NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

Canada

CAUSAL REASONING IN A SOFTWARE ADVISOR

by

Branka Tauzovich

Department of Electrical Engineering
University of Ottawa
Ottawa, Ontario, Canada

May 1987

A dissertation submitted to the School of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

ISBN 0-315-46864-5

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

ABSTRACT

Conventional knowledge sources which a user of a computer system might consult when he/she encounters a problem are manuals, on-line help facilities and human experts. Since these are often either inadequate or unavailable, a special kind of expert system, software advisors, is starting to emerge. One of the most important requirements of such advisors is the ability to explain various aspects of obtained results in terms of cause-effect relationships.

We investigate causality in the domain of the fourth generation languages using the QUIZ report writer as an experimental example. A formalism is introduced for encoding knowledge about causal links between fragments of programs and features of results. It makes use of several types of rules and frame-like structures. In the consulting environment, the knowledge thus encoded is a basis for building a causal model of a problem situation described by the query. Strategies for answering various types of user queries are proposed. The methodology has been successfully applied to the design and development of a component of a QUIZ advisor prototype.

Mojim Roditeljima

ACKNOWLEDGMENTS

I am very grateful to my employer, Cognos Inc., and the Natural Sciences and Engineering Research Council of Canada (NSERC), for the generous support I have received through the Industrial Postgraduate Scholarships program, and to the National Research Council of Canada, NSERC, Cognos Inc., and the University of Ottawa for sponsoring the QUIZ Advisor Project. I am particularly indebted to Bob Minns, VP Research at Cognos, for believing in me and giving me the opportunity to pursue my studies in AI.

My many thanks go to Prof. Douglas Skuce of the Department of Computer Science, my thesis supervisor, for the attention, guidance, and support he has given me during the period of my studies and research at the University of Ottawa.

I would also like to express my appreciation to Professors Stanislaw Matwin, Stanislaw Szpakowicz, and Morris Goldberg whose suggestions and comments on the thesis proposal have been most helpful.

Robert Stanley, Sylvain Delisle, Charles Truscott, and other members of the Advisor Project have provided a fertile research environment. I thank them all.

Finally, I would like to give special thanks to my family, for their patience, encouragement, and unfailing support.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1. Motivation and Objectives

To be able to perform any activity, one must possess the required skills (i.e., the relevant knowledge and the ability to use this knowledge effectively). Experts are distinguished by their high level of skills from others involved in a given area. It is mainly due to gaps in their domain knowledge that non-experts, especially novices, often need help in their work.

In a typical working environment help is usually sought from more experienced colleagues, preferably from ones who are considered experts. However, since this type of help is often unavailable, alternate sources of help have to be consulted. In using a software system an alternate source is the system documentation or manuals.

Manuals contain large amounts of knowledge but also have many serious disadvantages:

* finding parts relevant to solving the problem at hand is not always easy due to poor indexing, the lack of glossaries, etc.;
* text written in a natural language is often ambiguous or unclear;
* the material in a manual is sometimes poorly organized;
* the documentation may be inconsistent with the software or inconsistent internally; etc.

On-line documentation is in some way an improvement over the traditional

printed documentation: the contents of manuals are stored in computer files and conveniently indexed to provide faster retrieval of the relevant knowledge. Text contained in such documentation, however, has the same disadvantages as manuals (such as ambiguities or inconsistency).

On-line help facilities are another kind of knowledge source. They traditionally provide assistance in using system commands (e.g., the syntax of statements, examples). To be more helpful, these facilities should be context-sensitive and answer queries at a proper level of detail [Houghton 84]. To meet these objectives, such "intelligent" help facilities are bound to use artificial intelligence techniques, and thus evolve into software advisory systems.

An *advisory system* is an "expert system that interacts with a person in the style of giving advice rather than in the style of dictating commands. Generally advisory systems have mechanisms for explaining their advice and for allowing their users to interact at a detail level comfortable to the user" [Winston, Prendergast 84 (Glossary)].

The work described in this thesis is an integral part of the *Advisor Project*, a joint industrial-academic effort undertaken by Cognos Incorporated, a major Canadian software company, and the University of Ottawa, with financial support from the Federal Government of Canada and the Government of Ontario. The objective of the project is to develop a methodology for capturing and making easily accessible and understandable the knowledge about typical modern software, fourth generation languages. The motivation for focusing on fourth generation languages is the fact that these have been widely accepted in the business community, resulting

in an ever-increasing number of new users.

The methodology has been applied to the development of a prototype advisor for a particular language called QUIZ (described at the end of this chapter). QUIZ has been chosen as the domain for a number of reasons:

* it is a typical representative of fourth generation software;
* it is a mature, widely used product;
* the complexity of the package is appropriate for the intended kind of experiment;
* very knowledgeable QUIZ experts have been available for consultation;
* there is comprehensive documentation on typical problems encountered by QUIZ users.

The specific objectives of research described in this thesis are the study of causal relationships in a fourth generation language as a basis on which to build an advisor system, and the development of a prototype system, a part of the QUIZ Advisor.

The backbone of our system is a causal model of a part of the QUIZ system. Reasoning about causal relationships is essential for good understanding of any subject matter, for explaining system behavior, and for answering some commonly asked types of questions (e.g., "Why did ... happen?"). It is only in recent years that the importance of reasoning based on causal relationships has been recognized by developers of expert systems. De Kleer points out the difference between the approach based on causal models and the traditional expert systems methodology:

"Expert systems are aimed at producing what performance is possible in the short term without consideration of the longer term. Typically this is achieved by recording as many of the heuristics and rules of thumb that experts actually use in practice, as possible. This is misguided. The reasoning of experts is based on underlying theories that must be teased out" [de Kleer 84]. Currently there are few systems that can reason from first principles or causal models in order to infer how to handle unanticipated situations. Such approaches will become commercially important when the demand for robustness increases [Brown 84].

1.2. Interpretation of the Notion of 'Causality'

The notion of causality has been the focus of much discussion among philosophers for several thousand years. Although causality is a concept which all of us have some intuitive feeling about, it is not fully understood and remains a controversial subject in philosophy.

Aristotle pointed out that our enquiry is for the sake of understanding, and we do not understand a thing until we have acquired the 'why' of it, the *proximate cause.*[+] He finds that there are four different senses of cause: *material cause* (e.g., bronze is the material cause of the statue), *formal cause* (the form or the structure, e.g., the design is the formal cause of a house), *efficient cause* (it initiates change or motion or coming to rest, e.g., a moving bat is the efficient cause of a ball's motion), and *final cause* (for the sake of which the change occurs, e.g., the use to

[+] This brief survey of philosophical views of causality is based on the material from [Schlegel 73]. Quotation marks have been omitted for readability.

which the house is put is the final cause of building it).

Newtonian mechanics gave a model of causation in all investigations of the natural world. It reduced the concept of causality to the efficient cause: things causing other things to move becomes the mode of understanding the natural world. In the 18th century, however, the idea of efficient cause as a firm basis for philosophy of nature received a strong criticism from David Hume. After Aristotle, Hume's ideas are regarded as having the greatest influence on the philosophical views of causality. Hume divides objects of human reason into two kinds: *relations of ideas* and *matters of fact*. The former gives us certain truths (e.g., laws of geometry); the latter arise from experience, and, rather than being undoubted certainties, are such that we can conceive of their opposite as being intelligible. All reasoning concerning matters of fact seems to be founded, Hume says, on the relation of Cause and Effect. With that relation we can go beyond the evidence of our senses (for example, you believe that a person has recently been in a room because you find a fire in the fireplace). All causes and effects are discovered by experience. Hume claims that there is no necessity of causal connection in nature -- only repeated sequences of events to which we apply the name cause-and-effect.

Bertrand Russell does not deny that regularities occur, and that, in the case of frequently observed sequences, we may say that the earlier event is the cause of the later event, but the sequence is only probable. We should, he claims, only believe in causal sequences where we find them, without any assumption that they must be always found. Laws of probable sequence, although important in daily life and in the early stages of a science, tend to be displaced by the concept of *func-*

tional dependence as science advances.

It seems that philosophers have not produced many firm answers, but they did bring up some general questions about causality: determinism, necessity of temporal succession, relevance to answering 'why' questions, and degree of subjectivism involved. Knowledge engineers, being concerned with rather practical aspects of representing causality within specific domains, have dealt with these issues in a variety of ways (a more detailed survey will be presented in Chapter 2). Very few, however, have discussed the issues from a theoretical point of view, or defined precisely what they consider to be causality in their domains.

Within the artificial intelligence literature, Chuck Rieger has done significant work on the representation of *commonsense algorithmic knowledge* which he defines as knowledge that relates to actions, states and the notions of causality and enablement in the activities we all do day to day [Rieger 76]. Focusing on a more narrow domain of physical mechanisms, he introduces the declarative representation of a mechanism in the form of a cause-effect graph whose nodes are events and whose links are drawn from a set of ten theoretical forms of inter-event causal interaction [Rieger, Greenberg 77]. According to Rieger, each event falls into one of four categories: *action*, *tendency*, *state* or *statechange*. Tendency is an event in which there is an absence of intention (e.g., gravity), while other terms are intended to reflect their standard connotations. One of the characteristics of links that the taxonomy is based on is the types of linked events. In addition to links between an action (or tendency) as the cause, and a state (or a statechange) as the effect, which are explicitly called *causal*, major types of links are those between a

state and an action (*enablement*) and between two states (state *coupling*, *equivalence*, and *antagonism*). Although these do not represent direct causal relationships they are useful in reasoning for the purpose of building causal chains between events (e.g., state coupling links provide a means of expressing implicit intervening causal relationships that are either unknown or irrelevant to some description). Causal, enablement and state coupling links can be *continuous* (the continuous presence of the cause is required to sustain the effect) or *one-shot* (the cause event is required only at the start of the effect event). A link can also be *direct* or *gated* by some state specification (i.e., the state must be present for the causal relationship to exist).

Rieger claims that the type of knowledge described above explains the "why's" and "how to's" of the various components of mechanical objects and devices. In knowledge engineering, we see many examples of systems which deal with causality in an explicit manner for generating answers to the 'why' type of questions as its main purpose. It should be noted, though, that not all 'why' questions represent requests for explanations in terms of cause-effect relationships. When we ask, for example, "Why does the sum total of the angles in a triangle equal two right angles?", we expect a proposition providing a sufficient reason for the geometrical theorem. But in everyday language (and even in science), questions about logical reasons of a proposition are interpreted as questions about causes [Puterman 77]. Logicians have tried for some time to reduce causality to implication, if only because causality has proved to be a difficult notion to pin down, while implication is well understood [Charniak, McDermott 85]. One of the consequences are misunderstandings about the character of the temporal relation

between a cause and an effect. The temporal relation is not involved at all by the relation of the logical entailment of propositions, while it is significant for the concepts of causal connection. The notion of causation relies on the notion of change, which in turn relies on the notion of time [Shoham 85]. We may say that the concept of a causal relationship is related to the question 'why' in its version demanding an explanation of an actual state of affairs by reference to some preceding state. Such an explanation not only points to a regular succession of states, but also refers to the dynamic link between earlier and later events [Puterman 77].

Hence we may debate: are there causal relationships in a software system, or we find only logical implications and functional dependencies? Since our primary goal here is not a deep philosophical discussion of causality but rather a practical approach to system implementation, we will define what we *term* a causal relationship in QUIZ.

Speaking about software systems in general, we may say that certain characteristics of inputs (programs, data) cause certain characteristics of outputs (results). In our domain, we consider the presence or absence of some QUIZ statement (or part thereof) as causes, and characteristics of results as effects. Also, if some conditions on the inputs and execution environment are not satisfied, this may cause an error message to be issued. QUIZ being an interactive system, each user command causes the system to respond in some way. Once a statement or a command is issued there is a chain or sequence of events that take place (in a fixed temporal order) before the user can observe its effects. These events are actions (with QUIZ as the actor) or internal states of the system. We believe that such a chain,

presented in a convenient fashion, can best answer the 'why' questions about a software system. By *causal knowledge* we will mean knowledge needed to build such chains. As we will see in Chapter 4, such knowledge will be mainly encoded in what we will call *causal rules*. Among several types of rules, there are those that can be interpreted as logical (e.g., PT-rules), and those that can be considered functions (e.g., quantity-related R-rules), but they all help in establishing causal links between inputs and outputs. In that respect, our view of causality is similar to Rieger's.

1.3. An Overview of QUIZ

In this section we will give a brief description of QUIZ, the domain to which we have applied our methodology. QUIZ is a report writer developed by Cognos Incorporated (formerly Quasar Systems Ltd.) [QUIZ 85].

A QUIZ program consists of a sequence of statements that specify the desired report. The actual report is produced in two phases: parsing and reporting.

In the first phase, statements are interpreted one by one and an internal representation of the report structure is built. During this process QUIZ has access to a data dictionary in which various data elements (files, records, items) are named and given features (e.g., the type and default reporting format of an item) that QUIZ uses when building reports. Actually, the key to the simplicity of QUIZ lies in its relationship with the dictionary: if standard features defined by the dictionary are acceptable, the user can generate many different reports with a minimal amount of coding.

In the second phase, the system generates the actual report by combining report structure information built in the first phase with data obtained from the user's database.

Suppose that a certain large company has a database containing information about its employees, and that the personnel department needs, for example, a list of all employees who are located in Ottawa. In addition to his or her name, each employee's position should be indicated, and the employees should be reported in alphabetical order of their last names. The following is an example of a QUIZ program that creates such a report:

```
ACCESS EMPLOYEES LINK TO POSITIONS
SELECT IF CITY = "OTTAWA"
SORT ON LASTNAME
REPORT LASTNAME FIRSTNAME POSITION
GO
```

Each QUIZ program starts with an 'ACCESS' statement that declares the file (or files) to be used, i.e., states where the data to be reported comes from. It roughly corresponds to the 'join' database operator (more specifically, the 'equi-join'). In our example, two files are specified: 'EMPLOYEES' and 'POSITIONS'. The statement causes the corresponding pairs of records from the two files to be linked through a common item[+] (e.g, the employee number), thus making any data item belonging to either record available for reporting.

[+] an 'item' corresponds to an 'attribute' in the general database terminology.

One often wants to report only those records that satisfy a particular condition. The selection of records is done in QUIZ by using the 'SELECT' statement. The above program selects only records which have 'OTTAWA' as the value of the item called 'CITY'.

By default, QUIZ reports the records in the order in which they are retrieved from the database. If a particular order (e.g., alphabetical on an item, as in our example) is desired, the corresponding 'SORT' statement must be included in the program.

Another important QUIZ statement is the 'REPORT' statement. It indicates what items from the files declared in the 'ACCESS' statement are to be included in the report, thus corresponding to the database 'project' operation. In the above example, it is specified that we would like the report to contain the following information for each employee: last name, first name, and the position. The items appear in the report in the order in which they are specified in the 'REPORT' statement.

An indication that the report has been defined and, at the same time, the instruction to the system to generate (or execute) the report is given by a 'GO' statement. The report that QUIZ would generate based on our program would have the title (the default report title retrieved from the data dictionary), the date, and the page number shown at the top of each page. Data would be reported in three columns, one for each item specified in the 'REPORT' statement, and each column would have a default heading (because we did not specify any other heading to override the default).

The order of statements between 'ACCESS' and 'GO' is irrelevant (with some exceptions which do not seriously affect the philosophy of the language). This reflects the non-procedural nature of QUIZ: the program specifies the desired report, but does not prescribe the procedure for the system to follow in order to generate the report.

QUIZ has more than a dozen statement types, but the ones discussed above are sufficient for the creation of a large number of useful reports. Therefore, a novice can quickly master the basics of QUIZ. For example, in order to obtain an easy-to-read listing of the entire contents of file 'X', all the user needs to specify is

```
ACCESS X
REPORT ALL
GO
```

At its more advanced levels, QUIZ can be used to produce reports that involve complex manipulation of data or satisfy some complex formatting requirements.

The first version of QUIZ was released in 1979 for the Hewlett-Packard HP3000 Series of minicomputers. In recent years it has been incorporated into POWERHOUSE, Cognos' most widely installed fourth generation application development language on minicomputers, available on HP3000, VAX, and Data General's MV/Eclipse. In addition to QUIZ, POWERHOUSE comprises an on-line interactive screen developer and transaction processor called QUICK, a high-volume transaction processor QTP, and a data dictionary package. The components are integrated and work interactively through a common data dictionary. Each POWERHOUSE component has its own comprehensive documentation, as well as the established phone-in client assistance service. Robust advisory systems

could eventually replace both the documentation and the phone-in client support.

1.4. Outline of the Thesis

This thesis contains seven chapters and two appendices.

A review of the relevant literature is presented in Chapter 2. Very little has been done in the area of software advisors, and virtually nothing in the explicit modeling of causal relationships in such systems. Therefore, our review of literature takes a broader look at various ways causal relationships have been represented in the areas of knowledge and software engineering.

In Chapter 3, we analyze a sample of questions the users of QUIZ typically ask, focusing on those that can be interpreted as requests for an explanation of results (e.g., "Why ... ?"). Next, we propose a format for submission of queries, and outline a general query-answering strategy.

Knowledge representation is the topic of Chapter 4. A formalism is introduced that combines the use of several types of rules to represent different causal links between inputs and outputs of QUIZ, and organizes the rule base into frame-like structures. The formalism is syntax-oriented which facilitates reasoning about instances of QUIZ programs and encourages a systematic knowledge acquisition. Causal models of two aspects of QUIZ, programs and interactive sessions, are developed using this approach.

Chapter 5 describes the design and implementation of QAUZ, the causal query-answering module of the Advisor prototype. The primary role of QAUZ is

to provide explanations of unexpected or puzzling features of QUIZ reports. The system has an interface which provides a number of commands that enable the user to build background (context describing) information and issue a question. The appropriate versions of the two basic reasoning mechanisms, forward and backward chaining of rules, are presented, and their roles in query-answering are discussed. The internal form of the QAUZ knowledge base is described.

In Chapter 6, we present details of algorithms for answering major types of queries. In order to take advantage of as much information contained in a query as possible, the system explores individual elements of the query and tries to establish causal links among them. It parses QUIZ statements and uses forward chaining of rules to determine the effects the code has on the result. When analyzing descriptions of the report, it performs backward deduction to determine which features of the code caused that result. The information contained in the trace of the reasoning determines the type and content of the answer. The ability of QAUZ to discover contradictions hidden in queries is discussed, as well as its integration into the query-answering.

Chapter 7 contains the summary and an evaluation of the developed knowledge representation formalism and query-answering strategies. It also suggests possible directions for future work.

A listing of the QAUZ knowledge base in its external form is presented in Appendix A, followed by an example of a complete query-answering session that illustrates various features of the QAUZ prototype (Appendix B).

2. RELATED WORK

In this chapter we will review previous research efforts related to our area of interest. A survey of the literature has revealed several reports on the development of software advisors. Most of these treat causality as an issue of secondary importance, none of them has fourth generation software as its domain. Therefore, we extend the scope of our review to some examples of work that involves reasoning with causal relationships in the more general areas of expert systems and software engineering. The findings will be summarized at the end of the chapter.

2.1. Software Advisors

2.1.1. The UNIX Consultant

The best known software advisor today is the *UNIX Consultant (UC)* developed at the University of California at Berkeley [Wilensky 84; Wilensky et al. 84; Chin 83]. It is an intelligent interface which advises naive users in the domain of the UNIX operating system. UC accepts user queries and responds to them in English. Types of queries UC is capable of handling are those related to plans for accomplishing tasks in the UNIX environment, the syntax of commands, definitions of terminology, and help in using UNIX commands. UC has the following components: a robust language analyzer, a language generator, knowledge bases on UNIX and English, a goal analysis mechanism, a plan formation mechanism [Faletti 82], a context modeling mechanism, and a knowledge acquisition facility.

The knowledge about UNIX is represented using frame-like structures and stored in a database managed by the PEARL package [Deering et al. 81]. Each fact about UNIX is represented as a labeled predicate-argument proposition. The literature does not give an explicit account of kinds of frames in UC. Examples of queries and those of contents of the knowledge base suggest that the majority of frames describe a plan for achieving some goal state and are used for answering the "How do I ... ?" type of question.

In this framework, there are two ways to represent causal knowledge. The explicit representation of causal relationships uses the predicate 'causation' that takes two arguments: the antecedent and consequent. The meaning of the predicate is the fact that the action specified in the antecedent argument causes a state change described in the consequent argument. Since a predicate of this type is used as an argument of another predicate, knowledge about these causal relationships is not always easy to access.

Another way causality is represented in UC is by using the following two types of frames:

(planfor (result (X)) (method (Y)))

(preconds (plan (Y)) (are (Z)))

The first frame predicate states that the plan for obtaining result X is to apply method Y. The second predicate specifies preconditions Z for executing plan (i.e., for using the method) Y. In other words, a procedure Y executed in state Z causes X to happen. The above construct is used for answering a type of query in which the user describes an unexpected result and asks for an explanation. UC first

determines the user's goal, then looks for a 'planfor' frame which contains the goal in its result slot. Once it finds a relevant frame it checks if some step in the procedure prescribed in the method slot has been omitted. If this does not give an answer, UC finally looks for the corresponding 'preconds' frame and checks whether or not the preconditions have been violated.

UC is being redesigned using a relation-oriented knowledge representation language KODIAK [Wilensky et al. 86]. In KODIAK's network of concepts, the causal relationship between the 'Cause' and 'Effect' concepts is represented by the 'Causal-Event' concept. Various aspects of the Causal-Event concept can be represented (e.g., actor, intention). This improves primarily the natural language understanding and generation capability of the system.

The explanatory power of UC, i.e., the extent to which UC can handle the 'why' type of query is unclear. The papers that give comprehensive descriptions of the system, [Wilensky et al. 84; 86], contain many examples of 'method' queries (i.e., "How do I ... ?"), but none of the 'why' type.

2.1.2. DCL

DCL [Shrager, Finin 82] developed at the University of Pennsylvania is an advisor system that helps novice users use the VAX/VMS operating system. What makes DCL interesting is the mode in which it offers advice: rather than operating in the question-answering mode, DCL follows the user's interactions with the system and volunteers advice when it finds it appropriate. DCL recognizes inefficient sequences of commands and suggests features that are more efficient with respect

to the amount of user's work (e.g., using wild-cards in file names), or system resources (e.g., renaming a file vs. a copy/delete sequence).

The system is based on a catalog of inefficient plans novice users often use to achieve common goals. The plans are represented in a network that describes sequences of commands. The nodes correspond to commands and have associated actions that are activated when the node is instantiated. The action associated with the last command in the sequence typically involves an advice message to be issued. Based on the contents of the network as a dynamic user profile, the message is tailored to the case at hand.

The DCL network represents causal knowledge by conceptual triples: a user goal, a commonly used inefficient method for achieving the goal, and a corresponding efficient method. Both methods (sequences of VMS commands) cause the same effect (i.e., the achievement of the user's goal) but with different efficiency.

The system can be primarily criticized for its shallow reasoning with the domain knowledge that cannot explain why one sequence of commands is more efficient than another.

2.1.3. TEACHVMS and TVX

The AI team at the Digital Equipment Corporation (DEC) has been developing a methodology for buiding knowledge-based operating system consultants [Billmers, Carifio 85].

The first result of this research effort has been *TEACHVMS*, an expert system for helping users familiar with the TOPS20 operating system learn VMS. This is a simple forward-chaining rule-based system implemented in C that accepts TOPS20 commands and translates them into the corresponding VMS commands. The knowledge base contains two types of rules, 'syntax rules' and 'translation rules', coded in a LISP-like language and translated into C. The (only available) paper does not give any detail about the structure of either type of rules.

TVX is a research program under development that builds on the work on TEACHVMS to provide users of various operating systems with a knowledge-based shell for consultation. The research focuses on planning mechanisms and representational issues that can provide solutions to more complex user tasks. Knowledge is represented in TVX by two major types of entities: objects (implemented by frames), and tasks (implemented by rules). The design of TVX is based on a conceptual model of a generic operating system. As a result, the knowledge about tasks is separated into generic operating system knowledge and the knowledge specific to the target operating system (e.g., VMS). TVX accepts a request that specifies the user's task (i.e, the goal). The solution to the task is planned by goal-directed backward chaining approach in which, in each step, a goal is reduced to subgoals, until the relevant target operating system commands have been identified.

The paper does not explicitly discuss causal relationships. Both types of rules, however, can be interpreted as causal rules in the sense that the actions described by subgoals or operating system commands (constituting a procedure

rather than a mere set of tasks), cause the achievement of the corresponding goal. TVX is designed to answer the "How do I ...?" type of queries. The system's capability of answering 'why' queries is limited to the explanation of its reasoning by citing rules and environment which led to the suggested use of VMS commands.

2.1.4. The UNIX Computer Consultant

The *Unix Computer Consultant (UCC)* [Douglass, Hegner 82] has been developed at the Los Alamos National Laboratory and the University of Virginia. Like UC, it is designed to aid the users of the UNIX operating system.

The UCC system consists of two major modules: the front end, and the knowledge base and solver module. The front-end module accepts a natural language query, translates it into the formal query language UCCquel, and passes the translation to the solver. A formal query is expressed in a LISP-like notation and contains unbound variables that must be instantiated by the solver to answer the original query. The front-end also generates the English version of the derived answer, based on the question type and the main UNIX action mentioned in the question.

UCC answers two types of questions: 'static' and 'dynamic'. A static question is a request for the definition of a given concept (e.g., "What is ... ?"), answered by a simple retrieval of the definition from the knowledge base. All other types of queries are considered dynamic. The UCCquel translation of a dynamic query has three components: (pre-)state R_1, action F, and (post-)state Q (a

state is represented by a first order logic formula). Depending on what subset of {R,F,Q} is given, eight types of dynamic queries are distinguished, only two of which are implemented in the initial version of UCC: "How do I ...?" (R and Q are known, but F is not), and "What happens if ... ?" (R and F are known, but Q is not).

The paper does not describe the structure of the knowledge base, nor does it explicitly discuss causality-related issues and the intended strategy for answering 'why' queries. It seems that, as in UC, the research has focused on the natural language aspect of the system.

2.1.5. PROUST

PROUST [Johnson, Soloway 84; Johnson 85], developed at the Yale University, is one of the more recent expert systems that can identify non-syntactic errors in computer programs. Its domain is novice level programming in Pascal. For each detected error PROUST suggests a likely misconception that caused it and gives advice for correcting the program. It has a knowledge base of typical programming plans, another knowledge base of programming goals, and an indexing structure that links elements of these two.

