^binding** is created and added to the plan-element list with a unique plan-element number.

> **Note:** The following example plan-element file is a subset of the plan-element file used in Mannes (1989). The number of plan-elements was reduced in this example for simplicity and readability. In most cases the plan-element file used to run the simulation should contain all possible plan-elements that could be carried out given a current state of the world.

Example bound plan-element file *includeplan1.b:*

```
((L1((EDIT ?FILE^LETTER))
      ((AT-LEVEL SYSTEM) (KNOW ?FILE^LETTER LOCATION))
      ((AT-LEVEL EDIT ?BUFFER^LETTER)(KNOW ?TEXT^ADDRESS CONTENTS)))
(L2   ((CUT ?TEXT^LETTER ?FILE^LETTER))
      ((AT-LEVEL EDIT ?BUFFER^LETTER) (IN ?TEXT^LETTER ?BUFFER^LETTER))
      ((IN ?TEXT^LETTER ?BUFFER^LETTER) (~IN ?TEXT^LETTER ?BUFFER^LETTER)
      (KNOW TEXT CONTENTS)))
(L3   ((PASTE ?TEXT^LETTER ?FILE^LETTER))
      ((AT-LEVEL EDIT ?BUFFER^LETTER) (IN ?TEXT^LETTER ?BUFFER^LETTER))
      ((IN ?TEXT^LETTER ?BUFFER^LETTER ADDRESS) (KNOW ?TEXT^LETTER
      CONTENTS)))
(L4   ((PASTE ?TEXT^LETTER ?BUFFER^LETTER))
      ((AT-LEVEL EDIT ?BUFFER^LETTER) (IN ?TEXT^LETTER ?BUFFER^LETTER))
      ((IN ?TEXT^LETTER ?BUFFER^LETTER) (KNOW ?TEXT^LETTER CONTENTS)))
(L5   ((TYPE ?TEXT^LETTER))
      ((AT-LEVEL EDIT ?BUFFER^LETTER) (KNOW ?TEXT^LETTER CONTENTS))
      ((IN ?TEXT^LETTER ?BUFFER^LETTER ADDRESS)))
(L6   ((EXIT ?FILE^LETTER))
      ((AT-LEVEL EDIT ?BUFFER^LETTER)(IN ?TEXT^ADDRESS ?BUFFER^LETTER
      ADDRESS))
      ((~AT-LEVEL EDIT)))
(L7   ((FIND ?FILE^LETTER))
      ((AT-LEVEL SYSTEM) (EXIST ?FILE^LETTER))
      ((KNOW ?FILE^LETTER LOCATION)))
(L8   ((READ ?FILE^LETTER))
      ((AT-LEVEL SYSTEM) (EXIST ?FILE^LETTER) (KNOW ?FILE^LETTER LOCATION))
      ((READ ?FILE^LETTER) (KNOW ?FILE^LETTER CONTENTS)))
(L9   ((ENTER SYSTEM))
```

```
        ((AT-LEVEL MAIL))
        ((AT-LEVEL SYSTEM)))
(L10    ((CUT ?TEXT^ADDRESS ?FILE^LETTER))
        ((AT-LEVEL EDIT ?BUFFER^LETTER) (IN ?TEXT^ADDRESS ?BUFFER^LETTER))
        ((IN ?TEXT^ADDRESS ?BUFFER^LETTER) (~IN ?TEXT^ADDRESS
        ?BUFFER^LETTER) (KNOW TEXT CONTENTS)))
(L11    ((PASTE ?TEXT^ADDRESS ?FILE^LETTER))
        ((AT-LEVEL EDIT ?BUFFER^LETTER) (IN ?TEXT^ADDRESS ?BUFFER^LETTER))
        ((IN ?TEXT^ADDRESS ?BUFFER^LETTER ADDRESS) (KNOW ?TEXT^ADDRESS
        CONTENTS)))
(L12    ((PASTE ?TEXT^ADDRESS ?BUFFER^LETTER))
        ((AT-LEVEL EDIT ?BUFFER^LETTER) (IN ?TEXT^ADDRESS ?BUFFER^LETTER))
        ((IN ?TEXT^ADDRESS ?BUFFER^LETTER) (KNOW ?TEXT^ADDRESS CONTENTS)))
(L13    ((TYPE ?TEXT^ADDRESS))
        ((AT-LEVEL EDIT ?BUFFER^LETTER) (KNOW ?TEXT^ADDRESS CONTENTS))
        ((IN ?TEXT^ADDRESS ?BUFFER^LETTER ADDRESS)))...)
```

5.1.2 Keyboard Input

NETWORK has been written in generic Common LISP. It should compile and load in your favorite LISP environment without any changes. After you have loaded the LISP code call the top level function (start) to start the simulation.

This module requires initial keyboard input that is very similar to the keyboard input for the LTM matrix construction (see Section 3.1.1.2). The prompts from NETWORK are shown here in **bold face type** and example responses are in *italic*. NETWORK's replies to your keyboard input that do not require additional response from you are in ***bold italic.*** Hints and comments on what should be typed are in plain text.

Enter a run trace filename: *nsf/work/include1.trace*
In response to this prompt enter a trace file name (full path is required) where NETWORK can store a run trace of the program. This should be a unique name of 16 or fewer alphanumeric characters.

Enter proposition filename (q to quit): *nsf/work/include1.out*
Enter the full path and name of the output file created by LTM construction and knowledge activation, or a previous iteration of NETWORK that has had the manual coding of request and outcome propositions already done. This file is the input file used to build the task connectivity matrix for the first step of INCLUDE.

This file contains 28 propositions.
NETWORK counts the propositions in this file and lets you know how many there are. Where there are N propositions the matrix that is built is of size N x N.

Enter plan file name (q to quit): *nsf/work/includeplan1*
Enter the name of either an unbound plan-element file (if you are going to execute the variable binding module described in Section 4) or the previously bound plan-element file, includeplan1.b.

Enter matrix output filename: *nsf/work/include1.mat*
Enter the name of the file in which you want the task matrix stored. By convention this file should be named <*task*><*stepnumber*>.mat.

Do you want to build a ltm matrix? (y or n) *n*
Type n in response to this question. Remember that a LTM matrix is only built once for each task.

Do you want to bind variables in the plans? (y or n) *n*
If you have an unbound plan-element file answer y (yes) to this question and proceed to Section 4, otherwise reply n (no) and continue.

This portion of the program will construct a task connectivity matrix.
NETWORK is letting you know that you have entered the module that will build a task connectivity matrix.

Number between 0.0 and 1.0 for the argument overlap weight: [0.4] *0.4*
Enter any real number between 0.0 and 1.0 inclusive in response to this prompt. This number will be used as the weight to calculate argument overlap (see Section 3.1.1.1 for description of node relationships). 0.4 is the default weight for argument overlap and is the one used in the INCLUDE example. You have the choice of typing enter to use the default weight.

