An Approach to the Organization of Knowledge for the Modelling of Conversation

by

Gordon I. McCalla

Department of Computer Science
University of Toronto
Toronto, Ontario, CANADA

Technical Report 78-4

Based on the author's thesis of the same title,
submitted in June, 1977
as partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

February, 1978

Department of Computer Science
University of British Columbia
Vancouver, B. C., CANADA

ABSTRACT

This report describes an approach to modelling conversation. It is suggested that to succeed at this endeavour, the problem must be tackled principally as a problem in pragmatics rather than as one in language analysis alone. Several pragmatic aspects of conversation are delineated and it is shown that the attempt to account for them raises a number of general issues in the representation of knowledge.

A scheme for resolving some of these issues is presented and given computational description as a set of (non-implemented) LISP-based control structures called |LISP. Central to this scheme are several different types of object that encode knowledge and communicate this knowledge by passing messages. One particular kind of object, the pattern expression (|PEXPR), turns out to be the most versatile. |PEXPRs can encode an arbitrary amount of procedural or declarative information; are capable, as a by-product of their message passing behaviour, of providing both a context for future processing decisions and a record of past processing decisions; and make contributions to the resolution of several artificial intelligence problems.

Some examples of typical conversations that might occur in the general context of attending a symphony concert are then explored, and a particular model of conversation to handle these examples is detailed in |LISP. The model is goal oriented in its behaviour, and, in fact, is described in terms of four main goal levels: higher level non-linguistic goals; scripts directing both sides of a conversation; speech acts guiding one conversant's actions; and, finally, language level goals providing a basic parsing component for the model. In addition, a place is delineated for belief models of the conversants, necessary if utterances are to be properly understood or produced. The embedding of this kind of language model in a |LISP base yields a rich pragmatic environment for analyzing conversation.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

This technical report is based on my Ph.D. thesis recently completed at the University of British Columbia. Needless to say, both the report and the thesis have benefited from the comments and criticisms of others. First, I would like to thank my supervisor, Richard Rosenberg, who has lent his ear to my harangues and complaints and who has guided me through the long and painful evolution of this research. The other members of my Ph.D. committee, Alan Mackworth, Ray Reiter, Bary Pollock, and Bernie Mohan have also been of great help in various discussions and in their comments on earlier thesis drafts. Finally, I would like to thank the external examiner, Nick Cercone, for his thorough analysis - the technical report, especially, owes much to his comments.

In many stimulating coffee break conversations, and elsewhere, my fellow graduate students at UBC have been both useful critics and sources of ideas: I am particularly grateful to Rachel Gelbart, Bill Havens, Peter Rowat, Michael Kuttner, Brian Funt, and Jim Davidson. In addition, the unflagging enthusiasm of the Snarfler, L. R. Floyd, must be acknowledged.

Because this work was finished while I was teaching at the University of Toronto, a not insignificant contribution has been provided by my colleagues and friends there. For many all night discussions that provided several insights, my thanks to Hector Levesque and to Pat Levesque for putting up with us. My appreciation also to John Mylopoulos, Ray Perrault, Phil Cohen, Lou Melli, Dick Peacocke, and Nick Roussopoulos.

CHAPTER I
Introduction

This report is concerned with the computational modelling of natural language conversation. It is suggested that a prerequisite to the study of conversation is determining how to represent the vast amount of world and linguistic knowledge that is required in such an undertaking. A method of representation has been devised that allows much of this knowledge to be encoded reasonably conveniently in modules called pattern expressions. The representation problem then becomes primarily the problem of how to organize knowledge into appropriate pattern expressions. A basic organizational philosophy for conversation is described and is detailed by showing how it can be used to handle three conversations which might occur in the scenario of attending a symphony concert.

Several aspects of language and representation are at least touched upon in this report. There is a re-categorization of linguistic knowledge that tends to meld such traditional distinctions as those dividing syntax, semantics, and pragmatics, and also the division between linguistic and non-linguistic knowledge. A suggestion is made as to how to combine knowledge from a script, knowledge about intention and purpose, knowledge about the conversants, and linguistic knowledge. The necessity for explaining or excusing errors, the need for a context mechanism, and the usefulness of expectation in guiding the processing of natural language is pointed out. Finally both task-oriented and non-task-oriented dialogues are discussed.

Although all of these aspects are encountered, none of them is resolved completely; in some cases the analysis is only in its preliminary stages. In general I have not been as concerned with finding detailed solutions to particular problems as I have been with trying to accomodate a wide variety of phenomena, at the expense of depth in places. This seemed a necessary price

to pay in order to look at the problem in some generality.

1.1 Background: Some Trends

The approach to conversation taken here is based on three trends which I perceive in the study of language. The first trend is the ever broadening focus of attention of linguistics. After Bloomfield (1933), linguists felt constrained to focus their attention on a relatively narrow subset of linguistic phenomena, with most attention being paid to phonetics and syntax. The Chomskian revolution (see Chomsky (1957) for the opening shots) brought an impressive new descriptive power to linguistics. Of particular importance were the notions of "infinite capacity with finite means" (i.e. generating an infinite number of sentences using phrase structure rules and transformations) whereas previous corpus-based analyses had seemed to be attempting the impossible task of collecting all valid sentences; the competence / performance distinction, allowing knowledge about language itself to be separated from the vagaries of people's actual use of language; and the discovery of the underlying deep structure similarity of many seemingly different surface descriptions.

In the mid-sixties the Chomskian revolution itself came under attack. Linguists such as G. Lakoff (1971) started what became known as the generative semantics movement which attempted to point out flaws in the Chomskian view of language and which expanded the scope of the linguistic endeavour with the hope that this broader view would overcome the problems. Generative semanticists see a more central role for semantics in the study of language to account for things (such as scoping phenomena involving quantifiers and negation) that directly affect the surface structure without any intermediate syntactic phase. Chomsky (1971) has responded to these criticisms with some modifications to his theory and with the claim that generative semantics is a "mere notational variant" of his theory. I don't want to get embroiled in this dispute, my point

being merely that in recent times the semantic level is coming more and more into play (see Leech (1974) for a comprehensive description of the relative merits of current semantic theories).

The second relevant trend to the research reported here is the growing realization that the borders separating the various levels of linguistic description are not rigorously defined. In the last paragraph I mentioned that semantic phenomena can influence surface structure without going through deep structure. It is also the case that phonetics sometimes seems to influence semantics. Thus, discovering pronoun references appears to have important semantic aspects, but the process is not strictly semantic. For example, "George always wanted to be a guitarist but it wasn't the instrument that suited him best." is acceptable, but "George always wanted to be a flautist, but it* wasn't the instrument that suited him best." is not, at least partially because the word "flautist" is phonetically more distinct from "flute" than "guitarist" is from "guitar" (this example is based on observations by Lakoff and Ross (1972)).

The second trend has been emphasized in recent times by the appearance of word based case theories of language (Fillmore (1968), Chafe (1970)) which place linguistic information in case frames associated with words. Each such case frame is responsible for "filling in slots" for that word appropriate to the context in which the word appeared. Computer based case models (e.g. Schank (1972), Taylor and Rosenberg (1975), Martin (1975)) have tended to deepen case frames so they can do semantic and even pragmatic processing as well as the more syntactic things suggested by the earlier case theories.

The third major line of development has been the increasing interest in models of language which treat language as it is used rather than as some ideal grammatical abstraction. This is, of course, quite contrary to the competence / performance distinction which has enabled language to be studied in relative

isolation from the real world. However, I believe this isolation has generated many false issues. One such is the attempt to categorize sentences as "grammatical" or "ungrammatical" in some absolute sense. A more useful decision procedure might concentrate instead on what situations would make a sentence acceptable or unacceptable. Another example concerns the debate that often arises over whether two sentences are synonymous when, of course, at some level no two sentences mean the same. Once again, a determination of the situations in which they can be considered synonymous seems to be a more useful approach.

The attempt by philosophers of language to view language in terms of its intended effect on the hearer seems to avoid many of these problems by focussing on the purpose of language: the communication of ideas. Hence, slight deviations in the surface structure which don't affect the meaning are not important; two sentences are synonomous if their effect on the hearer is identical. Of particular interest here are the Gricean approaches of giving "rules of conversation" (1968) and giving intentional definition to utterances (1959), and the speech acts theory (Austin (1962), Searle (1969)).

The trend to viewing language in a more natural setting has been evident in artificial intelligence as well. Earlier systems severely restricted the domain of study to question answering (e.g. Schwarcz et al (1970)), blocks worlds (e.g. Winograd (1972)), baseball statistics (Green et al (1963)), family relationships (Lindsay (1963)), belief systems (Tesler et al (1968)), and so on. But more recently ever broader views of language have been taken, so that Schank's system (1972), already quite comprehensive, when extended by scripts (Schank and Abelson (1975)) became even more general. Charniak's (1972) work on stories has been extended to full scale frame descriptions (Charniak (1975)) of language scenarios. Bruce (1975) has been concerned with large paradigms of social action as they apply to language. P. Cohen (1978) studies several conversational scenarios using a methodology

based on a computational description of speech acts. Winograd (1976, 1977) is concerned with formulating a general, pragmatics centred, model of language. This search for generality is a manifestation of the third trend: trying to view language in use.

These three trends are not diverging; rather, they seem to be coming together into a single viewpoint: language should be studied as it is used, with semantic and pragmatic information being more central than the more purely surface aspects. However, this shouldn't preclude knowledge from whatever level being applied when relevant. These are the reasons why conversation, a domain in which language is used as naturally as possible and a domain in which semantic and pragmatic considerations are of utmost importance has been chosen for study.

1.2 Issues and Limitations

It is probable that fully general computational models for conversation are currently intractable. But under certain restrictions progress can be made. The first restriction is to only consider a particular conversational situation. A "concert scenario" is proposed which illustrate both the importance of conversation as a domain and also narrows the scope of the project. The scenario involves conversations which would take place during the events surrounding a symphony concert. Three particular conversations have been chosen for analysis:

(i) a conversation between a ticket seller and a concert patron who wants to buy a ticket to the concert;

(ii) a conversation between a bartender and a concert patron who wants to buy a drink during the intermission of the concert;

(iii) a conversation between two concert patrons who unexpectedly meet during intermission.

Conversations (i) and (ii) are "task-oriented" (Deutsch (1974)) and hence somewhat predictable. Conversation (iii), which is

much less predictable and hence much harder to analyze, has been given a rather cursory analysis.

The study of conversation has been further limited by considering only a few of the possible issues. In line with the focus on the pragmatic and semantic levels the following have been of central concern:

(i)-L whether world and linguistic knowledge can be effectively combined, and in particular whether language can be viewed as an activity like any other;

(ii)-L how the goals of a conversant affect what he says and how he understands;

(iii)-L how the knowledge a conversant has about the other conversants affect what he says and how he understands;

(iv)-L how the conversant is able to focus on the relevant aspects and ignore the irrelevant aspects of any conversational scenario.

Other potentially relevant issues have not been considered: for example, a detailed analysis of linguistic surface phenomena; an adequate treatment of the problem of generation; the phonetic or morphemic aspects of language; the problem of reference; the problem of handling massively unexpected utterances (or other surprises); and so on. However, since most of these problems unavoidably arise, suggestions as to possible solutions are often sketched out.

A third aspect which helps to make conversation a reasonable domain to study is the methodology chosen to test out ideas: building a computer model of the behaviour of one participant in a dialogue. While simplifying the problem by eliminating half of each dialogue, this viewpoint-dependent approach means that the issues of goal direction and conversant modelling become particularly relevant, thus further focussing the research.

The attempt to build a computer model, however, raises a number of artificial intelligence issues. In particular the following are of crucial relevance:

(i)-R the conglomeration of procedural and declarative

information of various sorts into one place, since the model obviously has to both know things and know how to do things;

(ii)-R the problem of accessing this knowledge; that is, how and when to search for information, when to inherit general information (see Levesque (1977) for example), when to make inferences, how to attach procedures (Winograd (1975));

(iii)-R the necessity for some sort of context mechanism (of particular relevance to issue (iv)-L);

(iv)-R the need to keep a record of processing decisions, not only to allow the model to keep track of what has just been said but also to allow it to reason about its own behaviour (see issue (ii)-L particularly);

(v)-R the incorporation of robustness, so that when something fails, appropriate explanations, excuses, or failure processing can be undertaken;

(vi)-R the standard artificial intelligence issues of complexity and combinatorial explosion.

1.3 What's Proposed?

An investigation into the issues raised in the last section tends to be cyclical, with language issues raising representation issues which reflect back into language and so on back and forth. Therefore, this report does not represent some "cast in concrete" final version of my ideas, but is much more a snapshot of my current thinking.

I would like to indicate the major concepts contained in the report. The discussion is divided into two parts: concepts of representation and concepts of language.

1.3.1 Concepts of Representation

A prerequisite to the study of conversation is the representation of knowledge. The representation scheme proposed here has the following main characteristics:

(i) It is modular, allowing many different kinds of objects to co-exist together. Objects are opaque to one another

and can communicate only by passing messages. Such modularity, it is hoped, will help solve some of the complexity issues (issue (vi)-R).

(ii) The most interesting such object, called a pattern expression (|PEXPR) is roughly based upon the frame idea of Minsky (1974). Most domain knowledge is represented in such objects as patterns. Since these patterns can be static or can contain certain "active" macro elements, either procedural or declarative information can be encoded (issue (i)-R).

(iii) Messages to a |PEXPR are also patterns that are handled by matching them against patterns in the |PEXPR. Since message passing is defined for all kinds of patterns, in particular procedural and declarative information can be accessed uniformly (issues (i)-R and (ii)-R).

(iv) If a message pattern cannot be matched in a |PEXPR, failure to match processing (associated with patterns of that type) can take place. Such failure processing can involve trying to "inherit" the pattern, performing appropriate inferences to discover the pattern, or in the worst case giving up completely (this addresses issues (ii)-R and (v)-R).

(v) A by-product of message passing is the creation of an activation record to which temporary variables and other local effects of the message passing are restricted. This activation record is called an execution instance and is a pattern expression like any other (and hence able to be accessed in identical fashion to other |PEXPRs (issue (vi)-R)).

(vi) A pattern expression may need to communicate with another pattern expression as it handles a message. Chains of messages can be set up this way with corresponding chains of execution instances. They form a dynamic environment (akin to that of ALGOL or LISP) called the execution environment which turns out to be a very useful focussing and context mechanism (issue (iii)-R, issue (iv)-L, as well as allowing the discovery of current goals (relevant to issue (ii)-L).

(vii) Execution instances are not automatically removed after a message is answered. Instead, they stay around and

chains of them are consequently preserved. Such venerable execution environments can be accessed if the details of what went on in the past are needed. They thus give the model a sort of episodic memory (issue (ii)-R).

1.3.2 Concepts of Language

The easiest way to describe the approach to language taken in this research is in terms of "level of goal", from higher level extra-linguistic goals through lower level goals that are called in to understand particular parts of an utterance. Each such goal can be more or less separated from other goals, and each is therefore encoded as a pattern expression. Since goals invoke subgoals arbitrarily, it is sometimes difficult to classify them precisely; they do seem, however, to fall into four main categories: non-linguistic goals, scripts, speech acts, and language level goals.

(i) non-linguistic goals: Goals at this level undertake significant plans of action such as attending a concert, buying a ticket to the concert, etc. Not primarily concerned with language, they do, however, know enough to call in linguistic subgoals when appropriate (e.g. to talk to the ticket seller). Perhaps as importantly, much of what is said is interpreted or produced in the context created by this level, giving it focus and direction.

(ii) scripts: Scripts (a term borrowed from Schank and Abelson (1975)) are subgoals of non-linguistic goals (or of higher level scripts) called in to actually direct a conversation (e.g. the script to direct the buying of something such as a ticket). They are responsible for keeping track of the utterances of all parties to a conversation, for determining the sequence of speaking, for recognizing the beginnings and endings of a conversation, for using script expectations to aid the interpretation and production of utterances, and for meshing these expectations with the actual utterances produced. Scripts have available to them models of the conversants for use in performing their varied tasks.

(iii) speech acts: Speech acts (e.g. inquire, respond, inform) represent ideas expressible in a single verbal action by a lone speaker. The name has been chosen because of the similarity of this level to the speech acts approach of Austin (1962) and Searle (1969). Speech acts are invoked by scripts to interpret or produce actual utterances, to check that the utterance is not in conflict with the special requirements of a speech act of its type, and to make sure the utterance doesn't violate anything known about the conversant (available from the conversant model). Speech acts sometimes deal directly with surface linguistic strings, but more often call in language level goals to buffer them from the "real world".

(iv) language level goals: The speciality of this level of analysis is language itself. Thus, there are pattern expressions which know about noun groups, verb groups, clauses, utterances, etc. This is the traditional parsing level, although the methods are not as inflexible as the usual parsing image suggests. The primary task of any goal at this level is to transform a sequence of words into internal concepts, a task which involves appropriately grouping words (syntax), performing checks that the groups are consonant with known information discovered in memory or in the currently relevant execution environment (semantics), and occasionally doing other tests perhaps involving such pragmatic considerations as looking at the conversant model (not directly available, but discoverable in the execution environment).

Much else obviously is involved in language analysis, and such aspects as encoding static (non-procedural) information, handling associative activation, doing bottom-up processing, performing morphological trimming, are touched upon in the report in varying amounts of detail. However, the analysis centres around the four levels just described.

1.4 Organization of the Report

The rest of the report is organized as follows: in Chapter II are further details about conversation as a domain of study; in Chapter III a scheme is proposed for representing knowledge; and it is evaluated in Chapter IV; an example of the representation being used to handle a conversation is shown in Chapter V; while in Chapter VI are some generalizations that can be extracted from the example as well as some extensions to the system; finally, Chapter VII sums up all that has gone before. In addition to the main body of the report, there are two Appendices: Appendix I, containing a description of many of the important objects in the representation; and Appendix II, with some of the more elaborate pattern expressions used in the extended example of Chapter V. Finally, there is an Index of important terms and concepts.

CHAPTER II

Analyzing Conversation

Conversation is an ideal "experimental laboratory" in which to study the interaction of linguistic and world knowledge. It is an area of linguistic performance where pragmatic considerations are uppermost, where things like models of the conversant, goals, context, etc. can be studied not as afterthoughts or in some secondary role, but as central concerns that in certain respects are more important than the analysis of actual utterances. Thus, it is important to analyze how conversants sequence their utterances, how they make use of expectations (both in relation to their goals and in accordance with what has been said so far) to guide them and how they can focus in on the relevant things at any stage of a conversation. Also occurring in conversations are linguistic phenomena such as partial sentences, ungrammatical utterances, and multiple sentence constructions which require language to be viewed as connected discourse not isolated sentences.

Perhaps the main reason for studying conversation is to show the inseparability of language from the context in which it is used. Sometimes the context imposes great control over what is said, sometimes less. Thus, there are very rigid conversational formats, such as ritualized ceremonial exchanges where even the actual words are prescribed; less rigid, but still top-down, task-oriented dialogues (Deutsch (1974)) such as a conversation to order a meal in a restaurant or buy a ticket to a concert; and, finally, more or less unpredictable conversations (e.g. dinner table chatter, talking with a friend) where it is difficult to determine what is going to be said next, but where context still has some role to play in foregrounding concepts as they become relevant.

In summary, conversation is a good area for exploring language because it doesn't arbitrarily restrict the domain of study. To explain, or even to begin to explain, what is going

on in a conversation, the vast number of things mentioned here must be analyzed and unified to yield insights into language use at all levels.

2.1 Background

The attempt to model conversation has not arisen in a vacuum. In this section I would like to very briefly look over related research in artificial intelligence, linguistics, philosophy of language, sociolinguistics, and elsewhere. Naturally, there is a vast amount of relevant work, and in what follows I can only hope to suggest various influences rather than give a detailed description of every piece of research. More specific debts are noted at appropriate places throughout the report.

Sociolinguistics is one area concerned with issues of direct relevance to conversation modelling since it is concerned with examining language in its social context. Work by Schegloff (1971) on sequencing, Garfinkel (1972) on social expectations, Linde (1974) on the choice of determiners in verbalizing internal concepts, Goffman (1974) on "frame analysis" of language, and the like, is indicative of the importance placed on the non-linguistic aspects of language use. Most of this work has been influential to this research in delineating general approaches rather than in yielding specific suggestions.

Philosophy of language also makes a commitment to viewing language as it is used. Thus, Grice (1957) defines language in terms of its intended effect on the hearer, the speech acts paradigm (Austin (1962), Searle (1969)) looks on language as composed of primitive units of meaning (the speech acts), and Grice (1968) and R. Lakoff (1973) attempt to formulate rules of conversation which not only guide speakers but which also form the basis for appropriately judging deviant utterances. Once again, the influence of philosophy of language research on this work has been a general one except, of course, for the speech

acts model, some aspects of which have been incorporated fairly directly.

The case grammar movement (Fillmore (1968), Chafe (1970)) is a major linguistics influence on this research. Case grammar is useful in at least two ways:

(i) it suggests a means of handling the partial and ungrammatical utterances rampant in conversation, in that a case frame is able to pick and choose what it needs from an utterance, ignoring extraneous words and substituting its own default values for non-existent parts of an utterance;
and (ii) the case frame concept can be readily "deepened" to include the necessary semantic and pragmatic checks, an absolute necessity in a domain such as conversation. Computational linguists such as Martin (1975) and Taylor and Rosenberg (1975) have used deep case frames with considerable success. That case frames likely need to be deepened still further is evident from the more recent work of Fillmore (1975) and Chafe (1975) who are looking at frames for encoding extremely pragmatic kinds of information. Obviously, such considerations are not out of line when trying to account for the many real world influences on conversation.

Recently, artificial intelligence has begun looking at language in more comprehensive terms. Winograd's (1972) work, while limiting the domain of study to the blocks world, accounts for an interesting array of linguistic phenomena, including the use of procedural semantics at appropriate times by the syntactic component, the use of history lists that enable the model to discuss previous episodes in the dialogue, the importance of real world knowledge in helping to disambiguate sentences. Moreover, the surface language handled by Winograd's system was extremely sophisticated compared to anything that had been done before.

Woods et al (1972) have also achieved impressive performance in the LUNAR system by using a representational scheme (augmented transition networks) which allows information to be used when appropriate and which has a perspicuous visual

description (i.e. graphs can be drawn representing a particular ATN's flow of control).

Schank (1972) has been a pioneer in focussing language analysis on semantics - almost all interpretation and production of surface language in the various Schank systems has been directed by the desire to fill in slots in his conceptual dependency semantic representation. More recent work (Schank and Abelson (1975)) has further deepened the analysis to pragmatic issues and has attempted to account for how certain kinds of conversations are undertaken with top-down direction by processes called scripts.

Charniak's (1972) model attempts to understand connected discourse, specifically children's stories. The demon style control structure which he proposes there, while containing many flaws, has provoked research into how to use more constrained kinds of control. For example, Minsky's (1974) frame proposal has been, in part, an attempt to overcome the explosive inferencing of demon based control schemes. Interestingly enough, Charniak (1975) himself has adopted some aspects of the frame proposal when modelling episodes in a supermarket, a situation where pragmatic context has been considered all important to appropriate linguistic processing. Another attempt to embed language analysis in a larger context has been Bruce's (1975) work with social action paradigms containing knowledge about stereotyped social situations.

Perhaps most directly relevant to the approach taken here has been the recent investigations into conversation by artificial intelligence researchers. Deutsch (1974) has already been mentioned for her work on task oriented dialogues; P. Cohen (1978) has been concerned with developing a model explaining conversations which take place at the checkout counters of supermarkets; Horrigan (1977) delineates scripts for a couple of task oriented dialogues; Grosz (1977) describes the use of focus in understanding task oriented dialogues; CAI systems (see Collins and Grignetti (1975)) have been simulating realistic natural language dialogues between student and tutor.

Winograd (1977) suggests some prerequisites for a model of dialogue.

2.2 Issues

This research concentrates on four issues of central concern to the modelling of conversation:

(i) whether world and linguistic knowledge can be effectively combined (or, indeed, is there a distinction?);

(ii) how the goals of a conversant affect what he says and understands;

(iii) how the knowledge of a conversant about himself and the other conversants affect what he says and understands;

(iv) how the conversant is able to use context to enable him to focus on the relevant and ignore the irrelevant.

(i) combining world and linguistic knowledge:

Perhaps the main problem that arises when modelling conversation is figuring out how to organize a vast amount of knowledge of various sorts. There are several dimensions along which to categorize this knowledge. A model of conversation needs to have information about various different subjects. Thus, it needs knowledge about language itself, i.e. how to relate words to concepts; how to interpret and produce single utterances; how to handle multiple utterances (perhaps by several different conversants). It needs knowledge about the topic under discussion, so that if a ticket buying conversation is initiated, knowledge about ticket prices, ticket locations, tradeoffs between these, etc. will be needed. It needs knowledge about the conversants, so that in the ticket buying situation the desires and motivations of each conversant need to be taken into account.

Along the procedural / declarative dimension, a model of conversation has to allow for more or less static facts (e.g. cost of a ticket, location of a concert), but also must be concerned with actually doing things (e.g. interpreting or

producing an utterance, engaging in a conversation).

Another kind of knowledge can be categorized according to what Schank (1974) terms the episodic / semantic memory distinction. Clearly a conversation model needs to have episodic capabilities (i.e. a memory of events) not only as to what actually happened in the past but also as to how the model perceived what happened in the past. But just as clearly, any such model would need a semantic memory containing specific knowledge about local things (e.g. location of a specific seat in a theatre) all the way up to general rules (e.g. the fact that people have two legs or a rule for accomplishing some class of tasks).

There are, no doubt, many other kinds of knowledge needed. Regardless, the point is that a major (in many ways <u>the</u> major) problem is to find some sort of scheme that allows appropriate information to be available at the right time while still being comprehensible to the modeller.

(ii) <u>goaling</u>:

The issue of "goaling" breaks down into two sub-issues: how a model's non-linguistic goals affect its linguistic goals; and vice versa. That non-linguistic goals are crucial to conversation is fairly obvious at a number of levels. First, it is clear that conversations are entered into to achieve subgoals of non-linguistic processes. For example, a ticket buying conversation is undertaken to obtain a ticket for some higher level goal such as the goal of attending a concert. And, because a non-linguistic goal "sets the stage" for all that follows, the interpretation and production of utterances is strongly affected by the goal. In the ticket buying conversation, for example, there should be no problem interpreting things associated with tickets, money, seat locations, etc., but talk of zoos, battleships, flying saucers, or the like, would seem out of place. Non-linguistic goals are often directly queried in a conversation, for instance by a ticket seller asking a buyer what he wants at the beginning of a

conversation. Finally, the importance of a non-linguistic goal can determine how perservering a conversant is in continuing a conversation. Thus, a door-to-door salesperson is very persistent in trying to talk to a householder despite open hints of hostility because the goal of selling the product requires such diligence.

It is also fairly evident that linguistic goals affect non-linguistic goals, although in a somewhat less domineering fashion. Thus, what is said can reflect back into and perhaps even alter a non-linguistic goal, so that if a discussion with a ticket seller indicates there are no tickets left, clearly the goal of attending the concert cannot be satisfied. More subtly, a linguistic goal can actually call in a non-linguistic subgoal (e.g. moving closer to a conversant to better hear what he says) or can be interleaved with a non-linguistic goal (e.g. exchanging money for a ticket during the conversation to buy a ticket).

From this discussion it should be clear that the intermeshing of linguistic and non-linguistic goals is an important aspect of the modelling of conversation.

(iii) <u>modelling conversants</u>:

Another obviously important aspect in processing conversations is using models of the conversants to help in the disambiguation and production of utterances. Clearly, who is talking is crucial at all levels of the linguistic endeavour: the type of conversation itself may be determined by the conversants (some married couples, for example, may be unable to engage in any kind of conversation except a vigourous debate); the particular utterances within a conversation may be strongly influenced by the conversant (e.g. when talking to a child certain subjects would be avoided); the style may depend on who is being talked to (e.g. the language of a clerk when talking to his boss is much more constrained and formal than when talking to his mate at the next desk); and even certain speech patterns and quirks of phraseology (e.g. "Y'all come back now,

y'hear?") could be expected to change as the conversant changes.

Conversant models seem to be useful, as well, in formulating concepts to verbalize. In certain rather undirected conversational scenarios (e.g. coffee break conversations, conversations with friends), deciding what to say next can be independent of current goals. One possible strategy is to use models of the conversants to compare what is said to the beliefs of the various conversants. Notable contradictions or similarities could form the basis of a response. Altogether, conversant models are crucial to adequately handling conversation.

(iv) context:

From the above discussion it should be clear that some sort of context mechanism is needed to focus on the relevant aspects of any situation. This is necessary not only for practical reasons of time and space, but also for linguistic reasons. Thus, the problem of ambiguity may be overcome with a sufficiently restrictive context, e.g. "They are flying planes." would be totally unambiguous in the situation where a fleet of airplanes is buzzing overhead. Knowing the surrounding context may also ease reference problems by limiting the number of possible candidates for any referent. For example the sentence "The brown hairy animal ran after the dirty green car." might actually be referring to a particular dog, "Ruff", chasing a particular car, "Smedley Hittite's mangy old Volkswagen" if these concepts have been foregrounded in the context in which the sentence is uttered.

2.3. Levels of Analysis

The issues of section 2.2 illustrate the need to view language in a wider context than just the processing of surface strings. One possible analysis involves viewing linguistic goals in the same terms as other goals of the model. Generally speaking, non-linguistic goals govern lower level linguistic

goals such as scripts, speech acts, and language level goals.

(i) non-linguistic goals:

Much of any realisitic conversation is concerned with things that aren't linguistic at all. As mentioned earlier, any conversation is driven by non-linguistic goals such as attempting to buy a ticket or trying to buy a drink. In addition to this primary role, non-linguistic goals do much foregrounding of useful extra-linguistic information. It is at the non-linguistic level, for example, that the conversants are recognized and pointers to conversant models are recorded for use at all levels. Also delineated at this level is information pertaining to the focus of attention of the model. This is typical of the non-linguistic level: the goal itself and extra-linguistic information available from the goal help focus things further down in the goal hierarchy.

(ii) scripts:

Scripts are the highest level of linguistic goal. They are invoked usually as subgoals of non-linguistic goals (e.g. the script to direct the dialogue to buy the ticket is called in by the non-linguistic goal overseeing the buying of the ticket generally), but can be invoked by other scripts as well (e.g. a subgoal of the buy conversation script is a greeting script that helps establish the conversational roles of the speakers).

A script's primary task is to keep track of utterances on all sides of a conversation. It must not only determine who should speak when and, depending on the identity of the speaker, interpret or produce an utterance, but it must also set up some sort of expectation as to what kinds of utterances will be forthcoming. If the utterance is to be produced, then this content forms the basis of the generated utterance; if the utterance is to be interpreted, then this expected content should serve as an aid to understanding the surface utterance. If notable differences are found between expectations and reality, the script is responsible for explaining them. A

script must also keep a record of any conversation it directs.

A script has available to it the models of the conversants discovered at the non-linguistic level. In many situations scripts will access information in these conversant models to determine that verbalizations are consonant with the beliefs of the conversants, to try to find something to say, etc.

(iii) speech acts:

It has already been explained that scripts can call in subscripts; what hasn't been spelled out is that scripts can also call in subgoals that involve interpreting or generating the speech of a single speaker. Such single verbal subgoals are called speech acts (after Austin (1962) and Searle (1969)). It is interesting to note that speech acts are also central to other computational approaches to discourse (e.g. Bullwinkle (1977), P. Cohen (1978)), thus lending extra credence to their usefulness.

A speech act is responsible for either generating or interpreting an utterance, depending on who the speaker is. In the former mode it takes an expectation as to the content of the speech act and builds a surface utterance which is then actually "spoken" (i.e. printed) under control of the act. In the latter case the speech act reads a surface utterance and tries to interpret it in a way which is both consonant with the act (e.g. "yes" will be interpreted by an "inquire" act as a question but by an "agree" act as a statement) and with the expected content (e.g. if the expected object of an "inquire" is the health of the hearer, then a statement such as "How about you?" must be an inquiry concerning the hearer's health rather than, say, into his desire to do something). Conversant models are available at this level as well, since to properly undertake a speech act may very well involve looking into the beliefs of the conversant. Finally, a speech act records for posterity the actual surface utterance, the actual meaning, and the speaker of and listener(s) to the speech act.

(iv) language level goals:

Sometimes a speech act will itself look directly at surface language, but more frequently it needs the help of a level of goals with language handling expertise. Such goals are grouped into the language level.

Language level processing involves either the interpretation of utterances or the generation of responses. The interpretation process involves breaking an utterance into groups of related words. Each subgroup is then interpreted (which in true top-down parsing fashion may involve still more breakdowns) into some internal concept representing the meaning of the group (the meaning is not finalized at this point, but can be further refined as more information about the concept is gained). These various group meanings must be combined into a single concept representing the meaning of the entire utterance (a process with many case-like aspects). This concept is then passed back to whatever higher goal wanted to understand the utterance.

The generation process has not been studied in detail, but it would involve a somewhat different kind of processing wherein a group would be asked to produce surface words that describe some concept. For example, a noun group might produce a surface level noun group "the brown curly haired animal" to describe the internal concept "Ruff". Clearly such processing would involve all sorts of esoteric decisions as to how much to verbalize, what words to choose, etc. These decisions would have to be made using knowledge about the conversant's knowledge, information about the current goals, and general knowledge about words and word groupings.

Many other kinds of processing would be needed to fully explain conversation, but these four levels do account for most of the issues deemed important earlier. The combining of world and linguistic knowledge is done at any level by mixing extra-linguistic information with the goals at that level; moreover, linguistic goals are supervised by non-linguistic

goals. The desired goaling has been achieved by the basic organization of the levels of linguistic analysis. Conversant modelling is handled by discovering the conversants' identities at the non-linguistic levels and using the information so realized at the lower linguistic levels. The desired context abilities are achieved by the successive focussing on ever narrower goals with a consequent narrowing of things needing to be looked into. The goal tree itself allows access to the current goals of a conversant at any level.

2.4 Where to Begin

There are persuasive reasons for studying conversation using a methodology that involves constructing a computer model to take an active role in a few simple dialogues. Computer modelling enables ideas to be tested, requires precision in the statement of concepts, provides a powerful process metaphor, and allows a performance model to be built. Moreover, as section 2.2 has shown, there is relevant current research which can be used to aid the endeavour.

A number of things must be handled when such a computer model is constructed:

(i) The static information in the conversant model and elsewhere must be handled; so must the dynamic goaling activities. This information must be stored in ways that allow general rules to be combined with specific information.

(ii) Techniques must be found to access knowledge about conversants, knowledge about subgoaling, knowledge about language, knowledge about the topic of discussion, etc.

(iii) There must be some way of keeping all this information in line so that the system is not overwhelmed by too much at once. The goal tree, as has been pointed out, is useful in this capacity, but exactly how it is to be represented and used must be determined.

(iv) At various levels it is important to record information as to what was said, who said it, what the context

was then, what processing decisions had to be taken, and so on.

(v) Naturally, when things go wrong (for example, when a script or speech act has its expectations violated), the anomaly must be excused or explained.

(vi) There must be a satisfactory tradeoff between doing all this efficiently and doing it in a way which is comprehensible.

Now, if the introduction is re-examined, these six requirements will be seen there in slightly disguised terms as the six representation issues of concern to this research. That is, (i) is the procedural / declarative controversy; (ii) is the problem of accessing information; (iii) is the need for context; (iv) is the necessity for keeping a record of the processing; (v) is the issue of robustness; and (vi) is the problem of devising methods which are non-combinatorially explosive yet within the complexity barrier. Therefore it should come as no great surprise that before modelling conversation, several representation problems must be handled.

The next couple of chapters will be concerned with doing exactly that. Chapter III outlines a scheme for the representation of knowledge which enhances the chances of achieving (i) - (vi). Chapter IV undertakes an analysis of the strengths and weaknesses of this scheme.

CHAPTER III
A Scheme for Representing Knowledge

In this chapter I would like to discuss a scheme for representing knowledge that has been developed as a preliminary to exploring the many facets of modelling conversation. The scheme is being tested by implementing it as a set of programs collectively called |LISP (because all system functions are preceded by a "|" to distinguish them from other functions). System notation is based as closely as possible on LISP, complemented where applicable by CONNIVER notation (McDermott and Sussman (1974)). The implementation is meant to be an extension of LISP in the sense that most of LISP can be invoked directly from within it. Finally, it must be emphasized that the version of |LISP described here has not been coded, although several predecessors have been. Hence, all code is "soft" in the sense it hasn't been run on the computer.

The chapter is organized as follows: section 3.1 contains an overview of the scheme's capabilities which, for all but the most intrepid, should provide background sufficient to understand the rest of the report. But, for interested readers, sections 3.2 through 3.7 describe in detail the various features of the scheme: objects, message passing, the matcher, pointers and searching, interrupts, and simulated parallelism. When reading the rest of this chapter (and for that matter the rest of the thesis), an important point to remember is that there are algorithms and detailed descriptions of many system functions in Appendix I.

3.1 Brief Overview

The system is divided into fundamental conceptual units called objects which can pass messages to each other and receive responses. All messages and responses are co-ordinated by an interpreter (named |EVAL) which reads messages from sending

objects, directs them to the proper receiving objects, and later steers the receiving objects' responses back to the appropriate sending objects.

There are many different types of object in the system, classified according to their message passing behaviour. The simplest are LISP SUBRs and EXPRs with standard LISP argument conventions. |EXPRs are similar to EXPRs, except that their internal structure is |LISP code and they use stacks local to objects called |PEXPRs for their argument binding. These types of object are all useful for doing relatively efficient, LISP-style processing, but they do not resolve many representation questions than efficiency.

The objects that are most crucial to the representation are objects called pattern expressions (|PEXPRs) which correspond (roughly) to frames in that they each represent a single large piece of knowledge in the system. Pattern expressions are meant to be major domain-dependent objects: most world and linguistic knowledge is contained in pattern expressions. (Much procedural knowledge still remains embedded in EXPRs or |EXPRs rather than the more analyzable |PEXPRs because a thorough analysis of the semantics of procedures is beyond the scope of the report).

|PEXPRs are objects whose structure is a list of patterns. A pattern is a list whose first element is the name of an object, and the rest of whose elements are either the names of objects or further sub-patterns. Moreover, any element of a pattern can be preceded by a single macro character ("?" "!" "$" "=" "↑" "#" "∠" "/") which has significance in its matching behaviour. The patterns in a |PEXPR are the major means of defining exactly what that |PEXPR means (another aspect of a |PEXPR's meaning comes, of course, from the use of its name in the patterns of other |PEXPRs).

An example of a |PEXPR is the WIDGET-PEDDLER pattern expression

```
<|PDEFUN WIDGET-PEDDLER
     S1 : (SUPERSET WIDGET-PEDDLER SELLER)
     S2 : (SUPERSET WIDGET-PEDDLER PERSON)
     S3 : (SELL ↓WIDGET-PEDDLER ↓WIDGET)
     S4 : (TRADE ↓WIDGET-PEDDLER ↓BUYER ?GOODS
                 !(|COND ((SUB-INSTANCE-OF GOODS WIDGET)
                          (CREATE-NEW 'MONEY))
                         (T (CREATE-NEW 'SERVICES)))))
     S5 : (CORE WIDGET-PEDDLER (/S3 /S4))
     S6 : (INSTANCE WIDGET-PEDDLER PETER)
     S7 : (INSTANCE WIDGET-PEDDLER MARTHA)
                    >
```

This |PEXPR contains knowledge about widget peddlers. It in itself does not have any "real world" reference; that is WIDGET-PEDDLER is not any particular widget peddler or group of peddlers, but is rather a description of the characteristics that widget peddlers have in common. It can be interpreted as follows:

(i) the name of the |PEXPR is WIDGET-PEDDLER;

(ii) the body of the |PEXPR is the collection of patterns S1 through S7;

(iii) each pattern in the body is labelled with a pattern name Si;

(iv) pattern S1 says that the |PEXPR WIDGET-PEDDLER has superset SELLER;

(v) pattern S2 says that the |PEXPR WIDGET-PEDDLER has another superset, PERSON;

(vi) pattern S3 says that an arbitrary instance of WIDGET-PEDDLER sells an arbitrary instance of WIDGETs;

(vii) pattern S4 says that an arbitrary instance of WIDGET-PEDDLER trades with an arbitrary instance of BUYER as follows: the widget peddler exchanges "goods" for either money or services depending on whether the goods are widgets.

(viii) pattern S5 says that patterns S3 and S4 are the core patterns of the |PEXPR; that is, they are more central to its meaning than are other patterns (this is useful mainly in comparing |PEXPRs to one another where core patterns are the central patterns which must be compared);

(ix) patterns S6 and S7 define a couple of instances of the widget peddler, namely PETER and MARTHA.

The other component of any |PEXPR is how it handles messages sent to it from other objects in the system. Another object, say the |PEXPR BUY-WIDGET, might formulate the message

```
(WIDGET-PEDDLER
        (TRADE PETER SELF SQUIGGLY-WIDGET ?WHAT-COST)
        (ATTITUDE PETER ?WHAT-ATTITUDE))
```

That is, BUY-WIDGET is interested in seeing what (according to WIDGET-PEDDLER) SELF might give to PETER in return for the SQUIGGLY-WIDGET; and also to see what attitude PETER should be expected to maintain.

Here is a simplified outline of what happens to the message:

(i) The message is read by the interpreter as it is executing code within a pattern of the BUY-WIDGET expression. The interpreter (named |EVAL in the implementation) acts as a central switchboard whose task it is to buffer contact between objects. Whenever an object formulates a message, it is read by the interpreter and dispatched to the appropriate receiving object where it is further processed. If this object should desire to send any messages of its own, it too must route them through |EVAL. When an object is finished answering a message, it notifies the interpreter which passes back the response and resumes execution of the object which sent the message.

(ii) So, in this case the interpreter sees that the receiving object is to be a |PEXPR named WIDGET-PEDDLER.

(iii) Setting aside for the moment the rather complex set of things that happens next, the eventual effect of the message is that the message pattern

(TRADE PETER SELF SQUIGGLY-WIDGET ?WHAT-COST)

is matched against patterns in the body of

WIDGET-PEDDLER.

(iv) The pattern labelled "S4" is discovered and is found to match under the assumption that PETER is a particular WIDGET-PEDDLER (a fact discoverable by looking at the |PEXPR to see if it has a pattern (INSTANCE-OF PETER WIDGET-PEDDLER)) and SELF is a particular BUYER. If the further assumption is made that SQUIGGLY-WIDGET is a particular WIDGET, then the pattern's last element will be MONEY1, a |PEXPR representing an individual price appropriate to the widget. Thus, the pattern to be returned in answer to the TRADE message is

(TRADE PETER SELF SQUIGGLY-WIDGET MONEY1).

This pattern is saved for return to BUY-WIDGET.

(v) But, first, the message pattern

(ATTITUDE PETER ?WHAT-ATTITUDE) must also be matched in WIDGET-PEDDLER. No match is found, so after a further set of rather elaborate machinations which I will go into shortly, the "ISA environment" "above" WIDGET-PEDDLER is looked into.

(vi) The only "ISA" |PEXPRs are SELLER and PERSON, so they are searched in breadth-first fashion (although in this case there isn't much breadth for long since the superset of SELLER is itself PERSON) for a pattern matching (ATTITUDE PETER ?WHAT-ATTITUDE).

(vii) If the knowledge of sellers is complete, it will be discovered that the attitude of any particular seller is that of politeness, so

(ATTITUDE PETER POLITE) will be the appropriate matching pattern for the message. This pattern will be inherited by WIDGET-PEDDLER and will be saved for return to BUY-WIDGET.

(viii) Since there are no more patterns in the message, the two response patterns

(TRADE PETER SELF SQUIGGLY-WIDGET MONEY1) and

(ATTITUDE PETER POLITE) are appended and returned to

BUY-WIDGET where they can be used.

This, then, has been a general look at some of the features of the representation scheme. I would now like to discuss these features in more detail.

3.2 The Objects

3.2.1 Pattern Expressions

A pattern expression is defined as an atom with a property list indicator |PEXPR designating its body. The body is a list of patterns, where a pattern is, in turn, a list of elements, the first element of which is an object name and the rest of whose elements can be object names, sub-patterns, or single-character macros. A macro indicates that the following element is to be treated in a special way by the pattern matcher, thus giving the ability to postpone decisions until the information in the pattern is actually needed. In particular there are macros which allow computations to be carried out, to allow matching of any element, to allow expansion to larger elements, and others. Macros will be more fully explained in the section on pattern matching.

Patterns can also be labelled so that other patterns can refer to them. This is done by preceding a pattern with a name, followed by a ":"; for example,

S1 : (SUPERSET WIDGET-PEDDLER SELLER)

designates the pattern (SUPERSET WIDGET-PEDDLER SELLER) with the label S1. These labels are local to the pattern expression (so that other pattern expressions can use the same name without confusion). Finally, only a top-level pattern can be labelled, i.e. labels cannot appear inside patterns.