Program analysis involves a combination of shallow reasoning about plans and errors, and causal reasoning about the programmer's intentions. From a given description of the problem requirements, PROUST constructs a causal model of programmer's intentions and their expected realization. The resulting plan which consists of a combination of Pascal statements and programming subgoals is then

matched against the code. Whenever a plan fails to match the code, a set of plan-difference rules are applied. They either transform the plan to fit the code better, or come up with an explanation of the mismatch in terms of errors from PROUST's "bug catalog".

Causal reasoning about intentions and their realization enables PROUST to diagnose semantic errors in a program which is often a difficult task for a human. However, this approach requires a precise definition of the problem to be given in a language other than the target programming language. Unless a natural language is used (which is the case in examples but the authors do not claim this feature), it may be practical perhaps only for grading simple programming assignments in the classroom situation.

A very similar approach has been taken in the design and implementation of the *SCENT* advisor, a system that diagnosis non-syntactic errors in programs written by beginning-level LISP programmers [McCalla et al. 86].

2.1.6. The Programmer's Apprentice

The *Programmer's Apprentice* is an interactive knowledge-based software tool that is capable of cooperating with a programmer in various phases of the programming task [Rich, Shrobe 79; Waters 85; Waters 86]. It is being designed and implemented at the Massachussetts Institute of Technology (MIT) Artificial Intelligence Laboratory. The Apprentice differs from a typical advisor system in that it is presumably less knowledgeable than its user (e.g., it is not capable of programming by itself), but can aid the expert programmer in many ways. The basic idea

is that the programmer does the difficult parts of design and implementation, while the Apprentice acts as a junior partner and critic, keeping track of details and performing routine tasks in the design, development, implementation, and maintenance of programs.

In the Programmer's Apprentice, the description of a program consists of three components: definition of objects, input-output specifications of program behavior, and a hierarchical representation of the internal structure of the program, called a 'plan' (these are used both for describing particular programs, and for describing common programming techniques). The plan representation formalism includes not only the data flow and control flow relationships between parts of the program, but also goal-subgoal, prerequisites, and other logical dependencies. A key feature of the formalism is that it abstracts the semantics of the program by capturing essentials of the algorithm. A plan is composed of units of behavioral description, called 'segments'. A segment can either correspond to a primitive computation, or contain a subplan that describes the computation performed by the segment. Each segment is defined by its 'specs', which are a formal statement of input expectations (conditions on or relationships between input objects that are expected to hold in the input situation) and output assertions (condition that will hold for the input and output objects in the output situation). Given a plan for a program and an initial situation, the Apprentice can symbolically evaluate the plan and discover semantic errors in the program.

The literature on the Apprentice does not explicitly mention causality. 'Specs' of a segment can, however, be interpreted as follows: the application of the algo-

rithm in the input situation causes the output situation. Thus, it represents causal relationships in the form of implicit precondition-action-postcondition triples.

2.2. Other Expert Systems

2.2.1. Medicine

The Heuristic Programming Project at Stanford University has been a major center of activity in the area of medical consultation. Probably the best known expert system to date is *MYCIN*, a program that diagnoses bacterial blood infections and recommends appropriate therapies [Shortliffe 76; Buchanan, Shortliffe 84]. Its knowledge base contains a homogeneous set of rules that encode the clinical judgement of an expert consultant. MYCIN does not represent causal relations explicitly. The causal models have been 'compiled' into associational rules. Even though MYCIN achieves high performance in consultation, its 'shallow' knowledge represents a severe limitation for explanation and justification of its reasoning. The explanatory facility for MYCIN is provided by *TEIRESIAS* [Davis 76], a knowledge acquisition system built on top of MYCIN. At any point of a MYCIN consultation session the user may ask 'why' and 'how' question and TEIRESIAS will explain the chain of reasoning by displaying a trace of the execution. This kind of explanation may be sufficient for consultation but is by no means adequate for teaching purposes where references to basic concepts and causal relationships among them are essential.

NEOMYCIN [Clancey, Letsinger 81] is a consultation system in which MYCIN's knowledge is restructured and extended for use in teaching. It has been

designed as a psychological model of a medical expert's reasoning process. The most significant achievement was the separation of various kinds of knowledge (e.g., diagnostic strategy, factual knowledge). Causal relationships are explicitly encoded as causal rules which are modified by a certainty factor, like MYCIN rules. Causal rules form a network of pathophysiological states and disease categories. Pathways through the network link observations with a taxonomy of diagnostic hypotheses. This provides a basis for justification of rules in terms of underlying causal processes.

A group of scientists at Rutgers University have developed a method of computer-assisted medical consultation based on a causal-associational network, *CASNET*, as the model of a disease [Weiss et al. 78]. The approach focuses on cause-effect relationships and temporal sequences of events. The network model contains three types of nodes: observations (the evidence obtained about the patient), pathophysiological states (summary descriptions of events that are deviations from normality), and diagnostic/therapeutic categories of diseases (summarized patterns of states and observations).

CASNET represents causal knowledge in the causal network of states, a part of the causal-associational network. The causal network is defined by a set of states, and a set of links between states that represent the relationships. A causal link between two nodes has an associated strength that is interpreted as the frequency with which the first node causes the second. States are also linked with observations and disease categories. Those links, however, are associational rather than causal.

The primary reasoning mechanism in CASNET uses propagation of probability weights through the network after the value for an observation is obtained. As a result of this process the status of a state may be 'confirmed', 'denied' or 'undetermined'. Each pathway through confirmed nodes in the network represents a model of how the disease evolves.

CASNET knowledge representation is adequate for certain medical areas (e.g., the glaucoma domain to which the method has been successfully applied) and other areas which are well understood and do not rely heavily on judgmental knowledge. The explicit representation of causal relationships provides for justification of results. However, the system does not have the ability to reason at different levels of detail, a feature much desired for applications in less understood areas, as well as in advisory or tutoring systems.

INTERNIST/CADUCEUS [Pople 77] is a large expert systems for internal medicine which has the knowledge representation similar to that of CASNET, but in which the use of causal relationships is limited to the propagation of likelihood measures.

An important characteristic of the work done by MIT researchers in recent years is the emphasis they put on the ability of an expert system to explain and justify its reasoning by referring to domain principles stated at an appropriate level of detail.

Swartout has developed the *XPLAIN* system [Swartout 81; Swartout 83] which provides an explanatory facility for the Digitalis Therapy Advisor [Gorry et

al. 78]. He took a novel approach of using an automatic programmer to produce a structure of background knowledge while creating the advisor program. Domain knowledge is separated into a body of domain principles (abstract procedures) and a domain model (facts). Causal relationships are represented within the domain model in a fashion very similar to that used in CASNET, except that the causal links are not weighted. Based on the domain knowledge, the writer generates the advisor by refining the top-level goal ('administer digitalis') and leaves a trace of this process in a tree of goals, called the 'refinement structure'. Justification of the advisor's actions is obtained by examining relevant parts of the refinement structure. This approach provides justification based on causality and domain principles. However, it involves the complex area of automatic programming.

A system that is considered to be the state of the art in medical knowledge representation [Clancey, Shortliffe 84] is *ABEL* [Patil 81; Patil et al. 81; Patil 82] developed by Ramesh Patil and his colleagues of the MIT Clinical Decision Making Group. This system for Acid-Base and Electrolyte disturbances describes a patient in a multilevel causal network called 'patient-specific model' that includes both patient data and developing hypothetical interpretations of the data. Causal concepts are symbolically manipulated on multiple levels of detail in order to take advantage of as much of the available knowledge as possible. Each level of the network provides a complete statement about the patient's illness and can be viewed as a semantic net of relations between diseases and findings. At the lowest level the description is given in pathophysiological terms, then gradually aggregated into higher-level concepts and relations, with the syndromic knowledge at the top level. Some nodes (states) are defined in terms of a causal network of states at

the next more detailed level of description. These are called 'composite nodes' as opposed to the unexplored, 'primitive' ones. The causal network defining a composite node is called its 'elaboration structure'. A node in the elaboration structure which identifies its essential part is designated as the 'focus node'. Various levels of the patient-specific model are aligned through such nodes.

An important novelty introduced in ABEL is the way causal links are treated. Patil and colleagues have found the simple representations of causal links present in previous systems inadequate to express all aspects of a cause-effect relationship (e.g., severity, context). They decided to represent causal links as objects, giving the system the ability to manipulate and reason about them. Like nodes, links can be categorized as primitive and composite. The result of elaborating a causal link is a causal pathway at the next level of detail.

Another important feature of ABEL is its ability to reason about the effect of more than one cause. This is achieved by component summation and decomposition: any primitive node in the hierarchy may have a group of other primitive nodes as its components. This provides a basis for the projection operation by which the existence of a causal link is anticipated in order to explain an unaccounted for component.

A patient-specific model is created by instantiating portions of ABEL's body of medical knowledge. This process starts with and is guided by an initial set of patient data. Further instantiation within a single level of the description is accomplished by component decomposition and summation, and projection operators. Connections between levels are established through the aggregation and elaboration

(focal and causal). The result is a model of patient's illness described at various level of detail with causal consistency of the descriptions being enforced by the operators.

Another medical expert system that deals explicitly with causality is *CAA* [Shibahara et al. 83; Shibahara 85]. Its domain is the interpretation of electrocardiograms. Several types of causal links have been introduced to represent causal relationships among underlying physiological events. A causal link includes the existential dependency and the implicit temporal constraint between the effect and the cause events. Concepts such as events are described by frames which have the 'causal-link' slots.

2.2.2. Electronics

There are a number of expert systems for diagnosing hardware failures that are designed using approaches quite similar to that of MYCIN (e.g., *DART* [Bennet, Hollander 81], *ARBY* [McDermott, Brooks 82]). These will not be presented here since reviewing them would not contribute any further to our investigation.

The study of causality involved in electronic circuits in general is based on notions of locality and directionality. Each component performs an action when a signal reaches it from one of its neighbors, and then it passes along the signal to its neighbors according to the component model. The signal propagation has one and only one direction at each component. Based on this a 'causal argument' for an output of a circuit can be constructed as a sequence of steps that link it with the input.

EQUAL [de Kleer 84], developed at Xerox PARC, is a system which constructs a description of the mechanism by which an electronic circuit operates from the structure of the circuit. The process is based on the qualitative causal analysis of the circuit which describes how the behavior of the individual components can explain the behavior of the composite system. EQUAL takes as input the schematic for the circuit and produces an output that includes a qualitative prediction of the circuit's behavior with explanations. It uses a knowledge base of component models which describes the behavior of every type of component. A qualitative prediction of the circuit's behavior (including an explanation for that behavior) is represented as a table in which rows correspond to steps and which has the following four columns: a list of antecedent steps, an indicator of whether the step is a premise (e.g., an input disturbance) or it follows directly from its antecedents (default for an antecedent is the previous step), a description of an event that assigns a value to a particular variable, and an explanation (the reason) in terms of the underlying theory applied to the component models. An important characteristic of this system is the modeling of causal relationships by qualitative descriptions (e.g., increasing, unchanging) for circuit attributes which are qualitative in nature. Since this kind of a model corresponds to the intuitive qualitative reasoning electrical engineers use when they analyze circuits, it gives an appropriate basis for explaining the circuit's behavior.

The Hardware Troubleshooting Group at MIT has suggested an approach to fault diagnosis based on reasoning from first principles [Davis et al. 82; Davis 83]. A specification of an electronic module consists of its structural description (in terms of physical as well as functional organization), and a behavior description in

the form of rules interrelating the information at the module's ports. Two kinds of rules are distinguished. The first kind, 'simulation rules', captures knowledge about the electrical behavior of a device (e.g., a simulation rule for an OR gate: "If either input is 1, then the output is 1, else the output is 0.") This kind of rule models physical causality within the device. The other group of rules, 'inference rules', capture conclusions we can make about the inputs of the device given its output. (e.g., "If the output is 0, then both inputs must have been 0.")

The approach to diagnostic reasoning combines simulation and inference: simulation rules and inputs are used to derive expectations about correct behavior (forward reasoning) while inference rules applied to the obtained outputs generate conclusions about actual behavior (backward reasoning). Troubleshooting is based on the analysis of differences between the obtained models of expected and actual behavior of the device.

The AI/VLSI group at Rutgers University has developed *CRITTER* [Kelly, Steinberg 82], a system that reasons about digital circuits by propagating information through their models. The schematic for a circuit is represented by 'modules' and 'data-paths'. A module consists of a single component or a cluster of components that constitute a functional unit. A data-path is a wire or a group of wires. Unlike in many circuit simulators which represent a single data value at a single time, in CRITTER the entire history of data flowing on a data-path is represented as an infinite sequence of data elements called a 'data-stream'.

The knowledge about the behavior of a data-stream is represented as a frame with slots corresponding to attributes of data elements. Each slot contains an

algebraic formula that gives the value of the feature as a function of the element's subscript.

A module of a circuit is described by 'operating conditions' and 'mappings'. Operating conditions are a set of predicates involving a module's inputs. Mappings are formulas that describe causal relationships between features of the output and the module's inputs. CRITTER uses these models to reason about a circuit in either of the following two directions: it can derive the behavior of the outputs given the behavior of the inputs or, given the behavior of the outputs it can derive specifications of the module's inputs. In the former case, which is based on forward propagation, CRITTER symbolically applies the module's mappings to the inputs by a process of substitution and checks whether the operating conditions are satisfied. In the latter case, backward propagation is performed and the operating conditions are a part of the propagated information.

Causal knowledge from CRITTER is used in a knowledge-based system *REDESIGN* [Mitchell et al. 83] which assists in the redesign of digital circuits to meet altered functional specifications, as well as in a VLSI design advisor *VEXED* [Mitchell et al. 84].

2.2.3. General

CLEAR [Rubinoff 85] is a system that provides explanations of concepts based on information contained in rules. It is part of a front end for expert systems being developed at the University of Pennsylvania. This system is based on a taxonomy of rules, and on roles a concept can have within a rule. For a given

concept that need be explained, the relevance of each rule that uses the concept is determined through a specific ranking schema. A natural language description of the contents of selected rules is displayed to the user as an explanation of the concept. The relevance of this work to our research is not its purpose (CLEAR answers 'what' rather than 'why' questions), but the fact that it recognizes cause-effect rules as a major type of rule that an expert system may have, regardless of its domain.

2.3. Software Engineering

An important issue in software testing is the design of effective sets of test cases. One of the techniques that aid this process is *cause-effect graphing* [Myers 79; Pressman 82].

Various input conditions defined by the specification of a piece of software are viewed as 'causes' of corresponding actions of the system ('effects'). A cause is a single input condition (e.g., "First input token is 2") or an equivalence class of input conditions (e.g., "First input token is numeric"). An effect can be either an output condition (e.g., a message displayed) or a system transformation, i.e., an effect on the state of the system. Relationships between causes and effects are represented in a Boolean logic network called the 'cause-effect graph'.

The systematic generation of test cases using the cause-effect graphing technique is performed through the following sequence of steps:

a) Causes and effects are listed and an identifier is assigned to each.

b) The cause-effect graph is developed based on the analysis of the semantic content of the specification.

c) Possible constraints on combinations of causes and/or effects are added (e.g., two conditions that cannot hold simultaneously).

d) The graph is converted into a decision table having a row for each cause and each effect. This is done by methodically tracing state conditions in the graph.

e) Columns of the decision table are converted into test cases.

This technique is considered very important in software testing because, unlike other well known techniques (e.g., equivalence partitioning, boundary value analysis, etc.), it explores combinations of input circumstances. Also, the process of building a graphical representation helps in detecting incompleteness and ambiguities in the specification. All this indicates that the cause-effect graphing can be a useful tool for knowledge enginering as well.

2.4. Concluding Remarks

We will end this review of the literature by summarizing methods used for representing causal knowledge and reasoning with it. The relevance of these approaches to our research will be indicated.

We have encountered several knowledge representation paradigms used for representing causal relationships:

* **Causal Rules.** Encoding knowledge in some kind of a rule format (e.g., antecedent --> consequent) has been the most commonly used representation paradigm in expert systems. In some systems, causal relationships are mixed with other kinds of knowledge and compiled into associational rules (e.g., the MYCIN family). The type of explanation these systems can give is limited to a display of a trace of their execution. In other rule-oriented systems, causal knowledge is represented explicitly by causal rules (e.g., NEOMYCIN, TVX, CLEAR, MIT Troubleshooting Project). Causal rules (like any other rules) may be assigned certainty factors (e.g., NEOMYCIN). They are interpreted as the strength of causal relationships the rules represent. The primary reasoning mechanisms are forward and backward chaining of rules. A chain of reasoning with causal rules not only explains how the system has reached its conclusions, but also provides a justification of the conclusions in terms of the underlying domain principles. This is crucial for answering the 'why' type of questions.

* **Causal Network.** In some systems, causal knowledge is modeled as a network: system states are represented by nodes, and causal relationships are shown as links between nodes (e.g., CASNET, ABEL, XPLAIN). This is a means of representing an overall conceptual model of causality in a given domain. Individual fragments of knowledge contained in the network are encoded using an appropriate formalism, usually rules. Causal network may be an integral part of a larger network that includes other types of knowledge (e.g., CASNET). Causal network may have

one or more levels. A multi-level approach (e.g., ABEL) provides for reasoning at different levels of detail, which is one of the features our system is expected to have.

* **Frames.** This is another general paradigm that has been used for representing causality. Causal relations can be represented by a single frame or a group of frames (e.g., UC - planfor, preconds), or as slot-fillers in a general frame (e.g., CRITTER, CAA, UC - the causation predicate). The frame approach is well suited for representing knowledge in our domain since a large portion of the body of knowledge can be partitioned by QUIZ statement types.

As the rest of this thesis will show, our research combines the advantages of the above paradigms which results in a hybrid representation of knowledge about QUIZ.

3. WHY? BECAUSE...

In order to determine the representation and role of causal relationships in the Advisor, we will first examine various categories of questions novice users typically ask, and analyze types of evidence users can provide within 'WHY' questions. Based on these findings, a uniform structure of the 'WHY' type of query and a general query-answering strategy will be presented.

3.1. Questions Which Programmers Ask

As a first step toward designing the QUIZ Advisor, the research team[+] conducted an analysis of real inquiries submitted to Cognos consultants by users of QUIZ. In a corpus of 2000 phoned-in and mailed-in inquiries about QUIZ we selected those that were technical (rather than requests for documentation and alike), general enough for their answers to be applicable to a class of similar specific problems, and categorized by consultants as beginner-level queries. This process resulted in a sample of approximately 200 inquiries. Since in many cases the question and/or answer was not fully or clearly recorded, most of them were rephrased in simple and unambiguous English, before any further analysis of the sample could be performed.

The methodology used in constructing the set of sample questions and determining the classification of questions is described in [Szpakowicz et al. 86] and

[+] The two-year project was started in August of 1985 with a mixed team of Cognos research group members (2) and University of Ottawa professors (3), graduate students (2 including the author of the thesis who is also a Cognos employee) and part-time assistants (2).

[Constant et al. 87].

3.1.1. Taxonomy of Questions

A major result of the analysis of about 200 selected and clarified real questions was taxonomy of questions, based on the syntax of the questions and the assumed intentions of the user. The following six categories were distinguished:

HDI (How Do I) or *method* questions, requesting QUIZ code for accomplishing certain task. A typical syntactic pattern of these questions is the following:

"How do I <verb phrase> ?"

The questions of this type constitute the majority of the sample (over 50%).

WHY or *causal* questions are requests for explanations of some unexpected behavior of the system. Most of these questions are affirmative, e.g.,

"Why am I getting the '#' sign
when printing negative numbers?"

but there are some negative ones as well, e.g.,

"Why am I not getting all records reported?"

About 20% of the sample questions fall into this category.

SYN or *syntax* questions are inquiries about syntax and static semantics of QUIZ statements and their components, e.g.,

"Where is file declaration used?"

This is the third most frequent category (13%).

ERR or *error* questions, which are requests for explanations of error messages issued to the user, e.g.,

"Why did I get the error message 'Expected: file IF eol'?"

Although these questions, considering their syntax, appear to be a subcategory of 'WHY' questions, they are sufficiently different from general 'WHY' questions to be declared a separate category in the taxonomy. They refer to explicit errors and a well defined environment. Error questions account for 7% of the sample.

HYP or *hypothetical* questions, which are requests for predictions regarding the use of some feature of QUIZ, e.g.,

"What happens if I use REPORT ALL?"

This category of questions constitutes about 4% of the question set, which is lower than one would expect in a sample of beginner-level questions. An explanation for this is the fact that the user often decides to run the QUIZ code to find out what it does rather than make a long-distance telephone call to ask a Cognos consultant. Should a

running Advisor be available to the user, it is anticipated that this category of questions would be much more common.

DEF or *definition* questions, asked when the user needs clarification of QUIZ terminology, e.g.,

"What is an alias?"

Approximately 4% of questions fall into this category.

3.1.2. 'HDI' and 'WHY' Questions

Upon the analysis of questions, the Advisor team decided to focus on 'HDI' (method) and 'WHY' (causal) questions, the two most frequently asked categories. Developers of some other advisor systems have also recognized these two categories as the most important ones [Swartout 83].

In the course of producing a report, typically before and while writing a program, a QUIZ programmer asks 'HDI' questions in order to learn a procedure or strategy for achieving desired results. Once a program has been written and executed, a mismatch between the expected and obtained results would be a reason for asking 'WHY' questions. What the user needs in such a case is an explanation of results in terms of their causes. In order to successfully answer 'WHY' questions, the Advisor should be based on a model of QUIZ that explicitly represents causal relationships between various components of the system. While essential for answering 'WHY' questions, causal models usually do not comprise the kind of knowledge required for answering 'HDI' questions. They can be used in

answering some 'HDI' questions (e.g., an advice to use a particular QUIZ statement or option), but in most cases what the user is looking for is a precisely described procedure or a sketch of an actual QUIZ program.

Since 'HDI' and 'WHY' types of questions are different in many ways, the Advisor project team decided to work on each of them independently. This resulted in the development of separate knowledge representation and question-answering methodologies, and, eventually, two separate subsystems of the Advisor prototype. This approach allowed the team to experiment with a variety of paradigms and take advantage of specific characteristics of the two question types. This thesis deals with the causal component of the Advisor and answering 'WHY' and other causality-related categories of questions. 'HDI' questions have been explored by a team led by D. Skuce. Their results are described in [Skuce 86], [Szpakowicz et al. 86], [Matwin et al. 86], and [Stanley 87].

3.2. Query: Question with Background Information

In the remainder of this section, we will analyze 'WHY' questions from the sample in more detail, focusing on types of information they contain. Then we will compare these with the general types of evidence the user can provide in an environment in which 'WHY' questions about QUIZ (or any similar software package) are likely to be asked. This exercise will result in a general format in which questions will be submitted to the Advisor.

3.2.1. A Closer Look at 'WHY' Questions

The analysis of 'WHY' questions and the corresponding consultants' answers in the observed sample (a total of 20 question/answer pairs) has revealed the following:

a) All questions included a description of an unexpected feature of the report, e.g.,

"Why am I not getting all records reported?"

This can be used as a pointer to an area of QUIZ about which the user has some misconceptions.

b) In 70% of questions the user made an implicit hypothesis about a causal relationship between an aspect of the report and the QUIZ code, e.g.,

"Why am I getting '#' sign when printing negative numbers?"

As a consequence, the question-answerer must be able to establish a causal link between the features of QUIZ code and the features of reports. This type of evidence is very useful provided the two components are indeed causally related.

c) In 35% of cases a piece of actual QUIZ code (one or more statements, some partially specified) was given as evidence, e.g.,

'Why does a *SELECT elem NE "123199"* not work, where 'elem' is a date?'

In order to take advantage of this type of evidence, the Advisor must be able to parse and analyze the QUIZ code (in the above example, the code is shown in italics).

d) The answer to 30% of questions was a consultant's assumption about the actual code (e.g., "The statement/option ... is used"). This indicates that some 'WHY' questions are properly answered by simply naming the component of the code that is the cause of the effect in question.

e) In 25% of questions the description of report features was extremely vague, e.g.,

"Why are my page skips off?"

These questions must be rephrased and submitted to the Advisor in some appropriate, unambiguous form.

f) In 10% of questions the user requested an explanation for a false assumption, e.g.,

"Only one copy of a report was printed
although the SET REPORT COPIES was 80. Why?"

In fact, the remaining 79 copies of the report were printed but on a different printer. This indicates that the Advisor must be able to detect and appropriately deal with possible contradictions hidden in questions!

All of the above discussed requirements of the causal component of the QUIZ Advisor have been taken into account in the design of the system.

3.2.2. Available Evidence

In general, a question is asked in a situation when there is an awareness of a lack of knowledge and an intention to acquire it. The user typically asks a 'WHY' question when he/she obtains some unexpected result from a QUIZ program. The reason for obtaining a result that differs from the expected one is an error that may be introduced at any stage of the report generation process shown in Figure 3.1.

We will be interested only in those errors that originate from the user's misconceptions about QUIZ. By answering 'WHY' questions the Advisor should help correct the user's mental model of QUIZ.

Since 'WHY' questions are normally asked upon the execution of a program, i.e., at the end of the report generation process, there are four kinds of information that the user may provide as evidence.

a) *What is WANTED*. The question sample suggests that this type of information is given implicitly. For example, when asking the following question:

"Why did I lose some digits when ...?"

the user implicitly states that, instead of the number obtained in the report, he/she expected to see a longer number, i.e., a number that contains more digits.

b) *What is SPECIFIED*. This is complete or partial code that produced the report, or a narrative description of some component of the code.

