Implementing an Intelligent Retrieval System: The CODER System, Version 1.0

Marybeth T. Weaver and Edward A. Fox

TR 88-6

Implementing an Intelligent Information Retrieval System

The CODER System, Version 1.0

by

Marybeth Therese Weaver

Edward A. Fox, Chairman

Computer Science and Applications

(ABSTRACT)

For individuals requiring interactive access to online text, information storage and retrieval systems provide a way to retrieve desired documents and/or text passages. The CODER (COmposite Document Expert/effective/extended Retrieval) system is a testbed for determining how useful various artificial intelligence techniques are for increasing the effectiveness of information storage and retrieval systems. The system, designed previously, has three components: an analysis subsystem for analyzing and storing document contents, a central spine for manipulation and storage of world and domain knowledge, and a retrieval subsystem for matching user queries to relevant documents. This thesis discusses the implementation of the retrieval subsystem and portions of the spine and analysis subsystem. It illustrates that logic programming, specifically with the Prolog language, is suitable for development of an intelligent information retrieval system. Furthermore, it shows that system modularity provides a flexible research testbed, allowing many individuals to work on different parts of the system which may later be quickly integrated.

The retrieval subsystem has been implemented in a modular fashion so that new approaches to information retrieval can be easily compared to more traditional ones. A powerful knowledge representation language, a comprehensive lexicon and individually tailored experts using standardized blackboard modules for communication and control allowed rapid prototyping, incremental development and ready adaptability to change. The system executes on a DEC VAX 11/785 running *ULTRIX*™, a variant of 4.2 BSD UNIX. It has been implemented as a set of MU-Prolog and C modules communicating through TCP/IP sockets.

Acknowledgements

The Version 1.0 CODER system represents the efforts of many Virginia Tech graduate students. At last tally, nearly thirty students had contributed, in varying degrees, to the development of the CODER system. The participation of these students, my colleagues and friends, is gratefully acknowledged. The assistance, support and friendship of my advisor's secretary, Joy Weiss, is also much appreciated.

My office mates, Whay Lee, Qi Fan Chen and Robert France, deserve special thanks. Whay's expertise with the SMART retrieval system and with C and UNIX have been invaluable on numerous occasions. The diligence and cheerfulness of my counterpart Qi Fan Chen, the developer of the analysis subsystem, have been a source of inspiration. Robert France, who contributed the original design of CODER, deserves much credit for the current implementation of the system. His experience, cooperation and perceptiveness have made him, to borrow one of his phrases, "a much preferred co-worker."

My committe members, Dr. Edward Fox, Dr. Terry Nutter and Dr. Debbie Hix, have been a highly respected source of encouragement and assistance. I especially thank Drs. Nutter and Hix for the examples that they set. As successful women in the field of Computer Science, they provide excellent role models for me. None of this work, however, would have been possible without the genius and patience of my advisor and committee chairman, Dr. Edward Fox. I am continually astounded by the breadth of his expertise, by his ability to quickly comprehend new ideas, and by his tireless patience and helpfulness. I wish him many successes with the CODER system and with all of his endeavors.

Finally, I thank those who are dearest to me, my family and friends. My parents, Marlyn and Mary Clare Weaver, have encouraged and supported me in ways too numerous to mention. Likewise, my sisters, brother and many friends, both old and new, have been an invaluable source of support. Last but not least, I acknowledge someone special who knows nearly as much about the trials and tribulations of this work as I do myself. Sincere thanks go to Paul Smith for his love and encouragement.

This research was funded in part by grants from the National Science Foundation (IST-841887 and IRI-8730580), the Virginia Center for Innovative Technology (INF-85-016 and INF-87-012) and by AT&T equipment contributions.

Table of Contents

List of Illustrations

List of Tables

1.0 Introduction

Research often starts from an idea, a question, or an extension of a previous line of enquiry. The first thoughts are often vague and rather grandiose, tending to overestimate the resources available and underestimate the time needed to complete the project. However, without optimism much research would never be started.
-- Martin J. Kendall
-- Clifford Hawkins, 1985

The first section of this chapter discusses how the suggested link between information storage and retrieval systems and artificial intelligence has motivated this research. Section 1.2, *Evolution*, provides a historical background of the work described herein. The final section outlines the scope of this research investigation.

1.1 Motivation

The growth of the information society is apparent in the expanding size and number of machine-readable text collections. Many developments have propelled us toward W.F. Lancaster's vision of "paperless information systems" [LANC 78]; office information systems for storing correspondence and documents, desktop publishing, library systems and electronic mail systems are a few. Moreover, increased storage capacity provided by media such as CDROM promotes development of online versions of printed literature such as dictionaries, encyclopedia, reference manuals and textbooks [FOXE 86c].

For those requiring access to online text, **Information Storage and Retrieval** (ISR) systems provide a way to retrieve desired documents and/or text passages. In traditional ISR systems, a trained searcher enters a query using a predefined query language; the system executes its search strategy for matching the query to documents and displays the retrieved documents. Early system designs were simpler and were mainly concerned with performance issues; they relied on human search intermediaries who were trained to enter users' queries in a manner that would hopefully result in acceptable levels of recall and precision. However, the proliferation of available online information has suggested the need for **end users**, unaided by search intermediaries, to search text collections directly.

Faced with the challenge of developing information storage and retrieval systems for direct user access, researchers began to investigate the applicability of **Artificial Intelligence** (AI) methods [SMIT 80]. Provision of ISR systems which perform like trained intermediaries prompted studies of the functions of an intelligent ISR system [BELK 83]. Distinctions among users and their searching methods suggested the importance of user modeling research [DANI 86, BORG 86, 87a, 87b]. Furthermore, realization of improvements in performance and effectiveness when AI techniques are applied has been hypothesized. The **CODER** (COmposite Document Expert/effective/extended Retrieval) system was designed purposely to test the hypothesis that AI techniques can significantly upgrade the productivity of ISR systems [FOXE 85, 86a].

A variety of AI approaches have been incorporated into the CODER system design. Distributed **rule-based experts** centered around a blackboard/strategist complex perform separate functions, such as browsing, query formulation or searching. **Inferential reasoning** is employed by modules using the logic programming language **Prolog**. Knowledge about documents and users is stored in frames in a Prolog frame-based **knowledge representation** system. **Reasoning with uncertainty** and **truth maintenance** have been built into the CODER design. Finally, document parsing techniques are used extensively during document analysis; however, natural language processing of queries has not yet been implemented.

Later versions of the CODER system will be capable of analyzing and retrieving domain-independent documents; however, the target collection for the initial version of CODER is an accumulation of issues of the ARPANET *AIList Digest* since April, 1983. The heterogeneous documents contained in the collection vary in length, style, content and form. The CODER system includes analysis of the structure and contents of documents, and subsequent retrieval including use of canonical knowledge structures such as dates and names. Therefore, with diverse document types such as those found in the AIList collection, the CODER system can be used to test the hypothesis that retrieval employing structured knowledge provides better results than conventional retrieval methods.

1.2 Evolution

Design of the CODER system began late in 1985, and is discussed in Robert France's M.S. Thesis, "An Artificial Intelligence Environment for Information Retrieval Research." [FRAN 86]. Additional description of the CODER design may be found in technical reports and publications written by the principal investigator, Edward A. Fox [FOXE 85, 86a, 86b, 87]. As nearly as possible, the CODER retrieval subsystem development has adhered to the original high-level design specifications. The knowledge representation language and blackboard/strategist functions, specified in detail, have been minimally modified. Specifications for remaining modules were not provided; therefore, module design and implementation techniques were undertaken as part of this research.

Many of the CODER modules envisaged in the archetype system illustrated in Figure 1 on page 5 have not been implemented and several new modules have been added. As the current retrieval subsystem diagram in Figure 2 on page 6 depicts, implementation of CODER modules has progressed somewhat differently than was originally planned. The work of Belkin et al. [BELK 83, 84] influenced design development and led to some of the changes between the original and current versions of the system.

The spine, composed of the knowledge administration complex, the document database and the lexicon, has been implemented as pictured in Figure 1 on page 5. The classification specialists, although relied upon by the analysis subsystem to derive structural and conceptual information about documents [FOXE 87], have not been included in the current CODER retrieval subsystem. Implementation of natural language query entry will prompt the inclusion of these specialists whose functions include extracting classifications (frames) from natural language queries and matching the classifications to document frames. Additionally, the query parser module is not needed until natural language queries can be analyzed.

The linguistic and cluster search specialists have not yet been incorporated. Current search strategies include p-norm, boolean and vector searching. A *query formulator* module has been added to formulate a searchable query from explicitly specified user data as well as from system information. The modules which receive and send information to the user interface manager, the *input analyst* and *report* modules respectively, have been added to clarify the flow of information to and from the user. Nonetheless, the overriding theme of distributed experts centered around a blackboard/strategist complex remains. Moreover, the modularity of the CODER system, designed as a research testbed, will permit rapid assimilation of modules to be added or enhanced later.

A complex, ambitious endeavor, the development of the CODER system has embodied the talents of many graduate students at Virginia Tech. Master's projects to develop the CODER communications functions [APTE 87a], to create Prolog facts from the **Collins Dictionary of the English Language** [WOHL 86] and to provide the initial user interface [KHAN 88] have aided the implementation of CODER. Qi Fan Chen, a doctoral student, has implemented most of the CODER analysis subsystem [FOXE 87]. Additionally, graduate student projects for Dr. Edward Fox's Information Storage & Retrieval class during Spring quarters, 1986 and 1987, offered insights regarding the implementation of several modules.

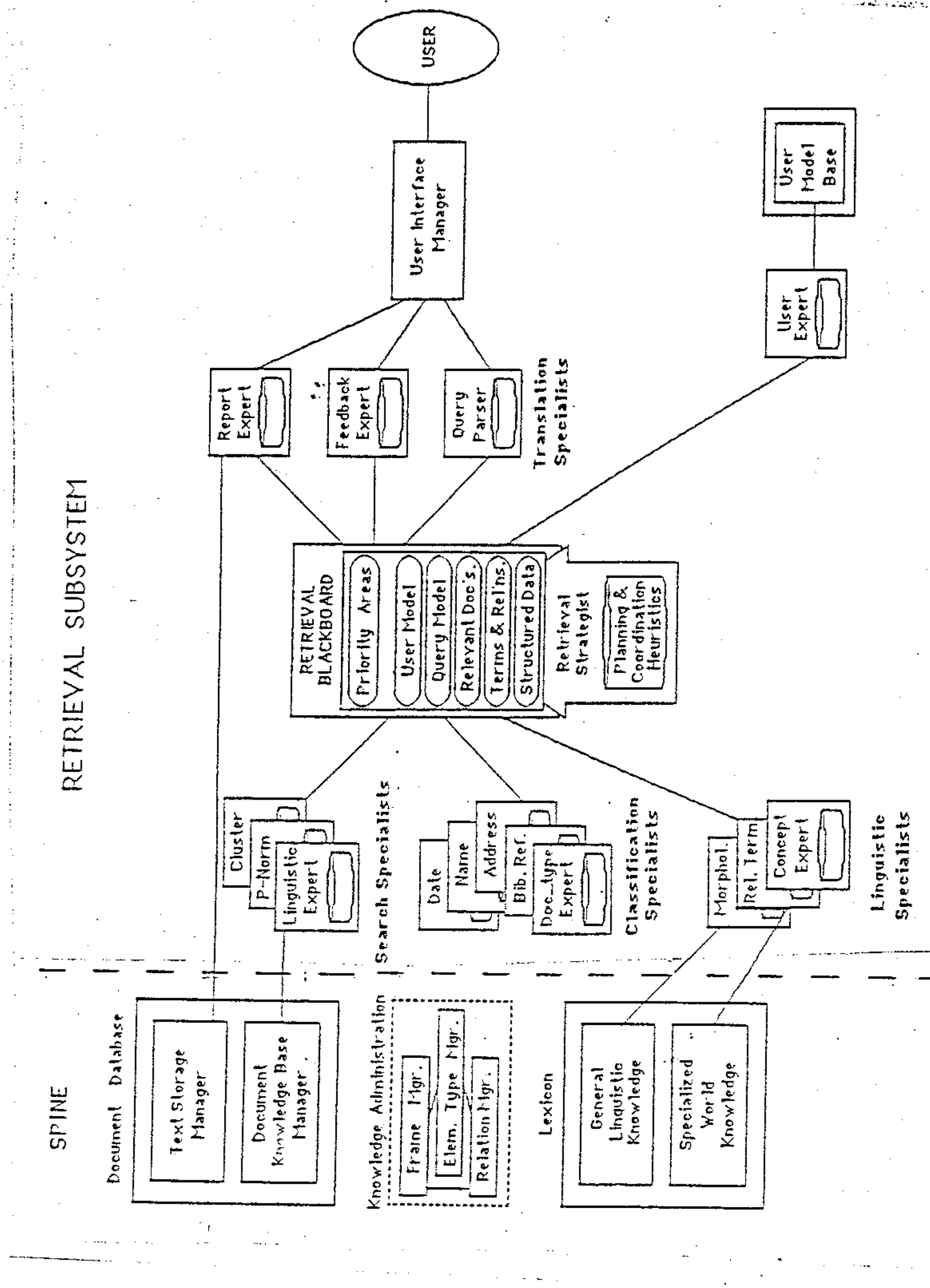
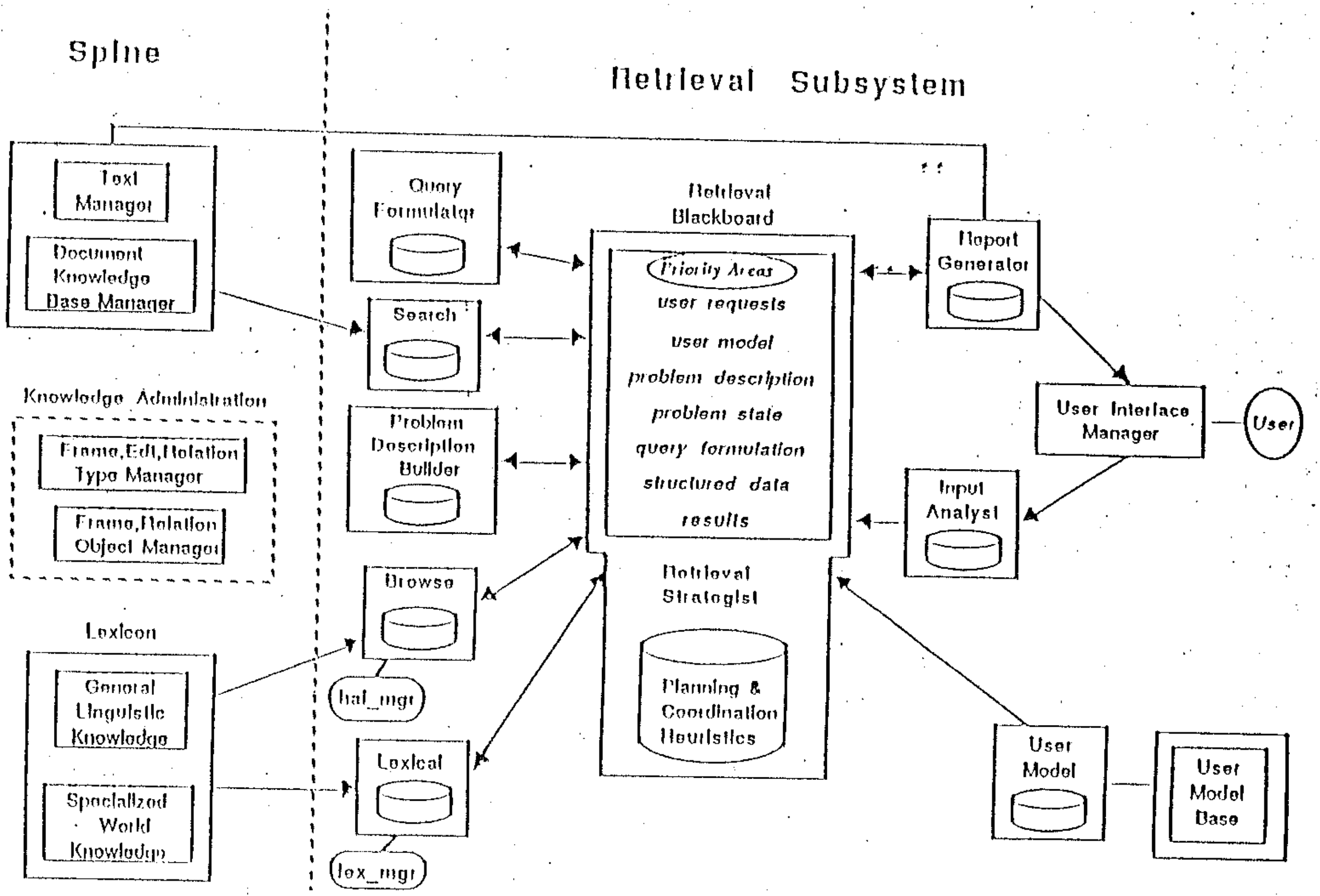


Figure 1. Original CODER Design



'The Knowledge Administration Complex is used by many modules.

Figure 2. Implemented CODER Design

1.3 Scope

This implementation of the CODER system addresses several hypotheses about the usefulness of AI methods in information storage and retrieval systems:

- *Logic programming is adaptable to information storage and retrieval.*
- *The knowledge engineering paradigm can be applied to information storage and retrieval systems.*
- *System modularity provides a more flexible research testbed environment.*
- *Users can perform more effective retrieval when structured knowledge is employed. The hierarchical organization of documents as well as concepts such as names, dates, and addresses represent structured forms of knowledge.*

Specifically, this research aims to prove that the logic programming language Prolog is suitable for the development of an intelligent information retrieval system; moreover, it demonstrates that system modularity simplifies implementation of an intelligent IR system, allowing participation of many individuals without hampering implementation efforts.

The research described here, carried out between January, 1986 and January, 1988, relates specifics of the following: development and testing of the knowledge administration complex, a frame system written in Prolog; development and testing of all retrieval subsystem inferential modules, also written in Prolog; integration of inferential modules with the blackboard/strategist complex; integration of inferential modules with non-inferential modules, written in C and hereafter referred to as **resource managers**; and testing and enhancement of communication between retrieval subsystem modules. The difficulties and highlights of the CODER retrieval subsystem implementation, a micro-level system evaluation and a synopsis of resulting accomplishments are included.

A review of relevant literature reflecting the current interest in artificial intelligence as it applies to Information Retrieval (IR) is provided in Chapter 2. The third chapter discusses technical details of the implementation: the computer systems and languages used, directories and files, communications and configuration issues and development and testing methods. Implementation specifics are provided in Chapter 4: prototype development stages are described and each of the retrieval subsystem modules is discussed in light of functions performed, callable predicates, system integration and heuristics adapted from information storage and retrieval literature. The final chapters summarize results and accomplishments. System performance, validation of original hypotheses, conclusions, and recommendations for future versions of CODER are addressed in Chapters 5 and 6.

2.0 Review of Related Literature

Knowledge is of two kinds. We know a subject ourselves or we know where we can find the information about it.
-- Dr. Samuel Johnson

Applications of Artificial Intelligence (AI) cross the boundaries of many domains. Similarly, Information Storage and Retrieval (ISR) systems are used in diversified areas, for example, medicine, law, electronic publishing, science and office information systems. Like the applications encompassed, the literature of AI and ISR is diverse and may be found in publications related to application areas as well as in documents written for AI or ISR researchers. Given the relative newness of AI in the field of information retrieval, most of the current literature regarding AI methods in ISR may be found in publications related to ISR systems. A general discussion of AI progress in information retrieval and relevant literature has recently been published [SMIT 87b].

This chapter examines publications containing research and ideas which can be compared and contrasted with those in the CODER system. Information and Library Science journals, conferences and textbooks are the source of the majority of the citations given. The focal points of this chapter have been narrowed to 1) related ISR systems, 2) frame representation systems, 3) blackboard architectures and 4) functions of an intelligent ISR system.

2.1 Related ISR Systems

The CODER system is not the first ISR system to be designed and/or implemented using AI methods. Experimentation with one or more expert modules aimed at simulating the performance of a trained intermediary has been carried out by other researchers for nearly a decade. Many have concentrated on particular functions of an intelligent ISR system: user interface issues [SMIT 87a, MALO 87, GOLD 78, BREN 81], natural language parsing and understanding [DOSZ 86, CHIA 87, GUID 83, METZ 85], query processing and search [CHIA 87, SALT 83a, 83b], knowledge representation schemes [PATE 84a, 84b, FICK 85, BRAC 83] or user modelling [DANI 86b, BORG 86, 87a, 87b]. Many *designs* for intelligent information systems have been proposed; however, few have been fully implemented.

In 1981, a Master's student at MIT proposed a single expert system module to aid inexperienced users of bibliographic retrieval systems in the search process [YIP 81]. Simple LISP-like production rules were suggested for knowledge representation and reasoning. Efforts at MIT have continued and expanded with the ongoing development of the **CONIT** system [MARC 85], an experimental study of heterogeneous system access through a common command-language (CCL) designed to aid the construction of Boolean queries. For example, the 1986 version of CONIT could handle three databases, Dialog, NLM and ORBIT, via its CCL. The proposed NISO standard language for information retrieval [NISO 87] has been incorporated into the CODER Retrieval subsystem as its "common command-language."

A prototype Prolog system, **PROBIB-S**, is being developed for online bibliographic retrieval [WATT 87]. The Canadian researchers involved examine the incorporation of special hardware and extensions to the Prolog language to overcome the problems inherent in Prolog searching of large databases. The CODER retrieval subsystem, also Prolog-based, has approached the concern with unacceptable search response times differently. First, a Prolog version which already includes extensions to handle large databases was located. Melbourne University's MU-Prolog 3.2db [NAIS 85], although it does not address special hardware, contains routines using superimposed coding for storage and access of large external databases [RAMA 85]. Secondly, adaptation of searching routines from the **SMART** system [BUCK 85, FOXE 83a, 83b], developed at Cornell University and written in C, has been undertaken.

Another Prolog-based document retrieval system, the **CANSEARCH** system [POLL 87], retrieves cancer-therapy-related documents from the MEDLINE database. The system acts as a front-end, but performs no actual searches. Instead, it provides a menu-driven interface for narrowing and selecting MeSH query terms which are then formatted into a query to be processed by the host computer at the National Library of Medicine. CANSEARCH achieves its objective of eliminating the user's need to know query formulation specifics; moreover, it demonstrates the practicality and usefulness of an "intelligent" front-end. However, its limited scope allows it to ignore some of the more advanced features of a retrieval system, such as user models, a natural language interface, term expansion or search strategies. Therefore, use of Prolog and a blackboard for message communication are the two noteworthy similarities between CANSEARCH and CODER.

The **MICROARRAS** system [SMIT 87a], a full-text retrieval and analysis system under development at the University of North Carolina, focuses on databases distributed over different hosts and well-defined user interfaces to support intelligent dialogues. Correspondingly, the CODER system uses TCP/IP protocols to handle databases and modules which exist on different hosts. The modular design of the CODER system simplifies experimentation with several different user interfaces, also a goal of the MICROARRAS system. Plans to develop a MacIntosh user interface for CODER in addition to the current UNIX *curses* interface are already underway. The two database search modes found in MICROARRAS, bibliographic and content, approximate CODER's Boolean/P-norm and structured knowledge search modes respectively.

The hypothesis CODER is being used to test, "Users can perform more effective retrieval when structured knowledge is employed" is also a premise of **The Information Lens** [MALO 87], a prototype intelligent information sharing system.

> *A rich set of semistructured message types (or frames) can form the basis for an intelligent information-sharing system. For example, meeting announcements can be structured as templates that include fields for "date," "time," "place," "organizer," and "topic", as well as any additional unstructured information.*

Based on studies of information sharing in organizations, the researchers have explored cognitive, social and economic *information filtering* designed to selectively filter the information that individual users really want to see, for example in electronic mail systems. As in the CODER design, the researchers involved in The Information Lens project have concluded that a frame inheritance lattice simplifies analysis and retrieval of messages. Analysis of document types in the CODER system is similar to the message type analysis suggested in this study.

Natural language access to information retrieval systems, one of the most promising applications of artificial intelligence to information retrieval, is the goal of many intelligent systems such as the **Natural Language Interface** (NLI) [BISW 87] conceptualized and prototyped at the University of South Carolina. **IR-NLI** (Information Retrieval - Natural Language Interface) [GUID 83], an Italian research project designed in 1983, is organized around "...semantics-directed and goal-oriented parsing". Its modules include 1) understanding and dialogue, 2) reasoning about the user's request and 3) a formalizer to formulate a searchable query. **IRES** (Information Retrieval Expert System) [DEFU 85, CHIA 87], developed in France, employs a separate query processing module called IOTA which includes morphological and syntactic analysis of the query. IRES also adapts to user characteristics and to the state of processing during a retrieval session. Like IRES, CODER pays special attention to the important distinction between domain and expert knowledge and the need for the two to remain separate entities; user modeling capabilities are also found in both systems. CODER does not yet include a natural language interface as does IRES, but does include a document analysis component.

Initial attempts to incorporate the **CHAT-80** [PERE 83] Prolog-based front-end natural language interface into the CODER retrieval subsystem [APTE 87b] indicated that implementation of natural language query processing should be handled as a separate graduate level project. Online versions of the **Collins Dictionary of the English Language** as well as the **Handbook of Artificial Intelligence** (HAI) allow users to 'mark' query terms or phrases during browsing so that they may be added to queries later. These same fact bases were designed to aid natural language understanding and will be used when the planned natural language interface is developed.

The **RUBRIC** system [TONG 86a, 86b], a commercial system based on production rules and written in CommonLisp, uses a manually built rule base to assist query construction and searching. Likenesses to CODER are noted in its object-oriented paradigm for expert systems, use of relevance values in the range [0,1] rather than in the set {0,1} and the availability of on-line help at any point in the retrieval process.

The distributed ISR system developed by Croft and Thompson [THOM 85, 87a, CROF 87] most closely resembles the retrieval portion of the CODER system. As in CODER, the architecture of **I^{3}R** (Intelligent Interface for Information Retrieval) is based on a group of cooperating experts centered around a blackboard/scheduler. User models, system state transitions, assignment of uncertainty values to rules and on-line help/explanation exist in both systems. Application of some of the research by Belkin, Brooks, Borgman and Daniels [BELK 83, 84, DANI 86b, BORG 85, 86, 87a, 87b] regarding the functions of an information retrieval system and user modelling strategies can be found in both systems. A user model builder, an interface manager, a search strategy module and a browsing expert perform parallel functions in the two systems.

Where CODER uses a Prolog frame representation system for representing documents and domain knowledge, I^{3}R, written in CommonLisp, uses a relational database to represent documents, and concept frames containing recognition rules to infer domain knowledge, mostly from the system's interaction with the user [CROF 87]. Differences in implementation of scheduling rules and expert system rule bases also exist. A separate expert for state transitions and problem description building can be found in CODER as well. Although neither system has yet had its planned natural language interface fully implemented, the CODER system has the advantage of an existing lexicon, frame-based representation of document text to allow structured knowledge matching and online availability of domain knowledge via the HAI.

2.2 Frame Representation Systems

In every intelligent ISR system, some formalism and corresponding notation with which to represent knowledge must exist. A variety of different knowledge representation (KR) schemes, for example, *logical, semantic network, procedural* and *frame-based* schemes have been studied [MYLO 84, BRAC 85]. Smith and Warner limit their discussion of knowledge representation schemes to those that are used in ISR systems [SMIT 84]. The CODER system has adopted a **frame-based** KR scheme to represent document, user and query knowledge. Frame-based systems are used to model entities where each entity is defined by a set of attributes called slots. A complex lattice structure defining the frame hierarchy may be found in a frame-based system. The standard terminology, methods and goals of generic frame-based systems [FIKE 85, HAYE 85, MINS 81] also apply to the CODER frame system.

During the past decade, a variety of frame-based languages have been developed. Such languages have included **KANDOR** [PATE 84a, 84b], **KRL** (Knowledge Representation Language) [BOBR 77], **KL-ONE** [BRAC 85], **Krypton** [BRAC 83, PIGM 84], **KEE** (Knowledge Engineering Environment) [NADO 87], and the languages in the **CYC** [LENA 86] and **TOPIC** [HAHN 86] systems. Several of the frame language developers have worked on more than one of the aforementioned languages, and commonalities among the languages may be found.

Many of the concepts in the CODER knowledge administration complex have roots in the Krypton, KRL and KL-ONE representation schemes. For example, Krypton's separation of two representation languages, a terminological one called **Tbox** and an assertional one termed **Abox** [BRAC 83, PIGM 84] are parallelled by CODER's segregation of **Type Managers** and **Object Managers** for processing frame definitions and instantiations respectively. The taxonomical structure of frames and strict inheritance found in KL-ONE and Krypton also exist in CODER. Most of the frame representation languages have been developed using some version of LISP, although frame-based languages written in PROLOG, like CODER, are beginning to appear [LEE 86, HUU 86]. One notable difference among the frame-based languages is their treatment of default values [NADO 87].

TOPIC [HAHN 86] and **ARGON** [PATE 84a, PATE 84b] are two knowledge-based text retrieval systems that use frame representation schemes. Like CODER, they have incorporated concepts from earlier frame systems; moreover, the representational structures of the frame languages used carry over into the ISR systems and influence their appearance and functioning. The frame system used in CODER is also evident in the capabilities of the system. Similarly, the blackboard architecture described in the next section influences the CODER system's functions.

2.3 Blackboard Architectures

In distributed systems, "blackboards" provide a central location for storage of messages passed among system modules. In addition, scheduling heuristics and control are centrally located within the blackboard architecture. The first blackboard system, the **HEARSAY-II** speech understanding system, was developed between 1971 and 1976 [ERMA 80]. Blackboard architectures since then have been adopted not only for ISR systems, but for varied scientific and problem-solving applications: in the HASP system for ocean surveillance [NII 82]; in CRYSALIS for use by protein crystallographers [TERR 83]; in TRICERO to monitor airspace [WILL 84]; in OPM to simulate human cognitive processes in planning [HAYE 79]; and in CODER, I^3R [CROF 87] and CANSEARCH [POLL 87] for information retrieval. Other blackboard systems, for example **BB1** [GARV 87, HAYE 84, JOHN 87], attempt to learn about their own behavior. Most of the fundamental concepts developed for the HEARSAY-II system are found in current blackboard systems. A comprehensive review of the framework of blackboard systems and the evolution of blackboard architectures can be found in [NII 86a, 86b].

Corkill reviewed issues of flexibility, efficiency and generality in blackboard architectures [CORK 87]. The **GBB** (Generic Blackboard) model that Corkill presents has implemented blackboard abstraction concepts in greater detail than is found in the CODER blackboard. Additionally, the ability to change the blackboard implementation without modifying knowledge sources is emphasized. Experiments with HEARSAY-II [ERMA 80] and DVMT (Distributed Vehicle Monitoring Testbed) [LESS 83] have demonstrated that the efficiency of the blackboard implementation has a significant effect where the interaction/computation ratio of expert modules is high. Currently in the CODER system, that ratio is low; however, as the retrieval subsystem becomes more fully developed, Corkill's conclusion may be tested in the CODER system as well.

Of the three blackboard characterizations presented by Corkill [CORK 87], the *unstructured blackboard*, the *general-purpose kernel* and the *customized kernel*, the CODER system most closely models a general-purpose kernel. When an unstructured blackboard exists, each module must worry about the entire retrieval process, selecting and maintaining appropriate messages contained in lists on the blackboard. At the other extreme, specialized mechanisms called kernels filter blackboard objects per module based on the application of the system to make module processing more efficient. However, changes to the system may also require changes to the blackboard structure when customized kernels are employed. As in a general-purpose kernel blackboard, the CODER blackboard "supports blackboard object retrieval based on the attributes of the objects" and "the application implementers retrieve objects by writing queries in the retrieval language." The blackboard object retrieval language used in the CODER system is described in Chapter 4, section 4.2.1.

Examinations of ***control strategies*** for distributed expert systems are also relevant to the CODER project. Control issues for conflict resolution in rule-based systems such as **ORBS** (Oregon Rule-Based System) are discussed by Fickas [FICK 85]. Agenda-based control strategies in ORBS are similar to those in the CODER strategist. However, neither automation of the strategy construction process nor dynamic alteration of scheduling rules exist in the CODER strategist. At a recent conference on Distributed Artificial Intelligence (DAI) systems, one of the roundtable discussions focused on issues of communication and control between agents [SRID 87]. Standard functions required by agents were suggested: Inform, Request to Do, Request to Send, Command, Reply, Acknowledge and No-acknowledge, Offer, Agree, Refuse, Accept, Bid and Propose. In addition, resource availability combined with the limitations on resource utilization was considered one of the most crucial concerns for designers of DAI systems.

Belkin et al. also discuss issues of control for distributed expert information systems [BELK 83, 84]. Based on their model of intelligent functions in an information retrieval system, they conclude that "a blackboard communication structure with modified distributed control seems optimum for system implementation." Comparing Belkin's work to Pollitt's CANSEARCH system, Sparck-Jones contends that "building the all-singing, all-dancing expert intermediary is a major enterprise." [SPAR 87]. Moreover, she suggests that control is a major problem with distributed systems.

2.4 Intelligent Information Retrieval Functions

One of the primary goals of the CODER system, as with any information retrieval system, is to create a representation of the user's problematic situation. Through observation and monitoring of human-computer interactions and user-intermediary dialogues, researchers have characterized the functions required in an intelligent information retrieval system. Early work identified 29 *search tactics*, moves made to further a search, and 17 *idea tactics* which foster new ideas or solutions to problems in information searching [BATE 79a, 79b].

Five years later, Belkin et al. introduced the concept of an *information provision mechanism* (IPM) [BELK 83, 84], the combination of the search intermediary and knowledge resource elements of an information system. They identified 10 functions of the IPM: problem state, problem mode, user model, problem description, dialogue model, relevant world builder, response generator, input analyst, output generator and explanation. These functions were specified in the **MONSTRAT Model**. Discourse analysis and observations of interactions between users and search intermediaries supported the previously derived functions [BELK 88, DANI 85] and led to identification of subgoals of those functions. Further research, also at the University of London, elaborated on the MONSTRAT model and its incorporation into the **PLEXUS** expert system for referral [BROO 85, 87, VICK 87]. In a counter-proposal, Borgman suggests that ″the generality of the model is both its virtue and its weakness.″ [BORG 87b]. She recommends that the broadness of the model be narrowed based on the domain of users and the domain of questions in the information system. The CODER retrieval subsystem has incorporated many of the ideas found in the aforementioned research (see Chapter 4).

One of the MONSTRAT model's ten functions, the **user model**, has received much attention recently. Borgman concludes that the user analysis tasks will offer the greatest challenge in the design of sophisticated information retrieval systems [BORG 87b]. Her research includes studies to assess the significance of a user's academic major as a variable that indicates information retrieval aptitude [BORG 86, 87a]. Daniels has augmented the user modeling function definition, identifying subfunctions as status, goals, background and experience, state of knowledge, and familiarity with IR systems [DANI 85, 86b]. Moreover, she concludes that a frame-type representation for the user modelling subfunctions is most appropriate. User model frames in the CODER system have been modelled after Daniel's subfunctions.

Study of *cognitive models*, the views that users have of themselves, others, the system and vice versa, have been suggested as an aid to developing user models [DANI 86a, CHEN 87]. The importance of the *user interface* and the variability of its use and effectiveness for different groups of users has also been examined [NORM 86, HOLC 85, TAGU 87, EGAN 87]. Research regarding the **acquisition** of implicit and explicit data about users has been incorporated into CODER [RICH 79, LOGA 86, FENI 81, BRAJ 87, KASS 87].

In conclusion, an abundance of literature has been collected and a broad spectrum of hypotheses and heuristics from that literature can be found throughout the CODER retrieval subsystem. Particularly, the work of Belkin, Borgman, Brooks and Daniels has been incorporated into the problem mode/state/description and user model modules; ideas from the frame languages developed by Bobrow, Brachman and Winograd are evident in the knowledge adminstration complex; Penny Nii's review of blackboard architectures provided valuable guidance during the development of the blackboard/strategist complex; and reports by Tong, Croft and Thompson regarding the development of the RUBRIC and I^3R systems allowed comparison and contrast between CODER and other distributed ISR systems. The contributions from these and other researchers are gratefully acknowledged.

3.0 Methods

Our amazing industrial development has been made possible by the vast accumulation of scientific knowledge and technical know-how, every single item of which is a result of someone's observing, thinking, and experimenting, that is, of research.
--Ebenezer E. Reid

This chapter describes the tools and methods employed to construct the CODER retrieval subsystem. Its secondary objective is to provide a micro-level view of the programming implementation of an intelligent IR system. First, the computer systems and programming languages used are discussed. Delineation of files, directory structures and databases is included. Second, communications and configuration issues not covered by a previous Master's project [APTE 87a] are presented. Finally, the process of implementing modules and integrating them with CODER's blackboard architecture is reported.

3.1 Systems and Programming

The CODER system is structured as a group of communicating objects running under UNIX™, using the TCP/IP protocol [LEFF 84]. Inferential modules are written in MU-Prolog; non-inferential programs are coded in C.

3.1.1 Operating System

The retrieval subsystem consists of a set of modules distributed around a blackboard/strategist complex. Modules communicate with the blackboard, and thereby with one another, via message passing using the client/server model [COFF 87]. UNIX-based systems often provide a protocol, TCP/IP, to support the client-server model; moreover a rich, productive programming environment has earned UNIX a reputation for its programmer-friendliness. Because it is written in C, the UNIX operating system is highly portable and easy to modify per particular system requirements [KERN 84].

The availability of several computers running UNIX operating systems supported the choice of UNIX as the operational environment for CODER. The majority of the system has been developed on a DEC VAX 11/785 running ULTRIX-32™ Version 2.0, a variant of 4.2 BSD UNIX. Early system modules, including the first CODER prototype, were programmed and tested on a SUN™ workstation, also running 4.2 BSD UNIX. Presently, the entire system resides on the VAX 11/785. Future versions of CODER running on an Apple Macintosh II™, also using TCP/IP, have been proposed.

3.1.2 Programming Languages

Research using an artificial intelligence environment as a testbed for ISR systems suggests that programs exhibiting some kind of intelligent behavior will be included in the system to model some of the functions of a professional search intermediary. AI programming languages have been developed to support and test AI concepts, such as manipulation of arbitrary symbols, list-processing and pattern matching. At present, the two most widely used AI languages are the functional language **LISP** and **Prolog** (PROgramming LOGic), a language based on first-order predicate calculus.

Invented by John McCarthy in 1958, LISP has been in existence longer than Prolog which was conceived by R. Kowalski at Edinburg in the early 70's. Although LISP is more prevalent in the United States, Prolog is more widely used in European countries. Furthermore, the Japanese have targeted Prolog as the language of their fifth generation computers [SIMO 83, MARB 85]. Dialects of both languages, including versions supporting compilation and/or concurrent programming, have evolved [DEER 85]. Since Prolog and C are used extensively in CODER, more specifics are given in the following sections.

3.1.2.1 Prolog

During the design phase of the CODER system, Prolog was selected as the language for coding of inferential modules. Several factors influenced this decision:

- The AI language used had to be capable of searching large databases within an acceptable period of time. A version of Prolog running under UNIX, MU-Prolog, includes extensions to handle large databases [NAIS 85].

- Availability of the C source code for the MU-Prolog interpreter allowed necessary in-house modifications to the interpreter.

- List-processing and pattern matching abilities suggested that Prolog would be more appropriate for natural language processing.

- Prolog is especially suited for problems that involve objects, particulary structured objects, and the relations between them [BRAT 86].

- Prolog clauses are similar to rules; therefore, rule-based programming style is encouraged.

- The CODER designers' affinity for Prolog also influenced the language selection process.

Software Engineering Considerations: Use of Prolog as the primary language for module development allowed rapid prototyping of modules and easy inclusion of rules to simulate intelligent functioning. A directory of over 100 common Prolog utilities was established (see Appendix A). Some were written during the course of this research; most were adapted from public domain libraries made available through Prolog Digest, an electronic mail digest, or from the growing supply of Prolog textbooks [CLOC 84, BRAT 86, STER 86, SCHN 87]. Minor modifications were required so that utilities written for other versions of Prolog would run under MU-Prolog.

Only one program documentation standard for CODER's Prolog programs has been adopted. All Prolog modules follow a standard for identifying the status and placement of predicate arguments. The standard, published by John Cugini in Prolog Digest [CUGI 86], appears in Appendix B.

Since all retrieval subsystem modules have been written and/or incorporated into CODER as part of this research, modules currently contain identically structured program heading documentation as well as similarly styled comments preceding predicate definitions. Rules for programming style and technique, suggested by Ivan Bratko in "Prolog Programming for Artificial Intelligence" [BRAT 86] have been followed. For example, program layout, spacing, indentation, clustering and ordering of clauses, appropriate use of comments and mnemonic naming conventions are consistent. Program clauses and procedures have been kept short. Care has also been taken to limit use of Prolog operators which are inefficient such as ***if-then-else, assert*** and ***retract*** and to avoid operators like ***cut*** and ***not*** which may produce unpredictable results.

Obstacles: The use of Prolog as CODER's AI language has not been without obstacles. Some of the difficulties encountered and resolved follow.

- Implementation of MU-Prolog ***external databases*** proved to be cumbersome and inefficient. Lack of documentation from the developers, incorrect documentation and difficulty assigning proper masks for superimposed coding of index arguments [RAMA 85] resulted in poor response times when external databases were accessed. A new version of MU-Prolog, NU-Prolog [THOM 87b], has improved external database facilities as well as the ability to compile Prolog programs.

- *Host memory limitations* prevented execution of CODER when more than approximately 10 Prolog processes were running concurrently. Two separate versions of MU-Prolog, a "small" and a "big", have been created [DATT 87]. *Prolog-s* contains a goal stack length of 75K and includes none of the external database facilities. The normal MU-Prolog contains a goal stack length of 500K and all database facilities. *Prolog-b* includes the external database facilities, but has a reduced goal stack size of 200K. Most of the CODER modules are small and can be processed with prolog-s. Prolog-b is required for modules such as the lexical expert which use the external database facilities. Predicates developed for message passing between CODER modules have been built into the prolog-s and prolog-b versions; thus, the DLOAD dynamic loading facility could be removed, further reducing memory requirements.

- *String length* limitations required further modifications to the Prolog interpreter. Frequently, messages passed by modules to and from the blackboard contain Prolog arguments which exceed the 199 character upper bound. String length has been increased to 32000 characters.

- Lack of *floating point* arithmetic in the selected version of MU-Prolog prevented computation of query-document similarity values in the range {0,1} when p-norm searching, a "softer" form of Boolean searching [FOXE 83a], was used. Floating point routines have been added to the MU-Prolog interpreter [DATT 87]; in addition, the forthcoming version of NU-Prolog is reported to contain floating point arithmetic. Incorporation of floating point routines required other modifications to modules. Dotted pairs, such as (A.B) used for frame and relation identification assignment, had to be rewritten as single items or as lists to avoid misinterpretation of the pair as a whole number, A, with decimal digits, B.

- ***Restrictions on the number of atoms*** allowed in the Prolog dictionary required modifications to large Prolog fact bases such as those associated with the **Collins Dictionary of the English Language** or the **Handbook of Artificial Intelligence**. Atoms, enclosed in single quotes, had to be replaced by character strings denoted by double quotes.

- Lack of compilation facilities for MU-Prolog requires reconsulting of each Prolog module every time the CODER retrieval subsystem was started. Although start-up as well as execution times are highly dependent on other system load factors, ***start-up*** frequently required over 5 minutes. Use of MU-Prolog's built-in *save* predicate, which stores an image of a Prolog state, reduced start-up time to less than 1 minute. However, several disadvantages arise from use of the save predicate: additional storage capacity is required to store the Prolog save states (from 1K to 4K per module), and modifications made to Prolog modules are not operational until modules are reconsulted and saved again. Compilation of Prolog modules would eliminate the need to use Prolog save states or to consult modules during start-up. The new MU-Prolog version, NU-Prolog, provides a compilation facility. Upgrading MU-Prolog to NU-Prolog should result in even lower start-up time, significant speed increases and elimination of the need to store Prolog save states.

- Prolog's treatment of ***special characters*** such as periods and single quotes demands extra processing for passed messages. Special characters passed between Prolog and C modules are converted so that data is processed and displayed properly. For example, single quotes within Prolog strings are passed as carats and are converted by the C resource managers.

- As an interpreted language, MU-Prolog is inherently ***slow when searching*** large local fact bases. A system-defined predicate, *clindex*, allows a group of static facts with the same functor to be indexed by a given argument. Based on some simple performance tests comparing search response times with and without the clindex clause, all CODER static local fact bases with over 20 facts are indexed with the clindex predicate to speed searching during computation.

3.1.2.2 The C Programming Language

Where modules are not required to exhibit intelligence, that is, to reason inferentially, the C programming language has been used.[1] Like UNIX, C is highly portable. Furthermore, "its absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages." [KERN 78]. Because the C programming language was originally written for the UNIX operating system, C programs execute efficiently under UNIX and can easily access standard UNIX routines which are also written in C.

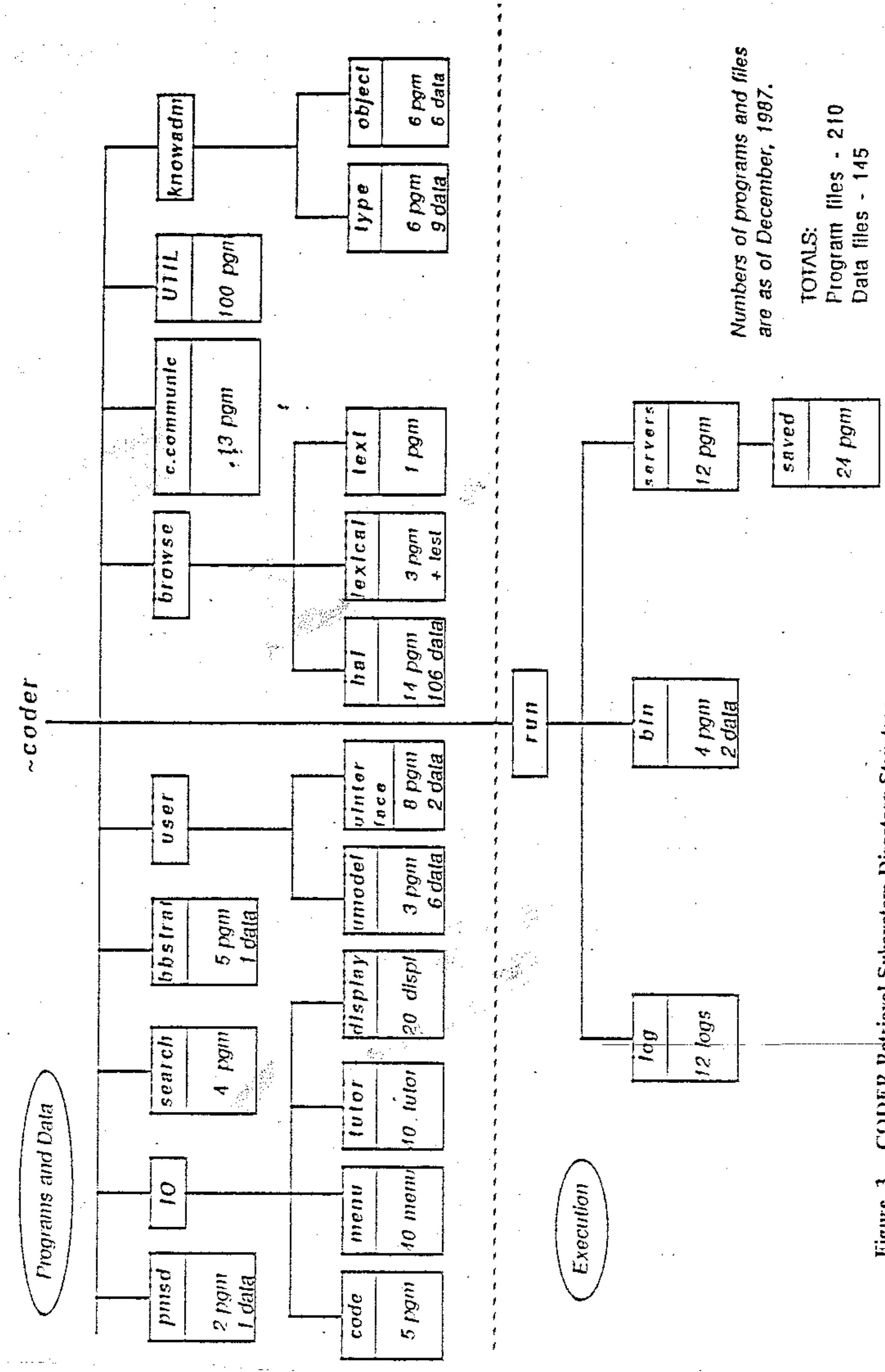
CODER's resource managers have been developed using C. Resource managers include the user interface manager and managers to extract text from the HAI and the document collection. Communications routines and modifications to the Prolog interpreter have naturally been coded in C also. Thus, the entire retrieval subsystem has been programmed using MU-Prolog, C, and UNIX shell commands. The location of CODER's program source code is discussed in the next section.

3.1.3 Directories, Files and Databases

Directories: The majority of the programs and files used by the CODER retrieval subsystem reside in user directory *~coder*. Other directories include the *~muprolog* directory for Prolog interpreter files and programs, the */tmp* directory used during CODER execution for storage of all temporary files, and the *~isr* directory which stores the lexicon facts derived from the Collins dictionary.

As illustrated in Figure 3 the coder directory may be viewed as having two distinct levels. One level contains all *program and data files*. The programs and data in the subdirectories at this level may or may not be used during a retrieval session. Programs at this level may exist for printing/auditing

[1] Although the original design of CODER suggested that the *C+ +* language be used, non-inferential modules have been coded in standard *C*.



Numbers of programs and files are as of December, 1987.

TOTALS:
Program files - 210
Data files - 145

Figure 3. CODER Retrieval Subsystem Directory Structure

of CODER files or for the creation of ancillary files such as those containing frame definitions. The other level of the directory structure contains ***execution*** modules and files. Many of the modules at this level use program components and/or data from the aforementioned level. Configuration files, run-time checking, execution modules and logged run-time output are stored in the execution directories. Following is a slightly more detailed synopsis of the CODER subdirectories and their contents.

Program and Data Subdirectories: To simplify testing of modules, subdirectories have been created to store both the programs and data that support specific functions in the CODER system; for example, all programs and fact bases which support the problem mode/state/description module are stored in the *pmsd* subdirectory. In total, 190 files, including the 100 program files in the Prolog utilities library, contain program source code for both Prolog and C modules. Approximately 20 files contain program object code used when linking C modules. Due to software engineering considerations which led to modular development of source code, the number of files containing object and source code is far greater than the number of CODER modules: multiple objects may be linked to form an executable C module, and Prolog modules consult multiple files. Approximately 145 data files contain Prolog fact bases, textual data or logged data, and another 100 files contain menus, tutorials and files containing information to be displayed to the user. Statistics regarding the sizes of the modules are provided in Chapter 5, section 5.1.2.

bbstrat This directory contains the blackboard module and all components of the strategist including its scheduling rules.

browse The browse directory is divided into three subdirectories: lexicon, hai and text. The subdirectories contain most of the programs, facts and text files needed to browse the **Collins Dictionary of the English Language**, the **HAI** and the **documents** in the collection being searched. These subdirectories are the largest in the CODER retrieval system. For example, the hai subdirectory consists of 106 files containing the 3,984K bytes of HAI text, and 10 files consisting of 293K bytes of Prolog facts representing key terms and relationships in the HAI. The majority of the lexicon facts, used in other research projects as well as in CODER, are stored in a user directory called *isr*. However, programs and facts used only by CODER for browsing the dictionary are stored in the lexicon subdirectory of the browse directory.

c.communic Special subroutines and files are required to create C programs which can communicate with the rest of the CODER system via the message passing functions built into special versions of MU-Prolog. Generic versions of those subroutines and files are stored in the c.communic directory.

IO This directory contains programs and data used for input/output between the user interface manager and the rest of the CODER system. Its four subdirectories contain menu files, tutorial (help) files, display files and program code for the input analyst and report modules. The IO directory and its subdirectories contain nearly 100 files.

know_adm The knowledge administration directory consists of the programs and files used to define the knowledge representation types used in the CODER system. Subdirectories for *type* and ***object*** managers, discussed in more detail in section 4.2.2, reside in this directory. The type definition files, with the exception of those defining user frames, can be found in the type subdirectory.

pmsd The problem mode/state/description directory contains the programs and fact bases needed by the problem description module.

search The p-norm, vector and boolean search routines as well as the query formulation module reside in this directory.

user The user directory contains two subdirectories: one to support the user modeling function and another containing the files and code needed by the user interface manager.

The buffer file for saving browsed terms to be used during query formulation and the prompt file, both used by the user interface, reside in the user interface subdirectory.

UTIL This directory contains over 100 commonly used Prolog utilities.

Execution: Programs, files and shell scripts needed to run the retrieval subsystem are in the ~coder/run directory.

bin This directory contains the UNIX shell script code needed to execute the CODER system. Files used to determine which modules are to be included and the hosts on which they reside are found here.

log When the retrieval subsystem is started, output from each module is directed to the log directory; therefore, this directory contains one file per CODER module. Files are assigned the same names as the modules whose output they contain; for example, the umodel file in the log directory contains all logged output from the user modeling module. Files in this directory should be reviewed periodically by the CODER system administrator since system run-time warning messages and/or error diagnostics will appear here. These files are an invaluable aid during testing and debugging of integrated system modules.

servers Object code for C programs and MU-Prolog modules to be consulted during start_up currently reside in this directory. Standardized location of all Prolog and C modules greatly simplifies system start-up. A subdirectory within servers, **saved**, contains the Prolog save states for any modules which have been saved. Thus, the modules in the server directory may contain any one of the following:

1. Dynamic segments of Prolog code to be reconsulted. More stable portions of code may be saved in a save state to speed loading of the module during system start-up.

For example, tested segments of code may be loaded from a save state while newly developed code may be reconsulted during each system start-up so that as changes are applied, the full module does not have to be reconsulted. This option simplifies testing during development stages when code is frequently modified. Note that when the map file (see section 3.2.2) indicates that a Prolog save state is to be used, the save state is loaded *and* any code in the module's entry in the servers subdirectory is reconsulted.

2. All Prolog code, but no fact bases as facts may be saved in the save state. For example, the files containing the Prolog facts representing the contents of the **Handbook of Artificial Intelligence** are saved in a save state while the code for the expert which accesses the facts is reconsulted during system start-up.

3. All Prolog code *and* all fact bases, since a saved version of each module is not mandatory. When modules and their corresponding fact bases are relatively small, use of a save state is not necessary.

Full program code for CODER modules always resides in the appropriate directory at the previously described program and data level. Implementation of NU-Prolog's compilation facilities will make the saved directory unnecessary; instead, object versions of all modules will reside in the servers directory.

Files: The data files used by the CODER retrieval subsystem contain Prolog fact bases, textual data, menus, tutorials and other information to be displayed by the user interface manager. Prolog fact bases will be detailed in later sections. However, design and use of the files used by modules to send data to the user merit additional explanation. Menu, tutorial, prompt and display files will be discussed briefly in this section.

Many of the files are needed by the user interface manager which has purposely been designed as an unintelligent resource manager; its callable functions have no awareness of the CODER system and its modules. Rather, the user interface accepts commands from the report module to display prompts, menus, messages or other files. It performs the function requested, may optionally send a user response to the input analyst and then waits to receive the next command issued to it. This design, depicted in Figure 4 has mandated standardization of menus and prompts to be issued by the user interface manager.

Menus: The initial version of CODER relies heavily on menu-driven processing, largely as a result of its unsophisticated user interface. When modules require that the user select one of a given number of options, the module may request that a menu be displayed. The request is initiated by posting the ***disp_menu*** hypothesis along with the menu name to the blackboard. Each menu is stored in a separate file in the menu subdirectory. All menu file names are prefixed by 'ui_menu' and adhere to the format shown in Figure 5.

For each menu requested, the "options" appear in the designated window; the user may be instructed to scroll forward to view options that do not fit in the window requested. Following user selection of an option, the "predicate_to_issue" is passed from the user interface manager to the "module_to_receive_response." Currently, all user responses are sent to the input analyst (ia) module. However, this parameter is provided for experimentation with future versions of CODER which could allow user responses to bypass the input analyst, and instead be sent directly to appropriate modules. A sample menu file appears in Figure 6. Eight options are listed under the heading *Browse Related Subjects.* When the user selects one of the options, the user interface manager sends the predicate listed at the end of the option line to the module listed, in this case the input analyst (ia). For example, if the user chooses option 3, the input analyst receives the hai_req(find_hierarchy) predicate, converts it to an internal system form and posts it to the blackboard. The first five options are posted as requests to search the **Handbook of Artificial Intelligence** using the given subject relations; the last three options result in display of previous menus.

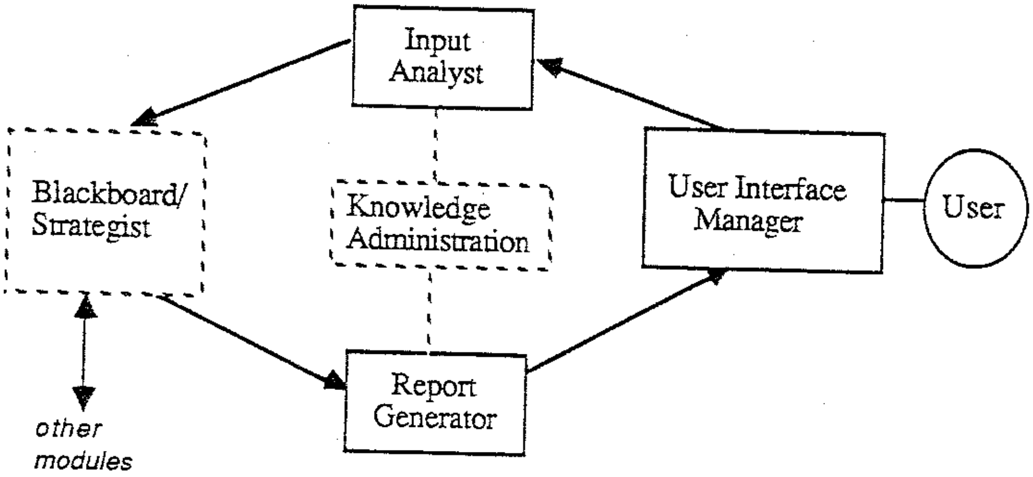
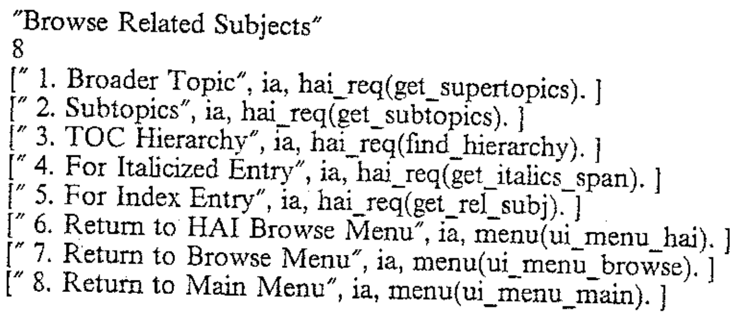


Figure 4. Generic User Interface

```
"Menu header"
Number_of_options
["1.  Option 1", module_to_receive_response, predicate_to_issue]
["2.  Option 2", module_to_receive_response, predicate_to_issue]
["3.  Option 3", module_to_receive_response, predicate_to_issue]
["4.  Option 4", module_to_receive_response, predicate_to_issue]
          etc.
```

Figure 5. Menu File Format



```
"Browse Related Subjects"
8
[" 1. Broader Topic", ia, hai_req(get_supertopics). ]
[" 2. Subtopics", ia, hai_req(get_subtopics). ]
[" 3. TOC Hierarchy", ia, hai_req(find_hierarchy). ]
[" 4. For Italicized Entry", ia, hai_req(get_italics_span). ]
[" 5. For Index Entry", ia, hai_req(get_rel_subj). ]
[" 6. Return to HAI Browse Menu", ia, menu(ui_menu_hai). ]
[" 7. Return to Browse Menu", ia, menu(ui_menu_browse). ]
[" 8. Return to Main Menu", ia, menu(ui_menu_main). ]
```

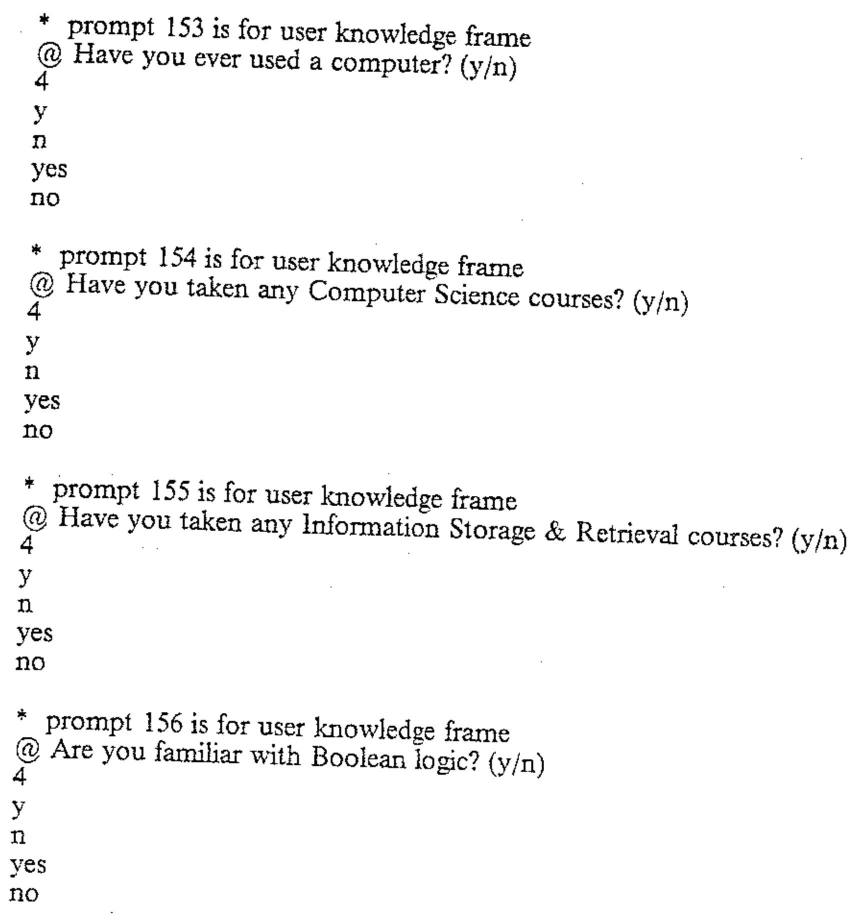
Figure 6. Sample Menu File

The disp_menu(X) hypothesis posted on the blackboard causes the strategist to schedule a task for the report module. The report module 'asks' the user interface to display the indicated menu to the user. When an option is selected, the "predicate_to_issue" is channelled to the input analyst where it is converted to an appropriate hypothesis to be posted to a blackboard area. Following posting to the blackboard, the strategist dispatches the expert(s) capable of processing the menu option selected.