Number between 0.0 and 1.0 for the weight for proposition embedding: [0.5] *0.5*
Again enter a real number between 0.0 and 1.0 inclusive. The default for this weight is 0.5.

Number between -15.0 and 0.0 for the inhibition weight ($): [-10.0] *-10.0*
Enter a real number between -15.0 and 0.0 inclusive. The default for this weight is -10.0.

Number between 0.0 and 5.0 for the free association weight (*): [.5] *.5*
Enter a real number between 0.0 and 5.0 inclusive. The default for this weight is .5.

Number between 0.0 and 5.0 for the next association weight ($*): [2.0] *2.0*
Enter a real number between 0.0 and 5.0 inclusive. The default for the association weight is 2.0.

Number between 0.0 and 1.0 for the plan relation weight: [0.5] *0.6*
Enter a real number between 0.0 and 1.0. The default for this weight is 0.5.

Number between 0.0 and 1.0 for the no relationship weight (@): [0.5] *0.5*
Enter a real number between 0.0 and 1.0 inclusive. The default for the no relationship (or null relationship) weight is 0.5.

A task-connectivity matrix is being built...Please wait.
NETWORK will now proceed to build a task connectivity memory matrix.

Common errors. There are several errors that can happen during keyboard input for this section. An easy one to make is to misspell or leave out the full path to one of your input files. If an input file cannot be found you will receive the following message:

ERROR! Can't find file */file/path/name*
Please re-enter correct filename.

To correct the error just retype the full path to the file correctly.

When the relationship weights are being input, a common error is to type a number out of range. When this occurs NETWORK will respond with the following message:

Bad Input: *10236*

To correct the error just retype the correct number within the ranges described above.

5.2 Process

This new task specific, INCLUDE, connectivity matrix is constructed according to the original LTM creation (see Section 3) procedure based on the seven mentioned types of relationship; (approximation processes described in Section 2.2.2) argument overlap, propositional embedding, associated knowledge, inhibition, null relations, plan-world inhibition, and interplan-element relations.

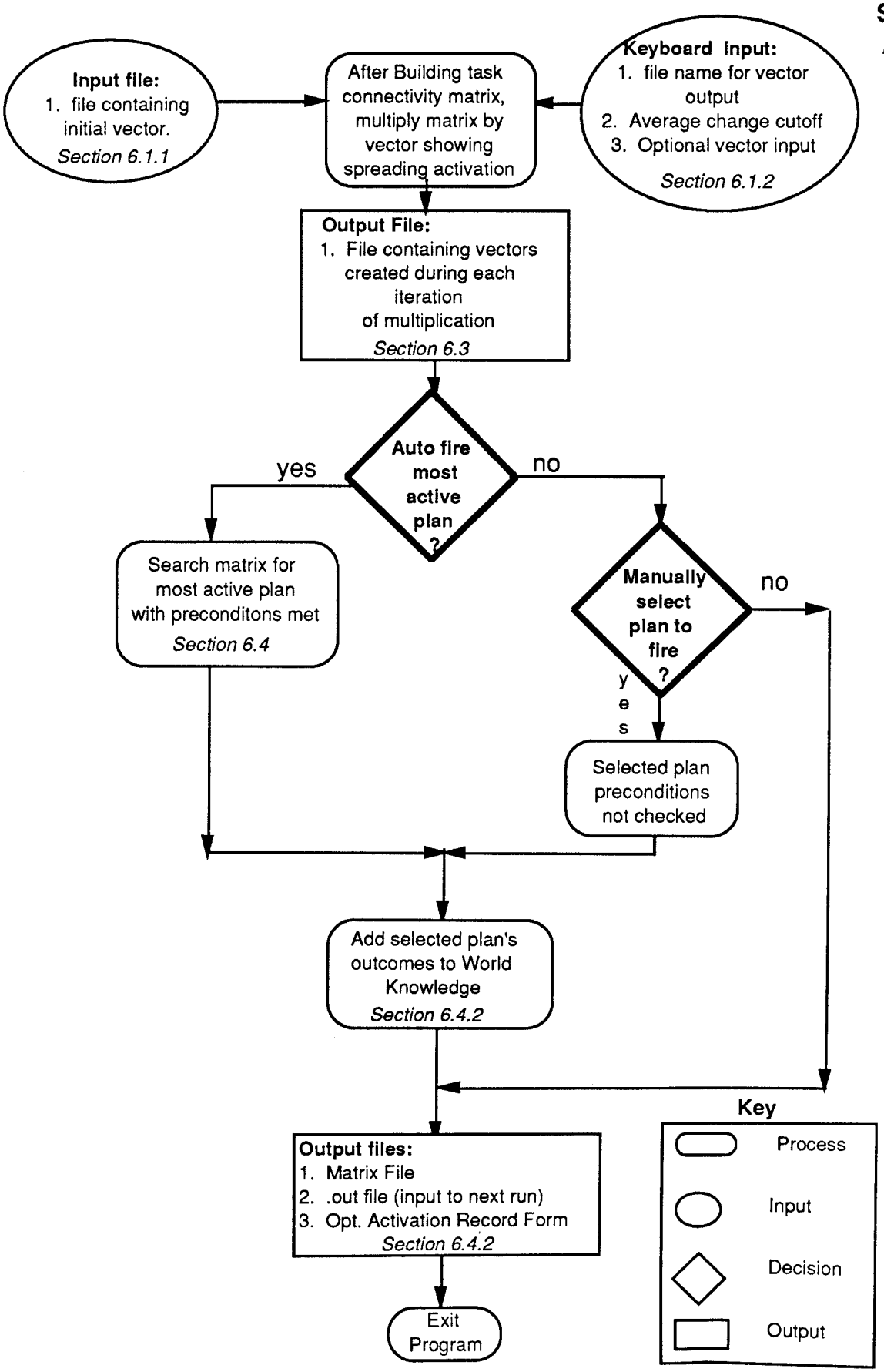
Additional relationships between nodes are also introduced as follows. Entries along the diagonal are assigned a value of 1 to represent the identity of a proposition and the matrix is not, in this phase, normalized. Also, although propositions and plan-element proposition names which share arguments receive a link indicating this by the argument overlap relationship type, plan-world inhibition relationships are also computed (see Section 2.2.3.1).

5.3 Output

This new connectivity matrix, representing the INCLUDE network for the first step, contains many types of information with varying degrees of inter-relations. Parts of this network might be contradictory and some of the included information might be irrelevant to the task solution. At the beginning of a solution attempt all plan-elements might be equally active and the appropriate ones need to become differentially strengthened. The matrix created in this step of the program becomes the input into the next phase of NETWORK, the calculation of spreading activation (see Section 6). This matrix is saved to a file when the simulation exits. In our example the file is called *include1.mat* and is listed below.