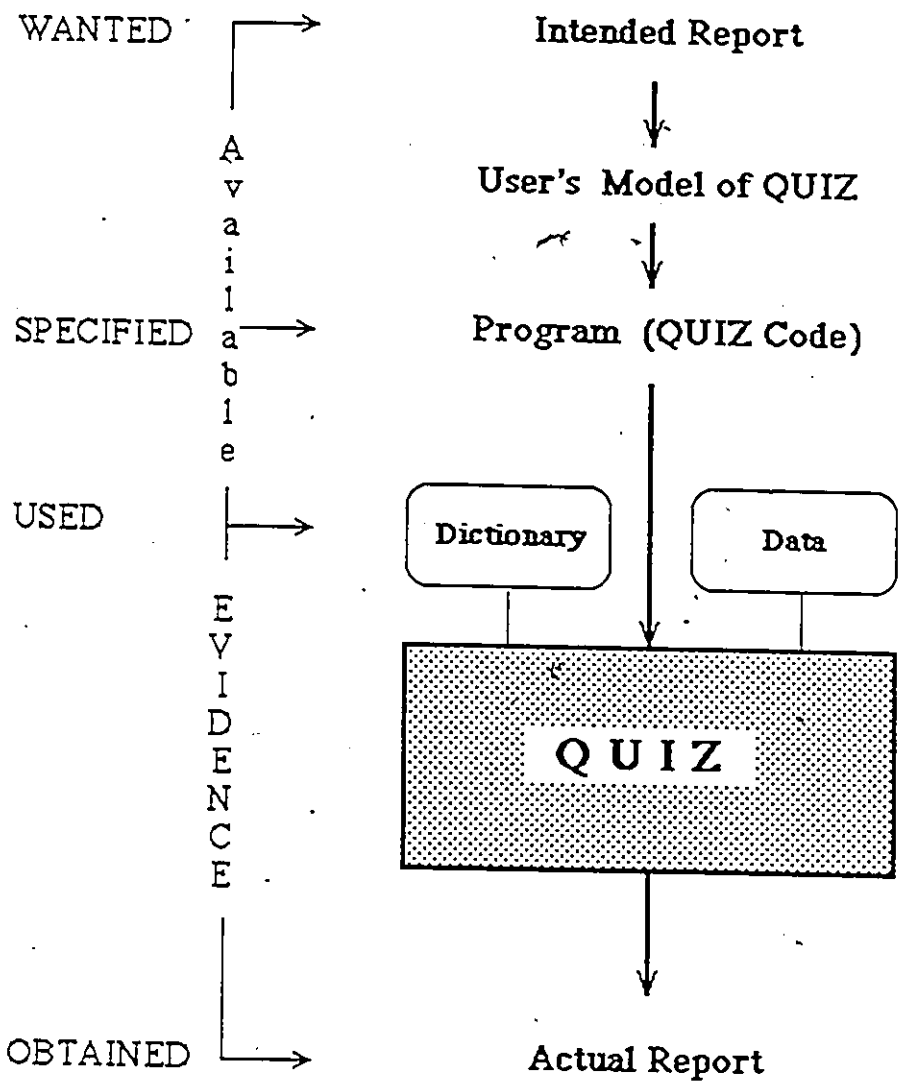


Figure 3.1. The Process of Creating a Report

c) *What DICTIONARY and DB were used.* In the process of generating a report, QUIZ accesses a specific data dictionary and the data base. The same code produces different results when the information in either of these is changed. For example, in:

"Why ... when printing negative numbers?"

the last part of the question provides this kind of information.

d) *What is OBTAINED.* This is evidence about features of the actual result. It often includes an explicit description of an unexpected report feature (e.g., the negation of what was wanted).

The above types of evidence are normally used to describe the problem when the 'WHY' questions are asked. The description of the puzzling feature of the result, the one the user needs an explanation for, must be specified. Each additional component of the problem description helps in building a better, more accurate model of the problem situation.

3.2.3. Queries

The discussion of factual evidence available from the user, and the analysis of the question sample suggest that the information contained in any 'WHY' question should be separated into two components: *background information* and the *object of interrogation.*

A long, complex question (like some of the examples we have encountered in

the sample) can be rewritten into a (possibly empty) sequence of simple affirmative statements and a simple, very focused question. The resulting set of statements will be termed the *NL-query*.

The set of affirmative statements in an NL-query describes the background or context of the question. Some of these statements may contain actual QUIZ code used to generate the result, e.g.,

"I used ACCESS A LINK TO B"

A portion of the QUIZ program that is specified this way will be called the *QUIZ code*. Other affirmative statements form a set of assumptions that describe various aspects of the context, such as features of the obtained report, and characteristics of the data dictionary and the database used. Each of these will be called an *assumption*.

The interrogative statement in an NL-query defines the object of interrogation, the aspect of the obtained result that the user needs an explanation for. This will be called the *question*. In order to avoid ambiguity, what we have been calling a question previously, from now on will be referred to as the *original question*.

In summary, instead of asking complex and often ambiguous original questions, the user will submit to the Advisor *queries* of the following structure:

QUERY = QUESTION + BACKGROUND

BACKGROUND = ASSUMPTIONS (optional)
+
CODE (optional)

For example, from the original question

'Why does a *SELECT elem NE "123199*" not work, where 'elem' is a date?'

we can construct the following query:

CODE:	SELECT IF ELEM NE "123199"
ASSUMPTIONS:	'ELEM' is an item; the type of 'ELEM' is date
QUESTION:	Why does the code not work?

More examples of queries can be found in Chapter 6 and Appendix B.

In addition to facilitating the process of answering a single original question, the separation of the question from the background information has a very important practical aspect: the Advisor can be designed in such a fashion that the background information applicable to more than one original question need be specified only within the first of the corresponding queries.

3.3. Answering Queries

As described above, an element of a query that will be accepted by the Advisor will be expressed either as a descriptive statement or a QUIZ statement, whichever is appropriate. In order to take advantage of as much information contained in a query as possible, the system needs the ability to explore individual elements

of the query, as well as to reason about the state of affairs defined by the query as a whole. Therefore, we need a knowledge base design and reasoning mechanisms that will enable the query-answerer to establish a link between any two elements of the query, if there is a causal relationship between them.

The next chapter proposes a combination of rules and frames as an appropriate knowledge representation paradigm. Forward and backward chaining of rules will be the primary reasoning mechanisms used to explore QUIZ code and descriptive statements, respectively. The trace of this reasoning left in a dynamic database will be an internal model of a query to which an appropriate answering strategy will be applied.

4. REPRESENTATION OF CAUSAL KNOWLEDGE

In this chapter we will focus on two aspects of QUIZ, programs and interactive sessions, and propose methods for representing causal knowledge about each of them. The emphasis will be placed on modeling programs, since this aspect is chosen as the domain of the QUIZ Advisor system.

4.1. Model of a QUIZ Program

The existence and characteristics of each statement in a QUIZ program have some effect on the appearance of the resulting report. In this section we will present a formalism for representing this kind of generic knowledge about QUIZ. We will build a multi-layer network of causal relationships and represent each layer by a set of rules of a special type.

In order to build rules that reflect causal relationships in QUIZ, we need some means of formally defining features of both QUIZ statements and reports. We have developed a metalanguage which not only describes the syntax of a statement but which can also provide a basis for reasoning about its components (e.g., reference to the statement's arguments, specification of conditions). We did not necessarily have to develop a new metalanguage. There are well known formalisms, like *attributed grammars* (see, for example, [Lewis et al. 78]), that would allow us to fully express the syntax-related knowledge. However, since we do not want to represent and use this portion of knowledge in isolation, but want to integrate it with causal knowledge, many conventions of such formalism would force us to compromise the clarity of the knowledge representation. For example,

should we use the standard rule format to represent the grammar, we would have to adopt a less natural format for causal rules in order to clearly distinguish the two types of knowledge. Also, to encode knowledge in an easy-to-understand form, we need simpler and more convenient ways to represent certain types of components of statements (e.g., optional and repeated entries) than those that standard formalisms offer.

The developed metalanguage is described in the next section. A language for describing characteristics of reports will be described later.

4.1.1. Augmented Syntax

In developing a metalanguage for describing the syntax of QUIZ, we followed the conventions from the QUIZ manual as much as possible. In addition to conventions for describing syntax, we introduce some new features that are required for reasoning. The metalanguage is called *the Augmented Syntax Language (ASL)*. The syntax of a QUIZ statement expressed in this language is called its *augmented syntax*.

4.1.1.1. Basic Metalanguage

ASL includes a set of conventions which are commonly used for describing the syntax of programming languages. The following is the summary of syntax conventions used in the QUIZ manual and the corresponding features of ASL:

a) *Keywords* are shown in uppercase in the manual as well as in the

metalanguage (e.g., ACCESS).

b) *Arguments* are words other than keywords. The manual does not have clear terminology for these ('general terms', 'terms', 'general forms' are used but it is not clear whether they are synonymous). An argument is shown in lowercase in the manual (e.g., 'file'). In ASL it is represented using a special structure which will be described in more detail in the next section. Depending on whether or not an argument is described as consisting of lower level syntactic components, it is categorized as *non-terminal* or *terminal*, respectively.

c) *Optional entries* are parts of a statement that may or may not appear at a given point in a statement. They are enclosed in square brackets ('[' and ']') both in the manual and in ASL.

d) *Repeated entries* are parts of a statement that may have zero or more occurrences. The manual does not state explicitly that the no-occurrence case is included but the examples suggest that this is the case. There, repeated entries are enclosed in square brackets followed by three dots ('[]...'). In ASL, square brackets preceded by an asterisk ('*[]') are used[+]. Repeated entries that have one or more occurrences are prefixed by an additional asterisk ('**[]').

[+] Prefix notation is more convenient for left-to-right processing of a syntax expression since it helps to earlier distinguish repeated entries from the optional ones.

e) *Alternatives* are mutually exclusive possibilities for contents of an entry, one of which should be selected. In the QUIZ manual alternatives are listed one by one, separated by slashes ('/'). In ASL notation they are separated by commas and enclosed in brackets prefixed by a vertical bar ('|[_,_, ... ,_]').

4.1.1.2. Additional Features

As already stated, the conventions adopted in the previous section are sufficient to describe the syntax of QUIZ. In this section we will introduce features that, when integrated into the basic syntax language, give additional capabilities for reasoning. By convention, most of these extensions are written in braces ('{' and '}'). The following features are introduced:

a) *Variables* which are words written in uppercase and prefixed by a question mark when unbound (e.g., '?FILE'). References to bound variables are prefixed by a period instead of a question mark (e.g., '.FILE'). Variables are introduced to link syntax expressions with the internal model of a QUIZ report. Some variables take an integer subscript for convenience in dealing with repeated entries. In deduction at the intensional level[+], variables are treated as proper names. When knowledge about syntax is used to parse and analyze actual QUIZ code, the procedural semantics of an occurrence of a variable that corresponds to a terminal argument is an

[+] By deduction at the intensional (generic) level we mean reasoning with uninstantiated variables. When all the variables are instatiated, the reasoning is done at the extensional (specific) level.

instruction to instantiate the variable to the next input token.

b) *Subscript control* is a mechanism for associating a subscript with one or more variables. This feature is used within a repeated entry as shown in the following simplified example:

*[{Sub: n from 2} LINK TO ?FILE]

This construct would assign a subscript to the variable FILE starting with 2 and increase it by 1 for each subsequent occurrence of the repeated entry.

c) *Type constraints* are used to impose type restrictions on terminal arguments. The name of an argument that is commonly used by users of QUIZ and in the QUIZ manual does not necessarily carry enough semantic information. The notation from the manual is, however, convenient for the user when referring back to the book. Therefore, a terminal argument is represented by the following notation that also includes a variable and a constraint:

{ name : variable / type}

where 'name' is, normally, the same as the name of the argument that appears in the manual (it is used only for communication with the user); 'variable' is the variable that corresponds to the argument; and 'type' is a type constraint. For example:

{item:?ITEM/item_name}

describes an argument given the QUIZ (external) name 'item' that may be instantiated only by a valid item name, and which will be represented by the internal name ITEM in the knowledge base. When the name and the variable are represented by the same word, as in the above example, the name is omitted to increase readability:

{?ITEM/item_name}

Nonterminal arguments do not need explicit type restrictions since they are defined through their components.

d) *Counts* are used to denote the number of occurrences of repeated entries. A count is placed immediately after a repeated entry, in the following format:

*[repeated_entry] {'Count': variable }

e) *Choices* are used to denote the chosen alternative. It is placed immediately after the alternatives as follows:

|[alternatives] {'Choice': variable }

f) *Conditions* are descriptive statements enclosed in braces. They impose additional restrictions (other than type constraints) on one or more arguments of a given QUIZ statement. For example, the condition:

{?QUAL is a declared file.}

states that the qualifier of an item must be a declared file, i.e., a file

declared in the 'ACCESS' statement. The way descriptive statements are represented will be discussed in Section 4.1.3.

g) *Statement terminator* {end} is used to explicitly terminate the augmented syntax of a statement. Each QUIZ statement starts with a keyword, usually a verb, which implicitly terminates the previous statement. The explicit terminator is particularly useful to avoid ambiguity when representing restricted cases rather than the full syntax of a statement.

4.1.1.3. Syntax of ASL

A formal definition of ASL is given in the Backus-Naur Form (BNF) on the next page (Figure 4.1). Definitions of keywords and names of arguments can be found in the QUIZ manual. Variables were described earlier in this chapter. Some components (e.g., 'lowercase_word') are given self-explanatory names and are not formally defined.

4.1.1.4. An Example

The use of most of the ASL features will be illustrated by the definition of the ACCESS statement restricted to simple linkages. The QUIZ manual notation of its full syntax is:

```
ACCESS file1 [ALIAS name1]
    [LINK TO file2 [ALIAS name2] [OPTIONAL]]...
```

The following is the syntax of the same statement expressed in ASL:

```
statement      =  keyword  [tokens]  '{end}'

nonterminal    =  nonterm_name  '='  tokens

tokens         =  token  [tokens]

token          =  option | loop | alternatives |
                  term | condition | keyword

option         =  '[' tokens ']'

loop           =  '*' '[' loop_control tokens ']' count

alternatives   =  '|' '(' token ',' alt_list ')' choice

alt_list       =  token   [ ',' alt_list]

term           =  nonterm  |  terminal

nonterm        =  '{' nonterm_name ':' variable '}'

terminal       =  '{' [argument_name ':']
                      variable '/' type '}'

condition      =  '{' 'Cond:' declarative-statement '}'

loop_control   =  '{' 'Sub:' subscript_variable
                          'from' integer '}'

count          =  '{' 'Count:'  variable  '}'

choice         =  '{' 'Choice:'  variable  '}'

subscript      =  subscript_variable  |  integer

subscript_variable   =  lowercase_letter

type                 =  lowercase_word
```

Figure 4.1. The Syntax of ASL

```
ACCESS {first_file_declaration:?FILE(1)/file_name}
        *[{Stub: n from 2} {LINK_TO_part:?FILE}] {Count:?NSF}
{end}

  first_file_declaration  =  file_declaration

  file_declaration  =
        {file:?DFILE/file_name}
        {Cond: .DFILE is described in the dictionary as a file}
        [ALIAS_part]
        {Cond: .FILE is a unique file}

  ALIAS_part  =  ALIAS {name:?ALIAS/file_name}

  LINK_TO_part  =  LINK TO  subsequent_file_declaration

  subsequent_file_declaration  =  file_declaration [OPTIONAL_part]

  OPTIONAL_part  =  OPTIONAL
```

Multiple lines in both syntax representations can be thought of as being concatenated, with blocks of blank characters, tabs, and line-breaks reduced to a single blank (except when within a string).

4.1.2. Internal Model of a Report

Given the language for describing statements (ASL) and a language for describing features of results (discussed in Section 4.1.3) one could write causal rules directly linking QUIZ statements and resulting reports, but this would not allow reasoning at different levels of detail, nor would it be powerful enough to express many subtleties of QUIZ. A reasonable solution is to develop an internal model of a report in such a way that it can be related to, both from the program (input) side and the report (output) side. This also corresponds to the actual imple-

mentation of QUIZ and its two-phase character of program execution (parsing and reporting).

One way to create the internal model of a report could be to base it on the actual method used in the QUIZ implementation, which builds a report as a set of internal tables that contain various report parameters. We have not done this for two reasons. First, since the Advisor differs significantly from the QUIZ itself, both in its scope and its purpose, it is likely that the implemented model would be, in some respect, much more complex than necessary for the Advisor (for example, we are not concerned with those parameters that have been introduced for the sole purpose of increasing the efficiency of QUIZ). Second, even though functionally effective, such a model might lack explicit parameters that would model complex concepts in an easy-to-understand fashion.

Our internal model of a report is a set of parameters that correspond to various objects related to a QUIZ report. Some parameters are imposed by the syntax of QUIZ (e.g., a parameter that corresponds to a variable in the augmented syntax of a statement or option), others are added by the knowledge engineer to reflect the remaining aspects of the report (e.g., the number of report items).

For a given assignment of values to parameters, the internal model defines the structure of the report (the actual appearance of the report depends also on the content of the database). Knowing the role of each parameter in the model, we can reason about the structure of the report even when values of some or all parameters are unknown.

4.1.2.1. Transparent vs Hidden Parameters

There are two kinds of parameters in the model: transparent and hidden. A *transparent parameter* or a *T-parameter* is a parameter the user is expected to be aware of. Such a parameter satisfies at least one of the following three conditions:

a) it corresponds to an argument of a QUIZ statement;

b) it may be used in the description of a report;

c) it relates to the QUIZ environment (e.g., database or data dictionary).

Because of its role in the system, a parameter of this kind has to be represented somehow no matter what approach one takes to designing the model. On the other hand, a *hidden parameter* or an *H-parameter* is any parameter that does not satisfy any of the above conditions. It originates from the knowledge engineer's mental model of the internal structure and operation of QUIZ, and participates in reasoning, but the user (especially a novice) need not be aware of its existence. Making an explicit distinction between transparent and hidden parameters is particularly useful for selective display of information during a consultation session.

Some parameters may have multiple occurrences. A subscript must be used when referring to an individual occurrence of such a parameter.

4.1.2.2. Parameter Attributes

Each parameter of the internal model of a report has the following attributes:

a) *Name* by which it is referred to in communications with the user. It is

represented by a noun phrase (e.g., 'the number of subordinate files').

b) *Variable* which is the internal name of the parameter. It is used to represent the parameter in the rest of the knowledge base (e.g., NSF).

c) *Type constraint* on the value of the parameter (e.g., integer).

d) *Value* assigned to the parameter through a reasoning process.

e) *Default* to be used if the value of the parameter is not assigned.

f) *Kind* that indicates whether the parameter is transparent or hidden.

g) *MO-flag* that indicates whether multiple occurrences are allowed.

h) *Owner*, a parameter of which this parameter is an attribute (e.g., 'FILE' is the owner of 'ALIAS'). This is a mechanism for logically grouping the parameters.

Each parameter is, thus, represented by a tuple of its attribute values.

4.1.3. LESK Statements

In addition to the Augmented Syntax Language for describing QUIZ statements, we need a language for describing reports and for stating other facts about the QUIZ world.

The decision to transform original questions into sets of simple statements, as well as the highly technical nature of the domain, allow us to adopt a simple,

unambiguous language like LESK [Skuce 77; Skuce 83] as an appropriate representation language.

LESK (Language for Exactly Stating Knowledge) is an English-like language based on first order logic. In fact, any LESK statement in which all variables (including those that represent parameters) are replaced by the appropriate noun phrases reads as a correct English statement. LESK supports nouns (count, mass, proper), adjectives, some determiners and quantifiers, simple verb structures, and prepositional phrases. It has been developed for general purpose knowledge acquisition and used in the development of another expert system [Skuce et al. 85a; Skuce et al. 85b; Tauzovich et al. 85]. Since the natural language processing aspect of the Advisor is not the focus of this research, we will not discuss this issue further.

Descriptions of results obtained from a QUIZ program, especially descriptions of problems expressed in questions, play a very important role both in the knowledge acquisition process and in the formulation of queries. Unfortunately, these descriptions are quite vague in many original questions. The analysis of 'WHY' questions has determined that the major cause of this vagueness is the user's reference to some quantitative characteristics of the report, made in poorly chosen qualitative terms (e.g. "Why are my skips off?"). This is an important issue that has to be dealt with in the design of the Advisor, especially since our goal is to give advice at an intensional rather than extensional level. We will, therefore, limit descriptions of quantities to statements that indicate that the number of some report objects is larger/smaller than expected. For example, the original, vague

question:

"Why are my skips off?"

will have to be made more specific, e.g,

"Why did I get a large number of blank lines at ...?"

This type of restriction is very appropriate when a system is based on causal relationships, some of which cannot be fully defined. For example, causal knowledge about QUIZ includes the fact that, in general, the existence of a record selection criterion in a program decreases the number of records to be reported. To be more specific about the number of reported records, one needs to access the database and the data dictionary used in the generation of the report, or, in some cases even simulate the actual execution of the program.

In summary, reports are described using two kinds of LESK statements:

a) *Regular statements* that describe qualitative aspects of the report, and

b) *Quantity-related statements* that describe the number of some kind of report objects as (too) large or small.

In the rest of the thesis, we will refer to these as *LESK statements*.

Our formalism allows the knowledge engineer to introduce *synonyms* and *antonyms*. For example, 'alternative_name' is a synonym of 'alias' in the QUIZ terminology. This can be indicated through any LESK statement that refers to the concept of an alias for a file, e.g,

.ALIAS is the alias (= alternative_name) of .FILE

The synonym that precedes the parentheses is considered to be *canonical*, i.e., the one used throughout the knowledge encoding process. Either of them may, however, be used by the user during a query-answering session. Antonyms (and antonymous phrases) are introduced in a similar fashion (e.g., " ... required (= not optional) ...").

4.1.4. Causal Rules and Frames

The execution of a QUIZ program can be logically divided into two processes: the Parser and the Reporter. The Parser interprets QUIZ statements and builds the internal description of a report. When an occurrence of the 'GO' statement is encountered by the Parser, the Reporter is invoked to produce the report, as specified by the internal description, using information from the database. Since the Parser and the Reporter are logically and temporally two distinct processes, we can partition our rule-base into Parser rules and Reporter rules.

4.1.4.1. Parser Rules

In this section we will further develop the representation of rules that will be used to express relationships between QUIZ code and the Parser's recognizing its semantics. We will introduce a method of scoping and organizing rules, as well as for dealing with errors in the code and with defaults of statement options.

The causal relationships within the QUIZ Parser are represented by the *Parser*

rules or *P-rules*. The purpose of the Parser is to interpret QUIZ statements and, based on their arguments, add appropriate details to the internal description of the report. This suggests types of statements for antecedent and consequent parts of rules, as well as the semantic reading of rules. A general pattern of P-rules is

S ---> I

It is interpreted as follows: the existence of an occurrence of the QUIZ statement (or part thereof) 'S' in a program causes the Parser to take action 'I', which changes the internal model of the report.

The consequent part of a P-rule describes a change of the value of one or more parameters. By definition, transparent parameters are the only type of parameter that may appear in the antecedent part of a P-rule. The consequent side deals with changes to the internal model of a report, thus, it may contain hidden parameters as well. Allowing both types of parameters to appear in the same rule may create unnecessary processing overhead when the rule is used in question-answering: the system would need some mechanism to eliminate references to hidden parameters without losing any semantic content of rules. In order to avoid this, we introduce an intermediate level of knowledge representation which deals with transparent parameters only and, therefore, is more suitable for communicating the knowledge to the user. This level contains statements that reflect the declarative semantics of QUIZ statements.

Having introduced the intermediate knowledge representation, we can now split any rule that contains both kinds of parameter into two rules, first of which

does not have any hidden parameters:

PT-rule: S ---> D

PH-rule: D_2 ---> I

With 'S' and 'I' meaning the same as above, the first rule reads "S causes the Parser to recognize that D holds"[+], and the second one "the fact that the Parser has recognized that D holds causes I to happen". We will refer to the two types of Parser-rules as *PT-rules* and *PH-rules*, respectively.

LESK is used as a knowledge representation language both for expressing the declarative semantics of QUIZ statements (D), and for describing the Parser's actions that change the internal model of a report (I). The syntax of QUIZ statements is encoded in ASL.

A few QUIZ statements are very simple (e.g., 'GO'), but most statements contain components, each of which has specific effects on the report being built. To create one rule for each of the QUIZ statement types would generally result in a small number of very large and impractical rules. Therefore, the scope of rules is narrowed down from the statement level to that of its parts.

In the rest of this section we will introduce three types of PT-rules that are appropriate for modeling effects of components of QUIZ statements, and then propose a method of grouping the rules.

[+] We have chosen to use as the consequent the act of the Parser recognizing that D holds. It is equivalent to saying that S causes predicate D to become true.

4.1.4.1.1. Types of PT-Rules

There are three types of PT-rules that deal with (present) components of statements, absent components (default), and error conditions.

Regular PT-Rules. The presence of a component of a statement has a semantics which can be described by a LESK statement which uses transparent parameters only. We represent the relationships between the presence of a component and the Parser's recognizing its semantics by the following type of rule:

S ---> D

where 'S' is a fragment of the statement's augmented syntax, and 'D' is its declarative semantics. For example, the presence of a valid file declaration after 'LINK TO' in an 'ACCESS' statement causes the Parser to recognize the file as a subordinate one. This is captured by the following regular PT-rule:

```
LINK TO {file_declaration:?FILE}
        ---> .FILE is a subordinate file
```

Default PT-Rules. In many programming languages, if the value of a program parameter is not specified in the code, then a predefined (default) value is assigned. In the general case, when a parameter value may be determined by more than one statement type or when multiple instances of a statement type are allowed, it is sometimes only towards the end of parsing that the

parser ean determine whether to use a default value for a parameter.

It is a characteristic of QUIZ that the value of each parameter appearing in the augmented syntax of a statement may be determined through only qne type of statement. Moreover, most statements cancel the previous occurrence of statement of the same type, i.e., the value of a parameter is based on a single statement of the program. We take advantage of this when representing the effect which the absence of an optional component of a QUIZ statement has on the internal model. This type of relationship is represented as 'absent-->'. When writing rules for an optional entry, we can include two causal links: one corresponding to the case when the entry is present (regular PT-rule), and another for the case when it is absent from a statement (default PT-rule). Due to the clear difference in the semantics of the two causal links, the regular and default PT-rules can be be coupled into a compact unit:

```
S          --> I1
    absent---> I2
```

For example, the QUIZ rule stating that "if an 'ALIAS' option for a file is used in an 'ACCESS' statement, then the file has the specified alias, otherwise it has no alias" can be represented as follows: (Assume 'DFILE' is a T-parameter denoting the dictionary name of a declared file):

```
[ALIAS_part]
                    ---> .ALIAS is specified
               absent---> .DFILE has no alias
```

Error PT-Rules. The QUIZ Parser issues an error message whenever a syntactic or semantic rule of the language is violated. As in many other systems,

messages are not always clear and users need explanations. In order to be able to explain errors, we have introduced a special kind of rule of the following form:

C fail---> E

where 'C' is an ASL condition and 'E' is a LESK statement that refers to the actual message issued when the condition is violated. For example, each file declared in the 'ACCESS' statement must be unique, or else an error message is issued. This is represented as:

{Cond: .FILE is a unique file}
fail---> error message ERRA2 is issued.