Prompts: Unlike menus, all prompts are stored in a single file called prompts.dat in the ~coder/user/uinterface subdirectory. The location of certain prompt number ranges on the prompt file is critical as ranges may be stored in facts used by the input analyst to determine how to post user responses to the blackboard. For example, prompts numbered 150-200 are assumed to be responses to user modeling questions while prompts between 100 and 150 are channelled to the problem description module. Specific prompt numbers are also used by the input analyst module. For example, prompt 50 identifies user entry of a NISO command; prompt 51 identifies user entry of a term for lexicon browsing. Certainly, prompts may be modified, added or deleted; however, the person implementing such changes must be aware of the impact on other modules in the system.

Prompt numbers are also used by modules to associate user responses to questions. For example, when the user modeling expert is scheduled to process the message from the input analyst indicating that the user's response to prompt 153 "Have you ever used a computer?" was 'y', its local fact base tells it that this response will fill the frame slot called *usedcomp* in the user *knowledge* frame.

The current user interface requires that prompts be ordered by prompt number. Lines beginning with an asterisk, '*', represent comments, while lines beginning with the '@' sign indicate the start of the prompt string to be displayed. Following the prompt string, the number of valid options and those options, if any, appear. Figure 7 contains a portion of the prompt file.



```
* prompt 153 is for user knowledge frame
@ Have you ever used a computer? (y/n)
4
y
n
yes
no

* prompt 154 is for user knowledge frame
@ Have you taken any Computer Science courses? (y/n)
4
y
n
yes
no

* prompt 155 is for user knowledge frame
@ Have you taken any Information Storage & Retrieval courses? (y/n)
4
y
n
yes
no

* prompt 156 is for user knowledge frame
@ Are you familiar with Boolean logic? (y/n)
4
y
n
yes
no
```

Figure 7. Sample Portion of the Prompt File

It is recommended that future versions of the user interface simplify prompt processing by adding parameters similar to those found in menu files. That is, the name of the module to receive the prompt response and the predicate to be posted should be attached to each prompt.

Display and Tutorial Files: Files to be displayed to the user reside in the ~coder/IO/display subdirectory. These files, for example the CODER welcome and exit files, are unstructured and may contain any type of data. Similarly, the CODER help files in ~coder/IO/tutor do not follow a homogeneous format. Display of tutorial/help files may be activated when the user chooses the *Tutorials* option from the main menu, or when the user enters the ***help*** command at any point during the retrieval session.

Databases: Several existing databases also aided the development of the retrieval subsystem.

1. The **Collins Dictionary of the English Language** containing roughly 85,000 headword entries has been provided in machine readable form. Lexical entries were analyzed and converted into Prolog facts; twenty-one Prolog relations have been extracted [WOHL 86]. The lexicon is available to the user for browsing and is also used for query term expansion. When natural language query processing is added, the lexicon will be used for morphological, syntactical and semantic analysis.

2. The staff of the SUMEX project at the Stanford University Medical Center has provided the machine readable version of the **Handbook of Artificial Intelligence**. The three volumes of text were analyzed and Prolog facts containing key terms, phrases and names with references to text passages were created [WENB 86]. The HAI is presently used only for browsing. However, domain knowledge provided by the HAI may eventually be used to expand user queries.

3. **AIList Digest** issues have been collected and stored at Virignia Tech. An archive of issues, from the digest's inception in April 1983 until present, contains approximately 8000 messages.

The raw text, an inverted index of terms and a database of frames and relations generated by the CODER analysis subsystem form CODER's document database.

4. The **User Model** database will consist of a set of frames and relations for each user of the CODER system. The frames contain information about the user acquired both explicitly and implicitly. The user modeling module uses the knowledge administration complex to build and maintain user frames and relations. This database will be constructed and expanded as the CODER retrieval system is used. The information contained within will enable evaluation of system effectiveness and efficiency and will allow the system to tailor its processing to individual users.

3.2 Communications and Configuration

Programming of most of CODER's communications predicates and routines was completed in June 1987 as part of a Master's project entitled "Communication in the CODER System" [APTE 87a]. Although an abbreviated overview of the communications layer supporting CODER will be provided, this section will not rediscuss sockets, TCP/IP, deadlock handling or the C functions used to provide prolog-to-prolog message passing within the CODER system. Rather, it will describe enhancements which were made as part of this research: revisions to the communications programs, newly developed communications for message passing between C and Prolog programs, start-up and termination of the retrieval subsystem and instructions for modifying the CODER configuration.

3.2.1 Communications Overview

Given limited machines with limited computational and storage capacities, the number of modules, the need for real time operation and the amount of knowledge required by the CODER system present implementation difficulties. Therefore, the communications layer supporting the CODER system was designed to allow modules to reside on different host computers. Thus, where large external knowledge bases must be accessed, modules may reside on machines having adequate storage capacity; inferential modules may execute on machines with sufficient computational power; and the user interface could reside on a separate workstation with bit-map display capabilities.

Modules written in either Prolog or C can communicate with any other CODER module on the same host or on a connected host supporting TCP/IP. The location of each module is transparent to CODER implementors as well as to the user. During system start-up and execution, a **map** file is consulted to determine the location of modules. The type of module, Prolog or C, is also indicated on the map file.

The client-server model allows asymetric communication between CODER modules. For example, all inferential modules act as **servers**, waiting for messages from the blackboard/strategist indicating actions to be taken. Modules also act as **clients** by sending messages to resource managers or to the blackboard/strategist. Queuing of messages to servers allows concurrent requests to modules or requests to modules engaged in previously scheduled tasks.

In order to pass messages between inferential modules, several predicates have been built into the MU-Prolog interpreters, prolog-s and prolog-b. The ***start_service*** and ***stop_service*** calls, required for initiation and termination of sockets, are used during the start-up and termination of the system. The ***reply*** and ***receive*** functions, transparent to module implementors, are used in conjunction with the ***ask*** predicate, the primary vehicle for communication among Prolog modules. When the

CODER system is configured, all servers enter a loop in which they repeatedly ***receive*** clauses, prove those clauses and ***reply*** back to the module which issued the ask.

The ask predicate is coded into the Prolog modules as:

ask(Module,Clause).

where Module is the name of any other CODER module for which a socket has been created using the start_service call, and Clause is any Prolog clause with instantiated and/or uninstantiated variables. The module receiving the clause attempts to prove the clause and to instantiate any unbound variables. If all arguments are instantiated, the clause is returned immediately to the module which issued the ask. When uninstantiated arguments exist, the clause is returned after it is proven and uninstantiated arguments have been bound. The communications predicates built into the MU-Prolog versions [APTE 87a] support message passing between Prolog modules. However, minor enhancements to those predicates as well as creation of routines to support communication between Prolog and C modules were required before the retrieval subsystem could be implemented.

3.2.2 Enhancements

As part of this research, modifications have been applied to the CODER **map** file used to index modules according to the hosts on which they reside. Originally only Prolog modules were included, all modules were assumed to run under standard MU-Prolog and no Prolog save states were used. Thus, the old map file contained two fields per inferential module: the name of the module and the host on which it resides. A third field has been added to allow inclusion of modules written in C and to indicate the version of Prolog under which the inferential modules should operate.

The newly added field is used only during system start-up to identify the type of environment to be used for CODER modules. For example, Prolog modules may use either the prolog-s or prolog-b version of MU-Prolog. The code also indicates whether a Prolog saved state will be used.

All modules other than the user interface manager are executed in background mode. The added codes and a sample map file appear in Figure 8. All consulted modules and C module object code reside in the ~coder/run/servers directory. Where saved states are employed, modules must have been previously saved using the same version of the interpreter indicated in the map file.

Other problems concerning the communications layer resulted in minor modifications. These changes are recorded here for the benefit of persons working on future versions of CODER.

- The directory structure defined for the first CODER prototype [APTE 87a] executed in January, 1987 has been streamlined. The new structure is described in section 3.1.3.

- System modifications/upgrades to the ULTRIX operating system caused incorrect addressing of sockets. Such problems, once identified, were easily rectified. Should sockets malfunction in the future, operating system modifications should be considered as a possible source of error.

- Socket names must be carefully assigned to server modules. Although nearly any name may be assigned to a UNIX socket, consideration must be given to Prolog's handling of particular names. Sockets originally called *time, name* and *user* had to be renamed. The MU-Prolog reference manual [NAIS 85] contains a list of Prolog reserved words.

- The server looping routines built into the CODER Prolog versions were modified so that backtracking stacks were not saved when modules loop continuously awaiting clauses to be proven.

- The server looping routines were further modified to prevent termination of modules which were asked to process clauses they could not prove.

In addition to the enhancements made to the existing communications functions, facilities were created to allow message passing between C and Prolog modules.

Code	Language/ Interpreter	Comment
c	C	resource manager written in C
s	prolog-s	consult module using prolog-s
p	prolog-s	use saved state plus consult using prolog-s
b	prolog-b	consult module using prolog-b
q	prolog-b	use saved state plus consult using prolog-b
f	C	execute C program in foreground

Sample Map File:

```
bboard          host1   s
strategist      host1   s
browse          host2   q
lexical         host2   q
ia              host1   p
report          host1   p
know_adm        host1   b
probmsd         host1   p
search          host1   b
user_interface  host1   f
hai_mgr         host2   c
qform           host1   s
```

Figure 8. Configuration Map File Codes and Sample Map File

3.2.3 Prolog to C Communications

To pass information between Prolog and C modules requires additional C functions. Implementation of a C module as a CODER client/server requires extensions to a standard C program. The tools provided to incorporate a C module into CODER are described in this section.

Four segments of code and a '.h' include file must be copied from the ~coder/c.communic directory to the working directory to implement a C program as a client/server, those segments of code must be integrated with the C program being implemented and some must be compiled with parameters determined by the new C program. The code required includes:

comm_funcs.c TCP/IP communications functions such as *servent, protocol, receive* and *reply*; these functions are used to start the C module socket and to send and receive data between the C module and Prolog modules;

srv_main.c the 'C' program driver to be adapted by compiling it with a parameter identifying the socket name assigned to the module;

srv_proc.c interprets argument strings and makes function calls, returning previously uninstantiated variables with values;

C program the 'C' program code which includes all functions needed;

srv_def.h definitions to be included in srv_main.c, srv_proc.c and the 'C' program.

To integrate the extensions with the C program, code modifications and/or compilations must occur for each of the four segments of code. Figure 9 contains instructions for adding the communications extensions to a C module. There are a few additional considerations.

- At present, a maximum of 10 arguments is allowed for a function.

- It is best if only 1 argument is uninstantiated.

A sample session testing the result of adding the communications extensions to a C module appears in Appendix C.

comm_funcs.c compile (cc -c comm_funcs.c) OR copy comm_funcs.o into your working directory.

srv_main.c compile as follows:

```
cc -c -DSNAME=\"server_name\" srv_main.c
```

where
DSNAME is set to the name of the socket.
e.g.,

```
cc -c -DSNAME=\"hai_mgr\" srv_main.c
```

srv_proc.c compile as follows:

```
cc -c -DNBRFCNS=integer srv_proc.c
```

where
DNBRFCNS indicates the number of callable functions in the C program.
e.g.,

```
cc -c -DNBRFCNS=4 srv_proc.c
```

C program

1. should not have a "main()" routine (main is in srv_main.c)
2. should include srv_def.h
3. should have added to program code, from ~coder/c.communic:
   a. srv_table: must make entry for callable functions in PROC(...) lines; the PROC table identifies any functions in the C program which may be called by other CODER sockets.
   b. srv_hello: must enter socket name in line of code containing
      servername = ...
      The hello function is used by the CODER configuration manager to insure that all modules in the configuration map file have been properly started.
4. should have callable functions with:
   a. function type: static int
   b. function arguments defined as type PPTRC (defined in srv_def.h)
   c. return of an integer value indicating the number of arguments
5. should be compiled using: cc -c cprogramname.c

When all 4 portions of C code have been compiled, they can be linked...

```
cc -o socketname cprogramname.o srv_main.o srv_proc.o comm_funcs.o (...other objects)
```

Figure 9. Adding Communications Extensions to C Programs

With development of the communications layer for C and Prolog message passing, nearly all tools are available for integration of CODER modules. The final mechanisms required are the retrieval system start-up and termination processes.

3.2.4 Configuration, Start-up and Termination

Before executing particular configurations of the CODER system, the *map* file must be edited. As discussed earlier, the map file determines the host location and program environment for each module. Addition of new modules or host relocation of modules may be effected by simple edits to the map file. When CODER modules reside on different host computers, the map file must exist on each host so that the TCP/IP communications programs know where to locate sockets. Presently, the map file must be manually entered/edited on each host.

In the first CODER prototype, the retrieval system was started using two computer terminals: one to start modules as background processes and another to send commands to the modules from Prolog via the ask predicate. To verify that all system modules were ready and had been consulted, a client connection check was entered for each module from the second terminal until modules responded as ready. System termination was later performed manually from the second terminal as well. A UNIX configuration shell script has since been coded to automatically start the system, execute the retrieval session and terminate the CODER processes, all from one terminal. If modules reside on different hosts, the host on which the user interface manager resides is the one from which the configuration shell script must be executed.

To initiate a CODER retrieval session, the *config* shell in ~coder/run/bin must be executed. Based on the map file, a Prolog process is created for each server executed in background mode and output from that process is directed to the log directory. C modules other than the user interface manager are also started in background mode. Next, a Prolog program uses the map file to determine when all modules have been consulted. This step is necessary because user entries via the user interface manager cannot be properly processed until Prolog modules are ready to accept messages from other modules. When all Prolog modules are ready, the user interface begins execution in the foreground by displaying the CODER welcome. At the same time that the welcome is displayed, the system establishes the system state as welcome and awaits the state transition welcome_done. The retrieval session is now underway.

When the user has finished the retrieval session, a final display file will appear, thanking the user. After display of the exit file, the user interface manager will use the map file to send the ***stop_service*** message to all other modules. The user interface then exits, terminating its own execution. Finally, all temporary files created during the retrieval session are deleted. The CODER retrieval session is ended.

3.3 Module Development and Integration

The tools discussed thus far have included the operating system, programming languages, files and communications functions. Before development and integration of other modules could proceed, the knowledge adminstration (KA) complex had to be coded. The KA programs support the creation of elementary data types, frames and relations [WEAV 86c] and are described in Chapter 4, section 4.2.2. These programs were required early in the CODER development so that modules could build and maintain knowledge bases of frames and relations. Following implementation of the knowledge administration complex, other modules were developed, tested and integrated with the rest of the CODER system.

3.3.1 Development and Testing

Both Prolog and C modules were first developed and tested as stand-alone programs. In Prolog modules, the *ask* was simulated by using standard MU-Prolog which does not include the built-in ask predicate. The ask predicate cannot be simulated using prolog-s or prolog-b because as a built-in predicate it is protected by the interpreter and cannot be redefined. By including the module asked with other consulted files, the ask could be simulated as follows:

ask(Module,Clause):-
Clause.

For example, when developing the report module, calls to the user interface manager could be simulated by including the above predicate and a few lines of additional code to simulate the *window* predicate issued by the report module to the user interface manager. Figure 10 contains an example of code that could be used to simulate the integration of the user interface manager with the report module. Clauses to be proven by the blackboard, for example

ask(bboard,post_hypothesis([Fact,Conf,Hyp,Expert,Depend],Area)),

were also simulated. C programs included a *main* routine for testing of C functions. Some C programs, for example the user interface, were prototyped in Prolog before being coded in C.

3.3.2 Integration

Prolog: To integrate a stand-alone Prolog module into the retrieval subsystem, Prolog goals must be added to the module so that calls from the blackboard/strategist may be received and processed. Furthermore, the module must notify the blackboard when it has finished the processing which was scheduled. A generic set of Prolog functions designed for modules to receive scheduling calls from the strateigst has been developed. Those functions appear in Figure 11 and Figure 12. They must be added to the new module and tailored to include entry of the expert's name as indicated in calls to the blackboard. Callable goals must also be added to the *process_goal* function listed.

```
ask(Module,Clause):-
    Clause.

window(Window,Header,File_or_prompt,Option,Clear_flag):-
    write('Window is '),
    writeln(Window),
    writeln(Header),
    disp_option(Option,File_or_prompt).

disp_option(0,_):-
    writeln('Window being cleared').

disp_option(1,Filename):-
    write('Display of '),
    write(Filename),
    writeln(' requested:'),
    more(Filename).

disp_option(2,Menu):-
    write('Display of '),
    write(Menu),
    writeln(' requested:'),
    more(Menu).

disp_option(3,Error):-
    write('*ERROR* ),
    writeln(Error).

disp_option(4,Editfile):-
    write('Edit of ')
    write(Editfile),
    writeln(' requested:'),
    more(Editfile).

disp_option(5,Prompt):-
    write('Prompt ')
    write(Prompt),
    writeln(' requested:').

disp_option(6,_):-
    writeln('Do nothing in this window').

disp_option(7,Msg):-
    writeln(Msg).

disp_option(Option,Data):-
    write('*Invalid* option '),
    writeln(Option),
    write('for Data '),
    writeln(Data).
```

Figure 10. Simulated Integration of User Interface with Report Module

When the strategist dispatches a module to perform a task, that task will be either **attend_to_area** or **attempt_hyp**. The functions listed allow modules to recognize either of those tasks, view the indicated area of the blackboard, call appropriate local procedures when goals exist for hypotheses, inform the blackboard when specific hypotheses have been processed and notify the strategist when all processing has been completed.

In addition to the integration code required in the Prolog module, two of the components of the strategist must be modified. Note that the strategist consists of a total of four components: the ***domain task scheduler*** for determining which modules are to be scheduled based on blackboard hypothesis postings; the ***task dispatcher*** for sending commands to the scheduled modules when they are available; the ***logic task scheduler*** for maintaining consistency of blackboard postings; and the ***question/answer handler*** for processing questions and answers posted to the blackboard. The first two of these are modified as follows:

1. To integrate a new CODER module, the scheduling rule base used by the domain task scheduler must be modified. Any new rules required for scheduling of the new expert must be incorporated.
2. The task dispatcher's list of available experts must be updated to include the new expert.

Finally, the configuration of CODER, discussed in section 3.2, must be updated to include the new expert.

1. The system administrator of the host on which the module will reside must be notified to create a new socket having the same name as the module.
2. The new module must be added to the configuration map file on the host from which the CODER system will be started.

C Programs: Non-inferential programs are integrated by incorporating the C communications extensions as defined in section 3.2.3. C programs may be required to act as servers as well as clients. For example, the user interface manager, written in C, is normally a server; however it may request that the input analyst module process user responses to prompts and menus. The ask predicate may be used by Prolog modules only; when C programs need to send messages to other CODER

```
%************************
%  ATTEND_TO_AREA
%   Entry point for awakening of an expert.  Initiates the
%   retrieval of hypotheses from the blackboard and processing
%   of any applicable hypothesis relations.
%
%  Notes to implementor:
%   1. Replace all occurrences of 'expert_name' with your module name
%   2. Modify the process_goal clause to include any
%       callable functions.
% Calls :
%  View_area   - to retrieve hypotheses from blackboard, via 'ask'
%  Process_hyp - to select hypotheses for which the module has goals
%                (see Figure 12)
%  Finished    - notifies the strategist that search task is complete
%************************

% attend_to_area(+Area)

attend_to_area(Area):-                 % check brand new hyps
    ask(bboard,view_area(Area,new,expert_name,Hyp_set)),
    process_hyp(Hyp_set,Found),        % always performed first
    finished(Found).

                                  % check area of bboard already
attend_to_area(Area):-                 % seen by other experts
    ask(bboard,view_area(Area,seen,expert_name,Hyp_set)),
    process_hyp(Hyp_set,Found),        % if no brand new hyps exist
    finished(Found).

finished(yes):-                        % Valid goals found in area.
    ask(strategist,done(expert_name)).

finished(no):-                         % No valid goals found in area.
    nl,
    writeln('No task found to be processed by *expert_name* expert'),
    ask(strategist,done(expert_name)).

% ATTEMPT_HYP(Relation)
% Expert begins a processing cycle limited to attempting to produce
% hypotheses with the head relation.

attempt_hyp(Relation):-
    Relation.
```

Figure 11. Prolog Module Integration with Blackboard

```
%************************
% PROCESS_HYP
%  Processes the hypothesis set retrieved from the blackboard.
%  The hypothesis set must be recursively reviewed as it may contain
%  multiple hypotheses which have been posted to the Area.
%  Only those hypotheses with a goal (relation) contained in the expert
%  will be processed.
%  The hypothesis set retrieved will contain a LIST of hypotheses:
%  [ [[Fact,Confidence,Expert,Hyp_id,Dependencies],Time,Area], etc.]
%
%  Called By:  attend_to_area  (see Figure 11)
%  Calls:
%   process_goal - to process an applicable goal, if found
%   itself       - to recursively select additional applicable goals
%************************

% process_hyp(+Hypothesis_set, -Found)

process_hyp([],Found):-
     var(Found),
     process_hyp([],no).

process_hyp([],Found).                %boundary case - no more hypotheses

process_hyp([[[Fact,Cnf,Exp,Hypid,Dep],Time,Area]|Resthyp],Found):-
     process_goal([[Fact,Cnf,Exp,Hypid,Dep],Time,Area]),     % has a goal
     time(Time_proc),
     ask(bboard,hyp_processed(Hypid,expert_name,Time_proc)),
     process_hyp(Resthyp,yes).

process_hyp([Firsthyp|Resthyp],Found):-  %hypothesis without a goal
     process_hyp(Resthyp,Found).

%************************
% PROCESS_GOAL
% Selects and processes any applicable bboard hypotheses.
%
% For Example:
% process_goal([[id_user(Timereq),Confv,Exp_id,Hyp_id,Dep],Time,Area]):-
%     id_user.
% process_goal([[um_req(Req,Timereq),Confv,Exp_id,Hyp_id,Dep],Time,Area]):-
%     um_req(Req).
%
% Note to Implementor:
% The process goal procedure must be added to your module.  Refer to the
% examples above.
%************************
```

Figure 12. Prolog Module Hypothesis Processing

modules they use the ***protocol*** function, one of the low-level communications functions used in the ask function definition.

protocol(Module,Clause)

The protocol function is defined in the comm_funcs.c code (see Figure 9) which must be included for a C program to be a CODER server. Following incorporation of the communications extensions in the C program, the configuration must be updated as described in section 3.3.2 for integration of a Prolog module. A socket must be added to the host's services and the C module name and location must be included in the configuration map file.

To remove a CODER expert, the steps followed to add the expert to the strategist components and to the configuration simply need to be reversed. However, consideration must be given to the impact that removal of a particular module may have on other system modules. For example, if the user modeling expert were removed, all modules which tailor processing to the type of user would have to be reviewed.

In summary, the tools and methods described in this chapter supported all development of the CODER retrieval subsystem. That is, the UNIX operating system, the Prolog and C programming languages, various file structures, existing databases, an underlying communications layer and code for integration of new modules provided all of the tools needed to implement this version of the CODER retrieval system.

4.0 Implementation

The implementation of the CODER retrieval subsystem has progressed in stages determined by the interdependencies of required modules and the availability of various segments of the communications layer. The earliest module, a p-norm search expert written in Prolog [WEAV 86a], was implemented as a stand-alone module, was not integrated with a blackboard/strategist, and searched a small test collection generated by the SMART system. Later, portions of the blackboard/strategist complex were written; simulated calls to the blackboard/strategist were then added to the search expert. Next, the knowledge administration complex was coded so that the analysis subsystem implementation as well as development of modules requiring knowledge structures such as frames and relations could proceed. When the communications layer was developed, Prolog modules could pass messages to one another. Enhancements to the communications functions allowed message passing between C and Prolog modules. As modules were written and tested, they were integrated with existing modules.

This chapter describes the implementation stages that preceded the current version of CODER. The first prototype will be discussed briefly. The functions performed by each module of the retrieval subsystem, as well as the incorporation of heuristics from ISR literature, will be presented. Nearly all modules are **prototypical** modules and require enhancements before they can be considered fully developed. However, they provide the foundation for further research and experimentation. Finally, the configuration of the implemented system will be illustrated.

4.1 Prototype I

The objective of the earliest retrieval subsystem prototype, **Prototype I**, was to test the communications functions written for message passing between Prolog modules. Prototype I was a skeletal implementation of the search expert and the blackboard/strategist complex. It demonstrated that modules could communicate with one another on different hosts by using the TCP/IP communications protocol.

As depicted in Figure 13, Prototype I executed on three computers running 4.2 BSD UNIX.[2] Six processes with related sockets were started, one each for the blackboard, strategist, search expert, query_parser, user_interface and report expert. A simple C program containing the communications function, ***protocol***, played the role of the "user_interface" manager and passed a pre-structured p-norm query to the query_parser. No binding of variables or other communication between C and Prolog modules was included. The "query_parser" module, not yet written, was simulated by a few lines of Prolog code. The module posted a p-norm query entered by the implementors in a pre-defined syntax with known concept numbers instead of terms. For example, prompted by the user interface, the implementors might enter the following.

Enter module > *query_parser*
Enter clause > *qparse(p_norm,[2,or,1,[565,7],[827,8]],10).*

where

query_parser was the name of the socket for the query parser module;

qparse was a callable Prolog function in the query_parser module;

p_norm was the function posted to the blackboard which resulted in scheduling of the search expert;

[2,or,1,[565,7],[827,8]] was the p-norm query where concept 565 with a weight of 7 was *ored* (using p value of 1) with concept 827 with a weight of 8 and was used to find and rank relevant documents having the same concepts; and

the last argument, **10**, indicated the number of documents desired.

[2] The configuration illustrated represents one of many that were tested.

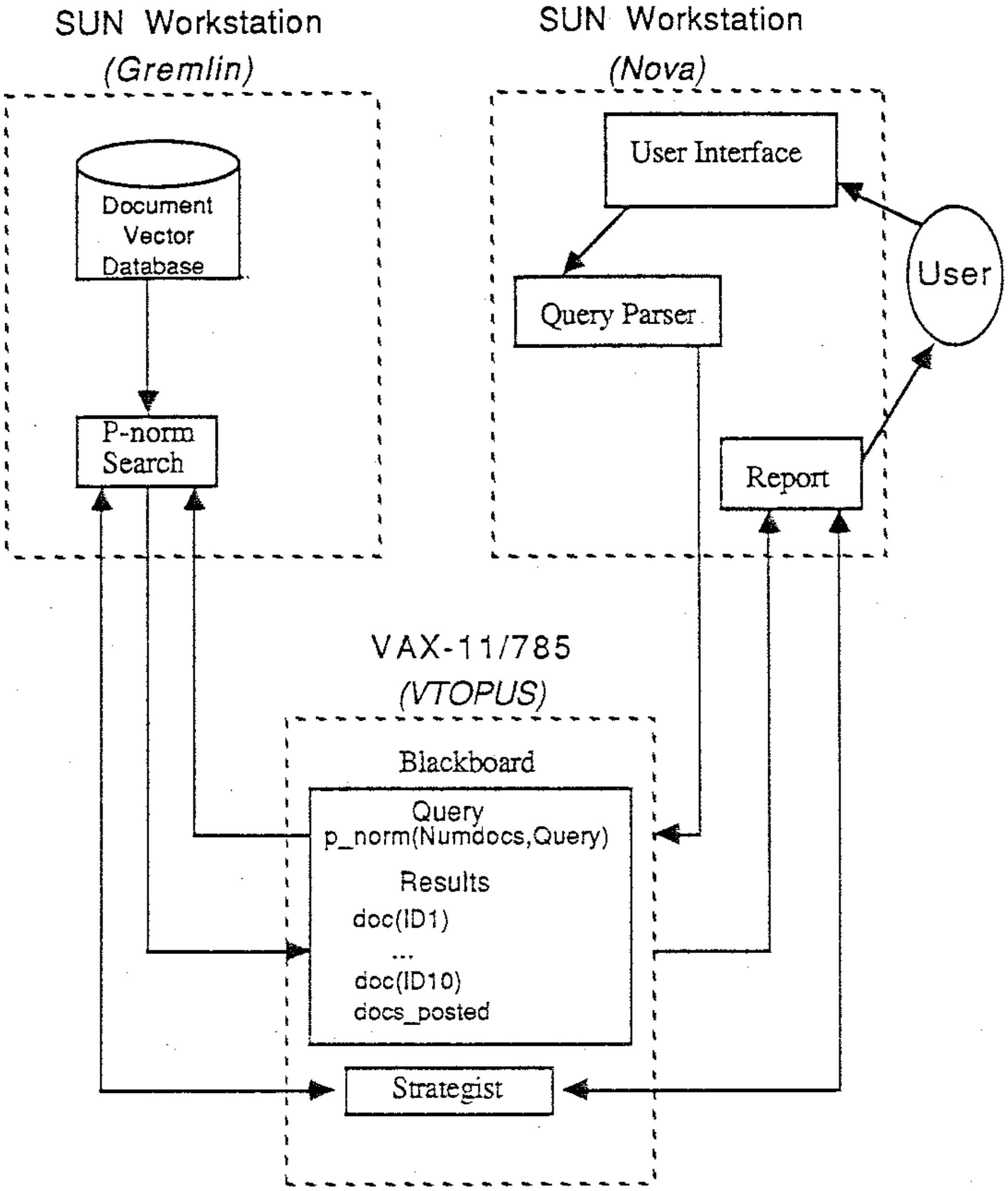


Figure 13. Prototype I

The "report" module, also not written, was simulated by a small Prolog program which passed retrieved document numbers posted on the blackboard to the display terminal.

The implementation of Prototype I, as limited as it was, indicated where modifications to original design specifications for proper blackboard/strategist functioning were required. It verified that Prolog modules residing on different host computers could communicate with one another efficiently when the client/server model was employed. It demonstrated that small external knowledge bases (EKB) could be easily accessed by modules using the MU-Prolog 3.2db extensions. The blackboard architecture was shown to be an effective architecture, at least for limited functioning of an IR system. Moreover, Prototype I fostered establishment of the methods used to integrate and test CODER Prolog modules.

4.2 Retrieval Subsystem Modules

Implementation of the retrieval subsystem began in Spring, 1986 when graduate students in an *Information Storage & Retrieval* class started to develop pieces of the CODER system to fulfill class project requirements. Although class projects served mainly as a pedagogical tool for students learning how to program in Prolog, fragments of the CODER system began to appear. As students became familiar with the CODER project, some continued their work on parts of the system to fulfill Master's or Doctoral degree requirements.

The stand-alone p-norm search expert, the beginnings of the blackboard [SEN 86], 2 of 4 components of the strategist [KOUS 86], and parts of the communications layer were written to satisfy class requirements in 1986. The knowledge administration complex and a simplistic prototype user model expert were developed and tested between June and December, 1986 as part of this research. In 1987, class project work included prototyping of experts to browse the HAI and the lexicon.

Additionally, modifications to the MU-Prolog interpreter, parsers for the analysis subsystem, a Boolean version of the search expert and a Boolean query formulation assistance program were initiated as class projects. By June 1987, bits and pieces of the CODER system, most only partially developed and none integrated, were available. An optimistic Gantt chart was developed to target implementation and integration of Version 1.0 retrieval subsystem modules (see Appendix D). Details of the implementation of each module are recorded in the following sections.

4.2.1 Blackboard/Strategist

Although the term **blackboard/strategist** is used throughout this thesis to denote the vehicle used for storing messages and for scheduling and control of modules, the blackboard and the strategist are in reality two distinct modules with separate but integrated functions. The CODER **blackboard** is an area for communication between modules. All messages from the community of experts are posted to the blackboard. The **strategist** handles the scheduling and control of modules. When messages are sent to the blackboard, the blackboard notifies the strategist which in turn schedules one or more experts to view the message(s) posted and to take appropriate action.

Communication between experts occurs primarily by means of *posting* and *viewing* hypotheses in blackboard subject areas. A **hypothesis** is a 5-tuple structured knowledge form

< Fact, Confidence, Expert id, Hypothesis id, Dependencies >

where

Fact is a hypothesized Prolog-type fact;

Confidence is an integer value in the range [0,100] representing the confidence the expert has in the fact. Experts assign confidence values according to whatever knowledge aggregation scheme is appropriate for the set of constraints and the knowledge sources available.

Expert id is the module name of the expert posting the hypothesis.

Hypothesis id is a unique id number assigned to the hypothesis by the blackboard when the hypothesis is posted.

Dependencies is a list of ids of other hypotheses which the expert has used to make this hypothesis. The dependencies list allows truth maintenance functions to be performed when hypotheses are retracted or confidence values are radically altered. This list may be null.

Questions and answers may also be posted by modules. Although a question/answer handler has been written as part of this research, it is not used by any of the Version 1.0 modules. A question may be posted as a 4-tuple. Arguments are the same as those for the hypothesis tuple less the confidence value. Like hypotheses, questions are posted as facts following Prolog syntax rules and a unique id is assigned by the blackboard to each. Posted answers contain six arguments.

< Question id, Answer id, Answer, Expert id, Confidence, Justification >

where

Question id must match the id of a previously posted question;

Answer id is assigned by the blackboard;

Answer is a list of one or more Prolog-syntax facts which answer the question posted;

Expert id is the module name of the expert posting the answer;

Confidence is an integer value in the range [0,100] representing the confidence the expert has in the answer; and

Justification is a list of Prolog facts which support the expert's answer. This list may be null.

Because new predicates have been added and nearly all of the predicates contained in the original blackboard/strategist specifications have been altered at least slightly, a modified set of blackboard/strategist functional specifications appears in Appendix E.

The Strategist: An illustrative numbered sequence of calls by an expert to and from the blackboard/strategist appears in Figure 14. Three more specific illustrations of blackboard processing limited to new hypotheses, modified hypotheses and retracted hypotheses follow. In general, the steps corresponding to those shown in Figure 14 occur as follows:

1. One of the modules in the CODER community of experts posts a hypothesis or a question to the blackboard.

2. The blackboard informs the strategist of any new dependencies for the hypothesis or question posted; for example, if the hypothesis posted has dependencies on other hypotheses, those dependency relationships are retained by the strategist.

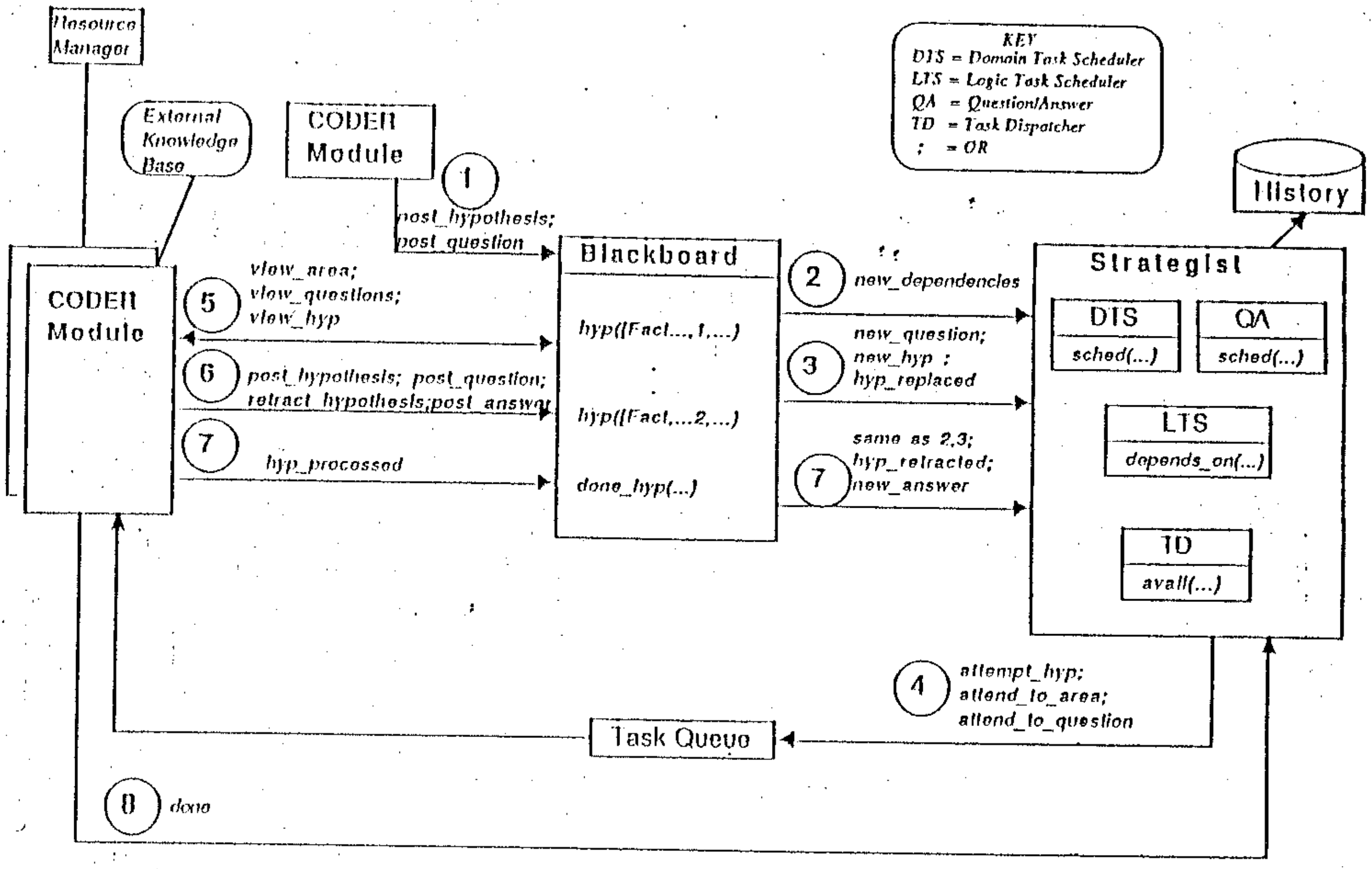


Figure 14. Sequence of Calls to and from the Blackboard/Strategist

3. The blackboard sends the new hypothesis or new question information to the strategist. If the hypothesis posted matches a previous hypothesis but has a higher confidence value, the blackboard notifies the strategist with the hyp_replaced predicate.
4. The strategist places a task for the appropriate module(s) in the module's task queue. That task will either be attend_to_area, attempt_hyp or attend_to_question, depending on what was posted. When the module is available for processing, it is dispatched by the strategist.
5. The module views the appropriate hypotheses or questions posted on the blackboard, as indicated by the task assigned to it.
6. As hypotheses or questions are processed, other hypotheses, questions or answers may be posted to or retracted from the blackboard.
7. The blackboard is informed of each hypothesis that the module has processed. At the same time, the blackboard and strategist may be processing the postings of the previous step.
8. When the module completes its task, it notifies the strategist that it has finished its processing and is available to perform another task.

As mentioned in chapter 3, the strategist consists of four components: the ***domain task scheduler*** (DTS) for determining which modules are to be scheduled based on blackboard hypothesis postings; the ***task dispatcher*** (TD) for sending commands to the scheduled modules when they are available; the ***logic task scheduler*** (LTS) for maintaining consistency of blackboard postings; and the ***question/answer handler*** (QA) for processing questions and answers posted to the blackboard.

Since the components of the strategist are each small (they contain from 30 to 135 lines of Prolog code, 1 goal per line) and their functions are non-overlapping, all four elements have been implemented as one module. Each module exists as a separate file of Prolog functions. However, all files are consulted in one Prolog process. This strategy eliminated the communications overhead costs which would have been incurred if the strategist components passed messages to each other. Due to the simplicity of the functions, little savings would have been realized as a result of concurrent processing among strategist components. If future versions of CODER include expanded functions within the elements of the strategist, for example enhanced scheduling strategies in the domain task scheduler, separation of strategist components as distinct modules should be considered. Such separation would require modification to *asks* made by the blackboard and other modules to the strategist, as well as creation of a new socket(s) and editing of the configuration map file.

The components of the strategist, although they are efficient and function properly, have been implemented as prototypical elements. Enhancements to the DTS are required to incorporate a **pending hypothesis area** of the blackboard to which hypotheses are moved when confidence values reach absolute levels. The scheduling strategies presently do not consider the context of the phases of the overall task in which the system is engaged. The TD does not inform the DTS when task queues are empty or when no tasks in a queue have priority greater than a certain threshold. Methods required to implement more complex control and scheduling strategies in this context are presently vaguely understood and can be more clearly defined when functions requiring them are pinpointed.

The Blackboard: The Prolog program containing the blackboard module is also referred to as the **posting area manager**. Initially prototyped in Spring 1986 [SEN 86] according to the blackboard functional specifications of the original CODER design [FRAN 86], the blackboard module [3] performs several primary functions.

- It receives and stores hypotheses [4] posted by modules.
- It notifies the strategist when new hypotheses are posted, retracted or modified with new confidence values.
- It maintains a local fact base containing a repository of messages grouped by **areas** available for viewing by any module. Thus, it represents an instantaneous snapshot of what the system as a whole hypothesizes about the problem situation.

New Hypotheses: Figure 15 illustrates the flow of processing when new hypotheses are posted to the blackboard. The following sections explain the steps shown in the figure.

[3] The blackboard has been implemented as the **bboard** module and socket.

[4] For the remainder of this section, the term **hypotheses** also refers to posted questions and answers.

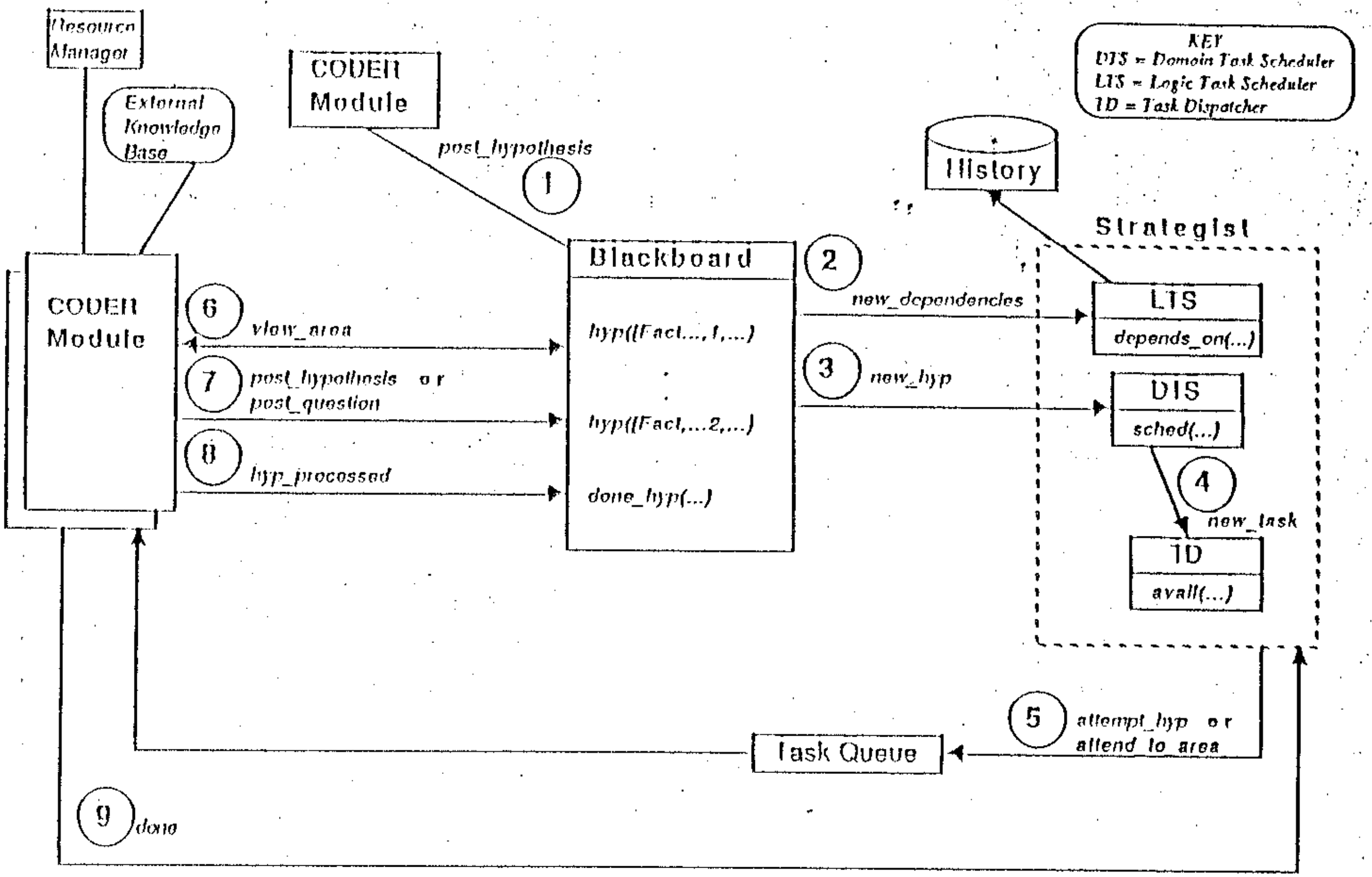


Figure 15. New Hypothesis Processing

Step 1: To begin the sequence, one of the CODER modules posts a new hypothesis with the *post_hypothesis([Fact,Conf,Expert,Hypid,Depend],Area)* fact. The **Area** argument conceptually indicates a structured panel of the blackboard in which all hypotheses relevant to a hierarchical section of the problem situation will be grouped. Hypotheses are stored in a local Prolog fact base containing facts in the following form:

hyp([Fact,Conf,Expert,Hypid,Dependencies],Time,Area)

where **Time** is a timestamp[5] assigned by the blackboard when the hypothesis is posted, and **Area** is the area specified by the expert when the hypothesis was posted.

Step 2: After hypotheses are asserted into the hypothesis fact base, the blackboard sends the new dependencies list to the strategist:

ask(strategist,new_dependencies(Hypid,Hyp_functor,Expertid,Conf,Dep))

Since all arguments in the *ask* clause are instantiated at the time it is executed, the blackboard continues its processing without waiting for a reply from the strategist. Thus, the blackboard and the strategist modules execute *concurrently*. The new dependencies are logged in a local fact base by the logic task scheduler code in the strategist module in case they are needed later for truth maintenance. A fact for each dependency is created:

depends_on(Hypid,Fact,Confidence,Expert,Depends_on_hypid)

This local fact base contains both parent and child dependencies; that is, for each hypothesis, the hypotheses that it depended on and hypotheses that depend on it are separately represented.

Step 3: The blackboard next notifies the strategist that a new hypothesis has been posted.

ask(strategist,new_hyp(Hypid,Hyp_functor,Dep,Conf))

Again, parallel processing of the blackboard and strategist modules occurs. The domain task scheduler portion of the strategist is triggered to schedule one or more experts capable of processing

[5] A predicate, time(X), has been added to the Prolog interpreter. It returns the number of seconds since January 1, 1987.

the new hypothesis. If a question is posted instead of a hypothesis, the question/answer handler instead of the DTS is notified to schedule the expert(s) capable of answering the question. To determine which expert should be scheduled, the DTS currently uses a local rule base of antecedent/consequents. Approximately 50 rules contain the Version 1.0 scheduling heuristics. Consignment of scheduling strategies to a rule base allows modification of the strategies without changing the blackboard implementation. Rules are of the form:

sched(Relation,Condition,Action)

where

Relation is the functor of the *fact* portion of a hypothesis.

Condition contains a pair of numbers representing absolute confidence and incremental confidence. Before an expert is scheduled to view the blackboard, the confidence value assigned to the new hypothesis must be greater than or equal to the absolute confidence. The incremental confidence is included for future enhancements to the domain task scheduler, for example varying the control or flexibility desired when scheduling modules.

Action is a list of lists of the form [expert id, task for expert, scheduling priority]. Expert id is the name of a module capable of processing the hypothesis. The task scheduled for an expert is either ***attempt_hyp*** or ***attend_to_area.*** Scheduling priority is an integer from 1 to 10 and is used by the task dispatcher when ordering tasks within expert queues. The highest scheduling priority is 1 and 10 is the lowest.

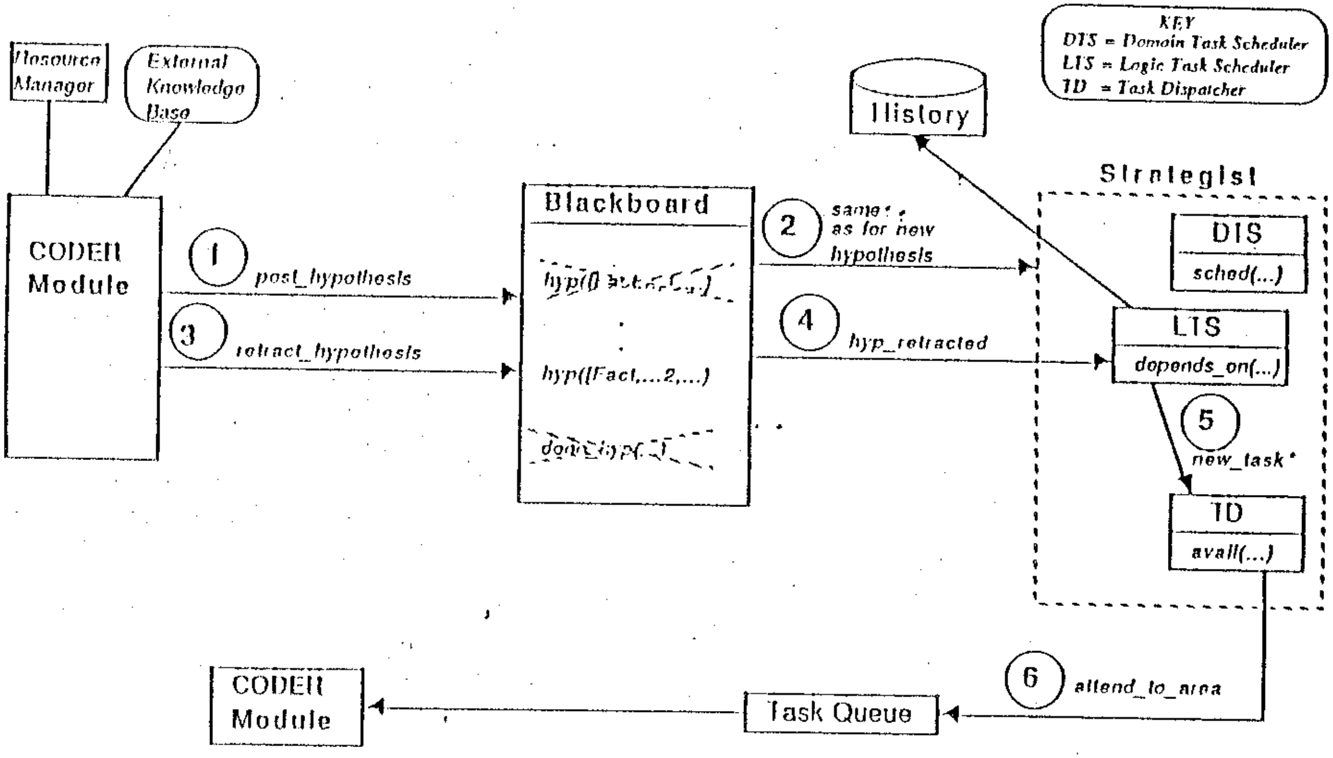
Step 4: When the DTS determines which expert(s) to schedule, it issues the *new_task* command to the task dispatcher component of the strategist. Note that since all strategist components reside in one module, the ask predicate is not used. The TD logs the task in an external history file, and places it in a local queue of tasks for the scheduled expert. Tasks are ordered by scheduling priority or first-in, first-out (FIFO) if more than one task has the same priority. A fact, *avail(Expert,Availability)*, exists for every CODER expert to indicate whether the expert is currently available. All experts are available at the beginning of a retrieval session. As a task is scheduled, the **Availability** parameter is set to **no** and the task is removed from the expert's task queue. When the expert notifies the strategist that it has finished performing the requested task, the module again becomes available and the next task in the queue, if any, is dispatched.

Steps 5 and 6: When the strategist dispatches an expert, it issues the command *ask(Expert,Relation)* where **Expert** is the name of the scheduled expert and **Relation** is either *attend_to_area* or *attempt_hyp*. Recall that the attend_to_area and attempt_hyp functions must be integrated into every CODER module to allow communication between the module and the blackboard/strategist complex (reference chapter 3, section 3.3.2). The expert dispatched retrieves hypotheses from the blackboard by issuing the *view_area* predicate and processing the hypothesis set returned. Note that three categories of hypotheses may be retrieved: brand new hypotheses that no expert has yet processed, hypotheses that other experts may have processed but this expert has not yet seen, and hypotheses with modified confidence values.

Steps 7 and 8: During processing of hypotheses, the expert may post other hypotheses on the blackboard using the *post_hypothesis* function in an *ask* to the **bboard**. For each hypothesis that the expert processes, it informs the blackboard that it has processed the hypothesis by issuing *ask(bboard,hyp_processed(Hypid,Expert,Time))*. The blackboard tags the hypothesis by creating a *done_hyp* fact indicating that the hypothesis has been processed by the expert. This fact is referenced when experts view areas of the blackboard since hypotheses in the area that the expert is viewing may include some that the expert has already processed; the hyp_processed fact allows these facts to be filtered from the set being retrieved.

Step 9: When the expert has finished processing all applicable hypotheses, it notifies the strategist that it has completed its task with the command *ask(strategist,done(Expert-name))*. While an expert is performing scheduled tasks, other experts may be performing tasks concurrently; the blackboard and strategist modules will also continue to execute in parallel.

Retracted Hypotheses: Based on new information on the blackboard or in knowledge sources, an expert may wish to retract a previously posted hypothesis. The *retract_hypothesis* command exists



Notes:
*All dependent hypotheses are rescheduled.
** The hyp and done_hyp facts are retracted.

Figure 16. Retracted Hypothesis Processing

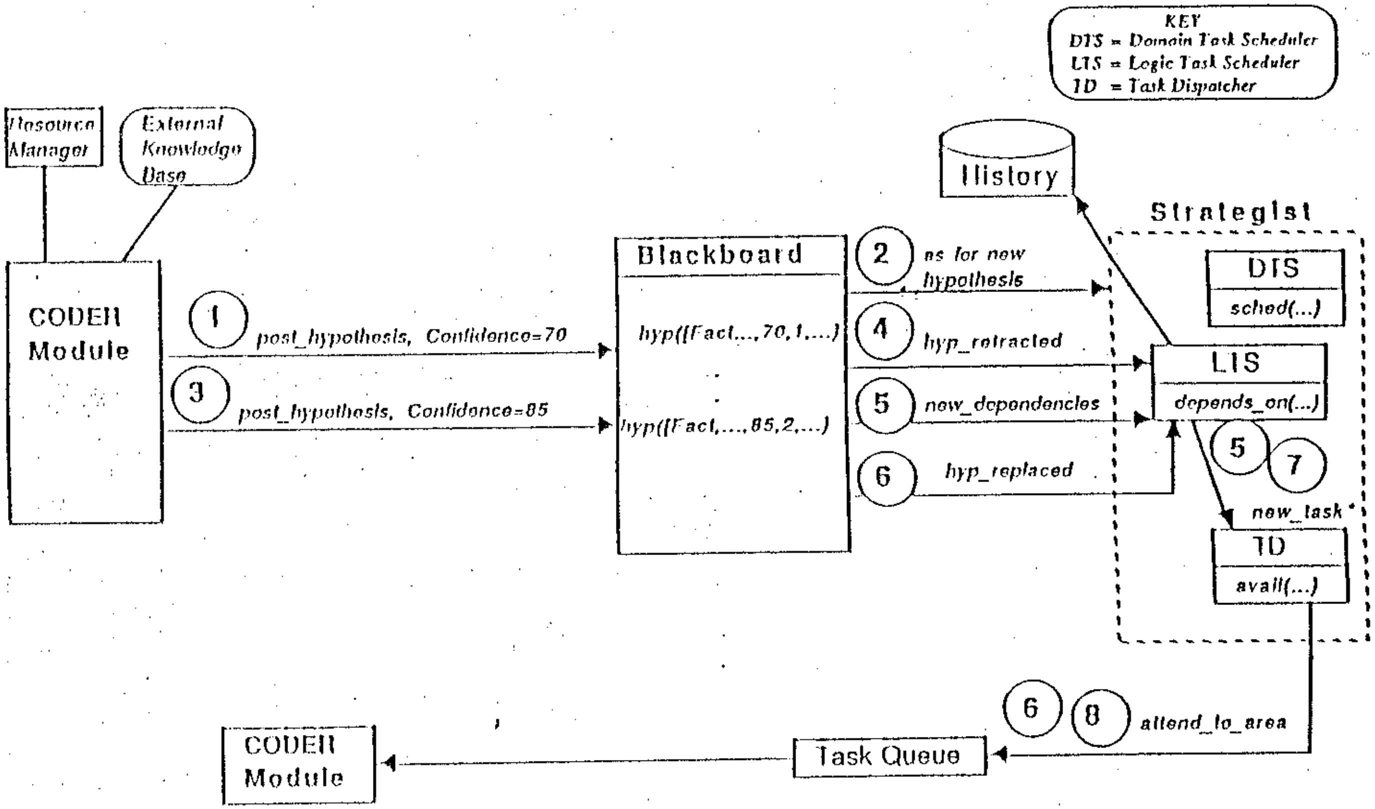
to allow hypothesis removal. Only the expert that originally posted the hypothesis may retract it. When a hypothesis is retracted, the blackboard informs the strategist and then removes the hypothesis and any related facts from its local fact bases. Figure 16 illustrates the sequence of calls made when a hypothesis is retracted.

The logic task scheduler of the strategist deletes all dependency relations for the hypothesis. The ***depends_on*** facts for hypotheses which list the retracted hypothesis as a dependency and for hypotheses on which the retracted hypothesis was dependent are removed from the dependency fact base. The LTS reschedules hypotheses dependent on the retracted hypothesis by issuing the ***new_task*** command so that the TD component can dispatch the expert who posted the hypothesis which was dependent on the one retracted.

Priority for a rescheduled task is based on hypothesis confidence and number of dependents. The heuristics used to determine scheduling priority have been arbitrarily determined. As the number of dependencies for the hypothesis being rescheduled increases, the priority approaches 1. In conjunction with a smaller number of dependencies, higher confidence values for the hypothesis being rescheduled where that confidence excludes the retracted hypothesis, shift the priority closer to 10.[6] For example, if a hypothesis dependent on only the retracted hypothesis has a confidence value less than or equal to 40 it is rescheduled with a priority of 6. If a hypothesis dependent on only the retracted hypothesis has a confidence value greater than 70 it is rescheduled with a priority of 10. A hypothesis with confidence value less than 20 and dependent on 4 or more hypotheses is rescheduled with priority 2. Sixteen rules such as these exist in the LTS for rescheduling.

Modified Hypotheses: A previously posted hypothesis may be re-posted with a higher confidence value. To insure proper control, only the expert that originally posted the hypothesis may issue the ***post_hypothesis*** for the existing hypothesis again. The confidence value assigned by the expert must

[6] Recall that priority 1 is the highest scheduling priority while 10 is the lowest.



Notes:
*All dependent hypotheses are rescheduled for both the hyp_retracted and hyp_replaced predicate since new dependencies may be assigned for the replace hypothesis.

Figure 17. Modified Hypothesis Processing

be higher than the previously assigned confidence or the posting will be ignored. If for some reason an expert wished to lower the confidence value of a previously posted hypothesis, the retract_hypothesis command must be issued first. Figure 17 illustrates the sequence of calls made when the confidence value for an existing hypothesis is modified.

When the blackboard receives posting of a hypothesis with a higher confidence value it notifies the strategist to retract the existing hypothesis. The logic task scheduler of the strategist proceeds to delete all dependency relations for the old hypothesis and to perform any rescheduling indicated. The blackboard, meanwhile, continues its processing by removing the old hypothesis from its local fact base, and asserting the new hypothesis with the higher confidence. It then creates a fact, ***mod_hyp***, to indicate that the hypothesis has been modified. Finally, it notifies the strategist of any new dependences and issues a ***hyp_replaced*** command so that confidence values stored in the logic task scheduler's ***depends_on*** facts may be updated. Note that the ***done_hyp*** fact created by the blackboard still exists if an expert processed the hypothesis previously. Therefore, if the expert is rescheduled it may or may not wish to process the hypothesis, now with a higher confidence value.

Blackboard Areas: As mentioned earlier, conceptual panels of the blackboard are indicated by the **area** argument of blackboard hypotheses. The blackboard areas used by this version of the CODER system appear in Figure 18. The areas defined evolved as the CODER system was developed. To reduce the time needed to process large sets of hypotheses, the problem description builder module periodically issues the ***clear_hyps(List_of_functors)*** predicate to the blackboard. For example, many hypotheses requesting display of prompts or menus are posted to the blackboard. Such hypotheses are not used after the prompt is displayed; therefore, issuing clear_hyps([disp_prompt,disp_menu]) to the blackboard will result in retraction of all disp_prompt and disp_menu hypotheses.