3.2.2 Other Objects

Aside from |PEXPRs, there are three kinds of object: primitives, EXPRs, and |EXPRs. Primitives are named atoms with no internal structure. Such objects are primitive in the sense they cannot be further analyzed; but, unlike Schank's (1975) primitives, much information could be added to the object if some future analysis made it necessary to have a more elaborate conception of the object. Moreover, in many situations, even well endowed objects need be considered only in terms of their names, not their contents, hence making them effectively primitive at that time. The idea that any object can be considered primitive or not, as desired, is fundamental to the system and is in conflict with Schank's hypothesis (this is not to say that it isn't often useful to consider a conveniently small set of primitives in certain circumstances).

EXPRs are objects whose structure is stored under the indicator EXPR on the property list of the object name. The body of an EXPR is a LISP LAMBDA, NLAMBDA, OR FLAMBDA expression.

|EXPRs are the same as EXPRs except the property list indicator is |EXPR and the body is a |LAMBDA, |NLAMBDA, or |FLAMBDA expression. These |LAMBDA expressions have a slightly different way of binding their arguments (using the current |PEXPR stack, something that will be explained shortly).

3.3 Details of Message Passing

3.3.1 Introduction

An object communicates with another object when it needs a piece of knowledge that it doesn't know directly, when it has information that would be of use to another object, when it wants to achieve a subgoal, when it wants to associatively signal a "neighbouring" object; and so on. Such communication is accomplished by passing a message from one object to another and receiving a corresponding response.

To be able to send a message to object B, object A must know B's name. This is not quite as much of a restriction as it might seem, since many computations can be carried out by A, if necessary, to discover the name. Once it has the name, A must then compose the message in the format which B expects (be it a list of arguments, a list of patterns, or whatever). A message form

(B "message")

is then created by appending the message to the receiving object's name and this form is sent off to the interpreter |EVAL.

The basic logic for |EVAL is given in Appendix I, but a brief description of its behaviour is in order here. Assume that |EVAL has received a message form of some sort. The message can either be (i) an atom or (ii) a list.

(i) If the message form is an atom, |EVAL realizes that no message is being sent to any other object; instead the value of the atom is desired. Atom-value pairs are stored within pattern expressions on a stack which is just a pattern of the form

(|STACK object-name (stack-values))

whose third element contains the atom-value pairs. To understand exactly which stack is to be searched for the value of an atom, it is necessary to realize that in some sense all processing is done within pattern expressions. Thus, the system is initialized with a top-level READ-|EVAL-PRINT pattern expression in control (see TOP-VIEW in Chapter V) and all top-level calls are in some sense embedded within this |PEXPR. Whenever a message is sent to some other |PEXPR, the new |PEXPR becomes the pattern expression in which subsequent processing takes place. It either finishes (in which case a previously initialized |PEXPR is resumed, probably the one to which the response has been sent) or it calls in still another |PEXPR to take control. In any case there is always unambiguously one pattern expression in control: it is called the current |PEXPR and all stack operations are performed on its stack.

Thus, in the situation where the message form is merely an

atom, the interpreter finds its value on the current stack and returns it as the response to the message (or NIL if no value is found).

(ii) If the message form is a list with message head (i.e. first element) either an atom or a lambda expression of some sort, then this is a message to be further processed. If the message head is

(a) the SUBR QUOTE; then the message is returned unchanged.

(b) a |PEXPR; then special processing takes place which I'll explain shortly.

(c) a |EXPR; then associated with the object is a |LAMBDA expression which accepts the message as actual parameters to be bound to its formal parameters on the current stack. The binding is done (perhaps requiring the |EVALuation of the actual parameters depending on whether the body is a |LAMBDA, |NLAMBDA or |FLAMBDA), the object's |LAMBDA body is then |EVALed. Eventually, the |LAMBDA body is done, so the bindings are undone, and the result of the |LAMBDA |EVALuation is returned in response.

(d) an EXPR; then associated with the object is a LISP LAMBDA expression accepting the message as actual parameters to be bound to its formal parameters on the LISP stack. The binding is done, requiring first the EVALuation of the actual parameters if the LAMBDA is a LAMBDA (but if it is a NLAMBDA or FLAMBDA, it is assumed that the programmer has indicated explicitly when to EVAL in the body of the function). The EXPR is then EVALed, and the value returned is the response to the message.

(e) a SUBR; then the object is APPLYed to |EVAL of each element of the rest of the message form.

(f) a NSUBR or FSUBR; then an error is generated since such objects cannot |EVAL the message elements properly.

(g) a |LAMBDA-expression or LAMBDA-expression; then the obvious binding, popping, and EVALing takes place on the PEXPR stack and LISP stack respectively, just as in the corresponding

EXPR or |EXPR case.

(h) any other form; will result in error.

3.3.2 Messages to |PEXPRs

So, we now return to |PEXPR message handling, both the most important and also the most complex kind currently in existence. If the function descriptions of Appendix I are studied the Detailed function descriptions are in Appendix I, so perhaps the best way of elucidating the process here is to do a step by step analysis of what would happen to one communication sent from object A1 to object B. Say the message form

(B (x y z)(r s t u))

is |EVALed within A1. The |LISP interpreter reads the message form, sees the object B (the message head) is a pattern expression (by discovering the attribute |PEXPR on B's property list). It creates a new, initially empty pattern expression to serve as a working-storage area for B as it answers the message. The new |PEXPR is given an internal name (B1, say) and is called an execution instance of B. Next, three patterns are asserted in B1:

```
(EX-INSTANCE-OF B1 B)
(EX-ENVIRON B1 A1)
(|STACK B1 NIL)
```

EX-INSTANCE-OF "points" to B, the object which has been sent the message; via this pointer all of the ISA environment can be accessed (where the ISA environment consists of all those |PEXPRs that can be reached by going along EX-INSTANCE-OF, INSTANCE-OF, or SUPERSET pointers). EX-ENVIRON "points" to the sending |PEXPR A1 in the execution environment or dynamic context of B1. Finally |STACK indicates the local pattern expression stack, initially empty.

Next, the stack must be initialized with a form called the message handler. This form can then be EVALed and will direct the pattern expression in its hunt for an appropriate response.

In this case, a form

(|PEXPR-MH (B (x y z)(r s t u)))

will be pushed onto the stack of B1 as the value of the special atom |EV. Such an atom / value pair is called a "|EVAL block". Whenever the interpreter decides to actually execute a |PEXPR, it merely looks for the top |EVAL block on the stack of that |PEXPR and executes the form it finds there. In this case, then, the call to the message handler |PEXPR-MH can be discovered and |EVALed to carry on the processing of the message. It would seem reasonable to make B1 the current pattern expression and do exactly this, thus effectively transferring control.

But, this is not what happens: B1 is merely <u>scheduled</u> to run by merging it into an execute queue along with a priority indicating its potential importance to the system. This priority defaults to 5 (highest priority = 9, lowest = 0) if it isn't explicitly given in the message. Such an explicit priority can be given using a pattern

(|PRIO= priority)

which is essentially an instruction to the interpreter rather than a pattern to be handled by B1. (Other such messages will be seen in section 3.6). The interpreter checks for all such patterns, strips them from the message, and acts upon them. For (|PRIO= priority), the action is to assert in B1 the priority (PRIORITY B1 priority), in this case, by default, (PRIORITY B1 5).

The scheduler is then summoned. After having "aged" the execute queue priorities (by increasing all priorities by some fixed amount so that eventually all objects will get to run), it calls upon the pattern expression with the highest priority to be made current and executed. By executing a |PEXPR, I mean |EVALing the top |EVAL block on its stack.

So, in due course B1 is called in, made the current pattern expression, and the top form

(|PEXPR-MH (B (x y z)(r s t u)))

is |EVALed. |PEXPR-MH, the EXPR which handles most pattern

expression messages is given in Appendix I. It matches each message pattern (i.e. (x y z), (r s t u)) against receiving patterns in the body of B. At the present time, the patterns of B are scanned from newest pattern to oldest (where the sequencing of patterns within a |PEXPR can be determined by looking at their position in the list of patterns that constitute the body of the |PEXPR). It would be better, in the long run, if |PEXPR-MH used an indexing scheme to access the patterns of a |PEXPR body, but the |PEXPRs considered so far are small enough to obviate the immediate need for such a feature.

If patterns matching the message patterns are found, they are put in an answer list and returned by |PEXPR-MH as the answer to the message. |PEXPR-MH also asserts each matching pattern in the body of B1 so that they can be later accessed if desired. |PEXPR-MH sends the answer list back to A1, and also asserts in the body of A1, a <u>return condition</u> of the following form:

(RETURN-COND
    ex-environ-ob return-type return-codes return-value)

where return-type is (usually) NORMAL, return-codes is a list of sub-|PEXPRs of importance to answering the message (usually the receiving object), and return-value is the answer list of matching patterns. In this example,

(RETURN-COND A1 NORMAL (B1) ((x' y' z') (r' s' t' u')))

might be asserted in A1. This return condition is basically unimportant for the NORMAL return, but for other more esoteric returns, return-codes become crucial.

The question arises: what would have happened if |PEXPR-MH had been unable to discover a match in B for some message pattern, say (x y z)? In cases such as this, the matcher does not give up. Instead, it looks at the first element of the pattern to be matched and asks the object with that name (i.e. x, here) for help. The rationale for consulting x is that the first element of a pattern usually acts as the relation connecting all the other elements, and is thus the most crucial part of the pattern. The hope is that x will have associated

with it a failure to match pattern that tells the matcher what to do if at any time an x pattern is unmatched. There are various possibilities: look into the ISA environment of B1; look into the execution environment of B1; perform some sort of inference; give up; and so on. The various possibilities for search will be elucidated in Chapter VI. With the system.

For the sake of illustration, assume that x has the pattern

(1) (FAILURE-TO-MATCH x ?PATTERN ?OBJECT
!(SEARCH-ISA PATTERN OBJECT))

in its body; that is, if PATTERN has failed to match in OBJECT, direct a search (using the EXPR SEARCH-ISA) for PATTERN into the ISA environment of object. What SEARCH-ISA does is to search up EX-INSTANCE-OF, INSTANCE-OF, and SUPERSET links, breadth-first, matching patterns against patterns in the body of each !PEXPR encountered until eventually the pattern is matched and the matching pattern returned, or until no ISA links remain to be traversed in which case the first message (x y z) cannot be answered. This eventuality would result in NIL being appended to the answer-list.

In the present circumstances, the matcher has failed to find a match for (x y z) in B1, so it formulates a pattern

(2) (FAILURE-TO-MATCH x (x y z) B1 ?MATCHING-PAT)

which it matches against patterns in the body of x, finding (1) above thus effecting an ISA search from B1. Eventually, MATCHING-PAT will be bound to a pattern matching (x y z) or NIL and this value will be returned as the response to the (x y z) message pattern. A final point: if x did not contain any FAILURE-TO-MATCH pattern, then the matcher would know automatically not to undertake failure processing (thus avoiding an infinite regress of attempted matches).

Failure to match processing means that quite robust behaviour can be achieved. A system can be designed so that it can, when temporarily stymied, make use of knowledge appropriate to the current context and the kind of data which is causing the trouble rather than relying on some more uniform mechanisms. Being able to use local knowledge such as this is one of the

strong points of this approach.

Meanwhile, once it has finished answering the message, |PEXPR-MH reschedules the sending pattern expression (discovered by looking up the EX-ENVIRON pointer), which is A1 in this case. The priority of A1 on the execute queue is set to its original (stored) value, even if A1 at some time had been given a larger queue priority as a result of the scheduler's aging technique. Eventually, A1 is re-started where it left off with the value of the message being readily available. Thus, if ((x y z)(r s t u)) had been embedded in a call to the |EXPR RALPH; e.g.

(RALPH (B (x y z)(r s t u)))

then RALPH would have access to the value of the message, i.e. RALPH would execute with argument

((x' y' z')(r' s' t' u')).

RALPH could, of course, find out the new execution instance created in answering the message, the return type, and so on, by looking at the RETURN-COND pattern.

One final note here: B1 has not disappeared (although its stack has been fully popped of |EVAL blocks). Execution instances stay around so they can (perhaps) be queried later as to various pieces of information in them such as the matching answer patterns, the execution environment or execution instance pointers, etc. This is important in many places, especially in looking at old execution environments to see the concerns of that time. Eventually, of course, some sort of garbage collection must take place. Moreover, given the potential importance of some execution instances, it seems crucial to do such collection <u>intelligently</u>. Such an intelligent garbage collector has not been worked out in any detail, although what it would need to do is discussed somewhat more fully in Chapter VI.

3.4 The Matcher

3.4.1 The Definition of the Pattern Matching Macros

Throughout the exposition of the representation scheme, the concept of pattern matching has cropped up with alarming regularity. The most ubiquitous use of pattern matching is by the system itself when it handles pattern expression message passing. But the pattern matcher can be used elsewhere, by other system programs or by the user. Because of this, more general terminology than "message pattern", "receiving pattern", "sending object", and "receiving object" is needed to describe the patterns being compared. So, I will speak of

(i) the source pattern, the pattern for which a match is being sought (corresponding in the message case to the message pattern);

(ii) the target pattern, the pattern which is a candidate as a possible match for the source (corresponding in the message case to the receiving pattern);

(iii) the source object, the pattern expression which is to serve as the context for any macro operations in source pattern (the sending object in the message case);

(iv) the target object, the pattern expression which is to serve as the context for any macro operations in the candidate pattern (the receiving object in the message case).

Whenever a source pattern is matched against a target pattern, the source object and target object must be specified as well. The basic principle underlying pattern matching can be summed up by the fundamental matching rule:

> A source pattern matches a target pattern if each element of the source matches the corresponding element of the target, unless one of the elements is NIL, in which case the patterns fail to match. If a source element designates an object, then it matches the corresponding target element only if they

designate the same object. If a source element is a sub-pattern, then it matches the corresponding target element only if the sub-patterns match.

Under the fundamental matching rule, the following source / target pairs will match:

(i) (A B C D):(A B C D)

(ii) (A (B C) D):(A (B C) D)

(iii) (((A (((B C)) D)))):(((A (((B C)) D))))

but, the following will not match:

(i) (A B C D):(A (B C) D)

(ii) (A (B) (C) D):(A (B) (C D))

(iii) (A (B (C (D)))):(A (B (C (D NIL))))

Pattern matching would be a trivial exercise indeed if this were all there was to it. But, matching is made more complicated by macros which are special characters (not objects) that have meaning to the matcher and certain other system functions. A macro character can precede any element in a pattern (including a sub-pattern). No more than one macro character per element is allowed, however. A macro character indicates to the matcher that the element is to be treated differently during matching. There are macros that tell the matcher to !EVAL the element before matching, or to consider that the element matches anything, or to match only elements which pass certain tests, or any of a number of other things.

In the following description of the macros, it should be kept in mind that their operation is totally symmetric. The macros are often described as if they appear before an element in a target pattern, but their action if they appear before a source pattern element would be exactly analogous.

(i) "!"---This macro is an instruction to the matcher to !EVAL the following element before attempting to match it (because the matcher works in inverse QUOTE mode, !EVALuation must be explicitly indicated). If a stack is needed, the stack corresponding to the pattern containing the element is used.

For example (INSTANCE-OF GEORGE-III KING) will match (INSTANCE-OF !KING KING) if and only if KING has value GEORGE-III on the stack of the target object. Another example:

(EXECUTE GOTO QET)

will match

(EXECUTE GOTO !(|COND ((EQ CONCERT 'SYMPHONY) 'QET)
(T 'COLISEUM)))

if and only if the |COND |EVALs to QET. Note that the target stack must be used to find the value of CONCERT.

It is through the use of "!" macros that a pattern expression obtains procedural ability; that is, often a rather long computation must be carried out before trying to match an element, a computation which could involve sending messages to other pattern expressions, etc. In the message passing example (section 3.3.1) the message to B could have been embedded in some sort of pattern in A1's body, i.e.

(EXECUTE FOIBLES !(RALPH (B (x y z)(r s t u))))

which was being matched in an attempt to answer a message sent to A1 by some other pattern expression. Before the third element could be matched, the "!" forces a |EVAL of the "RALPH" expression, thus setting off the chain of events described earlier.

A final note: "!" can often be used outside of patterns to indicate |EVAL where normally no such |EVAL could take place. In particular it can be used in the arguments of |ASSERT (see below) and in front of the message head in a message form (indicating that a message is to be sent to the value of a name rather than to the name itself).

(ii) "$"---This is almost exactly the same as "!" except that it says to the matcher to EVAL the element, rather than to |EVAL it. Although using "$" is often useful for efficiency reasons, caution should be exercised since no |EXPRs or |PEXPRs can be handled nor can any |PEXPR level variables.

A common use for "$" is when some EXPR must be executed with constant arguments, e.g. when matching (COMPUTE FACTORIAL-5 $(FACT 5)), the previously defined LISP

function FACT can be called in since it uses no |LISP features.

(iii) "/"---This macro character assumes that the following element is the label of a pattern in the body of the target element. "/" merely says to replace the label by the pattern itself during matching. Thus, the source (CHASE (BROWN DOG) HELICOPTER) will match the target (CHASE /S7 HELICOPTER) only if S7 labels the pattern (BROWN DOG) in the target object.

(iv) "?"---The processing for this macro character is slightly more complex than for the macros described thus far. "?" indicates that the target element is to match anything, but with the side-effect that the element is bound to the matching element on the target stack. Note that the target element must be an object rather than a sub-pattern.

For example the target pattern (SLOGAN WOMENS-YEAR (?WHAT)) will match the source (SLOGAN WOMENS-YEAR ((WHY NOT)) with WHAT being bound to (WHY NOT) on the source stack.

(v) "#"---Every so often it is desirable to suppress macro processing in the corresponding element of the other pattern so that the full blown macro code can be looked at. "#" is designed to do this. It acts like "?" in that it will match anything and bind its element to the matching element; however, it first turns off macro processing on the other side (except if the element has a macro that is itself # - see below). This is useful for getting "code" unevaluated so that it can be examined. Thus, the target (EXECUTE PARSER #PARSE-CODE) would match the source (EXECUTE PARSER !(|PROG()---)) with PARSE-CODE being bound to !(|PROG()---) in the source.

(vi) "="---This is a macro character that is restricted in the kind of element which can follow it. "=" must precede an element of the kind "(object-name message-form)", where object is any object in the system and message-form is any form which can be |EVALed. Thus, the full macro-element pair is "=(object-name message-form)".

Assume "=" is in the target pattern. It tells the matcher

1. to temporarily bind object-name (on the target stack)

to the corresponding source pattern element;

2. to |EVAL message-form;

3. if message-form doesn't return NIL, then the binding is kept and the elements match;

4. else, if message-form returns NIL, then the binding is discarded and the elements fail to match.

This feature allows a "condition" to be imposed on the kinds of elements that will match, a condition that can be arbitrarily complex.

For example, the pattern

```
(SELLS =(INDIVIDUAL (SUB-INSTANCE INDIVIDUAL SELLER))
        TICKET-TO-CONCERT)
```

will only match patterns whose first and third elements are SELLS and TICKET-TO-CONCERT and whose second element is some individual who is a SELLER. SUB-INSTANCE is an EXPR which returns T if its first argument names a |PEXPR that is an instance (or execution instance) of the |PEXPR specified in the second argument; else SUB-INSTANCE returns NIL.

If JOHN is such an individual, then the source pattern (SELLS JOHN TICKETS-TO-CONCERT) would match the target above with INDIVIDUAL being bound to JOHN on the target stack.

(vii) "↓"---Without loss of generality assume that "↓" precedes a target element. Then "↓" informs the matcher that the source element must be a SUB-INSTANCE (as just defined) of the target element. Thus, ↓SELLER will match only instances (or execution instances) of SELLER or instances (or execution instances) of subsets of SELLER. The matching SUB-INSTANCE is assigned as value of SELLER. Finally, note that ↓X matched against ?Y will result in the creation of a new instance of X to be assigned to Y (see macro conflict table, Figure 3.1).

(viii) "⌀"---This macro character precedes an atomic target element and binds it to the corresponding source element as long as that source element is the name of the source object. Thus, the target (BEATS ⌀ME) matches the source (BEATS YOU) if YOU is the source object. This is useful mainly in accessing data from internally named |PEXPRs or yet to be named |PEXPRs

(such as new execution instances).

This set of macros is a preliminary set which has been found to be useful in the examples I have considered. Many more could no doubt be contrived. Although they cannot be nested, they can be used together in the same pattern provided they are attached to separate elements. And when such features are used many standard programming language features can be simulated. "?" allows parameter passing and the binding of arguments; "!" gives a procedural capability and when a "!" is opposite a "?" then a call-by-result is evident. For example, the source (FWUMP X !Y ?ANSWER) matches the target (FWUMP ?PARAM1 PARAM2 !(CONS PARAM1 PARAM2)) with PARAM1 being bound in the target object to X; PARAM2 being bound in the target object to the value (in the source object) of Y (say VAL-Y) (effectively call by value); and ANSWER being bound to the result of CONSing PARAM1 TO PARAM2, e.g. (X . VAL-Y) (i.e. a procedure in the target object and a call by result in the source).

This brings up the question of what exactly are the advantages of using pattern matching rather than direct function calls? There is, first of all, the ability to match non-procedural patterns and the ability to use procedural information (through the use of "↑", "!", "=", and other macros) neither of which are normal function calling abilities. Moreover, there is the fact that pattern matching is less definitive about what are arguments, values, procedures, etc. In one case ?PARAM1 might act as a call by value, in another case something else, depending on its corresponding element. Another difference is that the use of pattern matching in message passing between |PEXPRs allows multiple procedures to be attached to the same object in the sense that more than one pattern could match (potentially), and the sender has no way of knowing which one will. Finally, failure to match techniques associated with the matcher go well beyond the scope of most

programming languages. In summary, matching has been designed to be more general than the usual procedure calling mechanisms. The paradigm that a subset of the matching capabilities co-incides with some of the standard programming language features.

3.4.2 Difficulties with Macros

Throughout the discussion of macros, an unstated problem has existed: what to do if an element in the source pattern and the corresponding element in the target pattern are each preceded by a macro character. Most of the time the resolution of the conflict is just common sense, as will be seen if the macro conflict table, Figure 3.1, is studied closely. The occasional difficulties are explained there as well. Note that the table is symmetric with respect to source and target elements.

!SO -
- !TA - |EVAL SO; |EVAL TA; if equal, match succeeds.
- $TA - |EVAL SO; EVAL TA; if equal, match succeeds.
- /TA - |EVAL SO; expand TA; if equal, match succeeds.
- ?TA - |EVAL SO; TA <-- result; match succeeds.
- #TA - TA <-- !SO (un|EVALed); match succeeds.
- =(TA cond-TA) - |EVAL SO; TA <-- result; |EVAL cond-TA; if non-NIL, succeed; else unbind and fail.
- ↓TA - |EVAL SO; if result is a subinstance of TA, TA <-- result and succeed; else fail.
- ∉TA - |EVAL SO; TA <-- source object; if equal, match succeeds; else fail.

$SO -
- $TA - EVAL SO; EVAL TA; if equal, match succeeds.
- /TA - EVAL SO; expand TA; if equal, match succeeds.
- ?TA - EVAL SO; TA <-- result; match succeeds.
- #TA - TA <-- $SO (unEVALed); match succeeds.
- =(TA cond-TA) - EVAL SO; TA <-- result; |EVAL cond-TA; if non-NIL, succeed; else unbind and fail.
- ↓TA - EVAL SO; if result is a subinstance of TA, TA <-- result and succeed; else fail.
- ∉TA - EVAL SO; TA <-- source object; if equal, succeed; else fail.

/SO -
- /TA - expand SO; expand TA; if equal, match succeeds.
- ?TA - expand SO; TA <-- result; match succeeds.

#TA - TA <-- /SO (unexpanded); match succeeds.
=(TA cond-TA) - expand SO; TA <-- result; |EVAL cond-TA; if non-NIL, succeed; else unbind and fail.
¥TA - match fails (pattern can't be subinstance of object).
¢TA - match fails (pattern can't be source object).

?SO -
?TA - SO <-- |UN; TA <-- |UN; match succeeds.
#TA - SO <-- |UN; TA <-- ?SO; match succeeds.
=(TA cond-TA) - SO <-- |UN; TA <-- |UN; |EVAL cond-TA; if non-NIL, succeed; else unbind and fail.
¥TA - TA <-- SO <-- a new instance of TA; match succeeds.
¢TA - SO <-- TA <-- source object; match succeeds.

#SO -
#TA - SO <-- #TA; TA <-- #SO; match succeeds.
=(TA cond-TA) - SO <-- =(TA cond-TA); TA <-- |UN; match succeeds.
¥TA - SO <-- ¥TA; match succeeds.
¢TA - SO <-- ¢TA; TA <-- |UN; match succeeds.

=(SO cond-SO) -
=(TA cond-TA) - TA <-- SO <-- |UN; if both |EVAL of cond-SO and |EVAL of cond-TA are non-NIL, then succeed; else fail.
¥TA - TA <-- SO <-- a new instance of TA; |EVAL cond-SO; if non-NIL succeed; else unbind and fail.
¢TA - SO <-- TA <-- source object; |EVAL cond-SO; if non-NIL succeed; else unbind and fail.

¥SO -
¥TA - match succeeds if TA is a subset of SO or SO is a subset of TA; a new instance of the lowest one is created and bound to both TA and SO.
¢TA - SO <-- TA <-- source object; if source object is a subinstance of the object SO, succeed; else unbind and fail.

¢SO -
¢TA - TA <-- source object; SO <-- target object; if they are the same, succeed; else unbind and fail.

Legend

SO is the source element (contained in source pattern).
TA is the target element (contained in target pattern).
X <-- Y means assign X to Y (undone if match fails).
|UN is a special NIL-like atom that means "unassigned".

Figure 3.1 - Macro Conflict Table

It should be noted that there can be situations when there are incompatible levels, specifically

(i) ?SO is matched against (TA1 TA2 ... TAn);

(ii) #S0 is matched against (TA1 TA2 ... TAn);

(iii) =(S0 cond-S0) is matched against (TA1 TA2 ... TAn).

In these cases the source element is uniquely replicated n times so that we try matching

(?S01 ?S02 ... ?S0n) against (TA1 TA2 ... TAn);

(#S01 #S02 ... #S0n) against (TA1 TA2 ... TAn);

(=(S01 cond-S01) =(S02 cond-S02) ... =(S0n cond-S0n)) against (TA1 TA2 ... TAn).

Then the source macro itself is matched against the list (S01 S02 ... S0n) formed as a result of the matching; that is,

?S0 against (S01 S02 ... S0n);

#S0 against (S01 S02 ... S0n);

=(S0 cond-S0) against (S01 S02 ... S0n).

Any other multilevel ambiguities are handled by the rules for pattern matching or the macro conflict table restrictions.

A more serious conceptual problem involving macros can arise because of the current left-to-right matching of message pattern elements against receiving pattern elements. Assume there is a macro within a pattern that is a call to a computation full of messages to other |PEXPRs and other side-effects. Assume further that the pattern macro is executed during a pattern match and returns a value which successfully matches the corresponding element of the source pattern. Finally, assume that the pattern match later fails on some other element. Then, all the side-effects of the first macro, including execution instances built during execution, erroneous patterns asserted, and so on, are still around!

There is no way around this in |LISP, but with sensible precautions it turns out that the problem can be circumvented. These precautions involve making sure that a |PEXPR has only a limited number of "procedural" patterns and that these have unique first elements. Possible corrections to |LISP itself could be made, such as for example, matching all constant elements first, or re-designing the matcher to have complete back-up capabilities including the ability to undo things. However, this is not an urgent priority, and, in fact, it's open

to debate whether it should be since a fundamental philosophy of |LISP is keeping around all information, including blind alleys, for later perusal.

3.5 Pointers and Searching

Many times I've spoken of "links" or "pointers" between pattern expressions. I've relied upon an intuitive grasp of these terms to get across most of what I have in mind, but there is a much more precise meaning for "pointer" in the system. Before giving it, however, I would like to bring out a couple of interesting features of patterns in general.

One way of regarding a pattern is as essentially an n-ary predicate whose head is a relation and the rest of whose elements are the objects filling the relation's slots. In this view, a pattern derives its meaning primarily as a result of how it matches other patterns, although it can be treated as a more procedural entity both in failure to match processing and in the action of macros associated with its elements.

Another equally useful view of a pattern is as an n-ary "link" among objects in a semantic network. The object containing the pattern is the source of the link; the pattern elements are the objects connected by the link. Thus, the pattern (TRADE SELF TICKET-BUYER MONEY TICKET) occurring in the BUY pattern expression might be diagrammed as shown in Figure 3.2, although many other network realizations are possible.

Thus, the system can be viewed as a large semantic network of nodes (objects) and arcs (patterns) connecting the nodes. From any particular pattern expression, only the closely connected nodes can be accessed directly (i.e. any objects occurring in patterns of the |PEXPR body can be "seen").

The usual semantic network allows only binary links between objects, and for good reason: they are often (although not always) the most important kind of link. Moreover, they are easily understood because they break down into three more or less well-defined parts (the source node, the destination node,

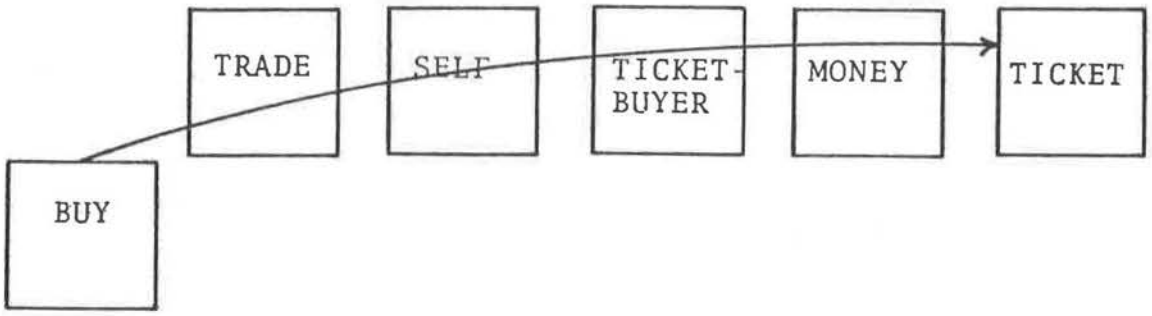


Figure 3.2 - A Small Network

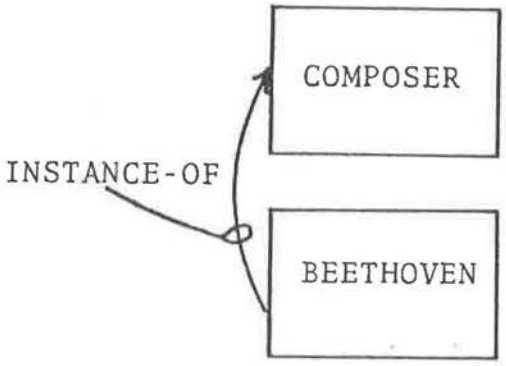


Figure 3.3 - INSTANCE-OF Link

and the arc itself) and also because they can be drawn, allowing people to use their spatial intuition in understanding them.

In this representation scheme certain kinds of patterns called pointers take the place of binary arcs. A pointer is a three element pattern, the first element of which is the arc and can be any relation, the second element of which is the source and must be the |PEXPR containing the pattern, and the third element of which is the destination and can be any object, macro expression, or sub-pattern. Thus,
(INSTANCE-OF BEETHOVEN COMPOSER) occurring in BEETHOVEN would be a pointer and could be graphically illustrated as in Figure 3.3. This would be termed the "INSTANCE-OF pointer from BEETHOVEN to COMPOSER". BEETHOVEN may also have a pattern

(STATURE BEETHOVEN !(|POINTER STATURE 'GREAT-MAN))

which would be a pointer to the macro expression

!(|POINTER STATURE 'GREAT-MAN)

(unevaluated); or could have

(WANT BEETHOVEN (LIKE ↑PEOPLE ↑BEETHOVEN-COMPOSITIONS 10))

which would be a pointer to the unevaluated sub-pattern

(LIKE ↑PEOPLE ↑BEETHOVEN-COMPOSITIONS 10).

It should be noted that the pointers described here are one-way pointers. Pointers to objects do sometimes have inverses, but they must be explicitly stored in the destination object. Thus, if (R X Y) is a pointer in X, then (R-INV Y X) would be stored in Y if R-INV were the inverse of the relation R. For example, (SUPERSET CAMEL DROMODERY) in CAMEL might have a corresponding (SUBSET DROMODERY CAMEL) in DROMODERY. Now, fortunately when a pattern is asserted, the |ASSERT |EXPR (see Appendix I) executes an IF-ADDED procedure associated with the head of the pattern. This IF-ADDED procedure should know enough to check if the pattern is a pointer, if so to check with the head of the pattern to see if there is an inverse, and if so to assert the inverse pattern in the destination object. Any other IF-ADDED processing could also be undertaken.

In the example, then, the relation SUPERSET might contain a pattern (INVERSE SUPERSET SUBSET), thus allowing |ASSERT to

effect the appropriate inverse pattern. This explicit inverse facility is useful in allowing the system to analyze its own relations (i.e. it can "see" what the inverse of a relation is), in forcing the user to recognize the issue of inverses rather than hiding it with automatic inversing, and finally in enabling the user to choose names for the inverse relation that are appropriate to it.

Pointers are very useful for accessing entire environments surrounding a pattern expression; that is, pointers of a particular type z can be followed from A to {Bi} then pointers of the same type z can be followed from {Bi} to {Ci}, and so on. All objects so accessed are said to be in the z-environment of A.

There are many possible environments in the system: PART-OF, THEN, ISA, EX-ENVIRON, etc. Special search routines have been devised to access data in the latter two. The ISA-environment is the name for the environment that can be accessed by following EX-INSTANCE-OF, or SUPERSET pointers from some |PEXPR. It is the traditional "generalization" hierarchy into which searches can be directed for knowledge that has been abstracted from instances. Such a search is called an ISA search and is carried out breadth-first until eventually it converges on the top object. As Fahlman (1975) has pointed out, the convergence property of ISA hierarchies, combined with their relative shallowness (he argues no more than 20 levels of ISA) gives hope that such searches won't be explosive. Moreover, if the ISA link isn't overused to delineate every kind of dependency, the branching factor should be fairly small, further enhancing search times.

The other major kind of search is into the dynamic or execution environment (EX-ENVIRON) surrounding an execution instance. In this environment are the supergoals for an object and the top-down decisions that have been made to this point. It thus forms a notion of context and is consequently often searched for information as to purpose, current status of certain features, and so on. Such a search is called an

EX-ENVIRON search and is linear with the number of |PEXPRs in the dynamic context.

These two search types are so vital, that the special non-pattern-matching macros "▲" and "%" have been set up to direct them. Thus,

▲(w x y z), if seen by |EVAL in OB1 will result in an ISA-search (breadth-first) from OB1 for a pattern matching (w x y z).

%(w x y z), if seen by |EVAL in OB1, will result in an EX-ENVIRON search from OB1 for a pattern matching (w x y z). These macros should not be confused with the pattern matching macros which are only seen by the matching routines and some other select operations. "▲" and "%" are merely convenient shorthand for actual calls to the general |SEARCH routine (see Appendix I). Thus they can be used wherever a regular call to |SEARCH could be employed.

There is still another non-pattern-matching macro, "¬" which is merely a "'" with the difference that all "!", "$", "%", "▲", "¬", ":", and "/" macros are executed inside it. Thus, ¬(A !B C) would be the same as '(A BLARNEY C) if the value of B were BLARNEY in the context of the executing |PEXPR.

3.6 Interrupts

In section 3.3.2 (describing the message passing behaviour of |PEXPRs), it was mentioned that some message patterns are fundamentally instructions to the interpreter rather than patterns to be matched in the body of the |PEXPR. One example was given showing how the interpreter intercepts a (|PRIO= number) message and asserts it in the body of the receiving |PEXPR. There are several other patterns of this kind which essentially allow a super-|PEXPR to impose certain limits on the execution of a sub-|PEXPR.

One such pattern is (|TIME= number) which puts a limit on the time a |PEXPR has to answer a message. The receiving |PEXPR has "number" units of time, where the time is measured in 100s

of calls to |EVAL (e.g. number = 1 implies that there is a limit of 100 clock units imposed on the sub-object's perambulations). When this limit is exceeded, control returns to the |PEXPR which set the limit, and it must decide what to do. Among its options are to give up, to restart the sub-object (in which case the same limits apply again), or to perform further computations in an attempt to ascertain what to do.

In more detail, this is what happens. When a (|TIME= n) pattern is discovered during message passing, the interpreter asserts the following two patterns in the body of the receiving execution instance:

(|TIME-LIMIT= receiving-ex-instance n)

(|TIME-NOW= receiving-ex-instance n)

Every 100 times through |EVAL, the EXPR |UPDATE-TIMER reduces all |TIME-NOW= patterns in the execution environment by 1, i.e.

(|TIME-NOW= receiving-ex-instance (n-1))

replaces the previous pattern. Note that this means that the |EVAL count could be out either way by up to 99 |EVALs since all |TIME-NOW= patterns are reduced at once (a simplification imposed for efficiency). When any given execution instance has a |TIME-NOW= pattern reduced to 0, then an interrupt is generated, |TIME-NOW= is reset to |TIME-LIMIT= (for the particular interrupted execution instance only), and the immediate super-goal (in the execution environment) of the interrupted execution instance is restarted. The return condition set for this kind of return is

(|RETURN-COND ex-environ-inst
    |TIME= (interrupted-ex-inst current-ex-inst) n)

asserted in the super-goal. If the super-goal eventually tries to restart the computation the same interrupt conditions apply. (Note also that it must go all the way to the executing ex-instance rather than the interrupted ex-instance, since the executing ex-instance was actually processing at the time of interrupt).

The timing mechanism described here is quite crude, but nonetheless is useful. |TIME= interrupts are useful for

allocating time resources, especially in simulated parallelism or for running in "careful" mode, where a process can be run for a short time before its caller re-imposes control to make sure it is proceeding on course.

There is another kind of condition that can be imposed by one pattern expression on its descendants: the (COND= pattern) message pattern. It essentially states that anytime there is a pattern matching the |COND= pattern asserted by the sub-goals then an interrupt should return control to the super-goal which set the condition. Much the same kind of processing occurs here as for the |TIME= interrupts. That is, a

(|COND= execution-instance pattern)

is left in the relevant execution instance, and everytime the |ASSERT function is called, it looks into the execution environment for any matching |COND= patterns for the |ASSERT pattern. Any such match results in an interrupt being generated, a return condition

```
(|RETURN-COND ex-environ-inst
   |COND= (interrupted-ex-inst current ex-inst) pattern)
```

being set in the super-goal to the interrupted execution instance, and finally, control being resumed in the super-goal. As in the |TIME= case, enough information stays around for the whole process to be started again if desired. |COND= limits are helpful when it is known that a certain sub-goal is going wrong if it asserts certain patterns..

A final pattern of this type is (|END= n) which just gives a limit on the number of times a particular sub-goal can be interrupted before it can no longer be restarted. Thus, a (|END= 5) message pattern would allow no more than 5 restarts of an interrupted sub-goal. After a |END= limit has been exceeded, the return condition is

```
(|RETURN-COND ex-environ-inst
   |END= (interrupted-ex-inst current-ex-inst) val-int)
```

where "val-int" is a time limit or a pattern depending on which kind of interrupt sent the |END= limit over the top. This is useful in eliminating potential infinite loops of sub-goal restarts.

One final point of interest: the EXPRs |SEMON and |SEMOFF will turn a flag (semaphore) on or off in the current pattern expression. Before doing any |TIME= or |COND= interrupt checking, the current status of the nearest such flag in the execution environment is determined, and if it is on, no interrupts are processed anywhere in the execution environment.

3.7 Simulated Parallelism

There are two ways to simulate parallelism in the system. The first uses |TIME= limits to essentially assign time slices to each sub-goal. A |END= limit can be used to limit the number of restarts for a particular object. Thus,

```
(|PARALLEL
      (object1 (|TIME= n1) rest-of-message-patterns)
      (object2 (|TIME= n2) rest-of-message-patterns)

                  . . . . .

      (objectk (|TIME= nk) rest-of-message-patterns))
```

will send object1's message patterns to object1 for time slice n1, then will send object2's message patterns to object2 for time slice n2, and so on through objectk. The process is then repeated, restarting every object which had a |INTERRUPT-TIME= return condition for another time-slice, but, of course ignoring those with NORMAL return. This goes on until all have terminated normally or until |END= limits (if any) have been exceeded.

The other way of simulating parallelism makes use of priorities and the execute queue. Several objects can be scheduled at once with differing priorities, and they will eventually run in the order of their priorities. If these are to be restarted, however, they must be explicitly resumed by the pattern expression which scheduled them. The EXPR |SCHEDULE can be used to accomplish this:

```
(|SCHEDULE object1 object2 . . . objectk)
```

Note also that the objects must somehow be initialized with their messages and so forth in place. This method doesn't tend

to be too convenient or to give much control over resource allocation for each object, but it is useful for a much quicker kind of breadth-first ability than the time-slice method gives.

This concludes the description of the capabilities of the current version of |LISP. In Chapter IV I would like to evaluate the scheme by comparing it to other approaches and by looking at its strengths and weaknesses.

CHAPTER IV
An Evaluation of the Representation Scheme

The system for representing knowledge presented in the last chapter has been described more or less at face value without any attempt to evaluate it. In this chapter I would first like to compare the scheme developed here to some other approaches, particularly the PLANNER / CONNIVER language development and the frame proposal in its many guises. In this way the influences on this research should be revealed and the strengths and weaknesses should become clearer. I would then like to focus back on the representation and language issues to see if the approach outlined here does indeed shed light on any of them.

4.1 The PLANNER / CONNIVER Approach

The most important influence on |LISP has been the PLANNER / CONNIVER (Hewitt (1972), Sussman and Winograd (1970), McDermott and Sussman (1974)) development, together with the Bobrow and Wegbreit (1973) control paradigm. Since CONNIVER embodies most of the crucial aspects of this line of research, I will mainly use it for comparison. The essential features of CONNIVER are

(i) a data base of contexts, each containing assertions and methods, and each representing some state of the world;

(ii) a pattern matcher, replete with special macros, to access data in this data base;

(iii) a varied set of procedures to manipulate the assertions;

(iv) a co-routine control structure to implement all this.

I would like to look at |LISP's capabilities in each of these areas.

4.1.1 The Data Base

The major difference between |LISP and CONNIVER with regard to the data base is the separation of context from control. In |LISP there is no separate context mechanism: contexts are inextricably bound to pattern expressions (since pattern expressions are the primary object of |LISP, their features will be used almost always in explanations and justifications), and cannot exist independently. This is, I feel, an important conceptual point (also illustrated in many recent systems, e.g. Bobrow and Winograd (1976), Hendrix (1975), Havens (1978), Sandewall (1975)): all data is associated with some definite "object" and has meaning only when that object has meaning to the system. To be sure if it is desired to tie a context to a particular access environment, CONNIVER can use a variable CONTEXT that can be set in that environment to point to the context, but this is both awkward and ignores the usefulness of considering contexts to be objects that can be reasoned about, sent messages to, and otherwise treated like other objects. The assumption of object - context identity is one of the more unifying concepts in the system, and eliminates the distinction between data in one place and procedures operating on the data in another.

Contexts also differ from pattern expressions in that contexts are arranged in a "visibility hierarchy" while |PEXPRs are on the surface all invisible to one another. But, this is only on the surface: |PEXPRs consist of patterns which can contain elements that are the names of other |PEXPRs, and hence there can be, in essence, arbitrary links between |PEXPRs. These links can be traversed by sending messages along them and receiving replies, thus making data in |PEXPRs available to one another. Although this is powerful, it does tend to be somewhat slow for simple data access; hence, the |LISP provision of special execution and ISA environment searches that speed up at least a couple of the more commonly needed access methods.

4.1.2 Pattern Matching

|LISP's matcher varies little in any theoretical way from CONNIVER's. There are, however, a couple of distinctions. First, the macros in |LISP are somewhat different and seem to provide a bit more flexibility (especially "=" and "#"). Second, the matcher is totally symmetric in |LISP, displaying no preference to source pattern over target pattern. In fact there is no distinction between patterns and assertions, further unifying the system. But, most important is the "don't give up" feature of |LISP's matcher: if it is unable to find a match within the body of a |PEXPR, it may (at the discretion of the pattern head) look elsewhere for a match. This gives robustness and power to the matcher; it also has the drawback of potentially getting out of hand combinatorially, like any other such automatic feature. With careful selection of failure to match conditions, I hope that this drawback will remain only potential!