ERRA2 is the code assigned to the message. Error messages are represented separately as code-text-explanation triples. The triple for the above message looks like this:

ERRA2 "Invalid file name"
(file names are not unique - need aliasing)

Messages that contain extensional information are represented as patterns, i.e., LESK statements that contain variables. The explanations are predefined ("canned") and represented as natural language texts.

4.1.4.1.2. PT-Sequences

For each nonterminal[+] in the syntactic hierarchy of the statement we can create a sequence of rules called a *PT-sequence:*

S1 --> D1
S2 --> D2
:
Sn --> Dn

If the ASL expressions that appear on the antecedent side of rules in a PT-sequence (S1,S2,...,Sn) are concatenated, they form the augmented syntax of the nonterminal. The decision on how to partition the augmented syntax is a matter of the knowledge engineer's judgement, yet, some heuristic rules are obvious (e.g. each optional entry must have at least one rule of its own).

By representing PT-sequences as extensions of the augmented syntax of QUIZ statements we can manage to engineer rules of reasonable size while keeping together the knowledge about the entire syntactic unit. Since the parts of most QUIZ statements appear in a fixed order within the statement, sequences rather than sets of rules are associated with nonterminals. Rules are ordered using the appropriate labeling system (the label of a rule is enclosed in angle brackets).

As an example, the following is the sequence of rules for the file declaration component of the ACCESS statement:

[+] In all references to a nonterminal, we mean a nonterminal other than a statement.

```
file_declaration =
      {file:?DFILE/file_name}
      {Cond: .DFILE is described in the dictionary as a file,}
            <t3>      fail---> error message ERRA1 is issued,
      [ALIAS_part]
            <t4>          ---> .ALIAS is specified,
            <t5>  absent---> .DFILE has no alias,
      {Cond: .FILE is a unique file,}
            <t6>      fail---> error message ERRA2 is issued,;
```

Text and explanations for error messages 'ERRA1' and 'ERRA2' can be found in Appendix A.

4.1.4.1.3. Hierarchy of Statements

The augmented syntax of a statement is partitioned over a number of sequences, one for each of its nonterminals, in order to explicitly show causal relationships that involve parts of the statement. The set of sequences corresponds to the statement syntax in its full form. By analyzing complex QUIZ statements we have realized that different patterns of statements within the same statement type often have significantly different semantics. For instance, an ACCESS statement with one or more occurrences of a 'LINK TO' option (commonly referred to as the 'ACCESS with a LINK TO option' statement) declares multiple files, the first of which is called the 'primary' file and the others 'subordinate' files. If no 'LINK TO' option is used ('ACCESS without a LINK TO option' statement), a single file is declared. A single file should not be called the primary file, so any reference (by role) to the file declared after the word 'ACCESS' would have to depend on the type of the 'ACCESS' statement and would require a conditional construct on the consequent side of a rule.

To avoid these difficulties, we introduce a hierarchy of QUIZ statements, in which a parent statement has a more general syntax than its children. The presence/absence of a top-level component (optional or repeated) normally determines the subtype of the statement. When more than one such component occurs in the statement, there are multiple ways in which the subtypes can be defined. We analyze statements left to right, so, for example, a 'SELECT' statement can be further classified as a 'SELECT with file' or a 'SELECT without file' statement. It would be equally valid to partition 'SELECT' statements into 'SELECT with IF' and 'SELECT without IF' statements. The formalism can be extended to provide both. As an example, the complete hierarchy of 'SELECT' statements is presented on the next page (Figure 4.2).

4.1.4.1.4. PH-Rules

While PT-rules represent relationships between QUIZ statements and the Parser's recognizing their semantics in terms of transparent parameters only, PH-rules express relationships between the latter and additional changes the Parser makes to the internal model of a report (initial changes are simple assignments of values to parameters as prescribed by the augmented syntax fragments within PT-rules). Unlike PT-rules, PH-rules may contain hidden parameters. The syntax of a PH-rule is

D ---> I

where 'D' is a conjunction of LESK statements and 'I' is a single LESK statement. Example:

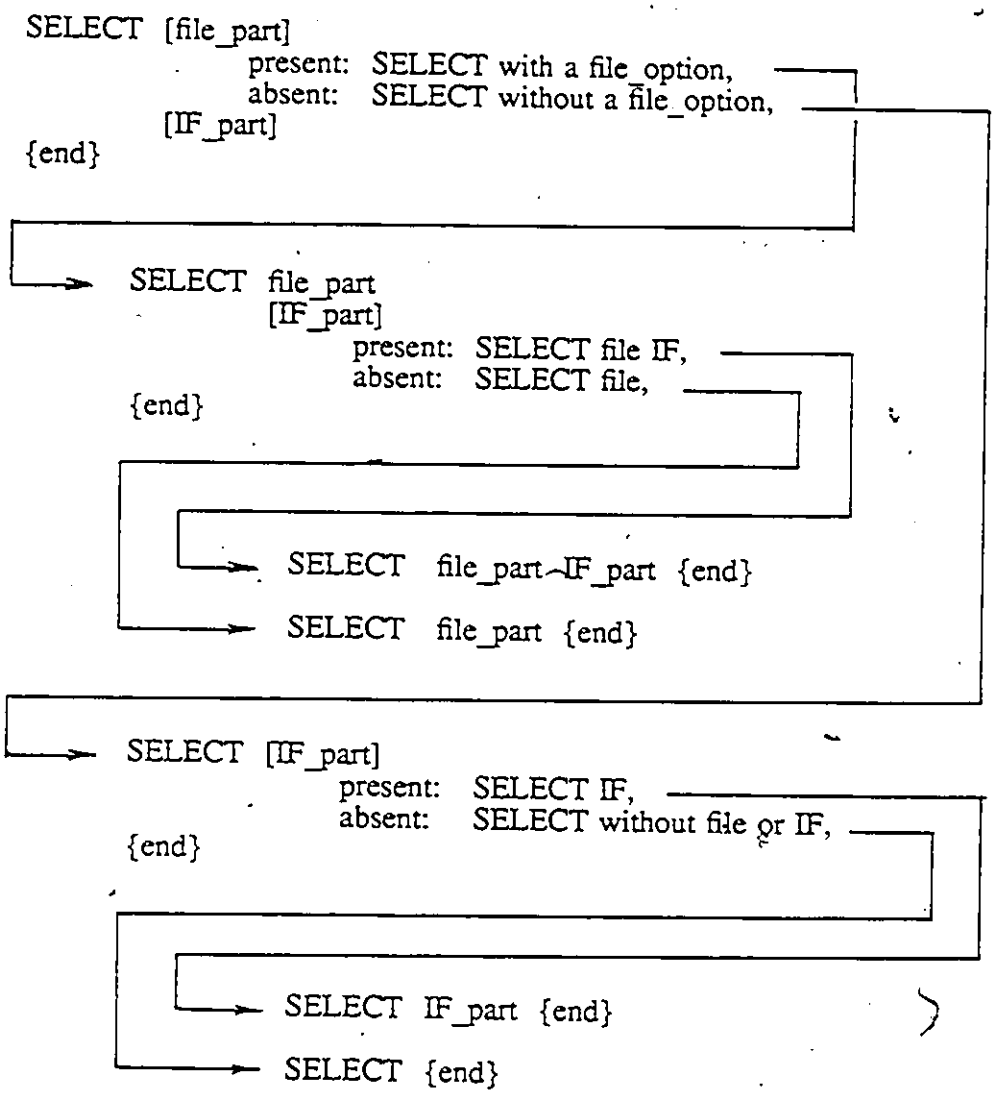


Figure 4.2. Statement Hierarchy: An Example

records of .FILE are optional
---> ?OPT is set to 'YES'

According to this rule, whenever the Parser recognizes that the records of a file are specified as optional, the value of the H-parameter 'OPT' is set to 'YES'.

PH-rules are also used to represent elementary procedural knowledge that involves any kind of parameters (transparent or hidden). For example, if a file has an alias, the file is referred to (in the rest of the program) by its alias name rather than its dictionary name. PH-rules that reflect this are the following:

.ALIAS is specified
---> ?FILE is set to .ALIAS,

.DFILE has no alias
---> ?FILE is set to .DFILE

Most actions described on the consequent side of a PH-rule are simple assignments of a value to a parameter. Some actions are more complex though, and have associated complex procedures[+] (e.g., the procedure by which the files declared in the ACCESS statement are linked). Representation and reasoning with procedural knowledge is outside the scope of this research. Therefore, the existence of an associated procedure is simply acknowledged as in the following rule:

.FILE is a subordinate file
---> a link for .FILE is established
through an internal procedure

If a rule that refers to a procedure is fired, and the user is interested in individual

[+] By a procedure we mean a sequence of actions for achieving a particular goal.

steps of the procedure, the Advisor should invoke its procedural knowledge component to provide the details.

4.1.4.2. Parser or Statement Frames

Knowledge about types of QUIZ statements (e.g., 'ACCESS' statement) is organized in *Parser frames* or *statement frames*. The hierarchy of Parser frames corresponds to the hierarchy of statements: each lower-level frame corresponds to a statement with a more restricted syntax than its parent. Sequences of rules are associated with the most general type of statement to which they apply and inherited by more restricted forms of the statement. Parser frames have the following slots (first three of them are mandatory):

Statement — The statement name.

Purpose — A LESK phrase that describes the purpose of the statement, i.e., the topic of the frame.

Syntax/Kinds — This describes the complete top-level syntax of the statement in ASL. It also names the more restrictive forms of the statement that are obtained when a certain syntactic component is present/absent.

PT-Sequences — A PT-sequence for each nonterminal of the statement. Each rule in the sequence represents a relationship between a syntactic feature of the statement and the Parser's recognizing its semantics.

PH-Rules A set of PH-rules that represent causal relationships between the Parser's recognizing the semantics of QUIZ code and changes made to the internal model of the report.

Intro-Params Parameters that are introduced within the frame and assigned a value through the rules of the frame. For each such parameter its descriptive name is given in a form of a LESK noun phrase.

Error Messages A code-message-explanation triple is given for each error message referenced in some PT-rule.

The high-level BNF for Parser frames is given in Figure 4.3.

```
parser_frame    =    'STATEMENT'        statement_name
                     'PURPOSE'          LESK_phrase
                     'SYNTAX/KINDS'     syntax_kinds
                   [ 'PT-SEQUENCES'     sequences ]
                   [ 'PH-RULES'         ph_rules ]
                   [ 'INTRO-PARAMS'     params ]
                   [ 'ERROR-MESSAGES'   errors ]
                     'END'

syntax_kinds    =  ASL_statement
                =  ASL_expression
                        present: statement_name
                        absent:  statement_name
                   ASL_expression

sequences       =  sequence  [sequences]

sequence        =  nonterminal '=' pt_rules

pt_rules        =  pt_rule  [pt_rules]

pt_rule         =  ASL_expression
                          label          '--->'  LESK_stmt
                =  ASL_expression
                          label          '--->'  LESK_stmt
                          label 'absent--->'  LESK_stmt
                =  ASL_expression  condition
                          label   'fail--->'  LESK_stmt
                =  ASL_expression

ph_rules        =  ph_rule  [ph_rules]

ph_rule         =  LESK_expr
                          label          '--->'  LESK_expr

errors          =  error  [errors]

error           =  error_code  message  explanation
```

Figure 4.3. The Syntax of Parser Frames

In addition to the already mentioned benefits from having an explicit hierarchy of statements, this structure increases the capability for more selective retrieval of relevant knowledge from the knowledge base.

We will end this section by presenting, as an example, the frames for the 'ACCESS' statement (Figure 4.4). More examples can be found in Appendix A.

```
STATEMENT      ACCESS

PURPOSE        to access (= declare) a single file
               xor
               to access multiple files

SYNTAX/KINDS

     ACCESS {first_file_declaration:?FILE/file_name}
            *[{Sub: n from 2} {LINK_TO_part:?FILE}] {Count:?NSF}
               present: ACCESS with a LINK_TO_option,
               absent:  ACCESS without a LINK_TO_option,
     {end}

PT-SEQUENCES

     first_file_declaration =
         file_declaration
             <t1>        --->  .FILE is a declared file,
             <t2>        --->  records of .FILE are required
                                          (= not optional),;

     file_declaration =
         {file:?DFILE/file_name}
         {Cond: .DFILE is described in the dictionary as a file,}
             <t3>    fail--->  error message ERRA1 is issued,
         [ALIAS_part]
             <t4>        --->  .ALIAS is specified,
             <t5>  absent--->  .DFILE has no alias,
         {Cond: .FILE is a unique file,}
             <t6>    fail--->  error message ERRA2 is issued,;

     ALIAS_part = ALIAS {name:?ALIAS/file_name};

     LINK_TO_part = LINK TO subsequent_file_declaration;

     subsequent_file_declaration =
         file_declaration
             <t7>        --->  .FILE is a declared file,
             <t8>        --->  .FILE is a subordinate file,
         [OPTIONAL_part]
             <t9>        --->  records of .FILE are optional,
             <t10> absent--->  records of .FILE are required,;

     OPTIONAL_part = OPTIONAL;
```

Figure 4.4. 'ACCESS' Frames (Page 1)

```
PH-RULES

     .FILE is the single file
          <h1>   --->  ?DDREC is set to
                              the dictionary_record of .DFILE,;

     .FILE is the primary file
          <h2>   --->  ?DDREC is set to
                              the dictionary_record of .DFILE,;

     .FILE is a subordinate file
          <h3>   --->  a link for .FILE is established through
                                          an internal procedure,;
          <h4>   --->  ?DDREC is augmented with
                              the dictionary_record of .DFILE,;

     .ALIAS is specified
          <h5>   --->  ?FILE is set to .ALIAS,;

     .DFILE has no alias
          <h6>   --->  ?FILE is set to .DFILE,;

     records of .FILE are optional
          <h7>   --->  ?OPT is set to 'YES',;

     records of .FILE are required
          <h8>   --->  ?OPT is set to 'NO',;


INTRO-PARAMS

     FILE[n]         is a file,
     DFILE[n]        is the dictionary_file of .FILE[n],
     ALIAS[n]        is the alias (= alternative_name) of .DFILE[n],
     OPT[n]          is the optional_records_flag of .FILE[n],
     DDREC           is the description of record_complexes,
     NSF             is the number of subordinate files,


ERROR-MESSAGES

     ERRA1           Expected: RECORD key file,
                     (the file was not found in the dictionary,)

     ERRA2           Invalid file name,
                     (file names are not unique - need aliasing,)
END
```

Figure 4.4: 'ACCESS' Frames (Page 2)

```
STATEMENT      ACCESS with a LINK_TO_option

PURPOSE        to access multiple files

SYNTAX/KINDS

     ACCESS   {first_file_declaration:?FILE}
              **[{Sub: n from 2} {LINK_TO_part}] {Count:?NSF}
     {end}

PT-SEQUENCES

              <t1>       ---> '.FILE is the primary file,;

END



STATEMENT      .ACCESS without a LINK_TO_option

PURPOSE        to access a single file

SYNTAX/KINDS

     ACCESS   {first_file_declaration:?FILE} {end}

PT-SEQUENCES

              <t1>       --->  .FILE is the single file,;

END
```

Figure 4.4. 'ACCESS' Frames (Page 3)

4.1.4.3. Reporter Rules and Frames

In this section we will examine types of causal relationships involved in the reporting component of QUIZ, and propose *Reporter rules* or *R-rules* for capturing this type of knowledge.

The Reporter generates a report based on two sources of information: the internal model of the report created by the Parser, and data retrieved from the database. Combinations of characteristics of the internal model and/or the database have specific effects on the resulting report. As discussed earlier in this chapter, reports are described using two kinds of LESK statements: those that describe qualitative aspects of the report (regular statements), and those that describe the number of some kind of report objects as (too) large or small (quantity-related statements).

4.1.4.3.1. Regular R-rules

The relationship between a characteristic of the report and its cause, if both are described using regular statements only, can be represented by rules of the following format:

I ---> R

where 'I' is a conjunction of statements about the internal model of a report and 'R' is a description of the resulting feature of the report. We term this type of a rule the *regular R-rule*. An example of such a rule is:

the status of .RITEM is 'OVERFLOW',
the type of .ITEM/.RITEM is 'NUMERIC'
---> ?FIELD is filled with the crosshatch sign

(where RITEM is a report item,
ITEM/RITEM is the item of the report item,
FIELD is the field of the report item)

4.1.4.3.2. Quantity-Related R-rules

Quantity-related statements are used in user's questions to indicate that the number of given report objects is larger (smaller) than expected. In order to answer this type of questions, the system must be able to identify those characteristics of the internal model of the report and/or the database that increase (decrease) the number of the objects. This type of causal relationship is represented by rules of the following format:

I incr---> N

or

I decr---> N

where 'I' is a conjunction of statements about the internal model of a report and database parameters, and 'N' is a LESK noun phrase denoting the numeric attribute of the report that 'I' increases or decreases, respectively. These rules are termed *quantity-related R-rules*. As an example, the following rule states that the existence of a selection condition on a file whose records are required decreases the number of report complexes (a report complex is a tuple of data items which is printed in the report):

there is a selection condition on records of .FILE
.OPT is set to 'NO'
decr---> ?NRC

where OPT is the parameter set to NO if records of the file are required, and NRC is the number of report complexes.

Increasing (decreasing) the value of some numeric attributes of the record specification or the environment (data dictionary or database) causes increases (decreases) in values of some numeric attributes of the report. These relationships are captured using quantity-related rules with 'prop--->' representing the causal link:

N1 prop---> N2

meaning that 'N2' changes proportionally with 'N1', where 'N1' and 'N2' are LESK noun phrases, usually parameters, representing numerical attributes mentioned above. For example, the larger the trailing skip specified (in the 'REPORT' statement) the larger the down-shift at each report complex (i.e., the number of blank lines skipped before next record complex is reported). This is represented by the following rule:

.TRS prop---> the downshift at .REPC

('TRS' and 'REPC' are parameters corresponding to the trailing skip and a report complex, respectively).

4.1.4.3.3. Reporter Frames

Reporter rules are organized into simple frames that have the following three slots:

Topic A LESK noun phrase that describes the topic of the frame.

Rep-Rules A set of Reporter rules, and

Intro-Params Parameters that are introduced within the frame and assigned a value through the rules of the frame. These parameters correspond to objects of a QUIZ report. As in Parser.frames, for each such parameter its descriptive name is given in a form of a LESK noun phrase.

Reporter frames inherit their hierarchy from the hierarchy of activities being developed for the QUIZ Advisor. In the small knowledge base for the prototype of the system, for simplicity, we have grouped all rules for the reporting phase into a single frame shown in Appendix A.

4.1.5. Causal Network and Reasoning

Having introduced types of rules that can be used to capture all types of causal relationships we have found in QUIZ programs, we can build a *causal network of a program*, as shown in Figure 4.5.

Reasoning within the network may be done by forward and/or backward chaining of causal rules as the primary deduction mechanisms. *Forward chaining* starts with a description of QUIZ code and works toward establishing some causal links between the given code and features of the report. If the actual code is given (e.g., 'ACCESS X'), then its syntactic features determine the applicable PT-rules. By chaining PT-rules, PH-rules, and R-rules, links that are labeled 'SYN-ILs'

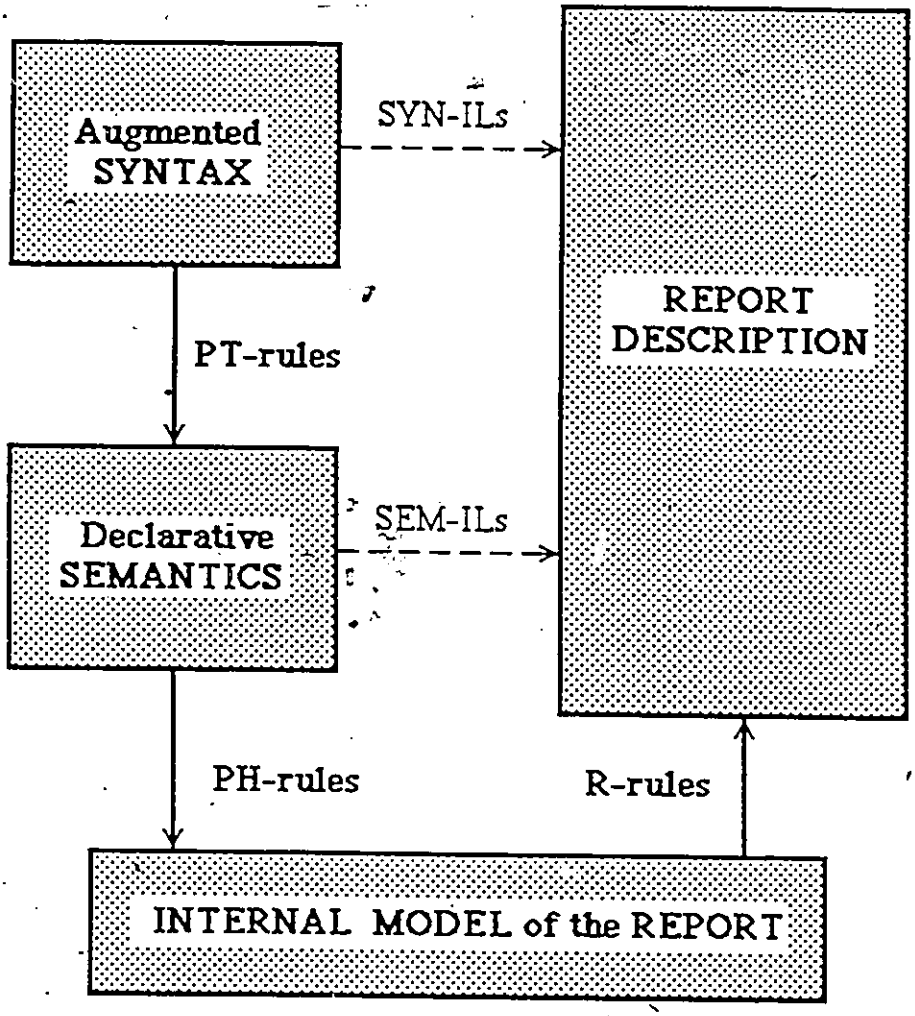


Figure 4.5. Causal Network of a QUIZ Program

(SYNtax-oriented Inferred Links) can be established. However, if the code is described in semantic terms (e.g., 'X is the single file'), the obtained links are the ones labeled 'SEM-ILs' (SEMantics-oriented Inferred Links). In *backward chaining* of rules, the direction of reasoning is from the description of the appearance of a report towards semantic and syntactic features of the code that produced the report (i.e., opposite from the direction shown in Figure 4.5).

The causal network and the two reasoning mechanisms are the basis for answering queries to the QAUZ subsystem of the Advisor (described in the next chapter).

4.2. Model of a QUIZ Session

QUIZ can be run in a batch processing mode, but it is far more frequently used as an interactive system. The causal relationships discussed so far concern any QUIZ program regardless of the mode of running it. In this section we look at the causality involved in a QUIZ interactive session.

In the model of a program, the agent of all actions is the system itself and actions are performed in a predefined temporal order. An action of the user issuing a statement is reflected in the fact that the statement is present in the program. In order to represent the interactive nature of a QUIZ session we start from the following observations about its states and actions:

a) There are two possible agents of actions: the User and QUIZ; (the initiation and termination of a QUIZ session also involve the operating system, but we are focusing here on the user-QUIZ dialogue);

b) User's actions and QUIZ's actions belong to two disjoint sets of action types;

c) A QUIZ session may be partitioned into time intervals which, in general, have three subintervals characterized by the following situations:

I1 User action (UA)

I2 one or more QUIZ actions (QA) caused by UA

I3 post-QA state

This pattern suggests the use of rules as an appropriate formalism for codifying causal knowledge in this domain.

4.2.1. Causal Rules

Causal relationships in the QUIZ session are captured in *Session rules* or *S-rules* of the following form (the arrow represents a causal link):

{ <Description of state(s) enabling UA> }

<Descr. of UA> -----> <Descr. of QA(s)>

{ <Description of the post-QA state> }.

The antecedent side of a causal rule describes both the state in which a user action can be performed, and the action itself. Whenever the user action is performed, it causes one or more QUIZ actions to be performed as a consequence. These are described on the consequent side of a rule, followed by the description of a resulting state. Each of the above descriptions is represented by a single LESK statement or a conjunction of LESK statements.

The temporal sequencing of the user action and QUIZ reaction is clear: a cause must occur before its consequences. The sequencing of QUIZ actions, in general, could be more complex and might require the use of some temporal relations, similar to the ones defined in [Allen 83, 84]. The examples we chose to experiment with (the material covered in Chapter 1 of the QUIZ manual) required only a relation for sequencing temporally disjoint actions: 'A, then B' means that action A terminates before action B can start.

4.2.2. Examples of Rules

The proposed formalism for codifying knowledge will be illustrated with several rules ('OS' denotes the operating system):

Example 1:

{OS is awaiting a top_level_command}

User enters 'QUIZ' ----->
OS invokes QUIZ, then
QUIZ displays the introductory_message, then
QUIZ displays the prompt_character

{QUIZ is awaiting a statement,
QUIZ is expecting an ACCESS_statement}.

Example 2:

{QUIZ is awaiting a statement}

User issues an ACCESS_statement ----->
QUIZ initializes the report_parameters, then
QUIZ processes the ACCESS_statement

{QUIZ is awaiting a statement}.

Example 3:

{QUIZ is executing}

User types <control,'Y'> ----->
QUIZ stops executing, then
QUIZ displays 'Do you wish to continue?'

{QUIZ execution is suspended}.

Example 4:

{QUIZ is awaiting a statement}

User enters 'HELP' ----->
QUIZ displays an explanation of the HELP_system, then
QUIZ displays a list of statement_keywords, then
QUIZ displays the HELP_prompt_character

{QUIZ is in the HELP_mode,
QUIZ is awaiting a statement_keyword}.