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P41	P39	P44	P48	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L11	L12	L13	
P1	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	P1
P2	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	P2
P3	0.0	0.0	1.0	0.0	0.0	0.4	0.4	0.0	0.4	0.0	0.4	0.0	0.0	0.0	0.4	0.4	0.4	0.4	0.0	0.0	0.4	0.4	0.4	0.0	0.4	0.4	0.0	0.0	P3
P4	0.0	0.0	0.0	1.0	0.0	0.0	0.4	0.4	0.0	0.4	0.0	0.0	0.0	0.0	0.0	0.0	0.4	0.4	0.4	0.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	P4
P5	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.4	0.4	0.4	0.4	0.0	0.0	0.0	0.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.4	0.4	0.4	0.4	P5
P6	0.0	0.0	0.4	0.0	0.0	1.0	0.4	0.0	0.4	0.0	0.4	0.4	0.0	0.0	0.4	0.4	0.4	0.4	0.0	0.0	0.4	0.4	0.4	0.4	0.4	0.4	0.0	0.0	P6
P7	0.0	0.0	0.4	0.4	0.0	0.4	1.0	0.4	0.4	0.4	0.4	0.0	0.0	0.0	0.4	0.4	0.8	0.8	0.4	0.4	0.4	0.4	0.4	0.0	0.4	0.4	0.0	0.0	P7
P8	0.0	0.0	0.0	0.4	0.4	0.0	0.4	1.0	0.4	0.8	0.4	0.0	0.0	0.0	0.4	0.0	0.2	0.2	0.2	2.4	0.0	0.0	0.0	0.0	0.2	2.4	0.2	2.4	P8
P9	0.0	0.0	0.4	0.0	0.4	0.4	0.4	0.4	1.0	0.4	0.8	0.0	0.0	0.0	0.8	0.2	0.2	0.2	0.0	2.0	0.2	0.2	0.2	0.0	0.4	2.8	0.2	2.4	P9
P10	0.0	0.0	0.0	0.4	0.4	0.0	0.4	0.8	0.4	1.0	0.4	0.0	0.0	0.0	0.4	0.0	-10.0	0.2	-10.0	0.2	0.0	0.0	0.0	0.0	0.2	2.4	0.2	2.4	P10
P11	0.0	0.0	0.4	0.0	0.4	0.4	0.4	0.4	0.8	0.4	1.0	0.0	0.0	0.0	0.8	0.2	-10.0	0.2	0.0	2.0	0.2	0.2	0.2	0.0	0.4	2.8	0.2	2.4	P11
P41	0.0	0.0	0.0	0.0	0.0	0.4	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-10.0	0.0	0.0	0.0	0.0	P41
P39	0.0	0.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	P39
P44	0.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	P44
P48	0.0	0.0	0.4	0.0	0.4	0.4	0.4	0.4	0.8	0.4	0.8	0.0	0.0	0.0	1.0	0.4	0.4	0.4	0.0	0.0	0.4	0.4	0.4	0.0	0.8	0.8	0.4	0.4	P48
L1	0.0	0.0	0.4	0.0	0.0	0.4	0.4	0.0	0.2	0.0	0.2	0.0	0.0	0.0	0.4	1.0	0.0	0.0	0.0	0.0	0.0	0.5	0.0	0.5	0.0	0.0	0.0	0.0	L1
L2	0.0	0.0	0.4	0.4	0.0	0.4	0.8	0.2	0.2	-10.0	-10.0	0.0	0.0	0.0	0.4	0.5	1.0	0.0	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	L2
L3	0.0	0.0	0.4	0.4	0.0	0.4	0.8	0.2	0.2	0.2	0.2	0.0	0.0	0.0	0.4	0.5	-10.0	1.0	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	L3
L4	0.0	0.0	0.0	0.4	0.0	0.0	0.4	0.2	0.0	-10.0	0.0	0.0	0.0	0.0	0.0	0.5	-10.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	L4
L5	0.0	0.0	0.0	0.4	0.0	0.0	0.4	2.4	2.0	0.2	2.0	0.0	0.0	0.0	0.0	0.5	0.0	0.5	0.5	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	L5
L6	0.0	0.0	0.4	0.0	0.0	0.4	0.4	0.0	0.2	0.0	0.2	0.0	0.0	0.0	0.4	0.5	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	L6
L7	0.0	0.0	0.4	0.0	0.0	0.4	0.4	0.0	0.2	0.0	0.2	0.0	0.0	0.0	0.4	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.5	0.0	0.0	0.0	0.0	L7
L8	0.0	0.0	0.4	0.0	0.0	0.4	0.4	0.0	0.2	0.0	0.2	0.0	0.0	0.0	0.4	0.0	0.0	0.0	0.0	0.0	0.0	0.5	1.0	0.5	0.0	0.0	0.0	0.0	L8
L9	0.0	0.0	0.0	0.0	0.0	0.4	0.0	0.0	0.0	0.0	0.0	0.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	L9
L10	0.0	0.0	0.4	0.0	0.4	0.4	0.4	0.2	0.4	0.2	0.4	0.0	0.0	0.0	0.8	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.5	0.0	L10
L11	0.0	0.0	0.4	0.0	0.4	0.4	0.4	2.4	2.8	2.4	2.8	0.0	0.0	0.0	0.8	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-10.0	1.0	0.5	0.0	L11
L12	0.0	0.0	0.0	0.0	0.4	0.0	0.0	0.2	0.2	0.2	0.2	0.0	0.0	0.0	0.4	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-10.0	0.0	1.0	0.0	L12
L13	0.0	0.0	0.0	0.0	0.4	0.0	0.0	2.4	2.4	2.4	2.4	0.0	0.0	0.0	0.4	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.5	1.0	L13

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P41	P39	P44	P48	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L11	L12	L13	


Input file:
1. file containing initial vector.
Section 6.1.1
After Building task connectivity matrix, multiply matrix by vector showing spreading activation
Keyboard Input:
1. file name for vector output
2. Average change cutoff
3. Optional vector input
Section 6.1.2
Output File:
1. File containing vectors created during each iteration of multiplication
Section 6.3
Auto fire most active plan ?
yes
no
Search matrix for most active plan with preconditons met
Section 6.4
Manually select plan to fire ?
no
yes
Selected plan preconditions not checked
Add selected plan's outcomes to World Knowledge
Section 6.4.2
Output files:
1. Matrix File
2. .out file (input to next run)
3. Opt. Activation Record Form
Section 6.4.2
Exit Program
Key
Process
Input
Decision
Output

6 Integration - Calculating Spreading Activation

Vector multiplication by the connectivity matrix is repeated until a stable, but arbitrary, user defined criterion has been reached. This activation-relaxation process is carried out by a vector-matrix multiplication module initially written by Turner (1987).

The initial input vector consists of a string of one's for each of the propositions which were either a part of the original task description or which have become a part of the network as a result of a plan-element proposition having been executed (plan-element outcomes). These all contain the argument INTHEWORLD. All other propositions (those chosen as associates during knowledge activation and plan-elements) receive initial values of zero. This provides for the initial activation of the task description from which activation will flow during the integration process.