4.1.3 Procedures

In CONNIVER there are at least four distinct procedure types: ordinary LISP functions, CONNIVER procedures, methods, and generators. In |LISP there are also several different procedure types, including LISP functions, |LISP procedures, and |PEXPRs. The interesting comparisons are between methods and generators on the one hand and |EXPRs and |PEXPRs on the other. Methods are part of the data base and work to keep it consistent as well as help to access data from it. The problem with them tends to be one of combinatorial explosion since they are called in automatically by pattern matching rather than in a more controlled fashion.

IF-ADDED and IF-REMOVED methods have been fairly directly incorporated as procedural patterns associated with selected |PEXPRs (i.e. those whose name can appear as a pattern head). Such "methods" are invoked whenever a pattern with that head is |ASSERTed or |REMASSERTed. This differs somewhat from CONNIVER in that the pattern matching to find a method is restricted to

only scanning patterns of the head |PEXPR, not the entire data base. Of course, the method itself, once activated, could do any amount of (possibly explosive) further processing, but since I am not overly concerned about the existence of inconsistent information (as long as it isn't present in the same execution environment), I don't visualize such methods as being too elaborate.

CONNIVER IF-NEEDED methods have an analogy in the failure to match processing of |PEXPR message passing; that is, a failure to match "method" is called in only if it is needed because of a lack of suitable patterns in a receiving |PEXPR. The difference between IF-NEEDED methods and failure to match processing is once again that the search for a FAILURE-TO-MATCH pattern requires the matcher to only look through the "head" |PEXPR rather than the entire data base.

Generators are CONNIVER co-routines which can be executed until they produce some datum, and can later be re-entered if the datum proves unsatisfactory. Both |EXPRs and |PEXPRs give a similar co-routine ability, |EXPRs by allowing the saving of stack pieces within a |PEXPR, and |PEXPRs by keeping an execution instance around which can later be restarted. For efficiency, the usual way of doing generation is by using |EXPRs; in fact the |SEARCH generator capabilities are implemented in just this way.

The important distinction between |LISP and CONNIVER procedures is, however, the centrality of their role. CONNIVER is set up as basically a LISP-style procedural language with occasional forays into pattern directed invocation as a necessary declarative component. |LISP on the other hand, is basically built around pattern matching, and the procedural capabilities drop out as more or less a side effect of this matching. Thus, most procedures are in fact embedded as "!" or "$" macros in patterns of |PEXPRs, not as separate functions. This means they are treated much as any data would be (e.g. they are present or absent only if the |PEXPR containing them can be looked at). Hopefully, this is one small step along the road to

program / data symmetry, although many global and distinctly functional objects remain (e.g. EXPRs, SUBRs, |EXPRs).

4.1.4 Control

CONNIVER's control is fundamentally that described by Bobrow and Wegbreit (1973). This is also true of |LISP: that is, CLINKs are equivalent to EX-ENVIRON pointers; execution instances are Bobrow and Wegbreit frames (except they aren't separated into two parts, and hence there is no BLINK); there is a continuation point, return condition, etc. similar to that of Bobrow and Wegbreit. Of course, there are differences. There is no ALINK since its data accessing function has been split between the CLINK and more semantically relevant (to the domain being represented) links such as SUPERSET, INSTANCE-OF, etc. That is, it is often necessary to access data in a variety of environments such as execution, ISA, etc., rather than in one all encompassing access environment.

The specific ability that the ALINK / CLINK distinction provides of allowing a process to access data in one environment while returning control to another can be accomplished in |LISP by sending a message to an old named execution instance. To handle this message a new execution instance of the old execution instance must be created. The old execution environment can be accessed using the old execution instance's CLINK; the new execution environment can be accessed using the new execution instance's CLINK. Having two execution instances is necessary if old "episodes" are to be kept distinct from new "episodes" (see Appendix I - |RESTART - for a description of how this methodology can be used to resume execution of a previously suspended |PEXPR).

This illustrates another major difference between |LISP and CONNIVER: execution instance data, including internal pointers, are stored as patterns like any other data in any other |PEXPR. In fact execution instances are |PEXPRs like any others. Thus, the patterns can be accessed using standard matching, and moreover, old execution instances can be queried to provide

episodic information about what went on in that context. Still anctner distinction between |LISP and CONNIVER control is that execution instances stay around rather than disappearing upon return and must be "intelligently" garbage collected at some later date.

In CONNIVER all procedures (with the exception of certain macro-directed LISP calls) have a Bobrow and Wegbreit frame created for them; this is not the case in |LISP. Except for messages between |PEXPRs, an execution instance is not set up upon a call to a procedure, the current |PEXPR stack being used instead. Variables are accessed on the current |PEXPR stack, and if not found there are bound in stacks of |PEXPRs in the execution environment of the current |PEXPR. This use of |PEXPR stacks is somewhat less wasteful on space and time in the many cases when |PEXPRs are not communicating; in the few cases they are, however, the overhead of initializing and later accessing data in patterns probably more than makes up for the saving in the majority of cases.

Another difference between LISP and CONNIVER control is the explicit use of a scheduler to buffer contact between |PEXPRs. Of course, it is fairly easy to implement such a scheme in CONNIVER (as was done in the Reference Manual (McDermott and Sussman (1974))), but |LISP explicitly does so. While engendering some overhead, such a scheme enables pseudo-parallelism, and adds flexibility to the system.

Related to this is the |TIME= and SPACE= conditions that can be imposed on the execution of a |PEXPR. CONNIVER has an interrupt feature that seems to be mainly useful for error generation and co-ordinating various conflicting methods. The one here is more fundamental to the problem of a super-goal limiting the allocation of resources to a sub-goal or otherwise imposing conditions on that sub-goal. This is still only a crude approach (being slow, cumbersome, and inexact), and more sophistication is needed if a truly useful interrupt feature is to be installed.

4.2 Frames

In this section I consider work which has gone under various labels: frames, schemata, scripts, knowledge sources, social action paradigms, etc. Among other things this work is centrally concerned with dividing a knowledge base into large modular units each of which contains all the information relevant to a particular concept. In ILISP, pattern expressions are analogous to frames, since they represent large chunks of semi-independent knowledge.

There are many issues raised by the various frame proposals and I would like to concentrate on a few important ones: why are frames needed, what do frames look like, and how do they connect to other frames both statically and dynamically. I intend to compare various other approaches to ILISP in the hopes of illustrating some of its contributions.

4.2.1 Why are Frames Needed?

As Winograd (1974) has suggested, a system must steer a middle course between having lots of local heuristics that help it decide precisely and efficiently what to to in various specific contexts (but perhaps make it incomprehensible), and using a few widely applicable techniques which are easy to understand because there aren't many of them (but which leave the system vulnerable to uncontrolled computation). Frames, hopefully, help to resolve this problem first because they give a way of packaging information into distinct modules that can be considered separately, thus keeping the system comprehensible; and second because they tend to represent much of their information declaratively while still allowing enough procedural information to ensure reasonable processing times. Winograd (1975) has called the issues raised by the latter trade-off the procedural / declarative controversy.

I feel that pattern expressions contribute some things of worth here. Certainly, they allow information to be packaged into groups of patterns which contain the basic "facts" about some concept, hence allowing the concept to be (at least

partially) considered apart from its comrades. Thus, they are quite well suited to the first role above. But what about information that isn't directly in a |PEXPR but must be accessed from some other object - is there too much of this kind of inter-object activity to maintain the modularity? This, of course, is a crucial problem for all frame systems, and I can only say that I share the general frame faith that the packaging can often be done successfully.

The second contribution of frames is to procedural / declarative issues. In this respect there seem to be two main views of frames:

(i) Frames are uniform, mostly declarative structures which are processed by a single global interpreter. Most of the purely procedural aspects are subliminated into the interpreter and more or less hidden from the user. Systems of this ilk include Schank and Abelson (1975), Charniak (1975), and Bruce (1975).

(ii) Frames consist basically of two parts, one a mostly declarative section containing static information, and the other a procedural component which is used to interpret the declarative portion and otherwise handle inter-frame communication. Among systems of this genre are MAYA (Havens (1978)) and KRL (Bobrow and Winograd (1976)).

Although there are aspects of both of these approaches in |LISP, the main thrust of the scheme is to lower the procedural aspects of a |PEXPR to the pattern level. Thus, procedures are encoded as "!" or "$" or "=" macro elements which are expanded when the pattern is being used to answer a message. Such macros allow a pattern to represent procedural kinds of goaling information or to represent very specific context dependent knowledge directly with the pattern that needs to use such knowledge so that the |PEXPR as a whole doesn't have to be concerned with it. Since such patterns are accessed using the same matching scheme as non-procedural patterns, a basic procedure / declarative unity begins to emerge. Furthermore after a |PEXPR has answered a message, the resulting answer

pattern is |ASSERTed in the receiving execution instance as a declarative residue of procedural activities.

Thus, use of pattern level macros allows procedural flexibility to be inserted into examinable, mostly declarative structures. It is nonetheless important for anybody using |LISP to attempt to abstract such low level procedures into a more declarative plane by somehow representing what they do in terms of higher level primitives. This is a difficult task but at least |LISP gives the modeller the choice of level at which to represent knowledge.

4.2.2 What do Frames Look Like?

There are as many different representations for frames as there are frame systems. I don't intend to go into a lengthy discussion of the syntax or the semantics of the various schemes; instead, I would like to discuss some pervasive frame concepts, specifically slots, preconditions, post-conditions, and finally some issues in representation of knowledge that frames illustrate.

Minsky (1974) has proposed that a frame is composed of fixed information at the top levels and slots at the bottom levels which must be filled when the frame is instantiated. A slot has associated with it markers or other indications as to how to fill it, the importance of the slot to the frame, what to do when the slot is filled, etc. Winograd (1975) and Bobrow and Winograd (1976) have generated a comprehensive computational description of slots, complete with attachment of procedures to fill the slots (perhaps by inheriting such procedures from other frames), the designation of important slots (so-called IMPs), the provision of procedures that are executed once the slot is filled, default values of varying "looseness" that fill the slot if nothing else can be found to do so, etc.

In pattern expressions there are no slots; at least, there are no explicit distinctions made between patterns which are unvarying in the sense they contain no macro elements and patterns which are changeable because of macro elements. The

role of slots is taken by such macro elements. These "slots" are filled when (and only when) the |PEXPR is being queried for information by some other |PEXPR. They can be filled by simple variable binding (as in the macro "?") or filled by complex procedure (as in the macro "!") or filled under certain conditions (as in the macro "=") etc. And if there is no pattern in the |PEXPR which matches the incoming pattern, then a procedural attachment from some other frame can occur (if the pattern head recommends it). Certain patterns can be indicated to be more important by using meta-patterns that contain them (e.g. (IMPORTANCE (TRUNDLE ELMER ROOM4) 9)). This use of meta-patterns is discussed more fully in Chapter VI, but it does provide a way to designate IMPs (even though I haven't found such designations to be all that helpful in the well-constrained examples I have explored). So, in |LISP the idea of slots is generalized so that they can be viewed as any other data is viewed.

Many frame systems (e.g. Bruce (1975), Charniak (1975), Schank and Abelson (1975)) talk in terms of pre-conditions which are rapid checks that must be satisfied before a frame can be considered to be relevant and post-conditions which are the residual effects of the frame's activation. These conditions are usually in declarative form so that they can be easily understood by the user and so that they can be matched against the data base on entry (for pre-conditions) or asserted in the data base on exit (for post-conditions). I believe there is something quite inflexible about the pre-condition / post-condition scheme as usually outlined: such processing is done all at once without regard to context. It seems more practical to provide some much more procedural capability where a frame is able to pick and choose which tests to carry out in deciding whether it is relevant. Moreover, it should be able to be contextually selective in regard to what it leaves as residue once it is done. Of course, these various tests would have to be fairly simple or the whole purpose of pre-conditions (and to a lesser extent post-conditions) would be

obviated.

In |LISP there are no explicit pre-conditions or post-conditions, although, as always, particular users have the capability of adding (PRE-CONDITION -------) or (POST-CONDITION -------) patterns to pattern expressions. What little pre-condition style testing there is is undertaken by procedures within elements of patterns as they attempt to respond to messages; post-condition style processing occurs when these procedures as a side-effect assert new patterns (usually in the body of the current execution instance). The key point is that the processing can be undertaken when wanted by the relevant receiving pattern, and isn't restricted to an all-or-nothing lump associated with the entire |PEXPR regardless of the receiving pattern.

This explanation has suppressed a rather important use of pre-conditions: their significance in pattern-directed invocation; that is, it is usually something like pre-conditions that are used when testing for the relevance of any frame before letting it loose. While for the areas in which |LISP has been used so far it has usually sufficed for |PEXPRs to know each other by name, it is a crucial ability to be able to invoke |PEXPRs in some less direct way. To this end, an associative activation scheme is in the early stages of development. It essentially allows |PEXPRs to contact each other along "links" in the implicit semantic network in which they are embedded (see section 4.2.3 for a fuller explanation of this semantic network). Associative activation itself is described to a greater extent in Chapter VI.

At this point I would like to consider some important issues about how information is represented in a frame. First, consider the distinction between concepts which directly concern the frame and concepts which are not so central. This is basically Woods' (1975) discrimination between definitional properties and assertional properties, a distinction which has been explicitly included in some systems (e.g. Levesque (1977), Schneider (1978)). I believe that the

assertional / definitional categorization is really a catch-all and should be split along many dimensions, each of which delineates a different aspect of a concept's relevance to a frame. This is not a new idea, so that Tesler et al (1968) outline several such dimensions (e.g. charge, significance, credibility, foundation), Winograd (1975) talks about IMPs, Becker (1969) proposes criteriality. In |LISP, of course, no commitment has been made to the particular dimensions to choose (although IMPORTANCE and CORE are two such dimensions used by some system procedures); but the user can specify any that he desires by using meta-patterns (see Chapter VI).

Another issue arises in systems (such as, for example, KRL (Bobrow and Winograd (1976)) or MAYA (Havens (1978))) which encode information concerning the containing frame in attribute / value pairs, while other information is encoded in assertions. Thus, the facts that CASA-LOMA is a castle and is in Toronto would be kept in the CASA-LOMA frame as attribute / value pairs (ISA CASTLE) and (LOCATION TORONTO) while another relevant piece of data (ISA PARAPET TOWER) would be stored in CASA-LOMA as a three element assertion. While often a useful abbreviation, such an explicit syntactic difference can be harmful in that the same piece of information may have to be represented in two different ways depending on where it is stored (e.g. it would be sufficient to use the attribute / value pair (ISA TOWER) in the frame PARAPET). In |LISP all such information is encoded in "full blown" patterns, since it it important to failure to match processing that the number of "arguments" of the pattern that has failed be consistent.

The use of such full blown patterns also helps point out a problem which many systems finesse by the use of attribute / value pairs. This is the problem of distinguishing information that is about a frame from information that is about a frame's instances. For example, an attribute / value approach might put (NUMBER-OF-LEGS 2) with the PERSON frame and be totally unambiguous. But when it is realized that what is

really being said here is (NUMBER-OF-LEGS PERSON 2), then a difficulty becomes apparent: the class PERSON does not have 2 legs; instances of the class do. |LISP forces this kind of distinction to be made explicitly, e.g. (NUMBER-OF-LEGS ↓PERSON 2). Even with attribute / value pairs, a system will fail if something like (SELLS ALCOHOLIC-DRINK) were to be associated with the BARTENDER frame. What is really meant here is that an arbitrary instance of the class of bartenders sells an arbitrary instance of the class of alcoholic drinks, i.e. (SELLS ↓BARTENDER ↓ALCOHOLIC-DRINK). To be sure, frames encoding the SELLS or NUMBER-OF-LEGS relations might keep this information straight, but for now, at least, it seems to me to be better to be explicit about what information is about instances and what about objects.

The more so since this has some relevance to a closely related issue: the problem of discriminating "meta" information about the frame itself from "real world" information about the object represented by the frame. This distinction has not always been well delineated in AI or elsewhere for that matter. Keeping information about instances distinct from information about classes helps somewhat here. However, it doesn't solve the whole problem: the pattern (NAME FRAME27 ALFRED) could mean that the instance FRAME27 itself is named ALFRED or that the person referred to by FRAME27 is named ALFRED. Which is right depends on what is meant by NAME in the system. If NAME means "the name of the object represented by the second element is the thing in the third element" then fine; but there had better be another relation INT-NAME, say, indicating "the name of the internal object that is the second element is the third element", so that (INT-NAME FRAME27 FRAME27) could be used. The burden for keeping this straight is on the system user, although once he has decided what's what, he can put information with the relation itself indicating whether it is a "real world" or "meta world" relation.

When two |PEXPRs are being compared, it is often inconvenient to query each pattern head as to whether it is

relevant to the comparison. Thus, a meta-pattern

(CORE ZORK /PAT1 /PAT2 . . . /PATn)

indicating sub-patterns which are the core concepts of ZORK can be asserted. This, too, is useful in delimiting internal from domain information. No doubt other meta-patterns could be built to further specify finer distinctions, but this hasn't been of central concern to the research.

Next an interesting point regarding the type / token distinction should be noted. When an old execution instance is sent a message, in order to respond to that message an execution instance of that execution instance is set up just as it would be for any message to any other |PEXPR. This allows the old execution environment to be discriminated from the new (see section 4.1.4); it also means that effectively there are instances of instances in |LISP. This approach is similar to the uniform subset / superset designation of TLC (Quillian (1969)), differing only in that in |LISP the user is encouraged to make a distinction between an instance, which represents an individual, and other |PEXPRs which are not individuals.

4.2.3 Inter-frame Connections

Because a frame cannot know everything about everything, an important aspect of frame theory involves the manner in which frames pass information amongst themselves. There seem to be at least two different aspects to consider: static connections and dynamic connections. In the former case, certain static pieces of information are contained in several frames at once; in the latter case, one frame explicitly calls in another frame to achieve some purpose or gain some information.

There are two basic issues that concern static connections: the sharing of static information amongst frames, and the presence of certain static links connecting frames. Minsky (1974) and later Charniak (1975) talk in terms of collections of related frames "sharing terminals", that is when a slot is filled in frame A as it recognizes a scene, the same

slot may also be filled in frame B automatically. Such shared information allows frame B to save work if it eventually must be called in to recognize the scene. This ability also is very useful in deciding what frames are related to one another; that is, if two or more frames share the same information, then this is persuasive evidence that they are somehow related.

In |LISP "terminal sharing" amongst |PEXPRs is not encouraged: patterns are contained in "boxes" which aren't supposed to share their information with one another. Sharing of patterns should only be undertaken by message passing between |PEXPRs. This means that if a |PEXPR turns out to be unsuited for a particular task, then any patterns it has asserted will not be automatically copied over to the new |PEXPR which is called in to replace the unsuitable one. Instead, the new |PEXPR is given the perogative to decide what things, if any, it may attempt to salvage from its comrade. This is a procedural approach to achieving "shared terminals". It is not altogether satisfactory since, although it allows arbitrary selectivity and preciseness in information sharing, it may be too complex to be useful. A purely declarative approach, on the other hand, probably wouldn't have enough selectivity, implying that some middle ground should be found.

A similar kind of sharing happens in the Hendrix (1975) semantic network formalism, where a net is divided into partitions which can share information. Such a division allows the same information to be "visible" from certain perspectives but not from others, and hence is of facility in perusing only those features which are relevant at any time. This is not the place for a full discussion of context (see Chapter VI), but it must be pointed out that the "partitions" of |LISP are the pattern expression "boundaries", and that there is thus no sharing of data <u>per se</u> between partitions. However, as in the case of shared terminals, a procedural solution involving message passing can be (rather unsatisfactorily) pressed into service whenever information is to be shared. It is safe to say that the lack of shared information is one of the weaker aspects

of |LISP as it stands today. The attempt to keep the system as modular as possible has perhaps been too severe and although it has meant greater precision by forcing procedural sharing, it seems to have contributed to rather than helped to solve the complexity problem.

The other static aspect of inter-frame connections is the attempt to essentially embed frames into a semantic network. Minsky's (1974) proposal suggests a similarity network connecting frames, the links of which would be "differences" to follow if the frame were not properly matched to a situation. Winograd (1975) has been keen on investigating how and whether frames fit into a Quillian (1969) style generalization (ISA) hierarchy; Levesque (1977) has also been interested in such a hierarchy as well as other "links" between frames; and Havens (1978) has a full-scale net surrounding his frames. These researchers have recognized that it is critical to the efficient operation of a system that a frame have semantically close neighbours that it is able to quickly access in a variety of common situations (e.g. inheritance of properties along ISA links, following a difference pointer when a certain kind of failure occurs within a frame).

|LISP pattern expressions can also often be profitably viewed as if they were embedded in a semantic network, but as will be shown, it is a rather strange network. This can be accomplished by viewing the patterns of a |PEXPR as (arc node-1 node-2 . . . node-n) combinations. Even though |LISP doesn't insist on this view (that is it would, for example, be possible to look at a pattern as (node-1 arc-1 node-2 . . . arc-n) or some such), it is in most cases convenient to consider the first element to be an arc and the rest nodes. In particular, the arc-first notation is most appropriate for patterns which are pointers (see section 3.5); that is the patterns (SUPERSET DOG MAMMAL) in DOG, (SUPERSET MAMMAL ANIMAL) in MAMMAL, and (SUPERSET ANIMAL PHYSOBJ) in ANIMAL could be viewed as in Figure 4.1.

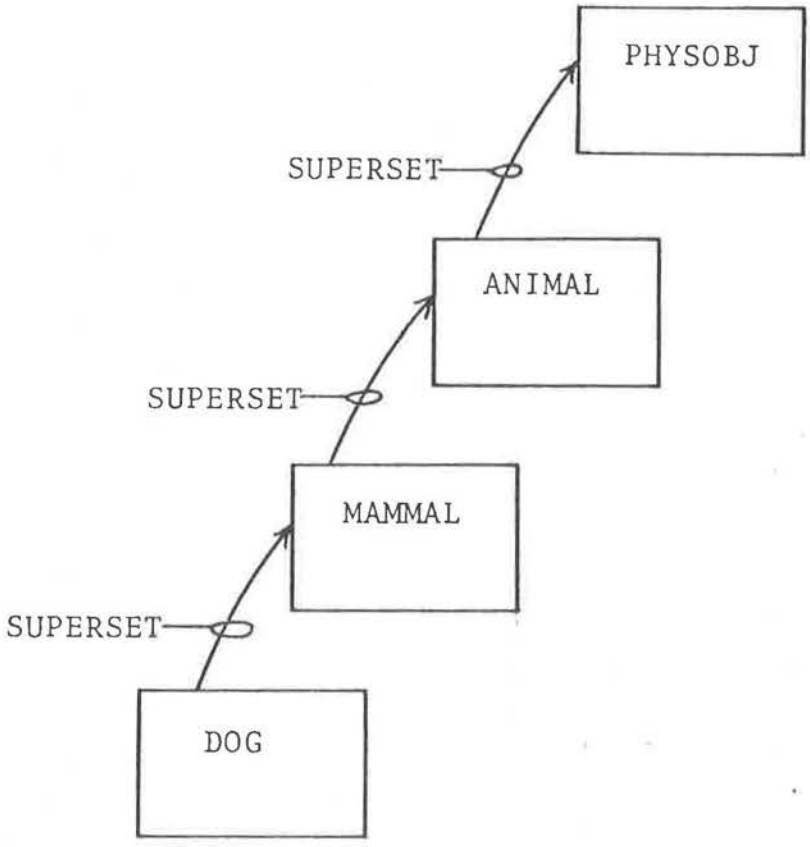


Figure 4.1 - SUPERSET Pointers

Although pointers are important, and do illustrate a relationship between |LISP and semantic networks, it must be remembered that they are only one particularly nice subset of the range of useful patterns available for use in a |PEXPR - there can be n-ary relations, relations which don't involve the name of the |PEXPR in which they are contained, relations with macros, etc. These are rather hard to draw (and hence somewhat opaque), but they do share the connectivity property so important to a semantic network in that the only other objects which can be seen from the viewpoint of any |PEXPR are those whose names appear in patterns of the |PEXPR.

The other major aspect of interframe behaviour involves dynamic communication amongst frames to accomplish various tasks, such as matching data, achieving a sub-goal, instantiating a frame. Minsky and most other frame theorists are not particularly concerned with the many procedural components that seem necessary; but these aspects are dealt with by several groups, including Bobrow and Winograd (1976) in KRL, Hewitt (1975) in his ACTOR formalism, and Havens (1978) in MAYA. This research has led to the definition of several important concepts, among them procedural attachment, multi-processing, messages and message passing.

Procedural attachment is Winograd's (1975) term for attaching a process to a frame so that it can be "triggered" to solve some problem. Bobrow and Winograd (1976) delineate at least a couple of useful classes of procedure: servants or TO-FILL triggers used to achieve some goal such as filling in a slot (and closely related to MICRO-PLANNER THCONSEs or CONNIVER IF-NEEDEDs); and demons or WHEN-FILLED triggers used to derive the consequences of achieving a goal (corresponding to THANTEs or IF-ADDEDs). Such procedures can be directly attached to the frame they serve or even to parts of that frame, or can be inherited from other frames.

|LISP, too, has procedural attachment as an automatic by-product of the message-passing paradigm defined earlier in the chapter. The procedural attachment for objects other than

|PEXPRs is trivial (since the object body is automatically the only procedure to attach); for |PEXPRs, however, things are more interesting. There are no explicit procedures to attach, but any pattern which matches could contain procedures that need to be executed to achieve the match (servants). These same procedures can arbitrarily assert or remove patterns as they proceed and thus indirectly invoke IF-ADDED or IF-REMOVED processing. Moreover, certain |COND= interrupts have effects similar to IF-ADDED methods in that control is shifted after the addition of a pattern; but this is a very constrained kind of demon which in effect prevents uncontrolled sub-goaling in contrast to the usual uncontrolled, potentially explosive role for demons.

|LISP allows the attaching of a pattern in a variety of ways if there isn't a suitable pattern directly associated with a |PEXPR. The semantics of such attachment are precisely defined in the section on the matcher; basically, it is up to the object represented by the pattern head to decide what to do in case of such a failure. This gives a fairly general, knowledge-dependent way of attaching procedural or declarative information to a |PEXPR.

KRL also makes a commitment to multi-processing as a fundamental control paradigm. Details are not specified in the Bobrow and Winograd (1976) paper, but it appears that processes are co-operatively scheduled to run (much as are |PEXPRs in |LISP) on an agenda (execute queue) with a set of priorities attached. Unlike the |LISP execute queue, the agenda is layered to give more flexibility. KRL processes are capable of sharing a limited resource pool with one another: how the pool is incremented or decremented is left up to the processes sharing it. In contrast, |LISP defines only a couple of "resources": time and conditions on what is to be asserted in sub-|PEXPRs, and these are controlled by the interpreter or its sub-functions. The KRL approach has the advantage that a lot of automatic processing is left out at the expense of added complexity for the users if they are to keep track of resources.

|LISP avoids this complexity by restricting the whole problem of resource allocation to a small set of useful features.

The concepts of messages and message passing have been most fully elaborated by Hewitt in his many ACTOR papers (Hewitt et al (1973), Hewitt and Greif (1974), Hewitt (1975)). Havens (1978) has implemented a frame system, MAYA, employing the basic idea of messages and message passing to good effect. At its most primitive level, the concept is a simple one: each actor (or frame) is an independent module which maintains its own representation of whatever knowledge it deems to be important. The representation can only be accessed by sending the actor a message (which is itself an actor), and if the actor then sees fit, it can respond by sending back a message. Conceptually, message passing differs from procedure calling in that it implies no top-down structure, but in effect allows any kind of top-down, bottom-up, or other scheme to be undertaken (a fact exploited especially by MAYA).

This description comes fairly close to describing the |LISP message passing scheme with the following exceptions:

(i) messages are not objects, but are s-expressions or patterns;

(ii) answers to messages are not messages, but are arbitrary s-expressions or patterns;

(iii) answers are guaranteed if a message has been sent; that is, continuations are not part of the message.

(iv) activation records (i.e. execution instances) created by message passing have names and can be accessed.

Such considerations are extremely useful and non-restrictive if the purpose is to model knowledge rather than to produce an elegant description of process and control. Points (iii) and (iv) are perhaps the most critical differences between Hewitt's approach and that of |LISP. By guaranteeing an answer, |LISP removes the incumbency on the sending object to set up a continuation, a simplification achieved at the cost of some lack

of control flexibility. However, it seems useful that a super-goal always be required to evaluate the results of its subgoal, especially so in the case of a failure of some sort, since as Sussman (1973) has pointed out, profiting from errors is important.

As to difference (iv), the naming of activation records allows |LISP to maintain a record of processing decisions, to be able to restart interrupted processes, and to otherwise manipulate its process activations. This is useful in many places, especially in execution environment searches and episodic memory construction.

4.3 Other Approaches

In this section, several schemes that have not been as criterial to |LISP as have frames and PLANNER / CONNIVER will be compared on at least a few interesting dimensions with |LISP. MERLIN (Moore and Newell (1973)) turns out to be quite important both because of its semantic network data structure and because of its ability to reason by analogy. Semantic network concepts have already been discussed; analogy will be taken up in Chapter VI.

A final comparison will be made to the production system approach. Production systems (Newell (1973)) were designed as a psychological model of human short term memory processing and have more recently been used in a number of practical applications (e.g. MYCIN (Shortliffe (1976)). The aims of the approach don't really have all that much in common with those of |LISP, but similarities between the production system "match ---> action" kind of control paradigm and |LISP's pattern matching are fairly obvious. Specifically a pattern of the form (ANYTHING !(---)) seems to essentially encode the same information as a production might (|LISP's "productions", of course, can have n elements and more complex kinds of processing within them). If this analogy were carried out further, then a |PEXPR would be a group of productions and hence a production

system. Since a system using |LISP would have many such |PEXPRs, it would in effect contain many such "production systems". The only thing I can't fit in very well is short term memory, an aspect central to production systems but peripheral to my main concerns. However, it is encouraging that a control paradigm similar in some ways to that of |LISP has been so successful in various applications. It gives hope for the ultimate feasibility of |LISP for actual use in the "real world".

4.4 Contributions to the Issues

I would now like to examine the representation scheme in light of the issues which it was supposed to treat (see issues (i)-R through (vi)-R in section 1.3).

(i) handling procedural and declarative information:

Pattern expressions contribute to a unified view of declarative and procedural information. There is no distinction between a procedural |PEXPR body and a separate declarative data base. All information (procedural or declarative) is stored in patterns within the body of the |PEXPR and is accessed uniformly via pattern matching. Moreover, execution instances contain patterns that are the declarative residue of procedural messages, further blurring the distinction between procedural and declarative information.

(ii) accessing knowledge:

The scheme presented here has several interesting features involving the accessing of information. Information is only accessed when some object wants some other object to answer a message. In trying to respond to a message, a receiving |PEXPR has a couple of options: it can know the answer directly (i.e. it has a pattern in its body that contains the answer) or it may need to look elsewhere for an answer, in which case a strategy is followed that depends on the kind of information

being sought. Thus, a search along very specific "links" emanating from the |PEXPR can be instituted (allowing inheritance up "ISA" links for example), or an arbitrary inferencing scheme can be entered into involving much thrashing about, or the user can be asked to help out, or any number of other things can be attempted.

Even if the |PEXPR does know the answer, it may need to perform a number of computaticns before providing it. In so doing it can send out other messages using a variety of control paradigms ranging from the ordinary serial processing to simulated parallelism.

(iii) the necessity for a context mechanism:

The nature of context is rather more complex than has been suggested at several places during the chapter. It has to do with deciding what things are relevant at any time; the nature of this decision-making process will be made explicit in Chapter V. Here it suffices to say that the major contribution to context is the execution environment which extends up EX-ENVIRON pointers from any execution instance. The execution environment can be accessed whenever contextually varying information such as purpose, location of real world objects, current time, etc. is needed.

(iv) the need to keep a record of events:

Execution environments are also useful in providing an episodic memory, not only because they don't disappear after they are complete, but also because they can be examined in the same way as other |PEXPR environments. The first capability means that all execution instances which completed still are there to be queried as to who called them, why they were called, what they answered to the activation message patterns, what |PEXPR they are an execution instance of, and so on. The second capability is a by-product of having execution instances with meaningful patterns rather than uninterpretable internal pcinters. It means that all the matching routines which work

for ordinary |PEXPRs can also be used for execution instances and that searching old execution environments is as natural as searching anywhere else.

(v) robustness:

An interesting issue is that of robustness, i.e. how to cope with anomalous data. Although capabilities in this regard are not fully developed, |LISP is presently capable of failure to match processing if a pattern isn't directly matchable in a |PEXPR. In addition, since messages are guaranteed to get an answer, a sending |PEXPR can plan on getting a failure response of some sort if the receiving |PEXPR is unable to handle the message. With this response, the sender should be able to take appropriate action. However, even though the machinery has been provided to handle them, virtually no analysis has gone into the nature of failure responses.

(vi) combinatorial explosion v.s. complexity:

|PEXPRs can conveniently be viewed in several ways, a flexibility which helps to resolve both efficiency and complexity problems. For instance, when a |PEXPR is regarded as a node in a network, then the fact that only nearby objects can be "seen" means that search can be limited. In fact such search can often be restricted to objects in the ISA or execution environments. Of particular interest here are objects in the execution environment, the primary means of focussing attention.

But |PEXPRs can also be considered as separate modules accessible only via their messages, a fact which helps resolve complexity issues. Such separability is extremely valuable in localizing effects, and also in dividing up the domain into comprehensible chunks (moreover, |PEXPRs are flexible enough that they don't impose many restrictions on how to break down world knowledge).

Many of the other non-central issues raised in the introduction also can be addressed by this approach. For

example, the issue of non-goal-directed processing: while it is true that most of the examples of Chapter V involve top-down activation in that a |PEXPR sends a message to another |PEXPR in order to achieve a sub-goal, this is an external interpretation put on the message by humans. To |LISP it is just a message, indistinguishable (in how they are handled) from messages that achieve associative activation (see Chapter VI) or messages that allow a set of words to be conglomerated bottom-up into a single phrase or any other kinds of messages. Moreover, since execution instances remain after creation, it is possible for several pattern expressions to communicate with a single pattern expression, or vice versa. This can be essential for bottom-up or associative activation when it is often important that several messages from different sources be sent before the receiving |PEXPR can be considered relevant. Thus, it can be seen that |LISP defines a message passing paradigm that is neither bottom-up nor top-down, but either one can be simulated if desired.

In conclusion, the major contribution of |LISP is perhaps the confluence of ideas in one place. Though many of the ideas derive from other approaches, putting them all together is useful in that the ideas take on a new perspective when viewed in the context of one another.

The contribution that the representation scheme makes to conversation will become clearer as the next chapter (Chapter V) is read. In it, a particular conversation scenario is delineated, and a model to handle several conversations in this scenario is discussed in detail.

CHAPTER V

Modelling Conversation: A Detailed Example

In this chapter the representation scheme is applied to the analysis of conversation. To this end, an active participant carrying out a (simulated) plan of attending a symphony concert is modelled. During execution of the plan, the model, among other things, engages in several conversations, including one to buy a ticket to the concert, another to buy a drink at intermission, and one with a friend to "pass the time".

My primary concern in this chapter will be to indicate the basic nature of the interactions that must go on, and to show how the kind of analysis described here might eventually be extended to a more sophisticated model for conversation. Much pseudo-code is given, but note that none of it has been fully debugged. Moreover, many functions serve as "black boxes" in the sense that their I/O behaviour has been delineated but more elaborate versions are unavailable.

The chapter is organized along these lines: first, an environment in which to study conversation (called the concert scenario) is described. This is followed by a brief overview of how this scenario has been modelled in ¦LISP. The model itself is then presented in detail in terms of the goals it undertakes to handle a dialogue to buy a ticket. Higher level non-linguistic goals are described first, followed by scripts, speech acts, and the language level. Finally, the two other dialogues the model undertakes, one with a bartender and one with a friend, are discussed very briefly.

5.1 The Concert Scenario

I will discuss here a single small scenario in which a model undertakes conversations that might occur during attendance at a symphony concert. In these conversations the

model takes an active role (i.e. is one of the conversants), hence forcing it to have a goal oriented viewpoint on ensuing events.

Before giving more details about the scenario, I would like to indicate some of the general reasons for choosing it. The "concert scenario" has been chosen because it is complex enough tc illustrate most of the representation and language issues, yet it remains within finite dimensions. Various kinds of dialogue can occur including task-oriented (see Deutsch (1974)), non-task-oriented, formal, informal, etc. Non-linguistic goals occur and interact with the linguistic goals. Finally, having a single scenario allows a small amount of information to be used in several, possibly quite different, settings.

The scenario is essentially this: the model is at home and decides to go to a concert at the Queen Elizabeth theatre. The model

(i) leaves home;
(ii) goes to the Queen Elizabeth theatre;
(iii) buys a ticket to the concert;
(iv) enters the theatre;
(v) takes in the first half of the concert;
(vi) buys a drink at the bar at intermission;
(vii) drinks it;
(viii) unexpectedly meets a friend;
(ix) takes in the second half of the concert;
and (x) goes hcme.

Three conversations occur: during step (iii) when the model must talk with the ticket seller in order to purchase a ticket to the concert; during step (vi) when the model talks with the bartender in order to buy a drink; and during step (viii) when the model talks to the friend. These critical steps are treated in some detail; the other steps are only examined at a very cursory level, and are included mainly to show continuity in the plan and to indicate the interaction of linguistic and non-linguistic goals.

I will now present sample dialogues which might actually

occur in such a situation. They are fabrications, but my informal analysis of similar situations convinces me that they are not unrealistic. They have served as a guide to the kinds of things that the model must allow for, but many of the phenomena (linguistic and otherwise) that occur in them have not been fully accounted for in this version of the model.

Conversation 1

This conversation occurs when the model buys a ticket to the concert.

Ticket-seller: "Yes?"

Model: "I'd like a ticket to the concert."

Ticket-seller: "How about K-5? It is right centre about 10 rows back."

Model: "Fine. How much is that?"

Ticket-seller: "10 dollars."

Model: "O.K." (hands over the money)

Ticket-seller: (hands over the ticket)

Model: "Thanks."

Ticket-seller: "Thank you sir."

This somewhat innocuous conversation illustrates a number of interesting things. First, it is a task-oriented dialogue in the sense that it is entered into to achieve some rather specific goal (i.e. to get a ticket to the concert). Moreover, the task is a common one, well understood by many people, so there is a well defined script that can direct the conversation. The script clearly must be able to handle all the bargaining and exchange components underlying the conversation, to smoothly enter the conversation and just as smoothly terminate it, to discover the purpose for the conversation (when asked "Yes?"), and to interleave non-linguistic actions (handing over the money or tickets) with the linguistic utterances. Conversant models must be kept for both the ticket seller and the model so that the minimal surface utterances seen here can be devised (after

seeing what knowledge is stored there) and so that politeness markers ("sir") can be attached appropriate to the relative status of the conversants. As in all these conversations, conversation 1 must be properly integrated into the general plan of the model.

This conversation has been fairly completely studied in terms of what knowledge is needed to properly undertake it, how that knowledge can be represented, and how it interacts to yield a proper sequence of surface utterances. Later in the chapter it is discussed from the highest levels right down to how it handles surface level input / output. However, because the analysis is general enough to handle many conversations of this type, the actual surface utterances suggested by the analysis would differ somewhat from those shown here.

Conversation 2

This dialogue takes place at the bar in the lobby when the model decides to obtain a drink.

Bartender: "Sir?"
Model: "Could I have a rye?"
Bartender: "On the rocks?"
Model: "Please."
Bartender: "There you go." (produces drink)
"That'll be $1.50 please."
Model: (produces 2 dollar bill)
Bartender: "Thank you sir." (produces 50 cents change)
Model: "Thanks."

This conversation is almost a duplicate of the first one in its interesting features. The main reason for including it is to try to show that the analysis proposed for conversation 1 isn't totally "ticket specific". In fact, all the main structures proposed for conversation 1 can also be used for conversation 2, the only difference being that certain kinds of ticket information are replaced by similar kinds of drink

information (i.e. the information about the product being bought is now appropriate to drinks not tickets, the conversant is now a bartender not a ticket seller, the model's purpose is to buy a drink not a ticket, etc.) The conversation is discussed briefly in the same terms as conversation 1, and is shown to be handled in a similar manner. The subtle differences in surface language are not at all dealt with.

Conversation 3

This conversation is entered into by the model when it unexpectedly meets a "friend", Jack, at intermission.

Model: "Hi Jack."

Jack: "Model! What's new?"

Model: "Not much. I didn't know you came to these things. Do you have season tickets?"

Jack: "No - but I couldn't miss this one. I really want to hear the Mozart concerto."

Model: "I came mostly for the Bartok myself. And I can't say I was disappointed by the first half. The orchestra played superbly, don't you think?"

Jack: "Frankly, I was bored. Don't much like Bartok really, or any other 20th century composer."

Model: "Well, I feel the same way about Mozart. I'm usually bored to tears by his stuff. Still the B-flat isn't bad so maybe I'll stay awake for a change."

Jack: "There goes the buzzer. I'd best get back to my seat."

Model: "Good to see you again. We'll have to get together sometime for a beer."

Jack: "Sure thing. See you soon."

Model: "Take care."

This conversation illustrates several features not encountered in conversations 1 and 2. For one thing, it isn't task-oriented; that is, there is no explicit, detailed goal the model has for taking part in the conversation. The model has more general goals such as filling in time, being socially gregarious, trying to win support for its views or arguing against those of Jack. Because there is a much looser structure

to what the model is doing here, the general script overseeing the discussion is of necessity much less well informed in a direct sense. Instead, it must make many inferences based on partial information, must make good use of the belief models of itself and Jack, must be flexible and ready for unexpected information, and so on.

This conversation is interesting, as well, in that it isn't undertaken as part of any pre-determined plan, but is started as the result of unexpectedly meeting Jack. The plan in control at that time would have to explain the unrehearsed arrival of Jack and suggest ways of rectifying the anomaly of his presence. Such rectification would involve the activation of an appropriate script to direct the conversation. Moreover, the conversation isn't terminated as part of a plan either (i.e. the buzzer sounds) and this would have to be accounted for as well.

The conversation has a much more serious need for surface level linguistic analysis than do the other two. The utterances are longer, more complex, and more informal. The reference problems (especially pronoun reference) are more subtle and quite difficult (e.g. "this one" in the fourth utterance). Special linguistic forms ("the Bartok", "the B-flat") peculiar to classical music further complicate matters.

The sequencing order is much more poorly defined here, presenting problems even at the level of deciding when to interpret and when to produce utterances, whether to interrupt a long winded discourse, etc. Several well defined scripts (for greeting and closing, at least) could be integrated at the appropriate time if the script were alert to bottom-up cues, an integration which would save much processing time since utterances could be much more easily understood and produced by these scripts.

This conversation also illustrates the need for an episodic capability. Questions about the first half of the concert would require the model to scan over episodes representing what occurred at that time and extract what seemed relevant.

Most of the things conversation 3 illustrates have not been

analyzed in detail, although a preliminary attempt has been made to analyze some aspects.

5.2 Overview of the Model

Most important pieces of knowledge needed by the model for its task of buying a ticket to the concert are encoded as pattern expressions. To handle the concert scenario two main categories of |PEXPR are necessary:

(i) primary pattern expressions to carry out parts of the main plan of attending a concert including going to the concert, buying a ticket, taking part in a conversation during the ticket purchase, and so on.

(ii) secondary pattern expressions, such as those representing models of the conversant, the agenda of the concert, and the like, that are sources of information for the primary |PEXPRs but aren't really part of the mainstream plan. The distinction between (i) and (ii) can be viewed as the difference between active objects calling in other active objects to accomplish subgoals and static objects standing by to provide certain pieces of "foregrounded" knowledge when asked to do so by the active objects. The division is not, of course, absolute in that primary |PEXPRs can ask one another for information without really giving up control; and secondary |PEXPRs can be activated as "temporary" subgoals when they are asked questions by primary |PEXPRs or each other.

Some of the primary pattern expressions used by the model to handle a portion of the conversation to buy a ticket are shown in Figure 5.1 which outlines in graphic form the important object / sub-object goal dependencies.[1]

These |PEXPRs are defined as follows:

(i) TOP-VIEW: overlord to the model; invokes parallel

---

[1] In Figure 5.1 the single lines represent subgoal links; the double line just beneath TOP-LEVEL indicates that META-VIEW and WORLD-VIEW are spawned as parallel subgoals of TOP-LEVEL; and the double arrow between INQUIRE and YES2 indicate that YES2 replaces INQUIRE as a subgoal of WHAT-DO-YOU-WANT.