4.2.3. Extending the Causal Network

Deduction mechanisms for reasoning with the above described kind of causal rules include appropriate variants of forward and backward chaining of rules as a basis for answering questions about the process of interacting with QUIZ.

This model can be integrated with the causal model of a program, as shown in Figure 4.6. For each LESK statement of the following form:

QUIZ processes the $_statement

where '$' stands for a statement name (e.g., 'ACCESS', like in Example 2), there is an applicable set of PT-sequences originating from one or more Parser frames. The set of PT-sequences may be interpreted as a description of a procedure for executing an action of the above form. Of course, in order to execute the procedure, the actual statement must be made available to the reasoning mechanism.

For a simple example, let us assume that the knowledge base contains the fol-

lowing rule:

{QUIZ is awaiting a statement}

User issues a REPORT_statement ----->
QUIZ processes the REPORT_statement

{QUIZ is awaiting a statement}.

Suppose that, during a QUIZ session, the user enters the following statement (while the system is awaiting a statement):

REPORT

The rule only tells us that the system will process the statement, then wait for the next one. The implications of the issued statement on the results can be analyzed using the causal network of a QUIZ program (Figure 4.5). Forward reasoning with the knowledge presented in Appendix A leads to the conclusion that, as a result of the above statement, nothing will be reported (this is discussed in detail in the next chapter).

Backward reasoning from results, on the other hand, points to the causes of the result, i.e., user's actions of issuing certain statements, and the context in which they are entered during a QUIZ session.

In Figure 4.6., causal links between a user's action and its effects on the resulting report are labeled 'SR-ILs' (Session-Report Inferred Links).

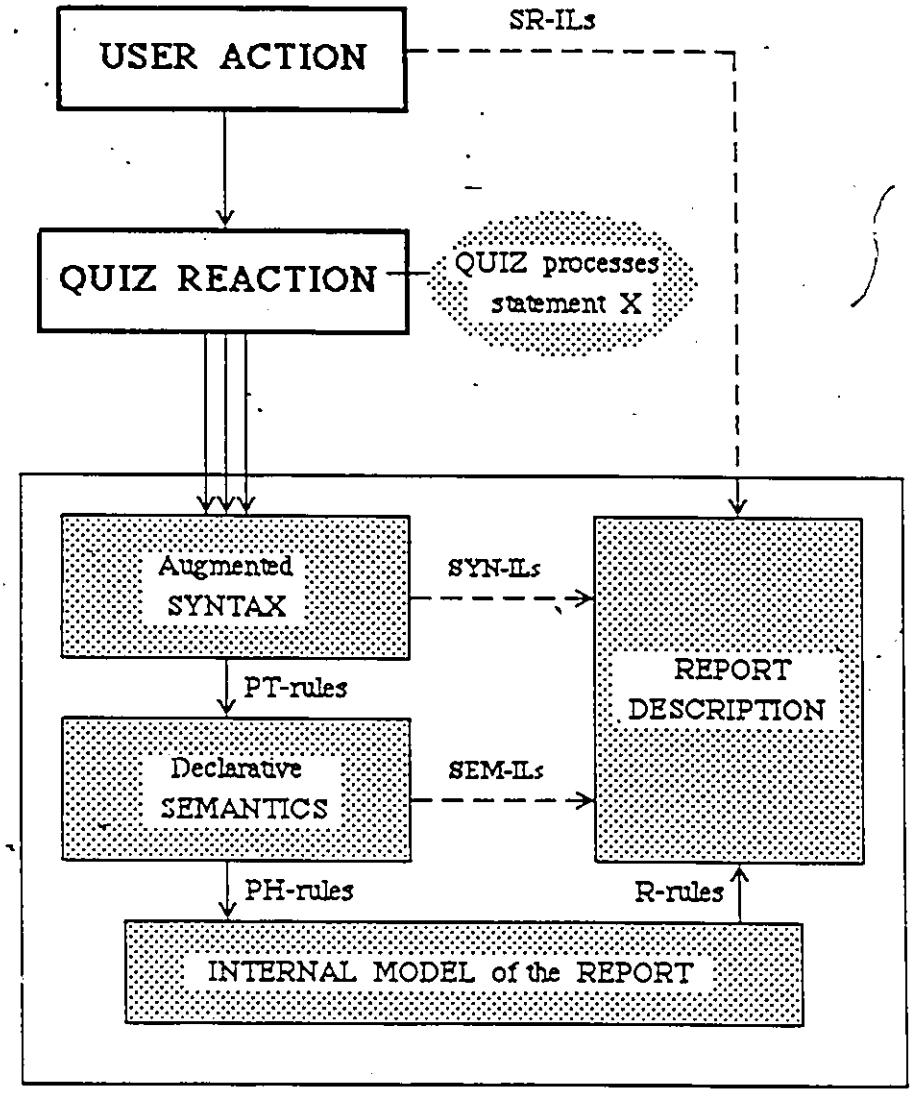


Figure 4.6. Causal Network of QUIZ

5. QAUZ PROTOTYPE

This chapter describes various aspects of the design and implementation of QAUZ, the causal component of the QUIZ Advisor prototype. Details of major query-answering strategies applied in QAUZ are the topic of Chapter 6.

5.1. Overview of the QUIZ Advisor

The prototype of the QUIZ Advisor consists of three major modules: HDI, QAUZ, and SEQAP.

The **HDI** subsystem accepts queries consisting of an optional set of assumptions and a 'method' question (e.g., "How do I ...?"). A query is submitted through an interface written in Prolog on a SUN-3 workstation. It is then parsed into an s-expression and passed to the answerer on a Xerox-1186 Lisp machine via Ethernet. The HDI query-answerer is implemented using IntelliCorp's KEE, a commercial knowledge engineering tool based on a combination of frame and rule paradigms. The answer to a query is produced in the form of a partial QUIZ program (usually one or two statements). Details about this subsystem can be found in [Skuce 86] and [Stanley 87].

The primary purpose of the **QAUZ** subsystem, the causal query component of the Advisor, is to give explanations in response to 'WHY' queries. QAUZ also handles most of the remaining types of queries (see the taxonomy in Section 3.1.1.). A justification for the decision to develop two independent query-answering subsystems (HDI and QAUZ) was given in Chapter 3. This chapter

describes the architecture of QAUZ, the creation and structure of its knowledge base, and the development of its basic reasoning mechanisms. The major query-answering strategies applied in QAUZ are the topic of the next chapter.

SEQAP (Subset of English for the QUIZ Advisor Project) is the natural language module of the Advisor. It is implemented in Prolog and organized into three components: lexical (based on a lexicon with approximately 1700 entries), syntactic, and post-processing (simple semantic verification and format translation). SEQAP is accessed by both HDI and QAUZ subsystems to translate portions of user input into internal forms suitable for further processing. It is also invoked by QAUZ during the compilation of frames to provide translations of LESK statements embedded in various slots. SEQAP is fully described in [Delisle 87].

5.2. Overview of QAUZ

The prototype of QAUZ is based on the knowledge representation formalism and the approach to query-answering introduced in the previous two chapters. It is implemented in Quintus Prolog [Quintus 86] on a SUN-3 workstation. The basic architecture of the QAUZ prototype is schematically presented in Figure 5.1 (arrows represent the flow of data).

The development of the prototype proceeded in two stages: the creation of the knowledge base, and the implementation of the question answerer.

a) **Knowledge Compiler.** The domain of the QAUZ knowledge base was defined (a subset of QUIZ), and the relevant knowledge was acquired and

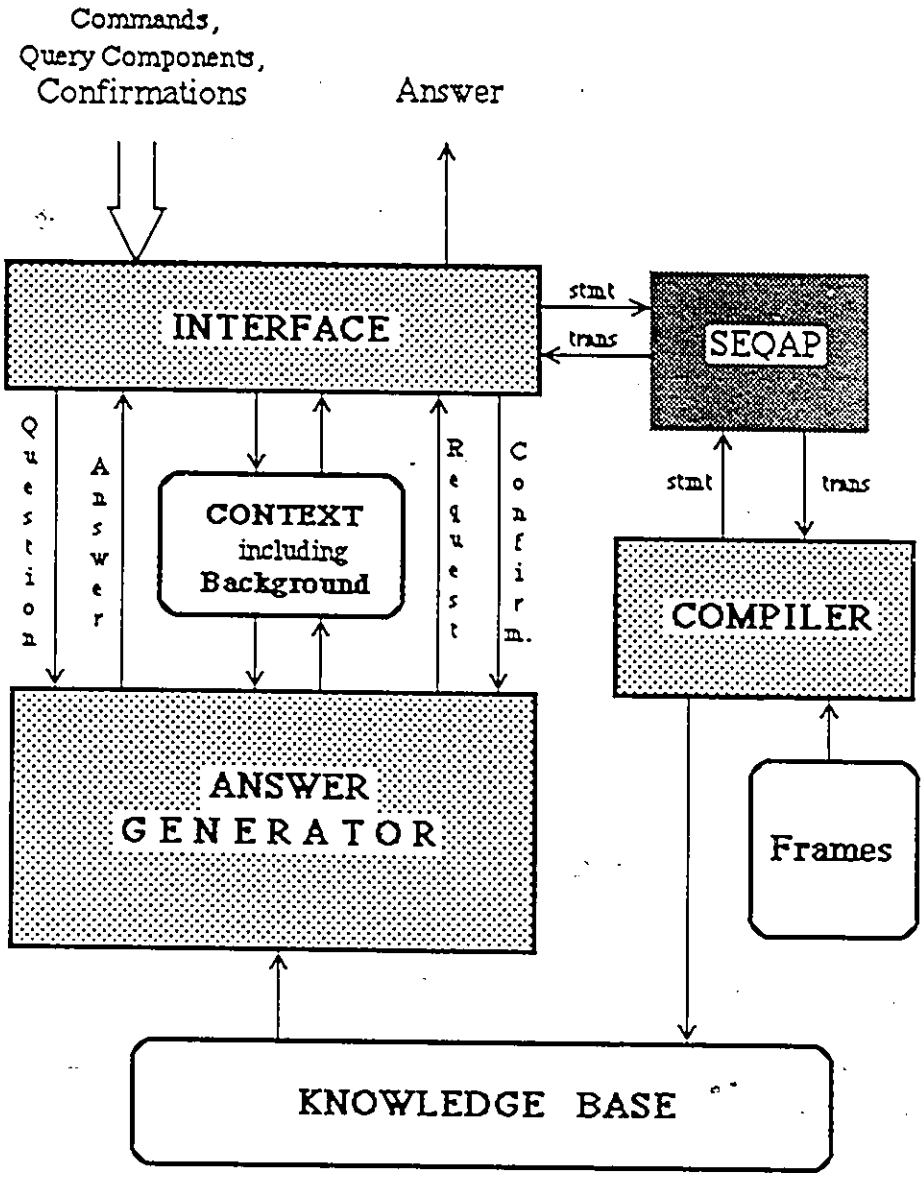


Figure 5.1. The Architecture of QAUZ

encoded in a number of Parser frames and a single Reporter frame. In parallel with this process, a knowledge compiler was implemented. It takes any number of Parser and/or Reporter frames as input and builds the corresponding portions of the knowledge base, as described in the next section. The size of the compiler is approximately 40K of Prolog code. The size of the knowledge base produced for the subset of QUIZ we chose as the domain of the prototype is close to 60K.

b) **Query-Answerer.** The query-answerer consists of the *interface* and the *answer generator* modules. The interface has a number of commands that enable the user to build and submit queries in an efficient manner (e.g., background information is a component of the global context and can be shared by consecutive queries). After a LESK statement (e.g., an assumption or a question) is entered through the interface, SEQAP is invoked to translate it into the internal form. Whenever a question is entered and successfully processed by SEQAP, the query is submitted to the answer generator which applies an appropriate answering strategy. In some cases the answerer has to request additional information through the interface, but the general approach is to produce an answer as focused and complete as the available information allows it to be. If the answerer succeeds in finding relevant knowledge, it generates an answer, otherwise the interface informs the user that the query cannot be answered. The query-answerer was implemented in 110K of Prolog code.

The rest of this chapter describes the prototype in more detail.

5.3. Knowledge Base

This section defines the subset of QUIZ chosen to be represented in the knowledge base of the QAUZ prototype, discusses the approach taken in knowledge acquisition, and describes the internal structure of the knowledge base.

5.3.1. Domain of the Prototype

A fully developed QUIZ Advisor would contain as much knowledge about QUIZ as possible. For prototyping purposes, however, we restricted the domain to a subset of QUIZ that was reasonably small, yet allowed us to apply and validate all the important aspects of the developed methodology while retaining the main functional features of the language.

The QAUZ prototype was built for a subset of QUIZ that we named Mini-QUIZ (wherever we use the word QUIZ in the rest of the thesis, the reference is actually made to MiniQUIZ). It consists of the following statements: 'ACCESS', 'SELECT', 'REPORT', and 'GO'. The last statement ('GO') is assumed to be present at the end of each sequence of statement, but is not explicitly dealt with in order to simplify the dialogue between the user and the system. The adopted restrictions on the syntax of the first three statements are shown in Figure 5.2 (the notation is close to that used in the QUIZ manual).

QAUZ has been limited to knowledge about QUIZ programs in order to make it consistent with the rest of the QUIZ Advisor prototype. Thus its scope does not cover the causal model of QUIZ interactive sessions (described in Section 4.2.).

```
ACCESS file1 [ALIAS name1]
       [LINK TO file2 [ALIAS name2] [OPTIONAL]]....

SELECT [file] [IF condition]

     condition:        content operator expression
     content:          record-item
     expression:       record-item/string/number
     record-item:      item-name [OF file]

REPORT [report-group / ALL [SKIP n]]

     report-group:     [report-item]... [SKIP [n/PAGE]]
                       [RESERVE n [LINES]]
     report-item:      [SKIP [n/PAGE]] [TAB n]
                       record-item [format]
     format:           [HEADING string] [PICTURE string]
```

Figure 5.2. MiniQUIZ

5.3.2. Knowledge Acquisition Process

The traditional approach to knowledge acquisition in expert system development relies mainly on interviews with domain experts. Eliciting knowledge from a human expert is very useful in those areas in which judgemental and heuristic knowledge is specially important, or in which the factual knowledge is not well documented. However, a problem often experienced in using this knowledge acquisition method is the lack of experts who can invest their time in this kind of

activity [Waterman 86, Ch.18]. In domains for which comprehensive documentation exists (e.g., regulations, manuals), an alternative approach is to use written documents as the primary source of knowledge [Skuce et al. 85; Tauzovich et al. 85].

In our domain of research the two approaches were combined throughout the knowledge acquisition phase. A significant amount of knowledge about QUIZ has been incorporated in the manual. We have been also fortunate to have available for consultation, not only QUIZ experts but the designers of QUIZ as well. One more available source of knowledge about QUIZ was the collected and analyzed set of questions and answers. In addition to the domain knowledge, this material also provided valuable statistics on topics of questions. This can be used to guide the knowledge acquisition process to ensure that knowledge required to answer questions most likely to be asked will be represented in the knowledge base. This approach was adopted by the research team that developed the HDI subsystem. Since questions usually refer to advanced topics, a large body of additional (mainly basic) QUIZ knowledge had to be acquired later to fill in the gaps.

For encoding causal knowledge, we adopted a knowledge acquisition process that is syntax-driven (i.e., focusing on one by one statement type) rather than topic-driven (i.e., focusing on one by one topic). The following points justify this decision:

a) In general, a mental model of a cause-effect relationship implies reasoning from a cause towards its effect, perhaps due to their temporal sequencing. This makes the direction from QUIZ statements (causes)

towards features of a report (effects) quite natural for knowledge acquisition.

b) Each statement or option in a QUIZ program has a unique effect on the resulting report. Some features of a report, however, may have different causes (e.g., the space between two columns may be the result of TAB, PICTURE, or a combination of these two options). Therefore, syntax-driven knowledge acquisition ensures the completeness of a knowledge base with respect to the defined domain throughout the process.

c) The QUIZ manual, an important knowledge source, is mainly organized around statements. This made locating parts of the text relevant to individual statements much easier than locating those relevant to other more general topics.

5.3.3. Internal Representation of Knowledge

Knowledge about QUIZ encoded in Parser and Reporter frames is compiled into several types of Prolog unit clauses.[+] These contain, in addition to domain knowledge, some useful meta information (i.e., provisions for indexing).

The compiler takes any number of frames as input and produces a file of corresponding knowledge base entries. A trace of the process is displayed on the screen and/or saved to allow the knowledge engineer to analyze the course of the

[+] The reader is advised to refer to Section 4.1.4. for the description of the source form of the knowledge base.

compilation and easily locate and correct possible errors in frames. After all frames have been successfully compiled, the resulting files are merged into the knowledge base.

Each ASL expression, i.e., a fragment of QUIZ syntax expressed in the Augmented Syntax Language (Section 4.1.1.) is represented by a list of its tokens. The compiled version of a token consists of its *type* (e.g., 'nt' for a nonterminal, 'op' for an optional entry, 'kw' for a keyword, etc.), and the *content*. For example, the compiled version of:

SELECT [file_part] [IF_part]

would be:

[[kw,'SELECT'],[op,[file_part,[]],[op,['IF_part',[]]]

The structure of the content depends on the type. As the above example shows, for the keyword token it is the keyword itself, and for the optional entries it consists of the name of the nonterminal and the associated parameter (if any).

LESK statements and phrases are translated into Prolog expressions by the SEQAP module. These expressions represent various linguistic components recognized in the input. A full description of the output from SEQAP is given in [Delisle 87]. SEQAP is designed as a one-way translator. In order to be able to generate natural language output from QAUZ, a variant of each original LESK expression is stored as well. This form consists of two lists: the input LESK expression (a list of LESK words) in which all occurrences of parameters have been replaced by stubs ('*'), and the list of replaced parameters ordered as they appear in the ori-

ginal expression. The separation of parameters from the rest of the expression is done for efficiency reasons. In the examples shown below, LESK statements will be represented in this (much more readable) form.

The rest of this section presents the relational view of information stored in the knowledge base. It does not necessarily reflect the structure of actual unit clauses (for efficiency some relations were represented by more than one type of a clause). The examples are based on the general 'ACCESS' statement for which the source frame is given in Figure 4.4.

STATEMENT:

Name LESK phrase identifying the Parser frame.

Purposes A list of LESK phrases representing mutually exclusive purposes of the statement.

Syntax The syntax of the statement is split into two parts at the point which distinguishes the substatements. In the example it is before the '0m' (zero or more occurrences) token. The substatement that has at least one occurrence of the repetitive entry has '1m' (one or more) as the type of the corresponding token in its syntax. If a statement has no substatements (i.e., the frame has no subframes) the second part of its syntax is an empty list.

Subframes A list of names of subframes (substatements). Since the statement hierarchy for the domain happens to be a binary tree, the

list is either empty or has two elements.

Superframe The name of the parent frame (if there is one).

PT-rules A list of rule labels. Each rule label must be unique within the frame.

PH-rules A list of rule labels.

Parameters A list of parameters introduced in the frame.

Errors A list of codes for error messages referred to in the rules of the frame.

Example: The information stored for the general 'ACCESS' statement.

```
Name:        ['ACCESS']
Purpose:     [[[access,a,single,file,[]],[[access,multiple,files,[]]]
Syntax:      [[[kw,'ACCESS'],
               [ac,[first_file_declaration,['FILE',?]]]],
              [['0m',n,2,
                [ac,['LINK_TO_part',['FILE',?]]],
                ['NSF',?]],
               [end]]]
Subframes:   [['ACCESS',with,a,'LINK_TO_option'],
              ['ACCESS',without,a,'LINK_TO_option']]
Superframe:  nil
PT-rules:    [t1,t2,t3,t4,t5,t6,t7,t8,t9,t10]
PH-rules:    [h1,h2,h3,h4,h5,h6,h7,h8]
Parameters:  ['FILE','DFILE','ALIAS','OPT','DDREC','NSF']
Errors:      ['ERRA1','ERRA2']
```

NONTERMINAL:

Name the name of the nonterminal.

Frame Indicates the frame in which the nonterminal is introduced, i.e., the frame to which its sequence of PT-rules belongs.

Sequence A list of sequence tokens, each of which is either a rule label or an ASL token. The augmented syntax of the nonterminal can be obtained by concatenating the ASL expressions involved (for a PT-rule this is its antecedent).

Context A list of pairs that describe contexts in which the nonterminal appears. The first element of the pair is the name of a parent syntactic component, and the second one is the type of the token in which the nonterminal occurs.

Example: The information stored for a file declaration.

```
Name:      file_declaration
Frame:     ['ACCESS']
Sequence:  [t3,t4,t5,t6]
Context:   [[first_file_declaration,nt],[subsequent_file_declaration,nt]]
```

PT-RULE:

Number The rule number consists of the name of the frame and the rule label. Since rule labels are unique within each frame, rule numbers are unique within the knowledge base.

Type Indicates whether the rule type is regular (r), error (e), or default (d).

Antecedent An ASL-expression.

Consequent A LESK statement.

Context The name of the nonterminal whose sequence contains the rule.

Example: The rule 't9' of the 'ACCESS' frame, which states that, if the 'OPTIONAL_ part' is present, then the records of the corresponding file are optional.

```
Number:      [['ACCESS'],t9]
Type:        r
Antecedent:  [[op,['OPTIONAL_part',[]]]]
Consequent:  [[records,of,*,are,optional],[['FILE',.]]]
Context:     subsequent_file_declaration
```

PH-RULE:

Number . The rule number consists of the name of the frame and the rule label.

Antecedent A LESK statement.

Consequent A LESK statement.

Example: The rule 'h1' of the 'ACCESS' frame, which states that, if a file is the single file, then the description of record complexes ('DDREC') is set to the record description of the file itself.

```
Number:      [['ACCESS'],h1]
Antecedent:  [[*,is,the,single,file],[['FILE',.]]]
Consequent:  [[*,is,set,to,the,dictionary_record,of,*],
                        [['DDREC',.],['DFILE',.]]]
```

PARAMETER:

Variable The variable that corresponds to the parameter.

Type The type constraint on the value of the parameter.

Subscript The applicable subscript ('nil' if a single occurrence parameter).

Kind Transparent (t) or hidden (h).

Syntax name The name used in syntax expressions (applies only to those parameters that appear in the ASL expressions).

Frame The frame in which the parameter is introduced.

Ext. name The LESK noun phrase used in communications with the user to refer to the parameter. This implicitly defines the grouping of parameters (e.g., 'DFILE' is the owner of 'ALIAS').

Example: The information stored for the 'ALIAS' parameter.

Variable:	'ALIAS'
Type:	file_name
Subscript:	n
Kind:	t
Syntax name:	name
Frame:	['ACCESS']
External name:	[[the,alias,of,*],[['DFILE']]]

INDEX:

Parameter The name of the parameter.

Location An indicator of where the parameter is referenced, e.g., 'tcause' for the antecedent side of a PT-rule.

Pointers A list of pairs consisting of a location of a reference to the parameter, and an indicator whether the parameter is assigned a value there ('?') or is assumed to already have a value ('o').

Example: The parameter 'OPT' occurs on the consequent (i.e., effect) side of PH-rules 'h7' and 'h8' in the 'ACCESS' frame.

Parameter:	'OPT'
Location:	heffect
Pointers:	[[[['ACCESS'],h7],?],[[['ACCESS'],h8],?]]

KEYWORD:

Keyword An uppercase word.

Frame The frame in which it is introduced.

Contexts A list of higher level syntactic components (nonterminals and/or top level statement) where the keyword is used.

Example: The keyword 'ALIAS' that occurs in the 'ALIAS_part' of the 'ACCESS' statement.

Keyword: 'ALIAS'
Frame: ['ACCESS']
Contexts: ['ALIAS_part']

ERROR MESSAGE:

Code The code given to the error message.

Text The actual message that QUIZ issues.

Explanation The canned explanation.

Frame The frame in which the error message is introduced.

Example: The information stored for the error message that is issued in case two files are given the same name.

Code: 'ERRA2'
Text: [[Invalid,file,name],[]]
Explanation: [[file,names,are,not,unique,-,need,aliasing],[]]
Frame: ['ACCESS']

5.4. Query-Answerer

In this section, we will first describe the way the queries are built and issued. Then, we will discuss the basic reasoning mechanisms, and approaches taken in answering some types of queries.

5.4.1. Query-Building Mechanisms

As defined in Section 3.2.3, a QAUZ query consists of a set of assumptions, (partial) QUIZ code, and a question. Assumptions and code are optional (for answering some types of queries, and even irrelevant, e.g., for syntax queries). They provide background information that may apply to more than one question pertaining to the same situation. Therefore, QAUZ allows the user to deal with each query component independently. This makes the query-building process both easy and efficient. There is also help available at every point of a QAUZ session at which the user input is expected.

5.4.1.1. Context Maintenance

The QAUZ interface provides the following commands that enable the user to enter, delete, or modify background information during the query-building process:

Update assumptions (a). When this command is selected, the set of currently active assumptions is shown and the user is given an opportunity to update the set by entering new assumptions and/or deleting some or all of the active ones. As each new assumption is entered, it is passed to SEQAP for

translation. If SEQAP detects either a lexical error (e.g., a word that is not found in its lexicon), or a violation of syntax, the user is informed about the error and the entry is ignored. An example is given in Figure 5.3.

```
> a
    ASSUMPTIONS:
                  1  'EMP' is a file
                  2  there is a selection_condition
    Update the assumptions:
                  3 >nothing is reported
                  4 >'NAME' an item
                  Illegal input - parsing failed
                  SEQAP: error(invalid_vp,[an,item])
                  (input ignored)
                  4 >'NAME' is an item
                  5 >del([2,3])
    ASSUMPTIONS:
                  1  'EMP' is a file
                  2  'NAME' is an item
                  3 >
```

Figure 5.3. Updating Assumptions: An Example

When the user exits the update process (by entering <cr> alone instead of a new assumption or a request), the updated set of assumptions is recorded and considered active until a new change to assumptions is made.