This initial vector is multiplied by the task connectivity matrix. The resulting vector is normalized[15] and is again multiplied by the matrix. After each multiplication each element of the resulting vector is compared to the corresponding element of the vector which resulted from the previous multiplication, and an average change is computed over all vector elements. This continues until the criterion representing system stability, a small average change in vector element activations, is achieved.

In this section of the documentation you learn how to use the NETWORK simulation to integrate the matrix created in Section 5, by calculating spreading activation. There is a complete description of the input files and keyboard input that are necessary to perform the integration. You are instructed on how NETWORK uses the most active plan-element with its preconditions met, to make additions to world knowledge. The output files from this module of the program are described and discussed.

The general flow of this section of the program is described in the flow diagram on page 47.

6.1 Input

Only two sources of input are needed to calculate spreading activation, the task connectivity matrix (include1.mat) which was just created and an initial vector file (by convention include1.*vec*) This file contains a vector which represents the initial activation of the task description (INTHEWORLD propositions), all given a 1, and the inactivation of all other network contents, all given 0's. Thus, the task description is the only information initially activated in step 1. The interesting point of the integration phase is to see where activation flows from the already activated task description.

---

[15]Actually, the entire vector is not normalized. The propositions which received initial values of 1 are allowed to keep their resulting values and the remainder of the vector is normalized.

6.1.1 File Input

6.1.1.1 Sample vector file *include1.vec*

```
1
1
1
1
1
1
1
1
1
1
1
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
```

6.1.1.2 The matrix

The task matrix to be used in the calculations is the one created in task connectivity construction. This matrix is saved to an output file when the NETWORK simulation is through with one complete iteration through both the construction and integration phases. Note that is it not possible at this time to perform integration only. Thus, the user is encouraged to complete one construction-integration cycle and then examine the resulting task connectivity matrix before using the vector values. In some cases, the task connectivity matrix might not be what the user expects and this pre-decision inspection will keep the user from continuing on to subsequent "blind" steps.

6.1.2 Keyboard Input

Do you want to produce a spreading activation vector (integrate)? (y or n) *y*
Answer yes to this question to integrate the task connectivity matrix created in Section 5.

If you wish to write output to the terminal enter t

If you wish to write output to a file enter f (t or f) *f*
Enter a t if you want output from this integration to go to the terminal. Enter an f if you want the output saved to a file. In this example we have the output saved to a file.

What file will you write out to? *include1.int*
In this example the output will be saved to a file called *include1.int.*

What is the avg. change criterion for stopping iteration? *.00001*
The integration cycle will continue until the average difference of the corresponding elements of the last two vectors produced is less than or equal to this number. In this example the number chosen is .00001.

To input the vector interactively enter i
To input the vector through a file enter f (i or f) *f*
The initial vector used for integration can be input either interactively by typing an i or through an input file by responding with an f. In this example the initial vector will be input through a file.

Enter the name of the file containing the initial vector? */nsf/work/include1.vec*
In this example the file containing the initial vector is called include1.vec. See Section 6.1.1.1 for a listing of that file.

Integrating task connectivity matrix...Please wait.
No response is necessary to this. The simulation is now integrating to spread activation. If output were going to the terminal instead of a file, we would see the results of each multiplication cycle and you would be asked if you want to continue.

6.2 Process

The initial activation vector is multiplied by the task connectivity matrix. A resulting vector is produced, normalized, and then multiplied by the initial vector. This multiplication repeats until the average change between the resulting vector and the previous one are less than or equal to the amount specified for minimum average change. After the first multiplication, the resulting vector is immediately normalized and multiplied by the matrix again. (Normalization of the vector is done to prevent the activation values from increasing unboundedly over iterations; Rumelhart & McClelland, 1987). The resulting vector is compared to the previous vector by computing the difference between each corresponding pair of vector elements, adding up all of the differences and computing an average by dividing the sum of the differences by the total number of vector elements. As long as this average change measure remains large, activation continues to spread throughout the network. When the average change is minimal (less than .00001 in this case) the network is considered settled. For the first step of the INCLUDE problem it took 4 successive multiplications for the average change to fall below this criterion.

6.3 Output

The output from integration is a series of activation vectors, the last of which is used to select a plan-element for possible firing. When integration output is sent to a file this will be one of the three output files produced by this section. It contains not only the activation vectors, but also the initial vector used to begin the spread of activation. In this example it is called *include1.int* and is listed below.

1.000000
1.000000
1.000000
1.000000
1.000000
1.000000
1.000000
1.000000
1.000000
1.000000
1.000000
1.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000

Iteration 1:

Sum = 47.400000
1.00000 1.00000 2.60000 2.20000 2.60000 3.00000 3.80000 3.80000
4.20000 3.80000 4.20000 1.40000 0.00844 0.00844 0.08439 0.03376
0.00000 0.05907 0.00000 0.15612 0.03376 0.03376 0.03376 0.00000
0.05907 0.25316 0.02532 0.21097

Iteration 2:

Sum = 47.286076
1.00338 1.00338 2.83629 2.28608 2.85316 3.23629 4.12236 5.35105
5.86245 5.00759 5.86245 1.40000 0.00864 0.00864 0.09530 0.04370
0.00000 0.06283 0.00000 0.15980 0.03526 0.03598 0.03526 0.00000

0.00300 0.26279 0.03216 0.21665
Average change in this iteration: 0.00660

Iteration 3:

Sum = 47.192613
1.00346 1.00346 2.82965 2.28905 2.84396 3.22965 4.11870 5.39189
5.90184 5.04033 5.90184 1.40000 0.00866 0.00866 0.09520 0.04382
0.00000 0.06316 0.00000 0.16019 0.03546 0.03631 0.03546 0.00000
0.00000 0.26376 0.03203 0.21730
Average change in this iteration: 0.00038

Iteration 4:

Sum = 47.193991
1.00346 1.00346 2.82927 2.28934 2.84331 3.22927 4.11861 5.39610
5.90582 5.04368 5.90582 1.40000 0.00866 0.00866 0.09517 0.04382
0.00000 0.06317 0.00000 0.16019 0.03546 0.03632 0.03546 0.00000
0.00000 0.26377 0.03201 0.21730
Average change in this iteration: 0.00001

6.4 Decision about a Plan-element for Action

The final vector calculated in Section 6.2 is used by this module to select the most active plan-element. The simulation evaluates the activation values of plan-elements by selecting the plan-element with the highest corresponding vector value. This is considered the most highly activated plan-element. NETWORK checks whether its preconditions exist in the world proposition list. If the preconditions are included in the world list, NETWORK adds this plan-element's outcomes to the world proposition list and the plan-element is considered fired. If the most active plan-element's preconditions do not exist in the world list, the simulation then considers the next most active plan-element until a plan-element is found that has all its preconditions existing in the world proposition list. In the case of a tie in activation, each proposition name and number will be listed and the program will allow manual selection of the plan-element to fire. In most cases plan-element selection should be according to some predefined resolution procedure, or one should be selected randomly.