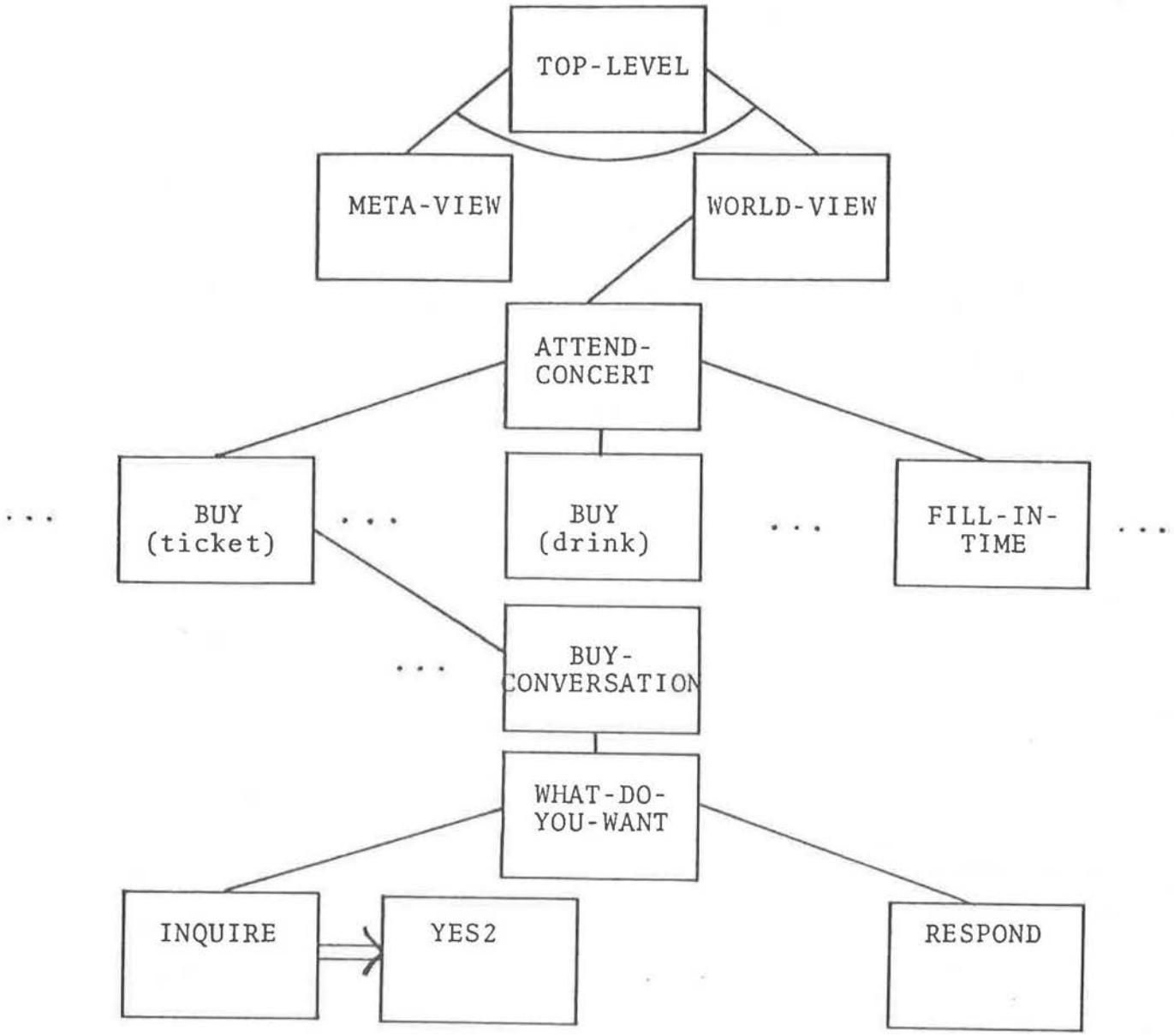


Figure 5.1 -

The Dynamic Linkages of Some Concert Scenario IPEXPRs

subgoals META-VIEW and WORLD-VIEW.

(ii) META-VIEW: the model's "intelligent" garbage collector; responsible for overseeing all clean-up operations on the system's structures.

(iii) WORLD-VIEW: essentially the "consciousness" of the model. Must co-ordinate, at the highest levels, the goals of the model; must decide what subgoals to achieve when and for what reasons.

(iv) ATTEND-CONCERT: the |PEXPR that directs the model's efforts to attend this particular concert. Conceptually, at least, it has been built by some plan-construction objects at the request of WORLD-VIEW when it decided that attendance at the concert would be a good idea. It must achieve the many subgoals necessary to accomplish this goal. The major three of importance for conversation are buying a ticket, buying a drink, and filling in time at intermission.

(v) BUY: the model's pattern expression to direct the buying of something. In this case ATTEND-CONCERT suggests buying a ticket to the concert and later calls BUY again to suggest buying a drink. BUY must direct the model to the place of purchase, must recognize the particular seller of the item, must access |PEXPRs which represent various bargaining positions of the two parties to the buying, and finally must engage in a conversation to effect the purchase.

(vi) BUY-CONVERSATION: is the script that controls a conversation to bargain for the exchange of one set of goods for ancther; in this case the exchange is between the money of the model and the ticket of the ticket seller. The pattern expression predicts the conversational sequence of events starting with the greetings which open the conversation, through bargaining over the location of the ticket, bargaining over the price, and finally ending with the exchange of the agreed upon goods and the making of accompanying comments. Each of these predictions takes the form of calling in a sub-script to direct the processing. The first is rather poetically named WHAT-DO-YOU-WANT.

(vii) WHAT-DO-YOU-WANT: a script that knows about the kind of language that accompanies a conversational exchange where one person asks another what he wants (i.e. his purpose). The script is invoked in this case by the EXCHANGE-CONVERSATION |PEXPR to handle the first couple of utterances in the conversation to buy a ticket. It expects the ticket seller to inquire into the purposes of the model; and expects the model to respond appropriately to this inquiry.

(viii) INQUIRE: a speech act which will either understand or produce an "inquire" utterance, depending on whether the model is listening or speaking. In the current example, the WHAT-DO-YOU-WANT pattern expression expects the ticket seller to utter an inquiry into the purpose of the model, so inquire is activated to understand such an utterance. Achieving this interpretation requires the |PEXPR to look into the input buffer for words which have actually been uttered. Discovering the word "yes", it checks to see if there is a speech act associated with the word "yes?" which could be construed as an inquiry. Finding that there is (YES2), INQUIRE is supplanted by YES2 which continues the processing (since the actual input should take precedence over any expectation).

(ix) YES2: represents the meaning of "yes" that corresponds to an inquiry (rather than the meaning "affirmative answer"). It is called in to continue the process of understanding the input. Since "yes?" is a surface string corresponding to the meaning of YES2, YES2 is able to achieve the proper interpretation that the utterance is an inquiry into the current purpose of the model (available in the execution environment). It then returns, satisfied, to its calling |PEXPR.

WHAT-DO-YOU-WANT regains control; sees that the first utterance is just about as expected (if it weren't, WHAT-DO-YOU-WANT would have had to explain what went wrong); integrates it into the conversation to date, being recorded in the script; and then proceeds to the second script utterance, the generation of a response to the inquiry. Before handling the next script utterance, however, WHAT-DO-YOU-WANT must first

decide if something in the conversation to date is demanding priority over its script expectations, and if so, what to do about it. In this case an analysis of the conversation to date suggests only that the unknown purpose of the first utterance should be discovered and, since this is in accord with the script's expectations, everything is in order.

(x) RESPOND: contains the model's ideas about responding to a query, including how to interpret or generate a response. Since WHAT-DO-YOU-WANT orders RESPOND to generate an utterance stating the purpose of the model, it does so, sending the resultant set of words (e.g. "I'd like a ticket to the concert.") to the EXPR SPEAK-WORDS. Clearly many other objects have been called on here to decide such things as how much semantic information will express the model's purpose to the ticket seller (this involves, at least, looking at the beliefs of the conversant, of the model itself, and into the execution environment); how to phrase the eventual output; and so on. These issues aren't treated to any great extent: I'm content at this stage that the system knows generally what to say.

Once RESPOND is done, it returns again to WHAT-DO-YOU-WANT which must check that the utterance produced is appropriate, tie it in to the conversation to date, and then proceed to the next script utterance. But, since the script is now complete, WHAT-DO-YOU-WANT returns to the EXCHANGE-CONVERSATION script which, if satisfied with its behaviour, ties the WHAT-DO-YOU-WANT utterances into its own conception of the conversation to date. It then proceeds to the bargaining scripts which carry on. Eventually, even the EXCHANGE-CONVERSATION pattern expression is done, so it returns to BUY, which, when finished, goes back to ATTEND-CONCERT to continue with the plan of attending the concert. Among other things, two more conversations will be undertaken before the ATTEND-CONCERT |PEXPR is satisfied, and these will be handled in much the same way as the ticket buying conversation has been.

Once ATTEND-CONCERT is done, WORLD-VIEW takes over to ponder what to do next. Note that working in parallel to

WORLD-VIEW has been META-VIEW cleaning up the large number of execution instances and other miscellany constructed during the reign of WORLD-VIEW.

This has been a brief look at some of the many action-packed primary pattern expressions in the concert scenario. But, these |PEXPRs need to access data contained in other, secondary pattern expressions. Secondary |PEXPRs are activated mainly for the information contained in their static patterns and are thus distinguished from primary |PEXPRs which are activated to achieve some major subgoal.

For any particular set of primary goals, a certain collection of secondary |PEXPRs is needed, constituting in a sense, the foreground information for the primary goals. The relevance of each is discovered at various points during the execution of the primary planning sequence, and pointers from these primary |PEXPRs are set up to record the relevance of the various secondaries to the main stream context. Later other |PEXPRs needing information can use these links to determine which secondaries to construct.

So, lets look at some of the secondary pattern expressions that have proven useful in the concert scenario. The first such |PEXPR is the CONCERT1 |PEXPR, containing all the model's knowledge about this particular concert such as the agenda, the location of the concert, the entrance requirements. Theoretically at least, it has been built sometime in the past (whenever the model discovered that the concert was to be performed and heard the details of the agenda), and has been re-discovered by some planning |PEXPR spawned by WORLD-VIEW to construct the plan to go to this concert. It is passed by WORLD-VIEW as part of the message that activates ATTEND-CONCERT and is used extensively in further sub-|PEXPRs (especially in retrieving characteristics of the ticket to the concert and in discovering the agenda).

Another secondary pattern expression, the TICKET-FOR-CONCERT1 |PEXPR (a new instance of the generic TICKET-FOR-CONCERT) is generated by ATTEND-CONCERT when it is

about to BUY the ticket. It inherits from TICKET-FOR-CONCERT knowledge as to the physical location of the ticket (in the Queen Elizabeth theatre's ticket booth), the desired location represented by the ticket, and the projected cost for such a location. Looking further up the ISA environment to TICKET, it can discover that a TICKET-SELLER sells tickets. Later, once actual characteristics (location, cost, etc.) of the ticket have been determined (by the BARGAIN sub-goal of BUY-CONVERSATION), they are added to TICKET-FOR-CONCERT1.

Perhaps the most important secondary pattern expression is TICKET-SELLER1, first created as a new instance of the generic TICKET-SELLER by the BUY subgoal when it recognizes the existence of a particular person in the ticket booth. BUY knows that the person is a TICKET-SELLER because TICKET-FOR-CONCERT1 tells it so. TICKET-SELLER1 constitutes the model's model of the conversant. It initially contains only the knowledge that it represents a TICKET-SELLER, but via its INSTANCE-OF link it is able to inherit much information from the ISA environment. In particular the immediately superior TICKET-SELLER contains information about ticket sellers: that they sell tickets, that they're willing to exchange tickets for an appropriate amount of money, that they are sellers, and so on. This information is of use later when various bargaining positions must be discovered. Moreover, a truly complete TICKET-SELLER would contain information about speaking habits, probable locations, potential scripts, etc. As time goes on, additional characteristics of TICKET-SELLER1 can be added as the model constructs an ever more accurate view of the conversant. A final note: the model also has a model of itself, the details of which are explained in section 5.3.3.

The preceding discussion gives a quick overview of the various capabilities of the model in the concert domain. I would now like to present a much more detailed analysis. The interactions among the various objects gets quite complex, so rather than following each message as it is dispatched, in general I plan to take the approach of describing each pattern

expression fairly completely before going on to other objects. The order of description is basically breadth-first and is shown in Figure 5.2.

Non-linguistic Goals:	TOP-VIEW META-VIEW WORLD-VIEW ATTEND-CONCERT BUY
Scripts:	BUY-CONVERSATION WHAT-DO-YOU-WANT BARGAIN EXCHANGE FAREWELL
Speech Acts:	INQUIRE YES2
Language Level Goals:	UTTERANCE CLAUSE NOUN-GROUP VERB-GROUP PREP-GROUP

Figure 5.2 - Order of Presentation

5.3 Non-linguistic Goals

5.3.1 The Highest Goals

At the highest levels the model is not concerned with language at all, but is instead interested in co-ordinating its various non-linguistic goals. These goals have been drastically simplified in this presentation in order to illustrate how they relate to one another and to language without going into the obviously large complexities of representing sophisticated non-linguistic goals.

The top-most goal in the system is TOP-VIEW whose main purpose is to co-ordinate the garbage collector and the other actions of the model. It is called in at system initiation by sending it (EXECUTE TOP-VIEW ?MATCH-PROGRAM), a message which will match TOP-VIEW's (EXECUTE TOP-VIEW !(---)) pattern. The attempt to match the third element will institute the computation which controls the rest of the system's actions.

```
<|PDEF TOP-VIEW
   (SUPERSET TOP-VIEW SYSTEM-OBJECT)
   (EXECUTE TOP-VIEW
      !(|PARALLEL
         (META-VIEW (EXECUTE META-VIEW ?META-RESULT)
                    (|TIME= 10))
         (WORLD-VIEW (EXECUTE WORLD-VIEW ?WORLD-RESULT)
                     (|TIME= 100))))    >
```

The receiving pattern's third element is a call to |PARALLEL that sends EXECUTE messages in pseudo-parallel to META-VIEW (the name of the garbage collector) and WORLD-VIEW (the name of the top-level READ-|EVAL-PRINT loop of the system). The time-slicing has been arbitrarily set at 10 to 1 in favour of the WORLD-VIEW pattern expression; that is the system will spend 90% of its time in action, the other 10% in garbage cclllection.

```
<|PDEF META-VIEW
    (SUPERSET META-VIEW SYSTEM-OBJECT)
    (EXECUTE META-VIEW $(RECLAIM))    >
```

Currently, the nature of META-VIEW has not been analyzed to any great detail; all it does to answer an EXECUTE message is to invoke LISP's garbage collector. Facets of a more sophisticated garbage collector are discussed in Chapter VI.

The other parallel subgoal of TOP-VIEW is WORLD-VIEW, potentially the most crucial pattern expression in the model since it is the top-level co-ordinator of the model's goals. But, as with META-VIEW, it has for the moment been drastically cut back and consists mainly of a READ-|EVAL-PRINT loop which gets entered upon receipt of an appropriate EXECUTE message (usually from TOP-VIEW).

```
<|PDEF WORLD-VIEW
    (SUPERSET WORLD-VIEW SYSTEM-OBJECT)
    (EXECUTE WORLD-VIEW
      !(|PROG ()
         L1 (PRINT (|EVAL (READ)))
            (|GO 'L1)))    >
```

Since the current WORLD-VIEW |PEXPR merely READs a form, |EVALs it and PRINTs the result of the form's |EVALuation, it is up to the user to specify the goals of the system. First, the pattern (LOCATION SELF HOME) is asserted, indicating that the model (SELF) is at "home". Then, the message form

```
(ATTEND-CONCERT
   (EXECUTE ATTEND-CONCERT SELF CONCERT1 ?ATTEND-RESULT))
```

is entered to call in the ATTEND-CONCERT subgoal.

5.3.2 A Major Subgoal: ATTEND-CONCERT

The user asks the model (SELF) to employ a plan of action called ATTEND-CONCERT in order to attend a concert (CONCERT1). The ATTEND-CONCERT |PEXPR (given completely in Appendix II) has the form

```
<|PDEF ATTEND-CONCERT
     (SUPERSET ATTEND-CONCERT ATTEND)
     (EXECUTE ATTEND-CONCERT SELF ?THIS-CONCERT
                    !(EVENT-SEQUENCE --- )) >
```

The actual steps the model is to undertake are contained in the !(EVENT-SEQUENCE ---) pattern element which will be |EVALed when the message pattern

(EXECUTE ATTEND-CONCERT SELF CONCERT1 ?ATTEND-RESULT)

is matched against the EXECUTE pattern of ATTEND-CONCERT.

Looking at the process in more detail, the first three elements of the message pattern trivially match the corresponding elements of the receiving pattern. Note that the third element, SELF, names a |PEXPR that contains the model's knowledge of itself (it is fully presented in section 5.3.3). The fourth message element, CONCERT1, matches the fourth receiving element, ?THIS-CONCERT, with the side-effect that the variable THIS-CONCERT is bound to CONCERT1 in the context of the new execution instance (called ATTEND-CONCERT-1) of the receiving |PEXPR ATTEND-CONCERT.

Diverting attention for a moment to the CONCERT1 |PEXPR

```
<|PDEF CONCERT1
     (INSTANCE-OF CONCERT1 CONCERT)
     (LOCATION CONCERT1 QET)
     (AGENDA CONCERT1 AGENDA-CONCERT1)     >
```

it can be seen that CONCERT1 is a particular CONCERT, located at the Queen Elizabeth theatre with the agenda AGENDA-CONCERT1. Of course, CONCERT is also a |PEXPR:

```
<|PDEF CONCERT
     (SUPERSET CONCERT EVENT)
     (ENTRANCE-REQUIREMENT ↑CONCERT
                ↑TICKET-FOR-CONCERT)     >
```

containing the information that the entrance requirements for a particular concert is a particular ticket to the concert (note

the "↓" macros indicating instances). Similarly, the QET |PEXPR:

```
<|PDEF QET
      (INSTANCE-OF QET THEATRE)
      (TICKET-BOOTH QET TICKET-BOOTH-QET)
      (LOBBY QET LOBBY-QET)
      (BAR QET BAR-QET)
      (AUDITORIUM QET AUDITORIUM-QET)
      (SEATS QET SEATS-QET)    >
```

and AGENDA-CONCERT1:

```
<|PDEF AGENDA-CONCERT1
      (INSTANCE-OF AGENDA-CONCERT1 AGENDA-CONCERT)
      (ORCHESTRA CONCERT1 VANCOUVER SYMPHONY)
      (CONDUCTOR ORCHESTRA CONCERT1 AKIYAMA)
      (FIRST-HALF CONCERT1 BARTCK-CONCERTO-FOR-ORCHESTRA)
      (SECOND-HALF CONCERT1 MOZART-PIANO-CONCERTO-27)
      (SOLOIST SECOND-HALF CONCERT1 BRENDL)    >
```

The patterns in these |PEXPRs, of course, contain the names of other |PEXPRs and these |PEXPRs contain patterns with the names of still other |PEXPRs, and so on. This connectivity can be better appreciated by representing the "links" among the previously mentioned |PEXPRs as shown in Figure 5.3.[1]

Returning to the message processing, it can be seen that SELF is going to attend CONCERT1 if the fifth element of the message, ?ATTEND-RESULT, matches the fifth element of the receiving pattern, !(EVENT-SEQUENCE ---) in the EXECUTE pattern of ATTEND-CONCERT (above). It, of course, does match, but only if the |EVALuation of the EVENT-SEQUENCE |EXPR returns non-NIL. For a fuller explanation of the action of EVENT-SEQUENCE, see Appendix I. Suffice to say here that it basically |EVALs each form in its body sequentially (much as would a |PROG), with the side-effect of asserting (THEN Xi Xi+1) patterns in all execution instances Xi that correspond to major steps in the event sequence (designated by a label in the event sequence body).

The first such major step encountered in the EXECUTE of ATTEND-CONCERT is

---

[1]The following notational conventions have been used: the nodes are the |PEXPRs; the arcs represent the patterns contained in the |PEXPRs; the labels on the arcs are the pattern heads; "e" means INSTANCE-OF; "s" means SUPERSET; "r-i" indicates RCLE-INSTANCE-OF; "↓" at the end of an arc means that the linkage is to an (arbitrary) instance of the node at that end of the arc; "!" means that the value of the node at that end of the arc must be computed.

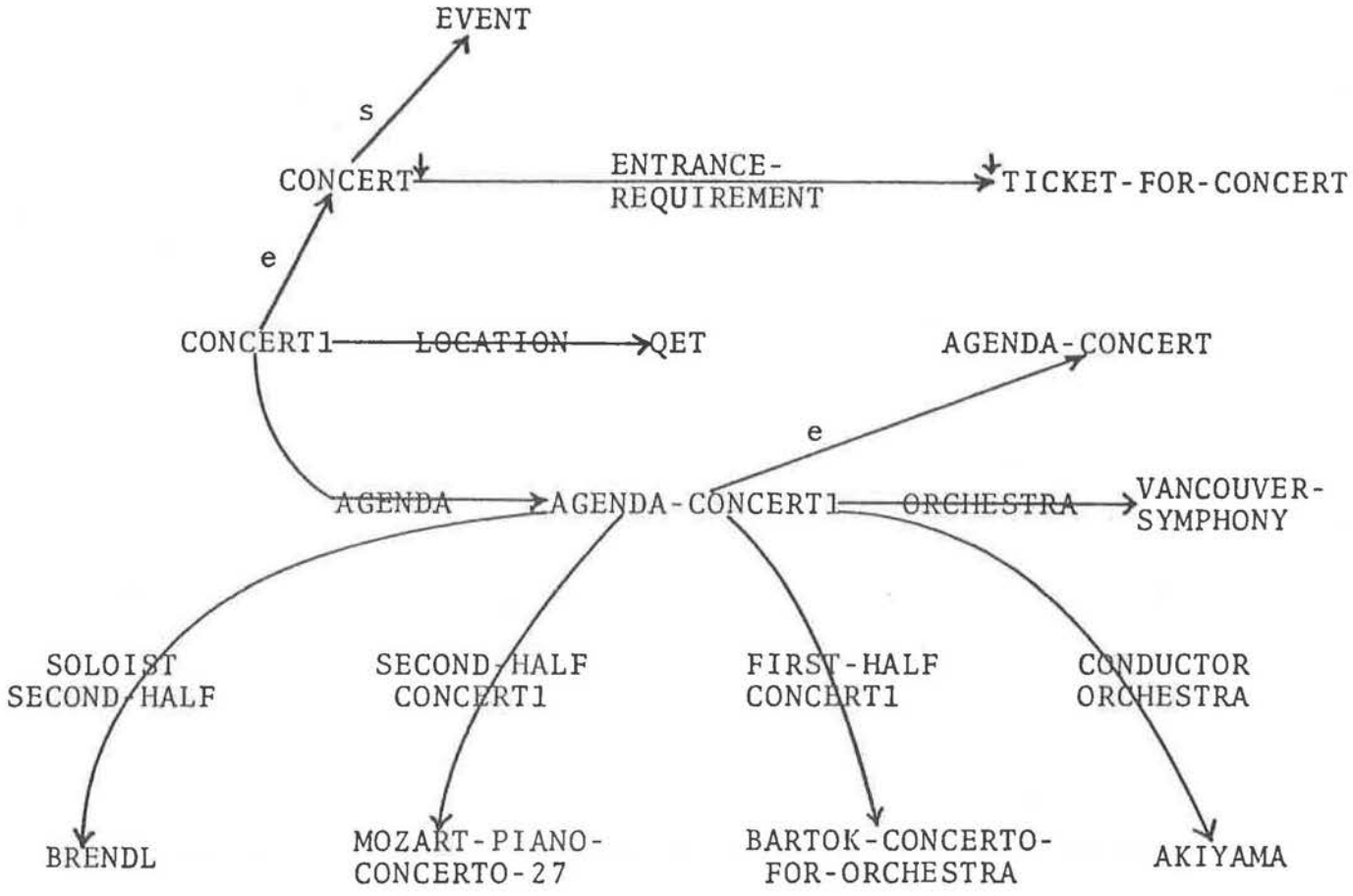


Figure 5.3 -
Network for Concert Information

STEP1
(GOTO (EXECUTE GOTO SELF !LOC-CONCERT ?GO-PLACE))

In this step the model leaves START-LOCATION (HOME, discovered by looking into the execution environment for the current location of self, i.e. %(LOCATION SELF ?START-LOCATION)) to go to the location of the concert (QET, pointed to from CONCERT1). GOTO is a !PEXPR which would be concerned with actually achieving the goal of going from HOME to QET, including managing all other cues which might be needed to ensure arrival at QET. GOTO would also be responsible for changing the current execution environment LOCATION pointer en-route so that it continued to accurately reflect changes in location of the model. Even this part requires some work; that is, there might be representations of location in the execution environment such as (LOCATION SELF VANCOUVER) which needn't be changed, but (LOCATION SELF HOME) certainly needs to be. Generally speaking, LOCATIONs in the execution environment can be changed according to the following rule: find the lowest PART-OF intersection between the start location (e.g. HOME) and the finish location (e.g. QET), in this case VANCOUVER. Then change all execution environment LOCATION patterns which designate locations below this intersecting node (e.g. change (LOCATION SELF HOME) and (LOCATION SELF POINT-GREY) to (LOCATION SELF QET); but don't change (LOCATION SELF VANCOUVER) or (LOCATION SELF CANADA)). They exist in the execution environment). This is quite reasonable since the PART-OF intersection defines precisely the level of detail being changed by the movement.

Similar kinds of automatic execution environment pattern changes must be carried out at many places in the model, such as when a timer is advanced (recall the timing of interrupts from Chapter III), when the model changes position in any way, and so on. The code to undertake such changes would be embedded in IF-ADDED or IF-REMOVED methods associated with objects such as LOCATION or TIME, and would be invoked when patterns headed by these objects are asserted or deleted by the model's actions.

Once GOTO has finished executing, the model has either

succeeded or failed in its attempted traversal of some route (of GOTO's own choosing) between HOME and QET. If it has failed, NIL will be returned to ATTEND-CONCERT-1. This eventuality is allowed for in the ATTEND-CONCERT-1 event sequence by calling in the |EXPR CHECK-FAILURE which will check for the failure and suggest a course of remedial action if a failure has indeed occurred. CHECK-FAILURE is currently quite simple, but it will eventually be upgraded.

Assume that the first step of the event sequence is successful; that is, the model is at the QET. Prior to buying a ticket to the concert, the model must find out the ticket's characteristics, so it asks CONCERT1 (the value of THIS-CONCERT)

(ENTRANCE-REQUIREMENT !THIS-CONCERT ?DESIRED-TICKETS).

Since CONCERT1 doesn't know this information directly, it must be inherited from the ISA environment where, stored with CONCERT, is a pattern

(ENTRANCE-REQUIREMENT ↑CONCERT ↑TICKET-FOR-CONCERT)

The attempt to match ?DESIRED-TICKET with ↑TICKET-FOR-CONCERT results in the creation of a new instance of TICKET-FOR-CONCERT (see the macro conflict table in Chapter III) called TICKET-FOR-CONCERT1, which is bound to DESIRED-TICKET. TICKET-FOR-CONCERT1 will, once the ticket is purchased, contain the characteristics of the particular ticket bought for this concert. Right now TICKET-FOR-CONCERT1 is a blank slate, containing only an

(INSTANCE-OF TICKET-FOR-CONCERT1 TICKET-FOR-CONCERT) pattern.

It can, of course, inherit from TICKET-FOR-CONCERT information regarding ticket costs and information about the QET location represented by the ticket (dubbed REPN here to distinguish it from the LOCATION of the ticket, i.e. the ticket booth of the QET).

```
<|PDEF TICKET-FOR-CONCERT
   (SUPERSET TICKET-FOR-CONCERT TICKET)
   (LOCATION ↓TICKET-FOR-CONCERT TICKET-BOOTH-QET)
   (REPN ↓TICKET-FOR-CONCERT =(X (SUBPART X SEATS-QET)))
   (COST ↓TICKET-FOR-CONCERT
     !(|COND ((EQ 'SEATSQET
                  (|POINTER REPN TICKET-FOR-CONCERT))
                '↓DOLLARS-10)
             ((EQ 'SEATSQETLEFTCENTRE
                  (|POINTER REPN TICKET-FOR-CONCERT))
                '↓DOLLARS-5)
             ((EQ 'SEATSQETRIGHTCENTRE
                  (|POINTER REPN TICKET-FOR-CONCERT))
                '↓DOLLARS-5)
             (T NIL)))     >
```

At higher levels in the ISA environment there is the TICKET |PEXPR:

```
<|PDEF TICKET
   (SUPERSET TICKET ENTRANCE-REQUIREMENT)
   (SELLER ↓TICKET ↓TICKET-SELLER)     >
```

This |PEXPR contains more information about tickets. All this ticket information can be represented in a network such as that shown in Figure 5.4.

Having determined its ticket needs, the model proceeds to the next major step of ATTEND-CONCERT: the purchasing of the ticket:

```
STEP2
 (BUY (EXECUTE BUY SELF !DESIRED-TICKET ?BUY-RESULT))
```

Here, BUY is asked to execute the purchase of TICKET-FOR-CONCERT1.

5.3.3 The BUY Subgoal

The BUY |PEXPR is given in full in Appendix II and has general form:

```
<|PDEF BUY
   (SUPERSET BUY ACTUAL-TRANSACTION)
   (EXECUTE BUY ?BUYER ?ITEM !(EVENT-SEQUENCE --- ))  >
```

Upon receipt of the EXECUTE message from ATTEND-CONCERT, ?BUYER matches SELF (and is bound to it in BUY-1, the newly created execution instance of BUY), while ?ITEM, the thing to be purchased, matches TICKET-FOR-CONCERT1 (and is also bound). Finally, the fifth element of the receiving pattern, !(EVENT-SEQUENCE ---) matches ?BUY-RESULT once it has computed to non-null.

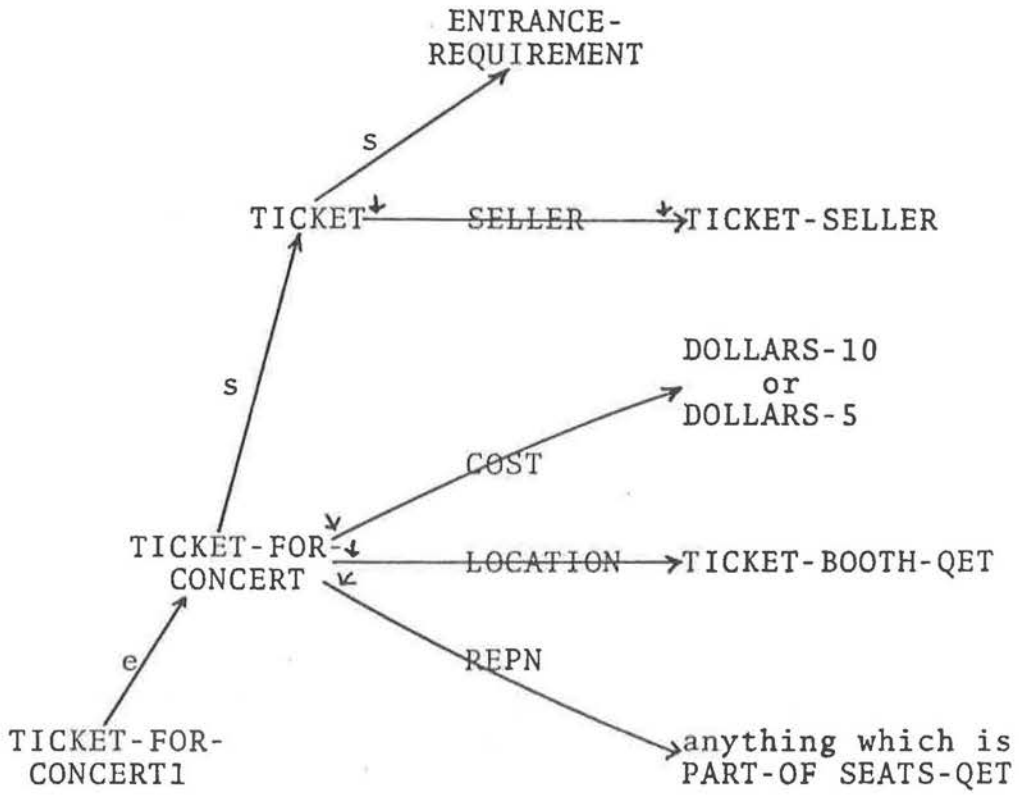


Figure 5.4 - Ticket Information

Before looking at this computation in detail, it is imperative to discuss SELF, the model's model of itself.

```
<|PDEF SELF
   (INSTANCE-OF SELF PERSON)
   (ROLE-INSTANCE-OF SELF
     !(|PROG (OBJECT TRANSACTION)
        %(PURPOSE SELF (?TRANSACTION SELF ?OBJECT))
         (|COND ((EQ TRANSACTION 'BUY)
                 (|COND ((SUBINSTANCE OBJECT
                           'TICKET-FOR-CONCERT)
                         (|RETURN 'SELF-AS-A-BUYER-
                           OF-CONCERT-TICKET))
                        ((SUBINSTANCE OBJECT
                           'ALCOHOLIC-DRINK)
                         (|RETURN 'SELF-AS-A-BUYER-
                           OF-ALCHOLIC-DRINK))
                        (T (|RETURN 'SELF-AS-A-BUYER))))
                (T (|RETURN NIL))))))    >
```

It is not feasible to store all the information the model knows about itself in one |PEXPR. SELF, therefore, acts mainly as a "central switchboard" through which requests for knowledge of SELF are filtered on their way to |PEXPRs representing views of SELF which are more appropriate in the circumstances of the request.

One particular view which the model can have of itself is in some "role", for example seller, buyer, lover, worker. It seems clear that a person's behaviour can alter drastically depending on what role he is playing. For example, the interactions between a supervisor and his student are different from those when the same two people are interacting as friends, different still when they are housemates sharing the household chores. Many characteristics, such as sound of voice, size, height, weight, sex, are invariant over these roles; but many more, such as attitude, status, kind of vocabulary used, wants and desires, are variable.

In the SELF pattern expression the difference between a rcle and the normal viewpoint of SELF is indicated by the ROLE-INSTANCE-OF pointer (in contrast to the straight INSTANCE-OF pointer). In all circumstances the model is a PERSON, but depending on the situation, the model can also take on various roles. The role variability is indicated by the "!" computational element in the pattern which will return a different |PEXPR (depending on context) for each role of the

model.

ROLE-INSTANCE-OF pointers are accessed when patterns like (STATUS ---), (ATTITUDE ---), (WANT ---), fail to match in a !PEXPR like SELF. Since such patterns represent qualities which are role dependent, a search for a match would be directed up RCLE-INSTANCE-OF links before anything else is done.

Role-instances (as !PEXPRs from which ROLE-INSTANCE-OF pointers are followed can often be designated) are interesting in that they give a way to view an object in many different lights. There is really no restriction on what constitutes a rcle instance. For example, an instance can be a role instance of another role instance, or of an instance, or of a class, or of an animate object, or of an inanimate object, etc.

Returning to the model's attempt to execute BUY, the attempt to match the ?BUY-RESULT message element results in the !EVALuation of the (EVENT-SEQUENCE ---) element of the receiving pattern. Before undertaking any steps, the model checks that BUYER is SELF and, finds the physical LOCATION of the ticket (TICKET-BOOTH-QET, inherited from TICKET-FOR-CONCERT). The model then executes STEP1:

```
STEP1
  (GOTO (EXECUTE GOTO SELF !PLACE-ITEM ?GOT-THERE))
```

that is, it goes to TICKET-BOOTH-QET. Having successfully completed this step, the model expects to find the seller of the item at this place: it thus asks TICKET-FOR-CONCERT1 for the seller of the item, the information this time being inherited from TICKET. It is interesting that matching (SELLER TICKET-FOR-CONCERT1 ?SELLER) against (SELLER ↑TICKET ↑TICKET-SELLER) succeeds with a new instance of TICKET-SELLER being created when ?SELLER is compared to ↑TICKET-SELLER (see macro-conflict table, Chapter III).

```
<|PDEF TICKET-SELLER1
   (INSTANCE-OF TICKET-SELLER1 TICKET-SELLER)    >

<|PDEF TICKET-SELLER
   (SUPERSET TICKET-SELLER SELLER)
   (SELL ↑TICKET-SELLER ↑TICKET)
   (WANT ↑TICKET-SELLER
     (EXCHANGE
        TICKET-SELLER-HAS-BARGAINING-POSN
        TICKET-SELLER-WANTS-BARGAINING-POSN))    >
```

This new instance, called TICKET-SELLER1, represents the new seller of tickets the model expects to find in the ticket booth and will have information added to it when details of the particular ticket seller are discovered during the ensuing transaction to buy the ticket.

Next, certain important patterns are asserted in the BUY-1 execution instance so that the information can be used by subgoals of BUY-1 to determine the identities of the seller and buyer, the purpose of the seller, the purpose of the buyer, and the focus of attention of the buyer.

Finally, the crucial part of executing BUY is encountered: STEP2, where the model engages in a conversation with the ticket seller in order to obtain the ticket. To this end, the script BUY-CONVERSATION is asked to EXECUTE itself:

```
STEP2
 (BUY-CONVERSATION
   (EXECUTE BUY-CONVERSATION !BUYER !SELLER
            !ITEM ?CONV-RESULT))
```

5.4 Scripts

The particular pattern expressions to be discussed in this section are called scripts (see Abelson (1973)). Scripts contain expectations regarding both sides of a conversation. They direct both the interpretation and production of utterances, structure the sequencing of the utterances, tie together the various utterances into some coherent whole, extract relevant information from the utterances, and if necessary act upon this information.

5.4.1 BUY-CONVERSATION

The script of most concern here is BUY-CONVERSATION (the complete version of which is given in Appendix II). Its form is

```
<|PDEF BUY-CONVERSATION
   (SUPERSET BUY-CONVERSATION
             SOCIAL-TRANSACTION-CONVERSATION)
   (EXECUTE BUY-CONVERSATION ?BUYER ?SELLER ?ITEM
             !(EVENT-SEQUENCE --- ))   >
```

BUY-CONVERSATION is called in EXECUTE mode by BUY. The message pattern is (after |EVALuation of "!" macros in the context of BUY-1)

```
(EXECUTE BUY-CONVERSATION SELF TICKET-SELLER1
          TICKET-FOR-CONCERT1 ?CONV-RESULT)
```

and it clearly matches the BUY-CONVERSATION pattern

```
(EXECUTE BUY-CONVERSATION ?BUYER ?SELLER
          ?ITEM !(EVENT-SEQUENCE --- ))
```

with appropriate bindings for BUYER, SELLER, and ITEM in the context of the newly created execution instance BUY-CONVERSATION-1. The matching is subject to the constraint that !(EVENT-SEQUENCE ---) |EVALs to non-null.

The event sequence, as is usual for EXECUTE patterns, contains the plan of action to be executed. It consists of the subgoals labelled STEP1, STEP2, ... , STEP5, denoting calls to five sub-scripts: WHAT-DO-YOU-WANT, BARGAIN, BARGAIN (again), EXCHANGE, and FAREWELL. The goal tree would thus look something like Figure 5.5 (where "ex" indicates an EX-ENVIRON pointer and THEN links have been added by EVENT-SEQUENCE).

STEP1 of BUY-CONVERSATION suggests that a sub-script called WHAT-DO-YOU-WANT is to be expected as the first conversational foray. This script will be discussed shortly; it merely represents a particular kind of "Hello" - "How are you?" utterance exchange which seems necessary in order to establish (or in this case confirm) conversational roles at the beginning of a conversation (see, for example, Schegloff (1971) for a discussion of this phenomenon).

If this step successfully concludes, then the sub-script (or utterance) must be "tied-in" to some sort of representation of the conversation to date. This promises to be fairly crucial

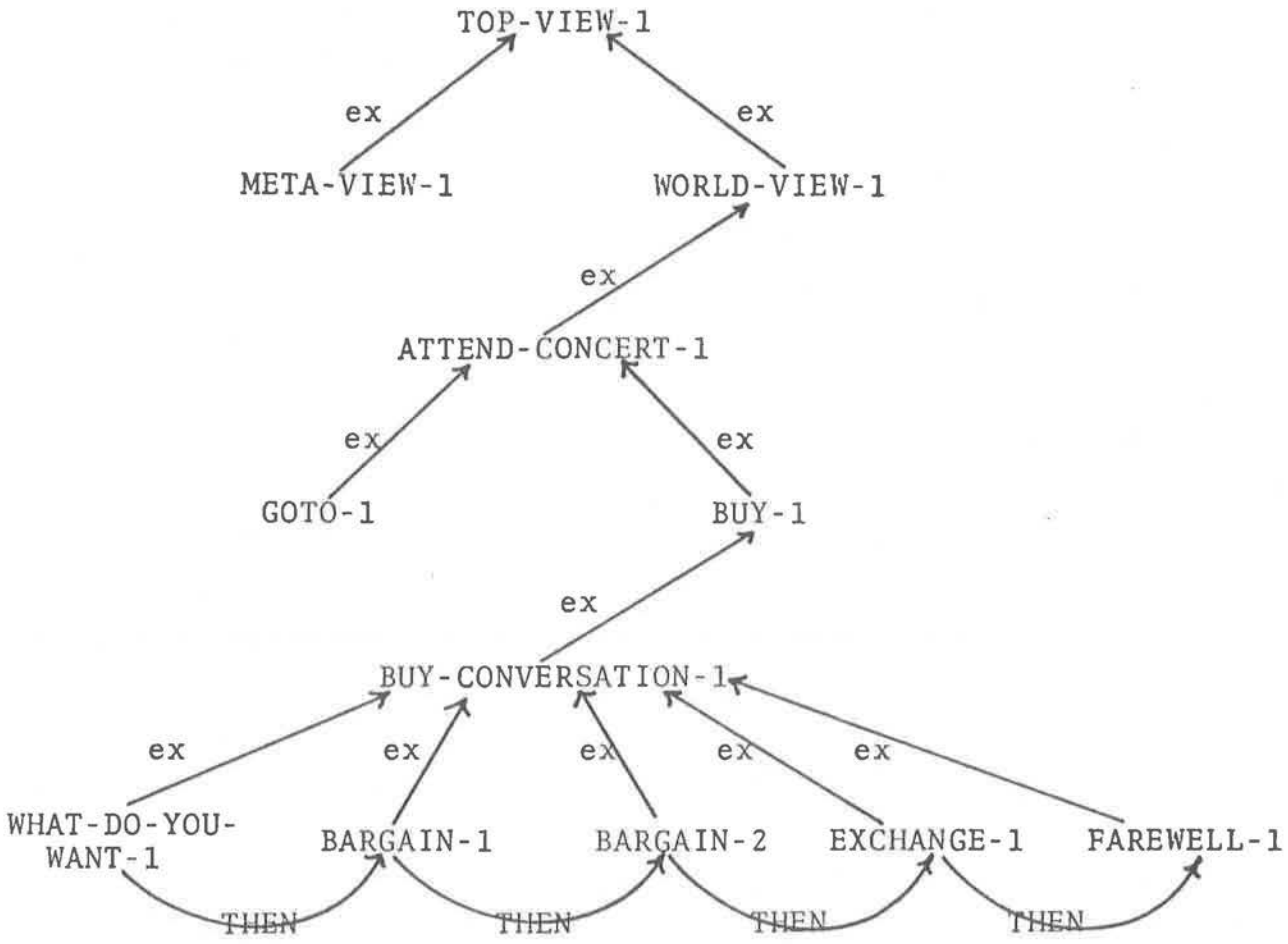


Figure 5.5 -
Goal Tree for the Ticket Buying Conversation

and will require such abilities as recognizing equivalent concepts in different utterances or performing conversational implicatures and real world inferences to link up diverse kinds of discourse. Right now, since the conversation proceeds more or less as expected, tieing-in is minimized (consisting only of recording the sequence of sub-scripts) since the EVENT-SEQUENCE macro code, containing the expected conversational structure, is available for inspection, and it will correspond closely to the tailored reality.

The second and third steps of BUY-CONVERSATION involve haggling over the characteristics of the item to be purchased: STEP2 over the characteristics the buyer wants; STEP3 over those the seller wants. Thus, in the ticket buying situation, the bargaining first involves the theatre location represented by the ticket, then the cost of the ticket. This is a fairly simplified version of what happens in a buying situation; BUY-CONVERSATION predicts things at this skeletal level in an attempt to be as general as possible. People, however, seem to have many much more specialized scripts that operate in particular contexts (e.g. department store scripts, oriental bazaar scripts, probably even ticket buying scripts). Moreover, even within a script, there seems to be a large degree of flexibility that enables skipping the actual verbalizations of large chunks of the script, allowing them to be inferred instead.

At any rate, assuming that BUY-CONVERSATION has successfully carried out the bargaining, the agreed upon location and cost are added to the model's knowledge of the ticket (i.e. TICKET-FOR-CONCERT1). Then, an EXCHANGE of the appropriate amount of money for the desired ticket can take place (STEP4). This is interesting in that it is not basically a verbal action, but is instead a motor action. The indistinguishability of verbal from other actions allows this kind of intermixture to be easily and conveniently accomplished. Theoretically at least, gestures and any other method of communication are feasible.