Update QUIZ Code (c). When this command is entered, if the context contains active code, the code is displayed and the user is prompted to select one

of the following three actions: keep, delete, or replace the code. If the user chooses to replace the code (the same applies when there are no active statements to start with), he/she is asked to enter a new sequence of statements. In order to prevent incorrect sequencing of statements in the code, the user is not allowed to delete individual statements. As each statement is entered, it is verified that its first word is a valid initial keyword for a statement. Further analysis of the statement is deferred until the query-answering process requires it. An example is given in Figure 5.4.

```
> c
    QUIZ CODE:
                    ACCESS EMP
                    REPORT NAME
    Action?  (<cr>=keep, r=replace, d=delete)  > r
    Enter your QUIZ code:
                 >> ACCESS EMPLOYEES
                 >> SELECT IF DATE1 GT 870101
                 >> RPRT ALL
                    Not a statement (input ignored)
                 >> REPORT ALL
                 >>
```

Figure 5.4. Updating QUIZ Code: An Example

Another way the user can replace the code is by entering a statement of the following form among the assumptions:

I used "<stmt1>;<stmt2>;..."

The statements are processed in the same way as when entered through the code update command. Figure 5.5 shows an example of the use of this feature.

```
> a
    ASSUMPTIONS:
                1  'EMP' is a file
                2  'NAME' is an item
    Update the assumptions:
                3 >I used "ACCESS EMP; REPORT ALL"
                          ACCESS EMP
                          REPORT ALL
                  (the above code is now in effect)
                3 >there is a wraparound
                4 >
```

Figure 5.5. Updating QUIZ Code Through Assumptions: An Example

Show assumptions and code (s). Sometimes during the consultation, the user needs be reminded of the currently active background information. Although this can be done through a combination of the commands described above, QAUZ provides a more convenient command that lists all active assumptions and QUIZ statements, as shown in Figure 5.6.

```
> s
    ASSUMPTIONS:
                        'EMP' is a file
                        'NAME' is an item
                        there is a wraparound
    QUIZ CODE:
                        ACCESS EMP
                        REPORT ALL
```

Figure 5.6. Showing Assumptions and Code: An Example

Delete all assumptions and code (d). Just as the above command provides the user with an easy way to see the background information, the delete command can be used to clear the background when the user wants to start building a new query that is unrelated to the previous ones.

If a set of assumptions applies to a sequence of consecutive queries, it is desirable to avoid repeating the basic portion of its analysis. Therefore, an indicator of whether or not the set of assumptions is 'new', or 'old' (i.e., has been already analyzed within another query), is added to the context. A separate indicator with the same purpose is maintained for the QUIZ code component of the background.

5.4.1.2. Issuing a Question

When the user decides that the background information is specified as desired, he/she enters a question through the 'q' command, or by typing in the question directly at the command level. The former method is more convenient for a novice user because some help for formulating the question is available at the next level prompt. The question is translated by SEQAP into the same form as the assumptions, with the addition of the question type.

The following is the list of accepted question patterns:

(WHY)	Why ...?
(WHY NOT)	Why ... not ...?
(ERR)	Why did I get the error message ...?
(HYP)	What happens if I use ...?
(WHEN)	When is it true that ...?
(SYN1)	What is the syntax of ...?
(SYN2)	What is the complete syntax of ...?
(SYN3)	What is ... a component of?
(SYN4)	Where is ... used?
(SYN5)	What are the different kinds of ... statement?
(SYN6)	What kinds of statements are there?

A successful translation of the question completes the query-building process. The answer-generator is invoked at this point to answer the query.

The answering strategy that corresponds to the individual question type is discussed in the remaining sections of this chapter and in Chapter 6. An example of a complete QAUZ query-answering session is given in Appendix B.

5.4.2. Deduction Engines

For answering various types of queries, the system needs the ability to both make conclusions based on individual pieces of information contained in the query, and to reason about their interactions. In order to determine the effects of the code, and to deduce the causes of the report characteristics, the answer generator requires the appropriate reasoning mechanisms based on forward and backward chaining of rules, respectively. In this section we will describe the major aspects of the implemented mechanisms.

5.4.2.1. Code Analysis = Parsing + Forward Chaining

Whenever a query-answering strategy requires knowledge about the code that is used in a given situation, the code analysis mechanism is invoked. It processes the supplied QUIZ statements sequentially, and makes conclusions about the effects each of them has on the result. The analysis of an individual QUIZ statement combines parsing with reasoning about the effects of each fragment of the parse, as shown in Figure 5.7 and described below. A trace of the analysis is offered as the answer to the 'HYP' type of queries.

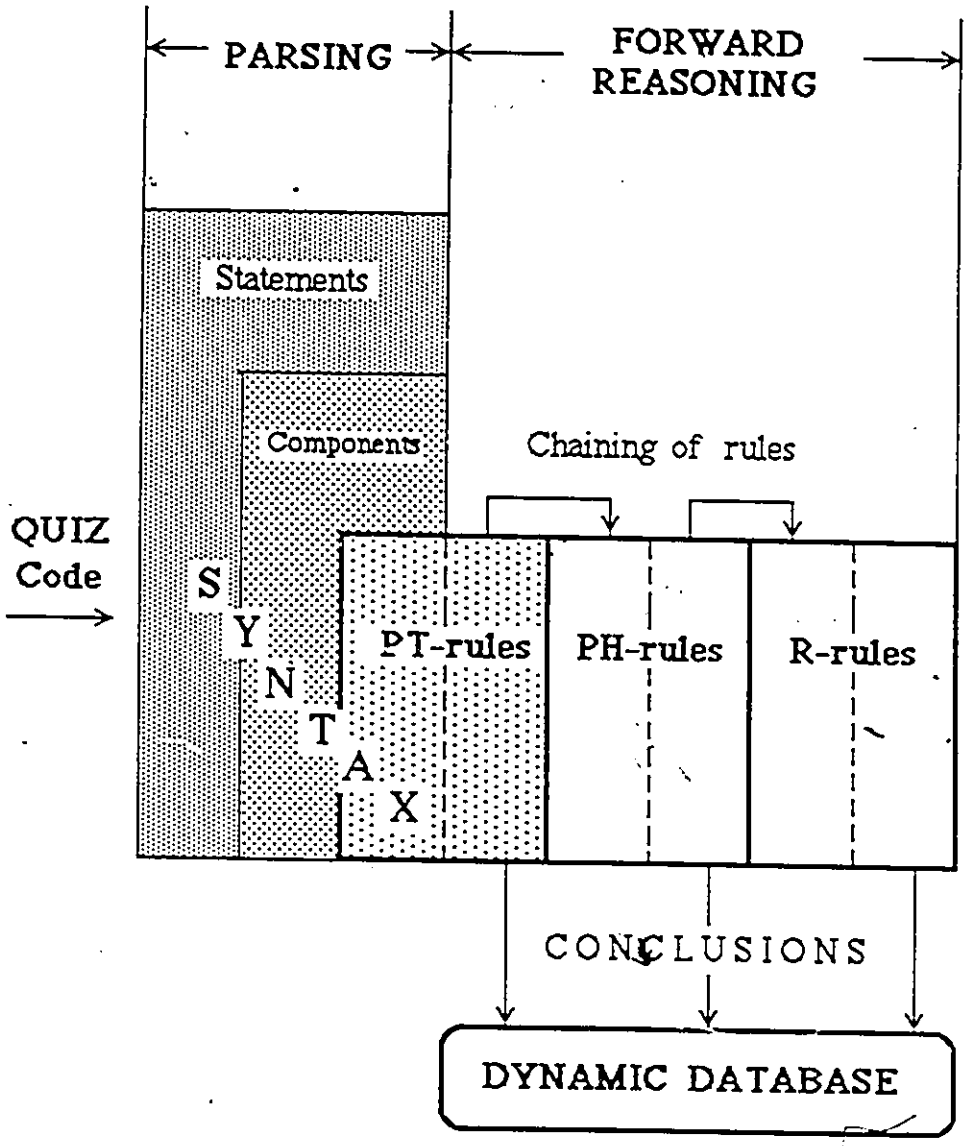


Figure 5.7. The Code Analysis Process

5.4.2.1.1. Parsing QUIZ Statements

The parsing process is driven by the augmented syntax contained in the appropriate frame. Initially, the appropriate frame is the top level frame for the (most general) statement type to which the given statement belongs. It is determined by the initial keyword of the statement. As discussed in Section 5.3.3., the full syntax of the statement is represented at three levels: statement level (the 'syntax' slot), component level (the 'sequence' slot), and rule level (the 'antecedent' slot for PT-rules). The ASL components represented at a lower level are retrieved when required. When and if the parsing reaches the point in the augmented syntax at which it can be concluded that the statement is of a more specific statement type, the process continues guided by the syntax from the appropriate subframe. After the statement has been successfully parsed, the last (i.e., the lowest level) frame applied determines the actual statement type.

A successful parsing step, done based on an ASL terminal that contains a variable, results in the assignment of the value to the corresponding parameter. The knowledge base contains information about parameter types and contexts in which they may occur. The analysis of an instance of a statement, however, has to deal with individual occurrences of parameters. Therefore, as a variable is encountered in the ASL terminal, a new parameter is created and is assigned the value obtained from the input.

An occurrence of a parameter is uniquely and uniformly identified by an integer subscript and a qualifier. The qualifier consists of:

1) the (unique) label of the statement being parsed, and

2) the shortest sequence of (subscripted) parameters, constructed based on the ownership hierarchy and the context of the parameter.

For example, 'ALIAS' needs only the statement label as the qualifier, while 'ITEM' in a 'SELECT' statement, requires either 'CONT' or 'EXPR', in addition to the statement label (because it can appear in more than one component of the condition).

5.4.2.1.2. Applying Rules

As discussed above, the parsing process uses the antecedents of regular and error PT-rules. Whenever a parsing step based on the antecedent of a regular PT-rule is completed, either the rule itself is fired (if the antecedent does not end with an option, or the option is present in the statement), or the corresponding default PT-rule is fired (otherwise). The consequent of the rule is instantiated, and the obtained statement is added to the set of conclusions. This may initiate a rule chaining process in which PH-rules and/or R-rules are fired, adding more conclusions about the effects of the code, as shown in Figure 5.7. Thus, forward reasoning in QAUZ can be viewed as a side effect of code parsing.

In order to prevent a combinatorial explosion, forward chaining with R-rules is limited to a certain number of steps (3 in the implemented QAUZ prototype).

The antecedent of each error PT-rule ends with a condition. When, while

parsing, a condition is encountered in the ASL expression, it is instantiated, and one of the following situations arises:

a) The condition is found to be satisfied because it matches an assumption. The statement is labeled 'ASSUMED'.

b) The condition is found to be satisfied through actual verification. This process involves procedural knowledge and should be performed by invoking the procedural component of the Advisor. Since this component was not planned to be implemented in the initial Advisor prototype, a set of simple verification modules (e.g., for verifying that a file name is unique) has been created within the QAUZ prototype. The statement is labeled 'VERIFIED'.

c) The condition contradicts an assumption, or its negation can be verified. In such a case, the corresponding error PT-rule is fired, resulting in the conclusion that a particular error message is applicable. The parsing of the statement is terminated.

d) There is not enough information to determine whether the condition is satisfied or not. Therefore, its truth value remains unknown and the statement is labeled 'UNCERTAIN'.

5.4.2.1.3. Model of QUIZ Code

A trace of the reasoning described above is kept in a dynamic database that,

like the knowledge base, consists of Prolog unit clauses. There are three major types of recorded information:

1) QUIZ statements, numbered sequentially and labeled 'active' or 'canceled' (as described in Section 5.4.2.1.5.);

2) parameter-value pairs (for all created instances of parameters);

3) a sequence of conclusions.

With each conclusion in the database, the following information is recorded: its origin (e.g., the rule number), the QUIZ statement it applies to (the statement number), and its type (e.g, 'stmt' for the conclusion about the type of the QUIZ statement).

The portion of the dynamic base created through parsing and forward reasoning can be thought of as an internal model of the analyzed code. In the query-answering context, it will be referred to as the *C-model.*

5.4.2.1.4. Answering 'HYP' Queries

The question patterns for hypothetical ('HYP') queries are:

What happens if I use "<stmt1>;<stmt2>..."?

and

What happens if I use the code?

In the latter case, the code is assumed to be the one currently active (if there is

none, the interface prompts the user to enter the code). In both cases, however, the resulting query consists of a sequence of QUIZ statements (the code), and an optional set of assumptions. The question, although present, is ignored after being processed by SEQAP, since it does not contain information other than the question type ('HYP'), and the code (the first pattern only).

The answering strategy for this type of query is based exclusively on the analysis of code, as it is described in the previous sections. The code analysis mechanism can be used in two modes: with an easy-to-follow trace of the process displayed to the user, or without the trace (in both cases the C-model is constructed). The former is used for answering 'HYP' queries, while the latter is applied as a step in answering some of the other types of queries (where the C-model contributes to the answer, but its details are not of interest to the user).

Figure 5.8 shows an example of an answer to a 'HYP' query. The conclusions are numbered and displayed to the user in the order in which they are arrived at. Chaining of rules is shown explicitly ('therefore' and indentation are used to show chaining) to better indicate the causal relationships. Each condition is labeled 'ASSUMED', 'VERIFIED', or 'UNCERTAIN', depending on the result of its evaluation.

The content and the format of answers to 'HYP' queries are particularly suitable for beginning-level users. These answers provide instant and detailed feedback concerning the effects of the code, thus increasing the user's understanding of the language, as well as teaching the user the proper QUIZ terminology.

```
<Assumption: 'BILLINGS' is described in the dictionary as a file>

  > What happens if I use the code?

     ACCESS EMPLOYEES ALIAS STAFF LINK TO BILLINGS OPTIONAL
     SELECT IF DATEJOINED GT 810101
     REPORT ALL

     Analyzing: ACCESS EMPLOYEES ALIAS STAFF LINK TO
                                          BILLINGS OPTIONAL  (ST1)

      (1)  EMPLOYEES is described in the
           dictionary as a file (UNCERTAIN)
      (2)  STAFF is the alias of EMPLOYEES
      (3)  STAFF is a unique file (VERIFIED)
      (4)  STAFF is a declared file, therefore
      (5)       STAFF is an accessible file
      (6)  records of STAFF are required
           For file 2 :
      (7)     BILLINGS is described in the dictionary as
              a file (ASSUMED)
      (8)     BILLINGS has no alias
      (9)     BILLINGS is a unique file (VERIFIED)
     (10)     BILLINGS is a declared file, therefore
     (11)          BILLINGS is an accessible file
     (12)     BILLINGS is a subordinate file, therefore
     (13)          a link for BILLINGS is established
                   through an internal procedure
     (14)     records of BILLINGS are optional
     (15)  STAFF is the primary file
     (16)  ST1 is an 'ACCESS with a LINK_TO_option' statement

     Analyzing: SELECT IF DATEJOINED GT 810101  (ST2)

     (17)  DATEJOINED is described in the dictionary
           as an item (UNCERTAIN)
     (18)  a default for the qualifier is used, therefore
     (19)       the qualifier is determined through an
                internal procedure
     (20)  the type of DATEJOINED is equal to the type
           of 810101 (UNCERTAIN)
     (21)  there is a selection_condition on record_complexes
     (22)  ST2 is a 'SELECT IF' statement

     Analyzing: REPORT ALL  (ST3)

     (23)  all record_items are requested for reporting
     (24)  ST3 is a 'REPORT with the ALL_option' statement
```

Figure 5.8. Answer to a 'HYP' Query: An Example

5.4.2.1.5. Truth Maintenance

Expert reasoning is often nonmonotonic. That is, it requires the making and retraction of assumptions [Hayes-Roth et al. 83]. *Truth maintenance* is the process by which an expert system, in response to new information, revises its current set of assumptions (or beliefs) in order to avoid contradictions (see, for example, [Doyle 79]). This process typically involves backtracking in which inconsistencies are traced back to the inferential steps that created them. Such a process requires the system to keep, for each belief, a *dependency record* that consists of the belief and its justification in terms of an inference rule and other beliefs.

In the system like QUIZ, which processes statements sequentially and updates the partial model of the report accordingly, each statement conveys new information. The order in which the statements appear within a QUIZ program (between 'ACCESS' and 'GO') is irrelevant as long as the program contains at most one instance of each of the general statement types (assuming that 'SELECT with a file option' statements for different files are considered to be of different types). Otherwise, each statement cancels the previous statement of the same type, while an 'ACCESS' statement cancels all preceding statements. In order to account for this aspect of QUIZ, QAUZ has a simple truth maintenance mechanism that cancels a statement by changing its status from 'active' to 'canceled', and removes all conclusions made based on the presence of the statement. Figure 5.9 shows the answer to a hypothetical query regarding a piece of code that contains canceled statements.

Another component of the system that could require a truth maintenance mechanism is the code analysis process. Parsing (Mini)QUIZ is deterministic except for a 'REPORT with a report group' statement containing one or more occurrences of the 'SKIP_part' (e.g., the role of that component in 'REPORT X SKIP PAGE Y' and 'REPORT X SKIP PAGE' is quite different, as demonstrated in Appendix B). Trying to keep the parsing process simple, while not compromising the quality of an answer to a 'HYP' query, we decided to treat all conclusions about any repeated entry (such as a 'report_group') as tentative until its parse is completed. If it succeeds, the conclusions are confirmed (and shown to the user, if appropriate), else they are ignored.

```
> What happens if I use the code?

   ACCESS EMPLOYEES
   ACCESS EMPLOYEES ALIAS STAFF
          LINK TO BILLINGS OPTIONAL
   REPORT LASTANAME
   SELECT STAF
   REPORT ALL

   Analyzing: ACCESS EMPLOYEES  (ST1)

    (1)  EMPLOYEES is described in the
         dictionary as a file (UNCERTAIN)
                              :
    (8)  ST1 is an 'ACCESS without a LINK_TO_option'
         statement

   Analyzing: ACCESS EMPLOYEES ALIAS STAFF
                      LINK TO BILLINGS OPTIONAL  (ST2)
   NOTE: All previous statements are canceled!

    (1)  EMPLOYEES is described in the dictionary
         as a file (UNCERTAIN)
                              :
   (16)  ST2 is an 'ACCESS with a LINK_TO_option'
         statement

   Analyzing: REPORT LASTANAME  (ST3)

         For report_item 1 :
   (17)     a default for the tab_column is used, therefore
   (18)          the tab_column is determined
                 through an internal procedure
                              :
   (30)  ST3 is a 'REPORT with a report_group' statement

   Analyzing: SELECT STAF  (ST4)

   (31)  error message ERRS1 is issued:
          " Expected : file IF eol"
    (The rest of the statement is ignored)

   Analyzing: REPORT ALL  (ST5)
   NOTE: ST3 is canceled!

   (32)  all record_items are requested for reporting
   (33)  ST5 is a 'REPORT with the ALL_option' statement
```

Figure 5.9. Canceled Statements: An Example

5.4.2.2. Backward Reasoning: Exploring LESK Statements

Given a LESK statement, the backward reasoning mechanism of QAUZ can be invoked to establish links between the statement and all those features of QUIZ code that are potential causes of the state described by the statement. In query-answering, it is used whenever an applied strategy calls for exploration of an assumption or a question.

We will first discuss the mechanism in some detail, and then describe how the structure it builds can be used for answering 'WHEN' type of queries. The role of backward reasoning in answering other types of queries will be discussed in the next chapter.

5.4.2.2.1. Deduction Tree of a Statement

Backward reasoning from a LESK statement creates an AND/OR tree which represents all possible ways the truth of the statement can be established. The root of the tree is the statement itself. Other nodes, each for one step in backward reasoning, are generated as follows:

a) **Determine Relevant Rules.** A set of relevant rules is determined by the parameters contained in the statement, and the index part of the knowledge base. For the root, the type of the statement is an additional constraint (for example, only 'decr' and 'prop' quantity-related rules can be relevant to a statement that refers to a 'small' number of certain objects).

b) **Determine Applicable Rules.** The statement is matched against the consequent of each of the relevant rules (the internal representations of statements (produced by SEQAP) are matched). The obtained set of applicable rules represents all alternative ways the statement can be established. For each such rule, an AND-node is generated as a child of the statement node.

c) **Apply the Rules.** Each AND-node generated in the previous step is expanded by generating a child OR-node for each of the statements contained in the antecedent of the rule.

Further reasoning is done by expanding each of the newly created OR nodes through the application of the above three steps, and continuing the process until no more nodes can be generated (i.e., until none of the leaf nodes matches the consequent of some rule). The resulting tree is termed the *deduction tree* of the original (root) statement. Each leaf node that is generated as a result of an application of a PT-rule contains an ASL expression that defines a feature of some QUIZ statement. As an example, the deduction tree for the statement "the column heading is set to the dictionary heading" is shown on Figure 5.10 (conjunctions are indicated by arcs). The nodes are numbered as generated.

The nodes of the deduction tree are represented in the dynamic database as Prolog unit clauses. In addition to the node number and the content, (i.e., a LESK statement or an ASL expression (OR-nodes), and the rule number (AND-nodes)), the arguments of the clause include parent and children pointers, and other information useful for efficient retrieval of nodes that have certain characteristics (e.g.,

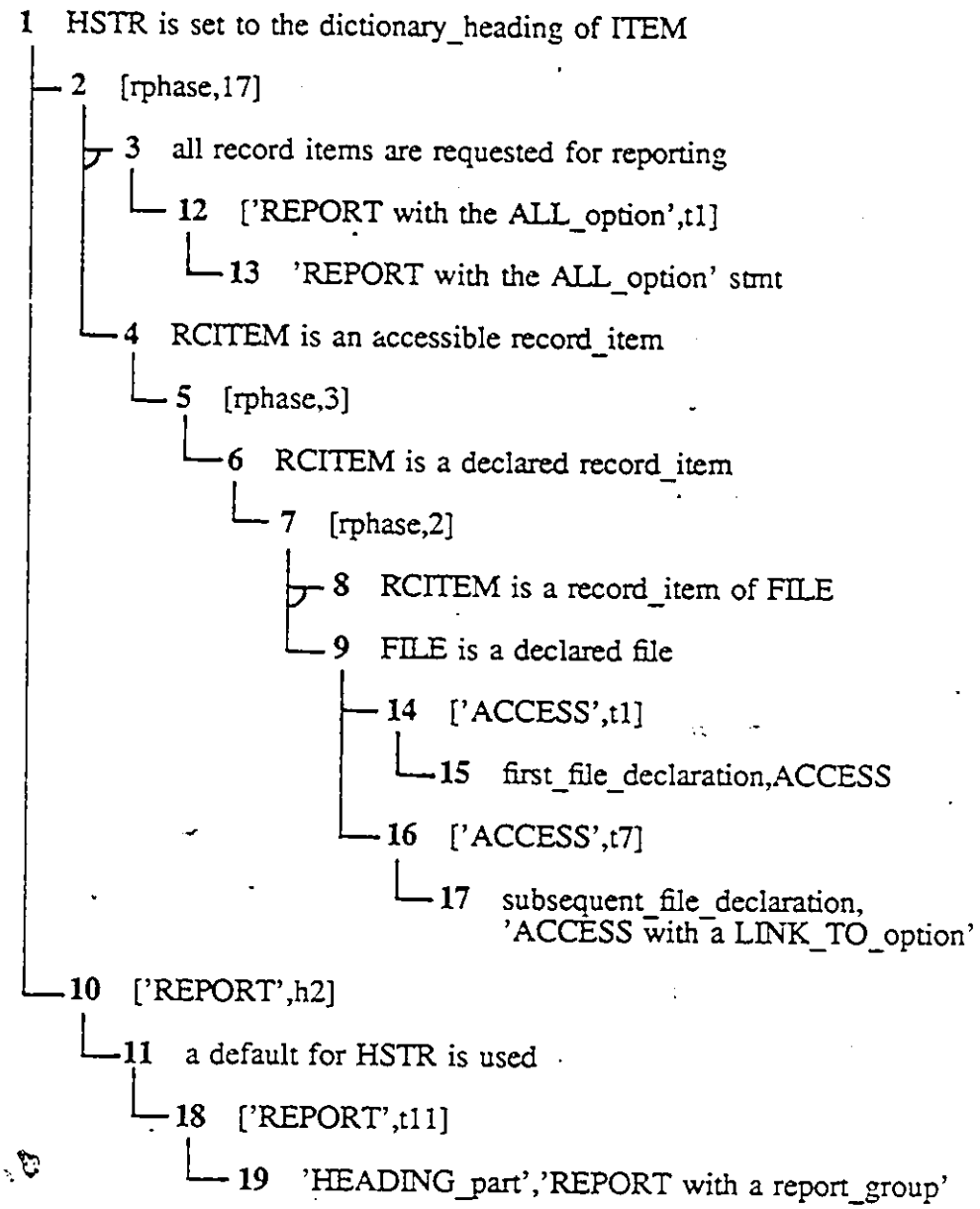
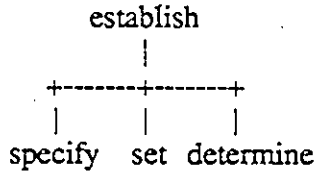


Figure 5.10. The Deduction Tree of a Statement: An Example

rule type, node type).

5.4.2.2.2. Reasoning Based on Generalization

The Advisor team has been developing a hierarchy of 'activities', one of the two major kinds of topic the 'HDI' module recognizes (the other kind is 'objects'). This hierarchy can be used by the QAUZ reasoning mechanisms as a basis for application of the following reasoning rule: If a statement containing activity X holds, so does the statement obtained by replacing X by its superactivity (since generalization preserves the truth value of a statement). If an OR-node cannot be expanded using the rules from the knowledge base, an attempt can be made to expand it through the above rule. Since in the initial version of the prototype the two modules are not integrated, to demonstrate this feature, we have implemented the following simple hierarchy:



Those nodes that are generated this way but that can not be further expanded are removed from the tree when the reasoning process is completed.

Examples of answers based on deduction trees that contain nodes generated through the generalization rule are given in Appendix B.

5.4.2.2.3. Answering 'WHEN' Queries

A 'WHEN' query is a query with a question of the following form:

"When is it true that ...?"

Any active assumptions or code are ignored (but not deleted from the context). This question pattern was not found in the analyzed sample of real questions, but we believe that it should be available: it shows the user all possible causal links between the code and results, outside the context of a specific problem. This is particularly useful when the user is puzzled by the answer to a 'WHY' or 'WHY NOT' query. Answering involves only the generation of a deduction tree (often already available from a previous query), and a convenient display of its content. This type of a question is also useful to the knowledge engineer for the examination of the knowledge base contents and experimenting with it.

The display of the deduction tree is a formatted trace of its preorder traversal. The level of each node is indicated through proper indentation. An example of a 'WHEN' query is given in Figure 5.11. The statement contained in the question is the same as the one used in the example of a deduction tree (Figure 5.10).