6.4.1 Input

Do you want the program to select the plan that can fire? (y or n) *y*
Answer y to this prompt if you want the simulation to automatically select the plan-element that should fire.

The plan which has the highest activation and all preconditions met is:
(L7 ((FIND ?FILE^LETTER))
((AT-LEVEL SYSTEM) (EXIST ?FILE^LETTER))

((KNOW ?FILE^LETTER LOCATION)))
Its outcomes will be added to the in the world list.

The following outcomes of the selected plan L7 will be added to the world knowledge:
(P9000(KNOW ?FILE^LETTER LOCATION))
The outcome field propositions are automatically numbered beginning with the number 9000, the argument INTHEWORLD is added to them, and they are added to the proposition list for the next step.

Enter new proposition output filename: *nsf/work/include2.out*
Enter the name of the file in which you want the proposition output from this run stored. Remember that after this file is checked by the user for inconsistent world knowledge it will be used as input into the next iteration of the simulation. By convention this file is named <*task*><*stepnumber*>.out. It is the second of the three output files produced here.

Do you want an activation record sheet? (y or n) *y*
Reply y if you would like an activation record sheet. The activation record sheet is a worksheet containing a list of the input file propositions and a space that can be used to manually record activations for each element in the vector.

Enter an activation sheet filename: *nsf/work/include1.act*
Enter the filename for output of the activation record sheet.

NIL
This is the end of the first iteration through the construction-integration simulation. The lisp program returns NIL to indicate that it is finished.

6.4.2 Output Files

Three main files are produced by this section of the program, two of which have already been mentioned. One, called *include2.out* will become input to the second iteration of the construction-integration model. This output file must be checked for conflicting world knowledge before it is used as input to the next run.[16]

Example file *include2.out*:

```
((P1(EXIST ADDRESS *P44 INTHEWORLD))
(P2(EXIST LETTER *P39 INTHEWORLD))
(P3(EXIST ?FILE^LETTER *P9 *P11 INTHEWORLD))
(P4(EXIST ?TEXT^LETTER *P8 *P10 INTHEWORLD))
(P5(EXIST ?TEXT^ADDRESS *P48 INTHEWORLD))
```

[16]Conflicting knowledge might come about due to NETWORK's lack of semantic knowledge. If a proposition becomes negated, for example, as the result of a plan-element firing, NETWORK will not notice this, the user must.

```
(P6(AT-LEVEL SYSTEM ?FILE^LETTER *P11 *P48 INTHEWORLD))
(P7(ISA ?TEXT^LETTER ?FILE^LETTER *P11 *P6 INTHEWORLD))
(P8(RQINCLUDE ?TEXT^ADDRESS ?TEXT^LETTER *P4 *P10 @L1 @L2 @L3 @L4 $*L5      @L6
@L7 @L8 @L9 @L10 $*L11 @L12 $*L13 INTHEWORLD))
(P9(RQINCLUDE ?TEXT^ADDRESS ?FILE^LETTER *P11 *P8 @L1 @L2 @L3 @L4 $*L5      @L6
@L7 @L8 @L9 @L10 $*L11 @L12 $*L13 INTHEWORLD))
(P10(OUTC8IN ?TEXT^ADDRESS ?TEXT^LETTER *P4 *P7 @L1 $L2 @L3 $L4 @L5 @L6     @L7
@L8 @L9 @L10 $*L11 @L12 $*L13 INTHEWORLD))
(P11(OUTC9IN ?TEXT^ADDRESS ?FILE^LETTER *P48 *P10 @L1 $L2 @L3 @L4 $*L5 @L6  @L7
@L8 @L9 @L10 $*L11 @L12 $*L13 INTHEWORLD))
(P41(AT-LEVEL SYSTEM INTHEWORLD))
(P9000(KNOW ?FILE^LETTER LOCATION INTHEWORLD))
(P39(SEND LETTER POSTOFFICE))
(P44(IS ADDRESS SHORT))
(P48(IN ?TEXT^ADDRESS ?FILE^LETTER))
(L1(EDIT ?FILE^LETTER))
(L2(CUT ?TEXT^LETTER ?FILE^LETTER))
(L3(PASTE ?TEXT^LETTER ?FILE^LETTER))
(L4(PASTE ?TEXT^LETTER ?BUFFER^ADDRESS))
(L5(TYPE ?TEXT^LETTER))
(L6(EXIT ?FILE^LETTER))
(L7(FIND ?FILE^LETTER))
(L8(READ ?FILE^LETTER))
(L9(ENTER SYSTEM))
(L10(CUT ?TEXT^ADDRESS ?FILE^LETTER))
(L11(PASTE ?TEXT^ADDRESS ?FILE^LETTER))
(L12(PASTE ?TEXT^ADDRESS ?BUFFER^ADDRESS))
(L13(TYPE ?TEXT^ADDRESS))...)
```

Another file is the file containing the vectors from integration. It was presented in Section 6.3 and shows the initial activation vector and those resulting from subsequent vector-matrix multiplications.

The last file produced in this section is the INCLUDE step 1 task connectivity matrix. This matrix file can be examined for link values.

Now that one step of INCLUDE is complete, you are on your way to solving the task. The first step in planning a solution, finding the requisite file, has been achieved. In order to determine whether to repeat the construction-integration cycle, the goal proposition(s) must be checked against the state of the world to see if it has been produced. If it has not, return to the beginning of Section 5 (Section 4 if plan-element variables need to be rebound) to produce a task connectivity matrix for step two of INCLUDE (or any other task). This new matrix can subsequently be integrated and a plan-element selection can be produced for the second step. A goal check is done and this repeats until the plan-element selection of a step produces a proposition matching the goal proposition. Then, the task is done.

7 References

Clark, H. H. & Clark, E. V. (1977). Psychology and Language. New York: Harcourt, Brace, Jovanovich.

van Dijk, T. A. & Kintsch, W. (1983). Strategies of discourse comprehension. New York: Academic Press.

Doane, S. M., Kintsch, W. & Polson, P. G. (1990). Modeling UNIX Command Production: What Experts Must Know. Manuscript submitted for publication. Institute of Cognitive Science, University of Colorado at Boulder, Technical Report 90-1.

Kintsch, W. (1974). The representation of meaning in memory. Hillsdale, NJ: Lawrence Erlbaum.

Kintsch, W. (1988). The role of knowledge in discourse comprehension: A construction-integration model. Psychological Review, 95, 163-182.

Kintsch, W. & van Dijk, T. A. (1978). Toward a model of text comprehension and production. Psychological Review, 85, 363-394.

Kintsch, W. & Keenan, J. M. (1973). Reading rate and retention as a function of the number of propositions in the base structure of sentences. Cognitive Psychology, 5, 257-274.

Mannes, S. M. (1989). Problem-solving as Text Comprehension: A Unitary Approach. Doctoral Dissertation, University of Colorado at Boulder.