The final portion of BUY-CONVERSATION is a call to FAREWELL, a sub-script which will make utterances that terminate the conversation much as WHAT-DO-YOU-WANT started it.

This concludes the BUY-CONVERSATION script, which returns the current execution instance, complete with the tied-in utterance patterns and any other information accumulated during the execution of the script. Thus, the original message pattern match and the BUY-CONVERSATION-1 execution instance is bound to CONV-RESULT in BUY-1, and BUY-1 is resumed. It can then make use of any information that it needs from this execution instance.

A final note about executing BUY-CONVERSATION: the execution here has been with the model as buyer and the ticket seller as seller, but the model could take the opposite role or in fact take both roles or neither role. This flexibility is extremely useful since this one script can handle many different situations.

The subgoals called during execution of BUY-CONVERSATION are themselves scripts. The first one to be executed is WHAT-DO-YOU-WANT, a subset of the more general GREETING pattern expression.

5.4.2 WHAT-DO-YOU-WANT

The WHAT-DO-YOU-WANT script (see Appendix II) is somewhat different from BUY-CONVERSATION in that it actually calls in speech act pattern expressions to direct the interpretation or production of utterances rather than the subscripts above. As usual, an EXECUTE message is sent, in this case

```
(EXECUTE WHAT-DO-YOU-WANT
     TICKET-SELLER1 SELF ?WHAT-WANT-CONV)
```

which matches the receiving pattern

```
(EXECUTE WHAT-DO-YOU-WANT
     ?SPEAKER1 ?SPEAKER2 !(EVENT-SEQUENCE --- ))
```

Everything matches except the last elements where the standard EVENT-SEQUENCE computation must take place first. This computation proceeds by establishing the conversational identities of the speakers, that is determining whether to

interpret (expect) an utterance from speaker-i or whether to produce (execute) an utterance for speaker-i. Essentially, if speaker-i is SELF, then the script must generate an utterance whenever it is speaker-i's turn to talk; otherwise, speaker-i is scmebody else and the script should attempt to comprehend an utterance from speaker-i. It is possible for the script to be used in situations where the model takes none, one, or both of the speaker roles.

The script contains 2 steps:

STEP1: send an EXPECT (or EXECUTE) message to the speech-act INQUIRE indicating that speaker-1 is to make an inquiry of speaker-2 as to the purpose of speaker-2;

STEP2: send an EXECUTE (or EXPECT) message to RESPOND indicating that speaker-2 should be responding to speaker-1's inquiry with his purpose.

Thus, for the ticket buying episode, the model should EXPECT speaker-2 (the ticket seller) to inquire as to the purpose of speaker-1 (the model) and should then EXECUTE a response indicating its purpose (obtained by looking into the execution environment for some pattern of the form (PURPOSE SELF ?WHAT); in this case the matching pattern
(PURPOSE SELF (BUY SELF TICKET-FOR-CONCERT1))
would be found).

An interesting problem here is determining just how far up the execution environment to go when looking for a match, since even higher order purposes could be strewn about at the upper levels (for example, here, to attend the concert). The assumption has been to go to the nearest such purpose, presumably the reason that the current subgoal has been called in.

Upon completion of the EVENT-SEQUENCE, the pattern matches and WHAT-WANT-CONV is bound to the new WHAT-DO-YOU-WANT-1 execution instance created to handle the message. Control then resumes in BUY-CONVERSATION-1 which goes to its next step: to BARGAIN between what the buyer wants and the seller has.

5.4.3 BARGAIN

Before actually calling in the BARGAIN script, BUY-CONVERSATION must first set up bargaining positions for the seller and buyer. The essential idea is that in bargaining for anything (including a ticket) each bargainer has two initial positions: he owns something and wants something else in exchange for it. A successful bargaining session will match what party A wants with what party B owns and also will match what party A owns with what party B wants. In the ticket buying situation the model itself wants a ticket representing a certain desired location and suspects that the ticket seller owns such a ticket; the model owns a certain small amount of money which it is willing to exchange in return for the ticket and suspects that the seller wants money appropriate to the cost of the ticket.

The model's bargaining positions are role dependent and are thus accessed up ROLE-INSTANCE-OF pointers from SELF as outlined earlier. In this case SELF-AS-A-BUYER-OF-CONCERT-TICKET

```
<|PDEF SELF-AS-A-BUYER-OF-CONCERT-TICKET
   (SUPERSET SELF-AS-A-BUYER-OF-CONCERT-TICKET
             SELF-AS-A-BUYER)
   (BUY SELF ↑TICKET-FOR-CONCERT)
   (WANT SELF
     (EXCHANGE
       SELF-HAS-TICKET-BARGAINING-POSN
       SELF-WANTS-TICKET-BARGAINING-POSN))  >
```

would be found in which resides the pattern

```
(WANT SELF
   (EXCHANGE SELF-HAS-TICKET-BARGAINING-POSN
             SELF-WANTS-TICKET-BARGAINING-POSN))
```

containing the model's bargaining positions.

The ticket seller's bargaining positions, or at least the model's perceptions of the ticket seller's bargaining positions, can be accessed from TICKET-SELLER1 by inheriting the

```
(WANT ↑TICKET-SELLER
   (EXCHANGE TICKET-SELLER-HAS-BARGAINING-POSN
             TICKET-SELLER-WANTS-BARGAINING-POSN))
```

pattern from TICKET-SELLER (there being no ROLE-INSTANCE-OF pointer in TICKET-SELLER1).

Thus, the BARGAIN |PEXPR (see Appendix II) is sent the message

```
(EXECUTE BARGAIN TICKET-SELLER1 SELF
      TICKET-SELLER-HAS-BARGAINING-POSN
      SELF-WANTS-BARGAINING-POSN
      TICKET-FOR-CONCERT1 ?BARGAIN1-CONV)
```

which will match the BARGAIN pattern

```
(EXECUTE BARGAIN ?QUESTIONER ?RESPONDER
      ?POSN-Q ?POSN-R
      ?ITEM !(EVENT-SEQUENCE --- ))
```

if the EVENT-SEQUENCE computation succeeds.

The purpose of the computation is to bring about a successful compromise between two probably initially distinct bargaining positions. In this example one of the positions is TICKET-SELLER-HAS-BARGAINING-POSN

```
<|PDEF TICKET-SELLER-HAS-BARGAINING-POSN
       (SUPERSET TICKET-SELLER-HAS-BARGAINING-POSN
                 HAS-BARGAINING-POSN)
 S1     : (REPN ↑TICKET !(|POINTER REPN TICKET))
       (IMPORTANCE ↑TICKET-SELLER /S1 6)
       (BARGAIN-ORDER TICKET-SELLER-HAS-BARGAINING-POSN
                      (REPN))  >
```

representing knowledge which the model believes the ticket seller to hold in regard to his wants. Looking at the |PEXPR, the following can be gleaned:

(i) by S1, the model believes the ticket seller to have an individual ticket (↑TICKET) which represents any location suitable to that ticket (!(|POINTER REPN TICKET));

(ii) the model believes the importance of fact (i) to the ticket seller is 6;

(iii) the model believes that bargaining can take place in regard to the location the ticket represents. The BARGAIN-ORDER pattern not only designates which patterns of the bargaining position are suitable for negotiation, but also indicates the order of resolution of various pieces of information (for example in the drink buying episode, the bargaining at this stage involves settling both the brand of the drink and the kind of mixer, in that order, rather than just the single issue of ticket REPN here).

The other bargaining position contains the model's own desires for the characteristics of an ideal ticket:

```
<|PDEF SELF-WANTS-BARGAINING-POSN
       (SUPERSET SELF-WANTS-BARGAINING-POSN
                 WANTS-BARGAINING-POSN)
 S1 : (REPN ↑TICKET-FOR-CONCERT ↑SEATSQETCENTRE)
 S2 : (REPN ↑TICKET-FOR-CONCERT ↑SEATSQETLEFTCENTRE)
 S3 : (REPN ↑TICKET-FOR-CONCERT ↑SEATSQETRIGHTCENTRE)
      (IMPORTANCE SELF /S1 10)
      (IMPORTANCE SELF /S2 8)
      (IMPORTANCE SELF /S3 8)
      (BARGAIN-ORDER SELF-WANTS-BARGAINING-POSN (REPN))  >
```

Similar to the ticket seller's bargaining stance, the model has three positions regarding the location represented by the ticket. The model wants (IMPORTANCE 10) a seat in the centre portion of the QET; failing that, it would like a seat in QET left-centre or QET right-centre (with equal IMPORTANCE of 8).

Successful execution of the BARGAIN script (see Appendix II for the code) involves finding a compromise between these two initial bargaining positions according to the following steps:

> STEP1: the questioner (the ticket seller) asks the responder (the model) about the next issue (determined by the BARGAIN-ORDER in the position of the questioner, in this case REPN) and goes to STEP2. If there are no more issues to be resolved, the script terminates successfully, returning the current BARGAIN-1 execution instance.
>
> STEP2: the responder states his initial stance on the current issue (i.e. the most important pattern appropriate to the issue in his bargaining position).
>
> STEP3: the questioner extracts the responder's stated stance and matches it to the questioner's current stance (extracted from his bargaining position). If they match, the agreed upon stance is asserted in item and onto STEP5; else the questioner reduces his demands by taking the next most important stance in his bargaining position, inquiring as to the

suitability of that, and going to STEP4. If the demands cannot be further reduced (i.e. he has run out of patterns), then go to STEP7.

STEP4: the responder goes through a similar process of extracting a stated stance from the questioner's last utterance, matching it to his own stance, and, if successful, asserting the agreed upon stance in item before going to STEP6; else, reducing his demands then stating them then going to STEP3; and finally, if there is no way of reducing the demands, going to STEP8.

STEP5: the questioner, having matched his position on the current issue with that of the responder, agrees with him on the current mutually held stance and goes to STEP1 so that the next issue (as determined by BARGAIN-ORDER) can be resolved.

STEP6: the responder, having matched his position with that of the questioner agrees with him on the mutually held position, and goes to STEP1 so that the next issue can be resolved.

STEP7: the questioner, having found irreconcilable disagreement on the issue with the responder, says so, the responder says so as well, and the script terminates in failure.

STEP8: the responder, having found irreconcilable disagreement on the issue with the questioner, says so, the questioner says so as well, and the script terminates in failure.

When BARGAIN is called the first time by BUY-CONVERSATION

the following assignments are made: the questioner is TICKET-SELLER1, the responder is SELF, the questioner's bargaining position is TICKET-SELLER-HAS-BARGAINING-POSN, the responder's bargaining position is SELF-WANTS-BARGAINING-POSN, and the item is TICKET-FOR-CONCERT1. The initial issue, extracted from the bargaining |PEXPRs, is REPN, that is the contestants will discuss where the seat for the concert will be. The bargaining is very co-operative in this case (this is determined by the closeness of the stances in the bargaining positions), and the model and ticket seller eventually agree on a ticket representing a location in the centre of the QET.

Obviously, this is a rather stylized version of bargaining, but it does illustrate some interesting facets of |LISP and the analysis of language. For the first time generators are used to cycle through stances as the questioner or responder compromise during the course of the bargaining. Thus, the EXPR MOST-IMPORTANT will return the most important pattern in a |PEXPR matching a particular other pattern, but MOST-IMPORTANT is defined as a generator so it can later be restarted to generate the next most important pattern and so on.

The second interesting thing is how the model is able to use its expectations to extract the actual stance of its conversant from his utterances. The EXPR COVER-PATTERN will look through the execution instance generated in the interpretation or production of an utterance for a pattern (CONTENT ex-instance ?ANYTHING) which will contain the basic content of what was said devoid of any "extraneous" things such as the speaker, the listener, the motivation for the utterance. Then, COVER-PATTERN will match its expected stance against patterns in the third element of the CONTENT pattern, hoping to find one which the stance covers in the sense that all elements of the stance are equal to or in the ISA environment of all elements of the CONTENT pattern. If such a covering match is found, the CONTENT pattern will replace the stance in the model's further deliberations.

If such a covering is not found, then the |PEXPR

EXPLAIN-BAD is asked to figure a way around things. As with CHECK-FAILURE, EXPLAIN-BAD is a largely unspecified failure handler which needs much more elaboration than has currently been given it. The whole problem of handling unexpected information and integrating it into a script is one that has been given only cursory treatment and clearly needs much more work. Presently, BARGAIN and other scripts naively assume that all will go as expected or that CHECK-FAILURE, TIE-IN, EXPLAIN-BAD, or the like will be able to explain anomalies so that processing can continue according to script.

When BARGAIN-1 is returned to BUY-CONVERSATION-1, the next step of BUY-CONVERSATION-1 initiates a new round of BARGAINing, this time with the questioner and responder roles reversed. It is quite likely, therefore, that the same speaker (the model or the ticket seller) will speak back-to-back utterances as the last step of BARGAIN-1 and as the first step of the new BARGAIN (called, say, BARGAIN-2). This illustrates the ease of sequencing utterances appropriately, even across script boundaries.

BARGAIN-2, of course, works in a similar manner to BARGAIN-1 with SELF-HAS-BARGAINING-POSN and TICKET-SELLER-WANTS-BARGAINING-POSN (see Appendix II) being the two new bargaining positions, and COST obviously being the issue at hand. Early agreement will be reached here as well, thus completing the bargaining.

5.4.4 EXCHANGE

When BARGAIN-2 is finished, the motor-action EXCHANGE is executed (see Appendix II). Of primary significance here is the similarity between motor-actions and speech actions, and the ease with which the two can mix. The message

```
(EXECUTE EXCHANGE SELF TICKET-SELLER1
      DOLLARS-10 TICKET-FOR-CONCERT1 ?EXCH-RESULT)
```

matches the pattern

```
(EXECUTE EXCHANGE ?PERSON1 ?PERSON2
      ?ITEM1 ?ITEM2 !(EVENT-SEQUENCE --- ))
```

with the appropriate bindings being made and the execution of

the EVENT-SEQUENCE ensuing. This sequence has four steps:

STEP1: where PERSON1 gives to PERSON2 the first item, in this case the model gives to the ticket seller 10 dollars.

STEP2: PERSON2, the ticket seller, says thanks.

STEP3: PERSON2, the ticket seller, gives to PERSON1, the model, the second item, the ticket for the concert.

STEP4: PERSON1, the model, says thanks.

Thus, steps 1 and 3 are motor-actions; steps 2 and 4 are speech acts. Note that if the model is the instigator of a particular action (speech or otherwise) the message will be EXECUTE; otherwise it will be EXPECT. In the linguistic case an EXECUTE means to generate an utterance; an EXPECT means to interpret an utterance. In the non-linguistic case an EXECUTE means to perform the action (e.g. giving); an EXPECT means to expect somebody else to perform the action, so that if the model is expecting a give, say, then it should expect to see certain actions such as hand movements, and should prepare itself to take the proferred item if the model is the intended recipient.

5.4.5 FAREWELL

After executing the EXCHANGE, the final step in the BUY-CONVERSATION script is entered: the execution of the FAREWELL script which merely consists of an exchange of goodbyes between the two participants. |LISP code for this |PEXPR also appears in Appendix II. This successfully ends the BUY-CONVERSATION script. BUY-CONVERSATION now returns to BUY which is also done so it returns to ATTEND-CONCERT to proceed to the next step in the concert plan.

5.5 Speech Acts

Speech acts are a level of linguistic description that form an interface between the internal meanings of concepts and the external linguistic realizations of these concepts. Concerned with single speech actions of a specific speaker, they are less general than scripts but are more general than the purely linguistic "language level" (described in section 5.6).

During the discussion of scripts, speech acts were those pattern expressions that were sent a message to EXPECT or EXECUTE a single utterance. Thus, INQUIRE, RESPOND, AGREE, DISAGREE, etc. are speech acts; there could, of course, be many more such as INFORM or REQUEST, in line with the Austin (1962) / Searle (1969) designations. Obviously, the level of linguistic description promoted here as the speech act level has much in common with the speech act concept of Austin and Searle, although neither Austin nor Searle pursue a computational approach, nor do I talk in terms of "locutionary", "illocutionary", or "perlocutionary" forces. Furthermore, I do not believe that speech acts are the crucial linguistic factor, but rather I regard them as subgoals of more general plans. This view is similar to that of P. Cohen (1978) where speech acts are actually planned much as other actions can be planned.

5.5.1 INQUIRE

To demonstrate speech acts, it would be useful to take a particular example, say INQUIRE, and examine what it does. The code for INQUIRE is in Appendix II. There are two main types of message that can be received by a speech act: EXPECT, when the speech act will be uttered by someone else and thus needs to be interpreted; and EXECUTE, the old standby, implying the speech act needs to be produced.

Assume, first, that INQUIRE is a subgoal of the WHAT-DO-YOU-WANT-1 script activation and has been sent the message

(EXPECT INQUIRE TICKET-SELLER1 SELF

(PURPOSE SELF *UNKNOWN*) ?NEW-UTT)

Then, the receiving pattern

(EXPECT INQUIRE ?SPEAKER ?LISTENER
?CONTENT !(|PROG ---))

will match, with SPEAKER bound to TICKET-SELLER1; LISTENER to SELF; CONTENT (i.e. what the inquiry should be about) to (PURPOSE SELF *UNKNOWN*); and the (|PROG ---), after doing the actual expectation processing, to NEW-UTT. (A |PROG is used because speech acts seem a low enough level of analysis not to require the memory preservation features provided by EVENT-SEQUENCE).

The first thing the |PROG does is to check the execution environment to see if a surface level utterance already has been "heard". If not, then the "read buffers", the model's single "sense", must be emptied (using the EXPR HEAR-WORDS) of their words and read into a list called UTTERANCE. HEAR-WORDS is responsible for reading the words, separating them from one another, performing morphological analysis on them; the possibility of ambiguities at this level is ignored. A unique list of words that have to be interpreted by the speech act and its language subgoals is produced.

A possible extension here would be to have HEAR-WORDS construct a pattern expression (rather than a list) to contain the words of the input. This |PEXPR would keep a complete record of sequencing and timing information, morphology decisions, as well as the words themselves. Whenever any other |PEXPR needed to access the words this |PEXPR could be queried. Such an extension awaits further experimentation.

Having read the utterance (and asserted it in the speech act's execution instance for the reference of subgoals) the speech act checks to see if any subset or subinstance of itself would be more appropriate for the interpretation of that utterance. Thus, it calls in the EXPR CHECK-FOR-ACTIVE-SUBSET which will look to see if there is any pattern expression that is a subset or subinstance of INQUIRE and that has been activated associatively (or in some other non-goal directed way)

and wants to run. Although non-goal directed processing will be discussed in Chapter VI, a brief introduction to associative activation is presented now.

5.5.2 The Associative Activation of YES2

A pattern expression does not always have to be called in to accomplish a subgoal of another pattern expression. It can instead receive an ASSOC message from a closely related |PEXPR that itself has been activated in one of three ways: (i) top-down as a subgoal of another pattern expression, (ii) associatively by the reception of an ASSOC message (see YES2 in Appendix II for an ASSOC pattern which might receive such a message), or (iii) (ultimately) directly by the presence of external stimuli. For example having spawned the BUY |PEXPR, a natural association might be the BUY-CONVERSATION script even before it is actually called in top-down; recognizing a DOG might trigger associations to BARK, TAIL, and other "doggy" things.

If enough of these associative triggers contact a pattern expression, it would likely consider itself strongly relevant to the current situation and would consequently like to be integrated into the top-down scheme of things. To this end, it "turns itself on" ("lights up") in the hopes that top-down |PEXPRs will notice it and try to incorporate it. Such recognition is done by EXPRs like CHECK-FOR-ACTIVE-SUBSET called in at the discretion of the top-down |PEXPRs (just when to make such checks is a difficult problem).

In this case INQUIRE asks if there are any active subsets of itself which, for whatever reason, want to be integrated into the model's goals. Assume that the word "YES"[1] has been read and has spread an associative activation trace to the speech acts YES1 (affirmative agreement) and YES2 (what can I do for you?), among others. YES1 and YES2 would both consider the

---

[1] A notational convention: any |PEXPR whose name is surrounded by " " marks stands for an actual word, not an internal concept representing the meaning of a word.

presence of the word "YES" as being conclusive evidence they are relevant to the current situation, so they would "light up". INQUIRE, looking for only those |PEXPRs which are subsets or subinstances of itself, would see YES2 (but not YES1) and hence decide that there is indeed an active subset of itself which would be appropriate here. It therefore decides to replace itself by the more specific YES2. Notice that YES1, although associatively active, will not get incorporated into the top-down context and hence will eventually atrophy and disappear (more work for the garbage collector!)

How does such replacement work? There is an EXPR called REPLACE to do this. It receives as "argument" the |PEXPR which is to replace the current |PEXPR. The message currently being processed (and all other messages left to process from the original message form) is then matched against patterns in the replacement |PEXPR in exactly the same way as they were in the original using the same execution instance, complete with all the patterns that are there already. Just two changes are made to this execution instance: first, the current stack is emptied; and second, an additional
(EX-INSTANCE-OF ex-instance REPLACEMENT-|PEXPR) pattern is added. This method of replacement allows results which have already been computed and asserted to be saved; moreover, the two execution instance pointers allow access to information from either of the two "ISA" |PEXPRs, the old one or the replacement one.

In the current example the execution instance INQUIRE-1 would have its stack emptied and have a
(EX-INSTANCE-OF INQUIRE-1 YES2) pointer added. The model would then try to match the message

```
(EXPECT YES2 TICKET-SELLER1 SELF
      (PURPOSE SELF *UNKNOWN*) ?NEW-UTT)
```

against patterns of YES2 as for a normal message; that is, YES2 becomes the new subgoal.

5.5.3 YES2

YES2 (see Appendix II) is the speech act which handles the specific inquiry "yes?" When YES2 receives the EXPECT message, it looks to see whether an utterance has been read. It has, since before the replacement took place INQUIRE had managed to do the reading of the words and this result has been saved in INQUIRE-1. Having already checked for associatively active |PEXPRs, the processing continues immediately to asserting the speaker of and the listener to the speech act.

The speech act then proceeds to interpret the utterance. In this case that merely involves looking for the words "YES" and "?" in order and alone in the input utterance (this hasn't been done yet top-down by YES2, only bottom-up). If they are not found, then the input is not considered to be a YES2 so a failure pattern is left in the execution instance, and the match fails. The failure to match processing which ensues may find a matching EXPECT pattern elsewhere (for example possibly the INQUIRE EXPECT pattern would be re-instigated); but, if not, then the failure pattern should provide valuable information to explanation procedures in the "calling" |PEXPR in determining what went wrong and what to do about it.

If the words "YES" and "?" are found, then this confirms the YES2 speech act. If somebody says "yes?", then he is asking what you want, assuming that you already know. Thus, the "meaning" of YES2 is the expected content (passed as part of the EXPECT message) and this is so indicated by asserting a pattern which in this case is
(CONTENT INQUIRE-1 (PURPOSE SELF *UNKNOWN*)). If there is no expected content, i.e. CONTENT has not been initialized during pattern matching, then the content defaults to (PURPOSE !LISTENER *UNKNOWN*); that is, it is assumed that the speaker is inquiring as to the unknown purpose of the listener.

Once interpretation has been accomplished, then the (|PROG ---) is done, so the newly endowed execution instance INQUIRE-1 is returned as value, the entire pattern matches, and control resumes in the calling |PEXPR, in this case the

WHAT-DO-YOU-WANT-1 script instantiation.

Now, this interpretation has been rather trivial: in most cases much more complicated kinds of processing need to be done, so complicated in fact that the speech act usually has to call on language-oriented processes to "parse" the input. For example if INQUIRE (Appendix II) is examined, it can be seen (after the discovery of speaker, listener, etc.) that to find the content of the utterance, INQUIRE sends an INTERPRET message to INQUIRE-CLAUSE in order to extract meaning from surface level language. In the next section more will be said about this "language level" of the model's processing.

I will conclude this section with a brief discussion of the other major message type that speech acts handle: the EXECUTE message. A |PEXPR representing a regular action, if EXECUTEd, performs the action; similarly, a |PEXPR representing a speech act, if EXECUTEd, produces the speech act. Thus, if either INQUIRE or YES2 is sent an EXECUTE message with the traditional speaker, listener, and content slots filled, then it is up to the speech act to generate the utterance appropriate to the content (or some default verbalization if the content is unassigned). In the case of YES2, this is simplicity itself: merely call SPEAK-WORDS to print out the words "YES" followed by "?".

SPEAK-WORDS, the generation analogue of HEAR-WORDS, must insert appropriate punctuation, ensure word ending agreement, and the like. It currently accepts as argument a list, but as with HEAR-WORDS, the eventual goal is to put the words in a pattern expression which can contain much meta-information having to do with the words, and have SPEAK-WORDS work with that.

For most speech acts, more complex kinds of generation require GENERATE messages to be sent to language level |PEXPRS. Of this more in the next section.

A final note indicating the underlying symmetry of interpretation and generation: regardless of whether it ran in EXECUTE or EXPECT mode, the speech act, when done, will have

asserted four patterns of importance:

(SPEAKER --- ---)
(LISTENER --- ---)
(CONTENT --- ---)
(SURFACE --- ---)

where the CONTENT is given and SURFACE produced for EXECUTE messages and vice versa for EXPECT messages. The CONTENT pattern particularly is used by scripts (e.g. BARGAIN) in discerning the meaning of a speech act, although all other patterns are also available.

The other speech acts have similar EXPECT and EXECUTE patterns associated with them. The main ISA-linkages connecting them are shown in Figure 5.6.

5.6 The Language Level

Letting a specific speech act direct the interpretation or production of an utterance works well if expectations are precise enough. Unfortunately, it is the rare conversation which proceeds as expected, and even when expectations are more or less met, there can be deviations in phraseology which can leave the speech act bewildered. Thus, most interpretation and production is done with the help of |PEXPRs whose expertise is in the area of language rather than in the conceptual domain of the speech act which called them in. The collection of these |PEXPRs I call the language level. The language level is not, as yet, much more than a skeleton of the kinds of things that need to go on.

The language level corresponds roughly to the syntactic level of processing. Differences will become obvious as the description proceeds, but the major ones are that the syntactic constraints are very loose and that semantic processing is interleaved throughout the syntax. A further distinction from traditional parsing strategies is that sometimes there is no need to use the language level at all (e.g. the speech act YES2 itself looks directly at the utterance).

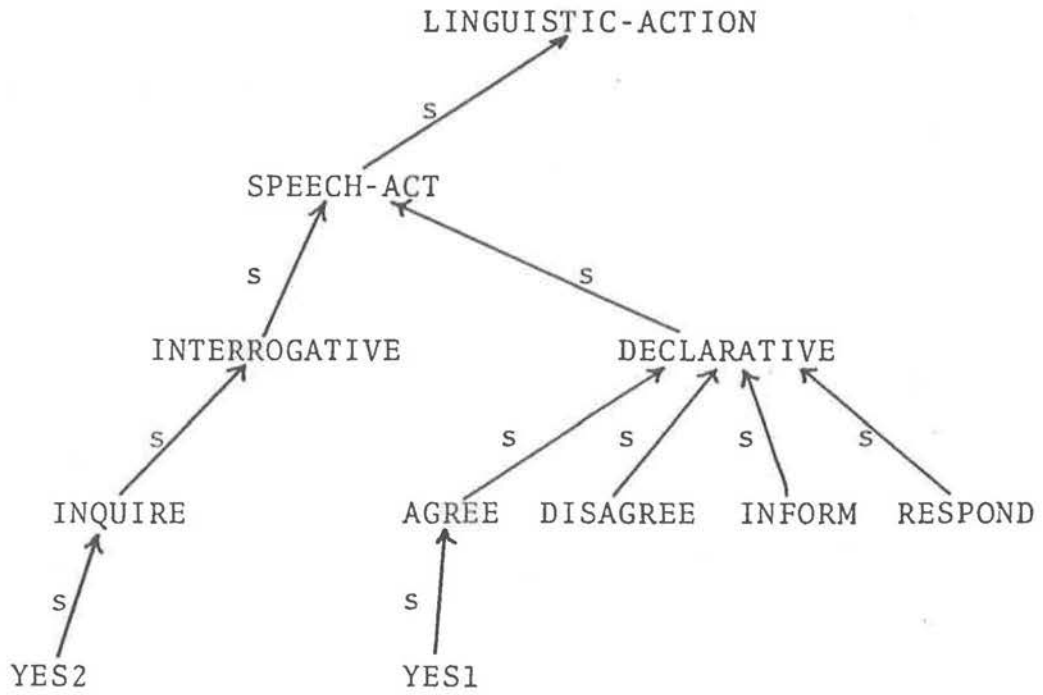


Figure 5.6 -
Speech Acts in the Model

The general flow of control at the language level during interpretation (in traditional top-down style) is to separate an utterance into clauses, and then divide clauses into noun groups, verb groups, preposition groups, and sub-clauses (which are, in turn, further broken down). Memory concepts representing the "meaning" of the noun groups, verb groups, and preposition groups are then built. These memory concepts are then linked together to yield an interpretation of the clause of which they are a part. Finally, the clause interpretations are linked together to form an interpretation for the whole utterance which is returned to the speech act for its perusal. The top-down breakdown of utterances into groups is similar to Winograd's (1972) parsing strategy; the amalgamation of groups via message passing has elements of a case approach (e.g. Taylor and Rosenberg (1975)).

So, the language level is concerned with partitioning an input utterance into sub-groups. There is a pattern expression for each type of group: UTTERANCE, COORD-CLAUSE, SUB-CLAUSE, REL-CLAUSE, NOUNG, VERBG, and PREPG. Of course, there can be more specific groups which refine the processing of their superiors, for example a BECAUSE-SUB-CLAUSE, or even more specifically a BECAUSE-I-HATE-CHEESE-SUB-CLAUSE. Figure 5.7 illustrates an ISA hierarchy of such word group !PEXPRs.

Word group !PEXPRs receive messages like any other !PEXPRs. Two of particular interest are INTERPRET and GENERATE, the former to understand a conversant's words and the latter to prcduce words for the model. In the following discussion I will concentrate almost exclusively on interpretation even though generation is just as crucial to a conversation.

5.6.1 Interpreting UTTERANCEs and CLAUSEs

UTTERANCEs and CLAUSEs are interpreted by being broken into NOUNGs, VERBGs, PREPGs, and sub-clauses. Certain clauses may have extra punctuation or conjunctions not incorporated into any group - these are merely noted and have use later when a clause is asked how it modifies some other concept (see below).

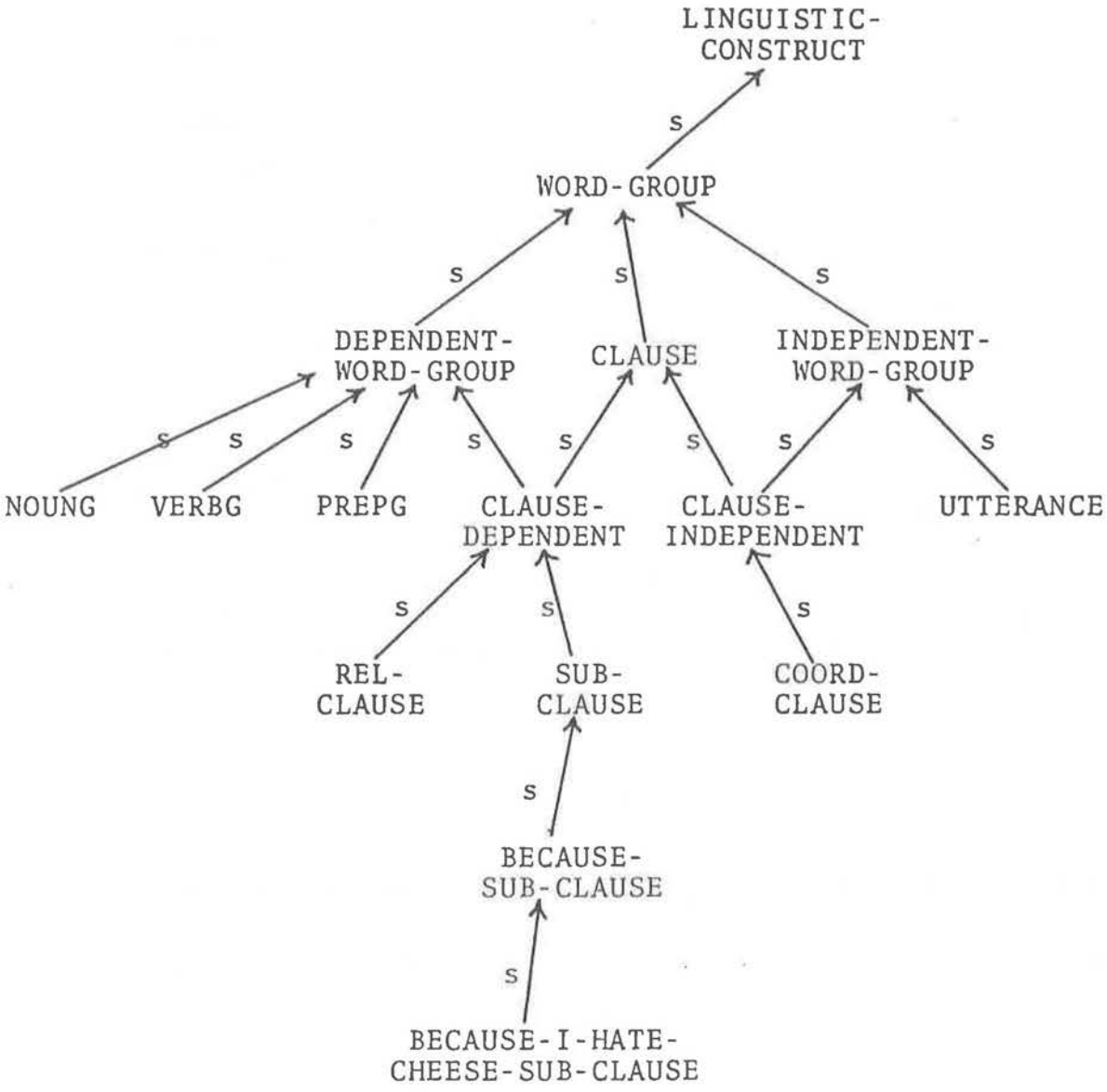


Figure 5.7 - Word Group Hierarchy

A group is asked to break itself into constituent sub-groups by sending it a message of the form

```
(INTERPRET language-level-|PEXPR
           (list of words) ?RESULT)
```

For example, suppose a speech act (e.g. INFORM) has read and wants to interpret this utterance:

"Beethoven composed the Fifth symphony with extraordinary skill. He was a great man."

Then the speech act would send a message

```
(INTERPRET UTTERANCE
    ("BEETHOVEN" "COMPOSE" "PAST" "THE" "FIFTH"
     "SYMPHONY" "WITH" "EXTRAORDINARY" "SKILL"
     "." "HE" "BE" "PAST" "A" "GREAT" "MAN" ".")
    ?RESULT-OF-INTERP)
```

to UTTERANCE. The double-quoted atoms above name word |PEXPRs containing patterns linking the word to its internal concept(s) and to its part of speech. To illustrate, "COMPOSE" might look something like

```
<|PDEF "COMPOSE"
   (INSTANCE-OF "COMPOSE" VERB)
   (CONCEPT "COMPOSE" COMPOSE)    >
```

Upon receipt of an INTERPRET message, UTTERANCE would divide the list of words into two co-ordinate clauses and pass INTERPRET messages to each COORD-CLAUSE, i.e.

```
(INTERPRET COORD-CLAUSE ("BEETHOVEN" --- ".") ?INTERP1)
(INTERPRET COORD-CLAUSE ("HE" --- ".") ?INTERP2)
```

Each of the COORD-CLAUSEs could be split into NOUNGs, VERBGs, PREPGs, or sub-clauses and interpret messages would then be sent to these constituents; e.g.

```
(INTERPRET PREPG ("WITH" "EXTRAORDINARY" "SKILL") ?INT14)
(INTERPRET VERBG ("COMPOSE" "PAST") ?INT12)
```

Returned as a result of any such interpretation is a <u>new</u> |PEXPR representing the concept described by the NOUNG, VERBG, PREPG, or sub-clause. When all constituents of a clause have been interpreted, they must be linked together, so various pairs of them are sent MODIFY-HOW messages. Thus, the new |PEXPR representing the interpreted main verb of a clause may be asked how it is modified by the newly interpreted NOUNG just preceding it; e.g.

```
(MODIFY-HOW VERBG-COMPOSE NOUNG-BEETHOVEN ?MOD-HOW1)
```

In this case, since Beethoven has all the characteristics

required of an agent for VERBG-COMPOSE (precedes the verb, is animate, is a composer), VERBG-COMPOSE would be happy to add to itself the pattern

(AGENT VERBG-COMPOSE NOUNG-BEETHOVEN)

Similarly, a NOUNG occurring after the VERBG might be incorporated by asking VERBG-COMPOSE

(MODIFY-HOW VERBG-COMPOSE NOUNG-SYMPHONY ?MOD-HOW2)

which, since symphonies have all the qualities needed to be a patient for the verb group would result in VERBG-COMPOSE adding

(PATIENT VERBG-COMPOSE NOUNG-SYMPHONY)

Note the case aspects of MODIFY-HOW processing: the newly created verb concept (using an inherited MODIFY-HOW procedure) is making all the decisions about what to check for, what patterns to add, and what other processing to carry out.

Other NOUNGs, PREPGs, and REL-CLAUSEs are similarly linked into the NOUNGs or VERBGs which they modify. In addition two clauses may have to be linked together. For a COORD-CLAUSE and a SUB-CLAUSE, the message would be to the concept representing the dominating COORD-CLAUSE, i.e.

(MODIFY-HOW COORD-CLAUSE-concept
SUB-CLAUSE-concept ?MOD-HOW3)

while for two COORD-CLAUSEs, the message would be to the concept representing the first COORD-CLAUSE. In cases involving PREPGs, REL-CLAUSEs, SUB-CLAUSEs, or dominated COORD-CLAUSEs, the linking word (preposition, relative conjunction, subordinate conjunction, co-ordinate conjunction, or punctuation mark) is available. The |PEXPR corresponding to this linking word can thus be accessed during MODIFY-HOW processing to add precision to the link-up. For example, the clause "I like stereo because it beats mono." breaks into two clauses that are interpreted as COORD-CLAUSE-LIKE and SUB-CLAUSE-BEAT. As a result of interpretation, SUB-CLAUSE-BEAT would contain a pattern

(SUB-CONJUNCTION SUB-CLAUSE-BEAT "BECAUSE")

so that when COORD-CLAUSE-LIKE tries to see how SUB-CLAUSE-BEAT modifies it, this pattern can be accessed, and the pattern

(REASON COORD-CLAUSE-LIKE SUB-CLAUSE-BEAT)

can be added to COORD-CLAUSE-LIKE (REASON being the meaning of "BECAUSE" in this context).

After all constituents of a CLAUSE or UTTERANCE are connected, the interpretation of the CLAUSE or UTTERANCE is complete, so the concept associated with the main verb is returned to whoever wanted the CLAUSE or UTTERANCE interpreted (another clause or utterance, or a speech act). This is because the main verb, so central in all MODIFY-HOW processing, and thus linked so closely to the rest of the concepts, in a sense represents the central meaning of any clause or utterance.

Throughout this processing, the CLAUSE and UTTERANCE |PEXPRs must choose how to divide words into appropriate sub-groups and must decide on an order in which to relate the sub-groups to one another. To make such decisions, traditional parsing expertise must be augmented by the ability to look into the execution environment for various kinds of information. Thus, the identity of the speech act which initiated the proceedings might be important (an INQUIRE speech act, for example, might suggest an interrogative grammatical construction). Similarly, it might be useful to consult the conversant models in order to discover individual peculiarities of phraseology. If, despite this information, a mistake is made, the CLAUSE or UTTERANCE must be able to try alternatives.

5.6.2 Interpreting NOUNGs, VERBGs, and PREPGs

NOUNGs and VERBGs represent concepts in memory. Thus, a VERBG usually stands for some relation; a NOUNG for some real world (e.g. Ruff) or abstract (e.g. the present King of France) object. The basic interpretation strategy for any such group is therefore to produce a new pattern expression which represents the concept stated in the group.

NOUNGs:

The interpretation of NOUNGs can best be described with an example. Assume that NOUNG is sent a message to interpret

("THE" "FIFTH" "SYMPHONY"). The first thing is to find concepts that are candidates for the meaning of each word (using the words' (CONCEPT ---) patterns). In the case of multiple candidates for a word, the NOUNG makes some choice, based on the importance of the concept to the word if such is available. (Of course, the NOUNG must be prepared to make another choice if the first one fails tc work out).

After doing this "dictionary look-up", the NOUNG picks off the concept associated with the noun (say SYMPHONY) and adds the pattern

(ROLE-INSTANCE-OF NOUNG-SYMPHONY SYMPHONY)

to the current execution instance (called NOUNG-SYMPHONY here); that is, the execution instance is playing the "role" of SYMPHONY in this context. Although it is debateable whether ROLE-INSTANCE-OF is the appropriate pointer name to use here, I feel it is more convenient to use an already existing link type with an inheritance feature built in (since much information relevant to the new concept will have to be inherited from SYMPHONY as well as from NOUNG). Of interest is that when future generations look at NOUNG-SYMPHONY, they will know by the EX-INSTANCE-OF pointer that it is a NOUNG, by the ROLE-INSTANCE-OF pointer that it represents the concept SYMPHONY in this particular "role", and finally by the EX-ENVIRON pointer the context in which the words were issued.

Next, the NOUNG must take care of all modifiers (adjectives, classifiers, determiners, etc.) in the group. Thus, the concept FIFTH (associated with the adjective "FIFTH") is sent a message

(MODIFY-HOW FIFTH NOUNG-SYMPHONY ?MOD-HOW7)

i.e. how does FIFTH modify a new role instance of SYMPHONY? FIFTH knows that it is the fifth number of whatever it modifies, so it adds

(NUMBER-OF NOUNG-SYMPHONY 5)

to NOUNG-SYMPHONY. If the modification means that NOUNG-SYMPHONY is now recognized to be a role instance of some more specific |PEXPR (e.g. FIFTH-SYMPHONY), then the old

ROLE-INSTANCE-OF pointer is removed and a new one added that points to the more specific !PEXPR. This, too, is done by the adjective concept during receipt of a MODIFY-HOW message.

NOUNG-SYMPHONY now resumes execution and checks if there are any other modifiers in the group. Indeed there is one more: the determiner "THE". The concept associated with "THE", THE, is sent

(MODIFY-HOW THE NOUNG-SYMPHONY ?MOD-HOW8)

THE performs an analysis similar to that for adjectives except that the main task of a determiner is to determine whether the noun group is a role instance of a new instance or merely a role instance of an old instance. The usual case with THE is that it designates an old instance (although not always). In fact here THE does signify that NOUNG-SYMPHONY is a role instance of an existing instance, so THE proceeds to try to find the instance: the candidates are instances of SYMPHONY or its subsets. If THE is able to find the particular symphony that fits, for example BEETHOVEN-SYMPH-FIVE, then a

(ROLE-INSTANCE-OF NOUNG-SYMPHONY BEETHOVEN-SYMPH-FIVE)

is added to NOUNG-SYMPHONY and the ROLE-INSTANCE-OF pointer to SYMPHONY is removed. If THE is unable to find the particular symphony that fits, then

(ROLE-INSTANCE-OF NOUNG-SYMPHONY ↑SYMPHONY)

is added to NOUNG-SYMPHONY (i.e. NOUNG-SYMPHONY is a role instance of some subinstance of SYMPHONY yet to be determined) and the old ROLE-INSTANCE-OF pointer is removed. Later, when other pieces of information become available, this pointer can be further specified. A final note: if a NOUNG refers to a new instance (as is sometimes the case with the article "A"), then the new instance is created (using the EXPR CREATE-NEW) and the ROLE-INSTANCE-OF pointer is directed to this new instance.

VERBGs:

Verb groups are handled in a manner similar to noun groups: the execution instance created when VERBG is sent an interpret message is a role instance of some particular action concept

associated with the main verb of the verb group. Adverbs and auxiliaries in the group further specify the time of the action, the manner of the action, the importance of the action, etc. To handle this, the concepts associated with verb modifiers are sent MODIFY-HOW messages which result in extra patterns adorning the verb group execution instance.

PREPGs:

A preposition group is interpreted by separating its object noun group and interpreting it. An additional pattern

(PREPOSITION NOUNG-ex-instance "prepostion")

indicating the preposition that heads the PREPG is asserted in the NOUNG's execution instance (and later used, perhaps, during MODIFY-HOW processing). The NOUNG execution instance is then returned as the "meaning" of the PREPG.