Nodes that contain LESK statements are presented in a natural language form generated from LESK through recursive substitutions of external names for parameters. Simple linguistic rules are used to insert articles where appropriate. Some sentences may seem rather long, but this is the price to be paid to avoid potential ambiguities.

```
> When is it true that the column heading is set to
  the dictionary heading?


  the column_heading of a report_item is set to the
  dictionary_heading of an item,
    CONSEQUENCE OF:
       all record_items are requested for reporting,
         CONSEQUENCE OF:
            using 'REPORT with the ALL_option' statement
       AND
       a record_item is an accessible record_item,
         CONSEQUENCE OF:
            the record_item is a declared record_item,
              CONSEQUENCE OF:
                 the record_item is a record_item of a file
                 AND
                 the file is a declared file,
                   CONSEQUENCE OF:
                      using first_file_declaration
                      in 'ACCESS' statement
                    OR
                      using subsequent_file_declaration
                      in LINK_TO_part
                         in 'ACCESS with a LINK_TO_option'
                            statement
    OR
      a default for the column_heading of the
      report_item is used,
        CONSEQUENCE OF:
           using format_part without HEADING_part
           in report_item
              in report_items
                 in report_group
                    in 'REPORT with a report_group' statement
```

Figure 5.11. Answer to a 'WHEN' Query: An Example

A leaf node that contains an ASL-expression is represented by a description of the syntactic feature (e.g., 'format_part without HEADING_part'), followed by the context it appears in, up to and including the appropriate statement type.

5.4.3. Answering Syntax Queries

Queries about the syntax of QUIZ have nothing to do with reasoning about causal relationships present in the language. However, since syntax is explicitly represented in the QAUZ knowledge base, this module has the basis for answering syntax queries.

Mechanisms have been implemented to provide answers to all six types of syntax questions (a query consists only of the question). Syntax is presented using the conventions from the QUIZ manual. The following information is contained in the answer for each of the question types ('se' is a syntactic entity; 'st' is a statement type):

SYN1 Question: "What is the syntax of <se>?"

Answer: The top level syntax of the entity. If the entity is a terminal, its type is displayed (e.g., a keyword).

SYN2 Question: "What is the complete syntax of <se>?"

Answer: The full syntax tree of the entity.

SYN3 Question: "What is <se> a component of?"

Answer: The parent(s) of the entity, i.e., the immediate context in which the entity may appear.

SYN4 Question: "Where is <se> used?"

Answer: The ancestor(s) of the entity, i.e., the full context in which the entity may appear.

SYN5 Question: "What are the different kinds of <st> statement?"

Answer: The subtree of the statement hierarchy below the specified statement type.

SYN6 Question: "What kinds of statements are there?"

Answer: The complete hierarchy of statement types.

Some examples of typical questions and answers are presented in Figure 5.12. More examples can be found in Appendix B.

```
> What are the different kinds of ACCESS statement ?

        ACCESS
             ACCESS with a LINK_TO_option
             ACCESS without a LINK_TO_option

> What is the syntax of ACCESS statement ?

         ACCESS first_file_declaration [LINK_TO_part]...

> What is the complete syntax of ACCESS statement ?

        ACCESS
        first_file_declaration
             file_declaration
                  file
                  [ALIAS_part]
                       ALIAS
                       name
        [LINK_TO_part]...
             LINK
             TO
             subsequent_file_declaration
                  file_declaration
                       file
                       [ALIAS_part]
                            ALIAS
                            name
                  [OPTIONAL_part]
                       OPTIONAL

> What is file_declaration a component of ?

        first_file_declaration
        subsequent_file_declaration

> Where is file_declaration used ?

    'file_declaration' is

        in--> first_file_declaration
              in--> 'ACCESS' statement
        in--> subsequent_file_declaration
              in--> LINK_TO_part
                    in--> 'ACCESS with a LINK_TO_option'
                          statement
```

Figure 5.12. Answer to a 'SYN' Query: Examples

6. QUERY-ANSWERING STRATEGIES

6.1. Answering 'WHY' Queries

A 'WHY' query consists of a positive (i.e., unnegated) question and of optional background information supplied as a set of assumptions and/or QUIZ code. An answer to such a query is produced in two phases. First, the information submitted by the user is analyzed and a causal model of the query (the Q-model) is built. Next, an appropriate answer is delivered to the user, unless no causal knowledge relevant to the question could be found in the knowledge base.

6.1.1. Phase 1: Building the Q-model

The algorithm that provides a basis for answering 'WHY' queries consists of the following six major steps (the code analysis process described in the previous chapter will be referred to as 'parsing', for short):

(1) Create initial D-models of assumptions

(2) Create the initial D-model of the question

(3) Pre-parse evaluation

(4) Parse code

(5) Post-parse evaluation

(6) Prune-evaluate-propagate loop

This process normally terminates after all six steps have been successfully completed, but a premature termination occurs as soon as an error has been

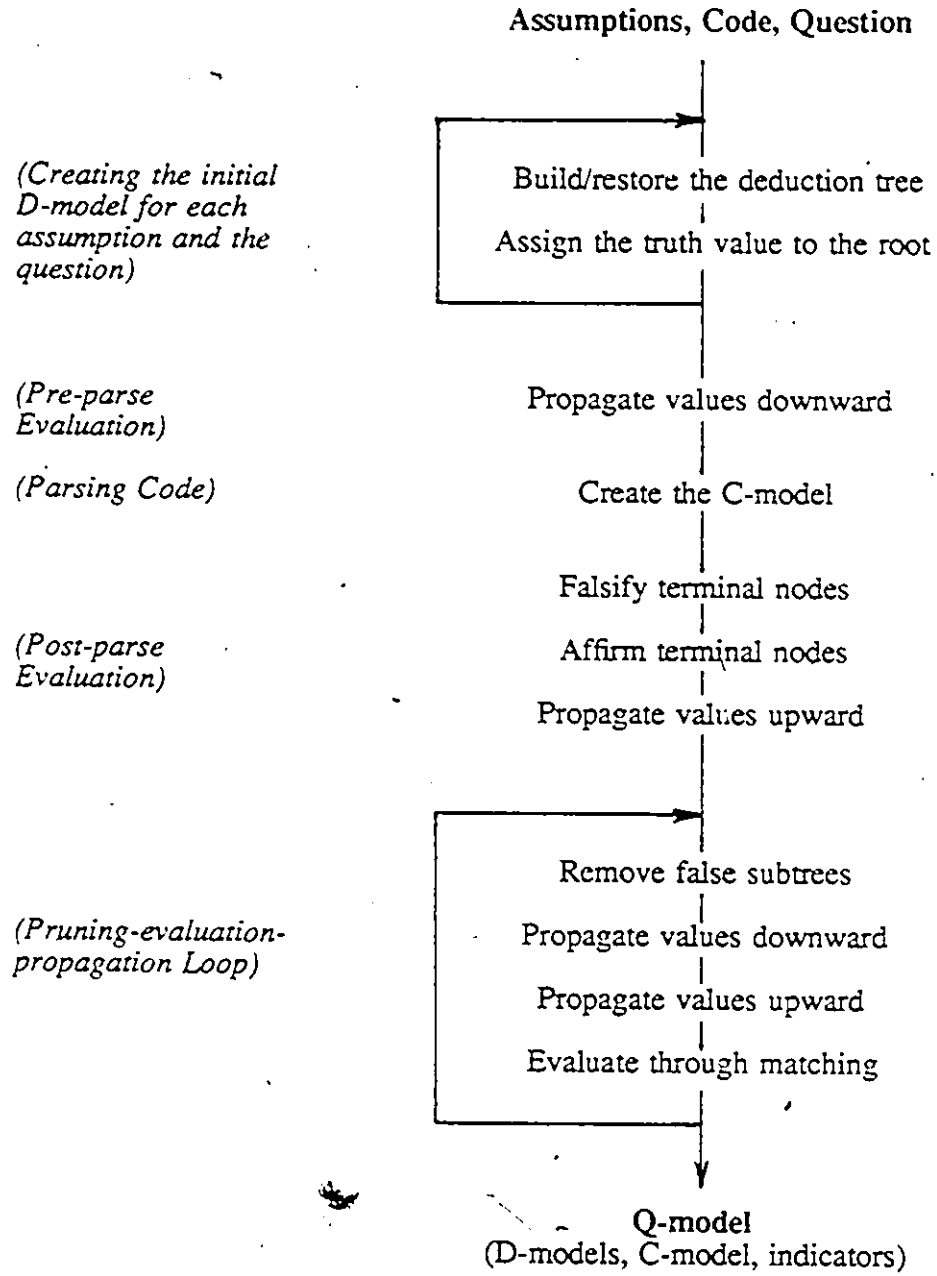


Figure 6.1. Building the Model of a Query (the Q-model)

detected in the QUIZ code or the system has determined that the query contains contradictory information.

The process of building the model of a query is presented in Figure 6.1. In the remainder of this section the individual steps of the algorithm will be described.

6.1.1.1. Creating Initial D-models of Assumptions

The *initial D-model* (or deduction model) of an assumption is an AND/OR tree built in the following manner:

a) The deduction tree is built for the positive form of the statement using the backward deduction mechanism described in the previous chapter.

b) The root node of the deduction tree is labeled according to the sign of the original statement: 'true' if the statement is positive, 'false' otherwise (i.e., if the main vero is negated).

c) It is verified that the statement does not contradict any assumption. A contradiction is detected if the given statement and some assumption form a pair of statements that satisfies one of the following two conditions:

   1) The statements match but the assigned truth values are different (e.g., 'the file has an alias' and 'the file does not have an alias' are two contradictory statements). There are two pairs of contradictory quantifiers: 'no' - 'all', and 'no' - 'nil' ('nil' is the default quantifier).

2) The statements have the same truth values and their translations match except for the contradictory quantifiers (e.g., 'all report complexes are printed' and 'no report complexes are printed' are contradictory).

An initial D-model is built for each assumption as described above and the set of resulting models is saved in case it is to be used later. If the initial D-models of assumptions have been built already, then they are restored rather than built again: this typically happens when the same set of assumptions has been used in the context of another query and has not been changed since.

6.1.1.2. Creating the Initial D-model of the Question

The initial D-model of a 'WHY' question is built in much the same way as the initial D-model of a positive assumption.

6.1.1.3. Pre-parse Evaluation

The initial D-model of a LESK statement (i.e., an assumption or the question) resulting from the above two steps is an AND/OR tree with the root labeled 'true' or 'false', while other nodes have no truth value assigned. In this step, each initial D-model is partially evaluated by propagating truth values from the root towards terminal nodes. The propagation is done systematically by preorder traversal of the model tree. This is the first step towards transforming the initial D-model of a statement into a structure that will be the model of the statement within the model of the query. We will refer to intermediate versions of the model as the *evolving D-model* of the statement.

6.1.1.4. Parsing Code

If the code and assumptions have not been changed since the last time parsing was done, there is no need to perform this step of the algorithm. Skipping this step should, normally, save a lot of time during a question-answering session since it is natural to expect more than one question about the problem at hand based on the same background information. To make this saving possible, we have introduced the restriction that the question may not affect the code parsing process. The verification of conditions related to the code can, however, use evolving D-models of assumptions as an external source of information.

The model of QUIZ code built in this step is called the *C-model* of the query.

6.1.1.5. Post-parse Evaluation

Evolving D-models of assumptions and the question are further revised using information obtained from the examination of the supplied QUIZ statements as described below. This step is skipped if no code is given in the query.

Falsifying Terminal Nodes. As the final conclusion in parsing an individual QUIZ statement, the system decides the type of the statement (e.g., 'ACCESS A LINK TO B' is an instance of an 'ACCESS with a LINK_TO option' statement). Due to the non-procedural character of QUIZ, at most one statement of each general statement type is active. We can, therefore, affirm or falsify some terminal nodes of the evolving D-models.

Each terminal node that is generated through an application of a Parser rule always refers to the relevant statement type. If there is a statement type in the code that is different from the statement type associated with the terminal node, but both are kinds of the same top-level QUIZ statement type and none is a superstatement of the other, the terminal node can be falsified. For example, if the code contains an 'ACCESS without a LINK_TO option' statement and the evolving D-model of an assumption/question has a terminal node that suggests the existence of an 'ACCESS with a LINK_TO option' statement, we can proceed by labeling that node as 'false'. There is an exception to the above with regards to 'SELECT' statements: the program may have a 'SELECT without a file_option', and a 'SELECT with a file_option' for each declared file. An easy way to deal with this exception is to treat them as separate top-level statement types.

A terminal node can also become falsified if the node has been generated using a certain Parser rule, and the code contains a statement of a type to which that rule is relevant, but the C-model indicates that the rule has not been used. The only statement type to which this falsification rule does not always apply is the 'SELECT with a file_option', because it may have multiple occurrences, not all of which are supplied as evidence.

Affirming Terminal Nodes. A terminal node is affirmed (i.e., assigned the truth value of 'true') if it has been generated by a rule used in parsing, provided that parameters are not ambiguous. An ambiguity arises if a parameter has multiple occurrences in the code provided by the user and there is not

enough information in the rest of the query to decide to which occurrence the node should refer.

Upward Propagation. Truth values assigned in the previous two steps to terminal nodes of evolving D-models are propagated towards the corresponding roots. The propagation is done by postorder traversal of each tree. A contradiction is detected when an attempt is made to propagate a value to a parent that already has the opposite value.

6.1.1.6. Pruning-Evaluation-Propagation Loop

The following sequence of steps is performed until an iteration fails to make any changes to evolving D-models of LESK statements:

a) Remove false subtrees

b) Propagate truth values downward

c) Propagate truth values upward

d) Evaluate through matching

The individual steps are discussed below. The execution of the loop must terminate because both the total number of nodes and the number of unlabeled nodes are nonincreasing within the loop.

Removing False Subtrees. After being partially evaluated in the previous steps, each AND/OR tree is pruned above false nodes that have a non-false parent node. This can be justified by the fact that the eliminated subtrees

would not contribute significantly to the answer while their presence burdens the reasoning process.

Downward Propagation. This is the same process that was used in the pre-parse evaluation. However, in contrast with that earlier step, a check for potential contradictions is now necessary. A contradiction is detected during this propagation process when an attempt has been made to propagate a value to a child node already assigned the opposite value.

Upward Propagation. This is the same value propagation process as that used in post-parse evaluation.

Evaluation through Matching. For each OR-node that still has no truth value, an attempt is made to assign a value to the node by finding an affirmed/falsified matching node in another evolving D-model or a matching node in the C-model. Matched are translations of LESK statements associated with the nodes. Once the truth value of the node has been changed, both its parent node and its children nodes are checked for contradiction. If a code is not given as part of the query, this evaluation step is performed immediately after the removal of false subtrees (for efficiency reasons).

6.1.2. Phase 2: Delivering the Answer

The final AND/OR tree created for a LESK statement (an assumption or a question) in the first phase of the query-answering process is the (final) *D-model* of

the statement. The triple consisting of the set of D-models (of assumptions and the question), the C-model, and a set of indicators describing the course of the process (e.g., contradiction indicator) is the model of the query or the *Q-model.*

In the second phase, the system generates an answer to the query depending on the characteristics of the obtained Q-model.

6.1.2.1. Regular Answer

The optimal case in answering a 'WHY' query is one in which QUIZ code has no errors, the query is consistent, and the knowledge relevant to the question has been found in the knowledge base. In such a case the D-model of the question is an AND/OR tree with the root labeled 'true' and with each of the remaining nodes either labeled 'true' or left unlabeled. This will be illustrated through an example. Suppose that the user supplies the evidence that he/she used the 'REPORT ALL' statement, gives no other background information, and asks the following question:

"Why is the column heading set to the dictionary heading?"

A trace of the reasoning process performed in Phase 1 to build the deduction tree for the question statement and transform the tree into the D-model of the statement is shown in Figure 6.2. The resulting D-model is presented in Figure 6.3. An indicator of the step that assigned the value to the node is included in the label of the node ('upward', 'downward', and 'code' denoting upward propagation, downward propagation, and assignment based on the C-model, respectively). It embo-

```
Applying reporter-rules to the question
Applying parser-rules
Assigning truth values to roots
       true: HSTR is set to the
             dictionary_heading of ITEM  (1)
Pre-parse evaluation
   Downward propagation
Parsing code
Post-parse evaluation
   Assigning truth values to terminal nodes
       false:  using format_part
               without HEADING_part (19)
       true:  using 'REPORT with the ALL_option'
              statement (13)
   Upward propagation
       true: [REPORT with the ALL_option, t1] (12)
       true: all record_items are requested for
             reporting  (3)
       false: [REPORT, t11] (18)
       false: a default for HSTR is used  (11)
       false: [REPORT, h2] (10)
Pruning-eval-propagation loop
   Removing false subtrees
       above: [REPORT, h2] (10)
   Downward propagation
       true: [rphase, 17] (2)
       true: RCITEM is an accessible record_item  (4)
       true: [rphase, 3] (5)
       true: RCITEM is a declared record_item  (6)
       true: [rphase, 2] (7)
       true: RCITEM is a record_item of FILE  (8)
       true: FILE is a declared file  (9)
   Upward propagation
   Removing false subtrees
   Evaluation based on matching
```

Figure 6.2. Trace of Building the Q-model: An Example

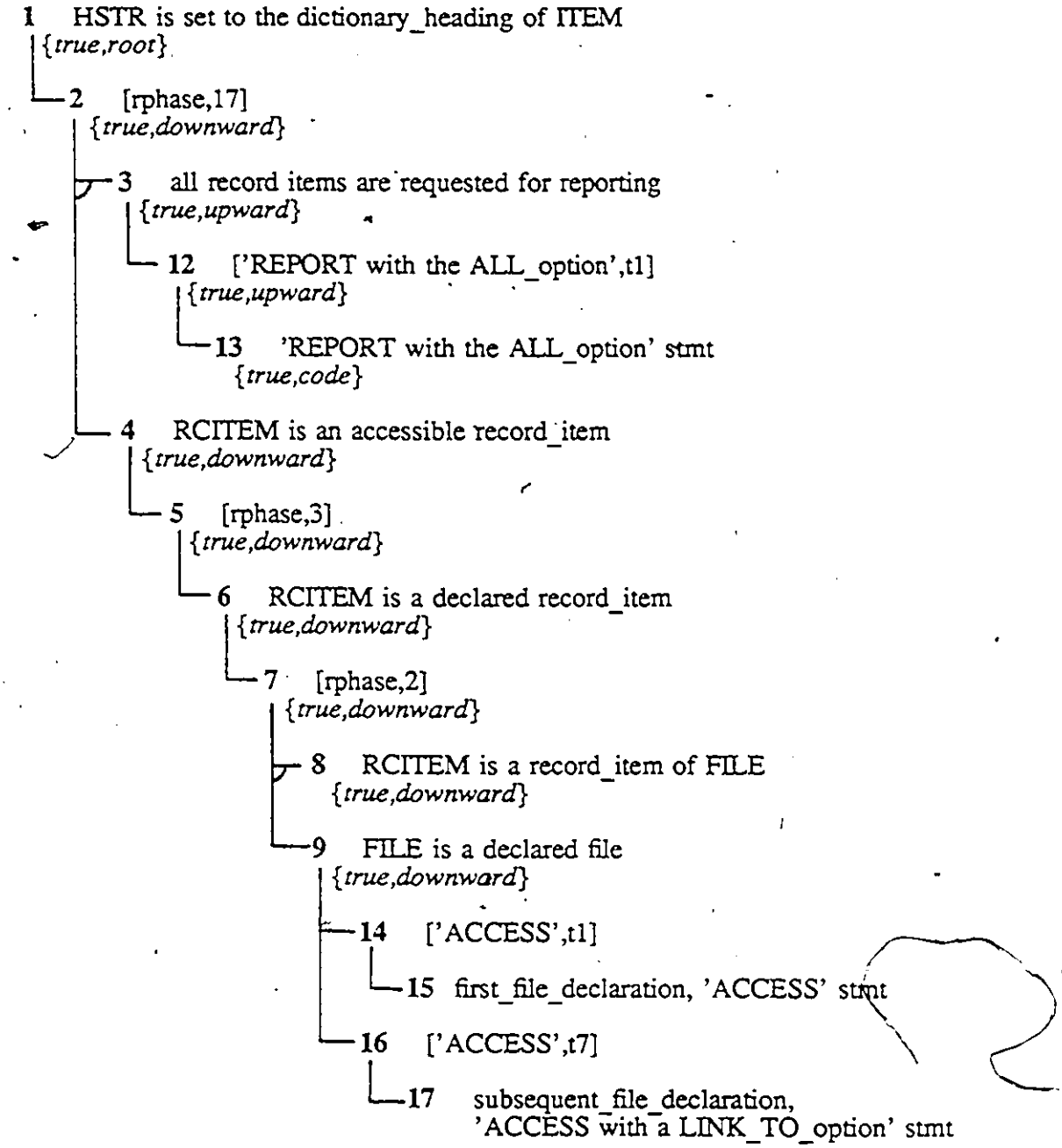


Figure 6.3. The D-model of a Question: An Example

dies an adequate explanation of why the question statement is true in the context of the query. Therefore, it is offered to the user as the answer. The tree is displayed in the same format as the answers to 'WHEN' queries (Section 5.4.2.2.3.). The actual answer to the above example is shown in Figure 6.4.

The nodes for which there is not enough information to establish the truth value (i.e., the ones that remained unlabeled) are indicated by '(?)'. They could be omitted in the answer, but their presence gives the user additional useful information: in our example they point to some relevant elements of QUIZ code that were not supplied with the query.

```
CODE:
                REPORT ALL

> Why is the column heading set to the dictionary heading?

  the column_heading of a report_item is set to the
  dictionary_heading of the item,
    CONSEQUENCE OF:
       all record_items are requested for reporting,
         CONSEQUENCE OF:
            using 'REPORT with the ALL_option' statement
       AND
       a record_item is an accessible record_item,
         CONSEQUENCE OF:
            the record_item is a declared record_item,
              CONSEQUENCE OF:
                 the record_item is a record_item of
                 a file
                 AND
                 the file is a declared file,
                   CONSEQUENCE OF:
                      using first_file_declaration
                      in 'ACCESS' statement (?)
                    OR
                      using subsequent_file_declaration
                      in LINK_TO_part
                         in 'ACCESS with a
                             LINK_TO_option' statement (?)
```

Figure 6.4. Regular Answer to a 'WHY' Query: An Example

6.1.2.2. Other Types of Answers

In this section we will describe how the system responds to a query that contains erroneous code or contradictory information (other than direct contradictions dealt with during the creation of initial D-models). There are three cases to consider.

a) **Errors Found in Code.** For each error found in the code, the appropriate error message is displayed, the culprit QUIZ statement indicated, and an explanation of the error message given to the user. Figure 6.5 shows an instance of this case.

```
CODE:
                ACCESS EMPLOYEES
                SELECT EMPLOYES
                REPORT

> Why is nothing reported?

You should have been issued error message:
     "Expected : file IF eol"
because the file was not declared
(Statement: SELECT EMPLOYES).
```

Figure 6.5. Error Found in Code: An Example

---

b) **An A-contradiction Found.** An *A-contradiction* is a contradiction between two assumptions, or an assumption and the QUIZ code. In such a case the statement that is contradicted is shown and the sources of contradiction are identified. Figure 6.6 shows the response regarding a query that contains both the QUIZ statement 'ACCESS EMPLOYEES', and 'an alias is specified' as one of the assumptions.

```
    ASSUMPTIONS:
                  an alias is specified

    CODE:
                  ACCESS EMPLOYEES

> (any question)

  Your query is INCONSISTENT!

     Contradiction found regarding:
         using ALIAS_part

     The contradiction results from
         the alias of the dictionary_file of a file is
         specified in the code  (assumption)
       and
         ACCESS EMPLOYEES  (code)

  Please examine your query...
```

Figure 6.6. A-contradiction Found: An Example

c) **A Q-contradiction Found.** A *Q-contradiction* is a contradiction that involves the question part of a query. Like in the A-contradiction situation, the contradictory statement is identified and the source of contradiction is explained. In this case, however, since the question is the central component of the query, and the background of the query (assumptions and QUIZ code) is consistent, the system tries to give a much more informative answer than just to draw the user's attention to the existing contradiction. Since the original question causes a contradiction, the negated form of the question must be consistent with the rest of the query! Therefore, the system generates an answer to the corresponding 'WHY NOT' query, but informs the user that a reference is made to the negated question rather than the original one. An example of this is given in Figure 6.7.[+] The message that describes the detected contradiction has the same format as in the A-contradiction case, but the question is indicated as one of the culprits. The format for presenting the violated condition is the same as the regular answer except that the question statement is not repeated. Statements found to be false are labeled by '(f)'.

[+] Note the brackets surrounding the noun phase of the question. These are due to the lexical convention of SEQAP by which a prepositional phrase modifies the main verb, unless its attachement to a noun phrase is made explicit by means of square brackets.

```
> s
    No assumptions
    QUIZ CODE:
                    ACCESS EMPLOYEES
                    REPORT NAME HEADING "Name"

> Why is [a default for the column heading] used?

    Your query is INCONSISTENT!

       Contradiction found regarding:
           using format_part without HEADING_part

       The contradiction results from
           a default for the column_heading of a report_item
           is used  (question)
         and
           REPORT NAME HEADING "Name"  (code)

    Either your QUIZ statements are incorrect
    or you did not formulate your question properly!

    Assuming the latter...

    The NEGATION of your question is TRUE
    because the following DOES NOT HOLD:
          using format_part without HEADING_part
          in report_item
             in report_items
                in report_group
                   in 'REPORT with a report_group' statement (f)
```

Figure 6.7. Answer to a 'WHY' Query with a Q-contradiction: An Example

6.2. Answering 'WHY NOT' Queries

A 'WHY NOT' query consists of a negative question and optional background information supplied as a set of assumptions and/or QUIZ code. Unlike 'WHY' queries, this category contains some subtypes that are exceptions to the general query-answering algorithm. An alternative approach to answering 'WHY NOT' queries will be also discussed.

6.2.1. Normal 'WHY NOT' Queries

The general algorithm for answering the 'WHY NOT' type of query is very similar to that for answering 'WHY' queries. In this section we will describe how it differs from the latter, rather than giving the entire algorithm and repeating most of what has been described before.