AREA	DESCRIPTION
user_req	Requests for menus and file displays from the user
problem	Problem description information
state	Problem state data
umodel	User modeling information
results	Retrieved documents
report	Retrieved browse data
query	Queries formed for the search expert
qform	Query formulation data
structured_data	Frame and relation input by user
cleanup	Used at end of session

Figure 18. CODER Blackboard Areas

4.2.2 Knowledge Administration Complex

As an AI retrieval system, CODER requires an appropriate method for *representing knowledge.* World knowledge and domain knowledge are provided in the *Spine* for the **Collins Dictionary of the English Language** and the **HAI**. However, specific knowledge about entities in the problem universe, information retrieval, is also needed. Entities include words, names, subjects and other lexical items; documents and fields of documents; and users of the system. Mechanisms to facilitate representation of entities and their attributes and to allow inference and reasoning about entity classes and relations is provided by the knowledge administration (KA) complex, a Prolog-based frame representation system.

The choice of a frame-based system written in Prolog as the method of knowledge representation was made early in the design of the CODER project. The specifications for the predicates required were sketched by Robert France in his M.S. thesis [FRAN 86] and will not be repeated here. These high level predicates laid the foundation for the implementation of the KA complex. This section will discuss that implementation and will explain the facility developed for building and storing "knowledge" in the CODER system.

The knowledge administration complex requires a system that can support the creation and manipulation of three levels of knowledge representation.

Elementary data types (EDT) Lowest level attributes, such as character, integer and atom primitive types. Restrictions and quantifications of the three primitive types define additional EDTs.

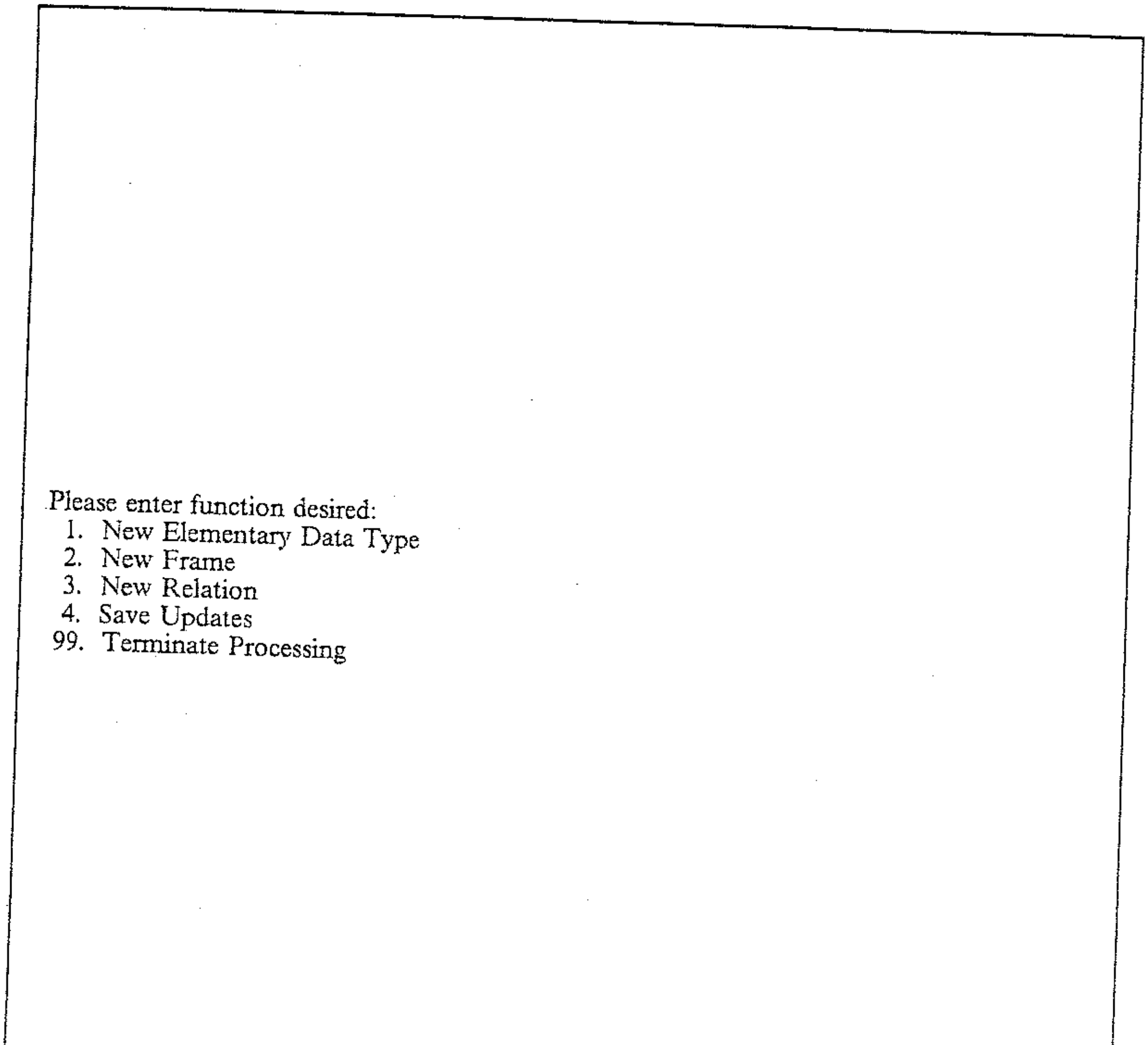
Frames Models of entities. Frames are classified by frame type where each frame type is defined by a set of slots and the EDTs, frames or relations associated with each slot.

Relations Logical relations over objects, where an object is an EDT, frame or other relation.

Each level of knowledge representation has a **type manager**; the frame and relation levels also have **object managers**. The type managers provide the ability to identify, test, and manipulate EDT, frame or relation *definitions.* The object managers support the creation and manipulation of objects, that is, the instantiation of *data*. The type and object managers will be used by the ***system administrator*** and by the CODER community of experts respectively. The system administrator, for example a designated CODER implementor, is an individual responsible for the definition and maintenance of EDT, frame and relation types. The CODER community of experts will use the knowledge administration predicates to define and represent factual knowledge, and to determine relations, such as subsumption or matching, among both objects and types. A clear distinction between objects and types, as with any frame representation system [FIKE 85, BRAC 83, HAHN 86], must exist in the KA complex. Therefore, separate Prolog programs contain the type and object managers.

4.2.2.1 Type Managers

All three KA type managers are contained in the program, ***knowadm***. To simplify entry of type definitions, the program may be driven by the main menu shown in Figure 19. When the goal ***update*** is entered, the menu will be displayed. For each of the first three options, the system administrator will be prompted to supply the data required to create the Prolog facts which represent type definitions. Since the program is written in Prolog, capitalized entries or entries containing special characters such as blanks, commas or periods must be enclosed in quotes. As required by Prolog, all entries must end with a period. A sample session appears in Appendix F. The information stored for each type definition is described in the following sections. Option 4, *Save Updates*, provides a checkpoint facility as a safety measure and is recommended when large numbers of type definitions are entered during one session. It creates a Prolog save state and explains how to re-enter the session at the point where updates were saved. The final option, *Terminate Processing*, appends the new type definitions to their respective type files.



```
Please enter function desired:
  1. New Elementary Data Type
  2. New Frame
  3. New Relation
  4. Save Updates
 99. Terminate Processing
```

Figure 19. Knowledge Administration Complex Main Menu

EDT Type Manager: The Elementary Data Type manager handles identification and coordination of EDTs. It provides functions for the creation of a new EDT, for testing the type of an EDT object and for navigation of the type lattices.

New EDT Types: A single Prolog fact represents each EDT.

ka_edt(Edt_name, Quantifier, Parent, Restriction)

Arguments consist of the following.

Edt_name Any name not already used as a frame, EDT or relation type name. Primitive types ***char***, ***int*** and ***atom*** are reserved.

Quantifier The number of required items in an EDT object list or set. Not yet fully implemented is the ability to assign a quantifier which is a list or set of characters which the EDT object must contain. For example, for EDT ***phone_number*** the quantifier field might contain [int, '-', '(', ')', '/']. This field may be null to indicate that quantification is not needed.

Parent The EDT name of a parent EDT whose restrictions and/or quantifications will be inherited. Minimally, the char, int or atom parent may be supplied.

Restriction A list of items with which the EDT object must comply. Restrictions include membership, minimum and/or maximum for integers, negation or any combination of these. Some sample restrictions include:
[min,01,max,12] (for month of the year);
[member,'Sat','Sun'] (for weekends);
[not,'Sat','Sun'] (for weekdays).

If a parent EDT having a non-null quantifier has been entered for a newly created EDT, the child EDT will inherit the parent's quantifier. When the child EDT is entered, its quantification must match the parent's or must be null in which case the parent quantifier will be inherited. If the parent quantifier is null, then a child quantifier may be entered. Since the parent of the new EDT may also have had its own parent, any quantification for an ancestor of the EDT will automatically be inherited because it has already been merged with the parent quantifier. The knowadm program includes checking of parental ancestry to ensure that a loop is not created somewhere by an EDT having a parent which has that EDT as an ancestor.

Complex routines have been written to merge EDT parent restrictions with child restrictions. Table 1 depicts the merging of inherited parent restrictions with child restrictions. Restrictions are grouped in the table by the not, member and min/max operators. When restrictions are inherited,

Table 1. EDT Restriction Inheritance

EDT RESTRICTION CASE TABLE

CHILD	PARENT	MERGED RESTRICTION	COMMENTS
[]	any	parent's restriction	Inherit parent
not	not	union of both 'not' sets	
not	member	set difference of child and parent sets	child 'not' set must be subset of parent 'member'.
not	min, max	merged member set of elements within min/max, minus NOT set.	merged member set must not be [].
not	[]	child 'not' set	no inheritance
member	not	child 'member' set	child set elements must not be in parent set.
member	member	child 'member' set	child set must be subset of parent set.
member	min, max	child 'member' set	each element of child set must be within parent min/max range
member	[]	child 'member' set	
min,max	not	child min,max	members of parent NOT set must not be in range of child min/max
min,max	member	new member set of elements of parent set within child min/max	parent member set mapped into new child member set.
min,max	min,max	child min,max	child min,max must be within range of parent min,max
min	min,max	child min,parent max	child min must be > = parent min; inherit parent max
max	min,max	parent min,child max	child max must be = < parent max; inherit parent min
min,max	[]	child min,max	
[min,max,[not]]			merged as member

the merged restrictions are stored with the EDT. This eliminates recomputation of merged restrictions when new EDT objects are created; the original specific child restrictions which have been merged with parent restrictions are lost although they could be deduced in most cases.

Testing EDT Object Type: In addition to creation of new EDTs, the knowadm program contains predicates which may be called by the CODER community of experts as well as by the system administrator. A single predicate, ***is_elt*** validates the type of an elementary object and is used by the object managers when new objects are instantiated. Given an EDT type name and an EDT data object, the predicate verifies the existence of the EDT type and validates the object based on the quantifier and the restrictions specified for the EDT type.

EDT Lattice Manipulations: The EDT type manager also contains predicates to navigate the hierarchy created by parent assignments to EDTs.

> *weaker(Weaker_type,Stronger_type)* tests whether the weaker_type is a descendant of the stronger_type.
>
> *supertypes(Weaker_type,Stronger_types)* returns a list of all ancestors of weaker_type.
>
> *subtypes(Stronger_type,Weaker_types)* returns a list of all direct descendants of stronger_type.

Frame Type Manager: The CODER frame type manager supports all functions required for the construction of frame definitions. Frame types provide structured representations of an object or a class of objects. Variable-sized sets of named attributes called *slots* characterize each frame type. By specifying each new frame type as a "subclass" of other more general frame types, frames may be ordered into taxonomies. Frame types may have more than one parent as well as more than one child, and thus a complex lattice framework may result. Predicates for navigation of frame taxonomies and for identifying subclasses of frame types are included in the frame type manager.

New Frame Types: For each frame type, a Prolog fact contains the frame characteristics.

ka_frame(Frame_name, Parents, Slot_list)

where

Frame_name is any name not already used as a frame, EDT or relation type name.

Parents is a list of frame type names representing classes of which this frame is a member. For example, a *journal_article* frame type could have parent frame type *journal* which could have parent frame type *bibliographic_reference.*

Slot_list is a list of attribute names and characteristics. All parent slots are inherited, that is they are added to the list of slots defined for a new frame type.

A list of characteristics per slot is contained in the slot_list. Slot characteristics are represented by the list

[Slot_name,Class,Type,Cardinality_min,Cardinality_max,Default]

where

Slot_name is a slot identifier. Although slot names must be unique within a given frame type, they need not be unique among all frame types. However, it is recommended that all slots be given unique names to avoid confusion.

Class identifies the kind of object required to fill the slot. It must be e (EDT), f (frame) or r (relation).

Type is the type name of the EDT, frame or relation to fill the slot.

Cardinality_min is the minimum number of values allowed for the slot when the frame object is instantiated. All slot values for frame objects are stored as *lists* of values. This value may be null if there is no lower bound on the number of slot values.

Cardinality_max is the maximum number of values allowed for the slot when the frame object is instantiated. This value may be null if there is no limit.

Default is the value assigned to an EDT slot when a frame object is created.

Default values are passed from parent frame slotlists when slots are inherited. However, inherited default values for child frame types may be modified; therefore, an inherited slot may have all of the parent characteristics except the default value. Default values are optional for slots which are EDTs. For a slot which has a frame or relation type, the *identifier* of a frame or relation object is stored as the slot value; therefore, default values are not permitted for such slots.

Some sample EDT and frame types appear in Figure 20. Note that not all frame and EDT definitions used in the example, such as the minerals and vitamins frames, are included.

Recall that the fact form for an EDT is:
ka_edt(Edt_name, Edt_parent, Quantifier, Restrictions).

```
ka_edt(temperature, int, [], [min,32,max,110]).
ka_edt(beings, char, [], [member,human,plant,insect,fish,bird,other_animal]).
ka_edt(color, char, [], [member,orange,yellow,red,green,blue,black]).
ka_edt(fruit_name, food_groups, [], [member,apple,orange,banana,tomato,kiwi]).
ka_edt(food_groups, char, [], [member,fruit_veg,dairy,meat,bread_cereal]).
ka_edt(sizes, char, [], [member,small,normal,large,huge]).
```

Recall that the fact form for a frame is:
ka_frame(Frame_name, Parents, [Slot,Class,Type,Min,Max,Default,Slot,...]).

```
ka_frame(food,       [],                [ storage_temp,e,int,[],[],68,
                                          eaten_by,e,beings,[],[],human]).

ka_frame(food_group, [food],            [ storage_temp,e,int,[],[],68,
                                          eaten_by,e,beings,[],[],human,
                                          characteristics,f,char,[],[],[],
                                          group_name,e,food_groups,[],[],[]]).

ka_frame(fruits, [food_group],          [ storage_temp,e,int,[],[],68,
                                          eaten_by,e,beings,[],[],human,
                                          characteristics,f,char,[],[],[],
                                          color,e,color,1,[],[],
                                          fruit_name,e,fruits,1,[],[],
                                          group_name,e,food_groups,[],[],[],
                                          grown_in,f,state,[],[],'Fla',
                                          minerals_in,f,mineral,[],[],[],
                                          vitamins_in,f,vitamin,[],[],[],
                                          size,e,int,[],[],normal,
                                          taste,e,tastes,[],[],sweet]).
```

Figure 20. EDT and Frame Type Examples

Since a single frame type may have more than one parent frame, slots from parent frames are merged according to the *class* and *type* for the slot. The following heuristics are applied.

1. If the slot name, class and type are unique, the slot is inherited.
2. Duplicate slot names with the same class and same type are inherited once.
3. Duplicate slot names having different classes (e, f, r) are not allowed. An error message is provided to the system administrator if this occurs.
4. Duplicate slot names with the same class but a different type are inherited as follows: a) if a subsumption relationship can be established, then the stronger frame type is used, b) if a subsumption relationship cannot be established, then an error message is generated.

Frame Lattice Manipulations: When a new frame type is created, Prolog facts per parent-child relationship for the frame are also asserted so that the frame hierarchy may be efficiently navigated. Facts contain the parent frame type and its direct descendant frame type: *ka_fparent(Parent_type,Child_type)*. Frame taxonomy relationships are reported by *subframe_list(Frame_type,Subframes)* and *superframe_list(Frame_type,Superframes)* which return a list of immediate subframe types and a list of all ancestors respectively. These predicates may also either succeed or fail if both arguments are bound, thus indicating whether the frame types provided have a parent-child relationship.

The *subsumes(Ancestor,Descendant)* predicate indicates whether one frame type is a generalization of another. One frame subsumes another if all of its slots are either included in the stronger frame type's slot list or are generalizations of slots in the stronger frame's slot list (for example, EDT parents and children). If every slot of a frame type A corresponds to a slot of another frame type B, with either the same name and type or with a stronger type than one in B, then frame A subsumes frame B. Therefore, any frame stored as a parent of another frame should subsume its child frame. However, a parent-child link is not required for one frame type to subsume another. Finally, the frame type manager of the KA complex includes the *Slot_list(Frame_type,Slotlist)* predicate to return a frame type's list of slots or to succeed if a given list of slots is a proper subset of the frame's slotlist.

Relation Type Manager: The highest level of knowledge in the system is provided by logical relations. Relations model propositions over objects. Relations may exist among EDTs, frames or other relations. For example, **synonymy** or **antonymy** relationships may exist between 'headword' elementary data types in the lexicon; a relationship ***author_of*** may exist between a person frame and a bibliographic frame. The *relation type manager* provides the tools needed to create and manipulate logical relations.

A relation type is specified by the form ***ka_relation(Relation_name, Arity, [Argument_list])*** where

Relation_name is any name not already assigned to an EDT, frame or relation.

Arity is the number of arguments required in the argument list when a relation object is instantiated.

Argument_list is a list of defined EDT, frame or relation names which this relation may model; that is, arguments represent the types of objects which may be related by this Relation_name.

When arity equals two, binary relation properties as illustrated in Figure 21 may also be assigned. Reflexive, transitive, symmetric and antisymmetric properties are asserted when applicable to binary relations, as indicated by the system administrator. The transitive relation may also be assigned a confidence value to indicate the strength of the transitivity, where 1 is the weakest and 10 is the strongest. Binary properties for relation types are asserted as separate facts of the forms:

reflexive(Relation_name).

symmetric(Relation_name).

antisymmetric(Relation_name).

transitive(Relation_name,Confidence).

The relation type manager also contains predicates which can supply information about relation type definitions. The ***arity(Relation_name,Arity)*** predicate returns the number of arguments required for a relation. The ***signature(Relation_name,Signature)*** predicate returns the argument_list for the relation. CODER modules which may use relation information to represent knowledge

1.
   ```
   ka_relation(synonymy, 2, [query_term,headword]).
   reflexive(synonymy).
   symmetric(synonymy).
   transitive(synonymy,8).
   ```

   Examples of argument lists are
   [domicile,home] and [house,home]

2.
   ```
   ka_relation(part_of, 2, [issue,article,journal,book]).
   antisymmetric(part_of).
   transitive(part_of,10).
   ```

   Examples of argument lists are
   [article#123,journal#456]

Figure 21. Relation Type Examples

will use the *ask* function with these predicates, for example, *ask(knowadm,signature(part_of,Arg_list))*.

4.2.2.2 Object Managers

As mentioned earlier, the object managers support the creation and maintenance of frame or relation objects; that is, data is assigned to slots or arguments of instantiations of previously defined frames or relations. To create an object, the frame or relation *type* must be specified so that appropriate slots may be filled or proper relation arguments may be specified. In addition, a unique identifier is assigned to the object being created. A predicate *unique(A,B)* has been added to the MU-Prolog interpreter for identifier assignment. It returns a unique pair of integers which are used to create an object identifier. Either the object manager or the module requesting creation of a new object may assign the object identifier.

The object manager program has been written so that any module which needs to build or maintain frames or relations may do so in its local Prolog fact base. Then, objects may be stored in appropriate EKBs such as the user model base, or objects such as structured names and addresses in queries may be created locally for use during one retrieval session only. The object manager program issues *asks* to the type manager program to validate objects entered as slot values or relation arguments. Unlike the type manager, the object manager does not have its own socket and is not included in the CODER configuration. Rather, modules requiring frame or relation objects *consult* the object manager code.

Frame Object Manager: The *new_frame* predicate allows establishment of a new frame object. This predicate is of the form: *new_frame(Frame_type, Frame_object)*. Any CODER expert may issue the new_frame predicate to create a new frame object as long as the object manager code has been consulted. When issued, the frame_type must be bound to a defined type; frame_object may be unbound, in which case it will be returned as the object identifier which has been assigned. If the module creating the object wishes to assign its own identifiers, it may do so. However, the identifier assigned must be unique within the module's local fact base of frame objects. Two types of assertions occur to create a new frame object:

fobjid(Object_identifier, Frame_type)
and
fobj(Object_identifier, Slot_name, Slot_value_list).

Sample facts representing two frames are listed in Figure 22. When new frames are created, the fobjid fact is asserted. Next, any slots having non-null default values are asserted as fobj facts. Therefore, a fobj fact will not necessarily exist for every slot defined for a given frame type. Indeed, no fobj facts are required when a frame object is created. Once a frame object has been created, values such as defaults may be removed and/or new values may be assigned.

The method used to store frame objects does make less efficient use of storage space than other methods considered. However, it simplifies processing of slot value manipulations and reduces execution time processing by eliminating the list traversal required by other methods. Two other possibilities were examined in conjunction with the one adopted.

1. For each new frame object, a single fact could be asserted as:

   fobj(Object_id,Frame_type,[Slotname.[Values],Slotname.[Values],...]).

   All slot names and a list of values for each would be included in a single fact. Although this method would considerably reduce the size of the knowledge bases created and eliminates the redundant storage of the object identifier necessary in the method chosen, it would require list processing and manipulation every time a slot value were added or removed. In addition, the frame object manipulation predicates would require excessive list processing. The frame objects in Figure 22 would be replaced by two facts:

   ```
   fobj(163610917,user_eval,[satisfaction.[8],usefulness.['67-90'],why_stop.['out of time'],...]).
   fobj(163610918,session,[nodoc_queries.[0],doc_quantity.[10],user_eval.[163610917],...]).
   ```

```
fobjid(163610917, user_eval).
fobjid(163610918, session).

fobj(163610917, satisfaction, [8]).
fobj(163610917, usefulness, ['67-90']).
fobj(163610917, why_stop, ['out of time']).
fobj(163610917, easy_to_use, [y]).
fobj(163610918, nodoc_queries, [0]).
fobj(163610918, doc_quantity, [10]).
fobj(163610918, user_eval, [163610917]).
fobj(163610918, sessionlgth, [154]).
fobj(163610918, session_id, [26968]).
```

Figure 22. Sample Frame Objects

2. To reduce storage requirements even more, the Slotname could be eliminated from the list of slot names and values. Instead, each list of values would positionally be matched to the slot names defined in the frame type definition. So, the single fact for each object would contain:

   fobj(Object_id,Frame_type,[[Values],[Values],...]).

   This method would require even more extensive list manipulation as well as matching of the frame type slots to the object slot value list for all frame object manipulation. In cases where no values were assigned to slots, each slot would still have to be included in the list so that the positional values could be properly matched to slots in the frame type definition.

The implementation strategy for frame objects could be rewritten according to one of the above methods or using some other strategy. Such modification, however, would require rewriting of all frame object predicates as well as rewriting of modules which use the current frame object structure.

Object Manipulation: Predicates to support the manipulation of frame objects allow updates to slot values, and support reasoning about the relationships between or among frame objects. *Is_frame* and *has_slot_value* predicates return information about the existence of frame objects and the values assigned to frame slots, respectively. Slot values are asserted using the *set_slot_value* predicate, and may be removed with the *remove_slot_value* predicate. The *equal_frames* and *matching_frame* predicates allow comparison between frame objects to establish similarity. A frame object A matches a frame object B if every filled slot of A matches a filled slot of B. Slot values *match* when slot types, classes and values match or when values for a slot which subsumes another slot match. *Matching* is an antisymmetric relation, whereas *equal* is a symmetric relation: every slot in A must match a slot in B **and** every slot in B must match a slot in A.

Relation Object Manager: The *new_relation* predicate creates a relation object. This predicate has the form

new_relation(Relation_type,[Arg,Arg_type|_],Relation_object).

As with the new_frame predicate, any CODER expert may issue the new_relation predicate to create a new relation object. When the new_relation goal is attempted, the relation_type must be bound to a defined type, and the argument list bound to a list of values associated with one of the types defined in the relation type definition. Relation object identifiers may be instantiated by the module creating the object or will be assigned by the object manager code.

When a new relation object is created, the following fact is asserted.

robj(Object_identifier, Relation_type, [Arg,Arg_type|_]).

Each argument in the argument list must be of the arg_type specified; moreover, the arg_type associated with each argument must be one of the allowable types defined in the argument_list of the relation type definition. For example, for the relation 'part_of' in Figure 21 only arguments which are issues, articles, journals or books would be accepted.[7] The number of arguments specified for the object must equal the arity defined for the relation type. So, an example of a relation object fact would be:

robj(12345678, part_of, [issue,23452345,article,12341234])

where 12345678 is the identification of this relation object, 23452345 is the identifier of a frame object for an issue frame, and 12341234 is the object identifier assigned to an article frame.

Object Manipulation: To retrieve information about relation objects, three additional predicates have been written. The *is_relation,* like the is_frame predicate, returns the relation_type for a given relation object identifier and vice versa. The *argument_list(Rel_object,Arg_list)* and *argument(Rel_object,Position,Argument)* predicates return the values and relative positions of arguments for a specified relation object respectively.

[7] The 'part_of' relation example does not include all relevant argument types for the CODER project.

4.2.2.3 Programs and Files

Type and object files for each of the three types of knowledge, EDTs, frames and relations, are maintained by the knowledge administration complex. Programs for the creation and manipulation of types and objects are also part of the KA complex. They reside in the ~coder/know_adm type and object subdirectories as follows.

knowadm is the knowledge representation **type** manager program,

ka_localobj is the knowledge representation **object** manager program,

type files include the ka_frame_file, ka_edt_file and ka_rel_file containing the type definition facts,

ka_inv_frame_file contains the inverted frame parent-child facts, and

object files per knowledge base exist for frame and relation objects. For example, the user model base contains the ka_user_fobj and ka_user_robj files.

The knowledge administration type manager, knowadm, acts as a Prolog **resource manager**. That is, other inferential modules may *ask* the knowadm module for information about EDT, frame and relation types. The knowledge administration system contains additional programs and files to complement those for types and objects. During development of the input analyst module, need arose for natural language description of slots to clarify slot name mnemonics. In addition, identification of the prompt to be used when prompting the user to supply slot data and the tutorial file to be displayed if the user requested explanation of the slot were needed. Therefore, the **ka_slotdesc** file was created as an ancillary to the ka_frame_file. Its facts contain a natural language description of each slot defined for a frame type and the prompt and tutorial file associated with the slot. The *build_desc* program was written to prompt the system implementor for slot description data. The same program may be used to add new slot descriptions as new frame types are defined. Programs also exist to print or display frame definitions and objects.

Although not all of the functions provided by the knowledge administration complex are used by the Version 1.0 modules, representations of entites, classes of entities and the relationships between them can be defined and stored using the KA type and object managers. The type and object programs are the largest of the CODER retrieval subsystem inferential modules. The files containing the type definitions and object data are relatively small and are consulted as local fact bases by this version of the retrieval subsystem. However, incorporation of the NU-Prolog version of MU-Prolog will support the minor modifications required to establish the KA files as external knowledge bases supported by special Prolog extensions.

A facility for modifying type definitions does not exist. Due to the complex taxonomies which may be created by EDT and frame type structures and the inherent difficulty associated with such modifications, maintenance to type definitions can be applied by the system administrator only by means of editor software such as vi. It is hoped that the structure of frames to represent documents and document fields will be stable enough that lack of a maintenance facility will not stifle the system's capabilities.

4.2.3 Input Analyst and Report Modules

With the blackboard, strategist and knowledge administration modules written and tested, development of the retrieval system "front-end" as shown in Figure 23 was targeted as part of this research. The CODER system design did not include details concerning the flow of information to and from the user. Review of the distributed problem treatment (DPT) model discussed by Belkin et al. [BELK 83] supported the decision to strictly control the flow of information between the user interface manager and the CODER community of experts. That is, rather than allow any module to send requests to the user interface manager, only two modules communicate with the user interface manager. The *Input Analyst* (IA)[8] receives all input *from* the user via the interface, and the *Report Generator* sends all system generated output *to* the user via the interface.

[8] The term *input analyst* has been adopted from the work of Belkin et al.

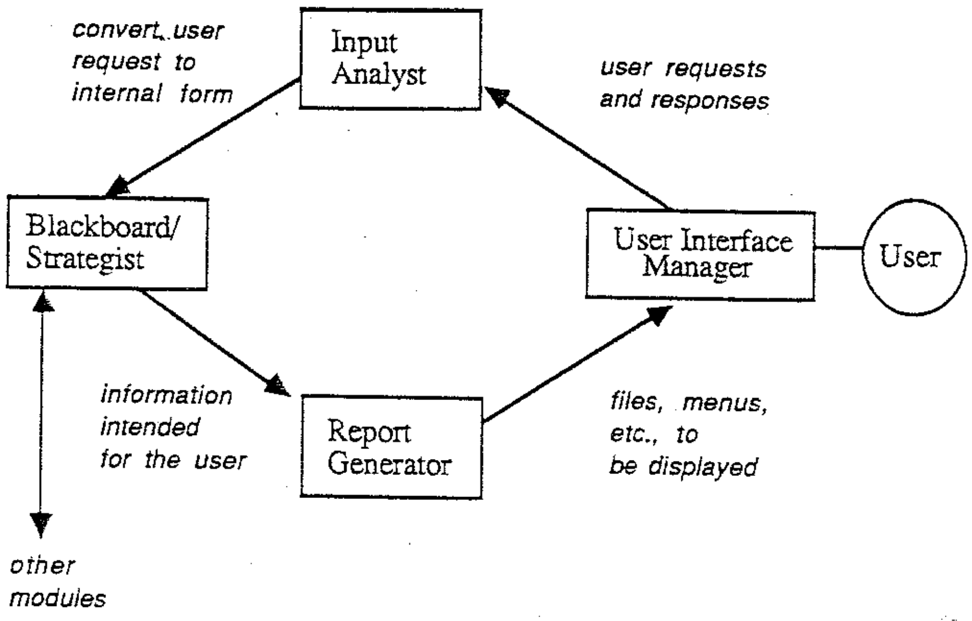

convert user request to internal form
Input Analyst
user requests and responses
Blackboard/ Strategist
User Interface Manager
User
information intended for the user
Report Generator
files, menus, etc., to be displayed
other modules

Figure 23. Retrieval System Front-End

As part of a Master's project, a generic user interface was written using the UNIX *curses* package [KHAN 88]. The user interface manager will not be discussed in detail here; however, the input analyst and report modules, developed and tested as part of this research, will be described in this section.

4.2.3.1 Input Analyst (IA)

Every user response to menus or prompts is passed directly from the user interface to the IA module. The module converts user input to an appropriate internal system form, determines on which area of the blackboard the input should be posted, and posts it as a hypothesis. The blackboard is the only module called by the input analyst, and the input analyst is the only module called by the user interface. It is also the only module which will **not** be scheduled by the strategist to perform tasks. The user interface manager, rather than the blackboard/strategist complex, provides all direction for the input analyst.

Prompts: The architecture of the prompt file was discussed in detail in chapter 3, section 3.1.3. When the user enters a response to a prompt, the response, validated by the user interface manager if required, is passed to the input analyst via the ***protocol(ia,resp(Response,Prompt_nbr))*** communications function. Based on the prompt number, the ia module may or may not store local information about the response, and will post a hypothesis to the blackboard indicating action to be taken based on the user response. For example, the user response to prompt number 53, "HAI lookup entry:", a term, phrase or person's name, is processed by the input analyst as follows:

1. Domain knowledge within the IA module indicates that prompts 52 thru 54 are requests to browse the HAI.
2. A local fact, stored by the input analyst when a HAI browse menu option was selected, identifies the type of browse, e.g., index subject, person or italicized references, requested.

3. The hypothesis ***hai_req(Type_of_browse,Response,Time)*** is posted to the ***user_req*** area of the blackboard. The strategist will subsequently schedule the browse expert to locate the requested information in the HAI so that it may be presented for user browsing.

NISO Command Input: The proliferation and diversity of online interactive information retrieval systems has resulted in different vocabulary and syntax for commands which perform the same function, but on different systems. To provide users with a common command language, the National Information Standards Organization (NISO) has defined NISO standard Z39.58-198. "The standard specifies the vocabulary, syntax, and operational meaning of commands in a command language for use with online interactive information retrieval systems." [NISO 87]. The CODER IA module includes processing of applicable NISO commands. Not all commands, such as database selection, are required by the CODER retrieval subsystem and only those functions which Version 1.0 modules can process are included.

Commands which may be entered in a designated window or in response to a NISO command entry prompt include:

EXPLAIN **[topic]** If entered alone, a list of topics that can be explained will be displayed. When entered with a topic, a tutorial file or menu for the topic will appear.

HELP Provides online assistance specific to the current situation.

STOP To terminate a session. This command will cause the system state to be changed to *user_done* and will initiate the user evaluation segment of the session.

FIND query To enter a search statement in Boolean format (not implemented). For example, *FIND TI treaties AND alliance.*

SCAN **[term]** To view an ordered list of search terms. This command cannot be fully implemented until the dictionary of search terms is provided. Then, when the user enters *scan some_term*, a list of terms, either related to the term entered, or in alphabetical sequence, may be displayed for user browsing.

RELATE **[term]** To view terms logically related to a search term. If entered alone, this command will cause the lexicon related term menu to be displayed. If entered with a term, the lexicon browse module will be scheduled to find related terms.

DISPLAY **[options]** To view the results of searches of the database.

PRINT **[options]** To request offline printing of search results.

SORT **[options]** To arrange records in search result sets by specified field.

REVIEW [datasets] To view search history, that is, search statements. This command will perform as does the "document archive" option of the browse main menu.

Although these commands are included in the input analyst module, and will be posted to the blackboard, the ability to properly perform the functions indicated by the commands is dependent on other modules. Only the help command functions properly at present. The majority of the commands listed depend on the search engine and the lexicon, both of which are still under development by other graduate students.

Structured Knowledge Entry: Approximately half of the Prolog code for the IA module is devoted to processing the user's entry of structured knowledge. The term ***structured knowledge*** refers to entry of data to fill one or more slots in a frame. Frames include issues, messages within issues, document types, names, addresses and dates. The current CODER system frames types, stored in the ka_frame_file by the KA complex, appear in Appendix G. Frame types were defined by Qi Fan Chen, the graduate student working on the CODER analysis subsystem. The input analyst creates frame objects containing user input and posts them to the blackboard where they can be viewed by the search expert to be matched to frame objects previously created during document analysis by the analysis subsystem.

When the user indicates the ability to provide information about canonical structures in the database being searched, the input analyst posts a hypothesis indicating that a frame has been requested. The report module, scheduled by the strategist, begins to prompt the user to enter slot information. As information is entered, the input analyst stores the responses so that a frame object can be created when the user has completed slot entries. Since slots of frames may be other frames which may have slots which are other frames and so on, all lowest level frame objects must be created first so that frame object identifiers may be used to set slot values.

Additional complications arise because some slots which are frames, for example the ***postal_address*** slot in the ***individual*** frame, may be generic classifications of more specific frame types, such as

U_S_mail or ***non_U_S_mail***. To narrow the user's search, the most specific frame desired is the one which should be created and matched to frames representing documents. Therefore, the IA module creates an indented tree structure of subframe types. Following display of the tree, the user is instructed to select the most appropriate frame type, and will be prompted to enter slot values for that frame type only. The structures and frame manipulations provided by the input analyst are used to test the hypothesis "Users can perform more effectively when structured knowledge is employed."

Callable Functions: The functions which the input analyst can process and the hypotheses which it posts to the blackboard are listed in Table 2. Many of the callable goals could be processed in one goal by the IA module. For example, where goals are passed as menu file parameters (see Figure 5), many goals could use the same Prolog functor and supply its arguments as the functor and arguments to be posted. The IA module could append a time to the arguments and then post the predicate to the blackboard. However, to maintain clarity of processing and to avoid confusion regarding the source of posted predicates, this strategy was avoided for the first implementation of the system. Because all goals are distinctly defined, persons developing future versions of CODER will be able to easily trace the source of blackboard postings.

4.2.3.2 Report Module

The CODER report generator module sends all information to be displayed to the user interface manager. It is scheduled by the strategist based on internal system processing or on user inputs channelled through the input analyst. The report generator is the only CODER module which directly sends information to be displayed or edited to the user interface manager. It controls the display of menus, files, editors, prompts and messages; this module also formats requested browse data for user viewing, and determines in which windows data will be displayed.

Table 2. Input Analyst Callable Goals

CALLABLE GOAL	FUNCTOR in HYPOTHESIS	AREA	COMMENT
setup_screen	state	state	initiates system states
menu(Menu)	disp_menu	user_req	menu requested
hai_req(Request)	disp_prompt	user_req	HAI browse request; lookup entry prompt.
lex_req(Request)	disp_prompt	user_req	Lexicon browse request; lookup entry prompt.
um_req(Request)	um_req	user_req	User Model browse request
tut(Tutfile)	tut	user_req	tutorial file request
displayf(File,Hdr,Next)	displayf	user_req	file display needed
state(State,Trans)	state	state	state transition indicated
pmsd_menu(Resp,Menu)	prob_resp	problem	response to a problem mode/state/descr menu
um_menu(Resp,Menu)	um_resp	umodel	response to a user model menu
resp(Resp,Promptid)*	lex	user_req	prompt indicates lex browse
	hai	user_req	prompt indicates HAI browse
	prob_resp	problem	response is for a problem mode/state/descr prompt
	um_resp	umodel	response is for a user model prompt
	disp_frame	user_req	if slot frame has no subframes
	displayf	user_req	if slot frame has subframes, indented tree is displayed.
	disp_prompt		
	nextslot	user_req	display next slot of frame
frame_req(Frame,Rel)**	relation	user_req	any relation objects for previous frame are posted.
	frame	user_req	any frame objects for previous frame are posted.
	disp_frame	user_req	if frame has no subframes.
	displayf	user_req	if frame has subframes, indented tree is displayed.
	disp_prompt	user_req	
frame_exit	done_frames	user_req	user is done frame entries

**Action taken depends on Promptid*

***Action taken depends on state of processing*

The report module assumes some of the roles that would be assigned to a ***dialog*** expert if one were included. That is, it manages windows and controls what the user sees. However, a true dialog expert would "determine the type of dialogue appropriate to the given context." [BELK 83]. Depending on the type of user, the problem state and the problem type, a dialog expert would determine the type of dialog used to elicit information from the user and to inform the user of the system's progress and intentions. The report module, on the other hand, uses only the type of dialog, for example the prompt of display message, indicated by the hypotheses which other modules post to the blackboard. It makes its inferences based only on blackboard information and on its own knowledge about current and previous data displayed on the user interface windows.

To instruct the user interface to display information, the 'window' function and the 'ask' predicate may be used as follows:

ask(user_interface,window(Win_num,Win_title,Data,Option,Clearflag)).

where

Win_num: 1-6 to indicate the window to be used,

Win_title: any title for the window, or " ",

Data: a prompt number, message string, or the name of a file.

Option: Options indicate the action that the user interface manager is to take. The options provided appear in Table 3.

Clearflag: 0 = no clear, 1 = clear.

Since all calls to the user interface manager are located in one module, replacement of one user interface with a different one will require changes to the report module only. If the same generic functions are performed by a new user interface, then no changes to the report module would be necessary.

The report generator module formats textual data requested for browsing. For example, lexical definitions or related terms are posted to the blackboard as *lists* of data items. The report expert converts the internal system formats into structures suitable for user viewing. Portions of text from

Table 3. User Interface Manager Options

Option	Data
0 = clear	Clear window (for multiple windows)
1 = display file	File name, including path
2 = display menu	Menu file name, including path
3 = display error	Error message string
4 = edit file	File name, including path
5 = display prompt	Prompt number
6 = do nothing	None (for multiple windows)
7 = display message	Message string

the HAI are also structured by the report module so that different text references and matched terms or names are distinctly displayed. Additional details are provided in section 4.2.4.

Structured Knowledge Entry: In conjunction with the input analyst module, the report generator determines the prompts required for user entry of structured knowledge. The ***ka_slotdesc*** file is consulted to determine prompts to be displayed by the user interface manager for each slot of a requested frame. Not all slots in every frame will have associated prompts. For example, the user would not be asked to enter data into the *dig_id* slot of the *issue* frame (see Appendix G) since that slot contains a value assigned by the CODER analysis subsystem.

The ka_slotdesc file, discussed in the previous section, contains facts in the form:

ka_slotdesc(Frametype,Slot,Class,Prompt,Slot_frametype,Tutorial).

The ***Class*** argument indicates whether the slot is to contain an EDT, frame or relation object. If the ***Prompt*** number is not zero, the report module asks the user interface manager to display the prompt identified by the number entered as this argument. Where slots are frames, the associated prompt is a yes/no prompt to determine whether the user has information about the frame. If the user responds positively, the report module proceeds to prompt the user for the slots for that frame. When prompts for all slots have been displayed, the report module uses local knowledge to return to the prompt for the next slot of the original frame. As with the input analyst module, the processing to support structured knowledge entry is somewhat complex, and is included to test the hypothesis regarding use of structured knowedge in an intelligent information retrieval system.

Callable Functions: The blackboard hypotheses which the report module can process appear in Table 4. The only functions which the report module ***requests*** are those processed by the user interface manager. Functions listed as ***callable goals*** represent hypotheses which other modules may post to the blackboard. The *Tm* argument is a distinct timestamp needed to differentiate hypotheses which have the same functor and other identical arguments. For example, the user may

Table 4. Report Generator Callable Goals

CALLABLE GOAL	OPTION in ASK to USER INTERFACE	COMMENT
setup_screen	setup_screen	initializes screen window
tut(Tut,Depends,Tm)	display file	tutorial file is displayed
displayf(File,Hdr,Nxt,Tm)	display file	requested file is displayed
disp_menu(Menu,Dep,Tm)	display menu	menu file is displayed
disp_frame(Frame,Rel,Tm)	display prompt	prompt for slot entry
done_frames	display file	display Thank you file
nextslot(Frame,Slot,Tm)	display prompt	prompt for next slot of frame
disp_msg(Msg,Tm)	display message	display text message
disp_msg(Msg,Value,Tm)	display message	append dynamic Value to Msg
disp_error(Msg,Tm)	display error	special error processing
disp_prompt(Prompt,Hdr,Tm)	display prompt	display prompt, send response
editf(File,Hdr,Depends,Tm)	edit file	display file for editing
lex_output(Kind,Term,Result,Tm)	display file	format lexical output
hai_output(Kind,Term,Result,Tm)	display file	format HAI output

request the same browse function many times during a single session. Without the timestamp, the hypothesis would be rejected by the blackboard with the message "*Warning - hypothesis already exists with equal or higher confidence."

4.2.4 Browsing

Two inferential browsing modules are included in the Version 1.0 Coder system. The ***lexical*** expert accesses the facts derived from the **Collins Dictionary of the English Language**. The ***browse*** expert extracts text relevant to a user's request for information from the three volume **Handbook of Artificial Intelligence**. A CODER resource manager, ***hai_mgr***, has been written to extract indicated text from specified files and line number ranges. A similar program, ***text_mgr***, extracts text from the document collection so that retrieved documents may be viewed. The modules to support browsing were prototyped by graduate students during Spring quarter, 1987 [BARN 87, BISH 87], and the databases browsed were created earlier by other graduate students [WOHL 86, WENB 86]. The user interface manager [KHAN 88] provides the ability for users to select words or phrases via a program function (PF) key. Selected words or phrases may be displayed during query entry so that users may incorporate them into queries as desired. No **automatic** inclusion of selected terms is available in this version of the retrieval subsystem.

Both inferential browsing modules require additional enhancements. The lexical module must be adapted to handle modifications made to lexicon facts as part of a separate research project, "Organizing Lexical Knowledge for Information Retrieval."[9] The browse module must be expanded to include enhanced browsing of text in the document collection. Implementation of natural language query processing would require additional functions in the modules.

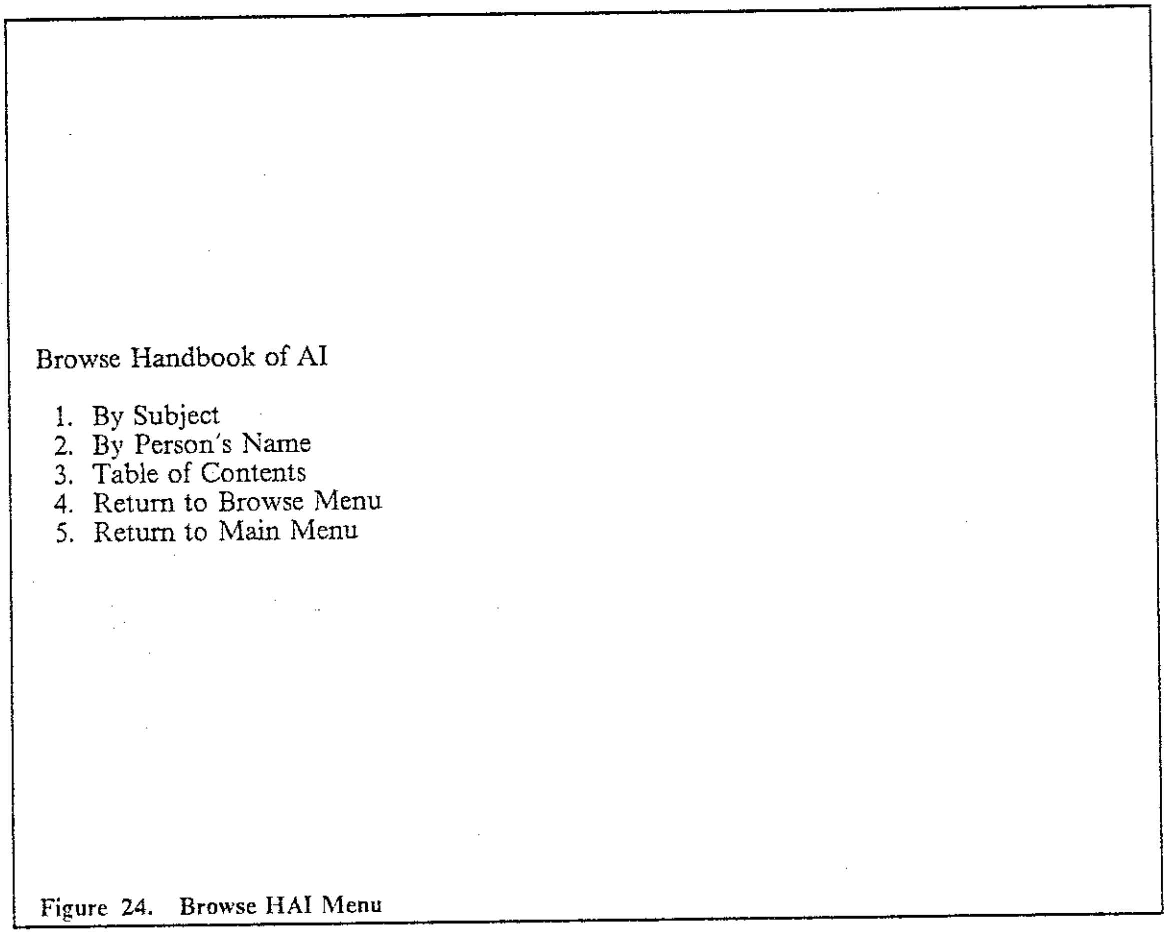
[9] The lexicon project is funded by National Science Foundation Grant IRI-8730580.

4.2.4.1 Browsing the HAI

Based on a user's request to browse the **Handbook of Artificial Intelligence**, the browse expert uses Prolog facts derived from the HAI to determine which sections of text are relevant to the user's request. Separate sets of Prolog facts, created to fulfill class project requirements during Spring quarter, 1986 [WENB 86], exist for subjects, index entries, italicized entries, person's names and the HAI table of contents. Each fact includes one or two file names (out of 106) and the line number or range of line numbers from which the reference was extracted. The current version of the CODER retrieval subsystem supports menu driven browsing of the HAI. When HAI browsing is requested, the menu in Figure 24 appears.

HAI Resource Manager: For each option selected, the user will be prompted with a secondary menu or with a prompt for entry of a subject or person's name. If the browse expert matches a user request to a HAI fact, the hai_mgr will be requested to extract the text and place it in a temporary file. The original hai_mgr was written as part of a Master's project [KHAN 88], and has been rewritten to include C communications functions. Before the hai_mgr can access a HAI file, the file must be preprocessed so that its lines may be indexed. The browse expert maintains a local fact base of files indicating which have been preprocessed. If a file has not been preprocessed, the browse expert issues ***ask(hai_mgr,preprocess(Filename))***. Since all arguments will be instantiated at the time the ask is issued, the hai_mgr and browse expert may execute concurrently. The hai_mgr also contains callable functions for extracting single lines or blocks of text. The functions which may be requested of the HAI resource manager appear in Table 5.

HAI Browse Expert: When the browse by **subject** option is selected, a secondary menu prompts the



Browse Handbook of AI

1. By Subject
2. By Person's Name
3. Table of Contents
4. Return to Browse Menu
5. Return to Main Menu

Figure 24. Browse HAI Menu

Table 5. HAI Resource Manager Callable Goals

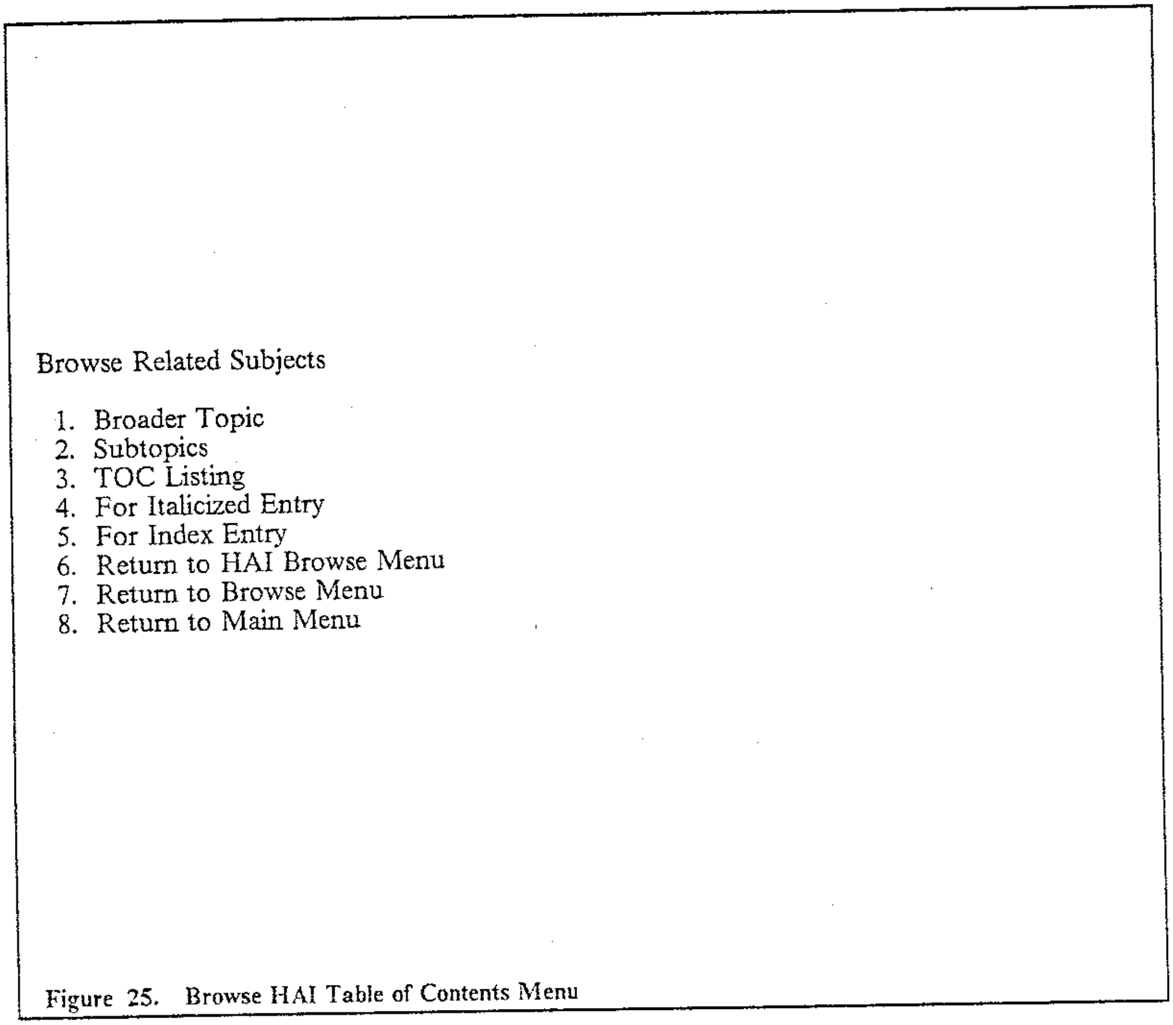
CALLABLE GOAL	COMMENT
preprocess(File)	Indexes HAI text file by line offset
extractline(File,Line,Tempfile)	Returns Tempfile as name of file containing relevant text.
extractbloc(File,Begin_line,End_line,Tempfile)	Returns Tempfile as name of file containing relevant text.
extractbloc2(File1,File2,Start1,End2,Tempfile)	Text spans 2 HAI files; text begins at Start1 in File1 and ends at End2 in File2. Returns Tempfile as name of file containing relevant text.

user to select browsing by index entries, italicized entries [10] or both. The user will then be prompted to enter a term or phrase which the browse expert attempts to match to HAI subject reference facts. Only an **exact match** will result in text extraction. Moreover, line numbers or ranges of lines in all HAI Prolog facts indicate only the lines which contain the matched terms. Therefore, an arbitrary number of lines surrounding the matched lines are extracted and placed in a temporary file. The name of each temporary file created is posted to the blackboard; all files containing relevant text will subsequently be concatenated and formatted for viewing by the report generator module. A more "intelligent" version of the browse expert or of the program which extracted HAI references originally should identify optimal line number ranges of paragraphs or other blocks of text which are relevant to the subject referenced.

Unlike subject browsing, browsing by a person's name does not require an exact match. Most names found in the HAI are stored in Prolog facts as a single argument containing 'last name, first name or initial, middle initial'. A parser was written to create additional facts which segregate the components of the person's names. When a search by name is indicated, the browse expert first attempts an exact match. If that fails, it tries to match as many as possible of the name components supplied by the user. For example, if the user only enters a last name, then all references to any person having that last name will be extracted. However, if the user enters a last name and first name, exact matches excluding middle intial, and matches to last name and first initial will be extracted. Naturally, some matched references will not give the user precisely what he/she was looking for. For example, a user's entry "Marigold Minsky" would be matched to facts containing the person 'minsky, m.', likely a reference to Marvin Minsky.

The third option, **Table of Contents**, allows the user to browse the HAI table of contents or to request broader or narrower terms for a given term or phrase. The Table of Contents menu is shown in Figure 25. A portion of the HAI Table of Contents (TOC) appears in Figure 26. Either the

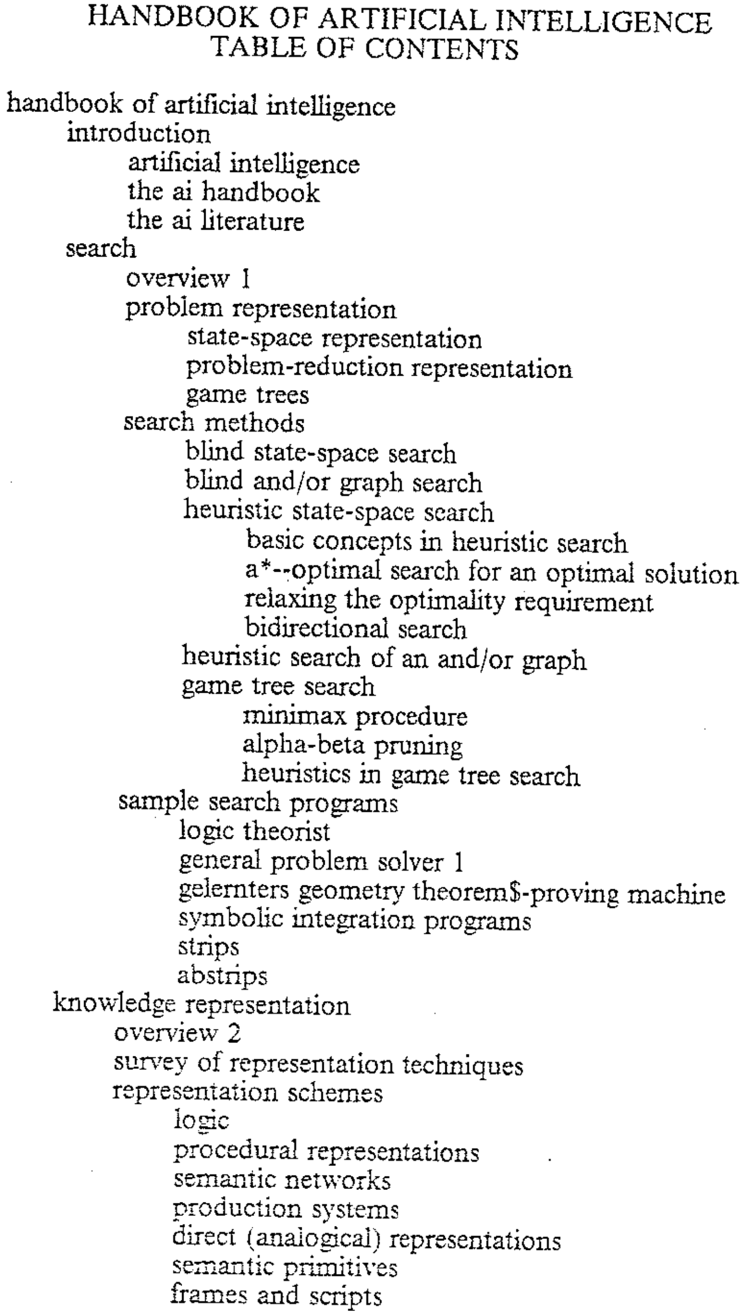
[10] Due to the size of the file containing italicized entry facts, only a subset of the italicized entries is included in this version. Implementation of external knowledge base functions will allow inclusion of all italicized entry facts.



Browse Related Subjects

1. Broader Topic
2. Subtopics
3. TOC Listing
4. For Italicized Entry
5. For Index Entry
6. Return to HAI Browse Menu
7. Return to Browse Menu
8. Return to Main Menu

Figure 25. Browse HAI Table of Contents Menu



```
HANDBOOK OF ARTIFICIAL INTELLIGENCE
        TABLE OF CONTENTS

handbook of artificial intelligence
    introduction
        artificial intelligence
        the ai handbook
        the ai literature
    search
        overview 1
        problem representation
            state-space representation
            problem-reduction representation
            game trees
        search methods
            blind state-space search
            blind and/or graph search
            heuristic state-space search
                basic concepts in heuristic search
                a*--optimal search for an optimal solution
                relaxing the optimality requirement
                bidirectional search
            heuristic search of an and/or graph
            game tree search
                minimax procedure
                alpha-beta pruning
                heuristics in game tree search
        sample search programs
            logic theorist
            general problem solver 1
            gelernters geometry theorem$-proving machine
            symbolic integration programs
            strips
            abstrips
    knowledge representation
        overview 2
        survey of representation techniques
        representation schemes
            logic
            procedural representations
            semantic networks
            production systems
            direct (analogical) representations
            semantic primitives
            frames and scripts
```

Figure 26. HAI Partial Table of Contents

entire TOC or just supertopics and subtopics for one of about 220 subjects in the TOC may be requested. The subject browsing options are also included so that users need not navigate the menu structure if they wish to see more information about a listed subject.

Callable Functions: Table 6 lists the callable goals in the browse expert.