Mannes, S. M. & Kintsch, W. (1989). Planning routine computing tasks: Understanding what to do. Manuscript submitted for publication. Institute of Cognitive Science, University of Colorado at Boulder, Technical Report 89-8.

Norman, D. A. & Rumelhart, D. E. (1975). Explorations in cognition. San Francisco, CA: Freeman.

Perfetti, C. A. (1985). Reading ability. New York: Oxford University Press.

Raaijmaker, J. G. & Shiffrin, R. M. (1981). Search of associative memory. Psychological Review, 88, 93-134.

Rumelhart, D. E. & McClelland, J. L. (1987). Parallel distributed processing: Explorations in the microstructures of cognition. Volume 1. Cambridge, MA: MIT Press.

Tulving, E. & Gold, C. (1963). Stimulus information as determinants of tachistoscopic recognition of words. Journal of Experimental Psychology, 66, 319-327.

Turner, A. A. (1987). A vector-matrix multiplication program. Personal communication.

Wheeler, D. D. (1970). Processes in word recognition. Cognitive Psychology, 1, 59-85.

ICS Tech Report
90-13

HP Common Lisp II and the GnuEmacs Editor

October, 1990

1. Introduction 1
2. Xwindows 1
3. Starting GnuEmacs and the Lisp Interpreter 3
3.1 Putting a Lisp File into an Emacs Buffer 3
3.2 Loading a File into the Lisp Interpreter 3
4. Evaluating Buffers, S-expressions and Defuns in the Edit Buffer 4
4.1 Defining a function 4
4.2 Evaluating an s-expression 4
4.3 Evaluating a region 4
4.4 Evaluating an entire buffer 5
4.5 Compiling a function 6
4.6 Data input 7
4.7 Tracing a function 7
4.8 Recovering from a lisp break 7
4.9 Stopping an infinite loop 8
4.10 Summary of lisp interaction key bindings 9
4.11 Super-parens 9
5. Evaluation in the Lisp Interaction Buffer (*lisp*) 10
6. Tags facility 11
7. Useful GnuEmacs Commands 12
7.1 Important GnuEmacs keybindings
7.2 Copying and Pasting Using the Mouse. 12
8. Additional Information 12

1. Introduction

This document is designed to help users with the interaction between Hewlett-Packard Common Lisp II and the GnuEmacs editor. It is a combination of the Help document provided with the clisp environment and helpful hints on GnuEmacs, Xwindows (X11 R3) and HP Common Lisp II. It is recommended that you use this document as a mini-tutorial for this lisp environment. The document will be much easier to follow if you actually execute the examples on a workstation that has GnuEmacs and lisp installed on it.

The file clisp.el establishes a set of key bindings and functions to support one or more Common Lisp processes running either locally or remotely. Clisp was written at Cornell University as a Cornell Apprentice Project (CAP). It was designed to provide a user friendly environment for lisp interpreters such as HP Common Lisp II. Clisp provides editing capabilities and X11 interaction for Lisp. Lisp s-expressions can be evaluated and lisp code loaded and executed while running Xwindows and GnuEmacs.

In this document C- followed by one or more characters represents the ctrl (control key). M- followed by one or more characters represents the meta key. On the Hewlett-Packard 9000 Series computers you can use either the extend char key (immediately left of space bar on HP keyboard) or the escape key as the meta key. > represents the lisp interpreter prompt, <CR> represents a carriage return and ▮ represents the cursor.

2. Xwindows

In order to use this lisp environment to its full potential be sure to start xwindows before using the emacs editor. This is done by typing
your-shell-prompt> x11start
to your shell prompt.
There are several initialization files that allow users to customize their xwindows environment. These files include .Xdefaults, .x11start, .keymap, .hpwmrc and .mwmrc. For more information about these files please contact your hp system administrator (email hptrouble@boulder.)

After X windows has started you should have at least one window with a shell prompt in it. If you want to make this window your active window click the left mouse button on it. If you want to create a new X window click and hold down the right mouse button on the gray background. This will display a menu of different Xwindows options. Select the New Window option from the menu and a new window will appear on your X background. When you are done with Xwindows and before you can logout you will need to stop X by pressing the control shift and reset (upper left function key on the HP) keys simultaneously.

3. Starting GnuEmacs and the Lisp Interpreter

3.1 Putting a Lisp File into an Emacs Buffer

Editing any file in lisp mode will cause HP Common Lisp II to be started automatically. If you edit a file with the extensions .lisp, .l or .lsp emacs will automatically start in lisp mode. For example:

emacs example.l

typed into any window will start up emacs in lisp mode. A new window will become visible when you start emacs. You can adjust the size of this window by holding the left mouse button down on the lower right corner of the window and dragging the mouse. If the file example.l exists it will be placed in a buffer and displayed in your emacs window, if not emacs will display an empty buffer. This empty buffer can be used to input and evaluate lisp code. Code that is typed into this empty buffer can be saved to a file with the emacs save file command C-x C-s.

Once your file is visible in the emacs buffer you can split the emacs window and make the lisp interaction buffer (*lisp*) visible by typing C-c l.

If you have already started emacs for editing a non-lisp file and then decide to load the lisp interpreter M-C-l will create a new lisp buffer and start the lisp interpreter

While editing a file in lisp mode you can switch windows by clicking the mouse button on the desired window or you can use the commands below. These commands are especially useful if the buffer you want to work with is not currently displayed in a window:

C-c l switches the cursor to the *lisp* buffer
C-c e returns the cursor to the last edited buffer of lisp code

3.2 Loading a File into the Lisp Interpreter

You now have started emacs along with the lisp interpreter and you have your file of lisp code visible in your emacs buffer. Even though emacs is aware of the contents of your lisp file, your lisp code has not been evaluated by the lisp interpreter. This means that you can not execute any of the functions defined in your lisp file. The next step in the process of running your lisp code is to get the lisp interpreter to evaluate the definitions in your file. This is accomplished by loading the file into the lisp interpreter with the lisp load function. For example to load the file example.l (assuming that the file example.l exists in your home directory) do the following:

1. In the *lisp* buffer (also know as the lisp interaction buffer) type the following at the prompt:

   > (load "example.l") <CR>

   NOTE: If the file you are loading is not in your current directory you must supply the full UNIX path name to that file (/users/mroushey/example.l)
2. Lisp will load the file (example.l in this example), evaluating the code and will reply with:

;;; Loading source file "example.l"

#P"/users/mroushey/example.l"

3. The code in example.l is now defined by the lisp interpreter and you can proceed to use it by evaluating function calls to the functions in example.l.
4. If you later make changes to code in this file you will need to redefine those changes in the lisp interpreter. One way to accomplish this redefination is to save the changes to your file with C-c C-s (emacs save command) and reload the file. You can also use any of the definition facilities described in section 4.