5.6.3 Generation

I will give here only a brief suggestion of what I have in mind for generation at the language level. A speech act receives an order

(EXECUTE SPEECH-ACT SPEAKER LISTENER CONTENT ?OUT-WORDS)

where CONTENT is either a |PEXPR name or a list of |PEXPR names representing the concept to be spoken. The speech act can sometimes directly output the CONTENT (as for THANKS, say, or YES2), but most often it must call on the language level UTTERANCE, CLAUSE, NOUNG, VERBG, etc. |PEXPRs to help it produce nice output. Thus, INQUIRE needs to use a subset of UTTERANCE, INQUIRE-UTTERANCE perhaps, to generate its output. INQUIRE-UTTERANCE would thus be sent a message

(GENERATE INQUIRE-UTTERANCE CONTENT ?OUT)

In response it would be expected to take the CONTENT description containing one or more |PEXPRs and figure out what patterns in the |PEXPRs to verbalize and what to leave non-verbalized (this would require, among other things, the examination of the belief models for the model itself and the conversant, available in the execution environment). For each pattern that is to be

verbalized, sub-clauses, adjectives, adverbs, preposition groups, must be constructed, and these must modify verbs and nouns in proper groups. Thus, at various stages PREPG, COORD-CLAUSE, SUB-CLAUSE, NOUNG, VERBG, etc. can be sent GENERATE messages.

When the time comes to get surface words for internal concepts, patterns such as

(SURFACE BUY "BUY")

associated with the concept |PEXPRs can be used. As each clause or group is completed, the words in that conglomeration are returned in a list to be amalgamated into other words returned from other clauses or groups. Eventually, words are returned to the speech act, which, using SPEAK-WORDS, outputs them.

Obviously, this is a very brief outline of the potential of the model in the difficult task of generation. Much further analysis needs to be done before any claims can be made in this area. For a good description of the problems that lurk in generating natural language output, read Wong (1975).

5.7 Conversations II and III

At the beginning of the chapter two other conversations, the drink buying episode and the conversation with a "friend" at intermission, were proposed for analysis. I do not intend to do very much explanation here because the basic principles have already been presented and the complexities of the third conversation have not yet been sufficiently well analyzed for detailed discussion. However, it is essential to demonstrate the generality of the approach outlined in the previous discussion.

The drink buying episode takes place as STEP7 of the ATTEND-CONCERT plan (see Appendix II) after the model has listened to the first half of the concert and gone out for a break at intermission. It turns out to be almost identical to the ticket buying episode: BUY is called with SELF as the agent, and with the desired drink being computed by asking

PRE-DINNER-DRINK (see Appendix II) the question (WANT SELF (DRINK SELF ?DESIRED-DRINK)). Since the model is attending a concert, PRE-DINNER-DRINK computes the model's desires to be a new instance, JOHNNY-WALKER-SCOTCH1, of the JOHNNY-WALKER-SCOTCH |PEXPR (see Appendix II). JOHNNY-WALKER-SCOTCH is, in turn, a subset of SCOTCH (see Appendix II). Thus, the ITEM to BUY is JOHNNY-WALKER-SCOTCH1.

BUY then is EXECUTEd and it finds the location of the drinks (QETBAR, discovered by asking JOHNNY-WALKER-SCOTCH1 which inherits it from PRE-DINNER-DRINK which computes it by asking for the location of the BAR at the location of the current event). The model goes to QETBAR where it expects to find the seller of JOHNNY-WALKER-SCOTCH1 to be a new instance, BARTENDER1, of BARTENDER (see Appendix II). This knowledge has been inherited by JOHNNY-WALKER-SCOTCH1 from the generic ALCOHOLIC-DRINK |PEXPR (see Appendix II). The people in Figure 5.8 are thus known to the model.

The BUY-CONVERSATION script is then EXECUTEd exactly as in the ticket buying episode to achieve the desired drink. As for ticket buying, bargaining positions must be obtained. The bargaining positions for BARTENDER1 can be determined by asking BARTENDER1, which inherits it from BARTENDER by accessing the EXCHANGE pattern there. The bargaining positions for the model are determined by asking SELF what it wants to exchange, and this information is inherited along ROLE-INSTANCE-OF links from SELF-AS-A-BUYER-OF-ALCOHOLIC-DRINKS in this situation. Thus, the model owns SELF-HAS-DRINK-BARGAINING-POSN and wants in exchange SELF-WANTS-DRINK-BARGAINING-POSN while the bartender has BARTENDER-HAS-BARGAINING-POSN and wants in exchange BARTENDER-WANTS-BARGAINING-POSN (all these "bargaining position" |PEXPRs are in Appendix II). It is interesting to note the ISA connections of the various bargaining positions known to the model, displayed in Figure 5.9.

The two BARGAIN sub-scripts of BUY-CONVERSATION (see section 5.4.3) are once again the most interesting aspects of the script. The first matches SELF-WANTS-DRINK-BARGAINING-POSN

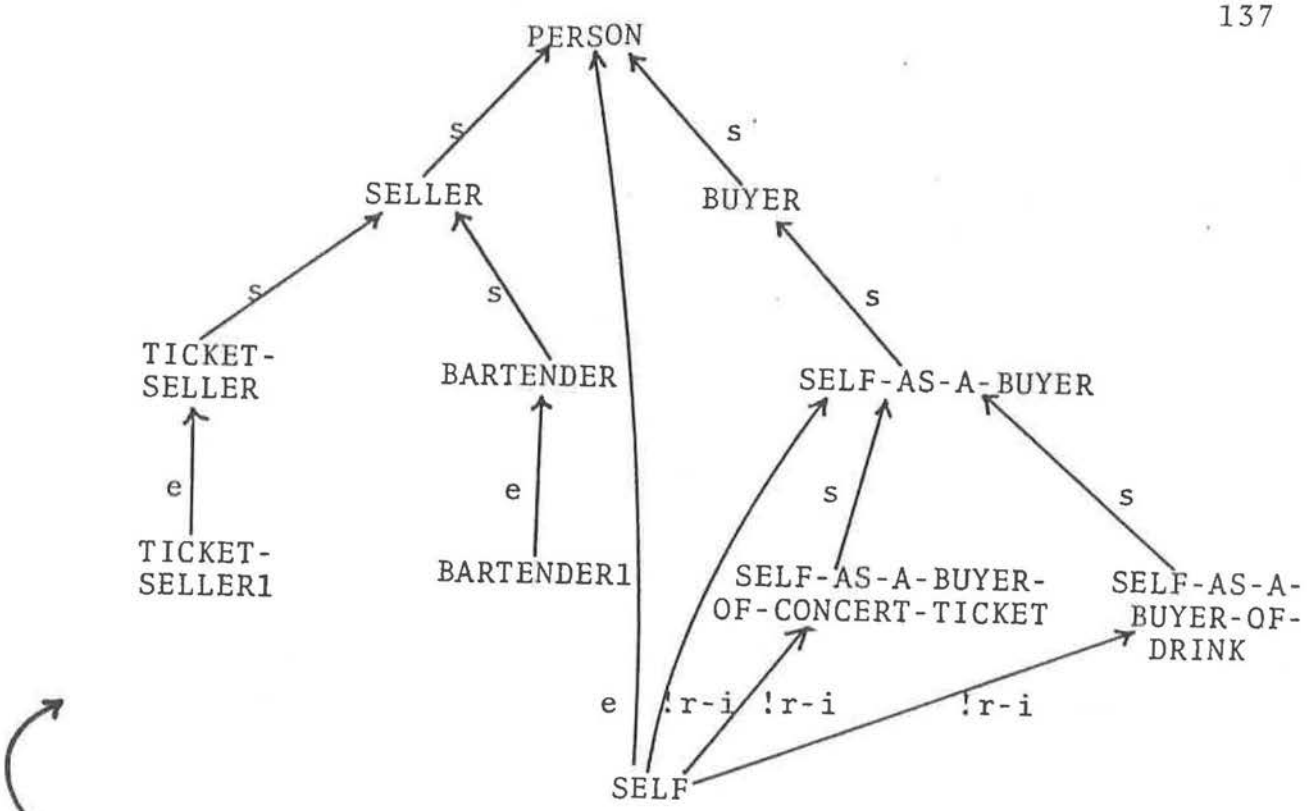
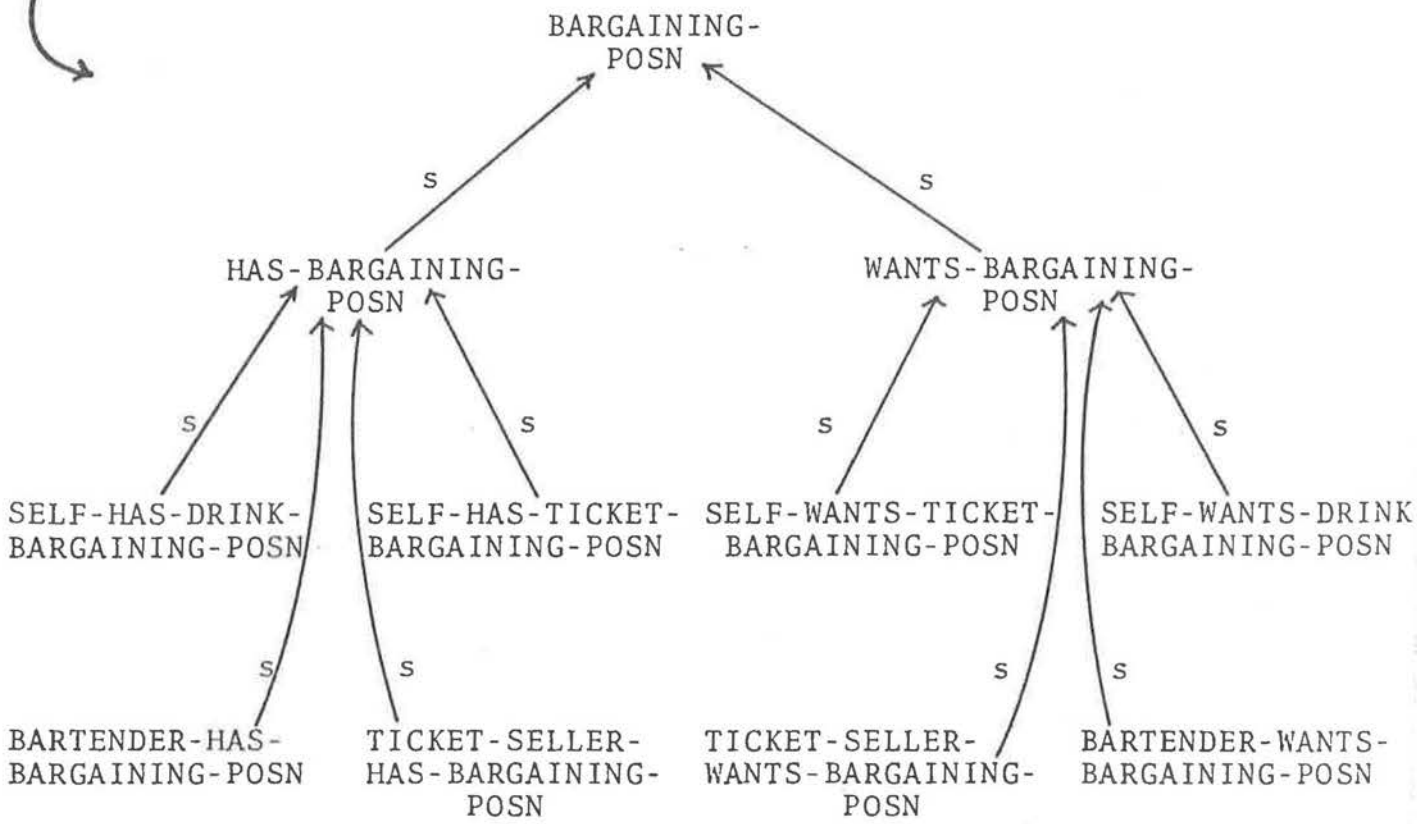


Figure 5.8 - People

Figure 5.9 - Bargaining Positions



with BARTENDER-HAS-BARGAINING-POSN and the second matches BARTENDER-WANTS-BARGAINING-POSN with SELF-HAS-DRINK-BARGAINING-POSN. The first bargaining differs from the ticket buying in that two things must be settled, namely the brand, then the mixer (see the BARGAIN-ORDER pattern). The second bargaining settles the cost of the drinks much as did the ticket buying situation.

The speech act and language level !PEXPRs are more or less the same as in the ticket buying situation so need not be discussed.

Upon completion of the drink-buying conversation, the ATTEND-CONCERT pattern expression resumes execution at STEP8 where the model, in parallel, fills in time and drinks its drink until it hears the buzzer marking the end of intermission. This particular set of subgoals has only been added to show the !DO-PARALLEL !EXPR (see Appendix I) in operation and the possibility of the model doing several things at once. The FILL-IN-TIME !PEXPR would be a very liberal plan which more or less allowed bottom-up suggestions to direct it as to how to pass the time. Thus, if the model "sees" an interesting mural on the wall, FILL-IN-TIME might decide to build and call a subplan to go look at the mural, or if the model is a smoker, FILL-IN-TIME might invoke a subgoal to light up a pipe.

When the model encounters a friend, FILL-IN-TIME would have to first of all recognize the friend and bring in its model of the friend, and secondly discover some script which would enable the model to talk with the person. Details as to how this would be accomplished have not been worked out, although it is clear that bottom-up and associative capabilities would be strongly employed in doing both tasks.

Assuming the conversant model and the script have been built, what would such a script look like? It would start with some sort of greeting script akin to WHAT-DO-YOU-WANT which would launch the conversation, to be followed by the most general two-person script DIALOGUE (see Appendix II). The DIALOGUE script expects SPEAKER1 to produce some speech act,

then SPEAKER2 to perform some speech act, then SPEAKER1, and so on, until a failure occurs. Note that a special failure checking routine called CHECK-FAILURE-DIALOGUE is used that would explain a lack of input by suggesting that DIALOGUE is done rather than as an error as CHECK-FAILURE might do.

COMPUTE-EXPECTATIONS is used to discover what to say at any stage based on what the speaker has said (available from the script), what he believes (available from his belief model), what the other speaker has said, what the other speaker believes, and on the model's knowledge about the various topics of discussion (contained in objects accessible in searches from |PEXPRs whose names can be found in the execution environment). This is, of course, extraordinarily difficult, and I can at best speculate on how it would work.

Probably one of the major ways of deciding what to say is to compare patterns in one speaker's belief model with those in the other's model, and also to compare their beliefs with their utterances in the conversation to date. Depending on their attitude towards one another, either contradictions or similarities could be emphasized in choosing amongst the comparisons. If the two parties are in an adversary position (e.g. in a debate), then important contradictions would be crucial; if the two are in a co-operating mood (e.g. in purchasing something), then important similarities would be emphasized; if one person is in authority over the other (e.g. in a job interview), the subservient one would be co-operative, the authority would be perhaps contrary; and so on. Some of the ideas in my first progress report (McCalla (1973)) would be relevant here.

Some sort of general conversation takes place, is concluded, the model and the conversant exchange farewells, and the model returns from FILL-IN-TIME to the execution of the ATTEND-CONCERT plan. The plan continues with the model listening to the second half of the concert, then going back home. Control at last returns to the execution of WORLD-VIEW where some other plan must be constructed and undertaken.

Thus, "one day in the life" of the model has been presented. Discussed has been the scope of the representation scheme, the type of language processing I envisage, and the problems which arise in using |LISP to undertake the processing. In the next chapter I examine some of the generalizations which can be extracted from these examples, and indicate some of the ways a system such as this can be extended to handle other kinds of phenomena.

CHAPTER VI
Generalizations and Extensions

The examples of the last chapter indicate in detail how the various parts of the model fit together. In this chapter I discuss some of the more general representation features in the model, to suggest what the model means for the analysis of language, and to illustrate some potential extensions to this kind of system.

6.1 Representation Properties of the Model

Chapter V has exposed a plethora of patterns, pointers, and macros which together define the relevant information in the model. A surprisingly small number of dimensions suffice to delineate the bulk of this information. I would like to investigate the following major categories: the ISA hierarchy, the PART-OF hierarchy, the execution environment, "one-shot" relations, and procedural knowledge.

6.1.1 The ISA Hierarchy

The ISA hierarchy consists of pattern expressions connected by INSTANCE-OF, ROLE-INSTANCE-OF, EX-INSTANCE-OF, or SUPERSET pointers. It is the standard generalization hierarchy (e.g. Quillian (1969), Levesque (1977), Fahlman (1975), Roussopoulos (1976)) which allows lower concepts to inherit information from concepts higher in the hierarchy.

It is generally a property of the ISA hierarchy that the higher up a |PEXPR is, the less specific is its information since (potentially) many sub-|PEXPRs can inherit from it. For example, looking at the ISA environment emanating from JOHNNY-WALKER-SCOTCH, it can be seen that stored with JOHNNY-WALKER-SCOTCH is very specific information as to the cost, the brand, and other features of this kind of Scotch; with its superset, SCOTCH, is more general knowledge about mixers that go with Scotch, and the model's preferences among Scotches;

with SCOTCH's superset, PRE-DINNER-DRINK, is more general information yet about the contexts in which the model wants to drink pre-dinner drinks; with its superset, ALCOHOLIC-DRINK, is the rather broad knowledge that bartenders serve alcoholic drinks; and so on up the hierarchy. Of course, it is always possible (although the examples have not shown it) to override any general information by placing more specific information lower down in the hierarchy; e.g.
(MIXER ↑TEACHERS-SCOTCH ↑WATER) in TEACHERS-SCOTCH would be an exception to the general "scotch-mixer" being ice.

Inheritance occurs during pattern matching when a source pattern has failed to match any target patterns. If this happens it is possible to look into the ISA environment surrounding the target object for a matching pattern. The two interesting things here are that inheritance is only attempted

(i) a particular piece of information is needed by some other ↓PEXPR and it can't be found directly;
and (ii) when the information is of a type appropriate for inheritance to be tried (determined by failure to match processing associated with the information itself).

The examples in Chapter V illustrate at least two types of ISA inheritance which I label <u>direct</u> ISA inheritance and <u>dependency</u> ISA inheritance. Direct inheritance occurs when the unmatched source pattern can be directly matched to patterns in the ISA environment <u>without modifying the source pattern</u>. Thus, in the drink-buying episode when it becomes important to discover the SELLER of the newly created SCOTCH1, the source pattern (SELLER SCOTCH ?WHO) is constructed, and an attempt is made to match it in SCOTCH1. This fails, so SELLER directs that the INSTANCE-OF pointer to JOHNNY-WALKER-SCOTCH be followed, where a direct match is undertaken for a pattern matching the source. This also fails, so the SUPERSET pointer to SCOTCH is followed (and another failure to match occurs), the SUPERSET pointer to PRE-DINNER-DRINK is followed (and still another failure to match occurs), and finally its SUPERSET leads to ALCOHOLIC-DRINK. There, the pattern

(SELLER ↑ALCOHOLIC-DRINK ↑BARTENDER) is discovered which matches (SELLER SCOTCH1 ?WHO) directly since SELLER = SELLER, SCOTCH1 is a SUB-INSTANCE of ALCOHOLIC-DRINK, and ?WHO matches ↑BARTENDER by creating a new instance of BARTENDER, say BARTENDER1, that is bound to WHO in the source object. The pattern (SELLER ↑ALCOHOLIC-DRINK ↑BARTENDER) has been in some sense "directly" inherited by SCOTCH1.

It is typical of most directly inheritable patterns that they are stored with a class, but their pattern elements pertain to instances (as indicated by "↑" macros in front of their elements). Thus, the class ALCOHOLIC-DRINK contains the pattern (SELLER ↑ALCOHOLIC-DRINK ↑BARTENDER) indicating that the SELLER of any sub-instance of ALCOHOLIC-DRINK is a sub-instance of a BARTENDER. An interesting observation is that use of =(---) macros would allow the placement of arbitrary conditions on a pattern element just as "↑" imposes the condition "sub-instance of" on its element, enabling the user of |LISP to more finely tune inheritance to his particular needs.

The other kind of inheritance, dependency inheritance, involves modifying the source pattern as ISA links are traversed. This most frequently occurs when the item being inherited is a property of the class itself rather than being a property of the instances of the class with which it is stored. The second element of the pattern is, thus, usually the class name and when a sub-class (or instance) is trying to inherit the pattern it must take this into account.

The most commonly inherited such patterns are the "procedural patterns" designated by EXECUTE, EXPECT, INTERPRET, etc. which are associated with their containing |PEXPR regardless of whether or not it is an instance. For example if the message pattern

(EXECUTE ATTEND-CONCERT SELF CONCERT1 ?ATTEND-RESULT)

were sent to ATTEND-CONCERT, and there was no matching pattern, EXECUTE would be called in to suggest what to do. EXECUTE patterns are dependently inheritable with the second element being dependent on (in fact equal to the name of) the |PEXPR

which contains the pattern. So, in this case the ATTEND |PEXPR (SUPERSET to ATTEND-CONCERT) would be searched for a pattern matching the modified source
(EXECUTE ATTEND SELF CONCERT1 ?ATTEND-RESULT)
and if this failed, the superset of ATTEND, say PLAN-OF-ACTION, would be searched for a pattern matching
(EXECUTE PLAN-OF-ACTION SELF CONCERT1 ?ATTEND-RESULT).

The next most common situation requiring dependency matching is when an execution instance or role instance is inheriting from an instance or another execution instance or role instance. In this case any references to the execution instance or role instance must be considered to be dependent, at least until the instance is encountered. Thus, if RUFF-1 is an execution instance of RUFF, a pattern (HOWL RUFF-1) would be modified to (HOWL RUFF) when searching RUFF. Note that, although the match is dependent when going from RUFF-1 to RUFF, it would be direct if the search were carried out in the ISA environment above RUFF, since this is above the instance level.

Dependency matches illustrate the value of forcing the semantics of failure to match processing to be explicitly stated in the pattern head, since the pattern head can do anything appropriate to the kind of information represented in patterns of that kind. In fact, it would even be possible to do away with pattern elements such as "↑" and let the pattern head realize that the relation it represents connects instances rather than classes. For example the |PEXPR SELLER might know that (SELLER ALCOHOLIC-DRINK BARTENDER) actually is meant to relate instances of ALCOHOLIC-DRINKs to instances of BARTENDERs. I felt, however, that it would be better to explicitly suggest via macro modification the conditions on various elements in a pattern. Another user of |LISP is, as always, able to make his own decisions in this regard.

6.1.2 The PART-OF Hierarchy

The PART-OF hierarchy connects pattern expressions by the pointer PART-OF, a link of recognized importance (see, for example, Levesque (1977), Havens (1978)). A |PEXPR, Y, is connected by PART-OF to another |PEXPR, Z, if the referent for Z is an aggregate consisting at least in part of the referent for Y (e.g. a pattern (PART-OF LEG-OF-RUFF RUFF) in RUFF).

As with ISA searches, PART-OF searches can be carried out in response to a failure to match; for example in the case of certain "attribute" patterns such as (COLOUR ---) or (TEXTURE ---), etc. where the attribute of the part is likely to be the same as the attribute of the aggregate. Moreover, such searches should be carried out with dependency PART-OF inheritance in force, if applicable. Thus, to find a pattern matching the source (TEXTURE LEG-OF-RUFF ?WHAT) might require patterns in LEG-OF-RUFF to be considered, then, if this failed, patterns in RUFF. The source would have to be changed to (TEXTURE RUFF ?WHAT).

PART-OF is also instrumental in intersection searches where it is necessary to see if several |PEXPRs are all PART-OF some more general |PEXPR. For example in the discussion of how the model updates its LOCATION pointer (section 5.3.2), PART-OF intersection was used to determine the level of geographic detail affected by a change in the model's position. Many other uses can be envisaged. Interestingly, there is a similar necessity for ISA intersection. For instance it is necessary in the semantic portions of the language level to check whether certain word concepts intersect up ISA links with the concept ANIMAL (i.e. to see if something is animate). For many examples of this phenomenom see Quillian (1969) and Fahlman (1975).

6.1.3 The Execution Environment

The execution environment is accessed from any particular execution instance by following EX-ENVIRON pointers to execution instances of higher level goals. Patterns can be matched in the execution environment (using direct matching techniques) if they

represent the type of information which changes as subgoals are undertaken. Thus, the LOCATION of the model, the current object on which the model's ATTENTION is focussed, and the like, would require looking up EX-ENVIRON pointers if necessary.

In addition the execution environment forms the basic context mechanism of the model in that it focusses the model's attention on the things relevant to the current goals. Thus, at the language level, the model can discover the current speaker by looking into the execution environment for the controlling speech act where a pointer to the relevant model of the speaker is recorded. Knowledge gained from this model may enable the current utterance to be more easily interpreted or generated than it would be without the contextually relevant speaker model. This is a fairly typical use of the execution environment: execution instances in it contain patterns pointing to relevant |PEXPRs which in turn contain patterns pointing to slightly less relevant |PEXPRs further away from the execution environment, and so on.

Thus, contextually relevant information is that information you can get at from the current execution environment. Deciding when to put something into the execution environment then becomes a major problem. There is no hard and fast rule about putting things into the context, but by far the most common way is for a script, or plan, or whatever to know directly by name the "foregrounded" |PEXPR to choose. The next most common way is by simple search up ISA or EX-ENVIRON links for data of certain kinds (e.g. to find (SELLER TICKET1 ?WHO) look up ISA pointers). Sometimes other searches can be used, as in the case of top-down expectations using associative activation (combined with ISA intersection) to exclude the YES1 affirmative answer meaning of "YES". Many more possibilities exist.

Execution environments also form the basis for a sort of episodic memory, consisting of particularly relevant execution environments incarnated when the model was achieving previous goals. These old episodes not only still maintain their EX-ENVIRON and EX-INSTANCE-OF pointers, but they also still

contain all other relevant patterns that were asserted during their active existence.

Accessing information in an old execution environment allows the entire context to be seen as it was then, with the exception, of course, that it is being viewed from the changed perspective of the current context. This gives a facility similar to the ALINK / CLINK distinction of Bobrow and Wegbreit (1973) frames in that it allows processes in one context to access information from another. Moreover, it extends this facility, since it is perfectly possible to have execution instances of execution instances of execution instances, and so on, and hence have many levels of context.

If an execution instance has been created as a subgcal of an EVENT-SEQUENCE, there will be a THEN link connecting it to the next subgoal at its level. Thus, old episodes are often stored as graphs looking something like Figure 6.1 (where each box is an execution instance created during EVENT-SEQUENCE processing).

Various episodic memory searching procedures can be visualized (EX-ENVIRON searches from some execution instance; scans from execution instance to execution instance along THEN links; searches along THEN links with EX-ENVIRON searches at each stage; etc.) Which search is suitable would depend on the reasons for undertaking the search. Since episodic memory hasn't been actualized in the examples I have considered, further detail as to its properties will have to await experimentation.

6.1.4 One-Shot Relations

There are a number of "relations" in the model called "one-shot relations" which are crucial but which don't connect to one another in long chains as do ISA, PART-OF, or EX-ENVIRON. Examples include concepts such as COMPOSER, SELLER, BUYER, CONCEPT, SURFACE, AGENDA, etc., which constitute the major portion of the model's knowledge base. These "relations" in their very ubiquity indicate that the model is not based on

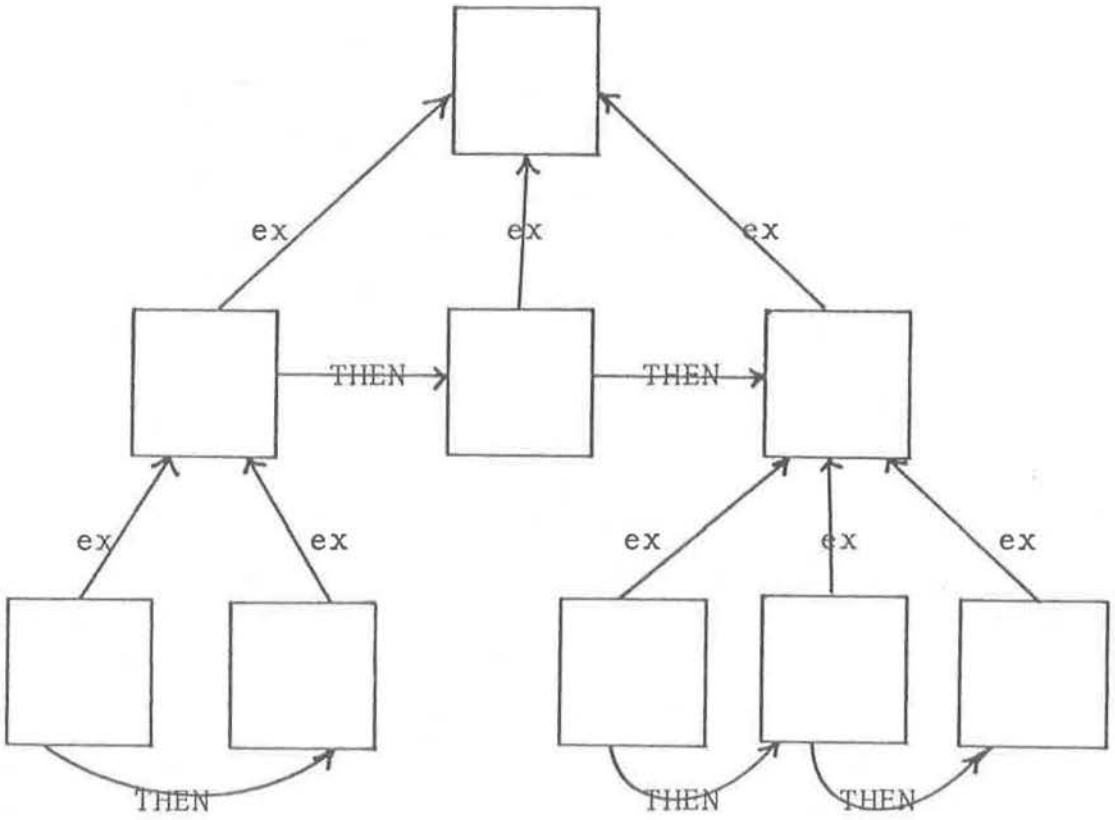


Figure 6.1 - An Episode in Memory

universal primitives but rather that any convenient domain dependent "primitives" can be chosen, can be endowed with a relevant name, given appropriate semantics (in how they handle failure to match, in how many elements patterns headed by them take, in IF-ADDED or IF-REMOVED processing, etc.), and used appropriately.

6.1.5 Procedural Knowledge

The final important aspect of the model's memory organization are the overtly procedural patterns in the memory. Thus, EXECUTE, EXPECT, INTERPRET, ASSOC, ... all designate patterns which undertake most of the model's active processing. The future hope is to reduce the number of COND, CDR, CAR, CONS, ... primitives and submerge them into higher level "knowledge-based" primitives. This would not only prune the size of these patterns and make them more readable, but would also go a long way towards making the model self-examinable. The following main types of procedural pattern have been used thus far:

Top-Down Patterns:

(i) EXECUTE: The main top-down aspect of the model, such patterns are used to undertake the "action" of the containing IPEXPR if the intended actor is the model itself. Thus, an EXECUTE sent to ATTEND-CONCERT puts into action the concert plan; an EXECUTE sent to a speech act will have the model speak the act; and so on.

(ii) EXPECT: The "dual" to EXECUTE, this kind of pattern is used when somebody else is expected to do the action. If somebody other than the model were to EXCHANGE an item with the model, then the model would EXPECT it; if the conversant were to undertake a speech act, then the model would EXPECT it. The close relationship that sometimes exists between EXECUTE and EXPECT is evidenced by the fact that an EXPECTed GIVE of a ticket from the ticket seller to the model is almost equivalent to the model EXECUTEing a TAKE of a ticket from the ticket

seller.

Language Patterns:

(i) GENERATE: This type of pattern is just the EXECUTE for the language level, distinguished from EXECUTE because this is a more traditional name in language processing, and also because GENERATE patterns do not take the speaker or listener as "arguments".

(ii) INTERPRET: This is the EXPECT for the language level, distinguished from EXPECT for reasons similar to those given for the EXECUTE / GENERATE distinction.

(iii) MODIFY-HOW "case" procedures: These are the case style messages (elucidated in section 5.6) to be used to check the semantic relationships among linguistic concepts.

Patterns Associated with Pattern Heads:

(i) IF-ADDED: This procedural pattern is associated with |PEXPRs that can be used to designate the head of a pattern (e.g. FIND in (FIND GOD)). Whenever a pattern is asserted, its head is asked if there are any IF-ADDED "by-products" which need to be done.

(ii) IF-REMOVED: This procedural pattern is similarly invoked to take care of side-effects resulting from the removal of a pattern.

(iii) FAILURE-TO-MATCH: This procedural pattern is associated with |PEXPRs that can be used as pattern heads and is called into play when a pattern with the appropriate head has failed to match in some |PEXPR (in a sense this makes it an IF-NEEDED method). It is this kind of pattern which determines whether inheritance, inference, or something else entirely must be undertaken in an attempt to overcome the failure.

Other Procedural Patterns:

(i) ASSOC: This pattern type contains the procedural information that is necessary when a |PEXPR is contacted in a non-goal-directed way. One example of this was indicated in

Chapter V where the YES2 speech act replaces INQUIRE because of ASSOCiative information. More non-top-down message types would be needed to handle sensory input, bottom-up PART-OF amalgamation, etc. (see section 6.2.4 for further speculation on this and the use of ASSOC messages to give a demon capability to the model).

(ii) context dependent patterns: Any pattern can contain a procedural element (such as "=" or "↑" or "!"), especially when it is necessary to encode context dependent information. The examples of the last chapter illustrated numerous such patterns (e.g. ROLE-INSTANCE-OF, drink preferences, bargaining positions).

6.2 Possible Extensions

In this section I would like to suggest several additional features which would enable the model of the last chapter to to handle a wider and more interesting array of problems. Some of these extensions involve figuring out how to use features already provided by ¡LISP; other extensions require an expansion of ¡LISP's abilities. The suggestions here are meant to be evocative of the kind of things that would be needed, rather than to be rigorous proposals with solid solutions.

6.2.1 Using Meta Patterns

¡LISP provides the user with the ability to arbitrarily nest patterns through the use of meta patterns, i.e. patterns which contain labelled patterns within them. In the model of Chapter V minimal use was made of meta patterns and no thorough classification of the kinds of things they could be used for was attempted. There are a number of possible uses for meta patterns.

(i) IMPs: Winograd (1975) uses the term "IMP" when discussing certain patterns that are in some sense more important than others. Various kinds of IMPs can be encoded using meta patterns. Already seen in Chapter V have been IMPs

such as

```
S3 : (IMPORTANCE SELF /S1 9)
S4 : (IMPORTANCE SELF /S2 4)
```

where S1 and S2 are patterns such as

```
S1 : (COST ↓TICKET-FOR-CONCERT ↓DOLLARS-5)
S2 : (COST ↓TICKET-FOR-CONCERT ↓DOLLARS-10)
```

One possible extension of IMPs is to even higher order nestings:

```
S5 : (IMPORTANCE SELF /S3 4)
```

Another possible extension is to allow different kinds of IMPs. Thus, as will be seen in section 6.2.4, it would be possible to designate certain patterns as being core patterns of a |PEXPR; e.g.

```
(CORE |PEXPR-name (/S1 --- /Sn))
```

Such patterns would be crucial to the definition of a |PEXPR, and would fill a role similar to Becker's (1969) criterial concepts or to the definitional concepts discussed by Woods (1975).

But, such a binary distinction is, perhaps, too simplistic and more subtle distinctions of relative importance among patterns may be needed. For example, psychological markers akin to those proposed by Tesler et al (1968) could be incorporated as meta patterns such as

```
(CHARGE SELF /S1 9)
(SIGNIFICANCE SELF /S1 7)
```

(ii) sequencing: There are many places where the sequence of some set of things is a key factor. Thus, the pattern matcher could be told to follow some sequence in choosing a target pattern; the conversation to date (in the script) could be kept as a sequence of elements, etc. This could be indicated using sequence meta patterns:

```
(SEQUENCE MATCH |PEXPR-name (/S1 /S2))
(SEQUENCE CONVERSATION |PEXPR-name (/S2 /S1))
```

showing that the pattern matcher should try to match pattern S1 then pattern S2, but that S2 preceded S1 in the conversation.

(iii) time: The myriad problems involved in handling time have been, by and large, ignored. The only reference to time has been the |TIME-LIMIT= and |TIME-NOW= patterns used by |LISP in the processing of interrupts. Clearly, it is necessary to

keep track of other times: the time when the model undertook various tasks, the time of internal events in the processing (interrupts, garbage collection, etc.), the time of events mentioned in discourse (historical time, the time of occurrence of various happenings in a story or movie).

An approach to handling this variety would be to devise "clock" |PEXPRs to represent different kinds of time. For example, there could be timers, internal clocks, historical clocks, story clocks, and so on. Such clocks would be responsible for keeping track of events relevant to them in time units appropriate to the events (seconds, days, eras). They could be queried for the time of occurrence of these events, could be sent messages to update themselves, and so forth.

Once such clocks were defined, meta patterns could be employed to record times associated with events. Thus,

(TIME ASSERTION WORLD-VIEW-CLOCK27 /S1 296)

might indicate the time of assertion of pattern S1 to be 296 according to an internal clock instantiation called WORLD-VIEW-CLOCK27.

(TIME OCCURRENCE TIME-LINE3 /S1 PAST)

might show that the event represented by the pattern S1 occurred in a PAST era according to TIME-LINE3, an instantiation of an historical clock; but

(TIME OCCURRENCE TIME-LINE2 /S1 FUTURE)

indicates that the same event occurred in a FUTURE era from the perspective of TIME-LINE2. Time is an interesting area needing much further exploration (see Bruce (1972), Kahn (1975) or R. Cohen (1976) for discussions of the problems of time).

6.2.2 Garbage Collection and Learning

Because many of the "temporary" |PEXPRs (execution instances, new instances, etc.) set up during processing turn out to be useful later in episodic and other considerations, I've talked of the necessity for doing garbage collection intelligently. Such processing is undertaken by a goal hierarchy headed by META-VIEW running in parallel to the

main-line WORLD-VIEW goals.

META-VIEW currently just calls the LISP garbage collector, clearly a much too limited capability. The first extension would be to clean up most execution instances created by WORLD-VIEW processing. It seems reasonable that all execution instances below a certain level of detail should be automatically destroyed (except, perhaps, in exceptional circumstances when the most trivial details might remain vividly in the system's memory). A general rule of thumb might be that if it hasn't been spawned as a subgoal of an EVENT-SEQUENCE, then it isn't worth keeping around, since the episode hasn't been deemed important enough by the user to warrant being connected up with THEN links. In the model of Chapter V the cutoff would occur beneath the script level. This heuristic presents problems (what happens if an EVENT-SEQUENCE is used in a subgoal of a non-EVENT-SEQUENCE?), and moreover ignores the fact that even many EVENT-SEQUENCE episodes should be destroyed, but otherwise seems reasonably appropriate.

Once execution instances are destroyed, other |PEXPRs will become expendable since they will be pointed to only from the destroyed execution instances (e.g. perhaps new conversant models would be eliminated when the episode in which they were created was destroyed). Furthermore, each time a pattern is removed during the destruction of an execution instance, its IF-REMOVED side-effect will be called to keep things consistent.

Many execution instances created as a by-product of accessing static pieces of data from secondary |PEXPRs are not records of subgoals in the same sense as are execution instances of primary |PEXPRs called in as subgoals. Such "secondary" execution instances should also be destroyed either using the EVENT-SEQUENCE criterion or, perhaps on the basis of having no "!" or "$" macros in the receiving pattern. Of interest here is the fact that this kind of garbage collection is unnecessary if |PATTERN or |POINTER |EXPRs are used when accessing patterns believed to be static. These |EXPRs violate modularity by locking directly into pattern expressions to match patterns; and

being !EXPRs, they create no new execution instances (using instead the current execution instance) and hence avoid the garbage collection problem by not creating any garbage in the first place. Obviously these are just preliminary suggestions: the whole garbage collection problem remains a major one.

A problem related to garbage collection is how to achieve the subtle kinds of memory modifications which loosely fit under the label "learning". The most basic kind of learning is the addition of new information to a system. Already it has been seen how WORLD-VIEW subgoals could create new !PEXPRs (such as TICKET-SELLER1 and BARTENDER1) and these could then be endowed with patterns as knowledge about them was gained. META-VIEW, not having direct access to the "real world", would have limited responsibilities for this kind of learning.

However, META-VIEW subgoals would be directly concerned with the kind of learning that involves generalization (a la Winston (1970)) up the ISA hierarchy. For example if every instance of PERSON had a pattern

(NUMBER-OF-LEGS instance 2)

then the concept

(NUMBER-OF-LEGS ↑PERSON 2)

should probably be added to PERSON, and the patterns with each instance could be removed. The "↑" macro would be used often as various specific elements in instances were "abstracted out". Of course, there remain many difficulties here.

META-VIEW would also have to undertake more devious kinds of learning, involving such things as the construction of plans from old episodes; the discovery of reasonable kinds of explanations for failure that can be incorporated into plans; the creation of "new" information based upon inferences made from already existing information; the recognition of the underlying similarity of two concepts and perhaps their amalgamation into one concept; and so forth. How to do these kinds of learning is clearly a major research question that has not been tackled at all. I mention the subject merely to indicate that META-VIEW will have to be at least as complex and

sophisticated as WORLD-VIEW.

6.2.3 Non-Goal Directed Processing

Most of the model's processing has been goal directed. In this section I would like to discuss three non-goal directed aspects (bottom-up processing, associative activation, and demons) and consider what effect they would have on the system.

When words are read by the model, the actual words are not controlled by any goal of the model but have an external source. Using PART-OF links, they could (theoretically) be amalgamated into groups, into clauses, and perhaps even higher up, often without reference to any top-down goal. Certain groups of words could even be critical parts of scripts and when they occur would strongly suggest that these scripts be activated. This sort of bottom-up processing is diametrically opposed to the strong top-down emphasis of the model.

The question arises: where would bottom-up information meet top-down information? The top-down / bottom-up point could be fixed almost anywhere. If the model were to expect an utterance, but had no further details, then the UTTERANCE |PEXPR would be sent an INTERPRET message, and no further strong top-down direction need ensue. If the model were to expect a particular kind of utterance (e.g. a "Because I ate the cheese." clause), on the other hand, it could send the |PEXPR BECAUSE-I-ATE-THE-CHEESE an INTERPRET message to process the utterance from its own specific viewpoint. Similarly, the bottom-up processing could amalgamate words into groups, into clauses, and even suggest the relevance of certain scripts or non-linguistic goals. These bottom-up traces, however, would have to be integrated into some sort of top-down interpretation of the world; and choosing which top-down |PEXPR to contact to do this integration is a problem I have not attempted, except to say the currently active execution instance. Nonetheless, top-down or bottom-up processing can gravitate toward one another as they like, a flexibility that is crucial for an intelligent system.

There are other places where non-goal directed processing must be taken into consideration. Whenever a |PEXPR is activated top-down, many closely connected |PEXPRs (perhaps |PEXPRs whose names are contained in patterns of the top-down |PEXPR) may be relevant. To discover which are relevant, (ASSOC ---) messages would have to be sent out to spread an associative activation to nearby |PEXPRs (this is similar to Rieger's (1974) idea of inference waves spreading out from concepts). In responding to an ASSOC, a target |PEXPR would decide whether or not it considered itself relevant (in which case it would communicate its desire to be processed (and in what way - EXECUTE, EXPECT, or whatever) to the top-down activation source) and also it would need to decide to whom (if anybody) to spread the activation.

There is probably a need, as well, for |PEXPRs that act essentially as demons (a la Charniak (1972)) in that they are created at one time for execution later when conditions are appropiate. Upon creation, such a demon would have an (ASSOC ---) pattern added to it, containing the activation conditions for the demon along with the message it is to handle when it is activated (all of this likely contained in some sort of procedural element in the pattern). In addition, at creation, pointers

(DEMON |PEXPR-i new-demon)

to the new demon would be added to all |PEXPR-i considered to be relevant to its eventual activation. When any of these |PEXPR-i happened to be called in top-down, they would associatively contact the demon "new-demon" (because of the DEMON pointer) which can then decide whether or not conditions are right for it to be activated top-down.

6.2.4 Comparing |PEXPRs

In much of the model's processing, to talk to or about a pattern expression has required somebody else to know its name. On many occasions it becomes important to be able to compare entire pattern expressions to one another by content rather than

merely by name. This happens, for example, during recognition or associative activation, in delineating ever more precise referents for concepts at the language level, and in finding contradictions and similarities between concepts as part of a non-directed conversation or debate.

This is a very difficult process when virtually no restrictions have been placed on the patterns that can go into a |PEXPR, when knowledge about a |PEXPR itself is mixed in with knowledge about the |PEXPR's referent, when patterns are varied in length and content, and when the presence of macro elements in patterns virtually eliminates any possibility of self-examinability or comparison with other patterns at the lexical level. These difficulties can be partially overcome by the clever use of things like meta patterns, but the real solution will not come until much more declarative realizations of information can be devised to replace the all too frequent occurrence of long, procedural patterns.