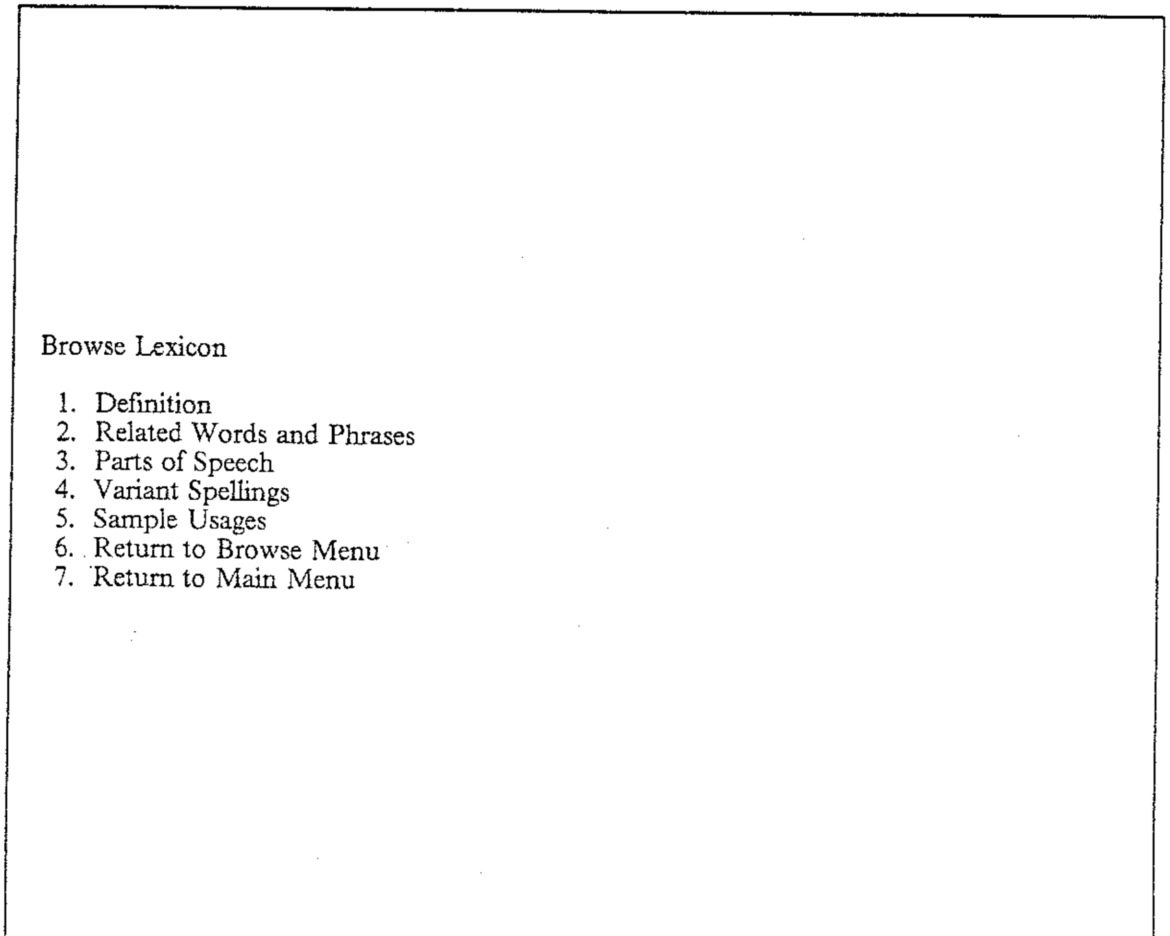
4.2.4.2 Browsing the Lexicon

The lexical expert accesses the Prolog facts contained in twenty-one relations derived from the Collins dictionary [WOHL 86]. Presently, the lexical expert only supports browsing of definitions and their components in the dictionary. Implementation of natural language parsing of queries may require additional functions in the lexical expert. Representing approximately 85,000 headword entries, the lexical facts are too numerous to be loaded into local Prolog fact bases. Therefore, only a subset of each of the relations has been used to develop the lexical expert. As part of the lexicon research project previously mentioned, methods using Prolog external knowledge bases, B trees, or C programs for searching the lexical fact bases are being explored.

When lexicon browsing is requested, the menu in Figure 27 is displayed. All options will eventually result in a prompt to the user to enter a lexical term. No stemming of terms or morphological analysis is presently provided by the system. The **Parts of Speech** and **Variant Spellings** options simply match the term entered to a headword stored in Prolog facts for part of speech or variant spelling. A list of parts of speech or alternate spellings is posted to the blackboard and subsequently formatted for viewing by the report module. The **Definition** option results in extensive navigation of the hierarchy of facts stored for each headword entry. A list containing the textual definition and all components of the definition, such as abbreviations, parts of speech, synonyms, variant spellings

Table 6. Browse Module Callable Goals

CALLABLE GOAL	FUNCTOR in HYPOTHESIS	COMMENT
hai(Function,Subj,Tm)	hai_output	functions are indicated by menu options selected. i.e., get_aih_person, get_index, get_subtopics, get_rel_subj, get_italics, get_italics_span, get_supertopics, find_hierarchy.
results	text_output	display documents retrieved



Browse Lexicon

1. Definition
2. Related Words and Phrases
3. Parts of Speech
4. Variant Spellings
5. Sample Usages
6. Return to Browse Menu
7. Return to Main Menu

Figure 27. Browse Lexicon Menu

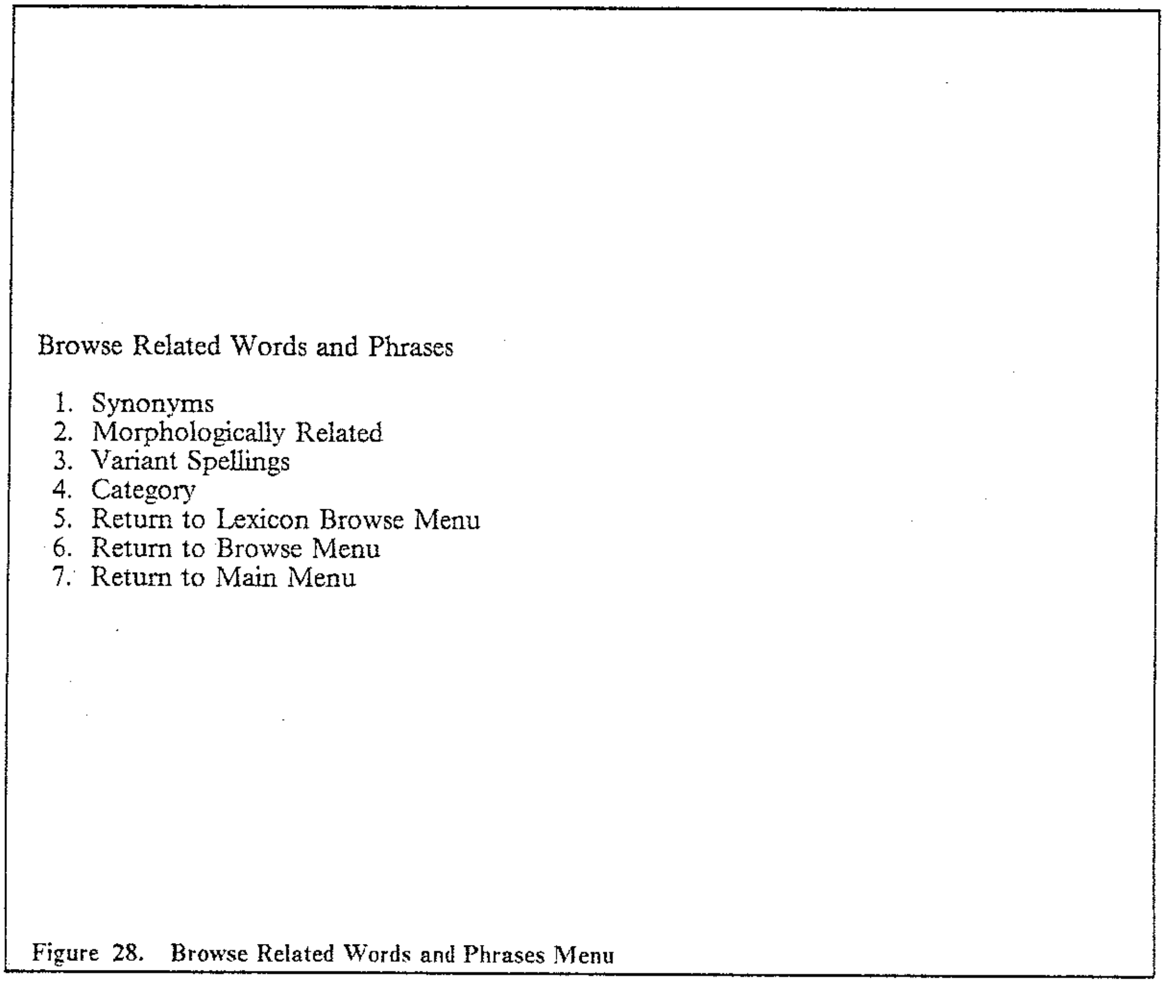
and morphological variations, is posted to the blackboard and processed by the report generator for display to the user.

The **Related Words and Phrases** option results in display of a secondary menu shown in Figure 28. Results displayed for each option are derived directly from the Prolog facts for the corresponding relation. As those relations and their derivation are discussed in detail in other reports [WOHL 86, FOXE 86d] they will not be rediscussed here.

Callable Function: At present, only one goal is recognized by the lexical expert: *lex(Function,Word,Timestamp)*. Functions are indicated by menu options selected during lexicon browsing, and include the following:

define	a definition
relt	related terms
use	sample usages
pos	parts of speech
varspell	variant spellings
morph	morphological variations
abbrev	abbreviations
synonym	synonyms

For each function, the hypothesis *lex_output* will be posted with the browse data requested.



Browse Related Words and Phrases

1. Synonyms
2. Morphologically Related
3. Variant Spellings
4. Category
5. Return to Lexicon Browse Menu
6. Return to Browse Menu
7. Return to Main Menu

Figure 28. Browse Related Words and Phrases Menu

4.2.5 Problem Description Builder

One of the primary functions of the CODER system, as with any information retrieval system, is to create a representation of the user's problematic situation. Users are often unsure of the precise nature of their problems; therefore, the system must attempt to obtain as much information as possible about the user's problem so that a model of the problem may be generated. The goal of the ***Problem Description Builder*** is to create this model. The model will be used by the Query Formulator expert to post a searchable query. In addition, information collected about the user's problem situation will be stored in *session* frames per user.

The Problem Description builder, identified by the system as ***probmsd***, builds a description of the user's problem state for a given search and controls the transition of stages in the retrieval process. It combines the functions defined by N. Belkin, P.J. Daniels and H. Brooks for the Problem Mode, Problem State, and Problem Description modules of an intelligent information retrieval system. Subgoals of these functions have been classified for the CODER system as belonging either to the *document, real* or *system* world. Per Belkin, Brooks, and Daniels, the problem mode, state and description are defined as follows.

Mode determines the system mechanism to be used and explains system capabilities to the user.

State determines the position of the user in the problem treatment process, for example, whether the user is just beginning search, refining the search or is looking for a specific document which he/she has already seen.

Description includes subject, context, terms, research area and subject literature references.

The approaches of the aforementioned researchers have been integrated into the CODER view of the Problem Description. Three decompositions of problem description have been reviewed and are discussed here briefly.

1. *per Belkin et al. [BELK 83]*, the problem description consists of 4 functions:
    a. problem *type*: for example, procedural, decision-making or learning.
    b. problem *structure*: whether the problem is coherent, unstructured or has gaps.

   c. problem *topic*: subject matter, terms and topics.

   d. problem *context*: purpose of the query, user's research area, user's area of interest, etc.

2. *per Daniels et al. [DANI 85]*, the problem description consists of 5 subgoals:

   a. *topic*: subject matter, terms and topics.

   b. *research*: user's area of research and/or research topic.

   c. *subject*: subject background, a broader subject area than topic.

   d. *document*: content of the documents the user wishes to retrieve.

   e. *subject literature*: literature of the subject domain already known to be relevant, such as key authors or works.

3. *in the CODER system*, the problem description contains 3 worlds:

   a. The *document* world deals with the physical/logical structure of documents to be retrieved, for example, message relationships within documents. This world is related to Daniels' document subgoal.

   b. The *real* world, the largest of the 3 worlds, contains the context, topic, terms and subject area to be retrieved. This world would include the topic, research, subject, and subject literature subgoals discussed by Daniels, et al. as well as Belkin's topic and context functions.

   c. The *system* world addresses functions of the information retrieval system being used, for example, the quantity of documents desired, sorting of retrieved documents or recall/precision level expectations.

Each of the three CODER worlds has been examined in light of the following:

- problem description subgoals and functions discussed by Belkin, Daniels, and Brooks;
- the CODER knowledge administration complex;
- message relations and AIList document structure; and
- sample queries generated by graduate students for different issues of AIList Digest.

Each *world* is discussed below. Where frames are listed, it is intended that the input analyst module, and later classification experts, may post partially filled frames. For example, the name frame for an individual may contain only the last name of the sender of a digest message; the first and middle name slots may be empty. The 'post_frame' predicate, one of the knowledge administration local frame predicates, may be used to post partially filled frames to the blackboard. Where elementary data items, lowest level data, are listed, it is intended that this data be posted as standard blackboard hypotheses.

Document World: The document world refers to the physical and logical structure of documents. It may contain EDTs, frames or relations to represent the physical or logical structure of documents.

1. *Frames* contain the user's entry of structured knowledge. Such frames may include:

   - Digest issue frames containing issue attributes such as volume, issue, topics and issue date;
   - Digest message frames having message attributes like sender of message and date sent;
   - Document type frames representing document logical structure, for example, seminars, conference announcements, news reports, humor, etc. Document types are not yet fully implemented in the analysis subsystem. The seminar announcement document type is the only type currently recognized by the analysis subsystem.

2. *Relations* indicate relationships among document physical or logical components.[11]

   - Message relations represent document logical structure and may include relations such as those depicted in Table 7 on page 115.
   - Physical relations represent document physical structure and may include relations such as those listed in Table 7 on page 115. Note that many of the document world frames contain slots which overlap with information which is relevant to the real world. For example, the digest_message frame contains slots for sender, an individual, and date_sent, a date frame.

Real World: The real world contains the context, topic, terms and subject area of the user's problem situation.

1. *Frames* contain the user's entry of structured knowledge.

   - Frames about individuals contain slots for names, addresses and affiliations of people. Individual frame types are used for message author, sender, person to whom reply should be sent, person to contact about a seminar, etc.
   - Organization frames include any one of the frames within organization frame hierarchy, such as educational institution, corporation or government agency. These frames have not yet been implemented by the analysis subsystem.
   - Address frames may include either postal or electronic mail address information.
   - Journals, articles, books and other citations are represented by bibliographic reference frames. These frames, when implemented, will address Daniel's "subject literature" subgoal.
   - Date frames will be posted with the relations: on, before or after. The user is prompted with a menu to indicate whether the time period for which the search is requested is on a given date, before a date, after a date or between dates.

[11] These relations are not yet implemented in the analysis subsystem

Table 7. Sample Message and Physical Document Relations

Sample Message Relations:

copyof
annotationof
citation
referto
samedigest
quotation
excerptfrom
beforedigest
replyto
by
afterdigest
aroundsametime

Sample Physical Document Relations:

centering
tables
middleofdoc
block
underlined
capitalized
multicolumn
page(X)
wholemsg
list
memo
topofdoc
paragraph
sentence
sectionhdgs
bottomofdoc
field
figures

2. *Relations* for the real world include relationships among terms, such as synonymy, broader term or verb relations. These relations are not included in Version 1.0. Presently, relationships among headwords in the lexicon, such as synonymy and broader terms, are indicated by the lexicon facts *c_ALSO_CALLED, c_COMPARE, c_RELADJ* and *c_CATEGORY*.
3. *Elementary Data* items include words and phrases to be included in the query formulation, for example topics, hardware, software or user's research area.

System World: The system world includes information required by the CODER system for its processing.

1. *Frames* such as those for user modeling may help to determine search strategy.
2. *Elementary Data* items provide most of the other system world information.
   - Quantity of documents the user wishes to retrieve may be an integer or 'all'.
   - The time the user has available to do the search should be considered.
   - Recall and precision level expectations may help to determine the search strategy.
   - Sorting of retrieved documents may be by relevance, most recent or author's last name.

In addition to the structured knowledge supplied by the user, information is explicitly acquired using the prompt and menu mechanisms. Sample prompts and menus used by the problem description builder to acquire document, real and system world information appear in Figure 29 on page 117.

Probmsd Functions: The probmsd expert performs the following functions:

1. It implements a nondeterministic finite state automaton based on the CODER retrieval subsystem flow developed in June, 1987 (see Figure 30 on page 119). The *state* hypothesis posted by experts will prompt the domain task scheduler to notify the probmsd. If the predicate in a hypothesis matches a transition arc for the current state of retrieval, the probmsd expert will "jump" to the next state and post a hypothesis to initiate the action(s) for the new state. Facts representing the finite state machine are of the form:

   fsm(Current_state,Transition,Next_state,Bboard_area).

   If the new state is one which the probmsd module should process, indicated by a Bboard_area

*** Sample prompts/menus for *System World*:

"How many documents would you like to retrieve?"
1. 1-5
2. 5-10
3. 10-20
4. 20-40
5. All

"Are you looking for particular Recall/Precision?"
1. Higher Recall
2. Higher Precision
3. Balance Recall and Precision
4. Don't Know

"Would you like the retrieved documents to be sorted by:"
1. Relevance
2. Author's last name
3. Date (most recent)

*** Sample prompts/menus for *Document World*:

"Are you looking for a specific, known document? (y/n/help)"

"What portion of documents would you like to retrieve?"
1. Whole document
2. Paragraphs
3. Document Header only

*** Sample prompts/menus for *Real World*:

"How far along are you in your search for information?"
1. Just beginning
2. Refining the search
3. Browsing
4. Other

"What authors have provided useful references?"

"Enter the titles of any books/articles which have been useful:"

"Do you wish to enter structured knowledge for query matching? (y/n/help)"

"Do you wish to enter terms for query matching? (y/n/help)"

Figure 29. Sample Problem Description Prompts/Menus

of *probmsd*, it will do so. Otherwise, the *Next_state* is posted as the functor of a blackboard hypothesis with timestamp as an argument. The area to which it is posted is contained in the *Bboard_area* argument.

2. Although the subgoals of Belkin and Daniels′ Problem Mode, Problem State, and Problem Description modules have been studied extensively, they are not incorporated in their entirety into this expert. Rather, the document, real, and system worlds devised for the CODER system direct the sequence of activity. The probmsd module contains local knowledge about the prompts and menus it must request for display to the user so that the problem description and state may be defined.

Goals which the probmsd module can process appear in Table 8 on page 120.

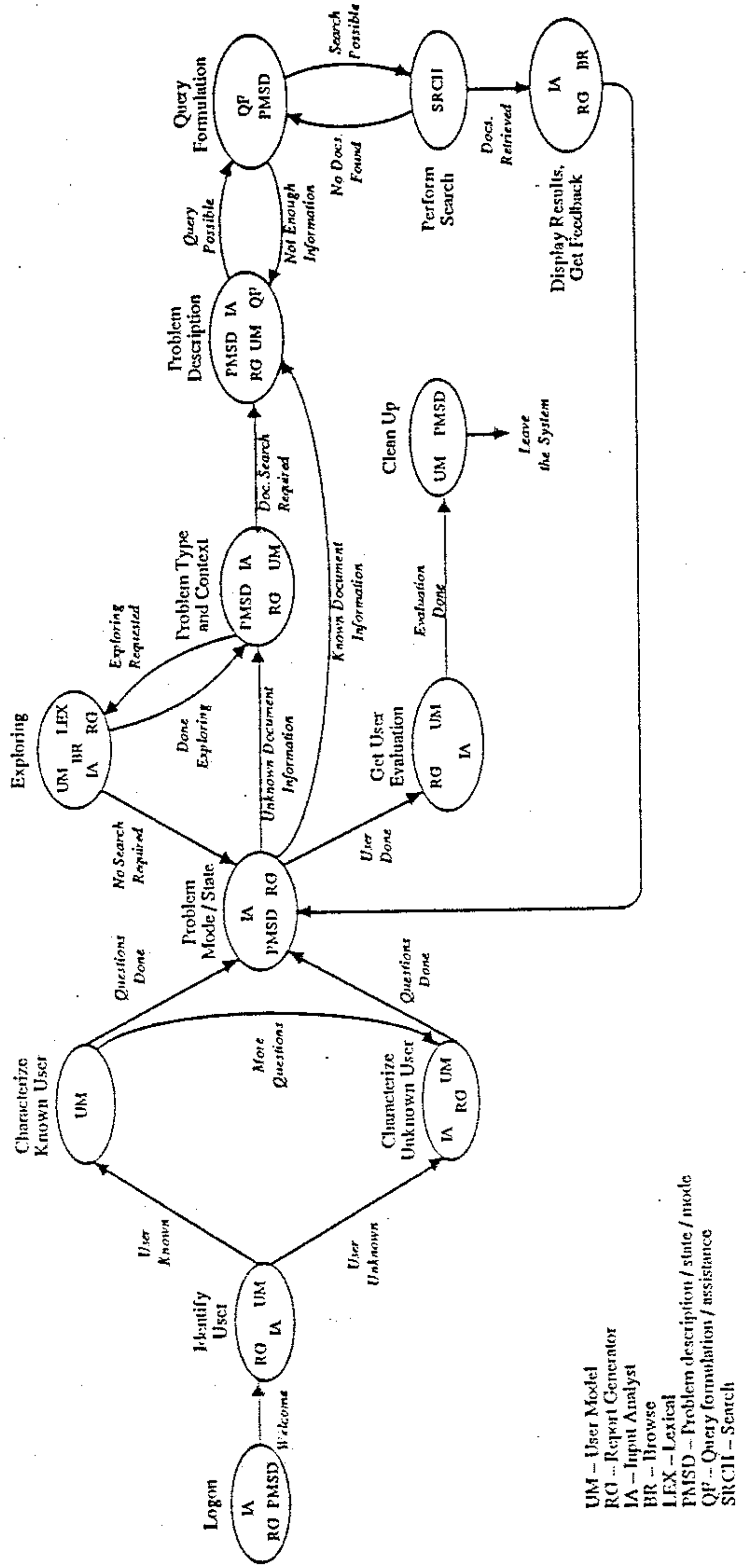


Figure 30. Retrieval State Diagram

Table 8. **Problem Description Callable Goals**

CALLABLE GOAL	FUNCTOR in HYPOTHESIS	AREA	COMMENT
state(Transition,Tm)*	id_user	umodel	identify user
	char_newuser	umodel	characterize new user
	char_olduser	umodel	characterize old user
	user_eval	umodel	user evaluation needed
	exploring	browse	user requesting browsing
	form_query	query	ready for query formulation
	search	search	ready for search of database
	results	results	retrieval results posted
	clean_up	cleanup	end of session
	disp_menu	problem	problem description transition
	disp_prompt	problem	problem description transition
	new_query	problem	new query, same session
	state	problem	transitions within probmsd
prob_resp(Resp,Id,Tm)	tut	user_req	user requested explanation
	disp_menu	problem	need more user info
	disp_prompt	problem	need more user info
	state	problem	got all user info for one phase; transition indicated
utype(User_type,Tm)	(none)		saves user type posted by umodel
done_frames(Tm)	state	problem	ia posted done frame entry

*Hypothesis posted depends on the current state and the transition arc.

4.2.6 User Modeling

The information retrieval user base is gradually shifting from a group of skilled intermediaries to a mass audience of users having diverse aptitudes, computer skills and needs. Intelligent information retrieval systems must support and adapt to a broad range of users, from novices to sophisticates; the retrieval system must fulfill the same functions that a human search intermediary performs. One of the primary functions performed is that of modeling the user.

When a user approaches a search intermediary with a database search request, the intermediary begins to mentally compile a set of characteristics about the user. As dialog between the user and the intermediary proceeds, the intermediary identifies personal characteristics and qualities about the user. Perceived user characteristics may be based on appearance, mannerisms, previous knowledge about the user, or user answers to explicit questions. Accordingly, the search intermediary will, for example, conclude whether the user is experienced or inexperienced, is just beginning the search or is refining it, has a well-formulated or a vague search request. Determining an appropriate dialog and the degree of assistance required for the user to formulate a precise query are functions that the intermediary performs. As an information retrieval system which employs AI methods to allow more effective retrieval, the CODER system attempts to simulate some of the functions normally performed by a human search intermediary.

The aim of the user model module of the CODER system is to identify relevant aspects of the user's short and long term goals, background, experience and knowledge. To accomplish its aim, the module uses the knowledge administration complex to build frames and relations for individual users of the system. From information collected during previous sessions as well as from user-supplied responses to menus and prompts, the user modeling expert builds frames for each user. Slot values in the frames determine the user stereotype to be posted to the blackboard; that type, as well as other information about the user, may assist CODER modules in determining action sequences and modes of interaction with the user.

User Model Frames: Based on research by P. Daniels [DANI 86b] frames containing information about the user have been defined. The user modeling subgoals specified by Daniels provided guidance for frame definitions. The subgoals include the following.

- The ***user*** subgoal determines the user's status, for example, graduate/undergraduate/faculty/staff for an academic user.
- The ***ugoal*** subgoal identifies the user's short and long term goals.
- Assessment of the user's state of ***knowledge*** about his/her problem situation is the third subgoal.
- The subgoal, level of ***experience*** with information retrieval systems, helps to determine the dialog mode and degree of interaction required between the system and the user.
- Finally, relevant aspects of the user's ***background***, such as education or employment, further characterize the user.

In addition to the user frames incorporating these five subgoals, frames have been created for storage of information about each different session in which a user is engaged. A **session** represents the time period from start-up of the CODER retrieval subsystem by a single user to termination by that user. An individual session may include multiple queries. Information accumulated during previous retrieval sessions is used to characterize users during new sessions. Like all of the Version 1.0 modules, the user model expert has been implemented as a prototypical module. Therefore, later versions will likely require expansion of the frames defined for this version of the system.

Seven frames have been defined for each user. All frames are either slots in the primary **user** frame, that is they are *part of* the user frame, or they are slots in the frames which are part of the the user frame. The user frame contains slots for *user identification, user type, frequency of use* and the following frames.

environment: preferred session environment, such as query type, document ordering and document quantity (experience subgoal). Preferences are inferred by reviewing up to ten of the last user sessions, and weighting the preferred environment selected during the current and previous session more heavily.

info: general information about the user's status (background and user subgoals).

knowledge: user's level of experience with computers and information retrieval systems (knowledge and experience subgoals).

loginfo: averages and totals for retrieval sessions; also contains the session slot for frames containing statistics about each session. The session frame contains user's long and short term goals (ugoal subgoal).

The user modeling frames and their slots are displayed in Figure 31 and Figure 32.

Sample prompts and menus used to explicitly acquire information about the user appear in Table 9. Whether prompts or menus are displayed depends on the type of user and the user's responses to prior prompts and menus. If the user has previously used the CODER system and information about the user's background and experience level has already been obtained, many of the user characterization prompts and menus will be bypassed.

Browsing the User Model: The user model expert also supports user browsing of information in the user model frames. Background, preferred environment or session statistics may be browsed by the user. Future versions of the user modeling module should allow a user to modify particular information in his/her user frames. In addition, the user should be able to browse a **document archive** of retrieved document ids and data about those documents which the user specified.

User Stereotypes: The model of the user contained in the user frames allows the user model expert to stereotypically classify CODER users. Pre-encoded assumptions about users provide three classifications: ***novice, average*** and ***expert***. When a user type has been inferred, the user model posts the type to the blackboard so that other modules may tailor processing accordingly. About a dozen rules, mostly in the query formulator and problem description builder modules, have been implemented in this version of the CODER system regarding how best to apply user classifications. Primarily, user types have been considered when determining the kinds of prompts and menus to be displayed; that is, the user stereotype affects dialog and degree of assistance provided. Advanced functions, such as applying user modeling data to determine whether to expand query terms, whether to provide general or specific references, or how to translate terms, have not yet been incorporated.

FRAME TYPE: *user*

Slot	*Class*	*Description*
userid	EDT	user identification
usertype	EDT	type of user (novice, average, expert)
freq_of_use	EDT	number of times CODER has been used
environment	frame	user environment preferences
info	frame	general information about the user
knowledge	frame	user experience and knowledge
loginfo	frame	session information

FRAME TYPE: *knowledge*

Slot	*Class*	*Description*
usedcomp	EDT	ever used a computer (yes or no)
cscrs	EDT	taken computer sci. courses
isrcrs	EDT	taken ISR courses
knowbool	EDT	familiar with boolean logic
otherisr	EDT	used other ISR systems (yes/no)
otherfreq	EDT	frequency if used other ISR systems

FRAME TYPE: *environment*

Slot	*Class*	*Description*
doc_qty	EDT	preferred document quantity
doc_ordering	EDT	preferred document sort
type_display	EDT	preferred portion of document
qtype	EDT	preferred query type (boolean, vector,...)
recall_precision	EDT	preferred level of recall/precision

FRAME TYPE: *info*

Slot	*Class*	*Description*
degree	EDT	highest educational degree obtained
educ_field	EDT	educational field for degree
firstlang	EDT	English as native language (yes/no)
gender	EDT	male or female
individual	frame	name, address, ... (not currently used)

Figure 31. User Model Frames

FRAME TYPE: *loginfo**

Slot	*Class*	*Description*
avgfdbk	EDT	average times feedback used
avglgth	EDT	average length of queries
avgqchgs	EDT	average changes to queries
session	frame	statistics per session
totdocs	EDT	total documents retrieved
totfdbk	EDT	total feedback searches
tothelp	EDT	total times help requested
totquery	EDT	total queries
tottime	EDT	total time on system

FRAME TYPE: *session*

Slot	*Class*	*Description*
nodoc_queries	EDT	nbr of queries with no documents found
fdbk	EDT	feedback query
qchgs	EDT	query changes
query	EDT	query
reldocs	EDT	retrieved document ids
sessionlgth	EDT	session length
session_id	EDT	session process id
nbr_docs	EDT	number of docs found
nbr_fdbk	EDT	number of feedback docs
user_eval	frame	user evaluation of session
doc_qty	EDT	document quantity requested
doc_ordering	EDT	document sort
qtype	EDT	query type
type_display	EDT	document portion
recall_precision	EDT	recall/precision level
research	EDT	research area
purpose	EDT	purpose of search

FRAME TYPE: *user_eval*

Slot	*Class*	*Description*
easy_to_use	EDT	system was easy to use (yes/no)
satisfaction	EDT	level of satisfaction (1 to 10)
usefulness	EDT	percent of documents found useful
why_stop	EDT	reason for stopping search

* Loginfo slots other than for session frame and total time on system are not yet filled.

Figure 32. User Model Session Frames

Table 9. Sample User Model Prompts and Menus

For *user characterization*:

"Enter a unique id (e.g., last name followed by first initial):"
"Have you ever used a computer? (y/n)"
"Have you taken Computer Science courses? (y/n)"
"Have you taken Information Storage & Retrieval courses? (y/n)"
"Are you familiar with Boolean logic (y/n)?"
"Is English your native language? (y/n)"
"Enter your gender (m = male, f = female):"
"Have you ever used an Information Storage & Retrieval System? (y/n)"

"How many times have you used Information Storage & Retrieval Systems?"
1. 1-5
2. 6-10
3. 10-25
4. over 25

"What is the highest level of education you have achieved?"
1. High school diploma
2. Two or more years college
3. Bachelor's degree
etc.

"In what field is your degree?"
1. Computer Science
2. Engineering
etc.

"What area of research are you currently pursuing?"
1. Knowledge Representation
2. Natural Language Processing
etc.

For *user evaluation of session*:

"On a scale of 1-10, (1 = dissatisfied, 10 = satisfied), enter satisfaction:"

"Was this system non-frustrating and easy to use? (y/n)"

"Did you stop searching because you:"
1. found what you wanted
2. found enough information
3. are frustrated with this system
4. ran out of time

"Please estimate the percentage of documents retrieved that were useful"
1. under 20%
2. 20-33%
3. 34-50%
etc.

The heuristics employed to classify users appear in Table 10. Explicitly acquired attributes as well as information accumulated by the system provide the knowledge needed to infer user types.

Callable Functions The callable goals for the user model module appear in Table 11.

4.2.7 Search and Query Formulation

Research regarding efficient methods for searching large document collections focuses on ways to reduce the number of database accesses required to respond to any given query [SALT 83c]. The goal is to eliminate sequential searching of entire collections and to examine as few document representations as possible without sacrificing retrieval effectiveness. A broad range of searching methods and algorithms [FALO 85, SALT 83c] have been developed to reduce the time required to locate relevant documents in large information collections. As a research testbed, the CODER system aims to include a wide variety of those methods so that comparisons between different algorithms in different situations may lead to strategies for selecting the best search method for a given query. Different similarity measures and/or search methods appear to be more effective for different types of queries; however, determination of which query characteristics suggest particular search strategies is an open research problem.

Retrieval Models: The most common arrangement for retrieval is a Boolean system accessed via an inverted file. Commercial systems like MEDLARS, BRS, DIALOG and STAIRS have adopted such a retrieval model. Key terms connected by the Boolean operators AND, OR and NOT are used in conventional *Boolean* retrieval. Document sets associated with key terms connected by AND are intersected while all documents associated with terms joined by OR operators are included. Documents containing negated terms are eliminated from the relevant set. Boolean re-

Table 10. User Stereotype Classification Heuristics

For user type *novice*:

Slot	*Value*	*Comments*
usedcomp	n	never used a computer
OR		
otherisr	n	never used an ISR system
OR		
otherisr	y	used other ISR systems less than
otherfreq	< = 10	or equal to 10 times

For user type *average*:

Slot	*Value*	*Comments*
otherisr	y	used other ISR systems
otherfreq	> 10	more than 10 times.
OR		
totsearches	> 2	Performed more than 2 CODER searches
OR		
totsearches	> 5	Performed more than 5 CODER
cscrs	y	searches and has taken CS courses

For user type *expert*:

Slot	*Value*	*Comments*
otherisr	y	used other ISR systems more
otherfreq	> 10	than 10 times, and performed
totsearches	> 5	more than 5 CODER searches
OR		
totsearches	> 12	Performed more than 12 CODER searches

Note:
Any users not classified by the above rules are classified as average and a warning message is logged for the CODER system administrator.

Table 11. User Model Callable Goals

CALLABLE GOAL	FUNCTOR in HYPOTHESIS	AREA	COMMENT
id_user(Tm)	disp_prompt	umodel	request user identification
char_newuser(Tm)	disp_prompt	umodel	begin to get user info
char_olduser(Tm)	utype state	umodel state	post user type post transition for probmsd
um_resp(Resp,Id,Tm)*	state disp_menu disp_prompt tut editf	state umodel umodel umodel umodel	process user responses to user model prompts/menus
um_req(Request,Tm)	displayf	user_req user model	user requested browsing of
user_eval(Tm)	disp_msg disp_prompt	umodel umodel	begin user evaluation of session
clean_up(Tm)	state	state	after frames are updated, post state transition

* Hypothesis posted depends on Id of prompt or menu.

trieval is popular in operational situations because the basics are easily understood and high standards of performance are achievable if searchers are creative and persistent [SALT 83b]. However, the set of relevant documents retrieved in response to a query is not ranked, and grappling with sheer size may be an obstacle to effective retrieval.

Between conventional Boolean retrieval and vector retrieval is *extended Boolean retrieval* which uses the *p-norm* formalism to determine document/query similarities [SALT 83b]. Boolean operators, query term weights and document term weights are all included in this model. Furthermore, Boolean connectives may be parameterized by a number from 1 to infinity, the *p value*, which asserts the strictness of interpretation of the connective. As p values move from infinity to 1, the Boolean operators are interpreted more and more loosely; when a p value equals 1, the distinction between AND and OR operators disappears.

A variety of additional retrieval methods have been developed, however they have not yet been included in the CODER system. *Probabilistic* retrieval uses probability theory in both the indexing and retrieval processes based on ratios of relevant and nonrelevant documents. A method for integrating *Boolean and probabilistic* retrieval methods has been proposed by Croft [CROF 86a]. *Clustered searching* [FALO 85] proposes that closely associated documents tend to be relevant to the same request and uses cluster centroids to contain average term weights representing all documents in a cluster. Retrieval based on *citations and references* employs a citation index and/or cocitation links to relate documents. Eastman and Weiss propose a tree algorithm for *nearest neighbor* searching [EAST 77]. As built into the MU-Prolog external database factilities, more sophisticated multi-attribute retrieval is provided by *superimposed coding* schemes [SACK 82, RAMA 85].

This version of the retrieval subsystem includes Prolog programs to support *vector, p-norm* and *Boolean* searching. However, as with the lexicon, the collection to be searched is too large to be consulted in a local Prolog fact base. Work is currently in progress to adapt indexing and searching routines from the SMART system. A *C* resource manager will be created to be accessed by the search expert so that time-consuming searching of Prolog fact bases, either local or external, is omitted. This section will describe the **current** functions of the *search* and *query formulator* modules. The incorporation of SMART routines, still in progress by other students, will not be discussed here.

4.2.7.1 Search Expert

The CODER search expert has its roots in an expert system tool, written in HC Prolog [ROAC 85], to aid the placement of foster children into appropriate homes. The HC Prolog code was converted to MU-Prolog code to fulfill class project requirements early in 1986 [WEAV 86a]. Later, search functions were expanded and calls to the blackboard/strategist and to a Prolog external database were added [WEAV 86b]. Boolean searching functions were developed during Spring quarter, 1987 [SIU 87].

The search expert uses an external Prolog database of facts containing a relational representation of key terms and document ids:

dv(Document_id,Classification_type,Query_term,Weight).

The facts were produced from vectors generated by the SMART system. Query terms are represented by concept numbers and weights are numbers in the range .00001 to .99999, where the decimal place is assumed. Classification types, such as author, title or body, are not used. The superimposed coding scheme provided with MU-Prolog was used to index the facts for faster retrieval. However, response times for very large numbers of facts (over 5000) were unacceptable, and only small fact bases of 2000 to 5000 facts have been used by the search expert.

The search expert performs three primary functions:

1. Retrieval of relevant document ids.
2. Computation of query-document similarity, if term weights are provided.

3. Ranking of relevant documents based on similarity, and posting of document ids up to the number of documents requested by the user.

P-norm processing: As mentioned earlier in this section, Boolean operators, query term weights and document term weights are all included in the p-norm model, and the *p value* allows fuzzy interpretation of Boolean operators. The search expert expects a p-norm query posted on the blackboard to adhere to a particular syntax. That is,

p_norm(Number_of_docs, [Weight,Boolean_operator,P_value,Clauses]).

where

Number_of_docs is the quantity of documents the user would like to retrieve.

Weight is an integer from 0 to 10.

P_value is an integer from 1 to infinity, where infinity is represented by '%'.

Clauses is a list of *term, weight* pairs where terms are of the same form as those stored in the document database facts, in this case, concept numbers, and weights are integers from 1 to 10 indicating the importance of query terms.

Therefore, a valid p-norm query representing "I would like to retrieve 10 documents about automation of catalogs or computer peripherals" would be

p_norm(10,[9,or,5,[8,and,1],[automation,10],[catalog,7],[9,and,1],[computer,5],[peripheral,10]]).[12]

When relevant documents have been retrieved, similarities between documents and queries are computed.[13] As described by Salton, Fox and Wu [SALT 83b], the similarity is calculated according to the following formula.

Consider, as an example, a document D with assigned terms A and B and let d_A and d_B represent the weights or importance of the two terms in the document. ... Given queries (A and *B) and (A* or *B), the following query-document similarity functions may be defined between these queries and a document $D = (d_A, d_B)$.*

$$\mathrm{sim}(Q_{(A\ and\ B)}, D) = 1 - \left[\frac{(1 - d_A)^P + (1 - d_B)^P}{2} \right]^{\frac{1}{P}}$$

[12] Note that terms such as automation and catalog would really be concept numbers.

[13] Similarity computations require MU-Prolog floating point number routines.

$$\text{sim}(Q_{(A \text{ or } B)}, D) = \left[\frac{d_A^P + d_B^P}{2} \right]^{\frac{1}{P}}$$

Following similarity computations, the retrieved documents are sorted by similarity, and the 10 highest documents will be posted to the blackboard with ***doc(Id)*** hypotheses using similarity rankings as confidence values.

Vector processing: Setting the p value to 1 in a p-norm syntax query will result in vector style processing. That is, AND operators are treated like OR operators; all terms are, therefore, *ored* together. For k equal to the number of query terms, "the actual similarity values obtained for p = 1 are exactly those produced by a vector processing system in which the similarity between a document $D = (d_1, d_2, ...)$ and a normalized vector query $Q = (\frac{q_1}{k}, \frac{q_2}{k}, ..., \frac{q_k}{k})$ is evaluated as the usual vector product." [SALT 83b].

Boolean processing: When p value is set to infinity, Boolean connectives will be strictly interpreted. Therefore, conventional Boolean queries could be processed by the p-norm functions. However, the p-norm routines implemented have no provision for handling of the Boolean NOT operator. Search functions which process Boolean logic queries including the NOT operator have been developed [SIU 87]. As with the p-norm and vector queries, the search expert expects a Boolean query to be properly structured in prefix notation format when received. For example, a Boolean search hypothesis might be

[boolean(5,[and,or,online,computer,not,retrieval,storage]).

representing the query "I would like 5 documents about online or computer and retrieval, but not storage." Up to five relevant document ids will be posted to the blackboard by the search expert. Additional details regarding the operations performed by the Boolean searching routines may be reviewed in the corresponding class project report [SIU 87].

Frame Matching: In addition to traditional retrieval where documents and structured queries are matched, the CODER system requires that partial frames created by users be matched to frames in the document knowledge base created by the analysis subsystem. A two-quarter project is currently underway to facilitate matching of user frames to document frames. Its twofold objective includes 1) constructing a "system" to store frames in C structures and 2) creating a CODER resource manager to determine *matching_frames* and to perform other frame object manipulations. Although the frame object predicates have been coded and tested using MU-Prolog, efficient processing to match frames could not be accomplished due to the size of the document knowledge base. Therefore, matching of partially filled user frames to document frames could only be performed for small subsets of the document database. The implementation of a CODER resource manager written in C should efficiently accomplish matching of frames.

Callable Functions: The callable goals for the search module appear in Table 12.

4.2.7.2 Query Formulator

The objective of the CODER query formulator module, *qform*, is to structure queries in formats recognizable by the search expert, and to assist users in formulating Boolean queries. Future versions of the query formulator may help users to formulate other types of queries as well. The query formulator was written as a stand alone module [QUIZ 87]. Later, calls to the blackboard/strategist and replacement of input/outupt functions with operations to be performed by the user interface manager were added.

The query formulator offers assistance for two types of users, novice and expert. Explanations of *facets, exclusion lists* and *Boolean query requirements* are provided. In addition, the user is prompted to enter term facets, subjects and exlusions, and is given an opportunity to modify system formulated queries before they are posted for the search expert. Further description of the query

Table 12. Search Module Callable Goals

CALLABLE GOAL	FUNCTOR in HYPOTHESIS	AREA	COMMENT
p_norm(Numdoc,Query,Tm)	doc	results	search database
	docs_posted	results	finished finding docs
boolean(Numdoc,Query,Tm)	doc	results	search database
	docs_posted	results	finished finding docs
vector(Numdoc,Query,Tm)	doc	results	search database
	docs_posted	results	finished finding docs

Table 13. Query Formulator Callable Goals

CALLABLE GOAL	FUNCTOR in HYPOTHESIS	AREA	COMMENT
form_query(Tm)	disp_prompt	qform	begin prompting user
do_query(Query_type,Tm)	disp_prompt	qform	begin prompting user
qf_resp(Resp,Id)	disp_prompt disp_menu displayf editf	qform qform qform qform	continue to get info to build query

formulator module is provided in the corresponding class project report [QUIZ 87]. The sample session in Appendix H illustrates some of the functions of the query formulator module.

Callable Functions: The callable goals for the query formulator module appear in Table 13.:

Conclusion: Although searching and retrieval of documents could occur, albeit less effectively, without the user model, input analyst, report generator, problem description builder or browsing modules, **no** retrieval can occur without the search expert. Elimination of the query formulator would require that the user know how to formulate searchable queries using the syntax expected by the search expert. Unsurprisingly, the search engine was the first module of the CODER prototype to be implemented and tested. However, the search module's need for document and term information dictates intensive input/output processing; furthermore, computationally expensive operations to calculate similarity measures must be performed. Of all of the Version 1.0 modules, the current search expert demands the most immediate attention so that all user queries can be *efficiently* matched to documents in the complete AIList collection.

4.3 Version 1.0 Prototype

The diagram in Figure 33 depicts the current CODER retrieval subsystem. All modules are functional and have been integrated with the blackboard/strategist complex, user interface manager and other resource managers. The subsystem consists of nine inferential modules, three resource managers written in C and four external knowledge bases. The implemented system runs on a single VAX-11/785 running ULTRIX™, a variant of 4.2 BSD UNIX. Although work is in progress to create a multi-user version of the retrieval subsystem, Version 1.0 handles only one user at a time.

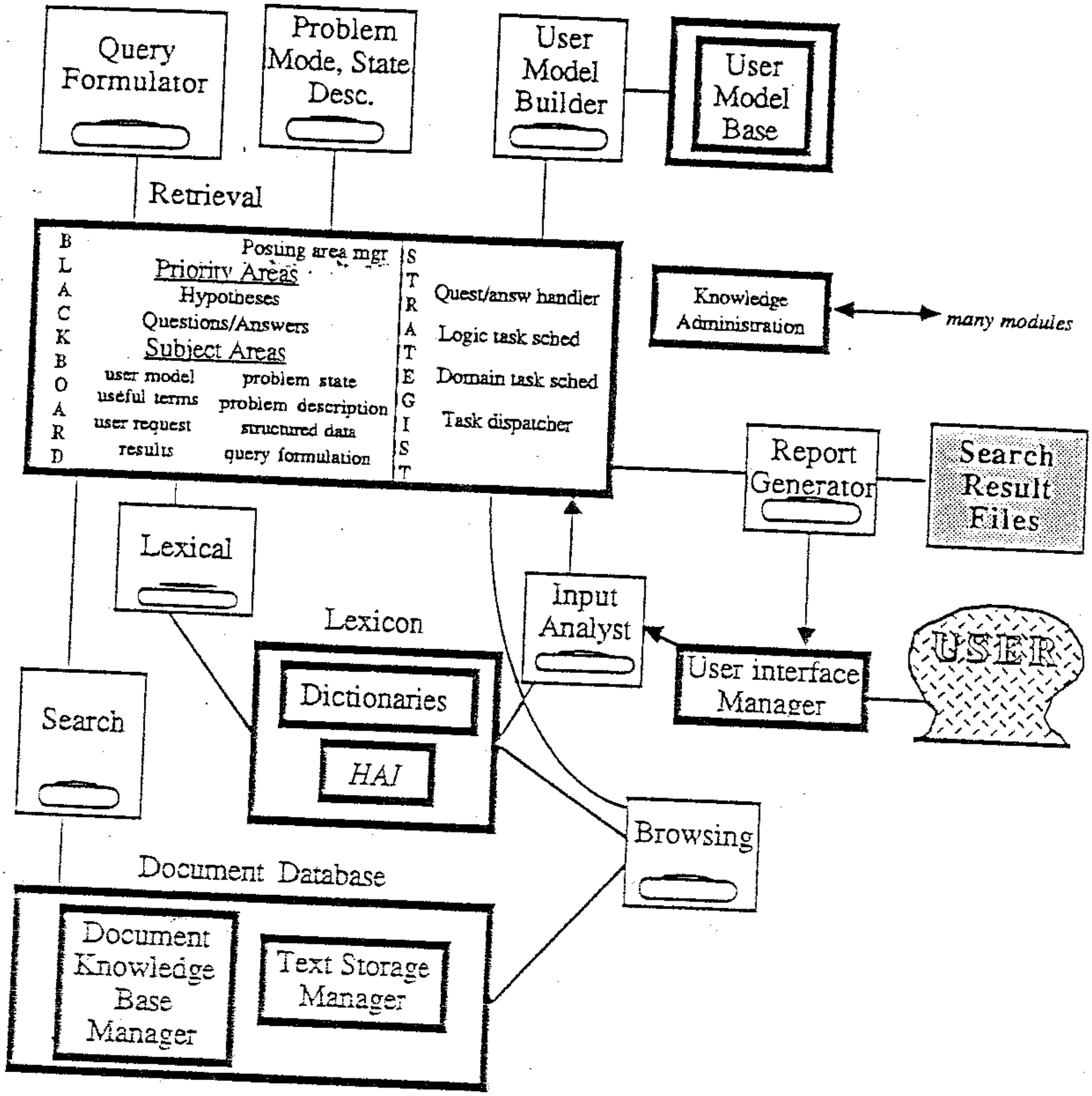


Figure 33. Version 1.0 Implementation

5.0 Discussion of Results

Based on the results of the Version 1.0 CODER system implementation, this chapter discusses how the original hypotheses of the CODER system research have been addressed by this investigation. In addition, a micro-level evaluation of system performance and details concerning the CODER modules are also presented.

5.1 Accomplishments

The hypotheses addressed by this implementation were presented in Chapter 1, section 1.3. Each hypothesis will be examined here again in relation to the conclusions garnered from the retrieval subsystem implementation.

Logic programming is adaptable to information storage and retrieval.

The Version 1.0 implementation of the CODER retrieval subsystem includes nine modules written in the logic programming language Prolog. The list processing, recursion and pattern matching abilities of the language allowed fast, efficient implementation of modules. Structures containing *lists* of query or document terms, frame slots, lexical definitions, document ids, blackboard hypotheses, and queues of expert tasks could be processed recursively; thus, Prolog's inherent efficiencies for recursive processing could be utilized. Prolog pattern matching abilities facilitated, for example, straightforward retrieval of hypotheses from specific blackboard areas, matching of user responses to particular prompts or menus, and tailoring of dialog to user types.

Programmers who were trained in conventional procedural languages did encounter a significant learning curve when attempting to develop programs in Prolog. However, once the fundamentals of a logic programming language were understood, module development could proceed rapidly. Inferential reasoning performed by the Prolog modules could not be as easily coded in a language such as Pascal or C. Rule bases, for example rules for scheduling strategies or for the problem state finite state automaton, would have required many if-then-else structures and/or special file or data structures if developed in a procedural language. Moreover, addition of new rules may be effected by simply adding new facts to Prolog fact bases. Although the time required for computationally intensive processing within Prolog modules was not optimal, compilation of modules when conversion from MU-Prolog to NU-Prolog occurs should significantly reduce computation times. The creation of the Version 1.0 retrieval subsystem, with most modules coded in Prolog, indicates that logic programming is adaptable to information storage and retrieval.

The knowledge engineering paradigm can be applied to information storage and retrieval systems.

The Master's thesis specifying the design of the CODER system [FRAN 86] proposed that knowledge engineering tools could be used by information storage and retrieval systems. A knowledge administration complex has been written and tested, and is an integral part of both the analysis and retrieval subsystems. Structures representing the content and hierarchical organization of documents have been defined and instantiated. *Knowledge* about documents and users is stored in frames and is used by retrieval subsystem modules to partially simulate the functions of a trained search intermediary and to intelligently match user queries to documents. The representation of knowledge in the domain of artificial intelligence has been accomplished in the CODER system through use of the knowledge engineering tools, frames and relations.

System modularity provides a more flexible research testbed environment.

Twelve modules, each performing distinct functions, form the present CODER retrieval subsystem. Each module was first developed independently as a stand-alone program. System development and module integration could proceed without completion of all modules. Moreover, as enhanced versions of modules were created, newer modules could be substituted for previous versions simply by replacing source or object files in the configuration directory and adding new scheduling rules to the strategist if necessary. Centralization of message communications on the blackboard promotes further independence of modules and provides additional flexibility. Adding new modules or removing modules, as discussed in Chapter 3, section 3.3.2, requires only two steps: creation or deletion of a socket, and modifications to the strategist if necessary. Although successful integration of the first module required considerable struggling and effort, subsequent modules were integrated in a few hours and generally performed correctly without substantial difficulties.

Nearly thirty graduate students at Virginia Tech have made contributions, of varying degrees, to the development of the CODER system. The number of different people working on pieces of the CODER system,as well as the ease with which new modules were integrated, supports the hypothesis that modularity does provide a quite flexible research testbed environment.

Users can perform more effective retrieval when structured knowledge is employed.

"Structured knowledge" includes the hierarchical organization of documents as well as concepts such as names, dates and addresses. The knowledge administration complex provides the foundation for the creation and storage of structured knowledge. Frames and relations representing documents and their contents are created by the analysis subsystem. The input analyst and report modules promote and support user entry of structured knowledge information. Finally, the search expert matches the user's structured knowledge entries to document frames and relations.

The facilities needed to support user retrieval employing structured knowledge have been provided in the Version 1.0 implementation. However, the processing required to *efficiently* match user frames to the ***complete*** set of document frames is still under development. As a consequence, experiments to evaluate user satisfaction with retrieval results when frame information is supplied have not yet been performed. When efficient frame matching can be accomplished, such experimentation should examine retrieval satisfaction of traditional vector-style queries versus structured knowledge queries; moreover, combinations of different kinds of traditional queries such as p-norm, vector and boolean, in conjunction with varying amounts of frame input should be compared. Such experimentation is not deemed trivial and could also include investigation of which query characteristics suggest specific search strategies.

5.2 *Micro-Level Evaluation*

System performance and details concerning CODER modules have been evaluated at the implementation level. This section discusses the performance of the retrieval subsystem, that is, the times required by the system to perform specific functions. In addition, the size of the modules and their rule bases are presented.

5.2.1 Performance

The timings for the session evaluated were taken during late evening hours when system load was minimal. They appear in Appendix H. The time(X) predicate added to the MU-Prolog interpreter was used to record the time, in seconds, at which hypotheses were posted to the CODER blackboard. Although times recorded to tenths of a second would have allowed more precise evaluation of system performance, the times listed provide a sufficiently accurate indication of the system's performance.

The tables contained in Appendix H reflect the sequence of control from module to module and the flow of a **typical** retrieval session. A typical session is illustrated by the screens printed in Appendix I. Areas where concurrent processing may occur, other than between the blackboard and strategist modules where it always occurs (see section 4.2.1), are indicated by '+'. Where many prompts, menus and display files are sent to the user interface so that modules may collect information for a single purpose, for example user evaluation of the session, functions are not repeatedly listed.

When modules require information from users, the system requires approximately one second to process the four steps listed below. Data is passed as follows:

1. a module posts a display request to the blackboard;
2. the blackboard notifies the strategist;
3. the strategist dispatches the report module;
4. the report module sends the request to the user interface manager for display.

Next, the user's response must be sent back to the module requesting the user input. The module receives the information and posts a request, for example for more information, in about one second.

1. The user interface manager sends the user response to the input analyst module;
2. the input analyst converts the response to an internal system form and posts it to the blackboard;
3. the blackboard notifies the strategist;
4. the strategist dispatches the requesting module;
5. the module processes the information and posts the next hypothesis or question on the blackboard.

Passing data to and from the user is performed well within acceptable response time ranges. Most general functions require approximately one second to be executed; however, *state transitions* normally require from two to three seconds due to the extra calling of modules. For example,

1. a module posts a state transition to the blackboard;
2. the blackboard notifies the strategist;
3. the strategist dispatches the probmsd module;
4. the probmsd initiates the next state by posting a hypothesis to the blackboard;
5. the blackboard notifies the strategist;
6. the strategist dispatches the appropriate module;
7. the module may request information from the user which it posts to the blackboard;

8. the blackboard notifies the strategist;
9. the strategist dispatches the report module;
10. the report module sends the request to the user interface manager for display.

Other critical processing steps which require significant times to process include the following.

- system start up requires slightly over one minute;

- system termination, including refreshing frame object files and removing temporary files takes up to one minute; this is invisible to the user since most of it occurs after the user's final message from the system has been received;

- the time required for building and posting of frames created by users depends on the number of frames created, number of slots in the frames and whether subframes were created; it often requires over ten seconds; however, this function is performed concurrently with other processing steps, so the time requried to post frames is transparent to the end user;

- the time required for searching the document data base using a test collection of 2500 Prolog facts depends on the query submitted, but often requires from five to ten seconds;

- the time required for searching the HAI or the lexicon also depends on the terms used to perform searching and usually requires up to five seconds;

- formulation of a Boolean query in the syntax expected by the search module, given user term and exception entries, requires three to four seconds.

5.2.2 Module Statistics

Sizes of Prolog modules in terms of lines of code, numbers of rules in rule bases, and the number of ancillary files such as menus and tutorials are presented in this section.

Source Code Statistics: Table 14 illustrates the sizes of the CODER inferential modules. The *Prolog code* column represents the number of Prolog goals or subgoals in each module. Comments, blank lines and print control lines are ignored. The third column includes comment lines. Interestingly, well over half of the total lines in each module is devoted to comments. The Prolog language does not require declarations of variables or definition of record and file structures. Therefore, the amount of code required to accomplish desired functions is much less than would be found in more traditional procedural programs, but comments are even more important.

Rule Base Statistics: The sizes of the CODER rule bases, of which some rules are single facts, are presented in Table 15. These numbers were not included in the sizes given for the Prolog modules.

Miscellaneous Statistics: The quantities of some of CODER's supporting files are shown below. The number of menus, prompts and display files changes frequently; therefore, the numbers shown represent a snapshot of the quantity of files only, as of December, 1987.

40	Menus
40	Tutorial Files
200	Prompts
20	Display Files

Since many of the C **modules**, such as the user_interface manager, the text manager and the lexicon manager, are still under development by other graduate students, exact sizes of those modules are not known. However, the current hai_mgr is completed and contains 237 lines of C code plus 387 lines of *included* code for Prolog to C communications; therefore, the hai_mgr contains a total of 624 lines of C code. At present, the user interface manager contains about 1200 lines of C code

Table 14. Module Source Code Statistics

Module	Prolog Code	Prolog Code With Comments
knowadm	1009	2378
user_model	634	1057
browse	499	831
report	486	873
qform	459	836
ia (input analyst)	380	864
search	349	1082
probmsd	266	527
ka_localobj	254	677
lexical	250	406
bboard	248	610
LTS(logic task scheduler)	133	287
TD (task dispatcher)	84	182
QA (question/answer)	59	165
DTS (domain task scheduler)**	32	78
Totals	5142	10853
** *(strategist subtotals:*	*308*	*712)*

Table 15. Rule Base Statistics

Rule	Description	Module	Number of Rules
fsm	finite state machine transitions	probmsd	25
sched	module scheduling	strategist (DTS)	50
classify_user	determine user type	user_model	9
priority	determine scheduling priority	strategist (LTS)	16
pos	converts lexical part of speech for user browsing	report	32
ka_frame*	frame type definitions		33
ka_edt	EDT definitions		13
ka_rel	relation type definitions		17
ka_slotdesc	describes slots and assignment of prompt numbers and tutorials	ia report	128
ka_fparent	inverted frames to show parent-child relationships		5

*Includes 7 user frames.

plus the 387 lines of code included for Prolog to C communications. Approximate statistics on **external knowledge bases** appear in Table 16.

Table 16. Approximate Statistics on External Knowledge Bases

A) AIList Document Collection	
Number of messages:	8000
Number of authors:	300
Number of digest issues:	950
Dates covered:	4/83 - 12/87
Characters of text:	16Mbytes
B) Handbook of Artificial Intelligence	
Number of files:	106
Characters of text:	4Mbytes
Number of Table of Contents subjects:	218
Number of Index entries:	853
Number of Index range entries:	158
Number of Index person names:	138
Number of Italicized words/phrases:	5009
C) Collins Dictionary of the English Language	
Number of relations produced:	21
Number of headwords:	85K
Number of different parts of speech:	46
Number of categories (used > = 30 times):	120
Number of definitions:	165K
Number of morphological variants:	28K
Number of usage samples:	17K
Number of comparisons:	8K
Numbers for parts of speech:	
Nouns:	63K
Verbs:	15K
Adjectives:	13K
Adverbs:	1300

strated "impressively", a more powerful user interface with functions such as those provided by a Macintosh-style interface is required.

3. Efficient access to sizable **external knowledge bases** has not yet been accomplished. The full set of italics reference facts derived from the HAI are not included in the HAI browsing; only a sample portion of the lexicon is searched by the lexical expert; and matching of queries is limited to a small subset of documents. So that the browse, lexical and search experts may function more efficiently, external knowledge bases such as the lexicon, HAI and document databases must be indexed and stored so that rapid access to desired data can be achieved. That is, either the *C* resource managers or the database facilities provided by NU-Prolog must be used to access large databases.

4. When the CODER retrieval subsystem is executing, 12 additional processes are placed on a computer system which is already operating near capacity and which numerous Computer Science graduate students and faculty rely on for other purposes. Consequently, during periods of peak usage, such as Mondays thru Fridays from 10 a.m. to 4 p.m., **system response times** are not within acceptable ranges. For example, after entering a user id, the user may be forced to view an unchanging display terminal for over a minute while waiting for the next prompt to appear. Response times during non-peak hours, however, are acceptable. Such dependency on system load suggests that the CODER system be moved to a dedicated computer or to one with greater computational and storage capacity.

5. As mentioned several times throughout this paper, **natural language queries** cannot be processed by this implementation of the CODER system. Inclusion of natural language queries was not an integral part of the original CODER proposal; however, partial parsing of natural language queries such as that found in the FRUMP system [DEJO 82] is desired. Nonetheless, the degree of effort required to implement even limited processing of natural language queries suggests that separate graduate research be dedicated to the development of a module to process natural language queries.

6.0 Conclusions

By the end of 1987, implementation of the first version of a comprehensive intelligent information retrieval system, conceptualized four years earlier, reached fruition. Although enhanced development of many subsystem modules remains to be accomplished, a robust research testbed for artificial intelligence techniques in information retrieval has been created. The system as it currently exists represents the efforts of many Virginia Tech graduate students as well as the thoughts and ideas of various researchers in the field of information retrieval.

As with other academic systems such as the SMART system began at Cornell University in the early 1960's and still used for information retrieval research [BUCK 85], the CODER system will likely be modified and expanded for many years to come. Its modularity and diversity of functions provide the foundation for continued research and development of AI methods in information retrieval systems. User modeling techniques and knowledge engineering tools may be investigated; natural language processing and the benefits of an online lexicon may be researched; and the retrieval effectiveness of different search strategies may be compared. Continued experimentation as well as further development of the basic methods supported by the system will likely support the hypothesis that AI techniques significantly upgrade the productivity of ISR systems.

This final chapter summarizes major system limitations and recommends future enhancements. The limitations and recommendations listed, alluded to in earlier chapters, are ordered by degree of importance, with the most critical listed first. The recommendations suggested by no means include all facets of the system requiring enhancements; future research projects may use the CODER system as the basis for investigation of a diversity of AI and ISR methods.

6.1 Limitations

Only the most significant limitations of the Version 1.0 CODER system are detailed in this section; minor limitations of various modules are not included.

1. As discussed in chapter 4, section 4.2.7, the **search** module is the one module that is not optional in a retrieval system. Although the existing search functions perform correctly and a small subset of the AIList archive may be searched, the complete document collection has not been searched by the CODER system. The Prolog facts containing vectors of terms and documents, as well as the frame objects representing document contents, are too numerous to be searched efficiently using the current version of Prolog. Therefore, the *C* routines adapted from the SMART system and the new routines to be used for frame matching must be provided before the full AIList collection may be searched effectively. Additionally, query terms must be converted to the assigned concept numbers before databases can be properly searched. Prolog functions to perform such conversion do not exist. Desirable word searching operations such as word nearness, word position, and word truncation are not yet provided.