4. Evaluating Buffers, S-expressions and Defuns in the Edit Buffer

Now that you have the lisp interpreter loaded and a buffer for lisp code you can pass lisp code to the interpreter for evaluation.

4.1 Defining a function

Placing the cursor on a defun and typing C-c d causes the function to be defined or redefined by lisp. The lisp interaction buffer will respond with the name of the function just defined or an error if the definition is incorrect (see section 4.7).
For example to define the function factorial:

```
(defun factorial (n)▮
  (cond ((zerop n) 1)
        (t (* n (factorial (- n 1))))))
```

place the cursor on the first line of the defun and type C-c d. The lisp interpreter will respond with:

```
>FACTORIAL
```

in the *lisp* buffer.
C-c C-d will evaluate a defun and switch to the lisp interaction buffer.

4.2 Evaluating an s-expression

Placing the cursor at the end of an s-expression (sexpr) and typing a C-c s will cause that sexpr to be evaluated by the interpreter. The lisp interaction buffer will respond with whatever output is generated by evaluation of the s-expression. Remember that a lisp s-expression can be any legal lisp expression. For example to evaluate:

```
(factorial 3)▮
```

type C-c s. Lisp will respond with:

```
>6
```

in the *lisp* buffer. C-c C-s will evaluate an s-expression and switch to the lisp interaction buffer.

4.3 Evaluating a region

If you have a contiguous section of code that you want evaluated by lisp use C-c r to evaluate the region. To define a region in emacs you first set a mark by typing C-@ or M-@ at one end of the region (contiguous section). You then move the cursor to the other end of the region and type C-c r. This will send the entire area between the mark and the cursor to lisp for evaluation. To evaluate the example region below:

1. Place the cursor on the blank line before the the definition of sum-lists and type C-@ or C-2.
2. Move the cursor to the end of the function definition for check-num.
3. Type C-c r

```
(defun sum-lists (x y)
 (cond ((check-num (append x y))
        (mapcar '+ x y))
       (t(write "ERROR:  Numeric lists only")
         (terpri))))

; example function definition
(defun check-num (num-list)
 (let ((answer t))
   (do ((num num-list (cdr num)))
        ((or (null num)(null answer))answer)
     (if (numberp (car num))
         (setq answer t)
         (setq answer nil)))))█
```

The lisp interpreter will respond with:
; ; ; Loading source file "/users/mroushey/.clispTMP/lspa01501.lsp"
Emacs has created a temporary file that contained the region marked in your buffer. This file was loaded into lisp for evaluation. C-c C-r will evaluate a region and switch to the lisp interaction buffer.

4.4 Evaluating an entire buffer

If you want to have your entire buffer evaluated by lisp, place your cursor anywhere in the buffer and type C-c b. This buffer evaluation actually reloads the lisp file in you buffer into the lisp interpreter. If you place your cursor anywhere in the buffer example.l:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This is an example file of lisp code used in the document
; HP Common Lisp II and the GnuEmacs Editor
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; example function definition
(defun factorial (n)
 (cond ((zerop n) 1)
       (t (* n (factorial (- n 1))))))

; example s-expression
(factorial 3)

; example function definition
(defun sum-lists (x y)
```

```
 (cond ((check-num (append x y))
        (mapcar '+ x y))
       (t(write "ERROR:  Numeric lists only")
        (terpri))))

; example function definition
(defun check-num (num-list)
 (let ((answer t))
   (do ((num num-list (cdr num)))
       ((or (null num)(null answer))answer)
    (if (numberp (car num))
        (setq answer t)
        (setq answer nil)))))

; example s-expressions
(setq a '(1 3 5))
a
(setq b '(2 4 6))
b
(sum-lists a b)

(defun squared ()
 (do ((number nil))
    ((eq number 'q))
   (format t "Enter number >")
   (setq number (read))
   (if (numberp number)
       (format t "~%The square is ~D~%" (* number number)))))
```

and type C-c b. Lisp will respond with:

; ; ; Loading source file "/users/mroushey/.clispTMP/lspa01437.lisp.

When you evaluate a buffer emacs creates a temporary file that contains that buffer and loads it into lisp. ```C-c C-b will evaluate your current buffer and switch to the lisp interaction buffer.

4.5 Compiling a function

If you want to compile a lisp function, place the cursor anywhere on the the defun and type C-c c. For example in example.l if you place the cursor on the defun factorial:

```
(defun factorial (n)
 (cond ((zerop n) 1)
       (t (* n (factorial (- n 1))))))
```

and type C-c c, lisp will respond with:

;;;Compiling Function FACTORIAL. . . assembling . . . emitting . . . done.
C-c C-c will compile a function and switch to the lisp interaction buffer.

4.6 Data input

When you need to respond to a prompt from your lisp code, type the response into your edit buffer and type C-c C-i. C-c C-i sends the line that your cursor is sitting on to the lisp interpreter as input to a lisp read function. For example you call the function squared by evaluating the following with C-c s:

```
(squared)▮
```

The lisp interpreter responds with

```
>Enter number >
```

in the *lisp* buffer. To enter the number 5 type 5 on a line by inself in you edit buffer and then type C-c C-i while you have the cursor on the same line

```
5▮
```

The response from lisp is

```
> 25
```

4.7 Tracing a function

You can place a trace on any lisp function for debugging purposes. To trace a function place the cursor on the defun and type C-c t. When you are finished with your trace C-c u shuts off tracing of the function (untrace). To trace the function factorial place the cursor on any line in the function and type C-c t:

```
(defun factorial (n)▮
  (cond ((zerop n) 1)
        (t (* n (factorial (- n 1))))))
```

The lisp interpreter responds with:

```
> (FACTORIAL)
```

any subsequent calls to factorial will generate trace output in the *lisp* buffer.

```
> 1  Enter FACTORIAL 3
| 2  Enter FACTORIAL 2
|    3      Enter FACTORIAL 1
|    |      4      Enter FACTORIAL 0
|    |      4      Exit FACTORIAL 1
|    3      Exit FACTORIAL 1
| 2  Exit FACTORIAL 2
1  Exit FACTORIAL  6
6
```

C-c C-t (C-c C-u) will trace (untrace) a function and switch to the lisp interaction buffer.

4.8 Recovering from a lisp break

Like most lisp interpreters HP Common Lisp II provides a break package for error processing. When the lisp interpreter encounters an error in the code it is

evaluating, lisp will switch to the break package in order to provide you with more information about the error.
When a break occurs you will be notified in the *lisp* buffer. Lisp will give you information on the type of break and where it occurred in the evaluation process. A list of options for exiting the break will also be displayed. To execute these options type one of the following into the lisp editor buffer: C-c :a to abort the break and return to the top level of lisp, C-c :c to continue with the evaluation after additional information as been given to the interpreter, and C-c :e to move up one level in the break package.
For example you define the following function (see section 4.1):

```
(defun factorial (n)
  (cond ((zerop x) 1)
        (t (* n (factorial (- n 1))))))
```

If you then call that function(see section 4.2):