In a manner similar to MERLIN (Moore and Newell (1973)), the fundamental comparison technique might involve viewing one pattern expression as another. Thus, it would be possible to view a dog as an animal, or a cat as a rock, or a car as a freight train, etc. Direction of view is important since, for example, the opposite view of a freight train as car, a rock as a cat, an animal as a dog, is quite different.

The fundamental comparison rule outlined below is based on the intersection technique described in Quillian's TLC (1969). The rule goes as follows:

> Pattern expression A can be viewed as pattern expression B if
>
> (i) B = A or is in the ISA environment of A;
>
> or (ii) there is some "nearby" pattern expression C in the ISA environment of both A and B, and moreover each pattern in A has a comparable pattern in B. "Nearby" is determined by counting the number of ISA links traversed from each |PEXPR before intersection is achieved. The number of such links varies with the

importance of the viewing. Two patterns X and Y are <u>comparable</u> if for each atomic element E in X there is an atomic element F in Y such that E can be viewed as F.

These rules can be tuned, if desired, to reflect a particular set of assumptions, as the following examples illustrate. Trivially, if the |PEXPRs are RUFF and ANIMAL, then by "view-as" rule (i), RUFF can be viewed as an ANIMAL. Next, assume the |PEXPR SMOKEY (a CAT) is to be viewed as ROCK1 (a STONE). Thus, SMOKEY could be

```
(|PDEF SMOKEY
       (INSTANCE-OF SMOKEY CAT)
  S1 : (NOISE SMOKEY MEOW)
  S2 : (SHAPE SMOKEY FOUR-LEGGED)
       (CORE SMOKEY (/S1 /S2)))
```

and ROCK1 could be represented as

```
(|PDEF ROCK1
       (INSTANCE-OF ROCK1 STONE)
  S1 : (NOISE ROCK1 NOSOUND)
  S2 : (SHAPE ROCK1 ROUND)
       (CORE ROCK1 (/S1 /S2)))
```

So, the command (VIEW-AS SMOKEY ROCK1) is issued and the intersection technique carried out resulting in an intersection at THING. Assuming this is near enough (i.e. not more than a user specified number of ISA links have been followed), the search for comparable patterns begins. Comparing the respective S1 and S2 patterns, the following first-level viewpoint can be built:

```
(|PDEF SMOKEY-VIEWED-AS-A-ROCK1
       (SUPERSET SMOKEY-VIEWED-AS-A-ROCK1 VIEWPOINT)
  S1 : (NOISE !(VIEW SMOKEY ROCK1)
              !(VIEW MEOW NOSOUND))
  S2 : (SHAPE !(VIEW SMOKEY ROCK1)
              !(VIEW FOUR-LEGGED ROUND))
       (CORE SMOKEY-VIEWED-AS-A-ROCK1 (/S1 /S2)))
```

which can be reduced still further to

```
(|PDEF SMOKEY-VIEWED-AS-A-ROCK1
       (SUPERSET SMOKEY-VIEWED-AS-A-ROCK1 VIEWPOINT)
  S1 : (NOISE SMOKEY-VIEWED-AS-A-ROCK1
              !(VIEW MEOW NOSOUND))
  S2 : (SHAPE SMOKEY-VIEWED-AS-A-ROCK1
              !(VIEW FOUR-LEGGED ROUND))
       (CORE SMOKEY-VIEWED-AS-A-ROCK1 (/S1 /S2)))
```

since SMOKEY-VIEWED-AS-A-ROCK1 is the result of executing !(VIEW SMOKEY ROCK1). Thus, SMOKEY can be viewed as ROCK1 if a MEOW can be viewed as NOSOUND and FOUR-LEGGED can be viewed as

ROUND. Clearly, each of these computations could be carried out to get second-level viewpoints and so on.

Obviously much has been left unsaid. First, "nearby" has been defined in this example to be quite far apart since the intersection at THING is very high in the ISA hierarchy. The decision as to how many links to traverse would depend on exactly how crucial the comparison is in the current context.

Second, how did VIEW know that S1 and S2 were grist for the comparison mill while the INSTANCE-OF and CORE patterns were not? The answer lies in the CORE patterns mentioned in section 6.2.1. Recall that the meta pattern (CORE ---) defines the patterns which are central to a !PEXPR. Thus, one way to refine the fundamental comparison rule is to use only CORE patterns in the comparison. A further refinement of this might also take into account certain other meta information such as IMPs, etc., in deciding which patterns to compare and which to leave alone.

A third refinement illustrated by the example is the fact that the comparison was stopped at the first level of viewpoints rather than being carried out ad infinitum. The decision as to how far to go with this would be dependent (i) on the degree of difference between the resultant views (e.g. if MEOW and NOSOUND were to intersect at too great a distance, then further comparison might seem futile); (ii) on the importance of finding a very refined match in the current situation; and (iii) on the reluctance of the model to posit (a term from MERLIN) the necessary views rather than to produce a further level of viewpoints (e.g. without further search FOUR-LEGGED could be posited as ROUND). All these parameters could be specified as arguments at the time of VIEW invocation.

Many difficulties have not been illustrated by the example. First, there is the absence of procedural pattern elements. These might be handled by testing for LISP EQUALity; but this seems a bit rigourous and a more flexible means of comparison would be desirable. The eventual hoped-for reduction in the number of such procedural patterns would certainly go a long way

towards alleviating this problem.

A second problem is how to choose the B pattern to compare to the A pattern, something that is fairly easy if a B pattern starting with the same head and of the same length as the A pattern can be found, but something that is very difficult otherwise. If no comparable pattern is found for A, then presumably some sort of pattern (NOVIEW PATTERN A) would need to be asserted in the VIEWPOINT |PEXPR. Too many such unmatched patterns, especially if they were IMPs, would be grounds for rejecting the VIEWPOINT. Similarly, too many extra B patterns could be cause for VIEWPOINT rejection.

Obviously the work on |PEXPR comparison is in a preliminary stage. Many issues have been ignored and even the ones that have been discussed have been only briefly examined. If the model is to be extended to recognition and other more general areas, |PEXPR comparison will have to be confronted more seriously.

6.2.5 Miscellaneous Considerations

Clearly the system here ignores many interesting areas that could use further elaboration. One such area is short term memory. Is the short term memory limitation to 7 plus or minus 2 items a limitation on the number of patterns in a |PEXPR? Or, perhaps, the number of execution instances on the execute queue? Or is there a separate short term memory where certain concepts must be placed before being considered? I don't have any idea except that short term memory seems a fairly important concept and would probably be helpful in cutting down on the vast number of things being considered at any given time by the model.

Other problems are planning (that is actually building such |PEXPRs as ATTEND-CONCERT in response to goals); deciding amongst many goals; deciding on what goals to undertake in the first place. Once again I can at best speculate on these things.

Work on so-called "analogical" representations, such as

carried out by Funt (1976) and others, suggests that perhaps a place should be found for such a level of description. Adding a new kind of object (e.g. some sort of direct representation of the Queen Elizabeth theatre or a motor-action) could be accomplished quite easily with minimal disruption to the interpreter; but deciding on what message passing semantics to enforce for the new object would be difficult. In sum this would probably be more trouble than it was worth, given the flexibility and power of the current object definitions, particularly the pattern expression.

This concludes the description of the generalizations and extensions. In Chapter VII I conclude with a discussion of the contributions and future directions of this research.

CHAPTER VII
Conclusion

In the preceding chapters a scheme for representing knowledge was delineated and given computational description as a set of LISP-based control structures called |LISP. Then, an approach to modelling conversation was described and a particular example illustrating the approach was detailed in terms of |LISP. In this chapter I would like to summarize the contributions and shortcomings of the representation scheme and the language model; I would then like to conclude with some thoughts on how to improve and extend the ideas presented here.

7.1 Contributions to Representation

The object centred representation allows a modularity which should enable large systems to be built. The pattern expression in particular is interesting in that it comes in so many guises: it can be seen as an indivisible primitive or as a node in a semantic network or as an examinable set of patterns or as a basically procedural object. Often the same pattern expression can be viewed in any or all of the four guises, depending on circumstances.

A pattern expression is also interesting in how it responds to a message. The incoming message is a pattern that is matched against patterns in the |PEXPR body. The message pattern has exactly the same restrictions as the |PEXPR patterns so the pattern matching is completely symmetric. Through the use of macros, the pattern matcher can be given instructions that can involve binding variables, computing something, or performing an action only if certain conditions are met. These capabilities allow a decision regarding the exact meaning of a pattern to be postponed until the information in the pattern is needed. This is useful when encoding context dependent knowledge.

Because of macros, the user of the representation scheme also has the choice of whether to encode information in a basically procedural way (using lots of macros) or in a basically declarative way (using very few macros). In the early stages of any research it may be easier to throw much of a system's knowledge into procedural patterns at the expense of examinability, but this should disappear as familiarity with the domain increases and the natural dimensions of the domain manifest themselves. As these dimensions are discovered, the procedural patterns can be replaced by more declarative patterns. Moreover, the transition can be smooth since all patterns (macro-filled or macro-less) are accessed in a similar fashion, using the same pattern matcher. Further contributing to the procedural / declarative intermixture is the fact that the answers to all messages are themselves patterns (containing virtually no macros) that are left in the execution instance of the |PEXPR which received the message as declarative fallout of procedural capabilities.

One final feature of the pattern expression is its capability of being "run" in several different modes merely by endowing it with several different patterns, one for each mode. Thus, in the language example of Chapter V, speech acts could be run in EXECUTE, EXPECT, or ASSOC modes; language level |PEXPRs could be INTERPRETed or GENERATEd; etc. This capability allows all information relevant to an object to be easily incorporated within the object.

The "multiple mode" feature is even more useful when combined with the representation scheme's failure to match processing. If a particular message pattern cannot be handled, the pattern head is contacted for its advice. The user can specify that the pattern head is to perform arbitrary inferences, but this would in general be explosive, so he is encouraged (by the provision of special search routines) to restrict failure to match processing to execution environment searches (for context dependent information) or to ISA searches (for information that can be inherited down a generalization

hierarchy). This kind of failure to match capability provides a way of performing procedural attachment (see Winograd (1975)); ensures that information is inherited if and only if it is needed; and defines the pattern head (usually acting as a relation) as the foremost authority on what to do in failure to match situations.

The representation scheme is substantially enhanced because its execution instances are themselves !PEXPRs that don't disappear after execution unless explicitly garbage collected. This allows them to be accessed using the same message passing paradigm as any other !PEXPR, a nice bit of uniformity that turns out to be very useful in episodic considerations (see below). Because of this feature, the power that an ALINK / CLINK distinction provides (Bobrow and Wegbreit (1973)) to execute in one environment and return control to another can be simulated by having an execution instance of an execution instance, each with its own execution environment. In fact, the ability can be extended by having arbitrary chains of execution instances, all of which have different execution environments. Finally, keeping old execution instances around allows them to be used in pseudo-parallelism, where an old execution instance can be stopped and later resumed in the exact same context it was inhabiting before it was halted.

The execution environment of active execution instances is basically a LISP or ALGOL style dynamic context, but the fact that the execution instances in the environment are !PEXPRs that can contain patterns implies it can be used in a knowledge-based way. In particular, patterns that contain context dependent information (e.g. the identity of the conversants, the purpose of one of these conversants, the current focus of attention) can be stored and accessed. Once control returns from an execution instance containing such patterns, they disappear from view. The active execution environment thus provides a focussing mechanism. It is also useful if the current goals of a system need to be accessed - the goals are strung upwards in the execution environment, attached (via EX-INSTANCE-OF pointers) to

the execution instances there.

But, perhaps the most interesting feature of execution environments arises in combination with the examinability and non-disappearance of the execution instances contained there. This combination of features allows the construction of an episodic memory (Schank (1974), Tulving (1972)) consisting of trees of old execution environments. If any particular execution instance in this environment is picked out, the entire context in which it was originally active can be seen. Although the mechanics of how much of old execution environments to save and how to search them have not been worked out, this seems to be a promising approach to episodic memory and one that is perfectly compatible with semantic memory (in contrast to Schank's (1974) arguments) since all execution instances have EX-INSTANCE-OF pointers to the "semantic" ISA environment as well as EX-ENVIRON pointers to the "episodic" execution environment.

A final aspect of the representation scheme is the fact that it allows arbitrary levels of detail along several dimensions (ISA, PART-OF, the subgoals created by message passing). In addition there is an implicit "containment" hierarchy stretching beneath a |PEXPR consisting of the |PEXPRs referred to in its patterns, the |PEXPRs referred to in those |PEXPRs' patterns, and so on. This ability means that a |PEXPR can always be broken down if necessary into subcomponents of some kind, in direct opposition to the primary tenet of primitive based systems (Schank (1972), Wilks (1973)). The ability to go into detail when necessary seems an essential component of intelligence, a point that is further argued by Rowat (1974).

7.2 Contributions to Language Analysis

The language model described in Chapter V has served a dual role: first, it has acted as a test bed for |LISP; second it has suggested an approach to modelling conversation. I would like

to describe the main contributions that the model makes to the analysis of conversation and also to suggest how |LISP has helped and influenced the construction of the model.

The model is founded on the principle that the problem of conversation must be treated as essentially a problem in pragmatics. This viewpoint is not unique - it is argued persuasively by Winograd (1976), by some philosophers of language (Grice (1968), Searle (1969)), and by linguists such as Fillmore (1975) - but it differs markedly from traditional linguistic approaches. The view is reflected in a couple of ways in the model: the indistinguishability of linguistic from non-linguistic goals (they are all |PEXPRs, and except at the language level all receive EXECUTE and EXPECT messages); and the emphasis in the model on scripts, speech acts, and conversant models rather than on more standard language aspects such as parsing, parts of speech, etc.

The model's emphasis on goals is not accidental. Other researchers (P. Cohen (1978), Levin and Moore (1977), philosophers of language) have suggested the importance of conversants' goals in the interpretation and production of utterances, although the rather rigid hierarchical structure of non-linguistic goals calling scripts calling speech acts calling language level goals is not emulated elsewhere. Aside from this, the most interesting general features distinguishing the model are |LISP-based: the ability to encode all goals as |PEXPRs and run them in EXECUTE mode (to produce utterances) or EXPECT mode (to understand utterances); the fact that from any goal all superior goals can be accessed in the execution environment and that the execution environment acts as a focus for all linguistic endeavour; the usefulness of being able to abstract processes into the ISA hierarchy and later inherit them when necessary; the fact that execution instances at all levels can act as repositories of information and in that way encode the "meaning" of the conversation.

However, at each level there are interesting features. Non-linguistic goals drive other goals and form the context for

all that follows. They can be accessed from all lower levels and are thus influential on what is said. Occasionally non-linguistic goals can be called by linguistic goals (e.g. EXCHANGE calls GIVE in the Chapter V examples) as well as vice versa. This flexibility is crucial in many linguistic situations (especially when language is being used to aid in the accomplishment of some task).

The script level is interesting in the model because it directs the entire conversation. Scripts are responsible for spawning sub-scripts or speech acts, making sure they have run correctly (although the model has a trivial "explanation" component at the present time), and in general keeping track of the sequencing of a conversation. Scripts also keep a record of the entire conversation by making assertions in the execution instances (this, too, is pretty trivial at the moment). Of particular note is the flexibility of scripts in regard to the identities of the speaker: the model itself can be identified with none, one, or both of the conversants. There is also no theoretical limit in the model to having only two conversants (although no n-person conversations were given).

The speech act level is not nearly so central in the model here as it is in, say, P. Cohen's (1978) work. Speech acts, nonetheless, form an interesting interface between scripts and the language level. Perhaps speech acts are most interesting here in that they take an active role in carrying out the act of speaking just as would a motor action when carrying out some sort of physical act. Because of the "multiple mode" aspect of |LISP, speech acts can be asked to either EXPECT (understand) or to EXECUTE (produce) a speech act. In EXPECT mode a speech act must read an utterance and interpret it in any way it sees fit (perhaps by calling in the language level); in EXECUTE mode a speech act must generate a surface utterance in any way it sees fit (perhaps with the help of the language level) and then print it.

The language level is the least developed of the model, but does have a couple of interesting features. It's syntactic

component, insofar as it can be distinguished, is more or less a top-down parser that groups words for its case-like semantic component. The combination of the two approaches isn't too common (but see Taylor and Rosenberg (1975) for one example) and seems to be a workable hybrid. The general ability of |LISP objects and message passing to accomodate successfully to this level is gratifying. Of particular effectiveness is the conceptualizing of a noun group or verb group as an execution instance with an EX-ENVIRON pointer to the context in which the words were uttered, an EX-INSTANCE-OF pointer to the linguistic object (NOUNG or VERBG) of which they are an instantiation, and a ROLE-INSTANCE-OF pointer to the concept which they represent. This allows information about the context of the word group, the word group itself, and the concept represented by the word group all to be accessed.

The conversant models in the model are not sophisticated at the present time, the main idea having been to find out how to connect conversant models into all the other processing rather than to build an intricate web of conversant knowledge. Scripts and speech acts have directly available to them conversant models for all conversants, whereas language level goals do not. Of course, for any subgoal of a script or speech act, the conversant models are indirectly available from the script or speech act. Conversant models have at least two interesting aspects: first, the use of ROLE-INSTANCE-OF patterns (often with context dependent macros embedded in them) allow conversants to be viewed in particular roles (e.g. seller, buyer) as well as in their standard "person" role; second, the existence of a conversant model for the model itself. Both these aspects are important for the completeness and symmetry of the processing.

7.3 Future Directions

There is, of course, much still to be done. The first step is to implement |LISP and then see how well it works by running through the currently pseudo-implemented ticket buying and drink buying conversations. This would doubtless raise several issues, most importantly the efficiency of execution. Failure to match processing and IF-ADDED and IF-REMOVED methods may be computationally expensive unless controlled with draconic prudence. Also arising here would be the problem of explaining the failure of subgoals in a much more serious way, a step which would become even more important in less directed conversations.

After running through the ticket buying and drink buying conversations, the next step would indeed be to try out less task-oriented dialogues such as the conversation with a friend. Handling such conversations would bring up many of the non-goal-directed issues raised in the last chapter. It seems clear that a much larger role for bottom-up processing and, if some version of it can be made combinatorially sound, associative activation is crucial if the model is to be extended to handle more fluid conversations. The need for |PEXPR comparison would also arise.

Crucial in conversation modelling would be a working episodic capability so that references to previous utterances wouldn't leave the model perplexed. First, an exploration of when and how to search episodic memory would have to be carried out. But, more important would be the need to figure out which old episodes to "remember" and which ones to "forget" (by freeing them for garbage collection). As Schank and Abelson (1975) have pointed out, and as the proposals presented here illustrate well, a theory of forgetting promises to be a central concern of much future work in artificial intelligence.

Because the current model of conversation and its |LISP base have been designed with generality and extensibility in mind, their further development along these lines seems to be possible. The research undertaken so far should provide a solid foundation for the endeavours to come.

BIBLIOGRAPHY

Abelson (1973). R. P. Abelson, "The Structure of Belief Systems", in R. C. Schank, K. M. Colby (eds.), Computer Models of Thought and Language, Freeman, San Francisco, 1973.

Austin (1962). J. L. Austin, How to do Things with Words, Oxford University Press, Oxford, England, 1962.

Becker (1969). J. D. Becker, "The Modelling of Simple Analogic and Deductive Processes in a Semantic Memory System", Proc. IJCAI1, Washington, D.C., May 1969, pp. 655-688.

Bloomfield (1933). L. Bloomfield, Language, Holt, Rinehart, New York, 1933.

Bobrow and Collins (1975). D. G. Bobrow and A. Collins (eds.), Representation and Understanding, Academic Press, New York, 1975.

Bobrow and Wegbreit (1973). D. G. Bobrow, B. Wegbreit, "A Model and Stack Implementation of Multiple Environments", CACM, 16, 10, October 1973, pp. 591-602.

Bobrow and Winograd (1976). D. G. Bobrow and T. Winograd, An Overview of KRL, a KnowledgeRepresentation Language, Tech. Rpt. CSL-76-4, Xerox Palo Alto Research Center, Palo Alto, Calif., 1976.

Bruce (1972). B. Bruce, "A Model for Temporal References and Its Application in a Question Answering Program", Artificial Intelligence, 3, 1972.

Bruce (1975). B. Bruce, Belief Systems and Language Understanding, BBN Rept. #2973, Bolt, Beranek, and Newman, Inc., Cambridge, Mass., 1975.

Bullwinkle (1977). C. Bullwinkle, "Levels of Complexity in Discourse for Anaphora Disambiguation and Speech Act Interpretation", Proc. IJCAI5, Cambridge, Mass., 1977, pp. 43-49.

Chafe (1970). W. Chafe, Meaning and the Structure of Language, U. of Chicago Press, Chicago, Illinois, 1970.

Chafe (1975). W. Chafe, "Some Thoughts on Schemata", Proc. TINLAP, Cambridge, Mass., June 1975, pp. 99-101.

Charniak (1972). E. Charniak, Toward a Model of Children's Story Comprehension, MAC-TR-51, MIT AI Lab., Cambridge, Mass., December 1972.

Charniak (1975). E. Charniak, "Organization and Inference in a Frame-like System of Common Knowledge", Proc. TINLAP, Cambridge, Mass., June 1975, pp. 46-55.

Chomsky (1957). N. Chomsky, Syntactic Structures, Mouton, The Hague, 1957.

Chomsky (1971). N. Chomsky, "Deep Structure, Surface Structure, and Semantic Interpretation", in Steinberg and Jakobovits (1971).

P. Cohen (1978). P. Cohen, On Knowing What to Say: Planning Speech Acts, Ph.D. Thesis, Dept. of Computer Science, U. of Toronto, Toronto, Ontario, 1978.

R. Cohen (1976). R. Cohen, Computer Analysis of Temporal

Reference, M.Sc. Thesis, Dept. of Computer Science, U. of Toronto, Toronto, Ontario, Dec. 1976.

Collins and Grignetti (1975). A. Collins, M. C. Grignetti, Intelligent CAI, BBN Report 3181, Bolt, Beranek, and Newman, Inc., Cambridge, Mass., October, 1975.

Deutsch (1974). B. Deutsch, "The Structure of Task Oriented Dialogues", IEEE Symposium on Speech Recognition, Carnegie-Mellon University, Pittsburgh, Penn., April 1974.

Fahlman (1975). S. Fahlman, A System for Representing and Using Real World Knowledge, MIT AI Memo 331, Cambridge, Mass., May 1975.

Feigenbaum and Feldman (1963). E. A. Feigenbaum, J. Feldman (eds.), Computers and Thought, McGraw-Hill, New York, 1963.

Fillmore (1968). C. Fillmore, "The Case for Case", in E. Bach, R. Harms (eds.), Universals in Linguistic Theory, Holt, Rinehart, and Winston, New York, 1968, pp. 1-88.

Fillmore (1975). C. Fillmore, "An Alternative to Checklist Theories of Meaning", Proc. First Annual Meeting of the Berkeley Linguistics Society, Berkeley, Calif., Feb. 1975, pp. 123-131.

Funt (1976). B. V. Funt, WHISPER: A Computer Implementation Using Analogues in Reasoning, Ph.D. Thesis, Dept. of Computer Science, UBC, Vancouver, B.C., March 1976.

Garfinkel (1972). H. Garfinkel, "Studies of the Routine Grounds of Everyday Activities", in D. Sudnow (ed.), Studies in Social Interaction, The Free Press, New York, 1972, pp. 1-30.

Goffman (1974). E. Goffman, Frame Analysis, Harper Colophon, New York, 1974.

Green et al (1963). B. F. Green, A. C. Wolf, C. Chomsky, K. Laughery, "BASEBALL: An Automatic Question Answerer", in Feigenbaum and Feldman (1963), pp. 207-216.

Grice (1957). H. P. Grice, "Meaning", Philosophical Review, July 1957, pp. 377-388.

Grice (1968). H. P. Grice, Logic and Conversation, Wm. James Lecture, Unpublished Mimeo, Berkeley, California, 1968.

Grosz (1977). B. Grosz, "The Representation and Use of Focus in a System for Understanding Dialogs", Proc. IJCAI5, Cambridge, Mass., 1977, pp. 67-76.

Havens (1978). W. S. Havens, A Computational Model for Frame Systems, Ph.D. Thesis, Dept. of Computer Science, UBC, Vancouver, B.C., 1978 (in preparation).

Hendrix (1975). G. Hendrix, "Expanding the Utility of Semantic Networks through Partitioning", Proc. IJCAI4, Tbilisi, USSR, Sept. 1975, pp. 115-121.

Hewitt (1972). C. Hewitt, Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot, MIT AI Memo 251, Cambridge, Mass., April 1972.

Hewitt (1975). C. Hewitt, "Stereotypes as an ACTOR Approach Towards Solving the Problem of Procedural Attachment in FRAME Theories", Proc. TINLAP, Cambridge, Mass., June

1975, pp. 108-117.

Hewitt et al (1973). C. Hewitt, P. Bishop, R. Steiger, "A Universal Modular Actor Formalism for Artificial Intelligence", Proc. IJCAI3, Stanford, Calif., Aug. 1973, pp. 235-245.

Hewitt and Greif (1974). C. Hewitt, I. Greif, "Actor Semantics of PLANNER-73", MIT AI Lab. Working Paper 81, Cambridge, Mass., November, 1974.

Horrigan (1977). M. K. Horrigan, Modelling Simple Dialogs, M.Sc. Thesis, Dept. of Computer Science, U. of Toronto, Toronto, Ontario, Jan. 1977.

Hurtubise (1976). S. J. Hurtubise, "A Model and Stack Implementation of a Conversation Between Some Man and a Smart-Aleck Computer", Third CSCSI / SCEIO Newsletter, August 1976, pp. 46-50.

G. Lakoff (1971). G. Lakoff, "On Generative Semantics", in Steinberg and Jakobovits (1971).

Lakoff and Ross (1972). G. Lakoff, J. R. Ross, "A Note on Anaphoric Islands and Causatives", Ling. Inq., 3, 1, Summer 1972, pp. 121-127.

R. Lakoff (1973). R. Lakoff, "The Logic of Politeness; or Minding your P's and Q's", Proc. CLS9, 1973, pp. 292-305.

Leech (1974). G. Leech, Semantics, Penguin Books, England, 1974.

Levesque (1977). H. Levesque, A Procedural Approach to Semantic Networks, M.Sc. Thesis, Dept. of Computer Science, U. of Toronto, Toronto, Ontario, Jan. 1977.

Levin and Moore (1977). J. A. Levin, J. Moore, Dialogue Games: Meta-Communication Structures for Natural Language Interaction, ISI/RR-77-53, Information Sciences Institute, U. of Southern California, Marina del Rey, Calif., Jan. 1977.

Linde (1974). C. Linde, "Information Structures in Discourse", NWAVE III, Georgetown University, Oct. 1974.

Lindsay (1963). R. K. Lindsay, "Inferential Memory as the Basis of Machines which Understand Natural Language", in Feigenbaum and Feldman (1963), pp. 217-233.

Martin (1975). W. A. Martin, Conceptual Grammar, Automatic Programming Group Internal Memo 20, Project MAC, MIT, Cambridge, Mass., 1975.

McCalla (1973). G. I. McCalla, A Model for Man - Machine Dialogue, Ph.D. Thesis Progress Report, Dept. of Computer Science, UBC, Vancouver, B.C., Nov. 1973.

McDermott and Sussman (1974). D. V. McDermott, G. J. Sussman, The CONNIVER Reference Manual, MIT AI Memo 259a, Cambridge, Mass., Jan. 1974.

Minsky (1974). M. Minsky, A Framework for Representing Knowledge, MIT AI Memo 306, Cambridge, Mass., June 1974.

Moore and Newell (1973). J. Moore, A. Newell, How Can MERLIN Understand?, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Penn., Nov. 1973.

Newell (1973). A. Newell, "Production Systems: Models of

Control Structures", in W. G. Chase (ed.), Visual Information Processing, Academic Press, New York, 1973, pp. 403-426.

Quillian (1969). M. R. Quillian, "The Teachable Language Comprehender: A Simulation Program and Theory of Language", CACM, 12, 8, August 1969, pp. 459-476.

Rieger (1974). C. J. Rieger, Conceptual Memory: A Theory and Computer Program for Processing the Meaning Content of Natural Language Utterances, Ph.D. Thesis, Stanford University, Stanford, Calif., 1974.

Roussopoulos (1976). N. Roussopolous, A Semantic Network Model of Data Bases, Ph.D. Thesis, Dept. of Computer Science, U. of Toronto, Toronto, Ontario, November 1976.

Rowat (1974). P. F. Rowat, Designing a Robot Controller: A Report on My Thesis Research, Ph.D. Thesis Progress Report, Dept. of Computer Science, UBC, Vancouver, B.C., March 1974.

Sandewall (1975). E. Sandewall, "Ideas About the Management of LISP Data Bases", Proc. IJCAI4, Tbilisi, USSR, Sept. 1975, pp. 585-592.

Schank (1972). R. C. Schank, "Conceptual Dependency: A Theory of Natural Language Understanding", J. Cog. Psych., 3, 4, October 1972, pp. 552-631.

Schank (1974). R. C. Schank, Is There a Semantic Memory?, Report #3, Istituto per gli Studi Semantici e Cognitivi, Castagnola, Switzerland, 1974.

Schank (1975). R. C. Schank, "The Primitive ACTs of Conceptual Dependency", Proc. TINLAP, Cambridge, Mass., June 1975, pp. 38-41.

Schank and Abelson (1975). R. C. Schank, R. P. Abelson, "Scripts, Plans, and Knowledge", Proc. IJCAI4, Tbilisi, USSR, Sept. 1975, pp. 151-157.

Schegloff (1971). E. A. Schegloff, "Sequencing in Conversational Openings", in J. A. Fishman (ed.), Advances in the Sociology of Language, Vol. II, Mouton, The Hague, 1971, pp. 91-125.

Schneider (1978). P. F. Schneider, Organization of Knowledge in a Procedural Semantic Network Formalism, M. Sc. Thesis, Department of Computer Science, U. of Toronto, Toronto, Ontario, January 1978.

Schwarcz et al (1970). R. M. Schwarcz, J. F. Burger, R. F. Simmons, "A Deductive Question Answerer for Natural Language Inference", CACM, 13, 3, March 1970, pp. 167-183.

Searle (1969). J. R. Searle, Speech Acts, An Essay in the Philosophy of Language, Cambridge University Press, Cambridge, England, 1969.

Shortliffe (1976). E. H. Shortliffe, Computer-Based Medical Consultations: MYCIN, American-Elsevier, New York, 1976.

Steinberg and Jakobovits (1971). D. D. Steinberg, L. A. Jakobovits (eds.), Semantics, an Interdisciplinary Reader, U. of Illinois Press, Urbana, Illinois, 1971.

Sussman (1973). G. J. Sussman, A Computational Model of Skill Acquisition, MIT AI TR-297, Cambridge, Mass., August 1973.

Sussman and Winograd (1970). G. J. Sussman, T. Winograd, Micro-Planner Reference Manual, MIT AI Memo 203, Cambridge, Mass., July 1970.

Taylor and Rosenberg (1975). B. H. Taylor, R. S. Rosenberg, "A Case Driven Parser for Natural Language", AJCL, Microfiche 31, 1975.

Tesler et al (1969). L. Tesler, H. Enea, K. M. Colby, "A Directed Graph Representation for Computer Simulation of Belief Systems", Mathematical Biosciences, 2, 1968, pp. 19-40.

Tulving (1972). E. Tulving, "Episodic and Semantic Memory", in E. Tulving and W. Donaldson (eds.), Organizatich of Memory, Academic Press, New York, 1972.

Wilks (1973). Y. Wilks, "Understanding Without Proofs", Proc. IJCAI3, Stanford, Calif., August 1973, pp. 270-277.

Winograd (1972). T. Winograd, Understanding Natural Language, Academic Press, New York, 1972.

Winograd (1974). T. Winograd, "Breaking the Complexity Barrier Again", SIGPLAN Notices, January 1974.

Winograd (1975). T. Winograd, "Frames and the Procedural-Declarative Controversy", in Bobrow and Collins (1975), pp. 185-210.

Winograd (1976). T. Winograd, "Towards a Procedural Understanding of Semantics", Revue Internationale de Philosphie, 1976.

Winograd (1977). T. Winograd, A Framework for Understanding Discourse, to appear as a SAIL Memo, AI Lab., Stanford University, Stanford, Calif., 1977.

Winston (1970). P. H. Winston, Learning Structural Descriptions from Examples, MAC TR-76, MIT, Cambridge, Mass., 1970.

Wong (1975). H. K. T. Wong, Generating English Sentences from Semantic Structures, M.Sc. Thesis, Dept. of Computer Science, U. of Toronto, Toronto, Ontario, 1975.

Woods (1975). W. A. Woods, "What's in a Link: Foundations for Semantic Networks", in Bobrow and Collins (1975).

Woods et al (1972). W. A. Woods, R. M. Kaplan, B. Nash-Webber, The Lunar Sciences Natural Language Information System: Final Report, BBN Report Number 2388, Bolt, Beranek, and Newman, Inc., Cambridge, Mass., June 1972.

Abbreviations:

AJCL: American Journal of Computational Linguistics
CACM: Communications of the ACM
CLS: Chicago Linguistics Society
IJCAI: International Joint Conference on AI
JACM: Journal of the ACM
J. Cog. Psych.: Journal of Cognitive Psychology
SIGPLAN: ACM Special Interest Group on Programming Languages
TINLAP: Conference on Theoretical Issues in Natural Language Processing

APPENDIX I

Some System Objects

In this Appendix several system objects are presented in terms of their input / output behaviour and also, for certain important objects, in terms of their logic flow. The Appendix forms a handy reference, especially when reading the detailed examples of Chapter V.

The following notational conventions have been used throughout the Appendix: Ai is an atom (i.e. a single name); Si is an s-expression; Li is a list; Pi is a pattern; Ni is a number; NIL is the null list () which stands for "false" or failure to match depending on the context; |U is a special NIL-like atom which stands for "un-initialized" and is used to assign a value to certain pattern matching macros before they are bound in the match; T is the "true" atom; EXPR indicates an object of type EXPR and has three possible sub-classifications: LAMBDA, NLAMBDA, or FLAMBDA; |EXPR indicates an object of type |EXPR and has three possible sub-classifications: |LAMBDA, |NLAMBDA, or |FLAMBDA. In addition to these standard assumptions, there are many special forms whose names are given meaningful mnemonics.

A1.1 Basic Interpreter Objects

(|EVAL message-form)	EXPR (LAMBDA)

|EVAL is the |LISP interpreter. Its basic logic follows:

Call in (|UPDATE-TIMER) before proceeding.

Then, if message-form is an atom, its value on the current execution instance's stack is returned; if none is

found, an error is generated.

If the CAR of message-form is an atom, then

-if it is QUOTE, message-form is returned;

-if it is the name of a |PEXPR, then (|SCHEDULE message-form) and resume execution of scheduler;

-if it is the name of a |EXPR, then construct the appropriate |LAMBDA expression, returning result of |EVALing this |LAMBDA expression with CDR of message-form as arguments;

-similarly, if it is the name of an EXPR, construct the appropriate LAMBDA expression returning the result of |EVALing this LAMBDA expression with the CDR of message-form as arguments;

-if it is the name of a SUBR, APPLY CAR of message-form to |EVLIS of the CDR of message-form and return the resulting value;

-if it is anything else, there is an error.

If the CAR of message-form is a list and the CAAR is

-|LAMBDA, then bind the |LAMBDA parameters to |EVLIS of the arguments (using the current |PEXPR stack); |EVAL the body of the |LAMBDA; return the result;

-|NLAMBDA, then bind the |NLAMBDA parameters to the arguments (not |EVALed) (using the current |PEXPR stack); |EVAL the body of the |NLAMBDA; return the result;

-|FLAMBDA, then bind the |FLAMBDA parameter to a list of the arguments (not |EVALed); |EVAL the body of the |FLAMBDA; return the result;

-LAMBDA, then APPLY the LAMBDA expression to |EVLIS of the arguments and return the result;

-NLAMBDA or FLAMBDA, then APPLY the NLAMBDA or FLAMBDA expression to the arguments (neither |EVALed nor EVALed) and return the result.

If message-form is anything else, then there is an error.

(|SCHEDULE message-form) EXPR (LAMBDA)

If message-form is not a list or its CAR is not the name of a |PEXPR, then there is an error. Otherwise, let CAR of message-form be known as the "receiving-object". |SCHEDULE first of all creates a new execution instance, "new-object", of receiving-object in which will be asserted

(EX-INSTANCE-OF new-object receiving-object)
(EX-ENVIRON new-object current-ex-instance)
(STACK new-object ())

Then, message-form is checked to see if there are any special messages:

(|PRIO= priority): if found, strip it from message-form and |ASSERT (|PRIO= new-object priority) in new-object; if not found, |ASSERT (|PRIO= new-object 5);

(|TIME= time-limit): if found, strip it from message-form and |ASSERT (|TIME-LIM= new-object time-limit) and (|TIME-NOW= new-object time-limit) in new-object;

(|COND= pattern): if found, strip it from message-form and |ASSERT in new-object (|COND= new-object pattern);

(|END= end-limit): if found, strip it from message-form and |ASSERT in new-object (|END= new-object end-limit).

After completing the search for special messages, the (reduced) message-form is embedded in a call to |PEXPR-MH, the standard |PEXPR message handler. Thus, (|PEXPR-MH message-form) is pushed onto the stack of new-object under the indicator |EV indicating that it is the top |EVAL block on the stack of new-object. A list (new-object priority) is then merged into the execute queue according to its priority.

Finally, |SCHEDULE returns new-object.

(|UPDATE-TIMER) EXPR (LAMBDA)

Every 100 times |EVAL is called, |UPDATE-TIMER reduces all |TIME-NOW= patterns in the current execution environment by 1. If any |TIME-NOW= patterns are at 0, then call

(|INTERRUPT-PROCESSING '|TIME= 0 ex-instance-int)

where "ex-instance-int" is the execution instance in which the time violation occurred.

(|INTERRUPT-PROCESSING type-int val-int ex-instance-int) EXPR (LAMBDA)

This object is called in when an interrupt of type-int (either |TIME= or |COND=) has been detected in ex-instance-int. val-int is either 0 in the case of a |TIME= interrupt or, in the case of a |COND= interrupt, the pattern which matched the |COND= interrupt pattern.

|INTERRUPT-PROCESSING first looks into the execution environment of the current execution instance for the nearest semaphore (if there is none, assume the semaphore is "OFF"). If it is "ON", return NIL; else, reduce |END-LIM= pattern in ex-instance-int by 1. If the |END-LIM= pattern is 0, then no interrupts are left, so set up a return condition in the EX-ENVIRON execution instance of ex-instance-int to be

(|RETURN-COND ex-environ-inst
  |END= (ex-instance-int ex-instance-current) val-int)

If it is not 0, then set up a return condition in ex-environ-inst to be

(|RETURN-COND ex-environ-inst
  type-int (ex-instance-int ex-instance-current) val-int)

If type-int is |TIME= then reset |TIME-NOW= pattern to be the same as |TIME-LIM=.

Merge ex-environ-inst into execute queue with its stored

priority and resume execution of the scheduler.

(|SCHEDULER)	EXPR (LAMBDA)

This object removes first object / priority pair from the execute queue, sets the current execution instance to be this object, takes the top |EVAL block from the stack of the object, and |EVALs the form there.

(|PEXPR-MH message-form)	EXPR (LAMBDA)

|PEXPR-MH is the usual message handler for |PEXPRs. The receiving object is the first element of message-form; the rest of the elements of message-form are patterns to be matched in the body of receiving-object. For each such mess-pattern, |PEXPR-MH calls

(|MATCH mess-pattern ex-environ-inst
receiving-pattern current-inst)

for all receiving-patterns in the current execution instance until a match is found. If no such match is discovered, failure to match processing ensues; i.e. a message

(FAILURE-TO-MATCH mess-head mess-pattern
current-inst ?matching-pattern)

is sent to mess-head, the first element of mess-pattern). If this still fails to find a matching pattern (i.e. matching-pattern is NIL), then NIL is concatenated to an answer list. If a matching pattern is found, the result of |MATCH is concatenated to answer list and |ASSERTed in the execution instance.

When all message patterns have been handled, a return condition is set in ex-environ-inst

(|RETURN-COND ex-environ-inst

NORMAL (current-inst) answer-list)

and its stack is modified so that answer-list will also be the value returned from the message sent to receiving-object. ex-environ-inst is then merged into the execute queue with its stored priority and the scheduler is resumed.

A1.2 Redefined LISP SUBRs

Many LISP functions can be used directly as objects in |LISP. This is not the case for LISP FSUBRs or NSUBRs which must be redefined so that their arguments can be |EVALed rahter than EVALed. In addition certain SUBRs which depend on EVAL for their meaning (e.g. EVAL, APPLY, APPLY1, ...) or which ordinarily use the LISP stack (e.g. SET, SETQ, ...) or which use the property list features of LISP (e.g. PUT, GET, ...) must also be redefined. The rest of the SUBRs (e.g. CAR, CDR, CONS, EQ, ATOM, ...) can be used as is, since the |LISP interpreter automatically |EVALs their arguments.

The following are rewritten versions of the corresponding LISP functions (all are EXPRs):

(|EVAL S1): see section A1.1;

(|EVLIS S1 ... Sn): like EVLIS, but |EVALs its arguments;

(|APPLY FN S1 ... Sn): the same as APPLY except |EVAL used;

(|APPLY1 FN S1 ... Sn): the same as APPLY1 except |EVAL used;

(|SET A1 S1 ... An Sn): uses current execution instance's stack;

(|SETQ A1 S1 ... An Sn): uses current execution instance's stack;

(|UNEVAL A1 <S1>): uses current execution instance's stack;

(|MAP FN L1): uses |EVAL rather than EVAL; similarly

the other MAPping functions;

(|COND (S11 ... S1n) ... (Sk1 ... Skm)): uses |EVAL rather than EVAL;

(|AND S1 ...Sn): uses |EVAL rather than EVAL;

(|OR S1 ...Sn): uses |EVAL rather than EVAL;

(|PROG L1 S1 ... Sn): works like PROG only binds on the current execution instance and |EVALs the Si;

(|GO A1): works like GO except goes to label inside of |PROG or |EVENT-SEQUENCE and argument is |EVALed;

(|RETURN A1 <A2>): works like RETURN but uses current execution instance stack. If A2 is the keyword |HANDLER, then control will resume in the matcher of the message handler of the current execution instance with the value of the current pattern element being A1. Thus, for example, if there were a pattern

(X Y !(|PROG () (|RETURN 'DOG |HANDLER)) Z)

in the |PEXPR FOO, and if the message (X Y ?VAL Z) were sent to FOO, then at the point the |RETURN was executed, control would resume in the message handler of FOO's execution instance with the value of the third element being DOG and the match continuing to the fourth element. To force failure of the match, |RETURN NIL to the |HANDLER.

A1.3 Objects Which Create Objects

(|DEFUN A1 <A2> L1 S1 ... Sn)	EXPR (FLAMBDA)

This object works like LISP's DEFUN except that in addition to EXPRs, |EXPRs can be defined by using the indicator |EXPR, |NEXPR, or |FEXPR for A2 (default |EXPR). Note that A1 is globally defined as a |EXPR, |NEXPR, or whatever, since the LISP atom is modified. It seems

unnecessary to me to have a procedure defining capability that defines procedures only within a given |PEXPR, since it is usually the case that a procedure name is something that remains fixed throughout the system (especially true here since EXPRs and |EXPRs are defined to be objects known system wide). The situation where locality is important can be adequately handled by either renaming the procedure or by using "!" or "$" macros in patterns of a |PEXPR.

(|PDEF A1 P1 ... Pn)	EXPR (FLAMBDA)

This object creates A1 as a pattern expression. Each pattern Pi is either of the form "name-i : (...)" Or simply "(...)". These patterns can contain sub-patterns of either type as well, so that

(DOG CHASE DESC : (CAR COLOUR RED))

would work perfectly well (without, I should add, the DESC sub-pattern being added as a top-level pattern, but being labelled nonetheless). Using |ASSERTLIS, each pattern Pi is asserted in A1 (with appropriate IF-ADDED checks - see below), resulting in a string of patterns being attached to the LISP property list of A1 under the attribute |PEXPR.

(|CREATE-NEW A1)	EXPR (LAMBDA)

This object will create a new instance of A1 with appropriate (INSTANCE-OF new-inst A1) pattern being asserted in the new instance along with a (CREATION-ENVIRON new-inst ex-instance-current) pattern. The name of the new instance is returned.