2. The UNIX *curses* **user interface** lacks flexibility and does not provide many of the functions desired for the CODER user interface. Presently, only one window may be displayed at any given time, and editing and prompting operations perform less than optimally. Moreover, the system guides the user through the retrieval session rather than both the user and the system initiating stages of the retrieval process. Section 6.2 recommends modifications to the user interface to alleviate this problem.

   As discussed in relation to the I^{3}R system, the user interface manager may contain 40 to 50 percent of all code in the system [THOM 87a]. Admittedly, development of a generic yet robust user interface is not a trivial task. However, before the CODER system may be demon-

6.2 *Recommendations*

1. The limitations of the **search** module discussed in the previous section must be addressed. That is, the *C* routines adapted from the SMART system and the new C resource manager to be used for frame matching must be provided before the full AIList collection may be searched efficiently. The use of feedback to perform additional retrievals and the incorporation of word searching operations like word nearness, word position and word truncation should also be provided.

2. Development of a Macintosh-style **user interface** will substantially improve the demonstrability of the CODER system. Providing the user with pull-down menus, pop-up windows and mouse selection of terms and options will allow CODER to be a *mixed initiative* system. That is, both the user and the system may initiate activities. For example, a menu of NISO commands could be provided for the user to request desired processing at any point during the retrieval session; at any point in the retrieval process, a browse menu could be selected for browsing world or domain knowledge; pop-up windows containing selected terms, phrases, or previous queries could be displayed during query formulation; and defaults for user environment preferences such as document sorting could be used unless the user selected a pull down menu to modify the environment default values. For experienced users particularly, such features should considerably improve the level of satisfaction with the system.

3. The updated version of MU-Prolog, **NU-Prolog**, was received at Virginia Tech at the end of November, 1987. Compilation of Prolog modules should significantly upgrade system performance; furthermore, experimentations with the improved external database facilities provided by NU-Prolog may alleviate some of the system limitations resulting from inefficient Prolog processing of large external knowledge bases. To implement NU-Prolog, the modifications applied to the MU-Prolog *prolog-s* and *prolog-b* versions must be incorporated into

new versions of NU-Prolog. In addition, the Prolog modules may require some modifications for successful compilation to occur.

4. To alleviate CODER system response time fluctuations due to computer system load, the retrieval subsystem should be ported to a **dedicated computer** or to a computer with increased computational power and storage capacity. The Macinstosh™ II supports the UNIX operating system, A/UX™, and provisions for TCP/IP communications exist. Availability of a Macinstosh II computer for CODER research suggests that it be considered as a home for future versions of the CODER system. The pending sale of the VAX computer on which CODER resides provides additional incentive to relocate the system in the immediate future.

5. The **query formulator** module presently provides assistance only for Boolean query formulation. Similar assistance for users who wish to formulate p-norm queries should be developed. As additional methods of retrieval are developed, assistance should be provided by the query formulator module.

6. The **user modeling** module in the CODER system provides the foundation for continued research of user modeling techniques in ISR systems. Development of rules used to characterize users and to tailor system performance may be accomplished with expansion of user modeling rules and user model frames. Rules gleaned from behavioral studies, dialogs between users and search intermediaries, system performance studies and empirical observation may be grouped and systematized into a knowledge base of heuristics for use by intelligent information retrieval systems.

7. The addition of a **dialog module** for providing enhanced assistance to users and for determining the type of interaction appropriate depending on circumstances of the user's situation should be considered. Functions to inform the user of the system's progress, intentions and perceptions should be included. Modes of interaction for different types of users and for different

stages of the retrieval session could be determined by a dialog module. Belkin et al. provide examples of functions to be performed by a dialog module [BELK 83].

> *For instance, at some states a natural-language interaction in an informal mode might be appropriate (in, say, initial problem description by the user with as yet nebulous perceptions of the problem). In others, interactive graphic interaction might be preferred (in, say, representation of the mechanism's perception of the user's anomalous state of knowledge for a user who prefers multi-dimensional to linear representations).*

Additionally, the tutorials provided for the system should be expanded.

8. The **NISO commands** for interactive online information retrieval systems [NISO 87] accepted by the input analyst may be expanded and incorporated into other modules. For example the *find* command specified in the NISO standard allows Boolean queries to be submitted to the system in an infix notation. Such syntax must be reformulated by the query formulator so that the query may be processed by the search expert. *Review* of search history and *print* commands must be incorporated with text browsing. The *scan* command to view an ordered list of search terms may be fully implemented when the dictionary of search terms is provided. In summary, many of the functions suggested by the NISO standard do not yet exist in the CODER retrieval subsystem. Periodic review of the commands so that they may be implemented in appropriate system modules as functions become available is advised.

9. Advanced functions for **browsing document text** have yet to be incorporated into the browse module. Highlighting of document terms matched to query terms; cross-referencing of documents, such as replies to previous messages cited within retrieved documents; display of documents at the point where relevant text begins; and browsing document archives are a few of the functions which a text browsing expert could include. More sophisticated browsing operations are described in recent IR literature [INGW 86].

10. As discussed in chapter 4, section 4.2.1, the **blackboard/strategist complex** does not contain all of the functions envisaged in the original design of the CODER system. Scheduling strategies must consider the context of the phases of the overall task in which the system is engaged. The task dispatcher should inform the strategist when task queues are empty or when no tasks

in a queue have priority greater than a certain threshold. Implementation of more complex control and scheduling strategies should be investigated.

11. The functions of the strategist, particularly of the logic task scheduler, are highly dependent on the **confidence values** posted with blackboard hypotheses. Confidence levels assigned to hypotheses by most Version 1.0 experts are hard-coded and have been arbitrarily determined. Except for the search expert which posts query-document similarity values as confidence levels, the modules do not use knowledge aggregation schemes or apply logical reasoning such as *modus ponens* to determine confidence values. Uncertainty based reasoning and proper assignment of confidence values may be investigated.

12. As discussed under system limitations, **natural language processing** is not included in this version of the CODER system. Partial parsing of natural language queries such as that found in the FRUMP system [DEJO 82] should be considered. Previously unsuccessful efforts to adapt the CHAT-80 Prolog parser as a quick implementation of a CODER query parser module suggest that separate graduate research be dedicated to the development of a module to process natural language queries.

13. The **subject browsing** options for the **Handbook of Artificial Intelligence** require that users enter subjects *exactly* as they appear in the HAI Prolog facts. Partial matching of users' terms and/or phrases, as provided for browsing by name, should be investigated. A more sohpisticated version of the browse expert or of the program which originally extracted the Prolog facts from the text would identify line number ranges of paragraphs or other blocks of text which are relevant to the subject referenced (see section 4.2.4.1). Other enhancements for HAI browsing should include highlighting of terms or phrases which match the user's entry, identification of volume and page numbers in the printed HAI where the relevant text may be found, and completion of the removal of typesetting codes in the HAI online text (currently in progress). The use of hypertext for cross-referencing should also be considered.

14. When **new frames** for organizations, bibliographic references and other document types are generated by the analysis subsystem, the retrieval subsystem's structured knowledge facilities should be reviewed. That is, the menu containing the structured objects for which users may supply data must be enhanced; slot descriptions and prompts for the new frames and their slots must be added to the *ka_slotdesc* file and to the *prompts.dat* file. Any modifications to frame structures require that corresponding prompts, tutorials and menus be reviewed.

6.3 Assessment

In summary, the objective of this research has been achieved: a prototypical intelligent distributed information retrieval system has been implemented. Artificial intelligence techniques such as *knowledge representation, logic programming, distributed rule-based experts,* and *inferential reasoning based on heuristics* have been included in the implementation. The efforts of many researchers and graduate students have been incorporated into the CODER modules. As a modular research testbed, the CODER system simplifies integration of new modules and experimentation with different information retrieval methods. Future work will lead to removal of the main limitations and to recommended improvements. It is hoped that the resulting system will be easier to use, more satisfying and more effective for users with information needs.

Bibliography

[APTE 87a] Apte, Sachit C. "Communication in the CODER system." Masters Project Report. Blacksburg, VA: Virginia Tech Department of Computer Science, June 1987.

[APTE 87b] Apte, Sachit C. "The Query Parser." Class Project Report. Blacksburg, VA: Virginia Tech Department of Computer Science, June 1987.

[BARN 87] Barnhart, Richard. "The CODER Lexical Expert." Class Project Report. Blacksburg, VA: Virginia Tech Department of Computer Science, June 1987.

[BARR 82] Barr, Avron, Edward A. Feigenbaum and Paul R. Cohen. *The Handbook of Artificial Intelligence.* Los Altos, CA: William Kaufmann, 1981.

[BATE 79a] Bates, Marcia J. "Information Search Tactics." *Journal of the American Society for Information Science* 30:2 (July 1979), pp. 205-214.

[BATE 79b] Bates, Marcia J. "Idea Tactics." *Journal of the American Society for Information Science* 30:4 (September 1979), pp. 280-289.

[BELK 83] Belkin, N. J., T. Seeger and G. Wersig. "Distributed expert problem treatment as a model for information system analysis and design." *Journal of Information Science* 5 (1983), pp. 153-167.

[BELK 84] Belkin, N. J., R. D. Hennings and T. Seeger. "Simulation of a Distributed Expert-based Information Provision Mechanism." *Information Technology Research & Development Applications* 3:3 (1984), pp. 122-141.

[BELK 87a] Belkin, N. J. and W. Bruce Croft. "Retrieval Techniques." *Annual Review of Information Science and Technology* 22 (1987), pp. 109-145.

[BELK 88] Belkin, N. J. and H. M. Brooks. "Knowledge Elicitation Using Discourse Analysis." *International Journal of Man-Machine Studies* 28 (1988), in press.

[BISH 87] Bishop, Lucy. "HAI Browsing." Class Project Report. Blacksburg, VA: Virginia Tech Department of Computer Science, June 1987.

[BISW 87] Biswas, Gautam, James C. Bezdek, Marisol Marques and Viswanath Subramanian. "Knowledge-Assisted Document Retrieval: I. The Natural-Language Interface." *Journal of the American Society for Information Science* 38:2 (1987), pp. 83-96.

[BOBR 77] Bobrow, D. G. and T. Winograd. "An Overview of KRL, a Knowledge Representation Language." *Cognitive Science* 1 (January 1977), pp. 3-46.

[BORG 85] Borgman, Christine L. "Designing an Information Retrieval Interface Based on User Characteristics." *Proceedings of the 8th Annual International ACM SIGIR Conference on Research & Development in Information Retrieval* (Montreal: June 1985), ACM, 1985, pp. 139-146.

[BORG 86] Borgman, Christine L. "Individual Differences in the Use of Information Retrieval Systems: A Pilot Study." *ASIS-86: Proceedings of the 49th Annual Meeting,* Knowledge Industry Publications, 1986, pp. 30-31.

[BORG 87a] Borgman, Christine L. "Individual Differences in the Use of Information Retrieval Systems: Some Issues And Some Data." *Proceedings of the 10th Annual International ACM SIGIR Conference on Research & Development in Information Retrieval* (New Orleans, LA: June 3-5, 1987), ACM, 1987, pp. 61-71.

[BORG 87b] Borgman, Christine L. "Information Systems Functionality: A User-Driven Perspective." Paper presented at the *Workshop on Distributed Expert-Based Information Systems.* School of Communication, Information and Library Studies, Rutgers University, (March 1987).

[BRAC 83] Brachman, Ronald J., Richard Fikes and Hector J. Levesque. "Krypton: A Functional Approach to Knowledge Representation." *Computer* **16**:10 (October 1983), pp. 67-73.

[BRAC 85] Brachman, Ronald J. and Hector J. Levesque. *Readings in Knowledge Representation.* Los Altos, CA: Morgan Kaufmann, 1985.

[BRAJ 87] Brajnik, Girogio, Giovanni Guida and Carlo Tasso. "User Modeling in Intelligent Information Retrieval." *Information Processing & Management* **23**:4 (1987), pp. 305-320.

[BRAT 86] Bratko, Ivan. *Prolog Programming for Artificial Intelligence.* Reading, MA: Addison-Wesley, 1986.

[BREN 81] Brenner, Lisa P., Mary Huston-Miyamoto, David A. Self, Phyllis C. Self and Linda C. Smith. "User-Computer Interface Designs for Information Systems: A Review." *Library Research* **21** (1980-81), pp. 63-73.

[BROO 85] Brooks, H. M., P. Daniels and N. Belkin. "Problem Description and User Models: Developing an Intelligent Interface for Document Retrieval Systems." *Advances in Intelligent Retrieval, Proceedings of Informatics 8* (London, 1985), ASLIB, pp. 191-214.

[BROO 87] Brooks, Helen M. "The Functions of an Information System: The MONSTRAT Model." Paper presented at the *Workshop on Distributed Expert-Based Information Systems.* School of Communication, Information and Library Studies, Rutgers University, (March 1987).

[BUCK 85] Buckley, Chris and Alan F. Lewitt. "Optimization of Inverted Vector Searches." *Proceedings of the 8th Annual International ACM SIGIR Conference on Research & Development in Information Retrieval* (Montreal: June 1985), ACM, 1985, pp. 97-110.

[BUSH 45] Bush, Vannevar. "As We May Think." *Atlantic Monthly* **176** (July 1945), pp. 101-108.

[CHAN 86] Chandrasekaran, B. "Generic Tasks in Knowledge-Based Reasoning: High-Level Building Blocks for Expert System Design." *IEEE Expert* 1:3 (Fall 1986), pp. 23-30.

[CHAR 86] Charniak, Eugene and Drew McDermott. *Introduction to Artificial Intelligence.* Reading, MA: Addison-Wesley, 1985.

[CHEN 87] Chen, Hsinchun and Vaant Dhar. "Reducing Indeterminism in Consultation: A Cognitive Model of User/Librarian Interactions." *Proceedings of the Sixth National Conference on Artificial Intelligence* (Seattle, WA: July 13-17, 1987), AAAI, 1987, pp. 285-289.

[CHIA 87] Chiaramella, Y. and B. Defude. "A Prototype of an Intelligent System For Information Retrieval: IOTA." *Information Processing & Management* **23**:4 (1987), pp. 285-303.

[CLOC 84] Clocksin, W. F. and C. S. Mellish. *Programming in Prolog*. New York, NY: Springer-Verlag, 1984.

[COFF 87] Coffield, David and Doug Shepherd. "Tutorial Guide to Unix Sockets for Network Communications." *Computer Communication* **10**:1 (February 1976), pp. 21-27.

[COHE 85] Cohen, Jacques. "Describing Prolog by its Interpretation and Compilation." *Communications of the ACM* **28**:12 (December 1985), pp. 1311-1324.

[CORK 87] Corkill, Daniel D., Kevin Q. Gallagher and Phillip M. Johnson. "Achieving Flexibility, Efficiency and Generality in Blackboard Architectures." *Proceedings of the Sixth National Conference on Artificial Intelligence* (Seattle, WA: July 13-17, 1987), AAAI, 1987, pp. 18-23.

[CROF 85] Croft, W. Bruce and Thomas J. Parenty. "A Comparison of a Network Structure and a Database System Used For Document Retrieval." *Information Systems* **10**:4 (1985), pp. 377-390.

[CROF 86a] Croft, W. Bruce. "Boolean Queries and Term Dependencies in Probabilistic Retrieval Models." *Journal of the American Society for Information Science* **37**:2 (1986), pp. 71-77.

[CROF 87] Croft, W. B. and R. H. Thompson. "I^{3}R: A New Approach to the Design of Document Retrieval Systems." *Journal of the American Society for Information Science* **38**:6 (1987), pp. 389-404.

[CUGI 86] Cugini, John. "Prolog documentation standard." *Prolog Digest* **4**:80 (1986).

[DANI 85] Daniels, P. J., H. M. Brooks and N. J. Belkin. "Using Problem Structures for Driving Human-Computer Dialogues." *RIAO '85* IMAG, (Grenoble, 1985), pp. 645-660.

[DANI 86a] Daniels, P. J. "Progress in Documentation. Cognitive Models in Information Retrieval - An Evaluative Review." *Journal of Documentation* **42**:4 (December 1986), pp. 272-304.

[DANI 86b] Daniels, Penny J. "The User Modelling Function of an Intelligent Interface for Document Retrieval Systems." *Proceedings of IRFIS 6. Intelligent Information Systems for the Information Society* (Frascati, September 1985), Amsterdam, North Holland, 1985.

[DATT 87] Datta, Sanjeev. "CS5332, Information Storage and Retrieval, Class Project Report." Class Project Report. Blacksburg, VA: Virginia Tech Department of Computer Science, June 1987.

[DEER 85] Deering, Michael F. "Hardware and Software Architectures for Efficient AI." *Byte*, April, 1985, pp. 193-198.

[DEFU 85] Defude, B. "Different Levels of Expertise For an Expert System in Information Retrieval." *Proceedings of the 8th Annual International ACM SIGIR Conference on Research & Development in Information Retrieval* (Montreal: June 1985), ACM, 1985, pp. 147-153.

[DEJO 82] Dejong, G. "An Overview of the FRUMP System." In Lehnert, Wendy G. and Martin H. Ringle (Eds.). *Strategies for Natural Language Processing*. Hillsdale, NJ: Lawrence Erlbaum Assoc., 1982, pp. 149-176.

[DOSZ 86] Doszkocs, Tamas E. "Natural Language Processing in Information Retrieval." *Journal of the American Society for Information Science* **37**:4 (July 1986), pp. 191-196.

[EAST 77] Eastman, Caroline M. and Stephen F. Weiss. "A Tree Algorithm for Nearest Neighbor Searching in Document Retrieval Systems." Technical Report TR 83-CSE-8. Dallas, TX: Department of Computer Science and Engineering, Southern Methodist Unversity, February, 1983.

[EGAN 87] Egan, Dennis E. "Individual Differences In Human-Computer Interaction." In Helander, M. (Ed.). *Handbook of Human-Computer Interaction.* North-Holland, Elseview Science Publishers, 1988.

[ERMA 80] Erman, D. L., F. Hayes-Roth, V. R. Lesser and D. Raj. Reddy. "The HEARSAY-II speech understanding system: integrating knowledge to resolve uncertainty." *ACM Computing Surveys* **12**:2 (June 1980), pp. 213-253.

[FALO 85] Faloutsos, Christos. "Access Methods for Text." *ACM Computing Surveys* **17**:1 (March 1985), pp. 49-74.

[FENI 81] Fenichel, Carol Hansen. "Online Searching: Measures that Discriminate among Users with Different Types of Experiences." *Journal of the American Society for Information Science* **32** (1981), pp. 23-32.

[FICK 85] Fickas, Stephen, David Novick and Rob Reesor. "Building Control Strategies in a Rule-Based System." Technical Report CIS-TR 85-04. Eugene, OR: Department of Computer Science and Information Science, University of Oregon, 1985.

[FIKE 85] Fikes, Richard and Tom Kehler. "The Role of Frame-Based Representation in Reasoning." *Communications of the ACM* **28**:9 (September 1985), pp. 904-920.

[FINI 84] Finin, Tim and David Silverman. "Interactive Classification." *IEEE Workshop on Principles of Knowledge-Based Systems* (August 1984), pp. 107-114.

[FORE 85] Forester, Tom, Ed. *The Information Technology Revolution.* Cambridge, MA: The MIT Press, 1985.

[FOXE 83a] Fox, E. A. "Extending the Boolean and Vector Space Models of Information Retrieval with P-Norm Queries and Multiple Concept Types." Phd. Dissertation, Cornell University, University Microfilms Int., Ann Arbor MI, August 1983.

[FOXE 83b] Fox, E. A. "Some Considerations for Implementing the *SMART* Information Retrieval System under UNIX." Technical Report TR 83-560. Cornell University, Department of Computer Science, September 1983.

[FOXE 85] Fox, E. A. "Composite Document Extended Retrieval: An Overview." *Proceedings of the 8th Annual International ACM SIGIR Conference on Research & Development in Information Retrieval* (Montreal: June 1985), ACM, 1985, pp. 42-53.

[FOXE 86a] Fox, E. A. and R. K. France. "A Knowledge-Based System for Composite Document Analysis and Retrieval: Design Issues in the CODER Project." Technical Report TR-86-6. Blacksburg, VA: Virginia Tech Department of Computer Science, March 1986.

[FOXE 86b] Fox, E. A. and R. K. France. "Architecture of a Distributed Expert System for Composite Document Entry, Analysis, Representation and Retrieval." *Proceedings Third Annual USC Comp. Sci. Symp.; Knowledge-Based Systems: Theory and Applications* (Columbia, S.C.: March 31-April 1), 1986.

[FOXE 86c] Fox, E. A. "Information Retrieval: Research into New Capabilities." In Lambert, S. and Ropiequet, S. (Eds.). *CDROM - The New Papyrus*. Redmond, WA: Microsoft Press, 1986, pp. 143-174.

[FOXE 86d] Fox, E. A., Robert Wohlwend, Phyllis R. Sheldon, Qi Fan Chen and Robert K. France. "Building the CODER Lexicon: The Collins English Dictionary and Its Adverb Definitions." Technical Report TR 86-23. Blacksburg, VA: Virginia Tech Department of Computer Science, October, 1986.

[FOXE 87] Fox, E. A. "Development of the CODER System: A Testbed for Artificial Intelligence Methods in Information Retrieval." *Information Processing & Management* **23**:4 (1987), pp. 341-366.

[FRAN 86] France, Robert K. "An Artificial Intelligence Environment for Information Retrieval." Masters Thesis. Blacksburg, VA: Virginia Tech Department of Computer Science, June 1986.

[GARV 87] Garvey, Alan, Craig Cornelius and Barbara Hayes-Roth. "Computational Costs versus Benefits of Control Reasoning." *Proceedings of the Sixth National Conference on Artificial Intelligence* (Seattle, WA: July 13-17, 1987), AAAI, 1987, pp. 110-115.

[GAUT 81] Gauthier, Richard. *Using the UNIX System.* Reston, VA: Reston Publishing, 1981.

[GEHA 84] Gehani, Narain. *C: An Advanced Introduction.* Rockville, MD: Computer Science Press, 1984.

[GOLD 78] Goldstein, Charles M. and William H. Ford. "The User-Cordial Interface." *Online Review* **2**:3 (1978), pp. 269-275.

[GUID 83] Guida, Giovanni and Carlo Tasso. "An Expert Intermediary System for Interactive Document Retrieval." *Automation* **19**:6 (1983), pp. 759-766.

[HAHN 86] Hahn, Udo and Ulrich Reimer. "TOPIC Essentials." *Postfach 5560, D-7750 Konstanz 1*,Universitat Konstanz, April 1986.

[HART 86] Harter, Stephen P. *Online Information Retrieval: Concepts, Principles, and Techniques.* Orlando, FL: Academic Press, 1986.

[HAYE 79] Hayes-Roth, B., F. Hayes-Roth, F. Rosenschein, and S. Cammarata. "Modelling Planning as an Incremental, Opportunistic Process." *Proceedings of the Sixth International Joint Conference on Artificial Intelligence.* Los Altos, CA: William Kaufmann, 375-383.

[HAYE 83] Hayes-Roth, Frederick and Donald A. Waterman. *Building Expert Systems.* Reading, MA: Addison-Wesley, 1983.

[HAYE 84] Hayes-Roth, Barbara. "BB1: An Architecture for Blackboard Systems that Control, Explain, and Learn About Their Own Behavior." Technical Report STAN-CS-84-1034. Standford, CA: Department of Computer Science, Stanford University, December 1984.

[HAYE 85] Hayes-Roth, Frederick. "Rule-Based Systems." *Communications of the ACM* **28**:9 (September 1985), pp. 921-932.

[HOLC 85] Holcomb, Richard and Alan L. Tharp. "The Effect of Windows on Man-machine Interfaces." *Proceedings of the 1985 ACM Computer Science Conference - Agenda for Computing Research. The Challenge for Creativity.* (March 1985), pp. 12-14.

[HUU 86] Huu, C. T. and U. Kekeritz. "Eine Frame Implementation in Prolog." *Rundbrief des Fachausschusses* 1.2 der GI, (April 1986), pp. 19-25.

[INGW 86] Ingwersen, P. and I. Wormell. "Improved Subject Access, Browsing and Scanning Mechanisms in Modern On-line IR." *Proceedings of the 9th Annual International ACM SIGIR Conference on Research & Development in Information Retrieval* (Pisa, Italy: September, 1986), ACM, 1986, pp. 68-76.

[JOHN 87] Johnson Jr., M. Vaughn and Barbara Hayes-Roth. "Integrating Diverse Reasoning Methods in the BB1 Blackboard Control Architecture." *Proceedings of the Sixth National Conference on Artificial Intelligence* (Seattle, WA: July 13-17, 1987), AAAI, 1987, pp. 30-35.

[JONE 87] Jones, William P. and George W. Furnas. "Pictures of Relevance: A Geometric Analysis of Similarity Measures." *Journal of the American Society for Information Science* 38:6 (1987), pp. 420-442.

[KHAN 88] Khan, Mahtab R. "Support Routines for the CODER System." Masters Project. Blacksburg, VA: Virginia Tech Department of Computer Science, in progress, 1988.

[KASS 87] Kass, Robert and Tim Finin. "Rules for the Implicit Acquisition of Knowledge About the User." *Proceedings of the 10th Annual International ACM SIGIR Conference on Research & Development in Information Retrieval* (New Orleans, LA: June 3-5, 1987), ACM, 1987, pp. 295-300.

[KERN 78] Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language.* Englewood Cliffs, NJ: Prentice-Hall, 1978.

[KERN 84] Kernighan, Brian W. and Rob Pike. *The UNIX Programming Environment.* Englewood Cliffs, NJ: Prentice-Hall, 1984.

[KOUS 86] Koushik, Prabhakar. "Project Report for CS5332, Information Storage and Retrieval." Class Project Report. Blacksburg, VA: Virginia Tech Department of Computer Science, June, 1986.

[LANC 78] Lancaster, Wilifrid F. *Toward Paperless Information Systems.* New York, NY: Academic Press, 1978.

[LEE 86] Lee, Newton S. "Programming with P-Shell." *IEEE Expert* (1986), pp. 50-63.

[LEFF 84] Leffler, Samuel J., Robert S. Fabry and William N. Joy. "A 4.2BSD Interprocess Communication Primer." In *ULTRIX-32 Supplementary Documents,* Vol. III, Merrimack, NH: DEC, 1984, pp. 3-5 to 3-28.

[LENA 86] Lenat, Doug, Mayank Prakash and Mary Shepherd. "CYC: Using Common Sense Knowledge to Overcome Brittleness and Knowledge Acquisition Bottlenecks." *AI Magazine* (Winter 1986), pp. 65-84.

[LEVE 84] Levesque, Hector J. "Foundations of a Functional Approach to Knowledge Representation." *Artificial Intelligence* 23:2 (July 1984), pp. 155-212.

[LESS 83] Lesser, V. R. and D. D. Corkill. "The Distributed Vehicle Monitoring Testbed: A Tool for Investigating Distributed Problem Solving Networks." *AI Magazine* 4:3 pp. 15-33.

[LOGA 86] Logan, E. L. and Nancy N. Woelfl. "Individual Differences in Online Searching Behavior of Novice Searchers." *ASIS-86: Proceedings of the 49th ASIS Annual Meeting,* Knowledge Industry Publications, 1986, pp. 163-166.

[MALO 87] Malone, Thomas W., Kenneth R. Grant, Franklyn A. Turbak, Stephen A. Brobst and Michael D. Cohen. "Intelligent Information-Sharing Systems." *Communications of the ACM* 30:5 (May 1987), pp. 390-402.

[MARB 85] Marbach, William D. "The Race to Build a Supercomputer." In [FORE 85], 1985, pp.60-70.

[MARC 85] Marcus, Richard S. "Design Questions in the Development of Expert Systems For Retrieval Assistance." *ASIS-86: Proceedings of the 49th ASIS Annual Meeting*, Knowledge Industry Publications, 1986, pp. 185-189.

[METZ 85] Metzler, Douglas, Terry Noreault, Douglas P. Haas and Cynthia Cosic. "An Expert System Approach to Natural Language Processing." *ASIS-85: Proceedings of the 48th ASIS Annual Meeting*, Knowledge Industry Publications, 1985, pp. 301-307.

[MINS 81] Minsky, Marvin. "A Framework for Representing Knowledge." In J. Haugeland (Ed.). *Mind Design.* Cambridge, MA: The MIT Press, 1981, pp. 95-128.

[MYLO 84] Mylopoulos, John and Hector J. Levesque. "An Overview of Knowledge Representation." In Michael L. Brodie, Joachim W. Schmidt and John Mylopoulos (Eds.). *On Conceptual Modelling.* New York, NY: Springer-Verlag, 1984, pp. 3-16.

[NADO 87] Nado, Robert and Richard Fikes. "Semantically Sound Inheritance for a Formally Defined Frame Language with Defaults." *Proceedings of the Sixth National Conference on Artificial Intelligence* (Seattle, WA: July 13-17, 1987), AAAI, 1987, pp. 443-448.

[NAIS 85] Naish, Lee. *MU-Prolog 3.2db Reference Manual.* Melbourne University, July 1985.

[NII 82] Nii, H. Penny, E. A. Feigenbaum, J. J. Anton and A. J. Rockmore. "Signal-to-Symbol Transformation: HASP/SIAP Case Study." *AI Magazine* 3:2 pp. 23-35.

[NII 86A] Nii, Penny H. "Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures." *AI Magazine* 7:2 (Summer 1986), pp. 38-53.

[NII 86B] Nii, Penny H. "Blackboard Systems: Blackboard Application Systems, Blackboard Systems from a Knowledge Engineering Perspective." *AI Magazine* 7:3 (August 1986), pp. 82-106.

[NISO 87] National Information Standards Organization. "Proposed American National Standard for Information Science -- Common Command Language for Online Interactive Information Retrieval," Draft circulated for ballot, Gaithersburg, MD: National Bureau of Standards, Z39.59-198x, 1987.

[NORE 82] Noreault, Tom and R. Catham. "A Procedure for the Estimation of Term Similarity Coefficients." *Information Technology: Research & Development* (1982), pp. 189-196.

[NORM 86] Normore, Lorraine F. and Louis Tijerina. "Evaluation of Guidelines for Designing User Interface Software." *Proceedings of the Human Factors Society - 30th Annual Meeting* (1986), pp. 1363-1365.

[PATE 84a] Patel-Schneider, Peter F. "Small Can Be Beautiful In Knowledge Representation." *Workshop on Principles of Knowledge-Based Systems* (Denver, CO: Dec. 3-4, 1984), IEEE, 1984, pp. 11-16.

[PATE 84b] Patel-Schneider, Peter F. "ARGON: Knowledge Representation meets Information Retrieval." *The First Conference on Artificial Intelligence Applications* (Denver, CO: Dec. 5-7, 1984), IEEE, 1984, pp. 280-286.

[PERE 83] Pereira, Fernando. "Logic For Natural Language Analysis." Phd. Dissertation, University of Edinburgh, January 1983.

[PIGM 84] Pigman, Victoria. "The Interaction Between Assertional and Terminological Knowledge in Krypton." *Workshop on Principles of Knowledge-Based Systems* (Denver, CO: Dec. 3-4, 1984), IEEE, 1984, pp. 3-10.

[POLL 87] Pollitt, Steven. "CANSEARCH: An Expert Systems Approach to Document Retrieval." *Information Processing & Management* 23:2 (1987), pp. 119-138.

[RAMA 85] Ramamohanarao, Kotagiri and John Shepherd. "A Superimposed Codeword Indexing Scheme for Very Large Prolog Databases." Technical Report 85/17. Victoria, Australia: Melbourne University Department of Computer Science, 1985.

[RICH 79] Rich, Elaine. "User Modelling via Stereotypes." *Cognitive Science* 3 (1979), pp. 329-354.

[RICH 83] Rich, Elaine. *Artificial Intelligence.* New York, NY: McGraw-Hill, 1983.

[SACK 83] Sacks-Davis R. and K. Ramamohanarao. "A Two Level Superimposed Coding Scheme for Partial Match Retrieval." *Information Systems* 8:4 (1983), pp. 273-280.

[ROAC 85] Roach, John and Glenn Fowler. "Virginia Tech Prolog/Lisp." Blacksburg, VA: Virginia Tech Department of Computer Science, 1985.

[SALT 83a] Salton, Gerard, C. Buckley and Edward A. Fox. "Automatic Query Formulations in Information Retrieval." *Journal of the American Society for Information Science* 34:4 (July 1983), pp. 262-280.

[SALT 83b] Salton, Gerard, Edward A. Fox and Harry Wu. "Extended Boolean Information Retrieval." *Communications of the ACM* 26:11 (November 1983) pp. 1022-1036.

[SALT 83c] Salton, Gerard and Michael J. McGill. *Introduction to Modern Information Retrieval.* New York, NY: McGraw-Hill, 1983.

[SEN 86] Sen, Mahasweta. "Communication Through The CODER Blackboard." Class Project Report. Blacksburg, VA: Virginia Tech Department of Computer Science, June, 1986.

[SCHN 86] Schnupp, Peter and Lawrence W. Bernhard. *Productive Prolog Programming.* Englewood Cliffs, NJ: Prentice-Hall, 1986.

[SIMO 83] Simons, G. L. *Towards Fifth-Generation Computers.* England: NCC Publications, 1983.

[SIU 87] Siu, Lydia. "CODER - Boolean Search Expert." Class Project Report. Blacksburg, VA: Virginia Tech Department of Computer Science, June, 1987.

[SMEA 81] Smeaton, A. F. and C. J. Van Rijsbergen. "The Nearest Neighbor Problem in Information Retrieval: An Algorithm Using Upper Bounds." *Proceedings of the 4th Annual International ACM SIGIR Conference on Research & Development in Information Retrieval* (Oakland, CA: May 31 - June 2, 1981), ACM, 1981, pp. 83-87.

[SMIT 80] Smith, Linda C. "Artificial Intelligence Applications in Information Systems." *ARIST* **15** (1980), pp. 67-106.

[SMIT 84] Smith, L. C. and A. J. Warner. "A Taxonomy of Representations in Information Retrieval System Design." In Hans J. Dietschmann (Ed.). *Representation and Exchange of Knowledge as a Basis of Information Processes,* NY: North-Holland, 1984, pp. 31-49.

[SMIT 87a] Smith, John B., Stephen F. Weiss and Gordon J. Feruson. "MICROARRAS: An Advanced Full-Text Retrieval and Analysis System." *Proceedings of the 10th Annual International ACM SIGIR Conference on Research & Development in Information Retrieval* (New Orleans, LA: June 3-5, 1987), ACM, 1987, pp. 187-195.

[SMIT 87b] Smith, Linda C. "Artificial Intelligence and Information Retrieval." *Annual Review of Information Science and Technology* **22** (1987), pp. 41-77.

[SPAR 87] Sparck-Jones, Karen. "Architecture Problems in the Construction of Expert Systems for Document Retrieval." Paper presented at the *Workshop on Distributed Expert-Based Information Systems.* School of Communication, Information and Library Studies, Rutgers University, (March 1987).

[SRID 87] Sridharan, N. S. "Report on the 1986 Workshop on Distributed Artificial Intelligence." *AI Magazine* **8**:(3) (Fall 1987), 75-85.

[STER 86] Sterling, Leon and Ehud Shapiro. *The Art of Prolog.* Cambridge, MA: The MIT Press, 1986.

[TAGU 87]. Tague, Jean. "Generating an Individualized User Interface: From Novice To Expert." *Proceedings of the 10th Annual International ACM SIGIR Conference on Research & Development in Information Retrieval* (New Orleans, LA: June 3-5, 1987), ACM, 1987, pp. 57-60.

[TERR 83] Terry, A. "The CRYSALIS Project: Hierarchical Control of Production Systems." Technical Report HPP-83-19. Stanford, CA: Stanford University, Heuristic Programming Project, 1983.

[THOM 85] Thompson, Roger H. and W. Bruce Croft. "An Expert System for Document Retrieval." *Proceedings of Expert Systems in Government Symposium* (Mclean, VA: October 1985), IEEE, 1985, pp. 448-456.

[THOM 86] Thompson, R. H. and W. B. Croft. "I^{3}R: A New Approach to the Design of Document Retrieval Systems." *Journal of the American Society for Information Science* (April 1986).

[THOM 87a] Thompson, Roger. "An Implementation Overview of I^{3}R." Paper presented at the *Workshop on Distributed Expert-Based Information Systems.* School of Communication, Information and Library Studies, Rutgers University, (March 1987).

[THOM 87b] Thom, James A. and Justin Zobel, Eds. "Extracts from NU-Prolog Reference Manual," Version 1.1. Technical Report 86/10. Victoria, Australia: University of Melbourne, Department of Computer Science, May 1987.

[THOM 87c] Thom, James A. and Justin Zobel (Eds.). "NU-Prolog Reference Manual." Victoria, Australia: Department of Computer Science, University of Melbourne, May, 1987.

[TONG 86a] Tong, Richard M., Lee A. Appelbaum, Victor N. Askman and James F. Cunningham. "RUBRIC III - An Object-Oriented Expert System for Information

Retrieval." *Proceedings of Expert Systems in Government Symposium* (Mclean, VA: October 22-24, 1986), IEEE, 1986, pp. 106-115.

[TONG 86b] Tong, Richard M., Lee A. Appelbaum, Victor N. Askman and James F. Cunningham. "Conceptual Information Retrieval using RUBRIC." *Proceedings of the 10th Annual International ACM SIGIR Conference on Research & Development in Information Retrieval* (New Orleans, LA: June 3-5, 1987), ACM, 1987, pp. 247-253.

[VANR 86]. Van Rijsbergen, C. J. "A New Theoretical Framework for Information Retrieval." *Proceedings of the 9th Annual International ACM SIGIR Conference on Research & Development in Information Retrieval* (Pisa, Italy: September, 1986), ACM, 1986, pp. 194-200.

[VICK 87] Vickery, A. and H. M. Brooks. "PLEXUS - The Expert System for Referral." *Information Processing & Management* 23:2 (1987), pp. 99-117.

[WATE 86] Waterman, Donald A. *A Guide to Expert Systems.* Reading, MA: Addison-Wesley, 1986.

[WATT 87] Watters, C. R. and M. A. Shepherd. "Towards an Expert System for Bibliographic Retrieval: A Prolog Prototype." *Proceedings of the 10th Annual International ACM SIGIR Conference on Research & Development in Information Retrieval* (New Orleans, LA: June 3-5, 1987), ACM, 1987, pp. 272-281.

[WEAV 86a] Weaver, Marybeth T. "The CODER Search Expert." Class Project Report. Blacksburg, VA: Virginia Tech Department of Computer Science, December, 1986.

[WEAV 86b] Weaver, Marybeth T. "The CODER Search Expert Integration." Class Project Report. Blacksburg, VA: Virginia Tech Department of Computer Science, June, 1986.

[WEAV 86c] Weaver, Marybeth T. "A Frame-Based Knowledge Representation System." Class Project Report. Blacksburg, VA: Virginia Tech Department of Computer Science, January, 1986.

[WEIS 84] Weiss, Sholom and Kulikowski, Casimir. *A Practical Guide to Designing Expert Systems.* Totawa, NJ: Rowman & Allanheld, 1984.

[WENB 86] Wenban, Jim. "CS5332 Class Project - Creating Prolog Facts from the HAI." Class Project Report. Blacksburg, VA: Virginia Tech Department of Computer Science, June, 1986.

[WILL 84] Williams, M., H. Brown and T. Barnes. "TRICERO Design Description." Technical Report ESL-NS539. ESL Inc., 1984.

[WOHL 86] Wohlend, Robert C. "Creation of a Prolog Fact Base from the *Collins English Dictionary.*" Masters Project Report. Blacksburg, VA: Virginia Tech Department of Computer Science, March 1986.

[YIP 81] Yip, Man-Kam. "An Expert System for Document Retrieval." Masters Thesis. Cambridge, MA: Massachusetts Institute of Technology, February, 1981.

Appendix A. Prolog Utilities List

PROLOG UTILITIES INDEX

For information regarding the arguments and details of each utility, reference the utilities library (~coder/UTIL).

aless	Less than comparison for atoms.
all	All solutions. Same as 'findall', 'setof'.
append	Appends two lists to form one new list.
append3	Appends three lists together.
append_n	Append a list of lists, ignoring the null list.
ansi_io	Group of standard predicates needed for terminal IO.
bestlist	Finds the best value in a list according to some binary predicate.
control2	'Loop' and 'invoke' for predicates.
convert	Conversions between integers, constants, and strings.
cqueen	For 8 queens problem.
cross	General purpose problem solving procedure.
db	For Mu-Prolog database.
delete	Delete all occurences of one element from a list.
delete_all	Delete all occurences of a list of elements from a list.
dsimc	For Mu-Prolog database, superimposed coding.
efface	Delete first occurence of an element from a list.
env_funt	Group of environment function enhancements.
epilog	Epilog -and- connective special coroutining.
extended_logic	Extended logic predicate, e.g., 'or'.
findall	Findall occurrences of an item in a prolog fact base. See 'setof' and 'setof1'.
for_all	Apply a predicate to each member of an op_list.
general	Anti-unification.
genwait	Generates wait declarations for a procedure.
gnot	Generalized negation predicate.
ground	Predicates for grounding.
hex	Conversion between hex integers and strings.
input_output	User I/O, includes pretty-printing trees, strings, ...
insert	Inserts an element into a list.
is_real_list	Allows only properly formed lists (i.e., no dotted lists).
islist	Allows any list.
isort	Insertion sort.

last	Last element in a list.
list_length	Number of elements in a list.
lqueen	Logic of 8 queens problem (input to preprocessor).
lquery	Logic query processor.
mac	Missionaries and cannibals problem.
maplist	Maps one list onto a new list based on a Predicate arg.
matchlist	Matches one list to another list based on a list of common elements.
maxlist	Maximum element in a list.
member	List membership.
member_rest	List membership and returns the rest of the list.
merge	Merges 2 non-decreasing lists (duplicates not removed).
merge2	Random merge of 2 ordered lists 〃
minlist	Minimum element in a list.
next_to	Adjacency between 2 elements in a list.
no_dupls	Succeeds if a list has no duplicates.
non_null_list	Succeeds if a list is non-null.
nqueen	Front-end for the n-queens problem.
numbervars	Number of variables in a term.
octal	Conversion of octal integers and strings.
osets	Ordered set manipulation utilities (R.A.O'Keefe).
perm	Permutation of 2 lists (with wait predicate).
permute	Permutation of 2 lists (no wait predicate).
plin.pl	Prolog formatter and comment remover.
position	Element in postition X in a list.
preds	'all', 'notall', 'some', and 'none' for predicate success.
prefix	Succeeds iff 'part' is a leading substring of a whole.
primes	Prime number sieve.
random	Generate random number between 1 and specified X.
rationalize	Rationalize number.
read_in	Create list of words from an English sentence.
reduce	Apply bin-op left associatively to elements in a list.
reduced	Reduce an expression to lowest terms.
remove_dups	Removes all duplicates from a list creating a new list.
repeat_list	Create a list be repeating an Element X times.
reverse	Reverse elements in a list.
rlhp	MuProlog DB, recursive linear hashing program.
setof	Like 'findall'; also contains 'bagof'. (Unsound inequality)
setof1	Like 'findall'; also contains 'bagof'. (Sound inequality)
sets	Predicates for sets (e.g., ordered lists). Includes subset, intersection, union, set-diff, set_equal, set_plus,powerset, partition, closure.
simc	MuProlog DB. Superimposed coding scheme program.
sl	Same leaves (for processing tree structures).
sort	Quick sort.
stdio	Standard I/O routines, e.g. centering display, menus.
string_of	Succeeds if string is composed of elements of given alphabet.
structures	Predicates for atomic/compound terms, and tree position.

sublist	List is a sublist in a list.
subst	Substitute all occurrences of one element by another.
suffix	Succeeds iff 'part' is a trailing substring of a whole.
system	All system predicates.
term_compare	Determines relationship between 2 terms (eg =, <, >)
terms	Tests term types, e.g., atomic, float, instantiated,...
times	Divide and multiply predicates.
traperror	Traps an undefined procedure call, and checks whether the procedure is available in some other library.
trim	Trim leading and/or trailing elements of a list.
unixutil	Contains UNIX utilites 'cat', 'cp', 'move'.
vtrace	Trace binding of variables in a goal.

Appendix B. Prolog Programming Standard

CODER Prolog Programming Standard

Adopted from Prolog Digest Volume 4, Number 80, Summer 1986. Submitted by John Cugini.

Many predicates expect certain of their arguments to be instantiated upon invocation. When such restrictions apply it is usually the leading arguments which are thought of as input (and hence uninstantiated), and the trailing arguments as output (and hence allowed to be uninstantiated). A standard way of denoting the status of arguments to a Prolog predicate is to include a comment line before the body of the clause, in which arguments expected to be instantiated are prefixed by '+', uninstantiated arguments arguments by a '-', and arguments where it doesn't matter (or where either can be used) by '?'. For example,

% append(+L1, +L2, -L3)

indicates the status of the arguments to the usual use of the standard 'append' clause.

Appendix C. Testing Communications Extensions

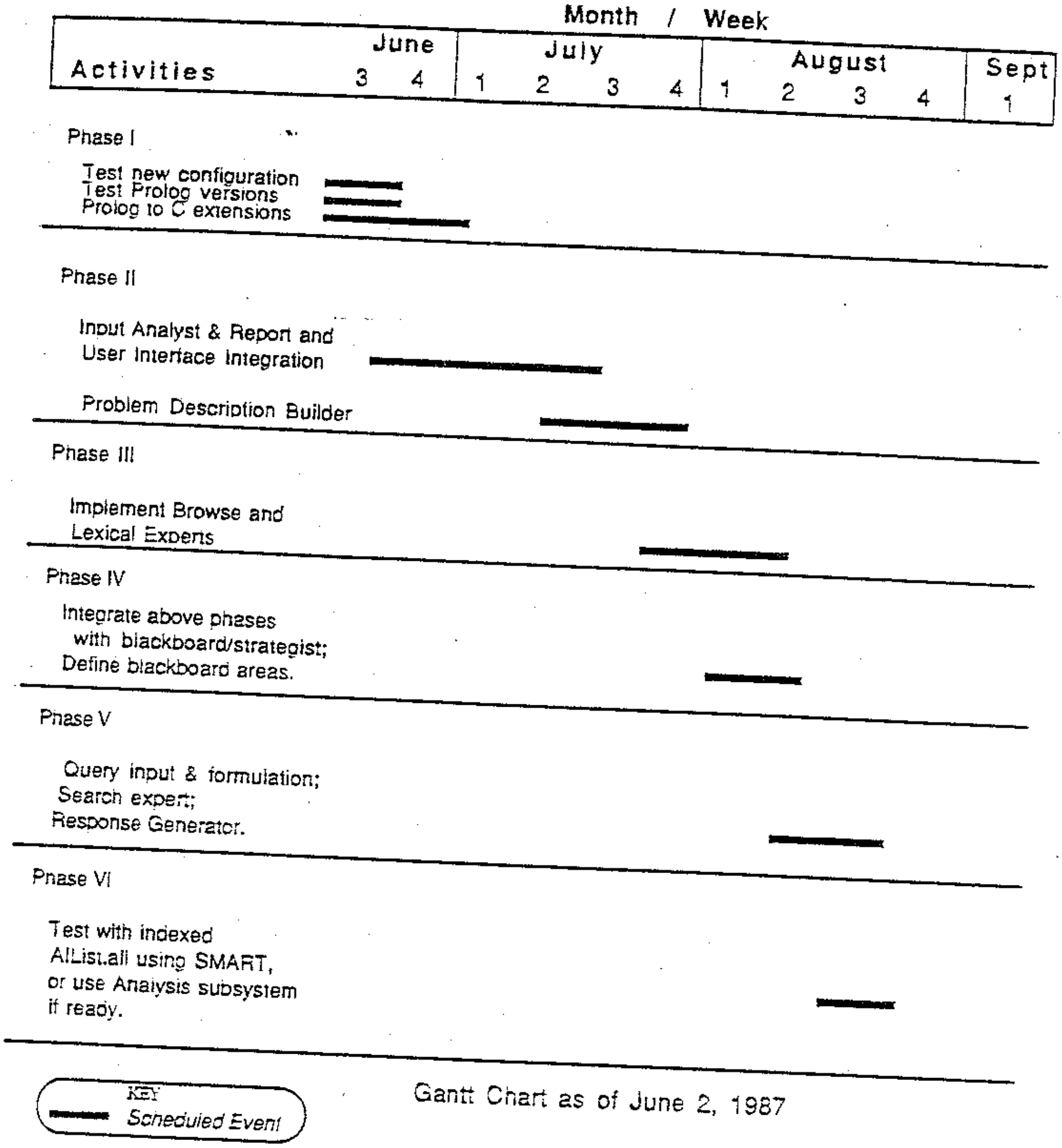
To test, for example, the hai manager:

```
# Execute hai_mgr in background.

 >hai_mgr&

% Go into one of the prolog versions with the built-in ask predicate.

 >prolog-s

% To be sure that the hai_mgr socket is functioning properly, test
% with the hello function.

 > ask(hai_mgr,hello(X)).

   X = hai_mgr

% Check some of the functions of the hai_mgr.

 > ask(hai_mgr,preprocess(test_filename)).

   true.

% Extract line number 10 from the file, test_filename.
% System should respond with the name of a temporary file.

 > ask(hai_mgr,extractline(test_filename,10,Text_filename)).

   Text_filename = /tmp/hai_test_filename12345

% Exit Prolog.

CtlD

# Terminate the hai_mgr running in background.

kill %1

# Remove any temporary files placed in the ~tmp directory.

rm /tmp/hai_test_filename12345
```

Note that the hai_mgr must be in the map file before Prolog can be used to transmit an ask command. Otherwise, the error "unknown server module" will be displayed.

Appendix D. Implementation Schedule, June 1987

CODER Retrieval Subsystem Tasks



Gantt Chart as of June 2, 1987

Appendix E. Modified Blackboard/Strategist Specifications

Blackboard / Strategist Complex: Functional Specification

There are five functional modules within the blackboard/strategist composite. In the blackboard proper, the **posting area manager** maintains the integrity of the posting areas while responding to the external *post, retract,* and *view* commands. The other four functional modules are considered parts of the strategist. The **logic task scheduler** is responsible for scheduling new tasks necessary to maintain the validity of the hypotheses on the blackboard, while the **domain task scheduler** schedules new tasks based on domain-specific blackboard events. The **question / answer handler** uses domain-specific knowledge of which experts are able to answer what questions to schedule the tasks involved in the question/answer process. Proposed tasks from these three sources are picked up by the **task dispatcher** and passed to the experts in the local community. The external calls **wake**, **attend**, and **stop** actually originate from the dispatcher, and the **done** and **checkpoint** communications from the experts are received by it.

0.1. Common Data Structures

Hypothesis -- A 5-tuple

<fact, confidence, expert, id, dependencies>

where:

'fact' is a CODER fact,
'confidence' the confidence the expert has in the fact,
'expert' the ID of the specialist making the hypothesis,
'id' a unique identifier for the hypothesis within the session, and
'dependencies' is a list [or possibly, a p-norm expression] of id's for the hypotheses used by the expert in producing the current hypothesis.

When a fact is posted by an expert, 'id' is passed as an unbound variable. A value is supplied for the variable by the blackboard and returned. All other elements of the tuple should be supplied by the posting expert: in the event that the hypothesis depends on no previous hypotheses, 'dependencies' may be bound to the empty list [].

Question -- A skeletal fact, i.e., a set of CODER variables coupled to a fact, one or more arguments of which are replaced by the variables. Each question is posted to the reserved Question area of the blackboard until it accumulates an adequate set of answers, upon which it is returned to the specialist that originated it. A question has an expert-id, dependency information and an id of its own, but no confidence.

Answer -- A binding list that partially instantiates the question with which it is associated (ie, a subset of the variables in the question together with CODER knowledge structures with which they can be replaced) signed, confidence-rated and justified by the answering expert. (In other words, an answer is exactly like an hypothesis, except that the fact is replaced by a binding list of variables from the question).

Task -- A 5-tuple

<expert, command, scheduler, priority, time>

where:

'expert' is the ID of an expert in the local community,
'command' one of the expert entry points (*wake, attend* &c.),
'scheduler' either 'logic', 'question' or 'domain',
'priority' the priority assigned by the scheduler, and
'time' the date/time when the task was entered.

1. The Posting Area Manager

All interactions with a posting area of the blackboard, either by the external community or by the other modules of the blackboard/strategist complex, occur through the posting area manager. Each area of the blackboard, whether a subject posting area, the pending hypothesis area, or the question/answer area, can be directly accessed only by the posting area manager. The area manager thus must provide the functionality for all post, retract, and view commands. In addition, it must notify the two task schedulers when events occur on the blackboard that may have scheduling repercussions. Specifically, whenever a hypothesis is posted to a subject area, both task schedulers must be notified;

the domain task scheduler with the type of hypothesis posted (the head relation) and the logic task scheduler with the dependencies of the new hypothesis. The domain task scheduler must similarly be informed when a question or an answer is posted, and the logic task scheduler when a hypothesis is retracted or replaced. The exact syntax of these calls is defined below under the headings for the two schedulers.

Still more specifically, the following lists the activity required by each of the calls to the posting area manager:

post_hypothesis (Hyp, Area). -- The hypothesis 'Hyp' is added to the subject posting area 'Area', and its id is bound to a new identifier. The hypothesis is time-stamped, and the logic task scheduler is passed its dependency information. If 'Hyp' involves a fact already posted to the area by the originating expert (if it **replaces** an earlier hypothesis), then the earlier hypothesis is removed from the posting area, and the logic scheduler is notified of this as well. If not (if the hypothesis involves a **new** fact), then only the domain task scheduler is notified.

retract_hypothesis (Hyp_id, Expert_id). -- The hypothesis 'Hyp_id' is removed from the blackboard, and the logic task scheduler notified, using the difference between the confidence value of the retracted hypothesis and 0 (or the nill confidence value) as the change in confidence. Only the expert which posted the hypothesis may retract it.

post_question (Quest_id, [Expert_id, Quest, Dep]). -- The question 'Quest' is added to the question posting area, and the question/answer handler notified.

post_answer (Quest_id, Ans). -- The answer 'Ans' is added to the set of answers to the question with 'Quest_id' in the question posting area. The question/answer handler is notified. Note: answers can be replaced under the same conditions as hypotheses (if the hypothesized fact and the answering expert are the same), but no notification is made of such changes, as no hypotheses can be dependent on answers still in the question/answer area.

retract_answer (Quest_id, Ans_id, Expert_id). -- The answer with 'Ans_id' is removed from the set of answers to question 'Quest_id' in the question posting area. Only the expert which posted the answer may retract it.

view_area (Area, Status, Expert_id, Hyp_set). -- Hypotheses currently in the subject posting area 'Area' are collected into 'Hyp_set' and returned. The Status and the id of the Expert requesting the hypothesis set determine which hypotheses are returned: those which no expert has yet processed (new); those which the requesting expert has not processed, although other experts may have processed them (seen); those the expert has already processed (old), those that have been modified, or all of the above. Status must be 'new', 'old', 'all', 'seen', or an integer representing the time since the hypotheses were last modified. When a time integer is supplied, only hypotheses which have been modified since that time will be returned in Hyp_set.

view_hyp(Hyp_id, Hypothesis). -- The hypothesis tuple for the given Hyp_id is returned.

view_quest(Quest_id, Question). -- The question list of [Originating_expert,Question,Dependencies] for the question 'Quest_id' is returned as Question.

view_questions (Quest_set). -- All hypotheses currently in the question posting area are collected into 'Quest_set' and returned.

view_answers (Quest_id, Ans_set). -- The current set of answers for question 'Quest_id' is returned as 'Ans_set'.

view_pending (Hyp_set). -- The current set of hypotheses in the pending hypothesis area is returned as 'Hyp_set'.

hyp_processed (Hyp_id, Expert_id, Time). -- Notifies the posting area manager that hypothesis 'Hyp_id' has been processed by 'Expert_id' at time 'Time'. This predicate updates the Status of blackboard hypotheses processed by the Expert.

clear_hyps([Functors]). -- Notifies the posting area manager that hypotheses with Facts having a head relation matching one of those in the 'Functors' list may be removed from the hypothesis fact base. This predicate prevents accumulation of hypotheses which will not be used again and could significantly slow processing.

All of the calls provided in the external view of the blackboard are available to the strategist scheduling modules. In addition, the posting area manager provides certain further calls to the strategist modules only. These are:

retract_question (Quest_id, Ans_set, Expert_id). -- The question with 'Quest_id' is removed from the question/answer area, and the (possibly empty) set of answers accumulated up to the time of call returned as 'Ans_set'.Only the expert which posted the question may retract it.

2. The Logic Task Scheduler

It is the responsibility of the logic task scheduler to maintain the consistency of the deduction trees implicit in the hypotheses on the blackboard under the conditions of possibly changing premises. The knowledge represented on the blackboard is non-monotonic: hypotheses may be retracted or replaced at any time. When this occurs, the hypotheses dependent on the retracted or replaced fact must sometimes be retracted or replaced themselves. The logic task scheduler accomplishes this by scheduling reconsiderations of hypotheses that may be effected: i.e., by scheduling tasks of the form *attempt_hyp (Rel)* for the expert hypothesizing the suspect fact.

In order to perform this scheduling, the logic scheduler draws upon dependency information and maintenance knowledge. The information is received from the posting area manager, and the knowledge is represented in a rule base relating the stimuli of retractions and changes of confidence together with the closeness of the dependency, to responses in terms of task postings at various priorities. For example, one rule might be:

```
IF
        The confidence level of Hyp_A has decreased Amount_1  AND
        Hyp_B is dependent on Hyp_A with level Amount_2,
THEN
        Schedule <Expert_B, attempt_hyp(head(Hyp_B))> with priority of
                Base_level*Amount_1*Amount_2.
```

Other information available for triggering rules includes the age of the hypotheses and the number of dependenceis an individual hypothesis has. Note that reconsiderations will propagate nataurally through the deduction tree above the replaced or retracted hypothesis as the logic-maintenance tasks result in the suspect hypotheses themselves being changed or withdrawn. It is the responsibility of the rule base creator to ensure that such propagation is damped appropriately and does not always result in every hypothesis in the tree being reconsidered.

The logic task scheduler reacts only to the operations of the posting area manager, to which it provides the following calls:

new_dependencies (Hyp_id, Rel, Expert, Confidence, Dependencies). -- Hypothesis 'Hyp_id' has just been posted. It has head-relation 'Rel' and is dependent on the hypotheses in 'Dependencies'. It was posted by expert 'Expert' with confidence value 'Confidence'. This call should initiate no scheduling activity, but the information must be logged so that dependencies can be traced.

hyp_retracted (Hyp_id). -- Hypothesis 'Hyp_id' has just been retracted. The dependency graph must be updated, and reconsiderations may need to be scheduled.

hyp_replaced (Hyp_id, Change_in_conf). -- Hypothesis 'Hyp_id' has just been replaced by an hypothesis which differs in confidence by 'Change_in_conf'. Note that 'Change_in_conf' may be either positive, if the new hypothesis has a higher confidence level than the old, or negative, if it has less confidence. The dependency graph must be updated, and reconsiderations may need to be scheduled.

3. The Question/Answer Handler

Of the two application-specific schedulers in the strategist module, the question/answer handler is the more straightforward. Based on a set of rules associating each type of question (canonically, each head-relation) in the set of all questions that may be posed throughout the community with the set of experts possibly able to answer them, the question/answer handler reacts to postings of questions by posting tasks of the form *attend_quest* to the task posting area. When answers are posted to questions, the question/answer handler evaluates them for adequacy based (at least) on the confidence of the answer and which experts of those capable of answering the question have made an attempt. If the answer set is judged inadequate, new answering tasks are posted;

otherwise, the question is removed with its answer set from the question/answer area and sent to the originating expert in the form of an **answers** task.

Answers to questions must be evaluated for adequacy in the context of what other experts are available to attempt answers. Each type of question may have an expert or a set of experts that are best fit to answer it, but other experts may need to be called in if the first attempt to provide an adequate answer fails. An inadequate set of answers might alternatively cause the process of searching for an answer to be restarted, if conditions on the blackboard have changed suffiiently in the meanwhile, or might cause the task that produced the question to be restarted in hopes that a better formulation of the question might be obtained.

The question/answer handler is thus the only module in the system that makes use of the **retract_question** entry to the posting area manager. It uses, in addition, the **view_answers** and possibly the **view_questions** entry. In return, it provides the posting area manager with the triggers:

new_question (Quest_id, Rel). -- Question 'Quest_id' with head-relation 'Rel' has just been posted. Experts capable of answering questions with this head-relation should be scheduled.

new_answer (Quest_id, Ans_id, Conf). -- An answer ('Ans_id') to question 'Quest_id' has just been posted with confidence level 'Conf'. If the answer produces (either alone or with previously received answers) an adequate answer set, the question should be retracted from the question/answer area and consideration of the answer posted as a task for the originating expert. Otherwise, other processing should be undertaken to obtain an answer.

4. The Domain Task Scheduler

The domain task scheduler is responsible for proposing new tasks based on the progress of the current session and the mix of hypotheses currently on the blackboard. It is also responsible for selecting the hypotheses to be posted to the pending hypothesis area, again based on what has happened on the blackboard and what is happening at the moment. The domain-specific strategies for these two types of actions are represented in a set of rules, the antecendents of which are combinations of events within contexts, and the consequents of which are expert tasks to be posted and/or types of hypotheses to be moved to the pending area. For instance, a rule in the retrieval strategist might run:

```
IF
      An hypothesis establishing the document type has been posted
                                                                AND
      It has a confidence larger than Min_level                 AND
      (There is no other hypothesis of document type posted     OR
       The new hypothesis is a refinement of the former pending hypothesis)
THEN
      Move hypothesis to pending hypothesis area                AND
      Schedule  <doc_type_expert, attempt_hyp(fill_missing_fields)>
            with priority of K.
```

Rules need to be provided in building this local base to deal with reactions to individual hypothesis types, to questions and to answers to questions, all in the context of the phases of the overall task in which the community is engaged. At different phases of the process, different hypothesis postings may require different actions by a different mix of experts.