```
(factorial 3)
```

You will see the following in your *lisp* buffer:

```
>>>ERROR:  X has no global value
SYMBOL-VALUE:
        Required arg 0 (S):  X
:A      Abort to Lisp Top Level
:C      Try Evaluating X again
->
```

From the example.l buffer type C-c :a
Lisp will respond with:

```
-> :A
->
```

Back to Lisp Top Level

4.9 Stopping an infinite loop

If you call a lisp function that has no termination condition you will need to break the function in order to terminate evaluation. This is accomplished by typing C-c C-c in the *lisp* buffer. For example evaluating the following region with a C-c r will produce an infinite loop (non-terminated recursion):

```
; example of function with non-terminated recursion.  Be careful with this one!
(defun forever ()
  (write "never ending")
  (forever))
```

To terminate the infinite loop click the mouse button on the *lisp* window and type C-c C-c. Evaluation of the function will end and the lisp interpreter will go into break mode (see section 4.7).

4.10 Summary of lisp interaction key bindings

To pass code from GNU to lisp:

C-c d	evals current defun in last lisp process
C-c s	evals last sexpr in last lisp process
C-c r	evals current region in last lisp process
C-c b	evals current buffer in last lisp process
C-c c	compiles current defun in last lisp process
C-c t	traces current defun in last lisp process
C-c u	untraces current defun in last lisp process
C-c C-i	Input a line of data to lisp
C-c C-a	beginning of current defun gives error message if parens are unbalanced
C-c C-e	end of current defun
C-c :a	aborts lisp break to top level
C-c :c	continues lisp break
C-c :e	aborts lisp break one level
C-c m	shows Common Lisp macro expansion of current form
C-c a	shows Common Lisp arglist for current function
M-p	set package for current buffer
C-c =	interactive definition facility (prompts you for function name and arguments creating a template function definition)
M-q	reindents current comment or defun. Place the cursor in the body of the defun or on the comment.
M-C-q	reindents current buffer
C-c ;	inserts a separator line of semicolons
C-c]	checks your lisp file for mismatched parens

4.11 Super-parens

Note that the "[" and "]" characters can be used as "super-parens" in either mode. A "]" closes as many open "(" exist up to and including an open "[". If no open "[" exists, "]" closes up to the top level. The square brackets are replaced by the appropriate number of "(" and ")" in the buffer, since Common Lisp doesn't understand super-parens. N.B.; To insert explicit square brackets, they must be prefaced by C-q (the emacs quote command). For example placing the cursor at the end of the function definition below and typing a "]" will close all the open parentheses:

```
(defun squared ()
  (do ((number nil))
     ((eq number 'q))
    (format t "Enter number >")
   (setq number (read))
   (if (numberp number)
        (format t "~%The square is ~D~%" (* number number▮
```

The emacs editor will replace the "]" with the correct number of parentheses.

5. Evaluation in the Lisp Interaction Buffer (*lisp*)

The Lisp interaction buffer (*lisp*) is actually the lisp interpreter loaded into an emacs window. You can evaluate any valid lisp s-expression by typing the s-expression followed by a <CR> directly into this buffer (see section 3.2 for an example). Remember that it is difficult to create a file of lisp code from this buffer, so it is recommended that you use this buffer to evaluate code that is not necessary to save.

While typing to an lisp process buffer you can switch windows by clicking the mouse button on the desired window or you can use the commands below. These commands are especially useful if the buffer you want to work with is not currently displayed in a window:

C-c e returns the cursor to the last edited buffer of lisp code
C-c l switches the cursor to the *lisp* buffer

There are some "ksh"-like features available in the lisp buffer (*lisp*). These features allow you to re-execute commands in the *lisp* buffer without retyping the entire command:

C-c h	show history. This creates a *history* buffer containing all commands executed in the *lisp* buffer. The commands are numbered 0-n (where n is total number of commands typed in). The command labeled 0 is the most recent command and the command labeled n is the first command entered.
C-c C-p	previous command in history list. C-c C-p allows you to recall each command executed in order of the most recent command first. The the previously executed commands are placed at the > prompt in the lisp buffer where they can be modified or re-executed.
C-c C-n	next command in history list. C-c C-n allows you to move forward through your history list bringing up previously executed commands most recent last.
C-c C-a	position cursor at previous prompt.
C-c C-u	flush current input. C-c C-u removes the current command from the prompt line.
C-c C-r	search backwards through history list for specific expression. If the search is successful the matched command is placed on the current prompt line.
C-c C-s	search forward through history list for specific expression. If the search is successful the matched command is placed on the current prompt line.
C-c C-d	flush output history. C-c C-d clears *lisp* buffer of most output.

Finally, while running in an lisp buffer, if you position the point after a previous input to lisp and hit return, the old input will be copied to the end of the buffer and resubmitted to lisp.

6. Tags facility

The GNU emacs tags facility is used to cross index your source code. The tags facility creates a tag file which is a database of all function names contained in one or more lisp files. The database can then be used to find function definitions and corresponding function calls in those files. This is an especially useful tool in debugging and editing modular programs contained in multiple files.

The tag file is created outside of emacs by typing the following to the shell prompt:

```
etags example.l test.1
```

This creates a file in your directory called TAGS which contains references to functions defined in both example.l and test.l. Once your tag file is created you can load a tag file by executing any of the commands listed below from your edit buffer.

Special bindings to support this feature include:

C-c .	finds defun for current function in other window
C-c ,	looks for next matching defun in same window
M-.	finds defun for current function (std GNU)
M-,	looks for next matching defun (std GNU)
C-c ?	lists files indexed by current tag table
C-c /	recomputes tag table

7. Useful GnuEmacs Commands

7.1 Important GnuEmacs keybindings:

C-h	Help
C-h t	Emacs tutorial
C-x C-f	Find file to edit and put it in a buffer
C-x C-w	Write buffer to a file (prompts for name)
C-x C-s	Save buffer to file
C-x C-b	Show a list of all buffers
C-x b	Switch buffer
C-@	Set mark
C-w	Kill region (cut)
C-y	Yank kill (paste)
C-x] or C-v	Forward page
C-x [	Backward page
C-x u	Undo previous command(s)
C-x o	Switches to other window
C-x 1	Deletes other windows
C-x 2	Splits current window
C-x C-c	Exit emacs and lisp

7.2 Copying and Pasting Using the Mouse.

You can also use the mouse to copy and paste text between windows. Click the left mouse button at the beginning of the region to be copied, then move to the end of the region to be copied and click the right mouse button. To paste the text simply click the middle mouse button (both right and left buttons on a two button mouse) in the window where you want the text to appear.

8. Additional Information

HP Common Lisp II provides no online help for predefined Lisp functions.

There are several sets of documentation on HP Common Lisp II available for loan.

A standard .emacs file is available for your use in customizing your emacs environment. A copy of this standard .emacs file can be obtained by contacting hptrouble (email hptrouble@boulder.)