A1.4 Objects for Communicating with Objects

| (|DYNAMIC A1 mess-1 ... mess-n) | |EXPR (|LAMBDA)|

This will send a set of messages to A1 and return the appropriate answer list. It is used when the receiving object or messages need to be |EVALed.

| (|RETURN-COND) | EXPR (LAMBDA) |

This object will return as value the return condition of the current execution instance. The possibilities:

-(|RETURN-COND ex-inst-current
NORMAL (ex-inst-receiving) value)

-(|RETURN-COND ex-inst-current
AU-REVOIR (ex-inst-receiving restart-name) value)

-(|RETURN-COND ex-inst-current
|TIME= (ex-inst-interrupt ex-inst-executing) value)

-(|RETURN-COND ex-inst-current
|COND= (ex-inst-interrupt ex-inst-executing) value)

-(|RETURN-COND ex-inst-current
|END= (ex-inst-interrupt ex-inst-executing) value)

The following can be used to access particular parts of the return condition:

(|RETURN-TYPE): will return third element of return condition;

(|RETURN-CODES): will return fourth element of return condition;

(|RETURN-VALUE): will return fifth element of return condition;

A1.5 Objects Involved in Matching

(|MATCH source-pat source-obj target-pat target-obj)	|EXPR (|FLAMBDA)

Of the "arguments" to this object, source-pat and target-pat are not |EVALed; source-obj and target-obj are. source-pat is matched against target-pat, with macros of source-pat elements being expanded in the context of source-obj while those of target-pat elements are expanded in the context of target-obj. Returned is either NIL (if there is no match) or the matching pattern (if there is a match). The basic logic flow of |MATCH follows:

If source-pat is not the same length as target-pat, the match fails so return NIL. Otherwise, set answer-pattern to NIL and compare each pair of source / target elements until all are handled (in which case return answer-pattern) or until comparison fails for some pair (in which case return NIL). The comparison for two elements:

1. if a macro precedes both elements, do macro-conflict processing (see macro-conflict table in Appendix II) and proceed; else if a macro precedes one element, expand it and replace the element by the result returned from the macro expansion (macros use the appropriate source-obj or target-obj for binding and |EVALuation if necessary);

2. if either element is NIL, the element comparison fails;

3. if both elements are atoms, then element comparison succeeds if they are the same atom in which case append the atom to answer-pattern and get the next source / target pair; else element comparison fails;

4. if both elements are lists, then
(|MATCH source-el source-obj target-el target-obj)

and if NIL is returned then element comparison fails; otherwise append the value returned to answer-pattern and get the next source / target pair;

5. in any other situation element comparison fails.

```
(|PATTERN pat <|PEXPR>)
                                                    |EXPR|
                                                 (|FLAMBDA)|
```

pat is not |EVALed; |PEXPR, if given, is. |PATTERN will match pat against patterns in |PEXPR (default the current execution instance) with the first matching pattern returned as value (using the standard |PEXPR-MH order of matching). Context for all macros is the current execution instance; failure to match processing takes place.

```
(|PATTERN-ALL pat <|PEXPR>)
                                                    |EXPR|
                                                 (|FLAMBDA)|
```

pat is not |EVALed; |PEXPR, if given, is. This object will match pat against all patterns in |PEXPR (default the current execution instance) with a list of matching patterns returned as value. Any assignments in pat are made to the last value; the current execution instance forms the context for macro-expansion; failure to match processing does not take place.

```
(|POINTER A1 <|PEXPR>)
                                                    |EXPR|
                                                 (|FLAMBDA)|
```

A1 is not |EVALed; |PEXPR is, if given; |POINTER will find the first pattern in |PEXPR (default the current execution instance) matching (A1 |PEXPR #X) and will return the X if successful (NIL otherwise). The current execution instance is used for macro expansion; failure to match processing takes place.

(|POINTER-ALL A1 <|PEXPR>)
|EXPR|
(|FLAMBDA)

A1 is not |EVALed; |PEXPR is, if given; |POINTER-ALL will find all patterns in |PEXPR (default the current execution instance) matching (A1 |PEXPR #X) and will return a list of all the X's so found (or NIL if none are found). Macros are expanded in the context of the current execution instance; failure to match processing does not take place.

(|UNASSIGNED A1)
EXPR|
(LAMBDA)|

This simple object will return T if A1 has the value |U (i.e. it has been unassigned as part of a macro-conflict); NIL otherwise.

A1.6 Objects Which Manipulate Patterns

(|ASSERT pat <|PEXPR <int-flag>>)
|EXPR|
(|FLAMBDA)

|PEXPR (default the current execution instance) and int-flag (default NIL) are |EVALed; pat is not, but any ":", "/", "!", "$", or "¬" macros are expanded (note that ":" and "/" designate pattern labels in |PEXPR, not in the current execution instance, unless they are the same). pat itself can be an ordinary pattern "(...)" or it can be a labelled pattern "name-i : (...)".

|ASSERT will assert pat in |PEXPR; i.e. will add it on to the beginning of the list of patterns already attached to |PEXPR under the marker |PEXPR on its property list. The order is important since |PEXPR-MH just scans down this list when looking for a match for a message. Adding more

recent patterns to the beginning of the list effectively gives a "most recent to least recent" search order. |COND= interrupt checking will be carried out if int-flag is T.

After adding the pattern, |ASSERT executes any IF-ADDED procedure that may be associated with the head of pat. Such patterns look like

(IF-ADDED head ?pattern !(|PROG () ---))

and allow such things as inverses and other "automatic" side-effects to be accomplished for a particular type of pattern. When it is finished, |ASSERT returns pat (with appropriate macro expansions substituted for macro elements).

(|ASSERTLIS list-of-pats)	|EXPR (|FLAMBDA)

This object will assert each pattern in the list of pats using |ASSERT. Each such pattern looks like

(pat <|PEXPR <int-flag>>)

and |ASSERTLIS returns the list of results returned from each |ASSERT.

(|REMASSERT pat <|PEXPR>)	|EXPR (|FLAMBDA)

The rules for |EVALing and macro-expanding are the same as those for |ASSERT except that pat can also be of the form "/pattern-label". If this is the case the pattern of that name in |PEXPR will be removed; for any other kind of pattern, |REMASSERT will look through |PEXPR for a pattern matching pat and remove the first one it finds (no failure to match processing takes place). After removing the pattern, |REMASSERT looks to the pattern head for a pattern of the form

(IF-REMOVED head ?pattern !(|PROG () ...))

Which will take care of any side-effects of the pattern removal.

|REMASSERT returns the pattern which was removed.

(|EL N1 pat)	EXPR (LAMBDA)

Returned as value from this EXPR will be the N1-th element of pat.

A1.7 Objects for Searching

(|SEARCH test-fn expand-fn <start-|PEXPR <restart>>)	|EXPR (|LAMBDA)

This is |LISP's breadth-first searching routine. test-fn and expand-fn are either |EXPRs or |LAMBDA expressions of one argument. start-|PEXPRs (default a list containing the current execution instance) is given as argument to test-fn which either returns NIL or non-null. If non-null, the search terminates with that as the answer. If NIL, the search continues with expand-fn being executed with the list start-|PEXPRs as argument. expand-fn returns either NIL or a list of |PEXPR names into which the search is to expand. In the former case the search terminates with a NIL answer; in the latter case the list of newly expanded names is passed to test-fn to see if it approves. If not, then another round of expansion must take place; if so, the value returned from test-fn is returned as answer. Note that all |EVALing takes place in the execution instance which initiated the search (although, naturally, the functions themselves can send messages to other objects if so desired).

The last argument, "restart", if given indicates that

the search can be restarted. It is the name under which is stored all data needed to restart a search. Thus, if the answer returned from a supposedly successful search later turns out to be inadequate, new answers can be generated. This is just the capability for generators (see CONNIVER, McDermott and Sussman (1974)). If no restart is specified, or if expand-fn ever returns NIL, then no data is stored for restart.

(|NEXT-SEARCH search-name)	|EXPR (|LAMBDA)

This is used to restart any search formerly suspended under the name "search-name". The search proceeds normally from where it left off.

A1.8 Objects Involved in Saving Stacks

(|AU-REVOIR A0 A1 <A2>)	EXPR (LAMBDA)

A1 and A2 are arguments like those of \|RETURN, that is they indicate a value and a return point on the current execution instance stack. If A2 is omitted, the nearest enclosing \|PROG or \|EVENT-SEQUENCE is assumed to be the return point. The additional argument A0 designates the place to save the portion of the stack stretching back from the \|AU-REVOIR form to the A2 form. A pattern

(\|STACK-SAVE current-ex-inst A0 ("the stack"))

is used for such storage.

As is the case with \|RETURN, if A2 is the keyword \|HANDLER, control goes back to the matcher within the message handler for the current execution instance; and if A1 is NIL, the match will fail at that point. If failure

to match processing fails to rectify the situation, then an |AU-REVOIR return condition

(|RETURN-COND ex-environ-ex-inst

    |AU-REVOIR (current-ex-inst A0) NIL)

is set up in the execution environment execution instance. If, on the other hand, failure to match does succeed in finding a matching pattern, then

(|RETURN-COND ex-environ-ex-inst

    |AU-REVOIR (current-ex-inst A0) matching-pat)

will be set up in ex-environ-ex-inst. In either case, the stack has been saved so that control can be resumed if somebody up there wants to |RESTART the |AU-REVOIRed execution instance.

(|RESUME A0 <S1>)	EXPR (LAMBDA)

This is the object used to resume the execution of the stack stored with A0. The stack is restored by replacing the |RESUME |EVAL block with the old stack and the old |AU-REVOIR |EVAL block by S1, if given (if not given, the value originally returned by |AU-REVOIR is substituted). Execution can then resume with the restored portion of the stack going first. The use of |RESUME in conjunction with |AU-REVOIR effectively gives a co-routining capability within an execution instance.

Note that incompatibilities are all too possible between the current stack and the restored stack; it is up to the user to be careful when using these functions. Also do not confuse |RESUME, used within an execution instance, with |RESTART, used from one execution instance to restart another.

(|RESTART old-ex-inst <A0 <S1>>)	EXPR (LAMBDA)

This object allows the restart of an execution instance previously suspended by an interrupt or an |AU-REVOIR. If old-ex-inst was suspended by an interrupt, A0 and S1 cannot be specified and the execution instance from which the command was issued should lie in the execution environment of old-ex-inst. If these criteria are met, old-ex-inst is scheduled with all its pointers intact and control is passed to the scheduler.

If old-ex-inst was suspended by an |AU-REVOIR, then A0, the marker under which the old stack is stored, must be specified. S1 has the same meaning as for |RESUME. |RESTART will create a new execution instance, with EX-INSTANCE-OF pointer to old-ex-inst and EX-ENVIRON pointer to the execution instance which issued the |RESTART. The stack of this instance will be initialized to the stack stored under A0 in old-ex-inst, with the same restrictions as in |RESUME. This new execution instance is put on the execute queue (with the same priority as old-ex-inst) and control resumes in the scheduler. Eventually, the new execution instance will run and when finished will return to the |RESTART execution instance rather than the original old-ex-inst EX-ENVIRON execution instance.

Note that what has been created here is an execution instance of an execution instance where the "higher" execution instance points to the old EX-ENVIRON and the "lower" execution instance points to the new EX-ENVIRON. This allows a similar distinction to the ALINK / CLINK distinction of the Bobrow and Wegbreit (1973) control scheme in that it allows the |RESTART and |AU-REVOIR environments to be kept distinct. However, most searches, as currently designed, will always search the new execution

environment even if (as would often be the case after a |RESTART) the old execution environment is more appropriate. Similar problems would arise when looking through old episodes in memory.

A1.9 Special Purpose Objects

```
(|EVENT-SEQUENCE (locals) S1 S2 ... Sn)
                                        |EXPR
                                    (|FLAMBDA)
```

This is a special purpose object whose format is very similar to that of a |PROG. The action of the object is very similar to |PROG in that the locals are bound to NIL on the current stack, and the Si are executed in sequence S1, S2, ... (unless over-ridden by a |GO). The difference lies in the meaning of an atomic Si, say STEPk; whereas in a |PROG STEPk would be ignored except as a label for a |GO, in an |EVENT-SEQUENCE it is a signal that another "step" in the |EVENT-SEQUENCE is about to take place. The next Sj, |PEXPR-call, which is a call to a |PEXPR is considered to be that step, and after it is taken (that is after the messages have been sent and the answer returned) |EVENT-SEQUENCE does some special processing. It asserts the following two patterns:

(STEP current-ex-inst STEPk |PEXPR-call-ex-inst)

is asserted in the current execution instance; and

(THEN STEPk-1 |PEXPR-call-ex-inst)

is asserted in the execution instance of the previous step (i.e. STEPk-1).

These two types of pattern essentially allow the preservation of a permanent record of the episode undertaken by the |EVENT-SEQUENCE, a record which can be accessed from the current execution instance by looking at STEPk patterns and which can be traversed in sequential

order from beginning to end along THEN links. The following idiosyncracies should be noted: there is obviously no THEN pattern to the first step; if no |PEXPR calls are found before the next step is encountered, |EVENT-SEQUENCE ignores the missing step; if more than one |PEXPR call is entered into before the next step, |EVENT-SEQUENCE ignores all but the first as far as keeping a record of them is concerned. Termination conditions for and values returned from |EVENT-SEQUENCE are identical to |PROG.

```
(|DO-PARALLEL (L1 ... Ln) <UNTIL test-cond>)
                                                    |EXPR|
                                                (|FLAMBDA)|
```

|DO-PARALLEL is a |EXPR which directs the (simulated) parallel execution of L1 through Ln each of which is assumed to be a call to a |PEXPR. On the first pass |DO-PARALLEL will |EVAL each of the Li in turn, and will also |EVAL test-cond (an arbitrary atom, |EXPR, |PEXPR, EXPR, or lambda of some description - if unspecified, it is assumed to always be NIL). When test-cond |EVALs to something other than NIL, |DO-PARALLEL is finished and returns the value returned from test-cond as result. Otherwise, it keeps cycling around the Li, |RESTARTing any that have been interrupted. It may eventually be the case that none of the Li can be |RESTARTed, having exceeded their |END= conditions, in which case |DO-PARALLEL will also cease and return NIL as value.

|DO-PARALLEL is useful if several things need to be done pseudo-simultaneously. Note that the user can specify the interrupt conditions on the Li to achieve any kind of time slicing desired.

APPENDIX II

Concert Scenario |PEXPRs

This Appendix presents some of the more elaborate pattern expressions which arise in the examples of Chapter V. The Appendix is divided into two parts, the first part outlining |PEXPRs which are needed for Conversation I (the ticket buying conversation), and the second part outlining |PEXPRs which are needed for Conversations II and III (the drink buying conversation and the conversation with a friend at intermission).

A2.1 Conversation I

```
<|PDEF ATTEND-CONCERT
   (SUPERSET ATTEND-CONCERT ATTEND)
   (EXECUTE ATTEND-CONCERT SELF ?THIS-CONCERT
     !(|EVENT-SEQUENCE ()
        (|ASSERT (ATTENTION SELF !THIS-CONCERT))
        (|ASSERT (PURPOSE SELF (ATTEND SELF !THIS-CONCERT)))
        (|SETQ LOC-CONCERT (|POINTER LOCATION THIS-CONCERT))
      STEP1
      ;go to theatre
        (GOTO (EXECUTE GOTO SELF !LOC-CONCERT ?GO-PLACE1))
        (CHECK-FAILURE)
        (!THIS-CONCERT (ENTRANCE-REQUIREMENT !THIS-CONCERT
                                ?DESIRED-TICKET))
        (|ASSERT (EVENT !DESIRED-TICKET !THIS-CONCERT)
                         DESIRED-TICKET)
      STEP2
      ;buy ticket
        (BUY (EXECUTE BUY SELF !DESIRED-TICKET ?BUY-RESULT))
        (CHECK-FAILURE)
      STEP3
      ;go into lobby
        (|SETQ LOBBY (|POINTER LOBBY LOC-CONCERT))
        (GOTO (EXECUTE GOTO SELF !LOBBY ?GO-PLACE2))
        (CHECK-FAILURE)
      STEP4
      ;go to seat
        (|SETQ SEATS (|POINTER REPN DESIRED-TICKET))
        (GOTO (EXECUTE GOTO SELF !SEATS ?GO-PLACE3))
        (CHECK-FAILURE)
      STEP5
      ;listen to first half of concert
        (SETQ THIS-AGENDA (|POINTER AGENDA THIS-CONCERT))
        (|PATTERN (FIRST-HALF !THIS-CONCERT ?BEGIN-PROGRAM)
                            THIS-AGENDA)
        (LISTEN (EXECUTE SELF !BEGIN-PROGRAM ?FIRST-DONE))
        (CHECK-FAILURE)
      STEP6
      ;go back to lobby
        (GOTO (EXECUTE GOTO SELF !LOBBY ?IN-LOBBY))
        (CHECK-FAILURE)
      STEP7
      ;buy a drink
        (|SETQ BAR-WHERE (|POINTER BAR LOC-CONCERT))
```

```
    (|PATTERN (WANT SELF (DRINK SELF ?DESIRED-DRINK))
              PRE-DINNER-DRINK)
    (BUY (EXECUTE BUY SELF !DESIRED-DRINK ?BUY-DONE))
    (CHECK-FAILURE)
  STEP8
  ;fill in time and drink beverage until buzzer
    (|DO-PARALLEL
      ((FILL-IN-TIME (EXECUTE FILL-IN-TIME SELF ?END-FILL)
                     (|TIME= 5))
       (SIP (EXECUTE SIP SELF !DESIRED-DRINK ?DRINK-RESULT)
            (|TIME= 1)) )
      UNTIL
       (NOT (NULL (CHECK-FOR-ACTIVE-SUBSET BUZZER-SOUND))))
    (CHECK-FAILURE)
  STEP9
  ;go back to seat
    (GOTO (EXECUTE GOTO SELF !SEATS ?IN-SEATS))
    (CHECK-FAILURE)
  STEP10
  ;listen to second half of the concert
    (|PATTERN (SECOND-HALF !THIS-CONCERT ?END-PROGRAM)
              THIS-AGENDA)
    (LISTEN (EXECUTE LISTEN SELF !END-PROGRAM ?SECOND-DONE))
    (CHECK-FAILURE)
  STEP11
  ;go back home
    (GOTO (EXECUTE GOTO SELF HOME ?BACK-HOME))
    (CHECK-FAILURE)
    (|RETURN (|CURRENT)) ))      >



<|PDEF BUY
    (SUPERSET BUY ACTUAL-TRANSACTION)
    (EXECUTE BUY ?BUYER ?ITEM
     !(EVENT-SEQUENCE ()
       (|AND (NEQ BUYER 'SELF)
             (|ASSERT (FAIL (NOT SELF BUYER)))
             (|RETURN NIL))
       (|PATTERN (LOCATION !ITEM ?PLACE-ITEM) ITEM)
       (|PATTERN (SELLER !ITEM ?SELLER) ITEM)
      STEP1
      ;go to ticket booth
       (GOTO (EXECUTE GOTO SELF !PLACE-ITEM ?GOT-THERE))
       (CHECK-FAILURE)
       (|ASSERT (BUYER !ITEM !BUYER))
       (|ASSERT (SELLER !ITEM !SELLER))
       (|ASSERT (PURPOSE !SELLER (SELL !SELLER !ITEM)))
       (|ASSERT (PURPOSE !BUYER (BUY !BUYER !ITEM)))
       (|ASSERT (ATTENTION !BUYER !ITEM))
      STEP2
      ;take part in conversation to buy ticket
       (BUY-CONVERSATION (EXECUTE BUY-CONVERSATION
                 !BUYER !SELLER !ITEM ?CONV-RESULT))
       (CHECK-FAILURE)
       (|RETURN (|CURRENT))))    >



<|PDEF BUY-CONVERSATION
    (SUPERSET BUY-CONVERSATION SOCIAL-TRANSACTION-CONVERATION)
    (EXECUTE BUY-CONVERSATION ?BUYER ?SELLER ?ITEM
     !(EVENT-SEQUENCE ()
      STEP1
      ;start up conversation
       (WHAT-DO-YOU-WANT
         (EXECUTE WHAT-DO-YOU-WANT !SELLER !BUYER
```

```
                ?WHAT-WANT-CONV))
    (CHECK-FAILURE)
    (TIE-IN WHAT-WANT-CONV)
    (|PATTERN (WANT !BUYER (EXCHANGE ?BUYER-HAS
                ?BUYER-WANTS)) BUYER)
    (|PATTERN (WANT !SELLER (EXCHANGE ?SELLER-HAS
                ?SELLER-WANTS)) SELLER)
  STEP2
  ;bargain over what buyer wants
    (BARGAIN
      (EXECUTE BARGAIN !SELLER !BUYER !SELLER-HAS
                !BUYER-WANTS !ITEM ?BARGAIN-1-CONV))
    (CHECK-FAILURE)
    (TIE-IN BARGAIN-1-CONV)
  STEP3
  ;bargain over what seller wants
    (BARGAIN
      (EXECUTE BARGAIN !BUYER !SELLER !BUYER-HAS
                !SELLER-WANTS !ITEM ?BARGAIN-2-CONV))
    (CHECK-FAILURE)
    (TIE-IN BARGAIN-2-CONV)
  STEP4
  ;exchange cost of item for ite
    (EXCHANGE
      (EXECUTE EXCHANGE !BUYER !SELLER
         !(|POINTER COST ITEM) !ITEM ?CONV-EXCHANGE))
    (CHECK-FAILURE)
    (TIE-IN CONV-EXCHANGE)
  STEP5
  ;close out the conversation
    (FAREWELL
      (EXECUTE FAREWELL !BUYER !SELLER ?GOOMBYE))
    (CHECK-FAILURE)
    (TIE-IN GOOMBYE)
    (|RETURN (|CURRENT))))    >


<|PDEF WHAT-DO-YOU-WANT
    (SUPERSET WHAT-DO-YOU-WANT GREETING)
    (EXECUTE WHAT-DO-YOU-WANT ?SPEAKER1 ?SPEAKER2
     !(EVENT-SEQUENCE ()
       (|SETQ DIRECTIONS (ESTABLISH-IDENTITIES
                SPEAKER1 SPEAKER2))
       (|SETQ DIRECTION-FIRST (CAR DIRECTIONS))
       (|SETQ DIRECTION-SECOND (CADR DIRECTIONS))
      STEP1
      ;speaker1 inquires as to purpose of speaker2
       (|APPLY 'INQUIRE DIRECTION-FIRST 'INQUIRE
         SPEAKER1 SPEAKER2
         ¬(PURPOSE !SPEAKER2 *UNKNOWN*) '?NEW-UTT1)
       (CHECK-FAILURE)
       (TIE-IN NEW-UTT1)
      STEP2
      ;speaker2 responds with his purpose
       (|APPLY 'RESPOND DIRECTION-SECOND 'RESPOND
         SPEAKER2 SPEAKER1
         %(PURPOSE !SPEAKER2 ?WHAT-PURPOSE) '?NEW-UTT2)
       (CHECK-FAILURE)
       (TIE-IN NEW-UTT2)
       (|RETURN (|CURRENT))))    >
```

```
<|PDEF BARGAIN
    (SUPERSET BARGAIN SOCIAL-TRANSACTION-CONVERSATION)
    (EXECUTE BARGAIN ?QUESTIONER ?RESPONDER
        ?POSN-Q ?POSN-R ?ITEM
     !(EVENT-SEQUENCE ()
       (|SETQ DIRECTIONS (ESTABLISH-IDENTITIES
                   QUESTIONER RESPONDER))
       (|SETQ DIRECTION-QUESTIONER (CAR DIRECTIONS))
       (|SETQ DIRECTION-RESPONDER (CADR DIRECTIONS))
       (|SETQ ORDER-Q (|POINTER BARGAIN-ORDER POSN-Q))
       (|SETQ ORDER-R (|POINTER BARGAIN-ORDER POSN-R))
     STEP1
     ;first set up initial bargaining positions
       (|SETQ CURRENT-ISSUE (CAR ORDER-Q))
       (|COND ((NULL CURRENT-ISSUE)
                (|SETQ CURRENT-ISSUE (CAR ORDER-R))
                (|AND (NULL CURRENT-ISSUE)
                      (|AU-REVOIR 'AGAIN (|CURRENT)))
                (|SETQ ORDER-R (CDR ORDER-R)))
              (T (|SETQ ORDER-Q (CDR ORDER-Q)))))
       (|SETQ STANCE-Q (MOST-IMPORTANT
                (!CURRENT-ISSUE !ITEM ?WHAT1)
                POSN-Q QUESTIONER 'NEXT-STANCE-Q))
       (|SETQ STANCE-R (MOST-IMPORTANT
                (!CURRENT-ISSUE !ITEM ?WHAT2)
                POSN-R RESPONDER 'NEXT-STANCE-R))
       (|COND ((|AND (NULL STANCE-Q) (NULL STANCE-R))
                (|AU-REVOIR 'AGAIN (|CURRENT)))
              ((NULL STANCE-Q)
                (|SETQ STANCE-Q
                 (LIST CURRENT-ISSUE ITEM '=(X T))))
              ((NULL STANCE-R)
                (|SETQ STANCE-R
                 (LIST CURRENT-ISSUE ITEM '=(X T))))
              (T T))
       (|SETQ STANCE-TEM-Q STANCE-Q)
       (|SETQ STANCE-TEM-R STANCE-R)
     ;questioner asks for responder's stance
       (|APPLY 'INQUIRE DIRECTION-QUESTIONER 'INQUIRE
           QUESTIONER RESPONDER
          ¬(!CURRENT-ISSUE !ITEM *UNKNOWN*) '?UTT-Q)
       (CHECK-FAILURE)
       (TIE-IN UTT-Q)
     STEP2
     ;responder replies with his current stance
       (|APPLY 'RESPOND DIRECTION-RESPONDER 'RESPOND
          RESPONDER QUESTIONER STANCE-R '?UTT-R)
       (CHECK-FAILURE)
       (TIE-IN UTT-R)
     STEP3
     ;questioner concocts a reply
       (|SETQ STANCE-R (COVER-PATTERN STANCE-R
                (|POINTER CONTENT UTT-R)))
       (|AND (NULL STANCE-R)
             (EXPLAIN-BAD
                (EXECUTE EXPLAIN-BAD !UTT-R ?EXPLAIN-ANS)))
       (|SETQ COMP-PAT (|MATCH STANCE-R (|CURRENT)
                               STANCE-Q (|CURRENT)))
       (|COND ((NULL COMP-PAT)
                (|SETQ STANCE-TEM-Q (|RESUME 'NEXT-STANCE-Q))
                (|COND ((NULL STANCE-TEM-Q)
                        (|AND (NULL STANCE-TEM-R)
                              (|SETQ FIRST-IN T)
                              (|GO 'STEP7)))
                       (T (|SETQ STANCE-Q STANCE-TEM-R)))
                (|APPLY 'INQUIRE DIRECTION-QUESTIONER
                        QUESTIONER RESPONDER STANCE-Q
                        '?UTT-Q)
                (CHECK-FAILURE)
                (TIE-IN UTT-Q)
                (|GO 'STEP4))
```

```
          (T (|ASSERT !COMP-PAT ITEM)
             (|GO 'STEP5)))
STEP4
;responder concocts a reply
  (|SETQ STANCE-Q (COVER-PATTERN
          STANCE-Q (|POINTER CONTENT UTT-Q))
  (|AND (NULL STANCE-Q)
        (EXPLAIN-BAD (EXECUTE EXPLAIN-BAD !UTT-Q
                              ?EXPLAIN-ANS)))
  (|SETQ COMP-PAT (|MATCH STANCE-Q (|CURRENT)
                          STANCE-R (|CURRENT)))
  (|COND ((NULL COMP-PAT)
          (|SETQ STANCE-TEM-R
            (|RESUME 'NEXT-STANCE-R))
          (|COND ((NULL STANCE-TEM-R)
                  (|AND (NULL STANCE-TEM-Q)
                        (|SETQ FIRST-IN T)
                        (|GO 'STEP8)))
                 (T (|SETQ STANCE-R STANCE-TEM-R)))
          (|APPLY 'RESPOND DIRECTION-RESPONDER
                  RESPONDER QUESTIONER STANCE-R '?UTT-R)
          (CHECK-FAILURE)
          (TIE-IN UTT-R)
          (|GO 'STEP3))
         (T (|ASSERT !COMP-PAT ITEM)
            (|GO 'STEP6)))
STEP5
;agreement reached by questioner
  (|APPLY 'AGREE DIRECTION-QUESTIONER 'AGREE
     QUESTIONER RESPONDER COMP-PAT '?UTT-Q)
  (CHECK-FAILURE)
  (TIE-IN UTT-Q)
  (|GO 'STEP1)
STEP6
;agreement reached by responder
  (|APPLY 'AGREE DIRECTION-RESPONDER 'AGREE
     RESPONDER QUESTIONER COMP-PAT '?UTT-R)
  (CHECK-FAILURE)
  (TIE-IN UTT-R)
  (|GO 'STEP1)
STEP7
;questioner irreconcilably disagrees with STANCE-R
  (|APPLY 'DISAGREE DIRECTION-QUESTIONER 'DISAGREE
     QUESTIONER $ESPONDER STANCE-R '?UTT-Q)
  (CHECK-FAILURE)
  (TIE-IN UTT-Q)
  (|COND (FIRST-IN
          (|SETQ FIRST-IN NIL)
          (|GO 'STEP8))
         (T (|GO 'STEP9)))
STEP8
;responder irreconcilably disagrees with STANCE-Q
  (|APPLY 'DISAGREE DIRECTION-RESPONDER 'DISAGREE
          RESPONDER QUESTIONER STANCE-Q '?UTT-R)
  (CHECK-FAILURE)
  (TIE-IN UTT-R)
  (|COND (FIRST-IN
          (|SETQ FIRST-IN NIL)
          (|GO 'STEP7))
         (T (|GO 'STEP9)))
STEP9
;finis with failure
  (|ASSERT ¬(FAIL !STANCE-Q !STANCE-R))
  (|AU-REVOIR 'AGAIN NIL)
  (|GO 'STEP1)))                    >
```

```
<|PDEF SELF-HAS-TICKET-BARGAINING-POSN
      (SUPERSET SELF-HAS-TICKET-BARGAINING-POSN
                HAS-BARGAINING-POSN)
 S1 : (COST ↑TICKET-FOR-CONCERT ↑DOLLARS-5)
 S2 : (COST ↑TICKET-FOR-CONCERT ↑DOLLARS-10)
      (IMPORTANCE SELF /S1 10)
      (IMPORTANCE SELF /S2 9)
      (BARGAIN-ORDER SELF-HAS-TICKET-BARGAINING-POSN (COST)) >



<|PDEF TICKET-SELLER-WANTS-BARGAINING-POSN
      (SUPERSET TICKET-SELLER-WANTS-BARGAINING-POSN
                WANTS-BARGAINING-POSN)
 S1 : (COST ↑TICKET !(|POINTER COST TICKET))
      (IMPORTANCE ↑TICKET-SELLER /S1 10)
      (BARGAIN-ORDER TICKET-SELLER-WANTS-BARGAINING-POSN
                     (COST))   >



<|PDEF EXCHANGE
    (SUPERSET EXCHANGE ACTUAL-TRANSACTION)
    (EXECUTE EXCHANGE ?PERSON1 ?PERSON2 ?ITEM1 ?ITEM2
      !(EVENT-SEQUENCE ()
        (|SETQ DIRECTIONS (ESTABLISH-IDENTITIES
                           PERSON1 PERSON2))
        (|SETQ DIRECTION-FIRST (CAR DIRECTIONS))
        (|SETQ DIRECTION-SECOND (CADR DIRECTIONS))
      STEP1
      ;person1 gives item1 to person2
        (|APPLY 'GIVE DIRECTION-FIRST 'GIVE PERSON1
                PERSON2 ITEM1 ?GIVE-RESULT)
        (CHECK-FAILURE)
      STEP2
      ;person2 thanks him for it
        (|APPLY 'THANKS DIRECTION-SECOND 'THANKS
                PERSON2 PERSON1 ITEM1 ?THANKS-UTT1)
        (CHECK-FAILURE)
        (TIE-IN THANKS-UTT1)
        (|AND (NULL (|RETVALUE))(|SETQ RUDE-FLAG T))
      STEP3
      ;person2 gives item2 to person1
        (|APPLY 'GIVE DIRECTION-SECOND 'GIVE
                PERSON2 PERSON1 ITEM2 ?GIVE-RESULT2)
        (CHECK-FAILURE)
      STEP4
      ;person1 thanks him for it if person2 was polite
        (|COND (RUDE-FLAG)
               (T (|APPLY 'THANKS DIRECTION-FIRST 'THANKS
                          PERSON1 PERSON2 ITEM2 ?THANKS-UTT2)
                  (CHECK-FAILURE)
                  (TIE-IN THANKS-UTT2)))
        (|RETURN (|CURRENT))))   >



<|PDEF FAREWELL
    (SUPERSET FAREWELL DIALOGUE)
    (EXECUTE FAREWELL ?PERSON1 ?PERSON2
     !(EVENT-SEQUENCE ()
        (|SETQ DIRECTIONS (ESTABLISH-IDENTITIES
                           PERSON1 PERSON2))
        (|SETQ DIRECTION-FIRST (CAR DIRECTIONS))
        (|SETQ DIRECTION-SECOND (CADR DIRECTIONS))
```

```
        STEP1
        ;person1 says goodbye to person2
          (|APPLY 'GOODBYE DIRECTION-FIRST 'GOODBYE
                  PERSON1 PERSON2 #ANY ?CONV-1)
          (CHECK-FAILURE)
          (TIE-IN CONV-1)
        STEP2
        ;person2 says goodbye to person1
          (|APPLY 'GOODBYE DIRECTION-SECOND 'GOODBYE
                  PERSON2 PERSON1 #ANY ?CONV-2)
          (CHECK-FAILURE)
          (TIE-IN CONV-2)
          (|RETURN (|CURRENT))))    >



<|PDEF INQUIRE
    (SUPERSET INQUIRE INTERROGATIVE)
    (EXPECT INQUIRE ?SPEAKER ?LISTENER ?CONTENT
     !(|PROG ()
       %(SURFACE ¢WHAT-|PEXPR ?UTTERANCE)
       (|COND
          ((NULL UTTERANCE)
           (|SETQ UTTERANCE (HEAR-WORDS))
           (|ASSERT (SURFACE $|CURRENT !UTTERANCE))
           (|SETQ APPROP-SUBSET
            (CHECK-FOR-ACTIVE-SUBSET 'INQUIRE))
           (|COND ((NULL APPROP-SUBSET))
                  (T (REPLACE APPROP-SUBSET))))
          (T (|ASSERT (SURFACE $|CURRENT !UTTERANCE))))
       (|ASSERT (SPEAKER $|CURRENT !SPEAKER))
       (|ASSERT (LISTENER $|CURRENT !LISTENER))
       (|AND (NEQ (LAST UTTERANCE) "?")
             (|ASSERT (FAIL (NOT INQUIRE !UTTERANCE)))
             (|RETURN NIL))
       (INQUIRE-CLAUSE
         (INTERPRET INQUIRE-CLAUSE !UTTERANCE ?RESULT))
       (CHECK-FAILURE)
       (|ASSERT (CONTENT $|CURRENT
                 !(|POINTER CONTENT RESULT)))
       (|RETURN (|CURRENT))))
    (EXECUTE INQUIRE ?SPEAKER ?LISTENER ?CONTENT
     !(|PROG ()
       (|COND ((|UNASSIGNED CONTENT)
               (|SETQ CONTENT '*UNKNOWN*)
               (NOUNG (GENERATE NOUNG *UNKNOWN* ?OUT)))
              (T (INQUIRE-CLAUSE (GENERATE INQUIRE-CLAUSE
                                  !CONTENT ?OUT))))
       (|SETQ DOWN-|PEXPR (CAR (|RETCODES)))
       (|SETQ OUT (SPEAK-WORDS (APPEND1 OUT "?")))
       (|ASSERT (SPEAKER $|CURRENT !SPEAKER))
       (|ASSERT (LISTENER $|CURRENT !SPEAKER))
       (|ASSERT (SURFACE $|CURRENT !OUT))
       (|ASSERT (CONTENT $|CURRENT
                 !(|POINTER CONTENT DOWN-|PEXPR)))
       (|RETURN (|CURRENT))))    >



<|PDEF YES2
    (SUPERSET YES2 INQUIRE)
    (EXPECT YES2 ?SPEAKER ?LISTENER ?CONTENT
     !(|PROG ()
       %(SURFACE ¢WHAT-|PEXPR ?UTTERANCE)
       (|COND ((NULL UTTERANCE)
               (|SETQ UTTERANCE (HEAR-WORDS))
               (|ASSERT (SURFACE $|CURRENT !UTTERANCE))
```

```
                (|SETQ APPROP-SUBSET
                   (CHECK-FOR-ACTIVE-SUBSET 'YES2))
                (|COND ((NULL APPROP-SUBSET))
                       (T (REPLACE APPROP-SUBSET))))
               (T (|ASSERT (SURFACE $|CURRENT !UTTERANCE))))
        (|ASSERT (SPEAKER $|CURRENT !SPEAKER))
        (|ASSERT (LISTENER $|CURRENT !LISTENER))
        (|AND (|UNASSIGNED CONTENT)
              (|SETQ CONTENT ¬(PURPOSE !LISTENER *UNKNOWN*)))
        (|COND ((EQUAL UTTERANCE '("YES" "?"))
                (|ASSERT (CONTENT $|CURRENT !CONTENT))
                (|RETURN (|CURRENT)))
               (T (|ASSERT (FAIL (NOT YES2 !UTTERANCE)))
                (|RETURN NIL)))))
    (EXECUTE YES2 ?SPEAKER ?LISTENER ?CONTENT
     !(|PROG ()
        (|ASSERT (SPEAKER $|CURRENT !SPEAKER))
        (|ASSERT (LISTENER $|CURRENT !LISTENER))
        (|AND (|UNASSIGNED CONTENT)
              (|SETQ CONTENT ¬(PURPOSE !LISTENER *UNKNOWN*)))
        (|ASSERT (CONTENT $|CURRENT !CONTENT))
        (|SETQ OUTPUT (SPEAK-WORDS '("YES" "?")))
        (|ASSERT (SURFACE $|CURRENT !OUTPUT))
        (|RETURN (|CURRENT))))
    (ASSOC YES2
     !(|PROG ()
        (|COND ((EQ (|POINTER EX-INSTANCE-OF
                      (|POINTER EX-ENVIRON)) "YES")
                    (|ASSERT (WANT-TO-GO YES2)))
               (T (SPREAD-ASSOC 'PART-OF 'DEMON))))))    >
```

A2.2 Conversations II and III

```
<|PDEF PRE-DINNER-DRINK
    (SUPERSET PRE-DINNER-DRINK ALCOHOLIC-DRINK)
    (WANT SELF
     !(|PROG ()
       %(ATTENTION SELF ?CUR-EVENT)
        (|COND ((SUBINSTANCE CUR-EVENT 'CONCERT)
                (|RETURN '(DRINK SELF
                           ↓JOHNNY-WALKER-SCOTCH)))
               (T (|RETURN '(DRINK SELF
                           ↓FIVE-STAR-RYE))))))
    (LOCATION ↓PRE-DINNER-DRINK
     !(|PROG ()
       %(ATTENTION SELF ?CUR-EVENT)
        (|RETURN (|POINTER BAR
                  (|POINTER LOCATION CUR-EVENT))))) >



<|PDEF JOHNNY-WALKER-SCOTCH
    (SUPERSET JOHNNY-WALKER-SCOTCH SCOTCH)
    (BRAND ↓JOHNNY-WALKER-SCOTCH JOHNNY-WALKER)
    (COST ↓JOHNNY-WALKER-SCOTCH ↓DOLLARS-3) >



<|PDEF SCOTCH
    (SUPERSET SCOTCH PRE-DINNER-DRINK)
    (SUPERSET SCOTCH AFTER-DINNER-DRINK)
```

```
    (MIXER ↑SCOTCH ↑ICE)    >


<|PDEF ALCOHOLIC-DRINK
    (SUPERSET ALCOHOLIC-DRINK DRINK)
    (SELLER ↑ALCOHOLIC-DRINK ↑BARTENDER)   >


<|PDEF BARTENDER
    (SUPERSET BARTENDER SELLER)
    (SELL ↑BARTENDER ↑ALCOHOLIC-DRINK)
    (WANT ↑BARTENDER
          (EXCHANGE BARTENDER-HAS-BARGAINING-POSN
                    BARTENDER-WANTS-BARGAINING-POSN))   >


<|PDEF SELF-AS-A-BUYER-OF-ALCOHOLIC-DRINK
    (SUPERSET SELF-AS-A-BUYER-OF-ALCOHOLIC-DRINK
              SELF-AS-A-BUYER)
    (BUY SELF ↑ALCOHOLIC-DRINK)
    (WANT SELF (EXCHANGE
                SELF-HAS-DRINK-BARGAINING-POSN
                SELF-WANTS-DRINK-BARGAINING-POSN))   >


<|PDEF SELF-HAS-DRINK-BARGAINING-POSN
      (SUPERSET SELF-HAS-DRINK-BARGAINING-POSN
                HAS-BARGAINING-POSN)
 S1 : (COST ↑ALCOHOLIC-DRINK
            !(|POINTER COST ALCOHOLIC-DRINK))
      (IMPORTANCE SELF /S1 10)
      (BARGAIN-ORDER SELF-HAS-DRINK-BARGAINING-POSN
                     (COST))   >


<|PDEF SELF-WANTS-DRINK-BARGAINING-POSN
      (SUPERSET SELF-WANTS-DRINK-BARGAINING-POSN
                WANTS-BARGAINING-POSN)
 S1 : (BRAND ↑ALCOHOLIC-DRINK
             !(|POINTER BRAND ALCOHOLIC-DRINK))
 S2 : (MIXER ↑ALCOHOLIC-DRINK
             !(|POINTER MIXER ALCOHOLIC-DRINK))
      (IMPORTANCE SELF /S1 8)
      (IMPORTANCE SELF /S2 10)
      (BARGAIN-ORDER ALCOHOLIC-DRINK-BARGAINING-POSN
                     (BRAND MIXER))    >


<|PDEF BARTENDER-HAS-BARGAINING-POSN
      (SUPERSET BARTENDER-HAS-BARGAINING-POSN
                HAS-BARGAINING-POSN)
 S1 : (MIXER ↑ALCOHOLIC-DRINK
             !(|POINTER MIXER ALCOHOLIC-DRINK))
```

```
 S2 : (BRAND ↑ALCOHOLIC-DRINK
             !(|POINTER BRAND ALCOHOLIC-DRINK))
      (IMPORTANCE ↑BARTENDER /S1 6)
      (IMPORTANCE ↑BARTENDER /S2 6)
      (BARGAIN-ORDER BARTENDER-HAS-BARGAINING-POSN
                     (BRAND MIXER))    >



<|PDEF BARTENDER-WANTS-BARGAINING-POSN
      (SUPERSET BARTENDER-WANTS-BARGAINING-POSN
                WANTS-BARGAINING-POSN)
 S1 : (COST ↑ALCOHOLIC-DRINK
            !(INCREASE-BY DOLLARS-2
                 (|POINTER COST ALCOHOLIC-DRINK)))
 S2 : (COST ↑ALCOHOLIC-DRINK
            !(|POINTER COST ALCOHOLIC-DRINK))
      (IMPORTANCE ↑BARTENDER /S1 9)
      (IMPORTANCE ↑BARTENDER /S2 6)
      (BARGAIN-ORDER BARTENDER-WANTS-BARGAINING-POSN
                     (COST))     >



<|PDEF DIALOGUE
    (SUPERSET DIALOGUE MULTIPLE-SPEECH-ACTION)
    (EXECUTE DIALOGUE ?SPEAKER1 ?SPEAKER2
     !(EVENT-SEQUENCE ()
        (|SETQ DIRECTIONS (ESTABLISH-IDENTITIES
                           SPEAKER1 SPEAKER2))
        (|SETQ DIRECTION-FIRST (CAR DIRECTIONS))
        (|SETQ DIRECTION-SECOND (CADR DIRECTIONS))
        (|SETQ EXPECT1
           (COMPUTE-EXPECTATIONS SPEAKER1 SPEAKER2))
      STEP1
      ;speaker1 makes an utterance
        (|APPLY 'SPEECH-ACT DIRECTION-FIRST 'SPEECH-ACT
                SPEAKER1 SPEAKER2 EXPECT1 ?FIRST-UTT)
        (CHECK-FAILURE-DIALOGUE)
        (TIE-IN FIRST-UTT)
        (|SETQ EXPECT2 (COMPUTE-EXPECTATIONS SPEAKER2 SPEAKER1))
      STEP2
      ;speaker2 makes an utterance
        (|APPLY 'SPEECH-ACT DIRECTION-SECOND 'SPEECH-ACT
                SPEAKER2 SPEAKER1 EXPECT2 ?SECOND-UTT)
        (CHECK-FAILURE-DIALOGUE)
        (TIE-IN SECOND-UTT)
        (|GO 'STEP1)))    >
```

INDEX

Index