The domain task scheduler is informed by the posting area manager whenever a new hypothesis is posted. This is the primary stimulus for triggering rules:

new_hyp (Hyp_id, Rel, Dep, Conf). -- Hypothesis 'Hyp_id' with head-relation 'Rel', dependencies 'Dep', and confidence level 'Conf' has just been posted.

In addition, domain scheduling may be triggered by the task dispatcher, for instance when the task queue is empty, or when no tasks in the queue have priority greater than some particular threshhold.

5. The Task Dispatcher

The task dispatcher coordinates the tasks proposed by the three scheduling units and sends the actual commands to the requested experts. It maintains a priority queue of tasks, which may, however, not necessarily be executed in priority order. As well as by the priority assigned by the scheduling unit to the task, order is determined by the availability of resources (experts, for instance, execute tasks serially, so a task for a given expert may have to wait until the expert is finished), by the location of system modules (tasks for modules resident on different machines may be allocated at different priority levels, depending on the demands for those machines), and by heuristics balancing how crucial tasks proposed by the three experts are relative to one another.

The task dispatcher is triggered by two sorts of events. First, it is triggered whenever a new task is proposed by one of the scheduling units:

new_task (Task). -- 'Task' should be added to the queue. If desirable according to the above heuristics, it should be dispatched.

Second, the dispatcher is also triggered whenever an executing task is completed:

done (Expert). -- 'Expert' has completed its current task. Based on the task mix in the queue and the scheduling heuristics, another task may now be dispatched.

The task dispatcher maintains a history file of the progress of all the tasks executed during a session. When each expert signals successful receipt of a task, the dispatcher notes the time the task has begun in the history file. It notes the time of completion of each task at the receipt of each *done*. It uses the information in this file to determine which experts are running at any point. The file is also used by the domain task expert, and possibly the question/answer handler.

Appendix F. Frame Creation Session

```
Script started on Thu Dec 31 09:08:25 1987
1%prolog
MU-Prolog 3.2ddb


1?- [knowadm].
consulting knowadm
done

true.

2?- update.

Please enter function desired:

 1.  New Elementary Data Type
 2.  New Frame
 3.  New Relation
 4.  Save updates
99.  Terminate processing
1.
Files being consulted, please wait

Please enter, one by one, each parameter of the new EDT followed by a period.

EDT type name

summer_qtr.

parent EDT

int.

quantifier (or [] )

[].

Restriction in list format:  [not,x,y,..]  [member,x,y,...]
[min,x,max,y]  or [])

[member,6,7,8].


You have added an EDT called summer_qtr with the following parameters:
quantifier  =  []
```

```
parent_type = int, parent restriction = []
restriction = [member, 6, 7, 8]

Please enter function desired:

 1. New Elementary Data Type
 2. New Frame
 3. New Relation
 4. Save updates
99. Terminate processing
2.

Please enter, one by one, each parameter of the new frame followed by .

Frame type name

summer_sched.

Frame parent(s) list , e.g., [parent1, parent2] or []

[].

PLEASE ENTER NEW SLOTS TO BE ADDED FOR THIS FRAME

Enter slot name  (or enter stop to terminate)

class_name.

Slot class  (e,f,r for edt, frame, or relation)

e.

Slot EDT, frame, or relation type

char.

Slot cardinality minimum (may be [])

[].

Slot cardinality maximum (may be [])

3.
You have added a new slot called class_name

More slots? (y/n)

y.

Enter slot name  (or enter stop to terminate)

month_offered.

Slot class  (e,f,r for edt, frame, or relation)

e.
```

```
Slot EDT, frame, or relation type

summer_qtr.

Slot cardinality minimum (may be [])

[].

Slot cardinality maximum (may be [])

3.
You have added a new slot called month_offered

More slots? (y/n)

y.

Enter slot name  (or enter stop to terminate)

instructor.

Slot class  (e,f,r for edt, frame, or relation)

f.

Slot EDT, frame, or relation type

individual.

Slot cardinality minimum (may be [])

[].

Slot cardinality maximum (may be [])

[].
You have added a new slot called instructor

More slots? (y/n)

y.

Enter slot name  (or enter stop to terminate)

course_nbr.

Slot class  (e,f,r for edt, frame, or relation)

e.

Slot EDT, frame, or relation type

char.

Slot cardinality minimum (may be [])
```

```
[].

Slot cardinality maximum (may be [])

[].
You have added a new slot called course_nbr

More slots? (y/n)

n.

Each EDT slot will be displayed, so that you may enter defaults

SLOT:class_name, char,   CARD: [],3, DEF: []

Enter new default ([] for no change):

[].

SLOT:course_nbr, char,   CARD: [],[], DEF: []

Enter new default ([] for no change):

[].

SLOT:month_offered, summer_qtr,   CARD: [],3, DEF: []

Enter new default ([] for no change):

6.

You have added a new frame type called summer_sched
Frame parent(s) []

SLOT:class_name, e, char,   CARD: [],3, DEF: []

SLOT:course_nbr, e, char,   CARD: [],[], DEF: []

SLOT:instructor, f, individual,   CARD: [],[], DEF: []

SLOT:month_offered, e, summer_qtr,   CARD: [],3, DEF: 6

Please enter function desired:

  1.  New Elementary Data Type
  2.  New Frame
  3.  New Relation
  4.  Save updates
 99.  Terminate processing
99.
EDT type file being rewritten, please wait

Frame type file being rewritten, please wait
```

```
No new relations created.
Termination in process as requested

3?- ka_edt(summer_qtr,Parent,Quantifier,Restrictions).

Parent = [],
Quantifier = int,
Restrictions = [member, 6, 7, 8]

4?- ka_frame(summer_sched,Parents,Slotlist).

Parents = [],
Slotlist =
            [class_name, e, char, [], 3, [],
             course_nbr, e, char, [], [], [],
             instructor, f, individual, [], [], [],
             month_offered, e, summer_qtr, [], 3, 6]

5?- ¬D
End of session
2%¬D
script done on Thu Dec 31 09:15:34 1987
```

Appendix G. Frame Type Definitions

CODER FRAME TYPE DEFINITION REPORT

FRAME TYPE: time *PARENT(S): []*

SLOTS:

Slotname	Type*	Type name
hour	e	hour
minute	e	minute
second	e	second
ampm	e	ampm
time_zone	e	time_zone

FRAME TYPE: date *PARENT(S): []*

SLOTS:

Slotname	Type	Type name
year	e	year
month	e	month
day_of_month	e	day_of_month
day_of_week	e	day_of_week

FRAME TYPE: date_time *PARENT(S): []*

SLOTS:

Slotname	Type	Type name
date	f	date
time	f	time

FRAME TYPE: name *PARENT(S): []*

SLOTS:

Slotname	Type	Type name
date_established	f	date
educ_status	e	char
first	e	char
last	e	char
middle	e	char
suffix_jr_sr	e	char
title	e	char

* Types are: e = Elementary Data Type (EDT), f = frame, r = relation.

CODER FRAME TYPE DEFINITION REPORT

FRAME TYPE: phone *PARENT(S): []*

SLOTS:

Slotname	**Type**	**Type name**
area_code	e	char
prefix	e	char
local_nbr	e	char
extension	e	char

FRAME TYPE: u_s_address *PARENT(S): [address]*

SLOTS:

Slotname	**Type**	**Type name**
p_o_box	e	char
street	e	char
city	e	char
state	e	char
country	e	char
zip_code	e	char

FRAME TYPE: non_u_s_address *PARENT(S): [address]*

SLOTS:

Slotname	**Type**	**Type name**
p_o_box	e	char
street	e	char
city	e	char
province	e	char
country	e	char
postal_code	e	char

FRAME TYPE: address *PARENT(S): []*

SLOTS:

Slotname	**Type**	**Type name**
p_o_box	e	char
street	e	char
city	e	char
country	e	char

FRAME TYPE: educational *PARENT(S): [postal_address]*

SLOTS:

Slotname	**Type**	**Type name**
department	e	char
school	e	char
university	e	char
address	f	address

CODER FRAME TYPE DEFINITION REPORT

FRAME TYPE: non_educational *PARENT(S): [postal_address]*

SLOTS:

Slotname	Type	Type name
organization	e	char
address	f	address

FRAME TYPE: postal_address *PARENT(S): []*

SLOTS:

Slotname	Type	Type name
address	f	address

FRAME TYPE: email_address *PARENT(S): []*

SLOTS:

Slotname	Type	Type name
user_id	e	char
source_node	f	node
relay	f	node
route	e	char

FRAME TYPE: node *PARENT(S): []*

SLOTS:

Slotname	Type	Type name
local	e	char
net_name	e	char
domain	e	char

FRAME TYPE: individual *PARENT(S): []*

SLOTS:

Slotname	Type	Type name
name	f	name
postal_address	f	postal_address
email_address	f	email_address
phone	f	phone
affiliation	e	char

FRAME TYPE: discussion_time *PARENT(S): []*

SLOTS:

Slotname	Type	Type name
start	f	time
end	f	time

CODER FRAME TYPE DEFINITION REPORT

FRAME TYPE: place *PARENT(S): []*

SLOTS:

Slotname	**Type**	**Type name**
room	e	char
floor	e	char
building	e	char
hall	e	char

FRAME TYPE: seminar *PARENT(S): [doctype]*

SLOTS:

Slotname	**Type**	**Type name**
seminar_name	e	char
seminar_time	f	time
seminar_date	f	date
seminar_place	f	place
seminar_title	e	char
discussion_place	f	place
discussion_time	f	time
speaker	f	individual
host	f	individual
abstract_span	f	span

FRAME TYPE: span *PARENT(S): []*

SLOTS:

Slotname	**Type**	**Type name**
start_line	e	int
end_line	e	int

FRAME TYPE: digests *PARENT(S): []*

SLOTS:

Slotname	**Type**	**Type name**
digests	e	char

FRAME TYPE: content_vector *PARENT(S): []*

SLOTS:

Slotname	**Type**	**Type name**
concept_nbr	e	int
weight	e	int

CODER FRAME TYPE DEFINITION REPORT

FRAME TYPE: digest_issue *PARENT(S): []*

SLOTS:

Slotname	**Type**	**Type name**
issue	f	issue
topic	f	topic
digest_message	f	digest_message

FRAME TYPE: issue *PARENT(S): []*

SLOTS:

Slotname	**Type**	**Type name**
dig_id	e	int
isu_date	f	date
isu_num	e	int
isu_vol	e	int

FRAME TYPE: topic *PARENT(S): []*

SLOTS:

Slotname	**Type**	**Type name**
header1	e	char
header2	e	char

FRAME TYPE: forward_from *PARENT(S): []*

SLOTS:

Slotname	**Type**	**Type name**
board	e	char
individual	f	individual

FRAME TYPE: digest_message *PARENT(S): []*

SLOTS:

Slotname	**Type**	**Type name**
msg_id	e	int
date_sent	f	date_time
from	f	individual
subject	e	char
forward_from	f	forward_from
reply_to	f	individual
span	f	span
doc_type	e	char
content_frame	f	doctype
content_vector	f	content_vector

FRAME TYPE: doctype *PARENT(S): []*

** END OF FRAME DESCRIPTION REPORT **

Appendix H. Performance Evaluation Tables

Function	Module	Posts	Sequence	Time (Sec)
request welcome file display	ia	setup_screen	config start_up →ia →bboard	80
			→strategist →report →user_interface	1
user sees welcome+ (wait for user to continue) {state transition, welcome done+	ia	state	→ia →bboard	1}
state transition, id user	probmsd	id_user	→strategist →probmsd →bboard	1
request prompt for user id	user_model	disp_prompt	→strategist →user_model →bboard	2
			→strategist →report →user_interface	1
user sees prompt (wait for user entry) post user response	ia	um_resp	→ia →bboard	1
state transition, user unknown	user_model	state	→strategist →user_model →bboard	1
new state, characterize user	probmsd	char_newuser	→strategist →probmsd →bboard	2
request user info	user_model	disp_prompt	→strategist →user_model →bboard	1
			→strategist →report →user_interface	1

+ *Indicates that functions are performed concurrently.*

Function	Module	Posts	Sequence	Time (Sec)
user sees prompt (wait for user entry) post user response	ia	um_resp	→ia →bboard	1
get more user info	user_model	disp_menu	→strategist →user_model →bboard	1
			→strategist →report →user_interface	1
user sees menu (wait for user selection) post user selection	ia	um_resp etc.	→ia →bboard	1
(continue until all explicitly acquired user information is collected)				
state transition, questions_done	user_model	state	→bboard	1
request problem state info +	probmsd	disp_prompt	→strategist →probmsd →bboard	1
			→strategist →report →user_interface	1
{post user type +	user_model	utype	→bboard	1}
user sees prompt (wait for user entry) post user response	ia	prob_resp	→ia →bboard	1
state transition, document unknown	probmsd	state	→strategist →probmsd →bboard	1
request main menu for search, browse or tutorials	probmsd	disp_menu	→strategist →probmsd →bboard	1
			→strategist →report →user_interface	1
user sees menu (wait for user selection) post user selection for secondary menu, browse	ia	disp_menu	→ia →bboard	1

+ *Indicates that functions are performed concurrently.*

Function	Module	Posts	Sequence	Time (Sec)
			→strategist →report →user_interface	1
user sees menu (wait for user selection) post user selection for secondary menu, HAI	ia	disp_menu	→ia →bboard	1
			→strategist →report →user_interface	1
user sees menu (wait for user selection) post user selection for browse by subject	ia	disp_menu	→ia →bboard	1
			→strategist →report →user_interface	1
user sees menu (wait for user selection) request user prompt for browse by index reference	ia	disp_prompt	→ia →bboard	1
			→strategist →report →user_interface	1
user sees prompt (wait for user entry) request HAI lookup	ia	browse	→ia →bboard	1
post results of browse	hai	hai_output	→strategist →browse →bboard	4*
			→strategist →report →user_interface	2
user reviews browse results (wait to continue) previous menu displayed by user interface user selects return to main menu request display main menu	ia	disp_menu	→ia →bboard	1
			→strategist →report →user_interface	1

* *Depends on search terms for HAI browsing.*

Function	Module	Posts	Sequence	Time (Sec)
user sees menu (wait for user selection) begin searching	ia	prob_resp	→ia →bboard	1
get problem description info	probmsd	disp_menu	→strategist →probmsd →bboard	1
			→strategist →report →user_interface	1
user sees menu (wait for user selection)			→ia	
(continue until all problem description information is collected)		etc.		
post user response	ia	prob_resp	→bboard	1
state transition, doc search	probmsd	state	→strategist →probmsd →bboard	1
get system world info	probmsd	disp_menu	→bboard	1
			→strategist →report →user_interface	1
user sees menu (wait for user selection)			→ia	
(continue until all system world information is collected)		etc.		
post user response	ia	prob_resp	→bboard	1
request prompt, structured data?	probmsd	disp_prompt	→strategist →probmsd →bboard	1
			→strategist →report →user_interface	1
user sees prompt (wait for user selection) post user response, yes	ia	prob_resp	→ia →bboard	1
request structured data menu	probmsd	disp_menu	→strategist →probmsd →bboard	1

Function	Module	Posts	Sequence	Time (Sec)
			→strategist →report →user_interface	1
user sees menu (wait for user selection) post user frame selection	ia	disp_frame	→ia →bboard	1
			→strategist →report →user_interface	1
user prompted for slot value entry (wait for entry) request next slot	ia	nextslot	→ia →bboard	1
user prompted for slot value entry (wait for entry)			→strategist →report →user_interface →ia	1
(continue until all frame slots for requested frame are displayed)		etc.		
(then, report module initiates redisplay of structured data menu)				
user sees menu (wait for user selection) user is finished frame entries	ia	done_frames	→ia →bboard	1
other query terms? +	probmsd	disp_prompt	→strategist →probmsd →bboard	
{post all frames created by user+	ia	frame	→bboard	10}
{post relations for those frames+	ia	relation	→bboard	2}
			→strategist →report →user_interface	1
user sees prompt (wait for user selection) post user response, yes	ia	prob_resp	→ia →bboard	1
state transition, form query	probmsd	state	→strategist →probmsd →bboard	
initiate query formulation	probmsd	form_query	→bboard	3

+ *Indicates that functions are performed concurrently.*

Function	Module	Posts	Sequence	Time (Sec)
query formulator started	qform	disp_menu	→strategist →qform →bboard	2
user sees menu to choose query type (wait for user selection)				
post selection, boolean	ia	do_query	→ia →bboard	1
begin boolean query assistance	qform	displayf	→strategist →qform →bboard	1
		etc.		
(continue until all boolean query information is collected; help files will be displayed, and menus/prompts will guide user.)				
post query formed	qform	boolean	→bboard	3
state transition, search_ready	qform	state	→bboard	1
{inform user of processing+	qform	displayf	→bboard	1}
post retrieved documents+	search	doc	→strategist →search →bboard	5*
state transition, docs posted	search	state	→bboard	1
request display of results	probmsd	results	→strategist →probmsd →bboard	1
send results file (simulated)	browse	displayf	→strategist →browse →bboard	2
			→strategist →report →user_interface	1
user browses results (wait for user to finish)				
state transition, done results	browse	state	→ia →bboard	2

+ *Indicates that functions are performed concurrently.*

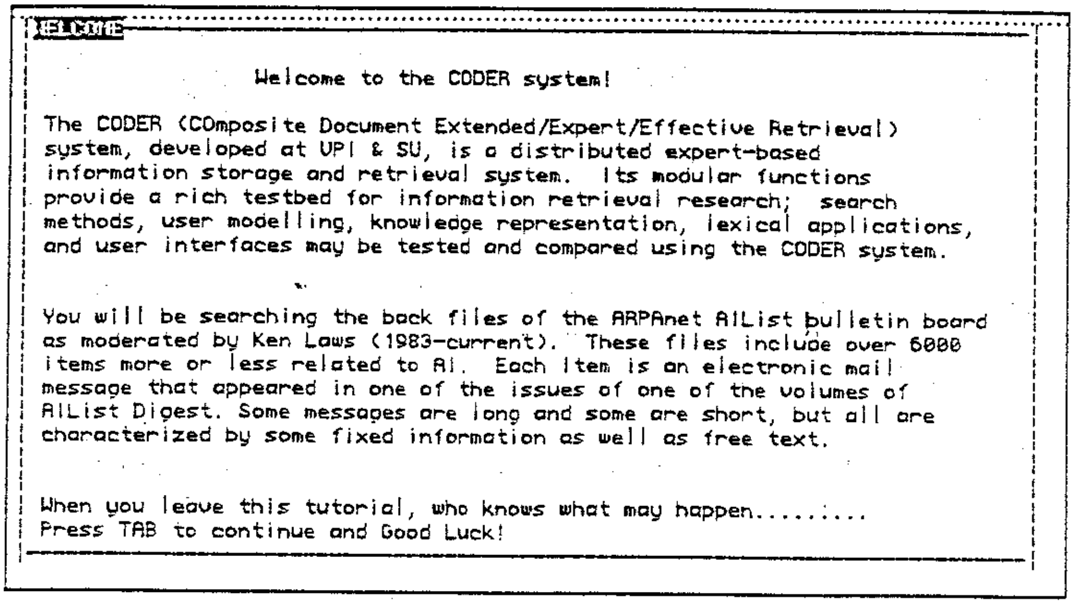
* *Based on 2500 record test set of Prolog facts. No frame matching. A* doc *hypothesis will be posted for each relevant document.*

Function	Module	Posts	Sequence	Time (Sec)
more queries or done?	probmsd	disp_prompt	→strategist →probmsd →bboard	3
			→strategist →report →user_interface	1
user sees prompt (wait for user entry) post user response (done)	ia	prob_resp	→ia →bboard	1
state transition, user done	probmsd	state	→strategist →probmsd →bboard	2
initiate user evaluation	probmsd	user_eval	→bboard	1
{inform user+	probmsd	disp_msg	→bboard	1}
collect user evaluation data+	user_model	disp_prompt etc.	→strategist →user_model →bboard	2
(continue until all user evaluation data is collected)				
state transition, done evaluation	user_model	state	→bboard	2
state transition, clean up	probmsd	state	→strategist →probmsd →bboard	2
initiate clean up	probmsd	clean_up	→bboard	1
{display final thank you+ +	probmsd	displayf	→bboard	1}
state transition, cleanup done+ +	user_model	state	→strategist →user_model →bboard	34
request termination	probmsd	kill_procs	→strategist →probmsd →bboard	1
			→strategist →report →user_interface	1

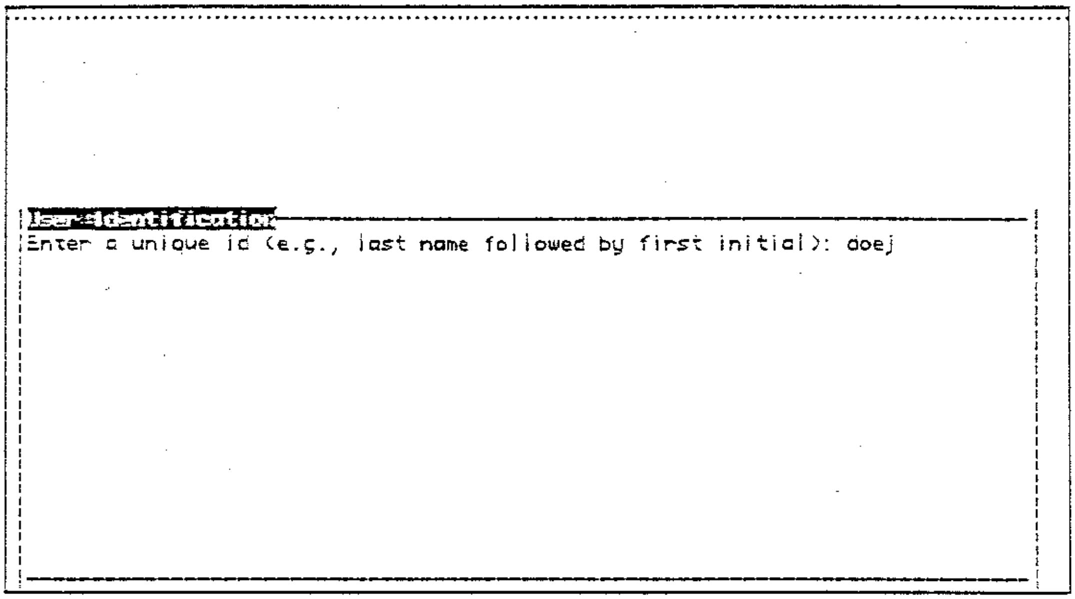
(user_interface issues stop_service commands to all modules, then exits its own processing to return control to config start up shell. Temporary files are deleted and processing terminates.)

+ *Indicates that functions are performed concurrently.*

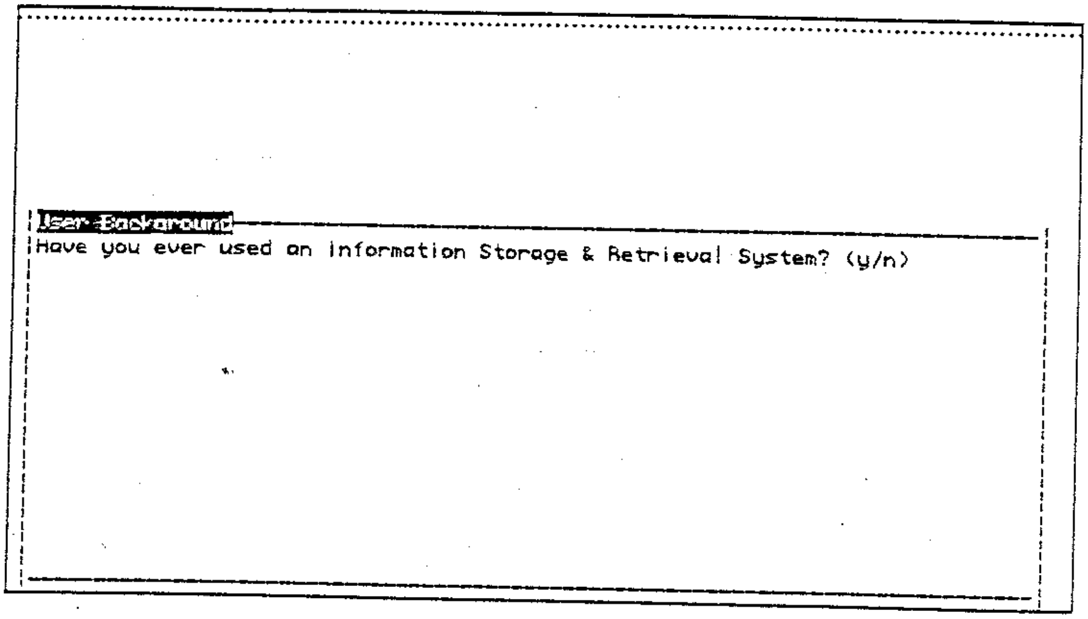
Appendix I. Sample CODER Retrieval Session



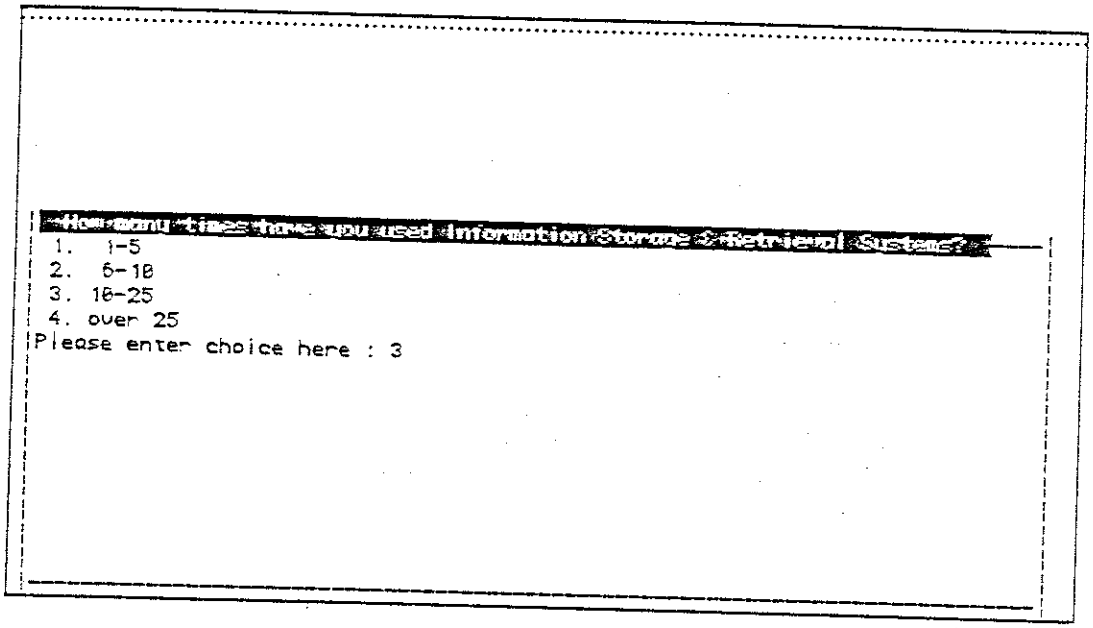
```
                    Welcome to the CODER system!

The CODER (COmposite Document Extended/Expert/Effective Retrieval)
system, developed at VPI & SU, is a distributed expert-based
information storage and retrieval system.  Its modular functions
provide a rich testbed for information retrieval research;  search
methods, user modelling, knowledge representation, lexical applications,
and user interfaces may be tested and compared using the CODER system.


You will be searching the back files of the ARPAnet AIList bulletin board
as moderated by Ken Laws (1983-current).  These files include over 6000
items more or less related to AI.  Each item is an electronic mail
message that appeared in one of the issues of one of the volumes of
AIList Digest. Some messages are long and some are short, but all are
characterized by some fixed information as well as free text.


When you leave this tutorial, who knows what may happen.........
Press TAB to continue and Good Luck!
```



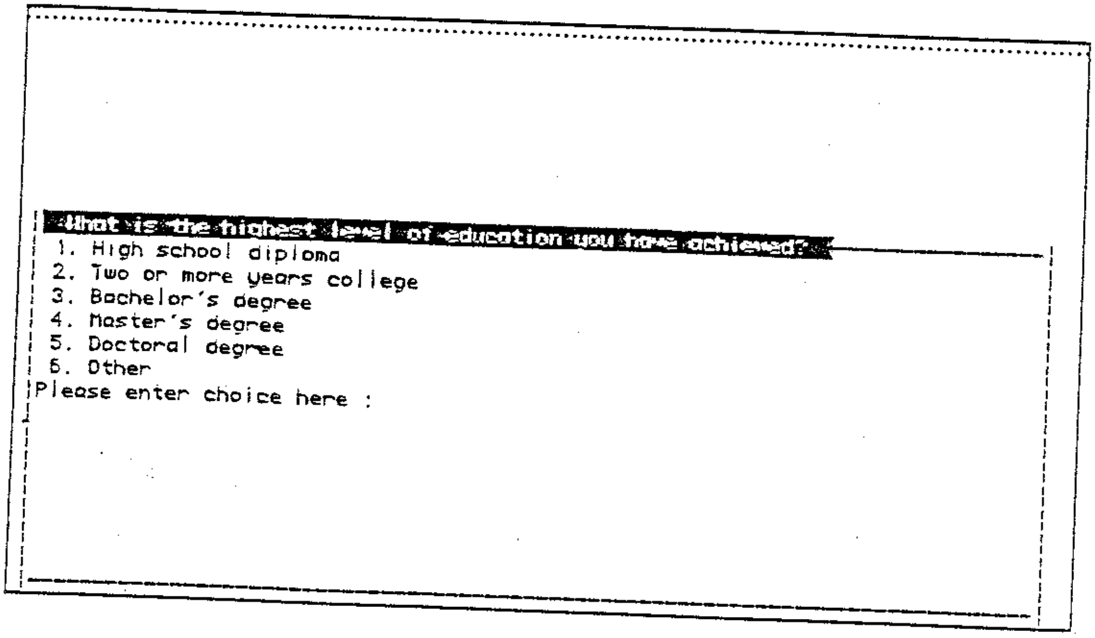
```
Enter a unique id (e.g., last name followed by first initial): doej
```



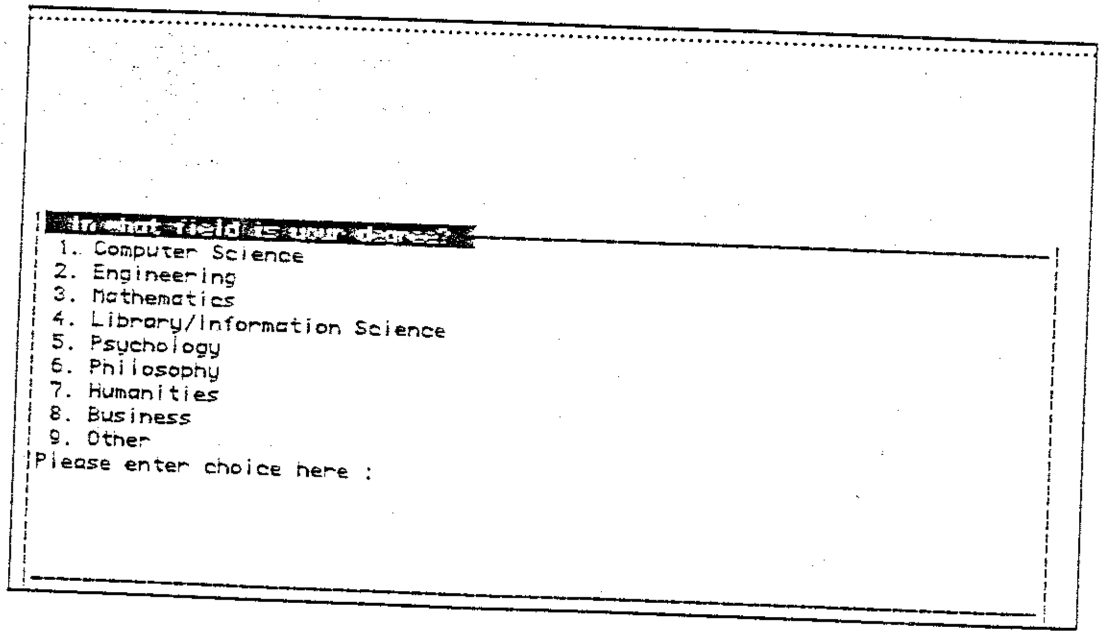
```
User Background
Have you ever used an Information Storage & Retrieval System? (y/n)
```



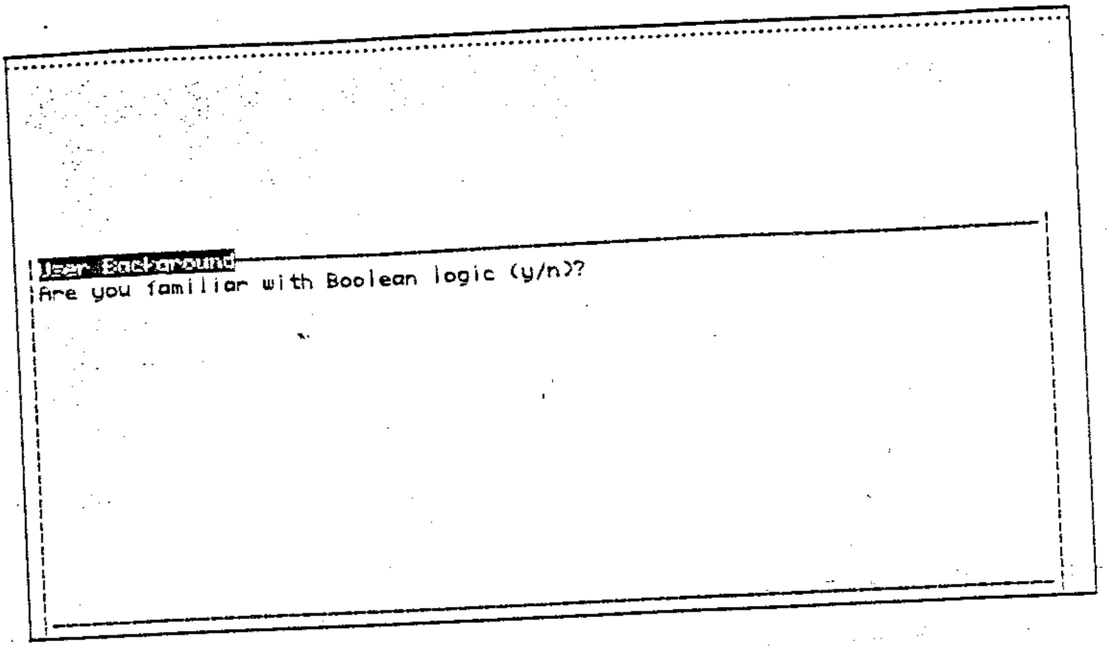
```
How many times have you used Information Storage & Retrieval System?
 1.   1-5
 2.   6-10
 3.  10-25
 4.  over 25
Please enter choice here : 3
```



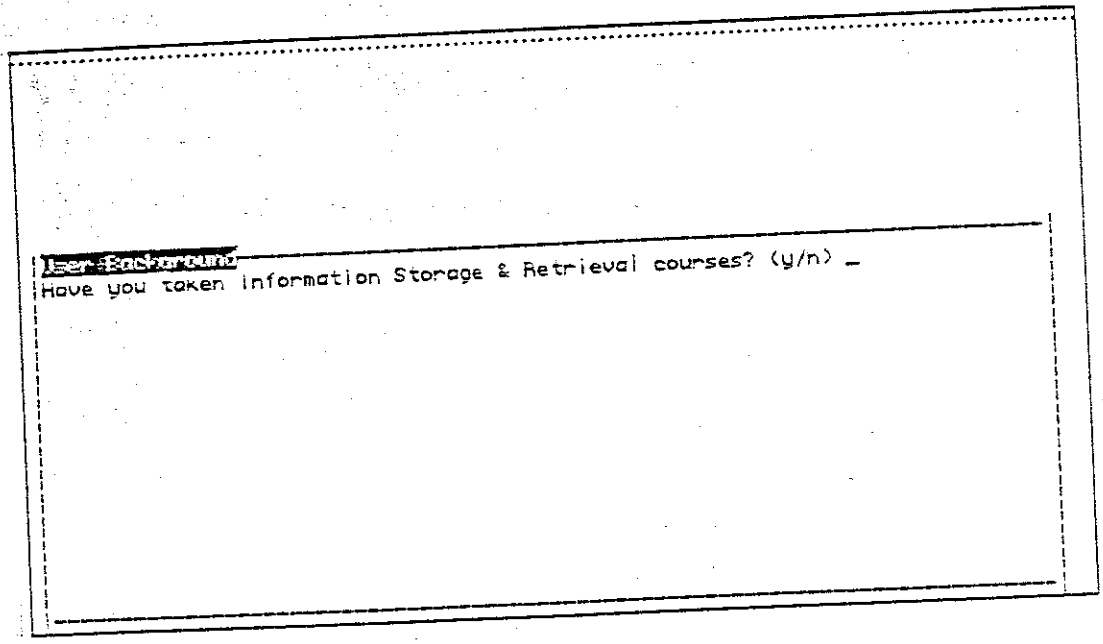
```
What is the highest level of education you have achieved?
 1. High school diploma
 2. Two or more years college
 3. Bachelor's degree
 4. Master's degree
 5. Doctoral degree
 6. Other
Please enter choice here :
```



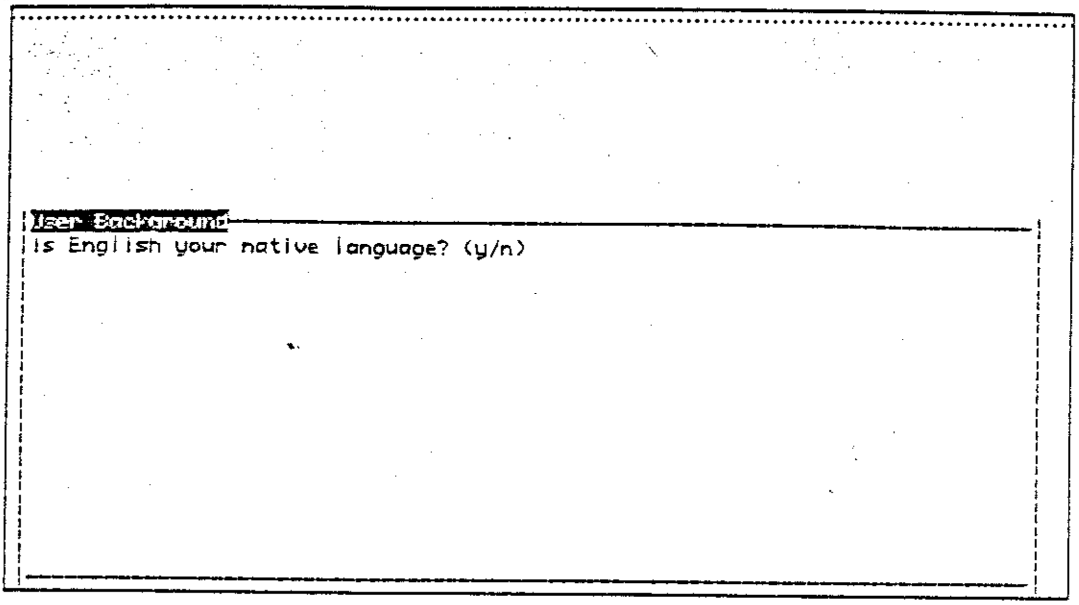
```
In what field is your degree?
 1. Computer Science
 2. Engineering
 3. Mathematics
 4. Library/Information Science
 5. Psychology
 6. Philosophy
 7. Humanities
 8. Business
 9. Other
Please enter choice here :
```



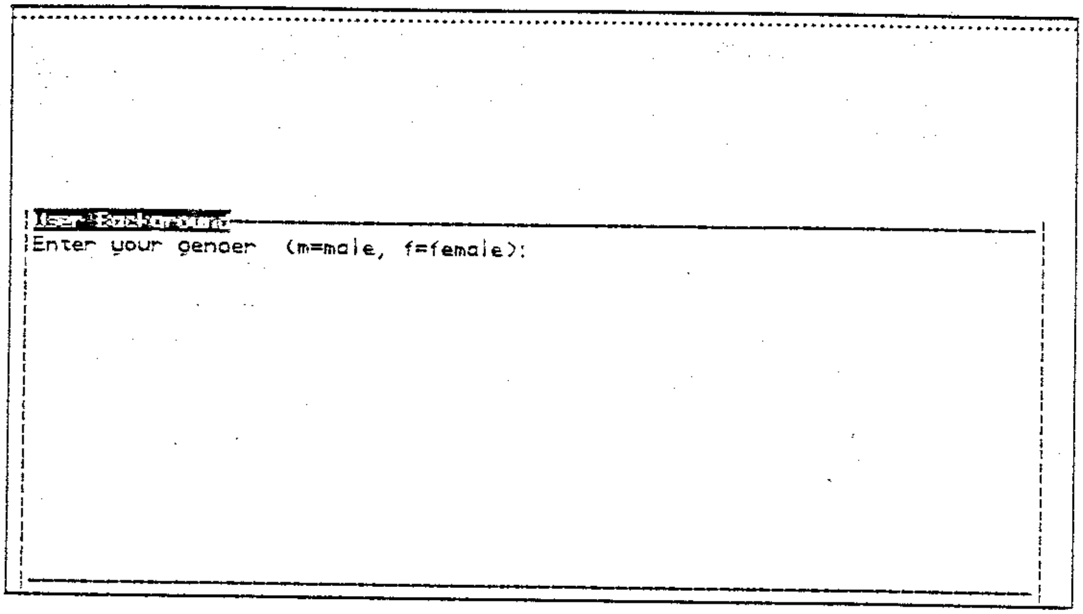
```
User Background
Are you familiar with Boolean logic (y/n)?
```



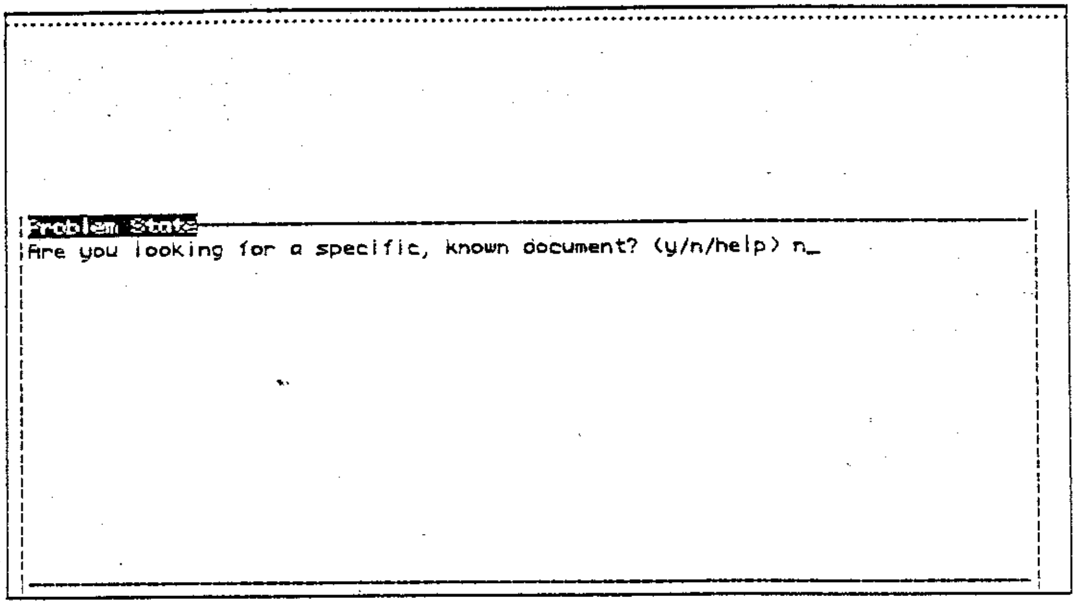
```
User Background
Have you taken Information Storage & Retrieval courses? (y/n) _
```



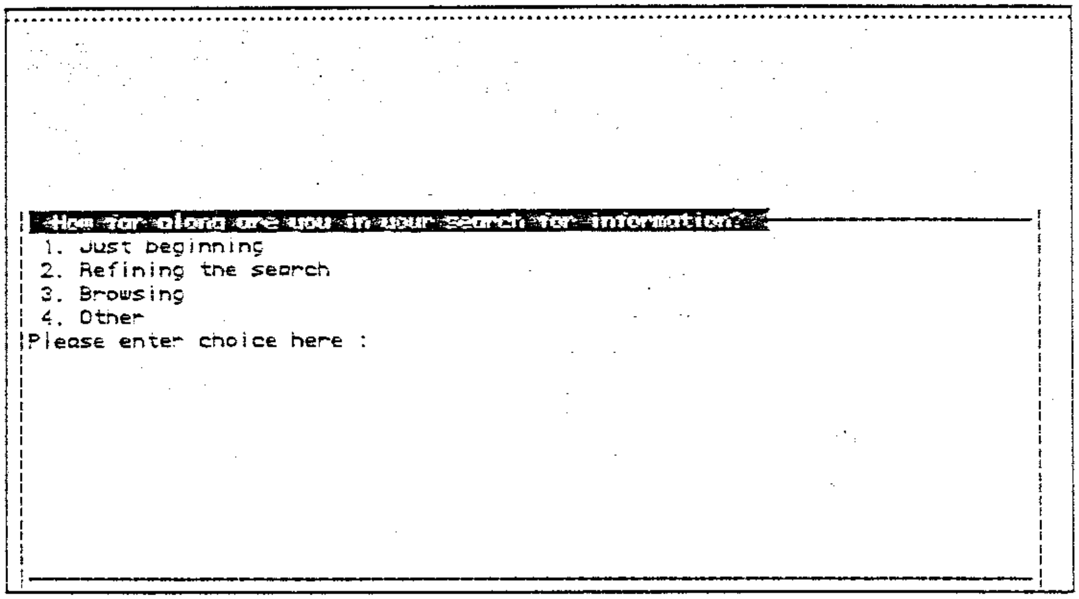
```
User Background
Is English your native language? (y/n)
```



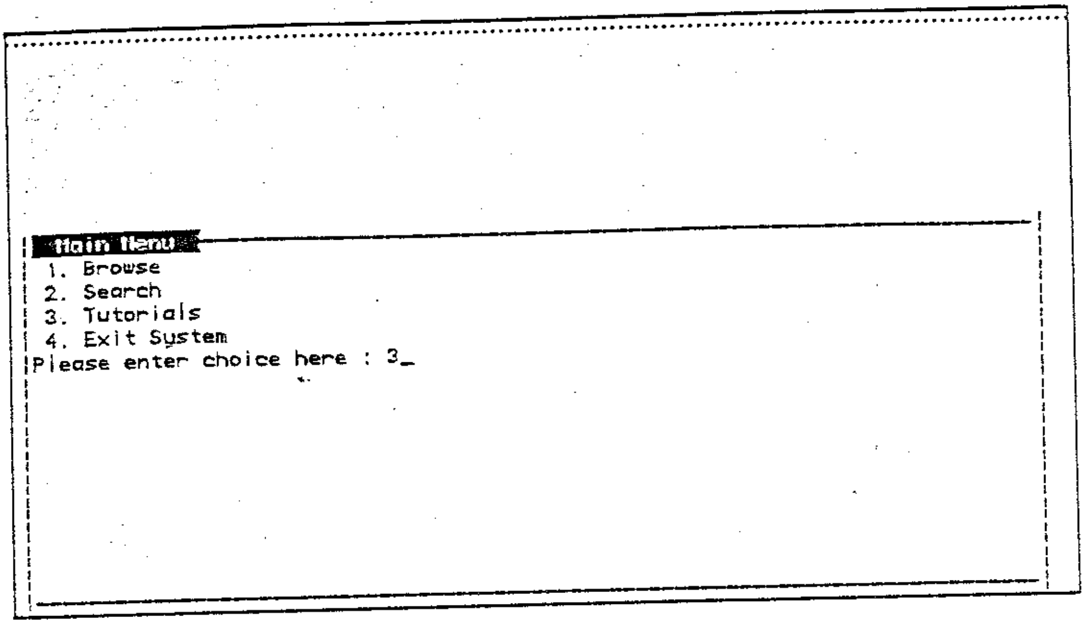
```
User Background
Enter your gender  (m=male, f=female):
```



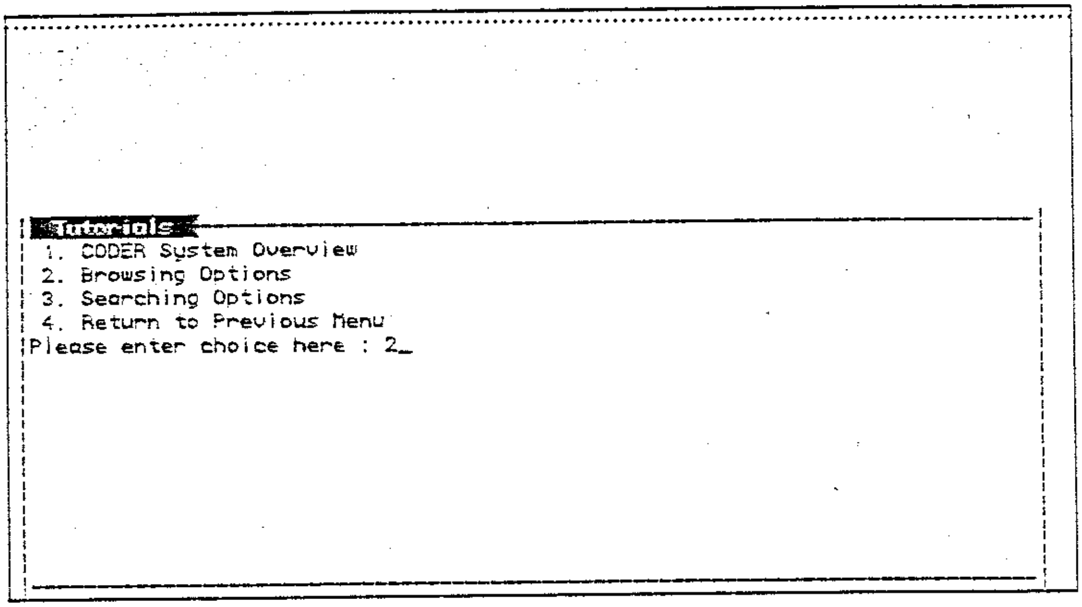
```
Problem State
Are you looking for a specific, known document? (y/n/help) n_
```



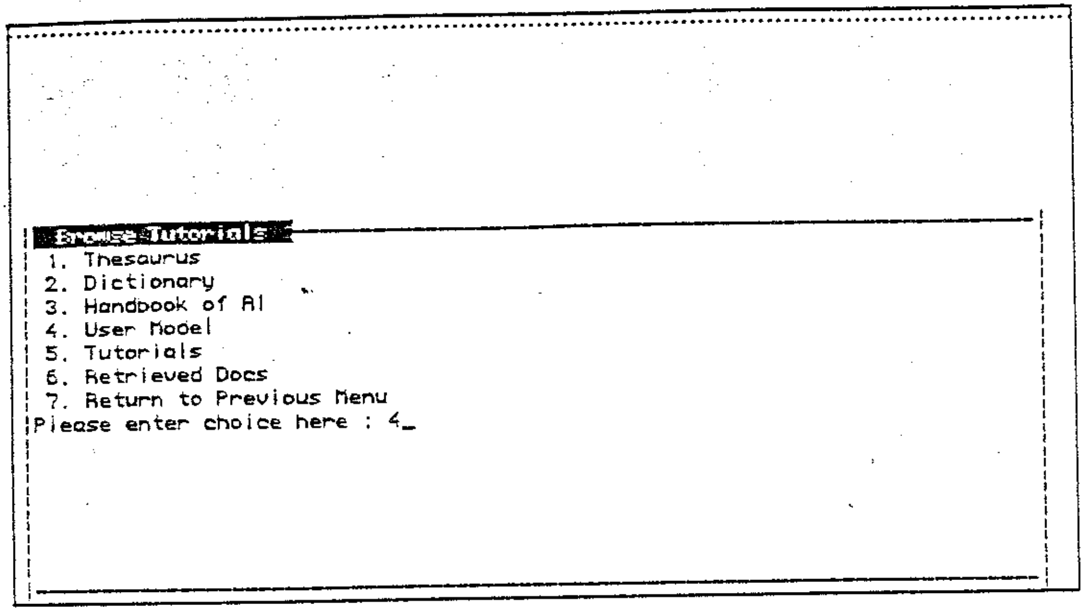
```
How far along are you in your search for information?
 1. Just beginning
 2. Refining the search
 3. Browsing
 4. Other
Please enter choice here :
```



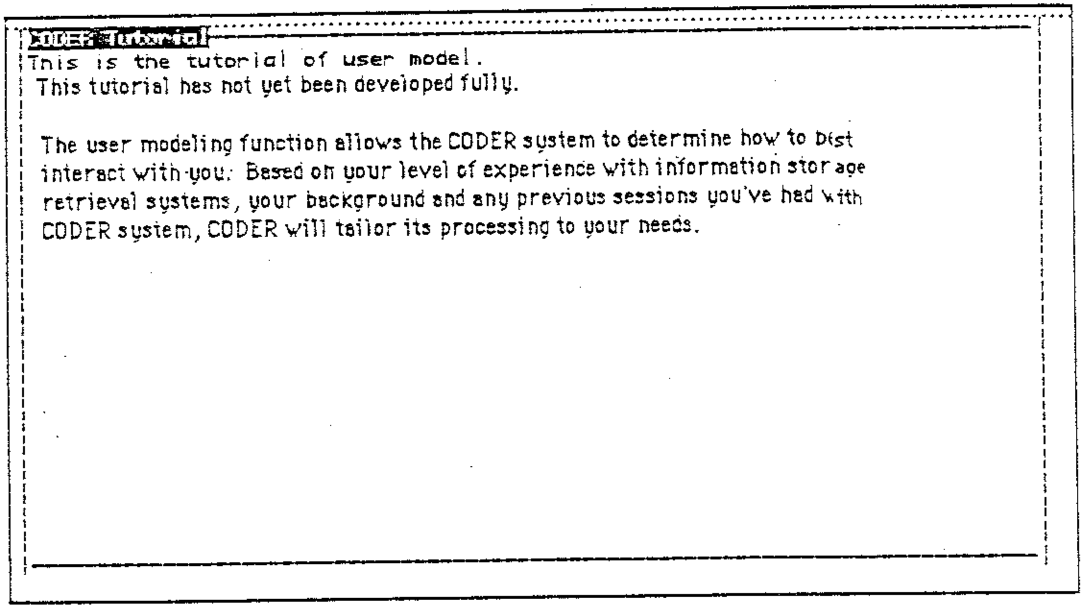
```
 Main Menu
 1. Browse
 2. Search
 3. Tutorials
 4. Exit System
Please enter choice here : 3_
```



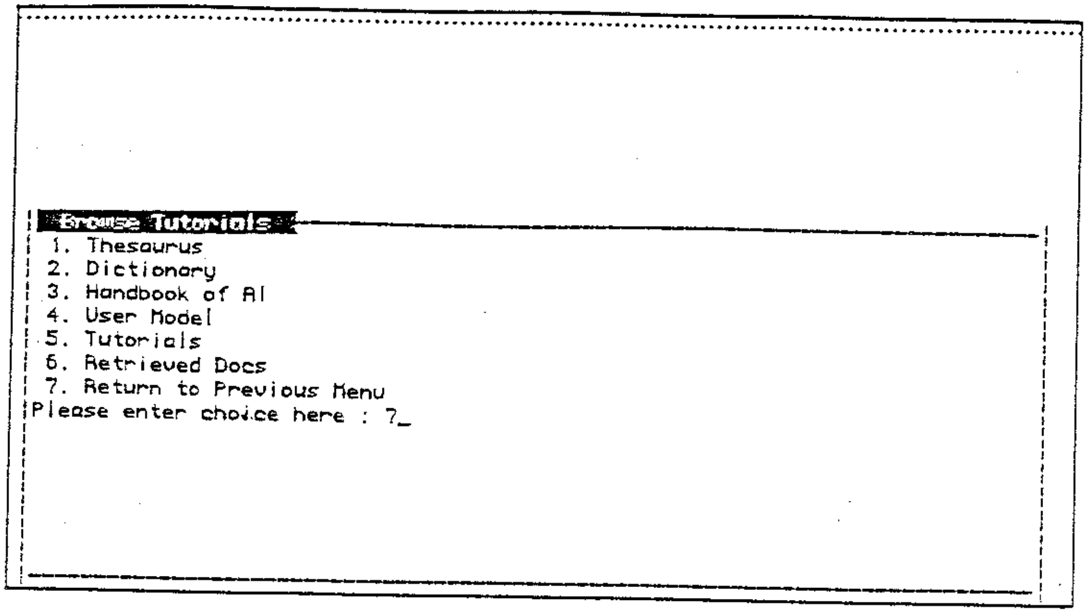
```
 Tutorials
 1. CODER System Overview
 2. Browsing Options
 3. Searching Options
 4. Return to Previous Menu
Please enter choice here : 2_
```



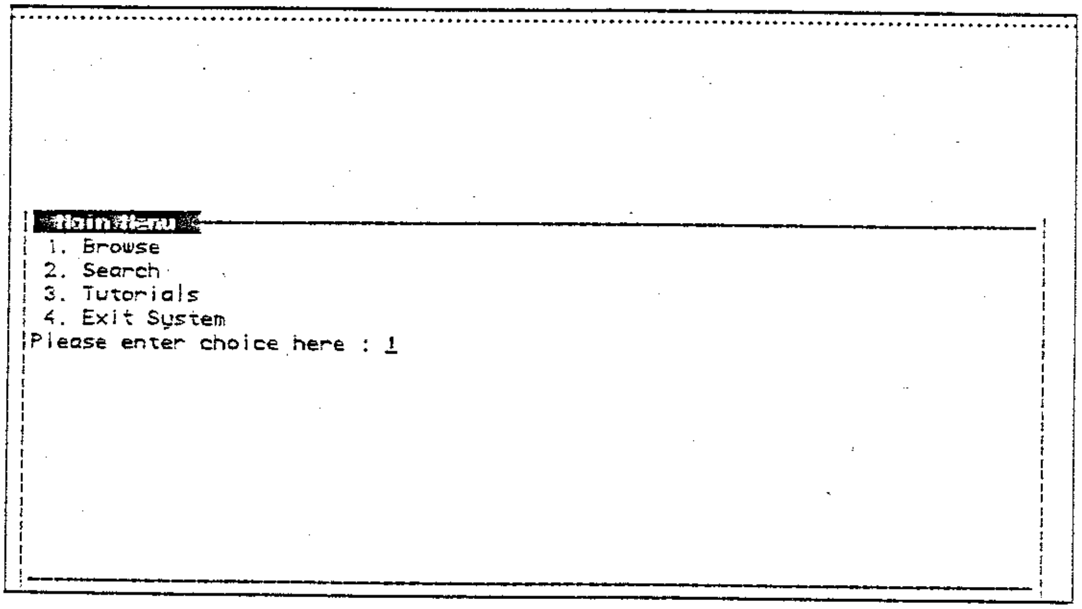
```
Browse Tutorials
 1. Thesaurus
 2. Dictionary
 3. Handbook of AI
 4. User Model
 5. Tutorials
 6. Retrieved Docs
 7. Return to Previous Menu
Please enter choice here : 4_
```



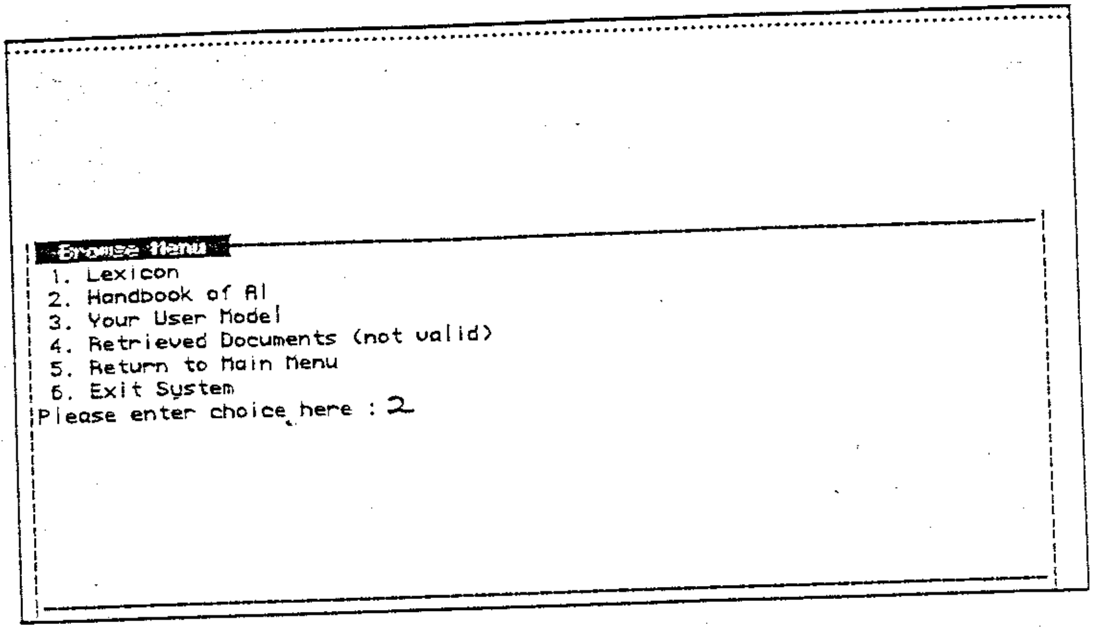
```
CODER Tutorial
This is the tutorial of user model.
This tutorial has not yet been developed fully.

The user modeling function allows the CODER system to determine how to best
interact with you. Based on your level of experience with information storage
retrieval systems, your background and any previous sessions you've had with
CODER system, CODER will tailor its processing to your needs.
```



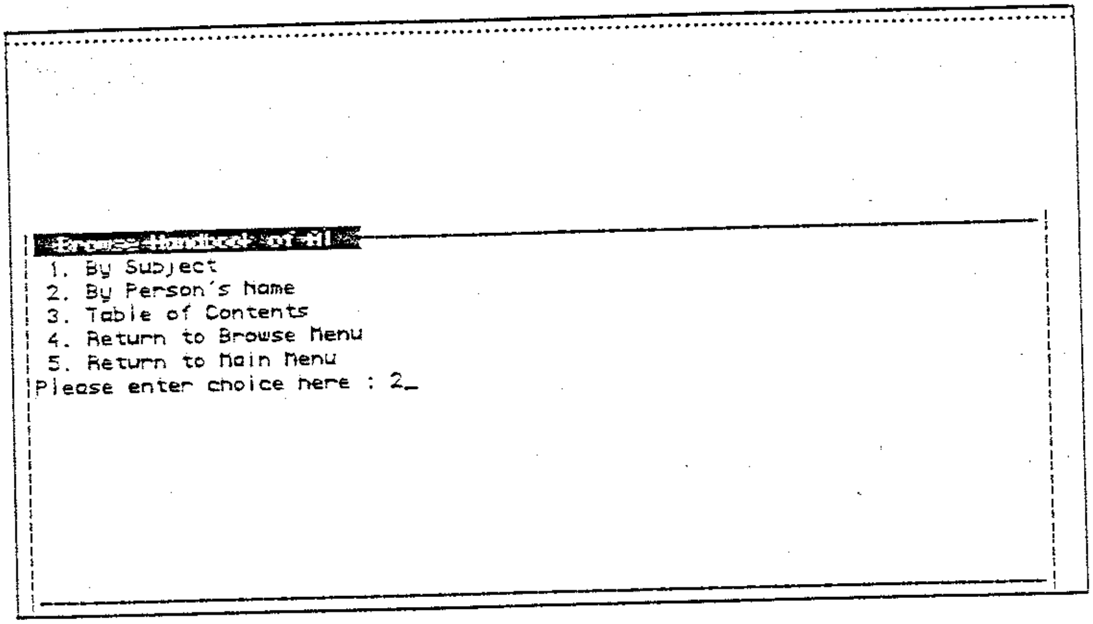
```
Browse Tutorials
 1. Thesaurus
 2. Dictionary
 3. Handbook of AI
 4. User Model
 5. Tutorials
 6. Retrieved Docs
 7. Return to Previous Menu
Please enter choice here : 7_
```



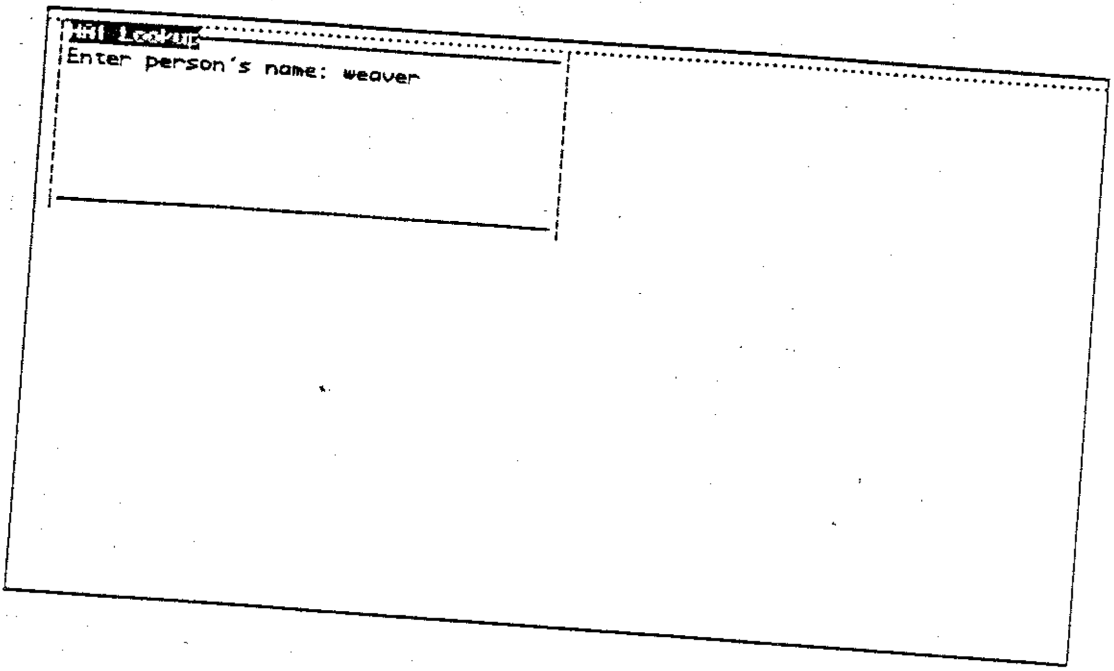
```
Main Menu
 1. Browse
 2. Search
 3. Tutorials
 4. Exit System
Please enter choice here : 1
```



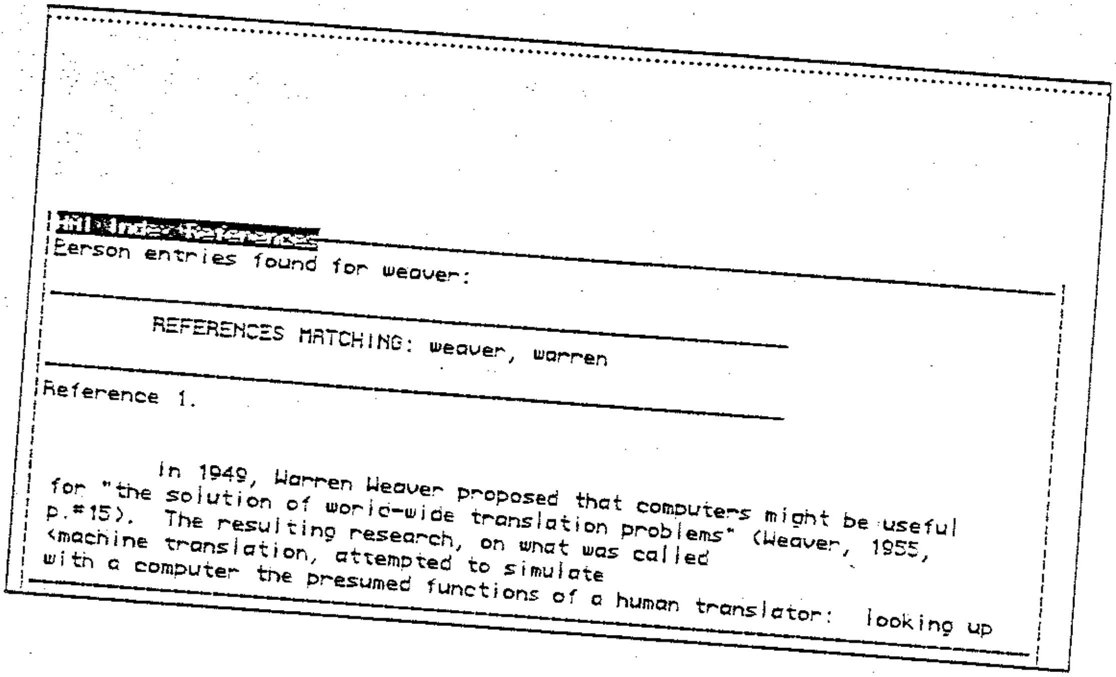
```
Browse Menu
 1. Lexicon
 2. Handbook of AI
 3. Your User Model
 4. Retrieved Documents (not valid)
 5. Return to Main Menu
 6. Exit System
Please enter choice here : 2
```



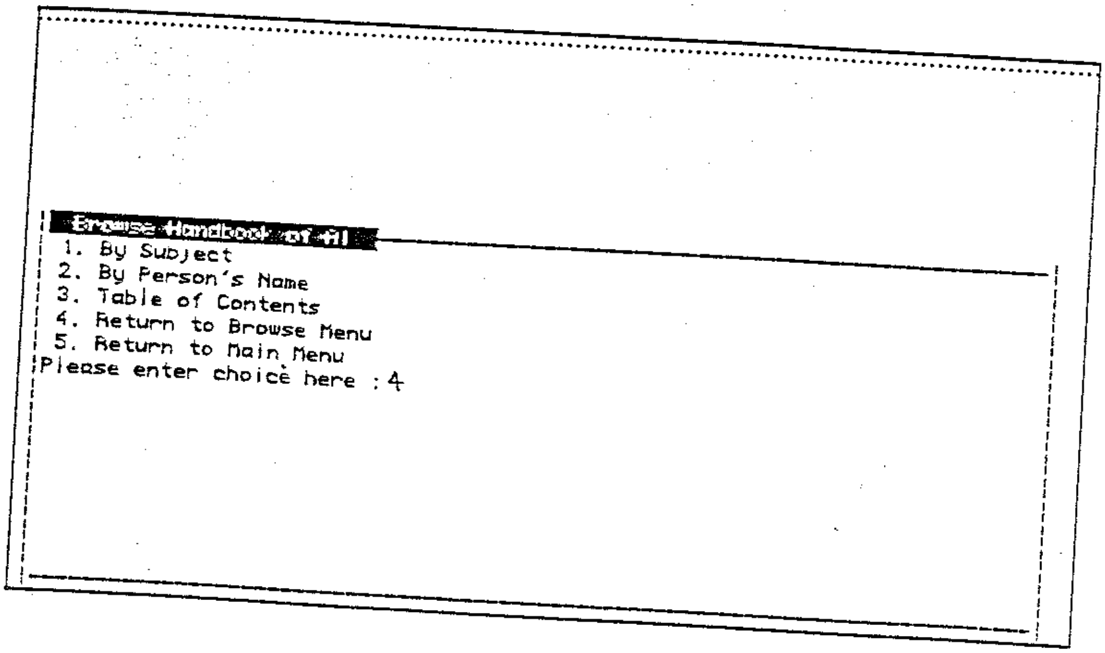
```
Browse Handbook of AI
 1. By Subject
 2. By Person's Name
 3. Table of Contents
 4. Return to Browse Menu
 5. Return to Main Menu
Please enter choice here : 2_
```



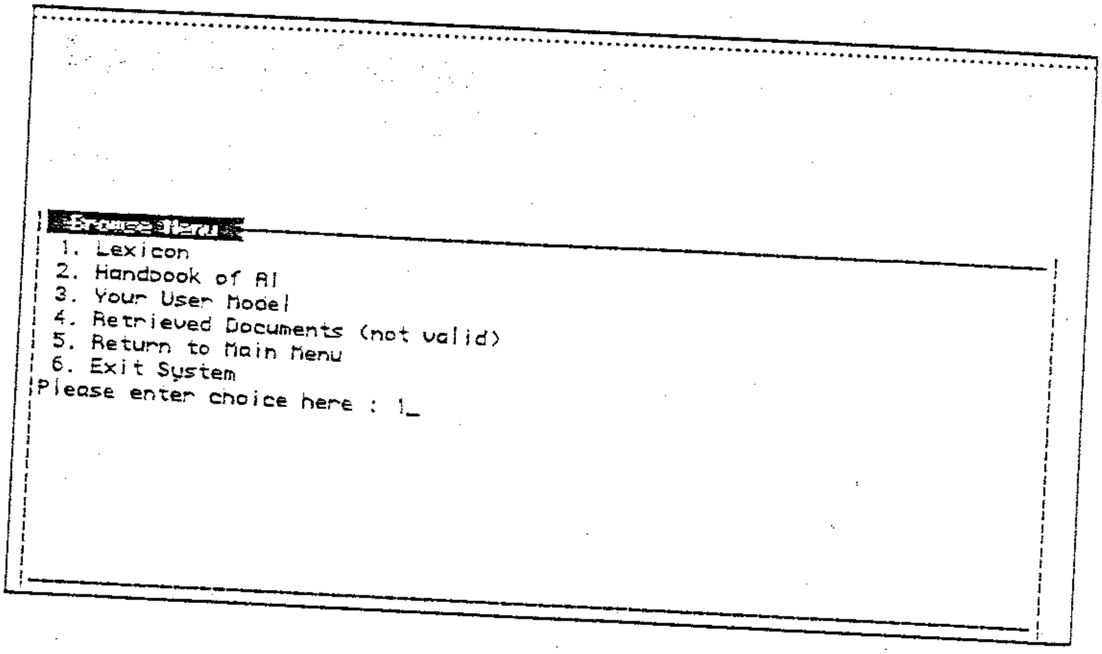
```
Enter person's name: weaver
```



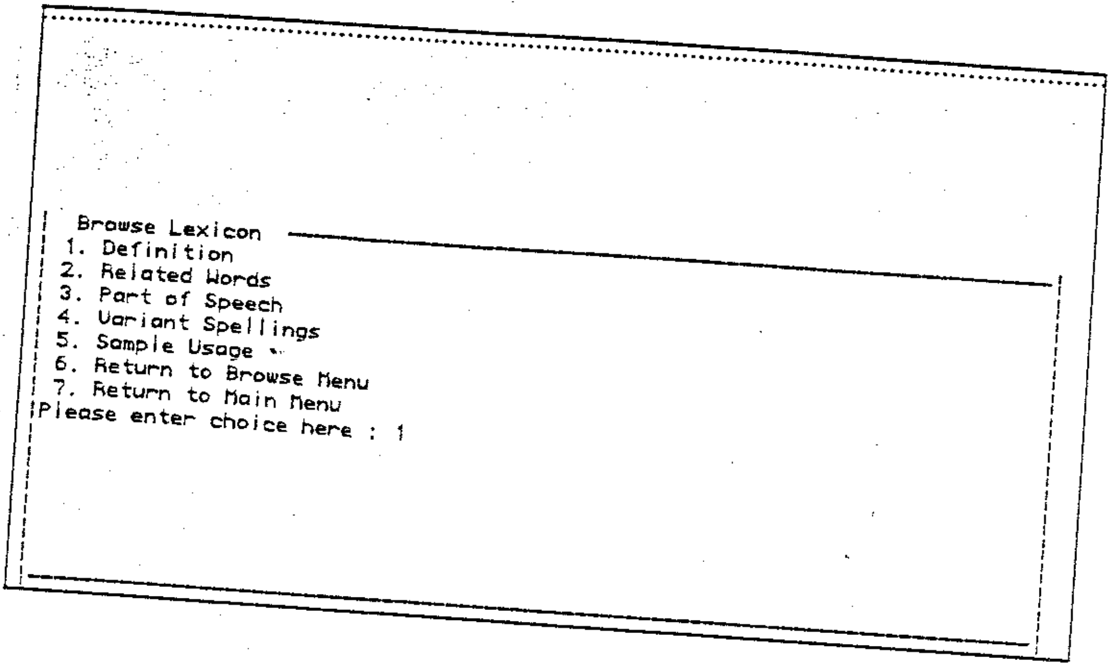
```
Person entries found for weaver:

        REFERENCES MATCHING: weaver, warren

Reference 1.


          In 1949, Warren Weaver proposed that computers might be useful
for "the solution of world-wide translation problems" (Weaver, 1955,
p.#15).  The resulting research, on what was called
<machine translation, attempted to simulate
with a computer the presumed functions of a human translator:  looking up
```



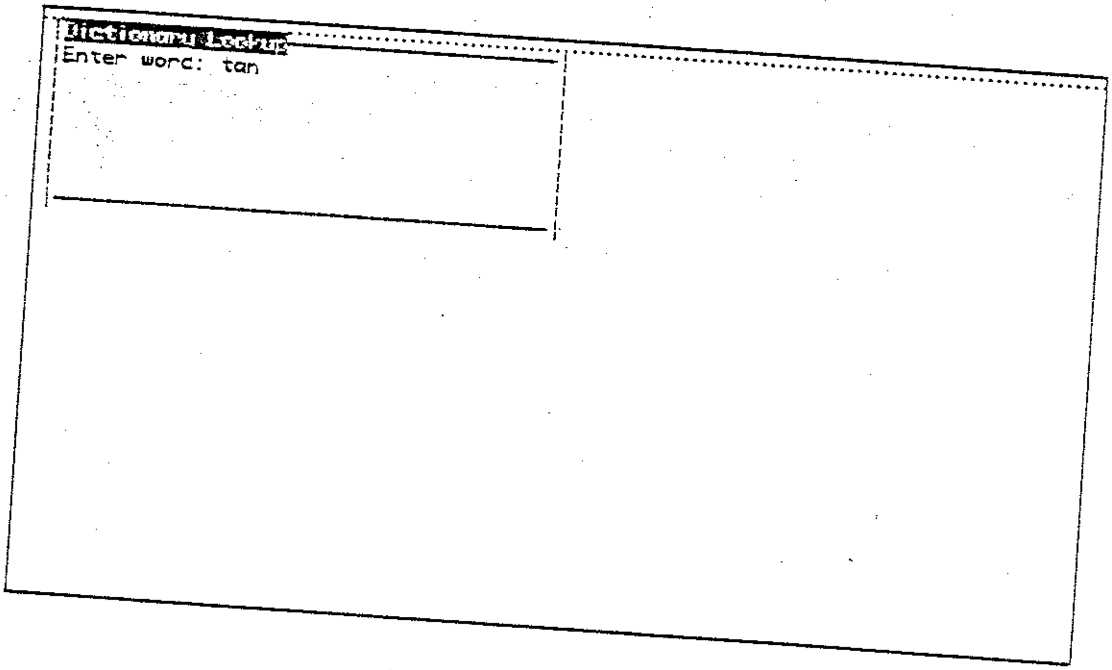
```
Browse Handbook of AI
 1. By Subject
 2. By Person's Name
 3. Table of Contents
 4. Return to Browse Menu
 5. Return to Main Menu
Please enter choice here :4
```



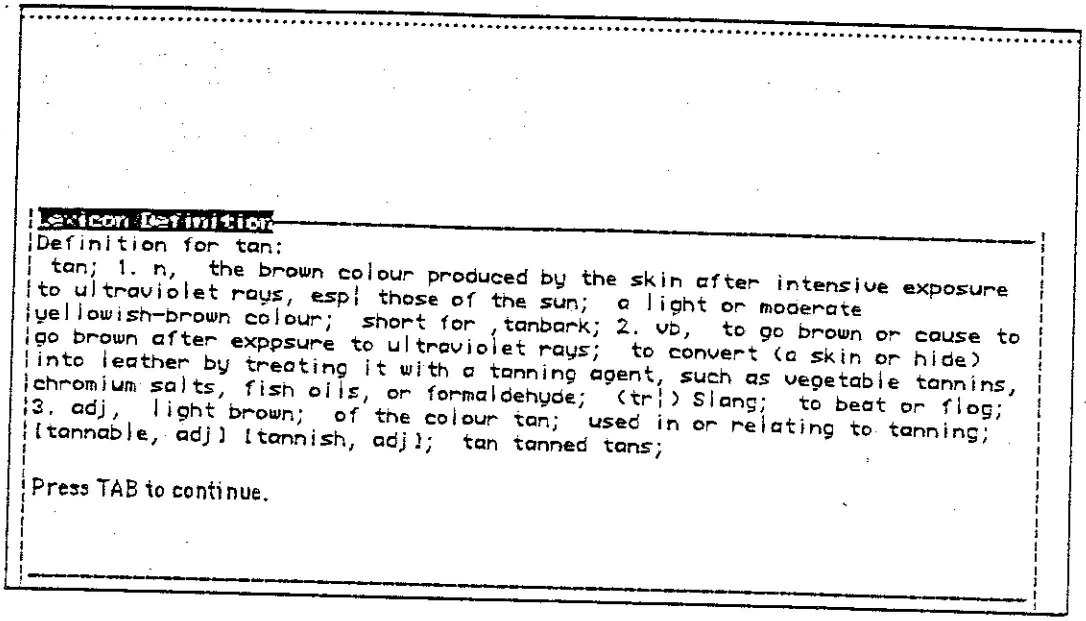
```
Browse Menu
 1. Lexicon
 2. Handbook of AI
 3. Your User Model
 4. Retrieved Documents (not valid)
 5. Return to Main Menu
 6. Exit System
Please enter choice here : 1_
```



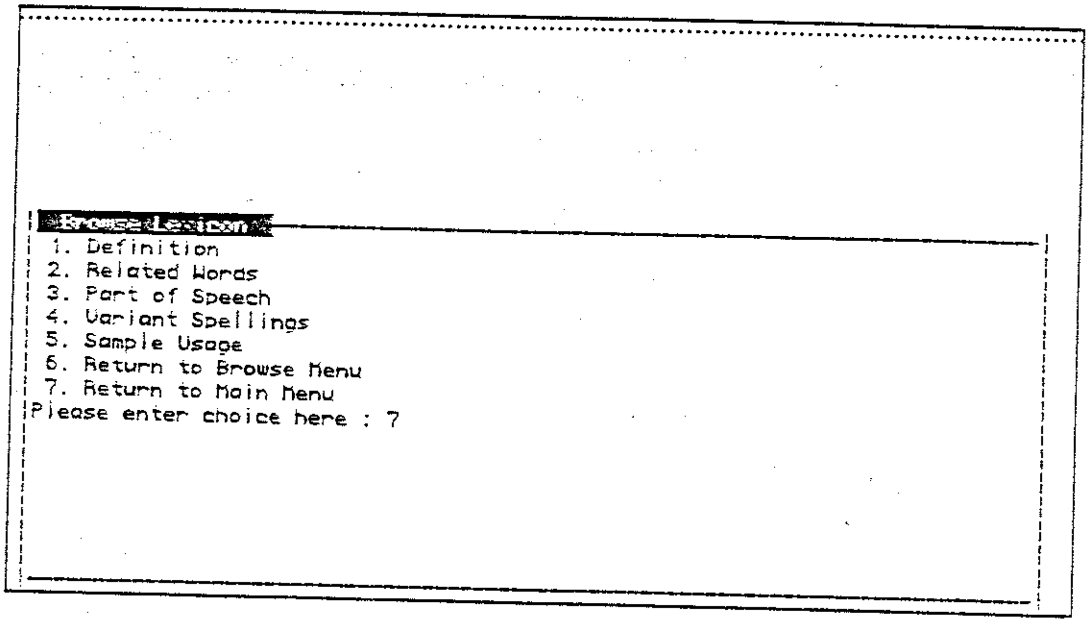
```
Browse Lexicon
 1. Definition
 2. Related Words
 3. Part of Speech
 4. Variant Spellings
 5. Sample Usage
 6. Return to Browse Menu
 7. Return to Main Menu
Please enter choice here : 1
```



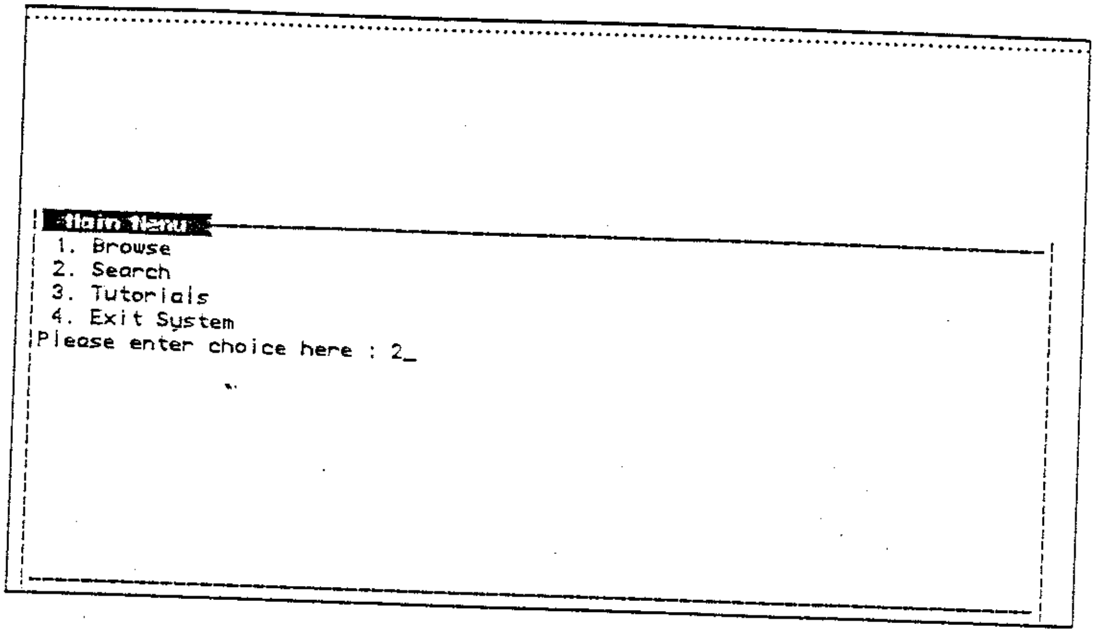
```
Dictionary Lookup
Enter worc: tan
```



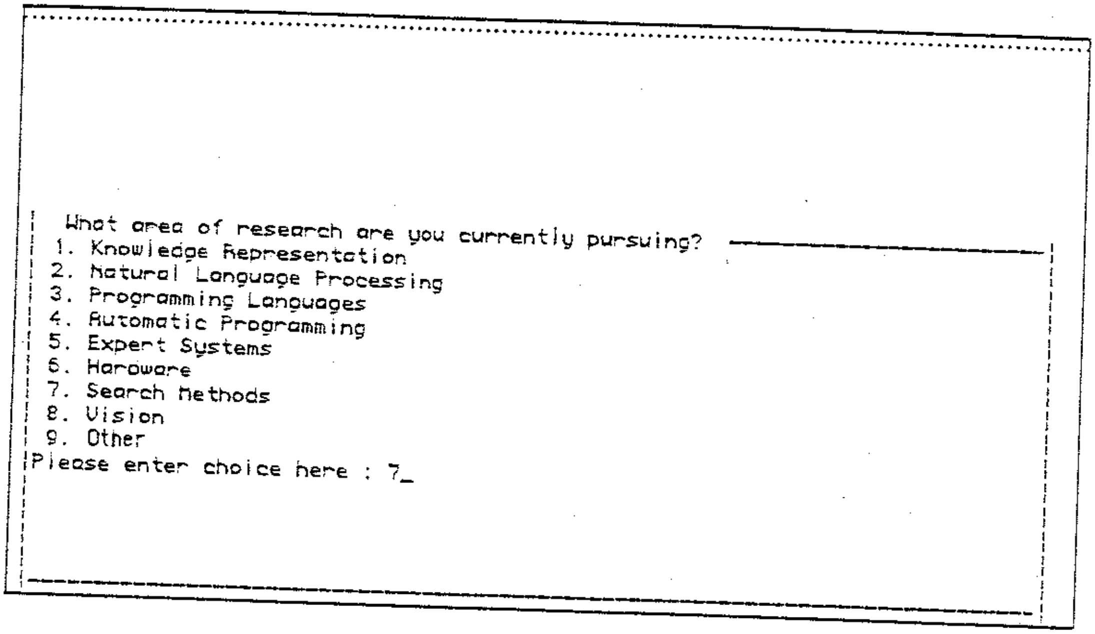
```
Lexicon Definition
Definition for tan:
 tan; 1. n,  the brown colour produced by the skin after intensive exposure
to ultraviolet rays, esp| those of the sun;  a light or moderate
yellowish-brown colour;  short for ,tanbark; 2. vb,  to go brown or cause to
go brown after exppsure to ultraviolet rays;  to convert (a skin or hide)
into leather by treating it with a tanning agent, such as vegetable tannins,
chromium salts, fish oils, or formaldehyde;  (tr|) Slang;  to beat or flog;
3. adj,  light brown;  of the colour tan;  used in or relating to tanning;
[tannable, adj] [tannish, adj];  tan tanned tans;

Press TAB to continue.
```



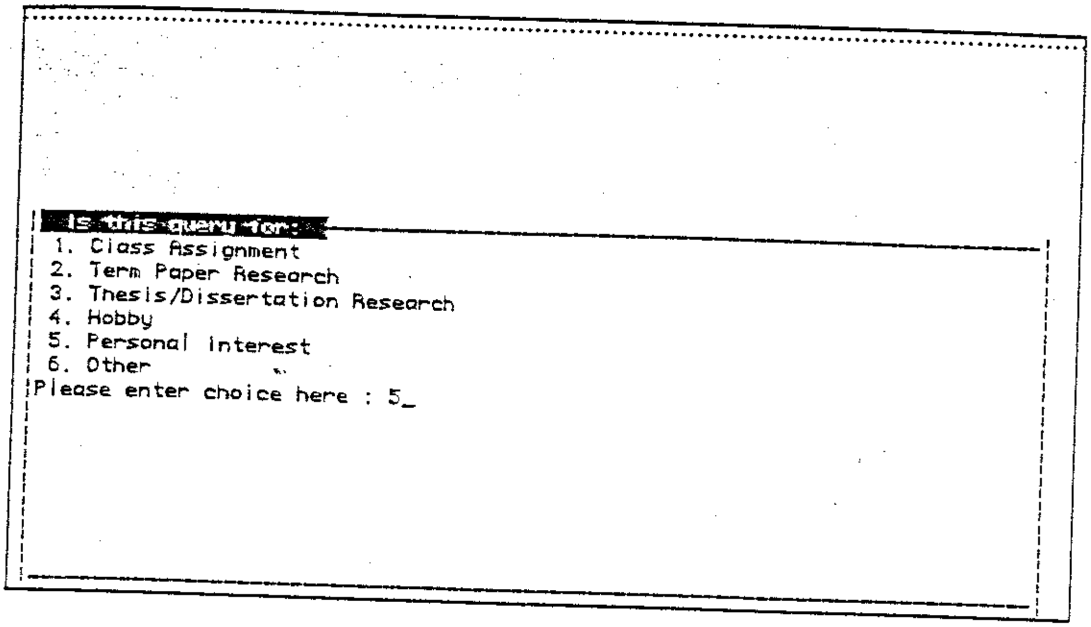
```
Browse Lexicon
 1. Definition
 2. Related Words
 3. Part of Speech
 4. Variant Spellings
 5. Sample Usage
 6. Return to Browse Menu
 7. Return to Main Menu
Please enter choice here : 7
```



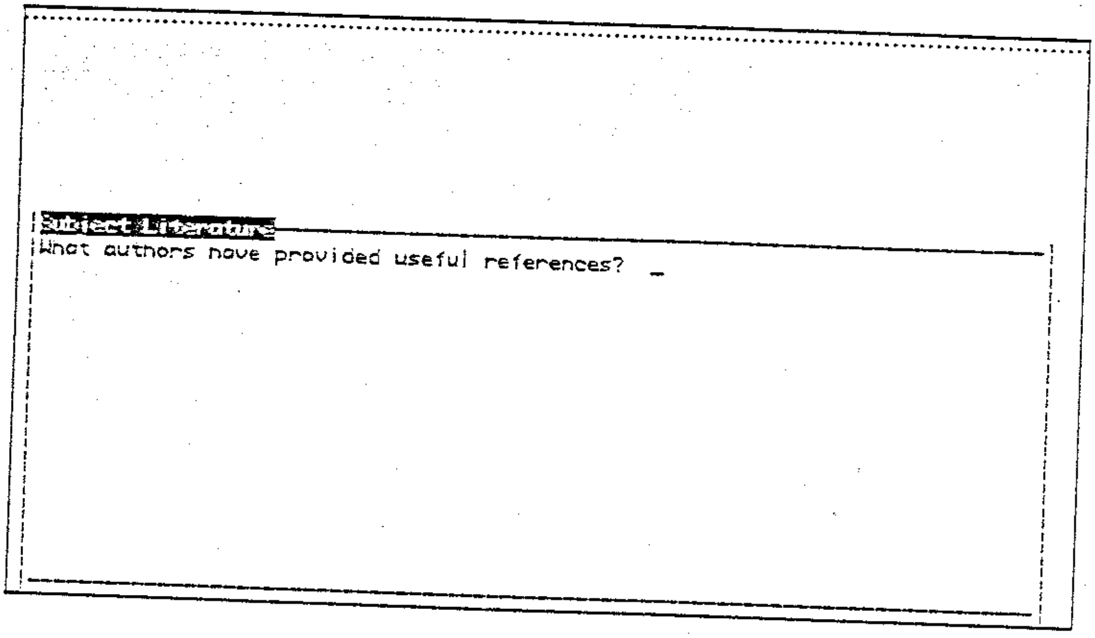
```
Main Menu
 1. Browse
 2. Search
 3. Tutorials
 4. Exit System
Please enter choice here : 2_
```



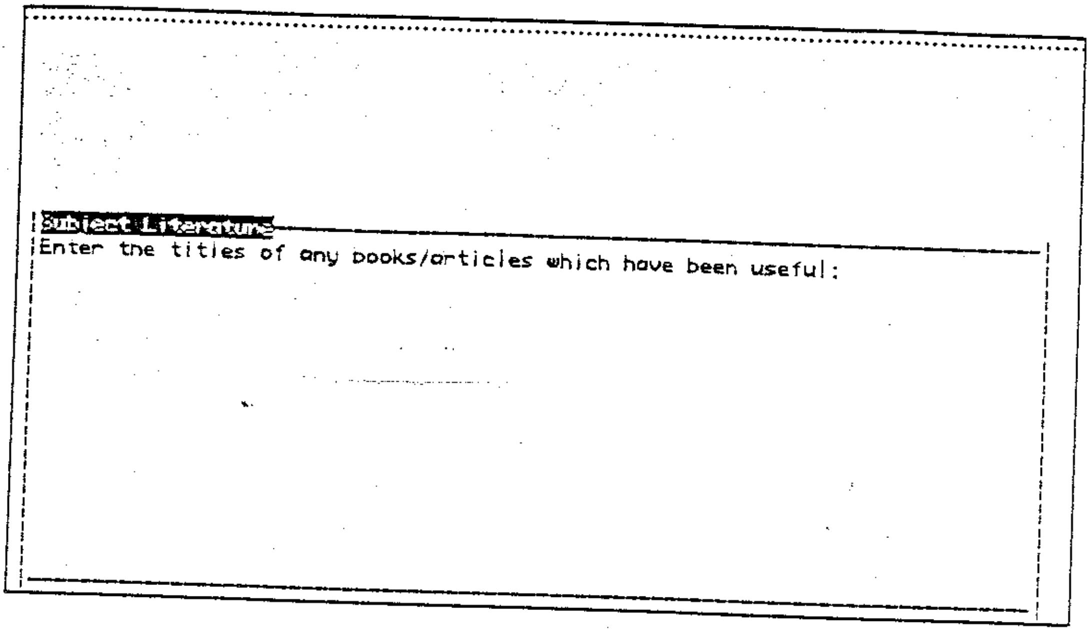
```
 What area of research are you currently pursuing?
 1. Knowledge Representation
 2. Natural Language Processing
 3. Programming Languages
 4. Automatic Programming
 5. Expert Systems
 6. Hardware
 7. Search Methods
 8. Vision
 9. Other
Please enter choice here : 7_
```



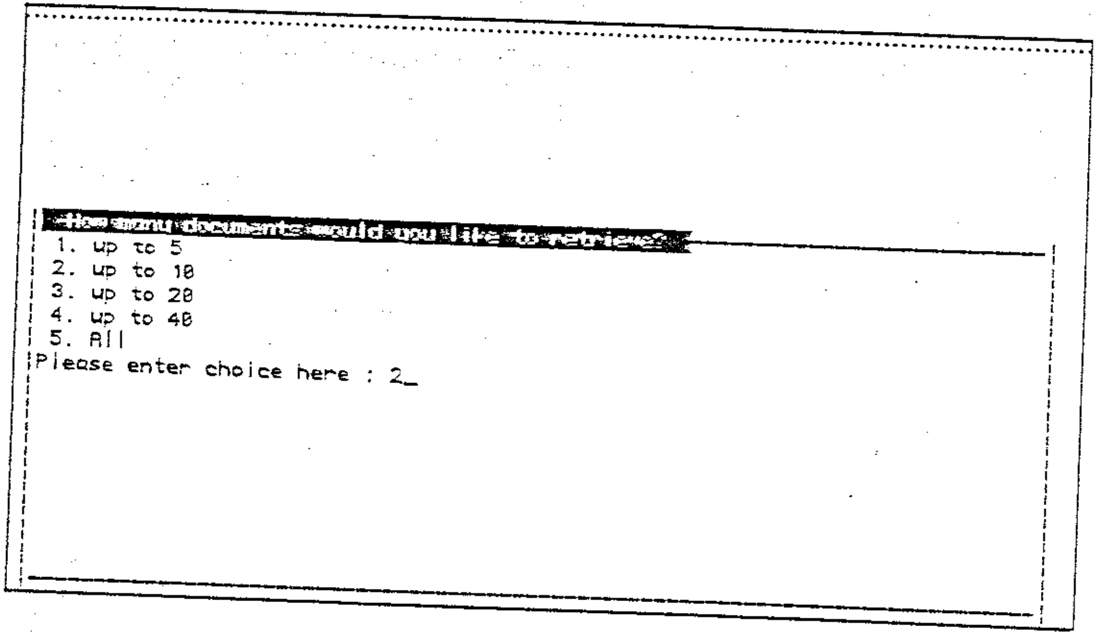
```
Is this query for:
 1. Class Assignment
 2. Term Paper Research
 3. Thesis/Dissertation Research
 4. Hobby
 5. Personal Interest
 6. Other
Please enter choice here : 5_
```



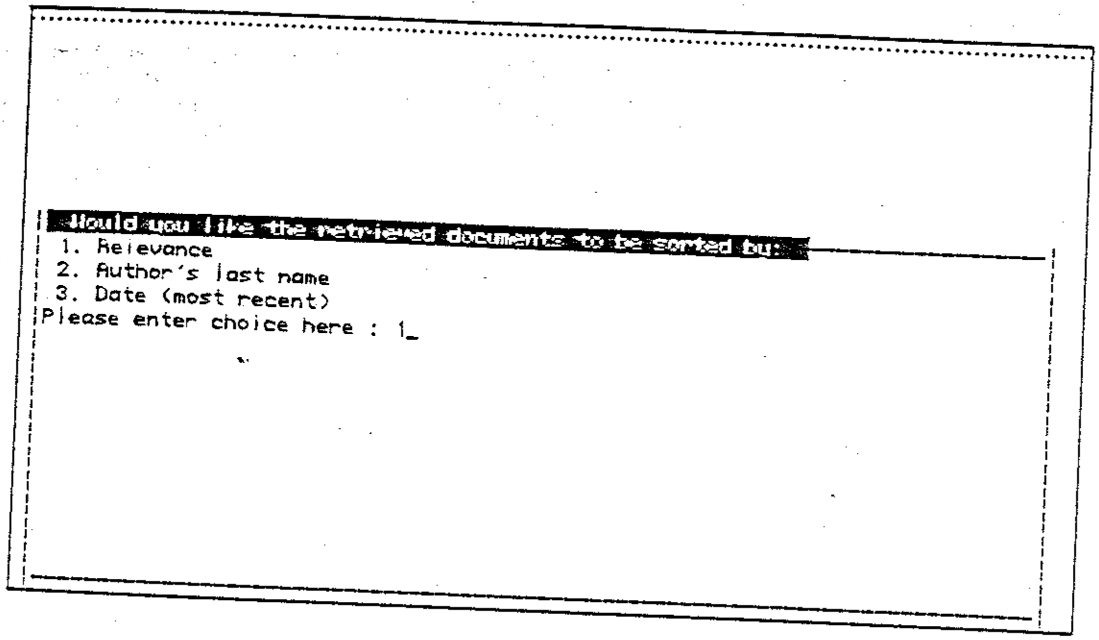
```
Subject Literature
What authors have provided useful references?  _
```



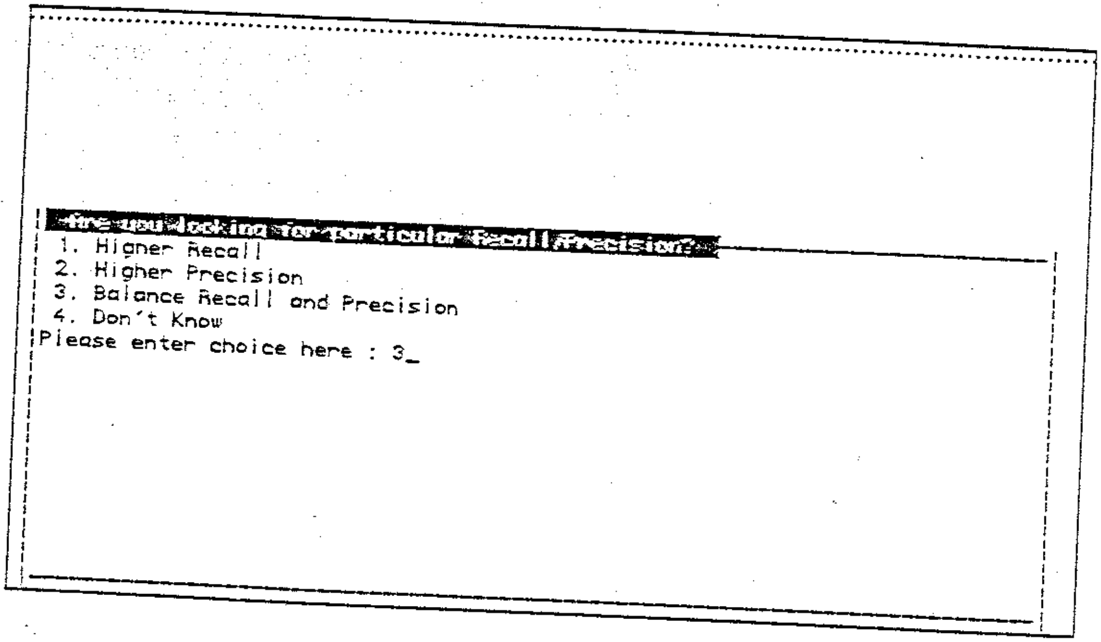
```
Subject Literature
Enter the titles of any books/articles which have been useful:
```



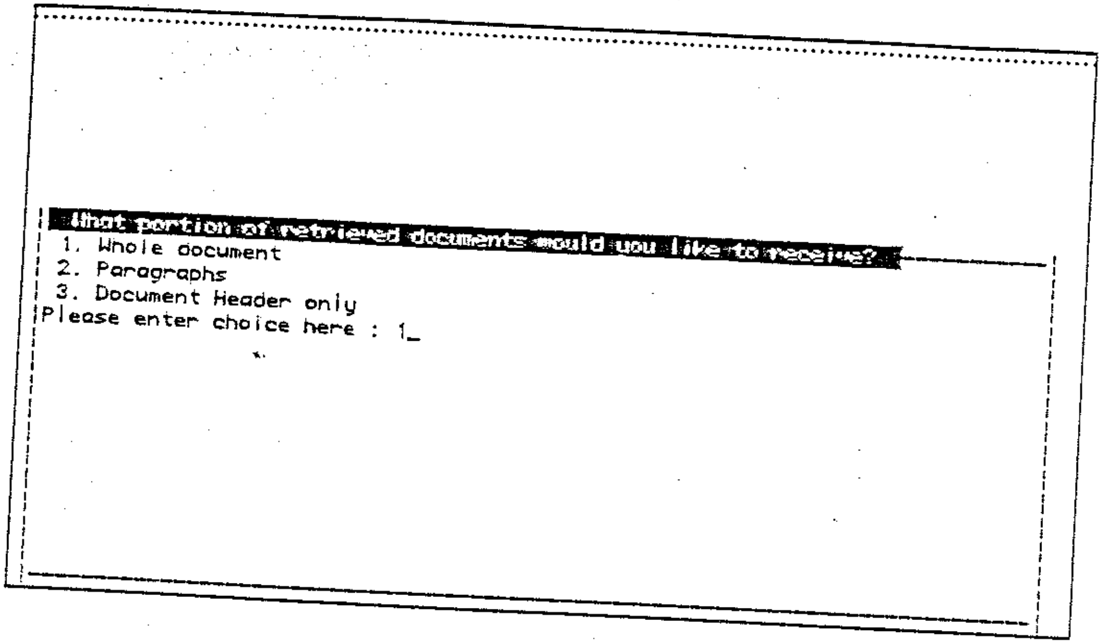
```
How many documents would you like to retrieve?
 1. up to 5
 2. up to 10
 3. up to 20
 4. up to 40
 5. All
Please enter choice here : 2_
```



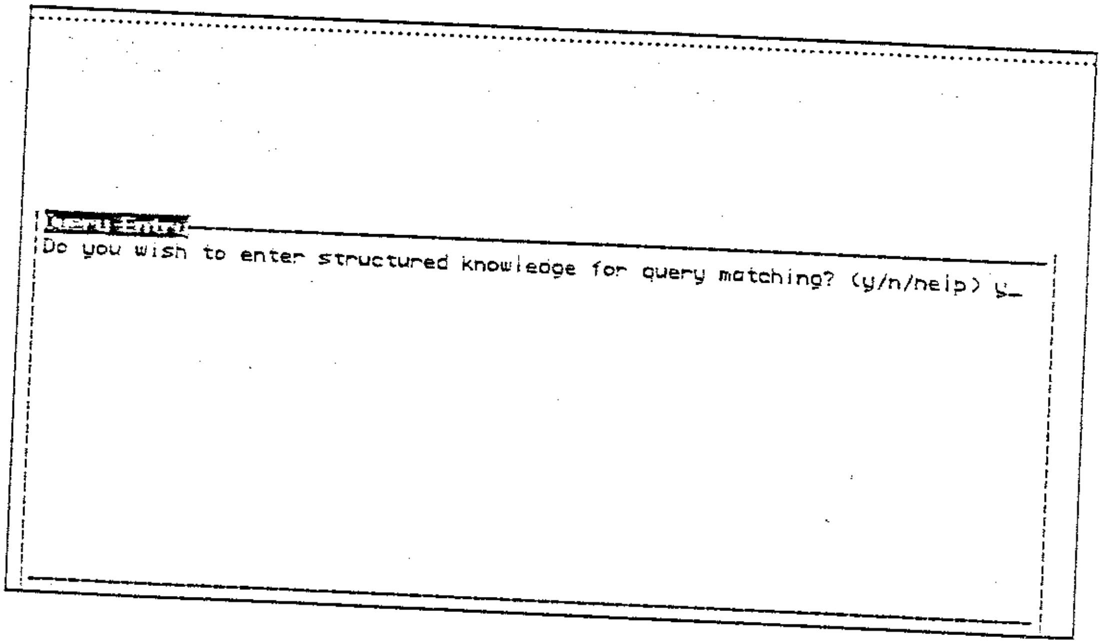
```
Would you like the retrieved documents to be sorted by:
 1. Relevance
 2. Author's last name
 3. Date (most recent)
Please enter choice here : 1_
```



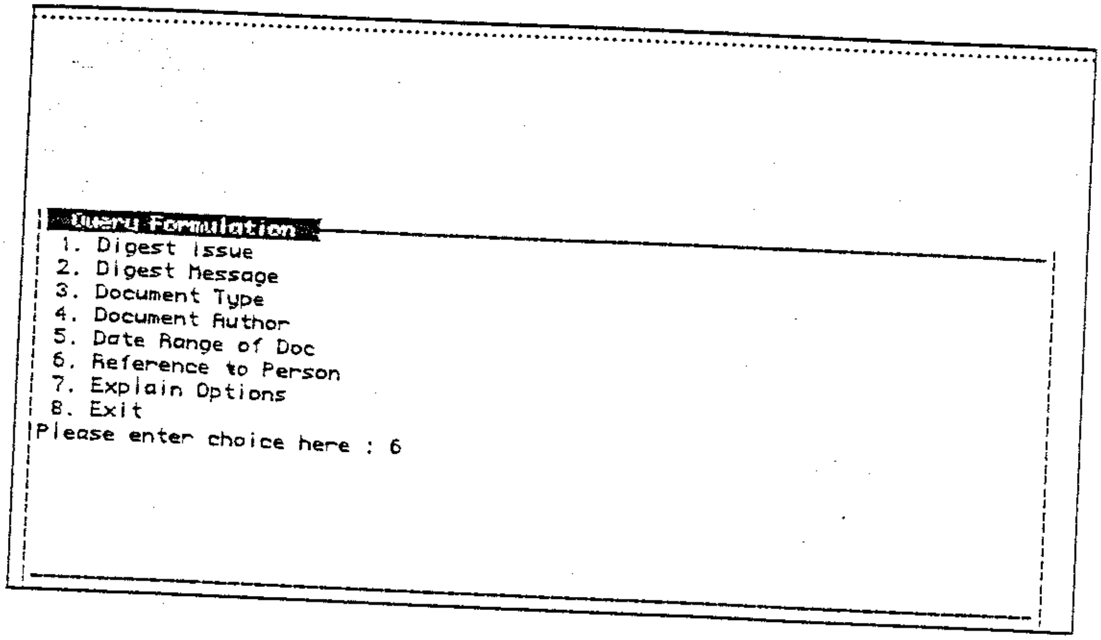
```
Are you looking for particular Recall/Precision?
 1. Higher Recall
 2. Higher Precision
 3. Balance Recall and Precision
 4. Don't Know
Please enter choice here : 3_
```



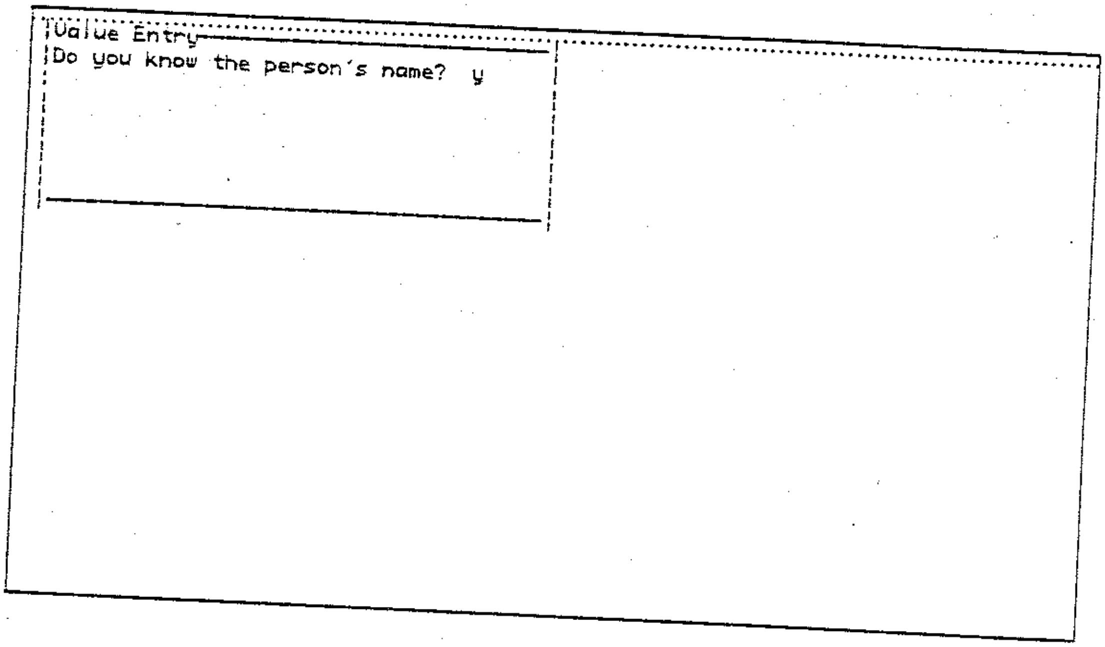
```
 What portion of retrieved documents would you like to receive?
  1. Whole document
  2. Paragraphs
  3. Document Header only
Please enter choice here : 1_
```



```
Query Entry
Do you wish to enter structured knowledge for query matching? (y/n/help) y_
```

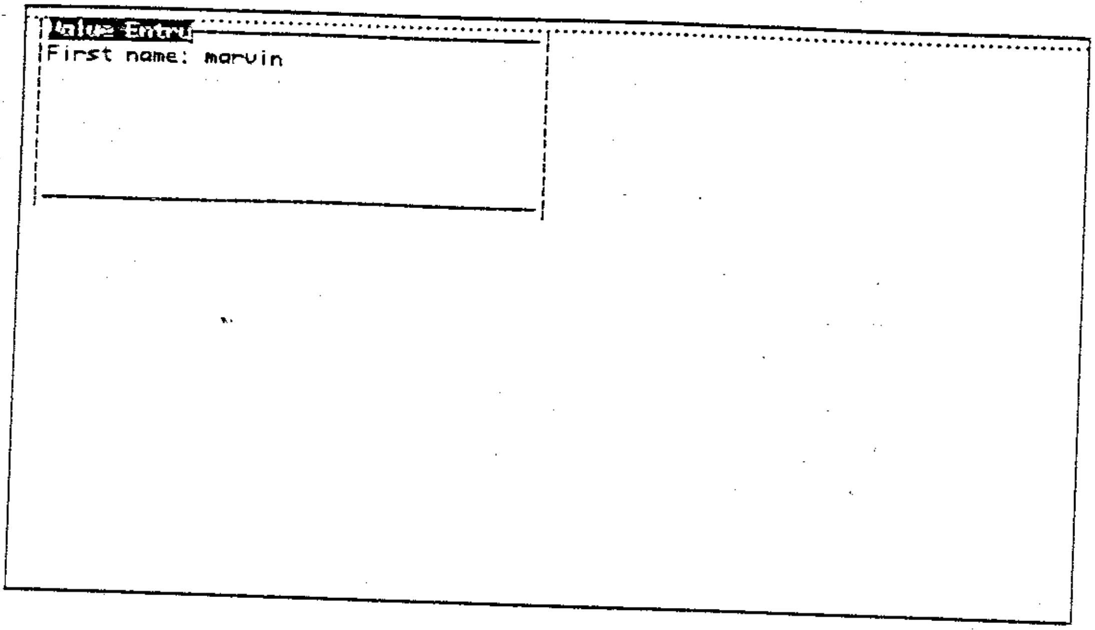


```
Query Formulation
 1. Digest Issue
 2. Digest Message
 3. Document Type
 4. Document Author
 5. Date Range of Doc
 6. Reference to Person
 7. Explain Options
 8. Exit
Please enter choice here : 6
```

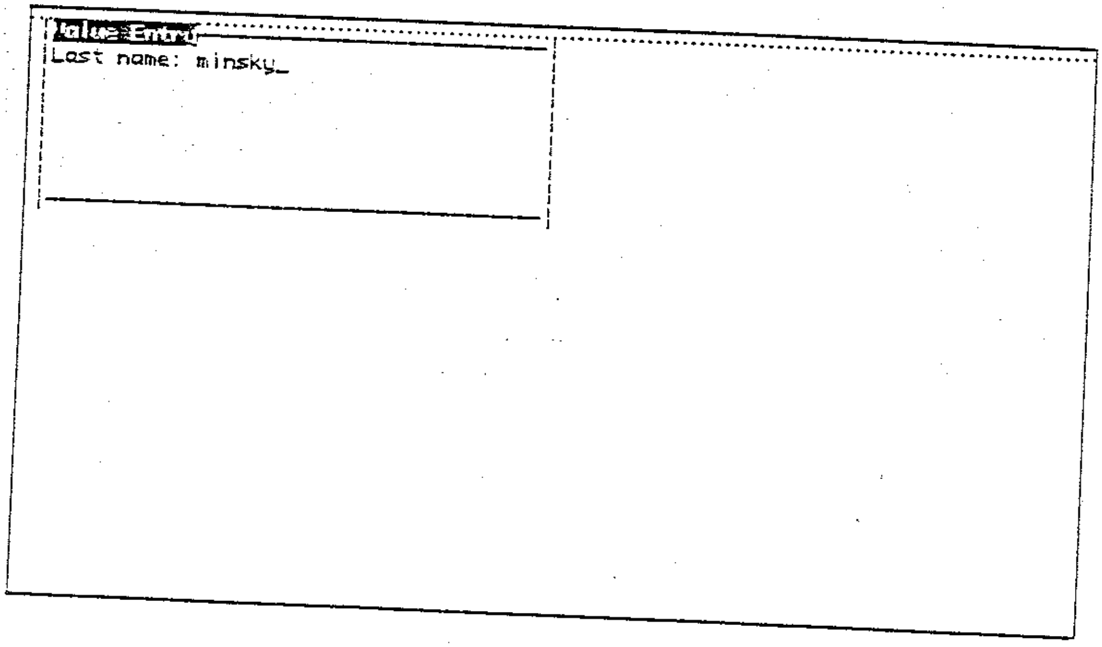


```
Value Entry
Do you know the person's name?  y
```

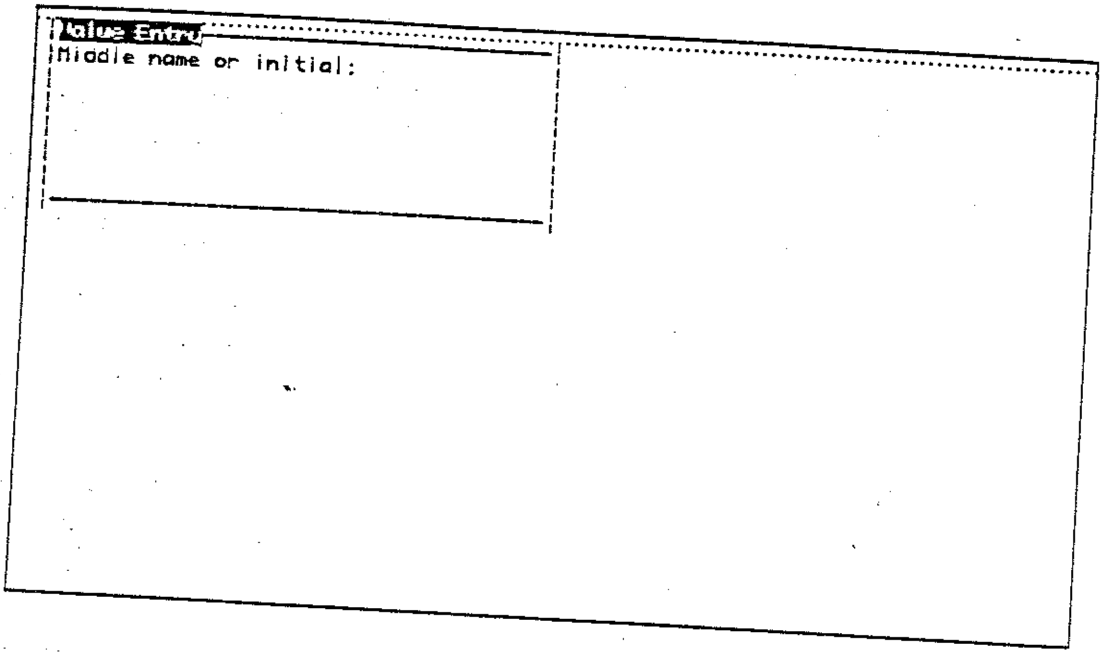
Appendix I. Sample CODER Retrieval Session



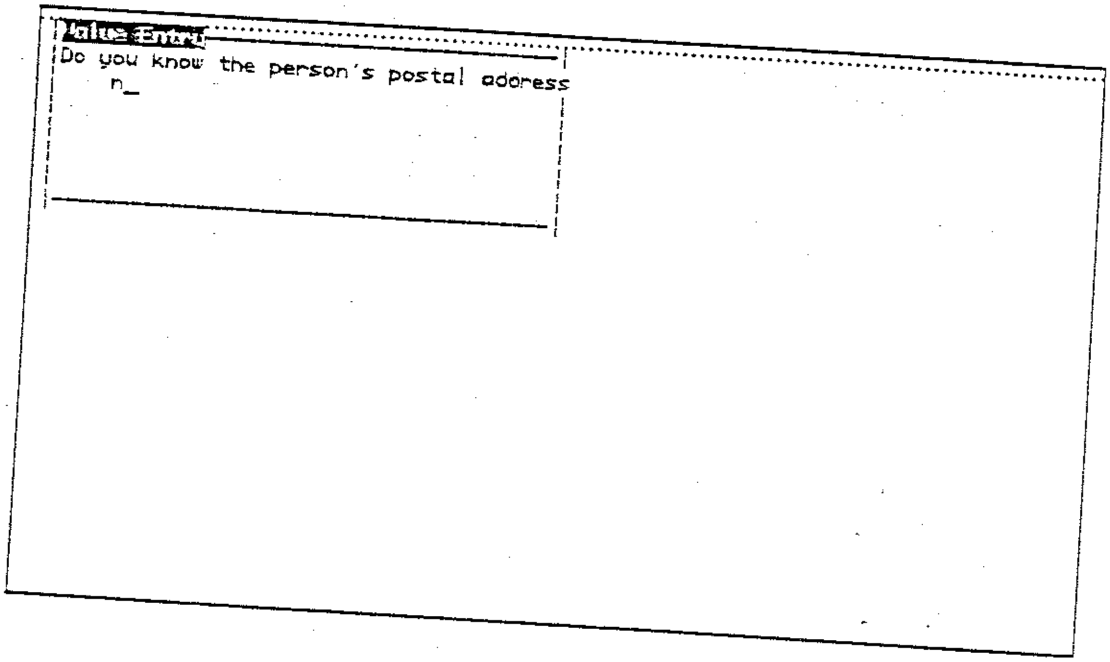
First name: marvin



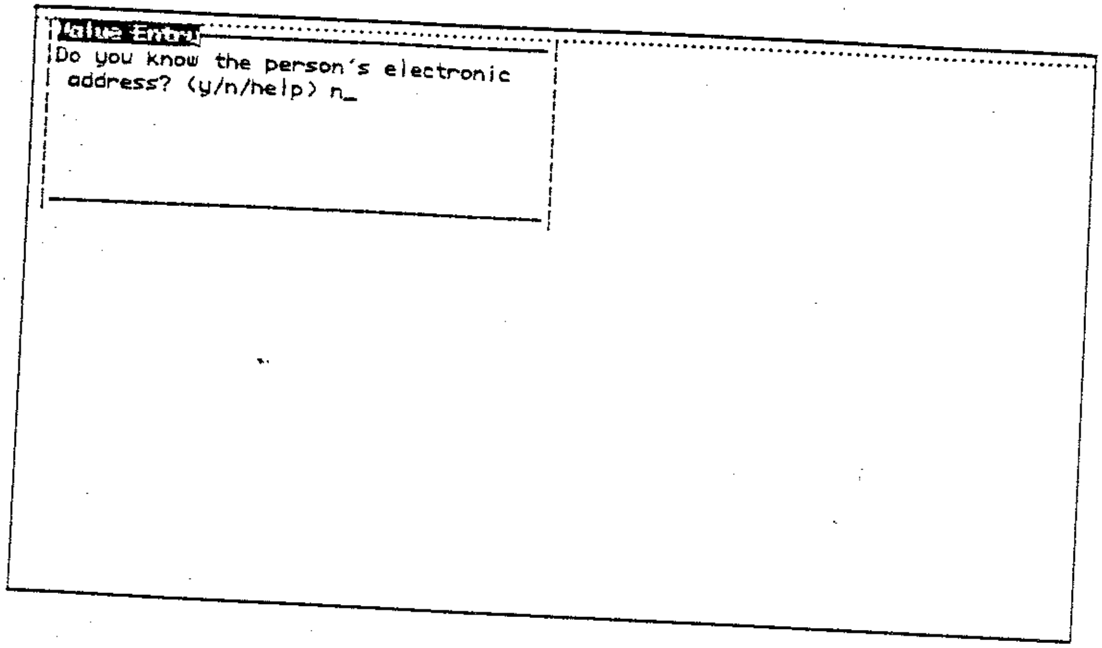
Last name: minsky_



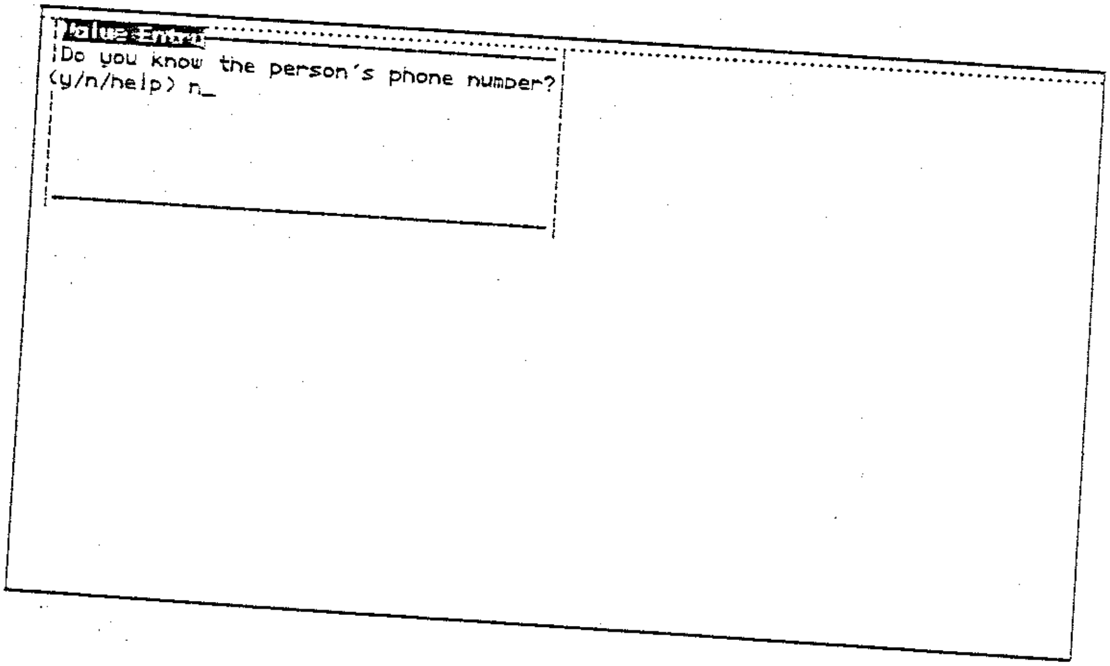
Middle name or initial:



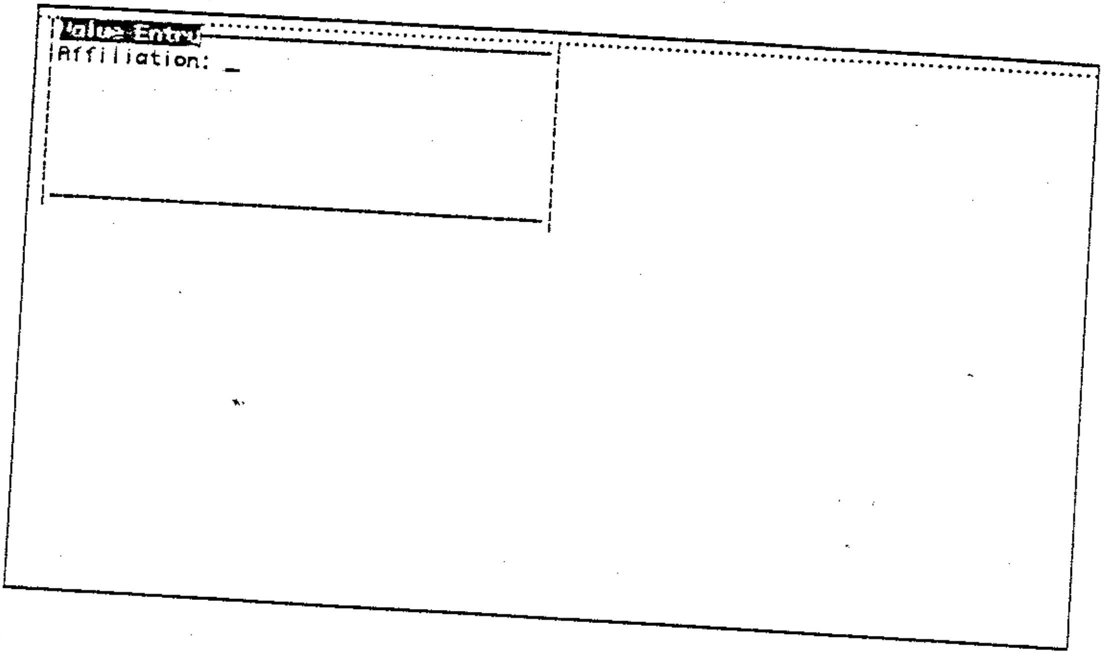
```
Do you know the person's postal address
    n_
```



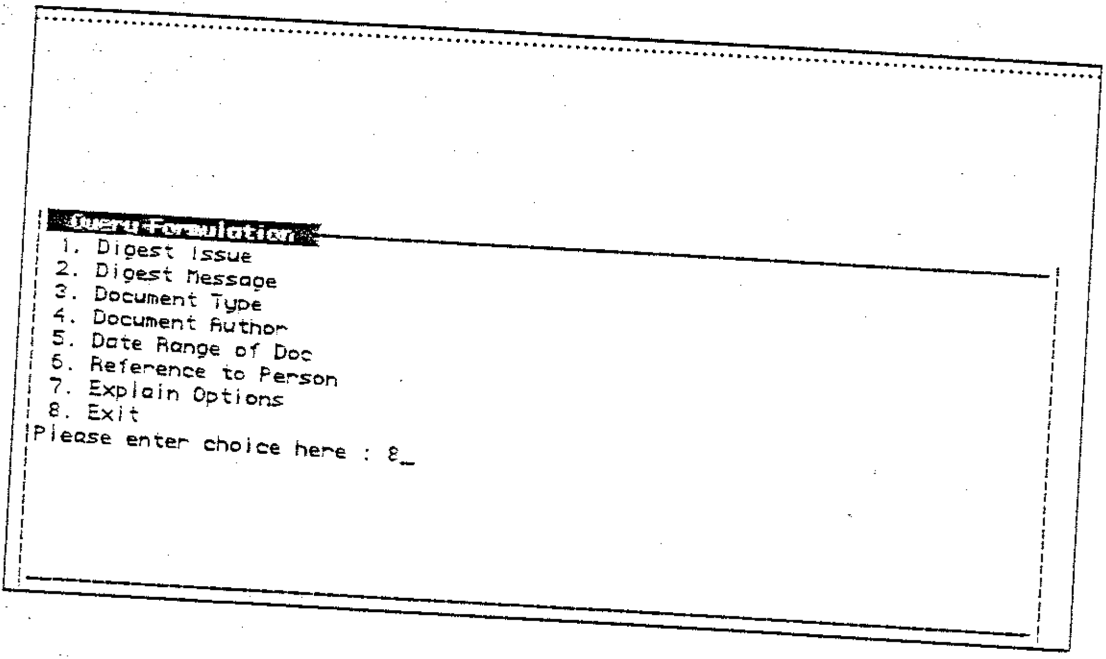
```
Do you know the person's electronic
 address? (y/n/help) n_
```



```
Do you know the person's phone number?
(y/n/help) n_
```

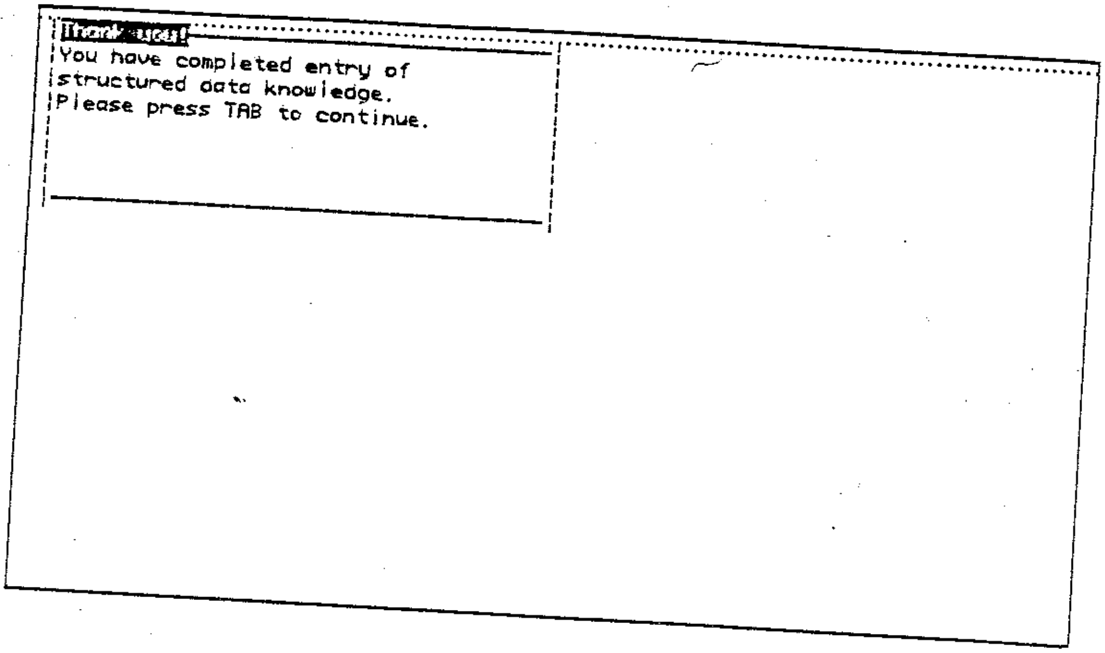


```
Value Entry
Affiliation: _
```

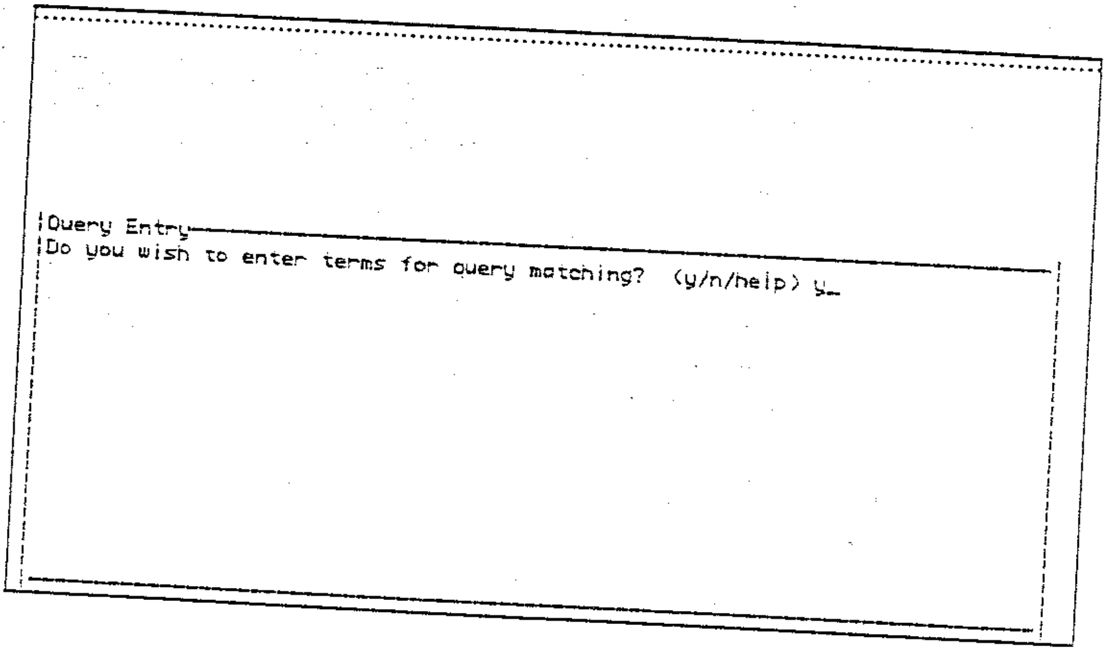


```
Query Formulation
 1. Digest Issue
 2. Digest Message
 3. Document Type
 4. Document Author
 5. Date Range of Doc
 6. Reference to Person
 7. Explain Options
 8. Exit
Please enter choice here : 8_
```

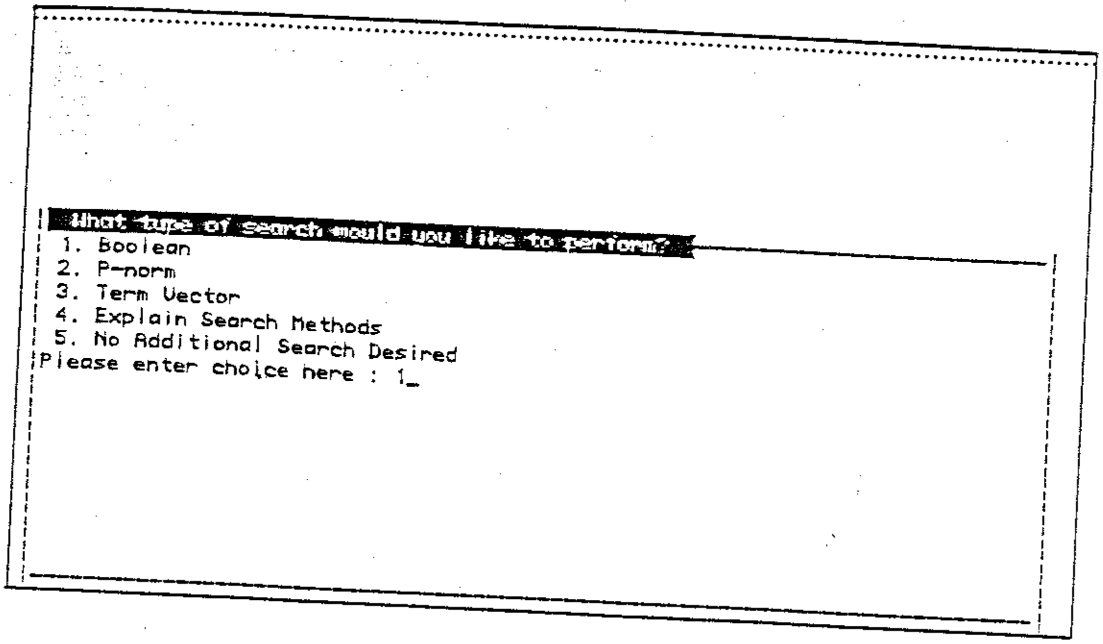
Appendix I. Sample CODER Retrieval Session



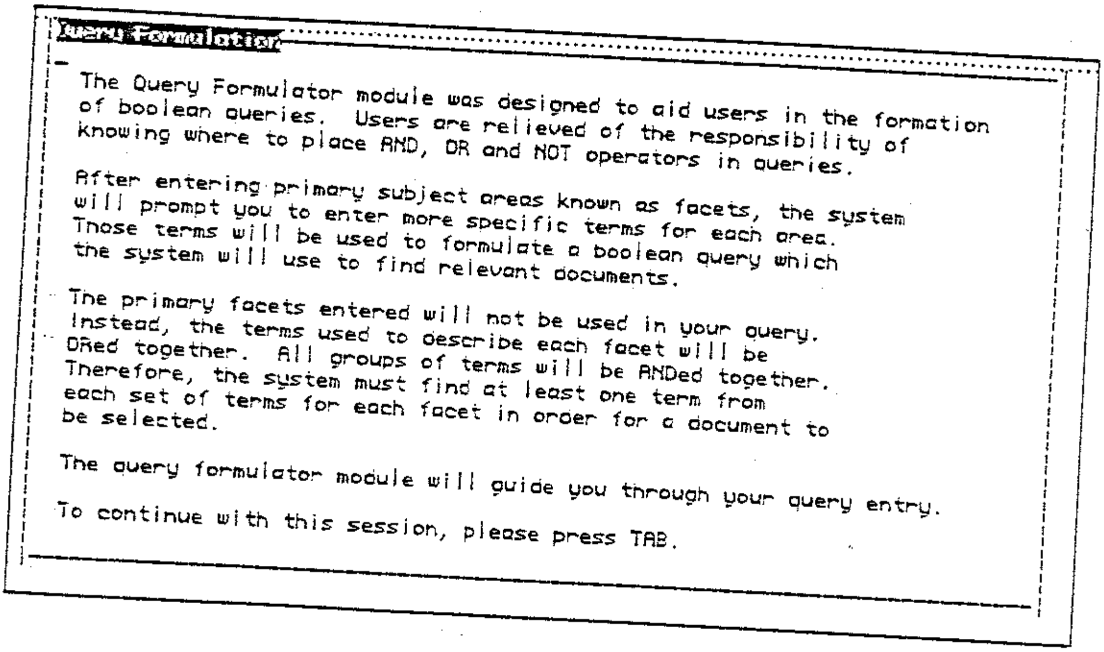
```
Thank you!
You have completed entry of
structured data knowledge.
Please press TAB to continue.
```



```
Query Entry
Do you wish to enter terms for query matching?  (y/n/help) y_
```



```
What type of search would you like to perform?
 1. Boolean
 2. P-norm
 3. Term Vector
 4. Explain Search Methods
 5. No Additional Search Desired
Please enter choice here : 1_
```



Query Formulation

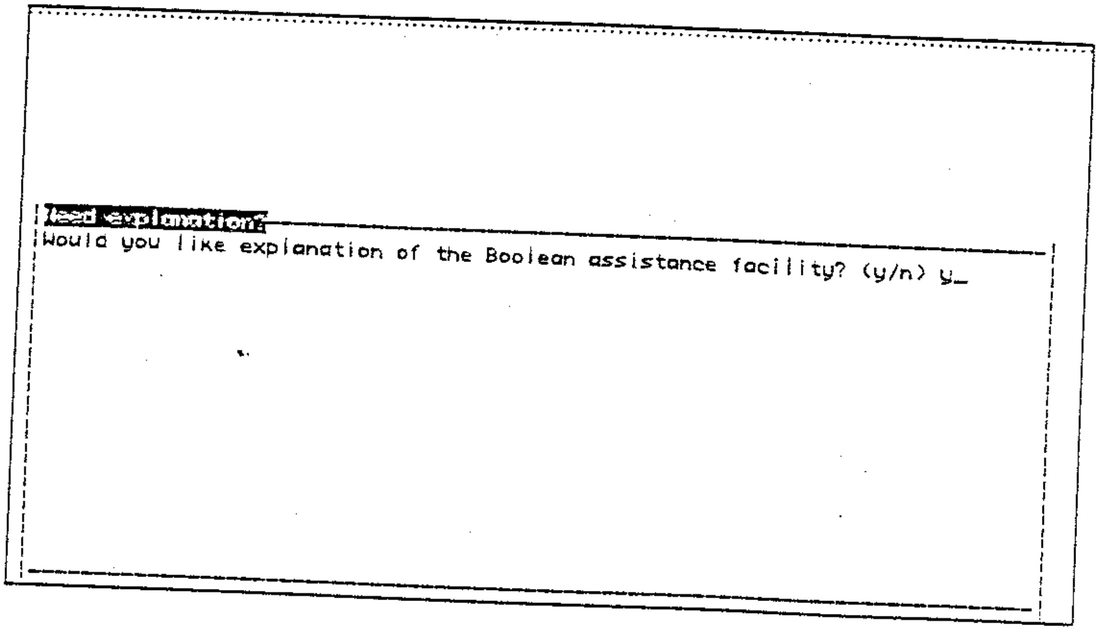
The Query Formulator module was designed to aid users in the formation of boolean queries. Users are relieved of the responsibility of knowing where to place AND, OR and NOT operators in queries.

After entering primary subject areas known as facets, the system will prompt you to enter more specific terms for each area. Those terms will be used to formulate a boolean query which the system will use to find relevant documents.

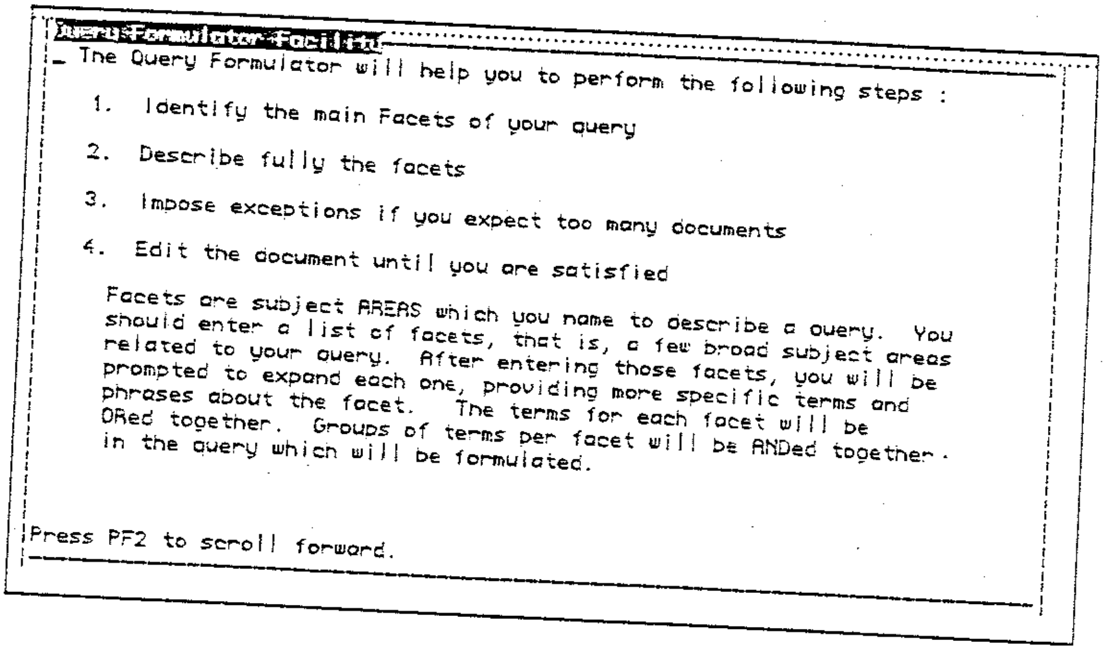
The primary facets entered will not be used in your query. Instead, the terms used to describe each facet will be ORed together. All groups of terms will be ANDed together. Therefore, the system must find at least one term from each set of terms for each facet in order for a document to be selected.

The query formulator module will guide you through your query entry.

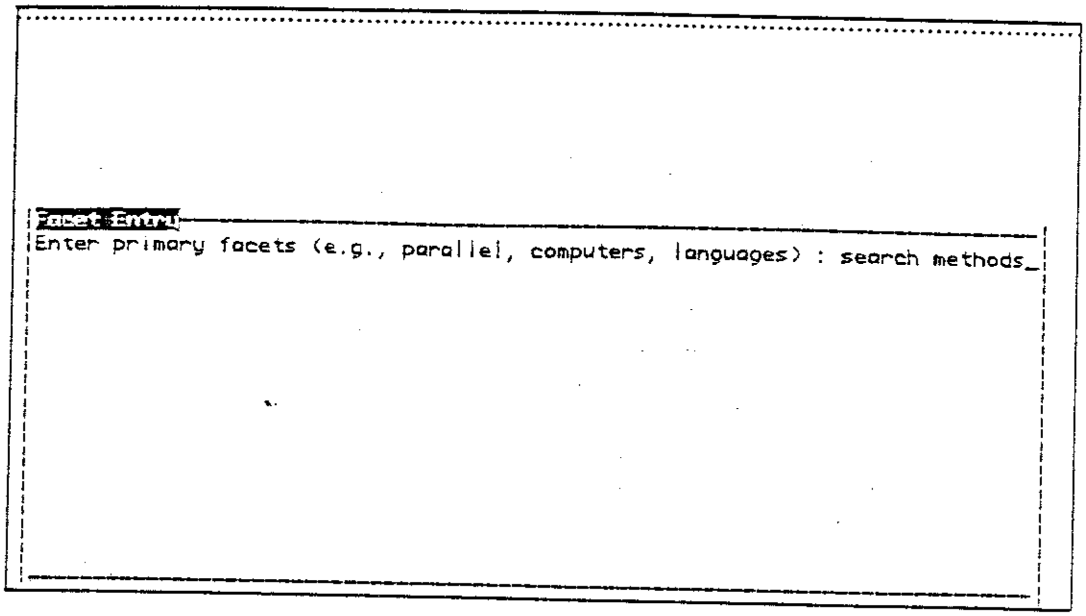
To continue with this session, please press TAB.



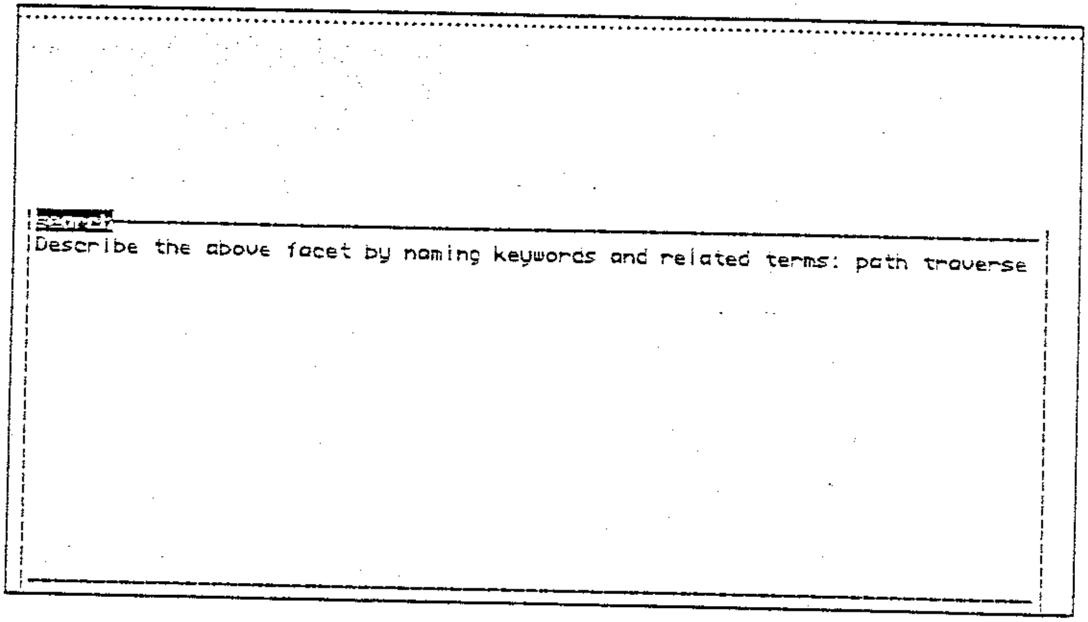
```
Need explanation?
Would you like explanation of the Boolean assistance facility? (y/n) y_
```



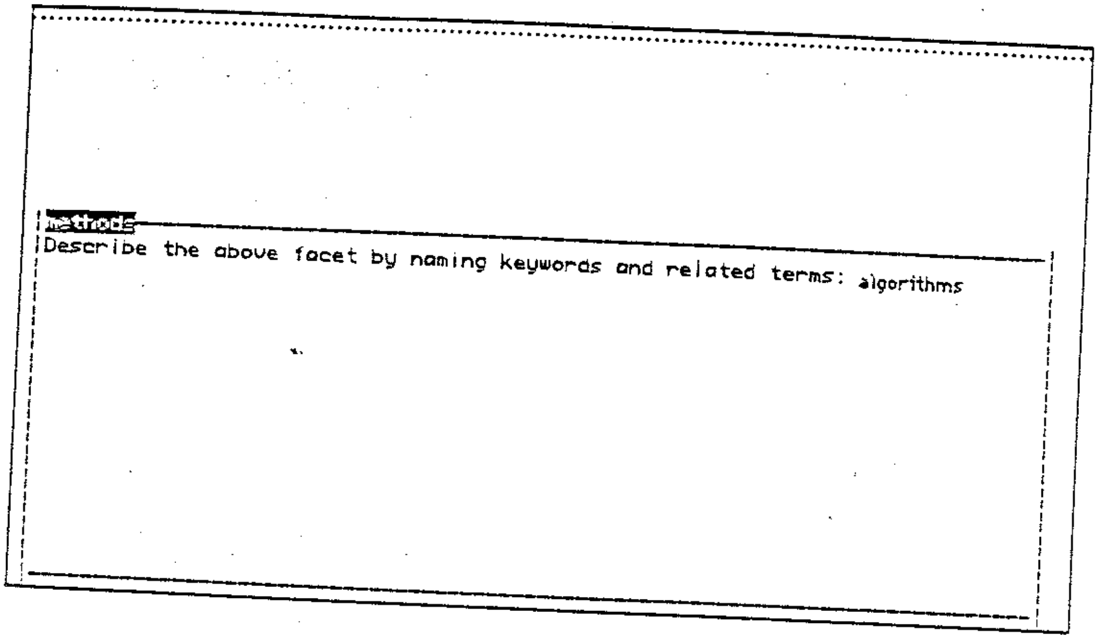
```
Query Formulator Facility
_ The Query Formulator will help you to perform the following steps :

   1.  Identify the main Facets of your query

   2.  Describe fully the facets

   3.  Impose exceptions if you expect too many documents

   4.  Edit the document until you are satisfied

    Facets are subject AREAS which you name to describe a query.  You
    should enter a list of facets, that is, a few broad subject areas
    related to your query.  After entering those facets, you will be
    prompted to expand each one, providing more specific terms and
    phrases about the facet.  The terms for each facet will be
    ORed together.  Groups of terms per facet will be ANDed together
    in the query which will be formulated.


Press PF2 to scroll forward.
```



```
Enter primary facets (e.g., parallel, computers, languages) : search methods_
```

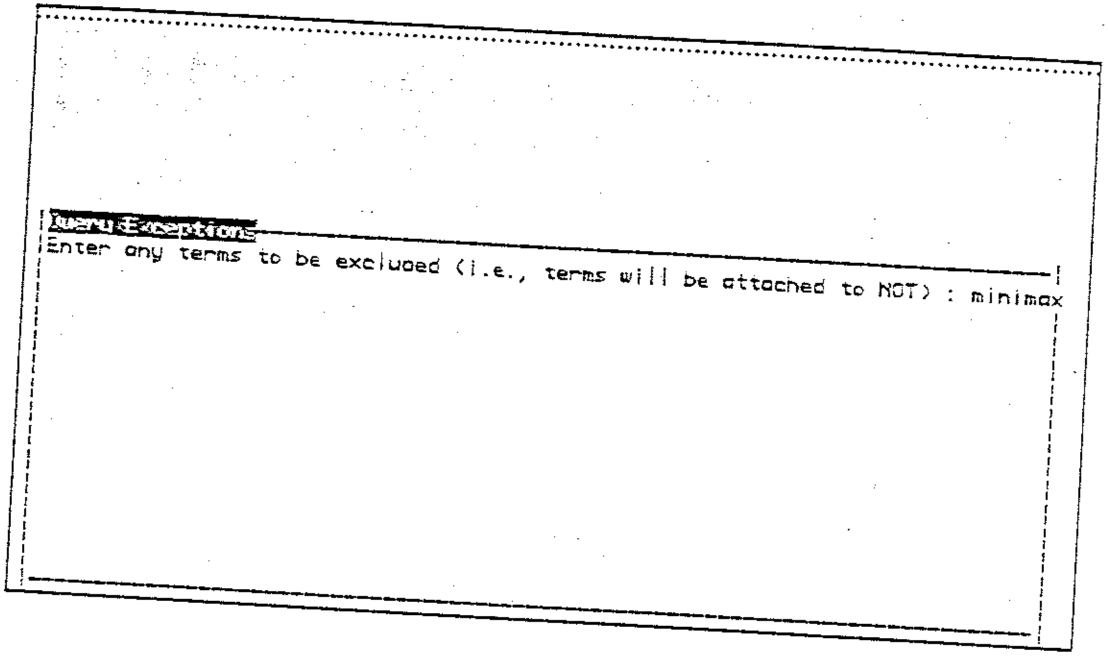


```
Describe the above facet by naming keywords and related terms: path traverse
```



Methods

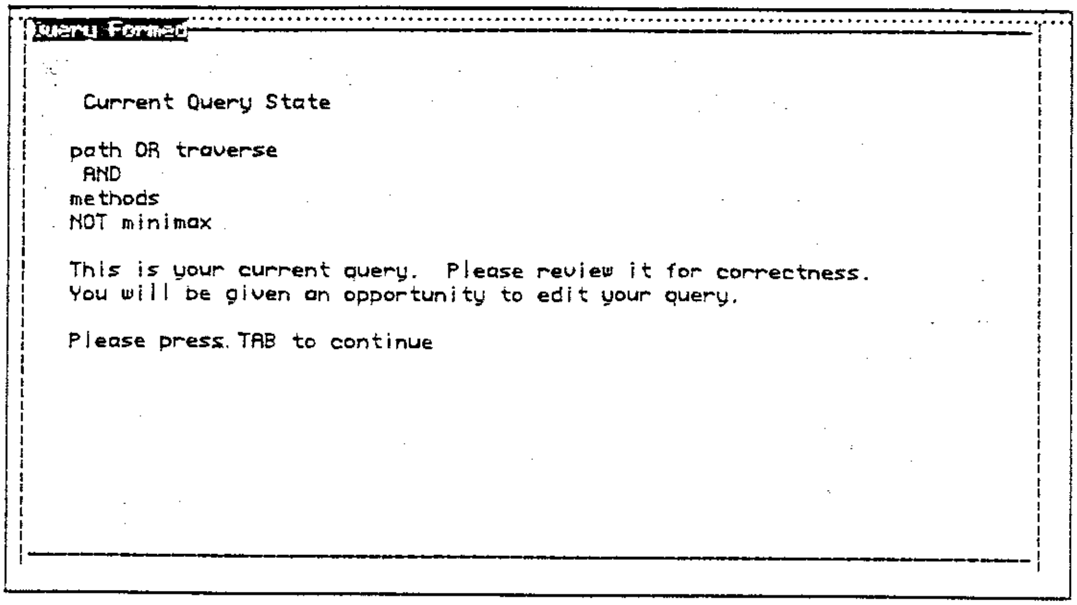
Describe the above facet by naming keywords and related terms: algorithms



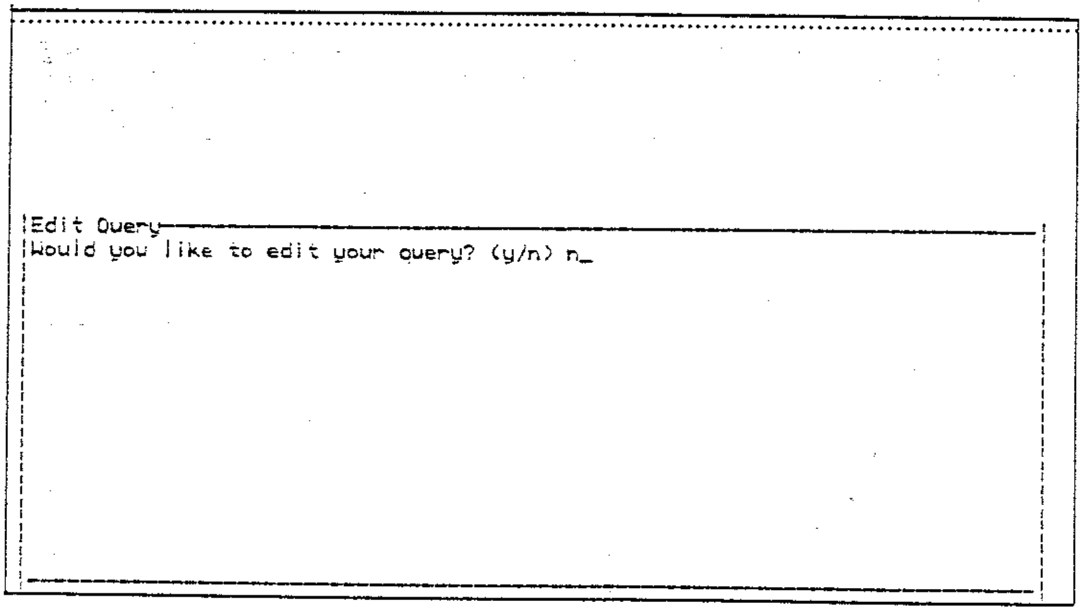
Query Exceptions

Enter any terms to be excluded (i.e., terms will be attached to NOT) : minimax

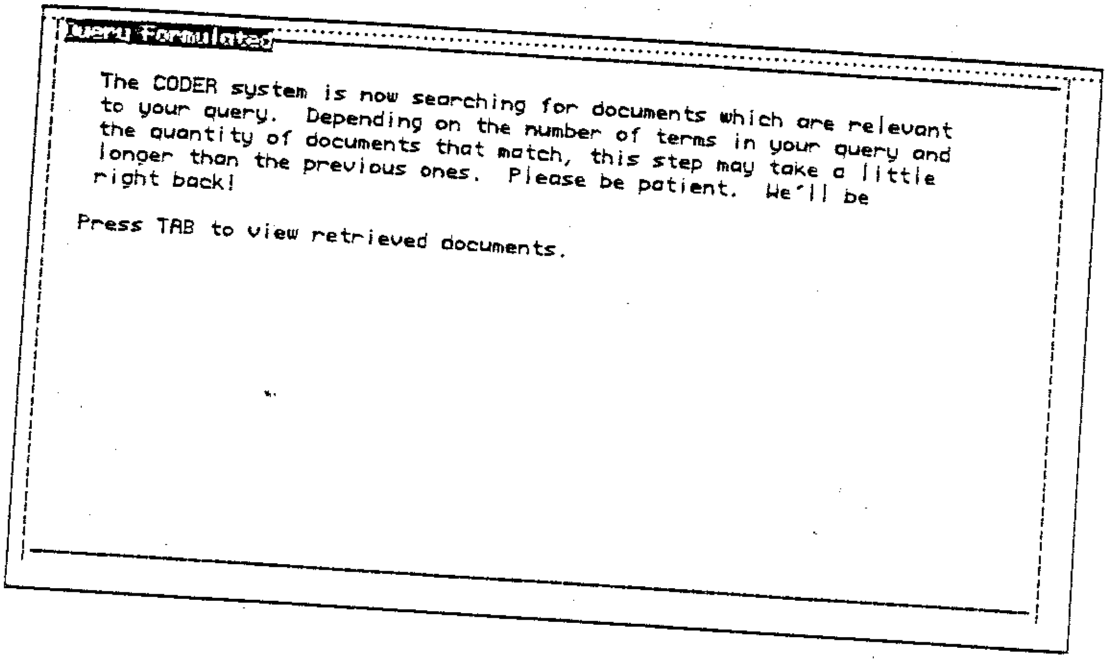
Appendix I. Sample CODER Retrieval Session



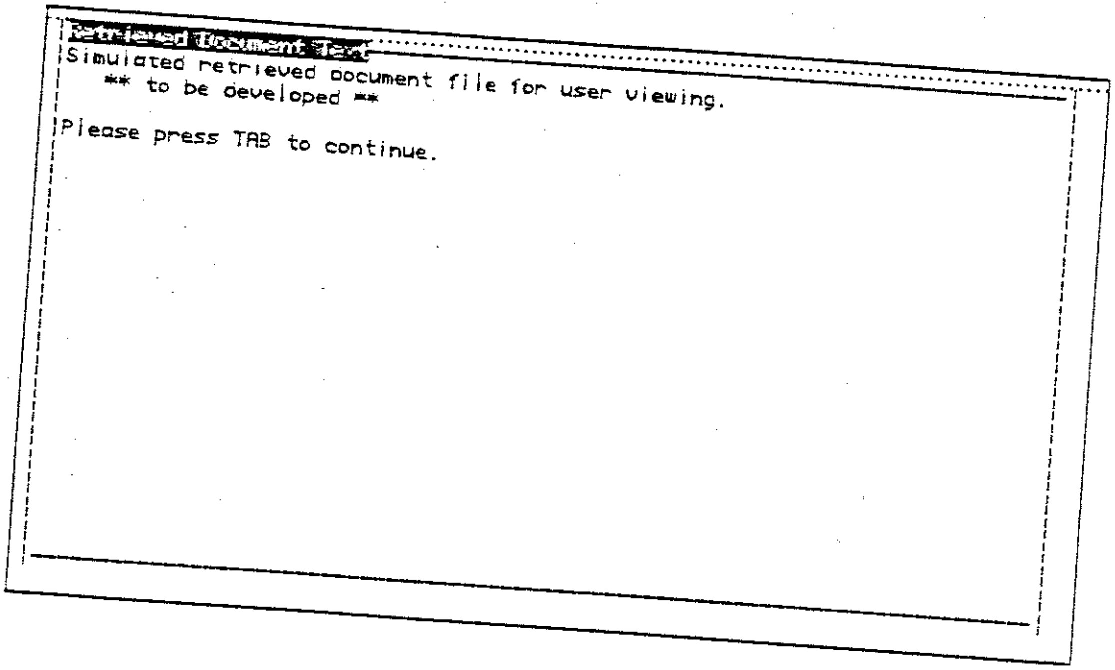
```
Current Query State

path OR traverse
 AND
methods
NOT minimax

This is your current query.  Please review it for correctness.
You will be given an opportunity to edit your query.

Please press TAB to continue
```



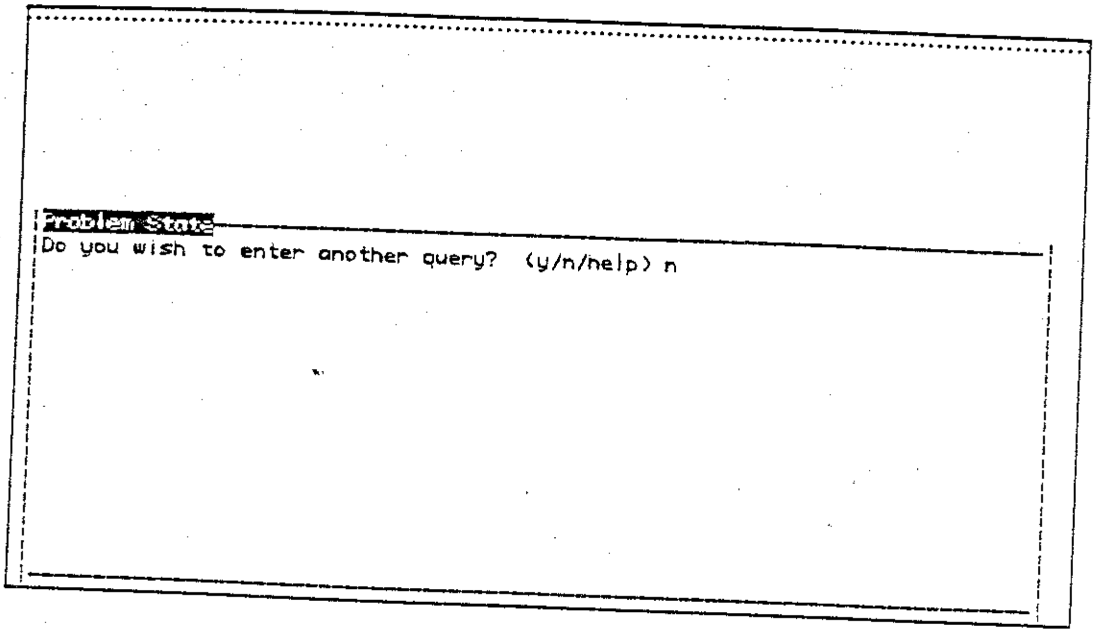
```
Would you like to edit your query? (y/n) n_
```



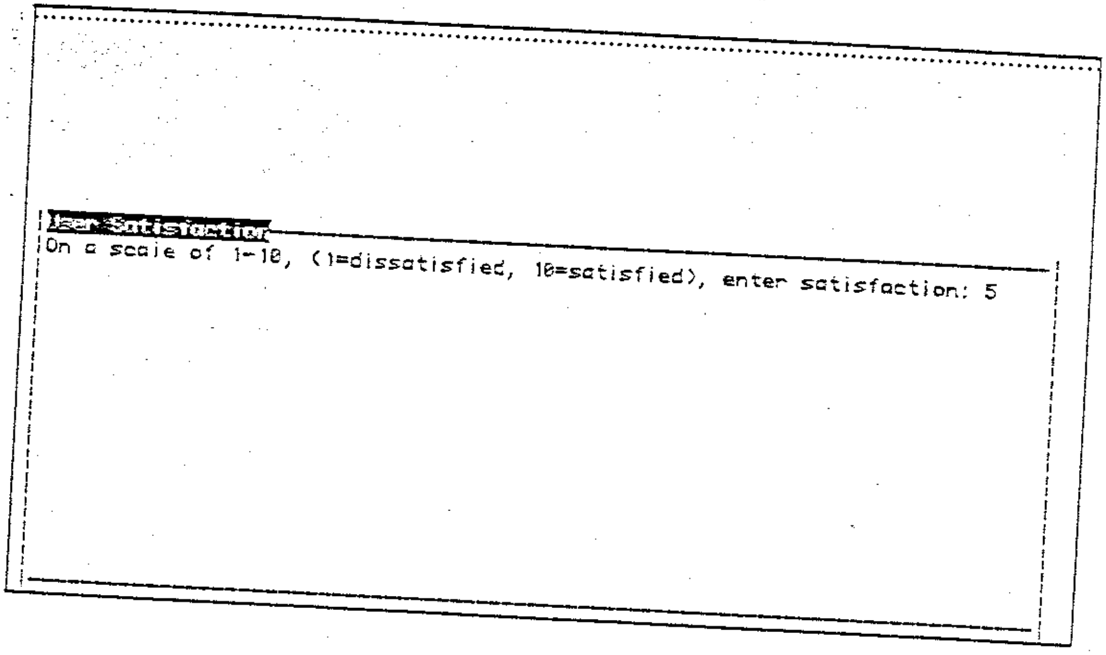
```
The CODER system is now searching for documents which are relevant
to your query.  Depending on the number of terms in your query and
the quantity of documents that match, this step may take a little
longer than the previous ones.  Please be patient.  We'll be
right back!

Press TAB to view retrieved documents.
```



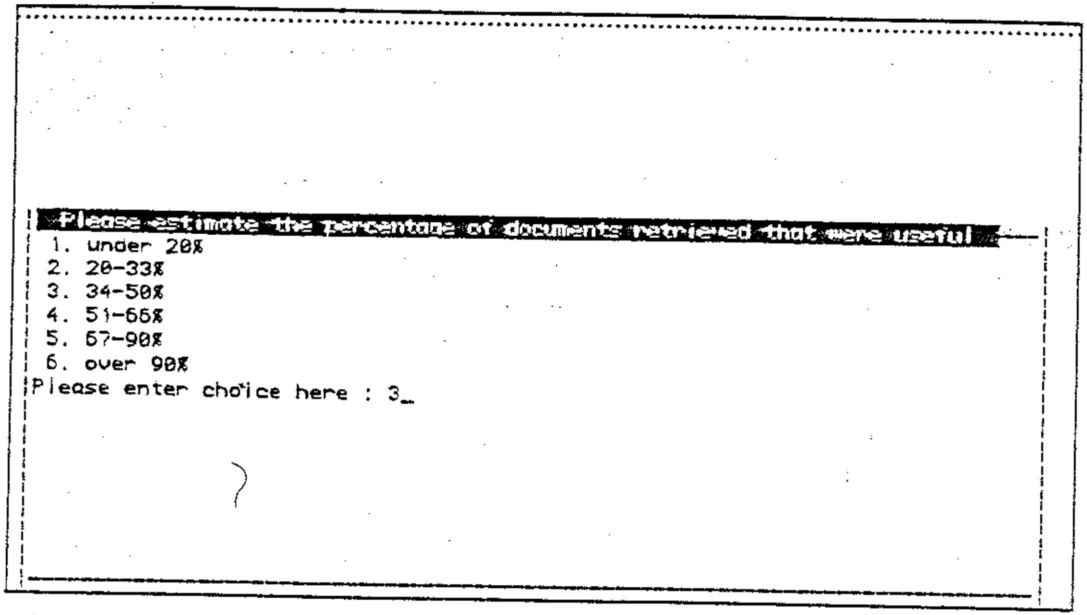
```
Simulated retrieved document file for user viewing.
   ** to be developed **

Please press TAB to continue.
```



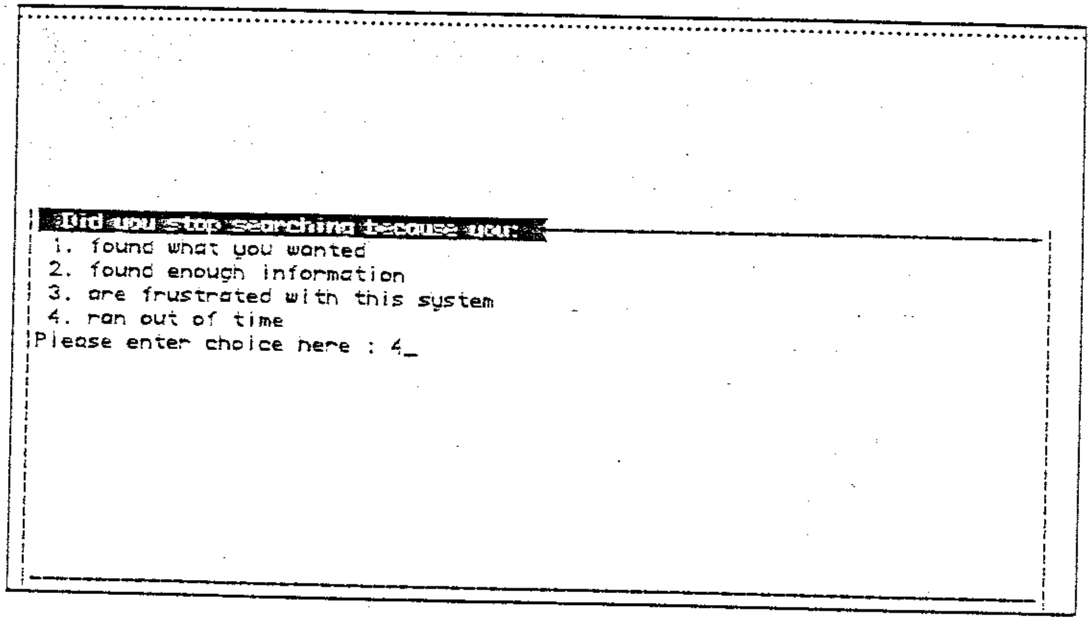
```
Do you wish to enter another query?  (y/n/help) n
```



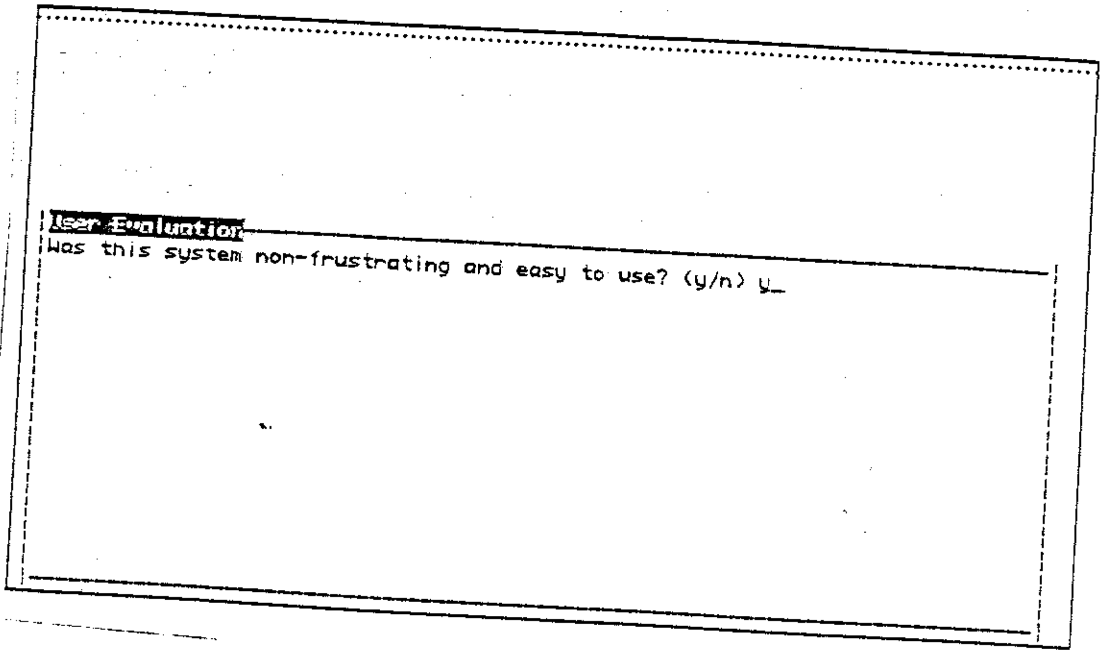
```
On a scale of 1-10, (1=dissatisfied, 10=satisfied), enter satisfaction: 5
```



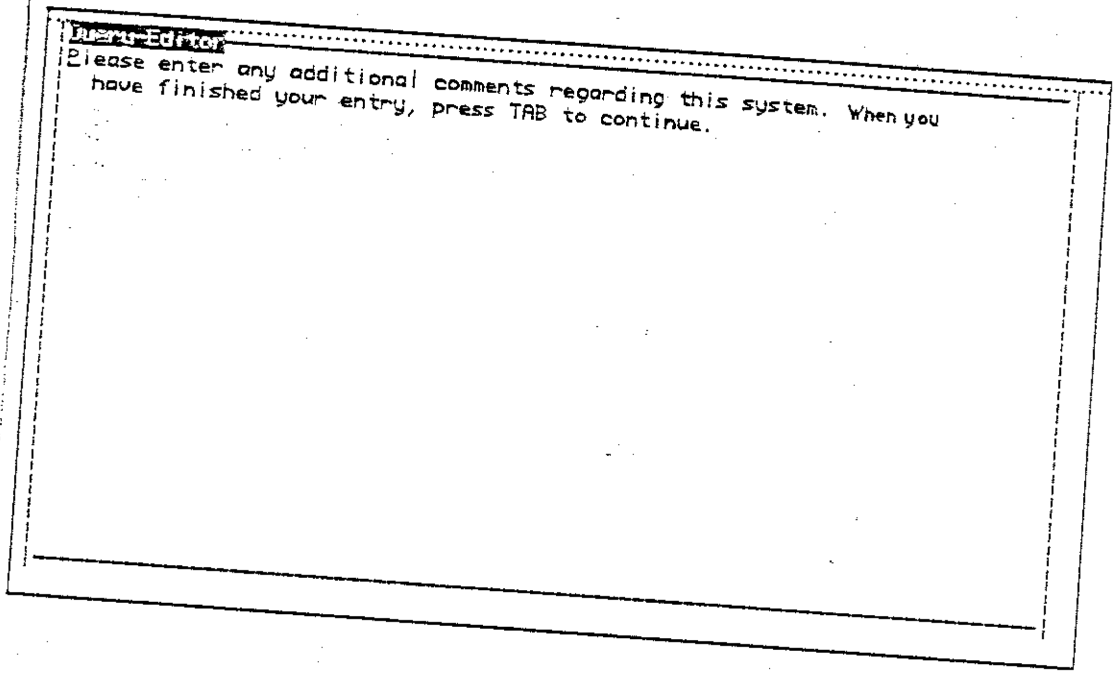
```
Please estimate the percentage of documents retrieved that were useful
 1. under 20%
 2. 20-33%
 3. 34-50%
 4. 51-66%
 5. 67-90%
 6. over 90%
Please enter choice here : 3_
```



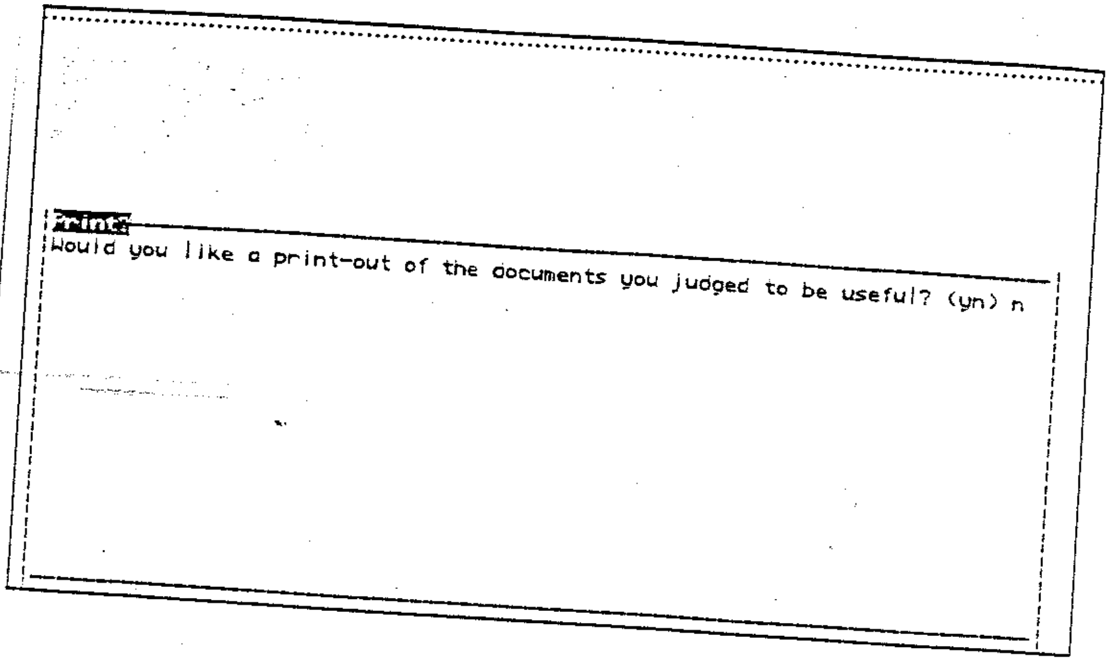
```
Did you stop searching because you:
 1. found what you wanted
 2. found enough information
 3. are frustrated with this system
 4. ran out of time
Please enter choice here : 4_
```



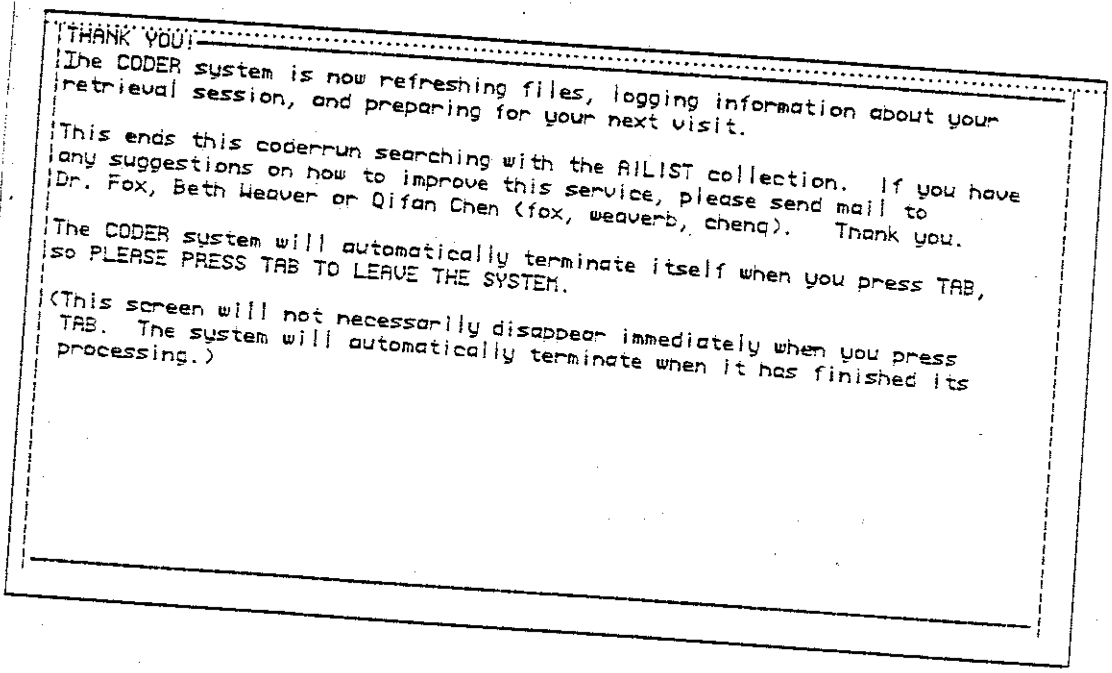
```
Was this system non-frustrating and easy to use? (y/n) y_
```



```
Please enter any additional comments regarding this system.  When you
  have finished your entry, press TAB to continue.
```



```
Print
Would you like a print-out of the documents you judged to be useful? (yn) n
```



```
THANK YOU!
The CODER system is now refreshing files, logging information about your
retrieval session, and preparing for your next visit.

This ends this coderrun searching with the AILIST collection.  If you have
any suggestions on how to improve this service, please send mail to
Dr. Fox, Beth Weaver or Qifan Chen (fox, weaverb, chenq).   Thank you.

The CODER system will automatically terminate itself when you press TAB,
so PLEASE PRESS TAB TO LEAVE THE SYSTEM.

(This screen will not necessarily disappear immediately when you press
 TAB.  The system will automatically terminate when it has finished its
processing.)
```