An answer to a 'WHY NOT' query is produced in two phases. The first phase, the process of building a model of the query (the Q-model), is identical to that described above, except that the negative sign of the question is taken into consideration when its initial D-model is built.

The delivery of an answer, when errors are found in the code or an A-contradiction is detected, follows the same approach as the one adopted for "WHY' queries. The remaining two types of outcome of Phase 1, however, reflect the duality of 'WHY' and 'WHY NOT' queries.

6.2.1.1. Regular Answer

If, during Phase 1, the relevant knowledge has been retrieved from the knowledge base and the reasoning process has not revealed any abnormality in the query (i.e., no errors or contradictions), the D-model of the question is an AND/OR tree with the root labeled 'false' and each of the remaining nodes either labeled 'false' or left unlabeled. Before the answer is generated, the tree is modified through the following two steps:

1) **Removing True Subtrees.** The tree is pruned above the 'true' nodes that have either a 'false' or unlabeled parent-node. The rationale for this step is the same as that given for the removal of false subtrees in the pruning-evaluation-propagation loop: they give an unnecessary level of detail.

2) **Downward Propagation.** Whether this step is necessary or not depends on the implementation of the propagation mechanism: removal of subtrees, followed by propagation based on simple rules, may be more efficient than a more complex value propagation mechanism alone.

Figure 6.8 shows an example of an actual regular answer to a 'WHY NOT' query. The violated condition is presented in the same form as the regular answer to a 'WHY' query except that the question statement is not repeated. Statements found to be false are indicated by '(f)'.

Another way of producing an answer will be examined in the last part of this section.

```
   ASSUMPTIONS:
                  'X' is an item
                  the type of 'X' is string
                  'F' is the field of 'X'

> Why is 'F' not truncated?


  Because the following DOES NOT HOLD:
        the status of the report_item is OVERFLOW (f),
          CONSEQUENCE OF:
             the length of the value of the item of the
             report_item is greater than the length of the
             picture of the report_item (f)
```

Figure 6.8. Regular Answer to a 'WHY NOT' Query: An Example

6.2.1.2. A Q-contradiction Situation

If the analysis of a 'WHY NOT' query reveals that the question is inconsistent with the rest of the query (i.e., a Q-contradiction is detected), the user is made aware of the contradiction, and the answer to the corresponding 'WHY' query is produced. Figure 6.9 shows an example of this case. This type of answer should explain why the opposite of the user's question is true provided the background information is correctly specified. This process is the mirror image of the one applied when a Q-contradiction is detected in answering a 'WHY' query.

```
  CODE:
                    ACCESS EMPLOYEES
                    REPORT NAME HEADING "Employee"

> Why was the column heading not established ?


  Your query is INCONSISTENT!

     Contradiction found regarding:
         using HEADING_part

     The contradiction results from
         not true that the column_heading of a
         report_item is established  (question)
       and
         REPORT NAME HEADING "Employee"  (code)


  Either your QUIZ statements are incorrect
  or you did not formulate your question properly!

  Assuming the latter...

   the column_heading of a report_item is established,
     CONSEQUENCE OF:
        the column_heading of the report_item is
        specified in the code,
          CONSEQUENCE OF:
             using HEADING_part
             in report_item
                in report_items
                   in report_group
                      in 'REPORT with a report_group'
                         statement
```

Figure 6.9. Answer to a 'WHY NOT' Query with a Q-contradiction: An Example

6.2.2. Special Cases

There are two types of 'WHY NOT' queries that are treated in special ways:

a) *"Why does the code not work?"* or equivalent. This is a very vague question but the analyzed set of real questions and answers suggests that the user most likely refers to a QUIZ program that does not produce a report due to errors in code. Therefore, the system tries to find an answer to the query by treating this as a request for an explanation of error messages (without any message text being supplied within the query). The applied algorithm is described in Section 6.3. (answering 'ERR' queries).

b) *Quantity-related questions.* As described in Section 4.1 4.3., there are two types of LESK statements: regular and quantity-related. A quantity-related statement indicates that the number of occurrences of a certain kind of QUIZ entities is too large or too small. If the question is a quantity-related statement, the 'WHY NOT' query is transformed into a 'WHY' query by replacing the original adjective by its complement ('large'/'small'). For example, the following question within a 'WHY NOT' query:

"Why did not I get a large number of fields?"

is replaced by

"Why did I get a small number of fields"

which, added to the original background information, forms a 'WHY' query. This is done in order to make the question more natural and, consequently, the answer more understandable.

6.2.3. Alternative Approach

A display of the violated condition is not the only type of an answer to a 'WHY NOT' query that our formalism could offer. An alternative answer could be based on a different method of building a model of a negative statement.

In our approach, the initial D-models of a statement and its negation are identical, except that the roots of the trees are labeled by different truth values. Alternatively, a model of a negative statement can be built by creating a deduction tree for the original statement (rather than its positive version) as the root. The resulting model would correspond to the transformation of the deduction tree for the positive statement that is obtained by propagating the negative sign from the root downward, in accordance with the DeMorgan's laws.

This alternative approach allows 'WHY' and 'WHY NOT' queries to share the same answer format. It could also be regarded as a more appropriate model from a logicians' point of view. In this context, however, it has some disadvantages, the most serious being that, any nontrivial answer may be quite difficult to read and comprehend since it may contain a large number of negative statements. A simple experiment with both methods applied to a sample of queries suggests that our approach is more effective. A more serious comparison of the approaches would require feedback from real users.

6.3. Answering 'ERR' Queries: Explaining Errors

Messages that software systems issue when they detect errors in programs are often very cryptic and of little or no help to the user. A typical question a programmer asks in such a situation is:

"Why did I get the error message <message text>?"

or

"Why did I get an error message?"

What the programmer needs, at least when seeing the message for the first time, is a clear explanation that will help him/her realize in what way the rules of the language have been violated.

A fully developed advisor, if coupled with QUIZ, would have direct access both to the QUIZ code and to the status information such as issued messages. This would make answering 'ERR' queries very simple. In this section, however, we discuss an approach that applies to a stand-alone advisor.

Even though questions that follow the above syntactic patterns could be looked at as components of 'WHY' queries, there are reasons to treat this category separately:

a) The type of the expected answer is very different than that for general 'WHY' queries.

b) For each error message, a general natural language explanation can be formulated, stored in the knowledge base, and easily retrieved when necessary. In

such a case, generating an answer to a query that specifies the actual message text does not require any deduction.

c) Any uncertainty in answering an 'ERR' query that does not contain a valid error message can always be fully resolved through a very brief and simple dialogue with the user, which is not the case with general 'WHY' queries.

Since our formalism has been developed with the above considerations in mind, the answering process for an 'ERR' query can be as simple as a direct retrieval of the appropriate explanation. This applies to questions that fit the first pattern shown on the previous page, provided that the error message has been correctly reproduced. Otherwise, a diagnostic process must take place. In any case, however, an 'ERR' query, like both previously discussed types of queries, may include background information in the form of assumptions and/or QUIZ code.

A guideline followed in the design of many diagnostic expert systems is that the system should use the 'least expensive' route to reach the diagnosis. For example, the system should not ask the user to provide additional evidence unless it is absolutely necessary, or the lack of that piece of information would require excessive computation. Therefore, the answering algorithm for 'ERR' queries will consist of a sequence of attempts to find the right answer, giving priority to direct information retrieval over the (more costly) deduction. Each subsequent attempt builds upon the information inherited from the previous attempts, and either finds the answer or makes additional progress towards one. Conditions that make each of the attempts (i.e., major steps of the answering algorithm) successful are the

following:

(1) A valid message is specified

(2) A close approximation of the message is specified

(3) Errors are found in the supplied code

(4) A discovered potential error is confirmed

(5) An error is located in the complete code

As mentioned above, a fully developed Advisor would have a direct access both to the QUIZ code and error messages. In such a case, there is no need to go through the algorithm.

In the remainder of this section, the algorithm will be described in more detail and illustrated by examples.

6.3.1. Answer Based on the Message Text

Normally, when the text of an error message is given as part of a query, finding the answer to the query is a trivial task. Even if the user fails to fully reproduce the message text, it is sometimes possible to make up for his/her mistake by looking for a similar message in the knowledge base. Otherwise, the message is useless and the answering proceeds as if no message had been specified.

6.3.1.1. Valid Message

When a question is asked which contains the full text of an error message that

makes no reference to a specific component of the code, the system attempts to find the identical message in the knowledge base. However, if the message includes explicit reference to the code, the system tries to match it with the corresponding message pattern from the knowledge base. If this step is successful, the explanation associated with the message is retrieved and shown to the user as the answer. An example is presented in Figure 6.10.

```
> Why did I get the error message
  "Cannot find the item NAME OF BRANCHES" ?

  The message "Cannot find the item NAME OF BRANCHES"
  is issued because the qualified item was not found
  in the dictionary.
```

Figure 6.10. 'ERR' Query with a Valid Message: An Example

6.3.1.2. Close Approximation of the Message

If there is no error message in the knowledge base that perfectly matches the given one, it is likely that the user misspelled a word or did not remember the full text of the message. The system can still benefit from such evidence. It can retrieve those messages that are similar to the specified one according to some established criteria. The most likely candidates are then presented to the user who is asked to confirm or deny one message at a time. The process terminates when

either one of them is confirmed or no more candidates are left to be offered. If a message is confirmed by the user, then its explanation is given as the answer, otherwise the message supplied with the query is ignored. Figure 6.11 shows an example of a successful application of this step.. The similarity criteria used in the prototype is a match on at least two words (a colon is treated as a separate word).

```
> Why did I get the error message "Expected : file" ?

  "Expected : file" is not a valid error message.

    Was the message "Expected : RECORD key file"?
                                              (y/n) >> n
      Was the message "Expected : file IF eol"?
                                              (y/n) >> y

  The message "Expected : file IF eol" is
  issued because the file was not declared.
```

Figure 6.11. 'ERR' Query with an Approximate Message: An Example

6.3.2. Answering Based on Deduction

If an 'ERR' query does not specify the text of the error message, or the message is specified but the above attempts failed to produce an answer, the system tries to discover errors in the actual QUIZ code that was used when the message was issued.

Assumptions given in the query, if any, are partially explored: the initial D-model of each assumption is created and the pre-parse evaluation is performed as described within 'WHY' and 'WHY NOT' algorithms. If the code is not supplied with the query, the user is asked to enter it at this pont. The code is then parsed and analyzed.

6.3.2.1. Errors in Partial Code

One of the tasks in the code analysis is checking whether a particular condition is met. Some conditions can be checked through the results of the analysis performed thus far, others use models of assumptions as an additional source of information. An error is discovered whenever an examined condition is found to be violated. Each violated condition causes the corresponding error PT-rule to fire. The effect side of such a rule specifies the applicable error message.

If the analysis of the QUIZ code discovers one or more errors, the system shows, for each error, the applicable message and its explanation, and specifies the statement to which this applies (unless the code contains a single statement, in which case there is no ambiguity). Figure 6.12 presents an example of an error discovered based on information contained in the assumptions.

```
ASSUMPTIONS:
                'DATEINPUT' is an item
                the type of 'DATEINPUT' is string
                'DATEENTER' is an item
                the type of 'DATEENTER' is numeric

CODE:           SELECT IF DATEINPUT EQ DATEENTER

> Why did I get an error message ?

  You should have been issued error message:
       "The data type is not the expected one"
  because the data types in the condition do not agree.
```

Figure 6.12. Errors Found in Code: An Example

The analysis may fail to find any errors because of one of the following:

a) At least one condition that was not met at the time the code was originally used (thus, resulting in an error), could not be determined as failed because there was not enough relevant information available during the analysis (e.g., conditions related to the data dictionary information);

b) The code that has been examined is error free. It is, however, a fragment of a program which does contain errors but the user did not supply the erroneous part;

c) The program is error free.

The remaining three steps cover the above cases, respectively.

6.3.2.2. Potential Errors

The analysis of the code has not discovered any errors but there are conditions that could not be verified. The system makes the user aware of the potential errors and explains each of them. Even though the actual error is not determined by the system, the query is successfully answered if the user recognizes the error among the displayed candidates, A confirmation from the user that he/she is satisfied with the answer will terminate the answering process. An example of such a case is presented in Figure 6.13.

6.3.2.3. Errors in Complete Code

As mentioned above, the code that is supplied as part of the query is not necessarily the entire sequence of QUIZ statements originally used when the error message was issued. If this is the case, the user is asked to provide the entire code which is then examined in the same way. Figure 6.14 shows the same example as the one presented in Figure 6.13, except that the user did not admit making any of the suggested errors.

6.3.3. "No News is News"

If all above described attempts to answer a query fail, provided that the user

```
CODE:
                REPORT NAME OF EMPLOYEES

> Why did I get an error message ?

  Conditions that are potentially violated:

     EMPLOYEES is described in the dictionary as a file
          (causing error message:
           "Expected : RECORD key file"
           at ACCESS statement).

     NAME is described in the dictionary as an item
          (causing error message:
           "Expected : item"
           at REPORT statement).

     NAME is described in the dictionary as
                                   an item of EMPLOYEES
          (causing error message:
           "Cannot find the item NAME OF EMPLOYEES"
           at REPORT statement).

  Does this explain your problem? (y/n) >> y
```

Figure 6.13. Potential Error Confirmed: An Example

has been giving correct answers during the diagnostic process, it can be concluded that the program has none of the errors that are known to the system. Assuming that the knowledge base contains all error messages for the particular release of the software product in question (MiniQUIZ in the case of the QAUZ prototype), the user is informed that the issued error message must have originated from some other source (e.g., the operating system).

```
                <potential errors>
                        :
                        :

Does this explain your problem? (y/n)>> n

The following code has been considered:
      REPORT NAME OF EMPLOYEES
Please provide your entire sequence of statements
Enter your QUIZ code:
              >> ACCESS EMPLOYEES LINK TO BILINGS ALIAS BILLS
              >> SELECT IF AMT OF BILLINGS GT 1000
              >> REPORT NAME OF EMPLOYEES
              >>

You should have been issued error message:
     "Expected : file LT LE EQ GE GT NE"
because the item qualifier BILLINGS is not a declared file
(Statement: SELECT IF AMT OF BILLINGS GT 1000).
```

Figure 6.14. Error Detected in Complete Code: An Example

7. SUMMARY AND CONCLUSIONS

In this chapter we will summarize our research and discuss the advantages and disadvantages of our approach for the development of software advisors. This will be followed by some suggestions for further work.

7.1. Summary and Evaluation

The research effort described in this thesis has mainly focused on two important knowledge engineering tasks: defining a formalism suitable for encoding causal knowledge, and developing strategies for using the knowledge in answering various types of queries.

7.1.1. Knowledge Encoding Formalism

We have developed a formalism that is based on two widely adopted knowledge representation paradigms: rules and frames. We use several types of rules to represent different causal links between components of a program and features of the result. The syntax of a QUIZ statement is described in the Augmented Syntax Language (ASL) introduced in this thesis. Descriptive statements are encoded in the English-like language LESK. Knowledge pertaining to an individual statement type is organized around the description of its syntax and is contained in simple frame structures. Frames form a hierarchy that reflects the hierarchy of QUIZ statement types.

Our formalism has a number of advantages. It forces the knowledge engineer

to encode knowledge in a formal and disciplined fashion, yet one that is easy for humans to use and understand. Many systems require knowledge to be encoded in a form very similar to its internal representation (e.g., LISP-like structures in UC). Even if the knowledge engineer does not find it difficult to use, it is bound to be an obstacle in the process of knowledge verification by domain experts. We have asked our QUIZ experts to review and verify the contents of the knowledge base (the material presented in Appendix A) without giving them any instruction or description of the formalism. They did not experience any problem whatsoever in performing this task. Being quite self-explanatory, the represented knowledge does not have to be extensively documented. In fact, it could serve as a basis for the documentation of the domain software: technical writers can benefit from it, and even parts of the natural language documentation could be generated automatically from the representation. It can also be used as a software engineering tool, since it integrates knowledge about the syntax and semantics of the domain language, and provides means for specifying the internal structure and operation of the system.

Another benefit from using our formalism is that its statement-oriented approach encourages systematic knowledge acquisition. Starting with the representation of a statement syntax, the knowledge engineer proceeds by concentrating on fragments of the statement and attaching the appropriate causal knowledge to them. This way the degree to which the task has been completed can easily be assessed at any time, and the knowledge engineer thus reminded of the areas that remain to be covered.

7.1.1.1. Generality

For the initial prototype, we have limited the domain to a subset of QUIZ and successfully encoded knowledge about it. Although we experimented within a relatively small domain, our objective was to develop a general formalism, applicable to the entire QUIZ language, as well as to other similar languages. We believe that this objective has been met.

The formalism builds on general, widely used and domain independent paradigms (rules, frames), and a simple English-like knowledge representation language (LESK). It also provides features for capturing types of knowledge necessary for reasoning about computer programs (ASL constructs, specialized rules). Based on knowledge of and experience with QUIZ, we expect little difficulty in extending the knowledge base to include the remaining causal knowledge about QUIZ. This is because we restricted the domain in such a way that, although its size is relatively small, it still is typical of the complexity and major characteristics of QUIZ.

A possible question to ask is how difficult it would be to apply our formalism to other software systems. There are a large number of fourth generation languages which are in many ways similar to QUIZ (e.g., SQL (see, for example, [Date 82]), NATURAL [NRM 82]). Rather than discussing individual cases, we will identify those common characteristics of these languages that are critical for the applicability of the formalism.

We made use of ASL to represent the syntax of the language, fragments of which constitute the antecedent side of one type of causal rules (PT-rules). This is

appropriate for systems that accept typed commands as the input. However, the use of syntax in our representation would be a disadvantage in encoding knowledge about systems that are not statement-oriented (e.g., modern spreadsheet packages). Users of such systems normally specify commands through the cursor movement keys and special devices (e.g, a 'mouse') which would require the development of a different metalanguage as the replacement for ASL.

In declarative languages like QUIZ, the role of each statement is typically independent of that of other statements. This makes causal relationships easier to discover and represent. We have taken advantage of these characteristics of the domain which greatly contributed to the simplicity and clarity of the knowledge representation (e.g., the convenient coupling of regular and default PT-rules). This, however, puts limitations on the applicability of the formalism, in its present form, to significantly different types of programming languages. For example, additional features are needed to model the more complex causal relationships that can be found in procedural languages.

Another characteristic of our formalism that reflects the nonprocedural nature of QUIZ is its suitability for representing different phases of the program execution. In the case of a report writer like QUIZ, our formalism allows the knowledge engineer to model parsing and reporting phases independently, with an internal model of the report as the "common denominator" of the two. This feature also facilitates the generation of better explanations. It does not, however, limit the applicability of the representation to the multi-phase systems.

In summary, we expect our knowledge representation formalism to be applicable (in its present form or with minor modifications) to nonprocedural languages which are statement-oriented and in which the roles of different types of statements are largely independent of each other. It is particularly suitable (but not limited to) those systems that execute programs in multiple phases.

7.1.2. Query-Answering

A causal model of QUIZ that was built using the knowledge representation we have introduced is the basis of the QAUZ subsystem of the QUIZ Advisor. The primary role of QAUZ is to provide explanations of unexpected or puzzling features of QUIZ reports. The prototype of QAUZ has been implemented in Quintus Prolog (in approximately 150K of code) on a Sun-3 workstation, and was successfully demonstrated to the sponsors of the Advisor Project in November 1986.

A typical query contains a question and background information consisting of a fragment of QUIZ code and a number of LESK statements that describe various assumptions. In order to fully use the information contained in a query, QAUZ examines individual components of the query and tries to establish causal links among them. It parses QUIZ statements, and uses forward chaining of rules and property inheritance of frames to determine the effects the code has on the result. On the other hand, when analyzing descriptions of the report (specified through assumptions), the system performs backward deduction to determine features of the code that caused the result. While doing this, it constructs an internal representa-

tion of the problem state which explicitly shows the various causal relationships among the components of the query. The type of the query and the internal representation of the problem determine the format and content of the answer. QAUZ applies a number of answering strategies (all of which make use of the described basic reasoning mechanisms) to answer various types of queries (e.g., why, why not, error, hypothetical).

A strong point of our query-answerer is the ability to reason with incomplete information. Some expert systems look for a very precise answer to the submitted question. This often requires the user to supply a great deal of additional evidence, without which the system cannot answer the question. Obviously, this is unavoidable in some domains of consultation (e.g., medical diagnosis). Our approach is different: the user supplies the query that describes the problem and gets the most specific answer that the available information and the contents of the knowledge base allow for, without being burdened with requests for more evidence (the only exception is in answering an error query that contains neither QUIZ code nor the text of an error message). We believe that such an answer, although less focused than it would be if more information were made available through a dialogue with the user, is normally more educational: it applies to a whole class of similar queries, while clearly indicating the information missing in the particular case. Our domain experts who have seen examples of queries and corresponding answers agree with us.

Another strength of our query-answerer is its ability to discover and deal effectively with contradictory information contained in a query. This is an impor-

tant feature, especially in a consultation environment in which contradictions can be well disguised due to a variety of forms the input can be given in (e.g., QUIZ code and LESK statements). Our system can find contradictions, and, under certain circumstances, can go much further. This is particularly evident in answering 'why' and 'why not' queries: in addition to pointing at the source of contradiction (which happens whenever a contradiction is discovered), the system modifies the query in order to eliminate the contradiction, and produces an informative answer by reasoning about the modified problem.

Although in the development of the initial prototype we have not given a high priority to the issues of efficiency and the ability to answer a large number of real questions, we are satisfied with the system's performance with respect to both. It takes less than 1.5 seconds to answer a query of an average complexity, and no more than 5 seconds in the most complex cases (e.g., a query that contains previously unexamined QUIZ code and a number of assumptions, and even conveys contradictory information). From the analyzed set of real questions ('why' and error), we have selected those which fall into the domain we have been dealing with. Queries constructed from these have all been successfully answered by the prototype. These are shown as the last six examples in Appendix B.

The main purpose of the initial prototype was to demonstrate the feasibility of our query-answering strategies. Although this objective was fully met, we are aware of many shortcomings of the system. First, its interface has a somewhat simplistic design. It does not take advantage of windows, interactive graphics, and other modern techniques (these were not available to us during the implementation

of the prototype). It would be much more convenient for the user to have a set of windows, each dedicated to an individual component of a query, and so see the entire query and the answer on the screen at the same time.

QAUZ typically accepts one or more QUIZ statements as part of a query. The text of an error message is also acceptable within an error question. In the implementation of the prototype no provision was made to allow direct access to machine readable QUIZ code or issued error messages, but this should certainly be done for a more ambitious implementation. The benefit from such a feature would be twofold: the user would need to provide less input, and the information obtained directly from QUIZ would be both correct and complete.

Another area of improvement concerns retrieval of relevant knowledge. At present, this is done through an index based on parameters of the internal model, combined with matching of internal forms of LESK statements. Information stored in the knowledge base, however, allows for a more sophisticated retrieval strategy (e.g., the purpose slot can be used for guiding the process).

7.2. Directions for Further Research

We have identified some possible lines of research that would result in various improvements and extensions to the work presented in this thesis: extending the domain to full QUIZ, experimenting with other fourth generation languages, enhancing the knowledge encoding formalism to cover procedural languages, and developing more sophisticated user interface and knowledge retrieval methods.

An outstanding issue to be dealt with is the integration of the QAUZ and HDI subsystems into the QUIZ Advisor. The two initial prototypes have been developed independently, on different machines, and with different tools. This has given us a chance to experiment in various ways and learn valuable lessons in both tracks of the project. Perhaps the best way to continue would be to build on these experiences and develop a new prototype in ART, currently one of the most powerful expert systems development tools [ART 85], which has been recently made available to us on a SUN-3 workstation. Since the first QAUZ prototype was implemented in Prolog, which is a lower level tool, we believe it would be appropriate to include some general ideas on how we could approach the task. Therefore, we will end this discussion by briefly describing ART and offering a sketch of a possible implementation of QAUZ in ART.

ART (Automated Reasoning Tool, developed by Inference Corporation) provides the knowledge engineer with a sophisticated environment and a variety of powerful features including the following major ones:

* Rule-based programming, featuring the integration of five type of rules (forward chaining, backward chaining, hypothetical, constraint, and belief) with a powerful pattern-matching language,

* Schema-based knowledge representation (compiled into logic-like assertions) which allows the program to reason about objects and their mutual relations,

* The viewpoint mechanism, which enables modeling hypothetical worlds

by defining contexts in which facts and rules apply, and situations that change dynamically with time,

* An interactive development environment (ART Studio Interface) that comprises a variety of monitoring and debugging aids for system development, and

* A graphics interface package (ARTIST) that helps the user create a custom interface using both animation graphics and menus which can be driven directly by rules.

The analysis of actual QUIZ code can be done primarily through forward chaining rules (these are typical condition-action pairs) in a fashion similar to that used in modeling time-related processes. In this case, however, viewpoints would be created for elements of the processed input rather than for points in time. This should be combined with hypothetical reasoning (achieved by using hypothetical rules to generate alternative viewpoints, and belief rules to subsequently select the 'believed' hypothesis) in order to resolve those few nondeterministic cases the parser might have to deal with. The truth maintenance problem that arises from the cancellation of QUIZ statements can be solved using constraint rules that 'poison' unwanted branches of the viewpoint structure. The final result of this process would be a chain of viewpoints, in which each node contains conclusions made in the corresponding step of the analysis, while inheriting those made in the preceding steps.

A correct ART implementation of backward reasoning (i.e., reasoning from

results toward their causes) seems to be very close to that already proposed for forward reasoning. This is because backward chaining is achieved through the same basic ART mechanism used for forward chaining, but performed with special (backward-chaining) rules. Backward chaining in ART simply serves to achieve certain kind of execution control. In ART, one cannot use the same rule in both directions as we have done in the Prolog implementation. This means that, in many cases, we would have to represent the same piece of knowledge in two symmetric forms: cause-effect and effect-cause rules. This would be a price to pay for many conveniences which ART otherwise offers to the knowledge engineer.

* * *

In summary, we have described research aimed at answering various questions about the role of, representation of, and reasoning with causal knowledge in software advisors. This research has not given all the answers, but it has demonstrated a promising new method of augmenting such systems with a powerful explanatory component based on a causal model of the domain. We believe that it is a step made in the right direction.

REFERENCES

[Allen 83]
Allen, J. F., "Maintaining Knowledge about Temporal Intervals", *Communications of the ACM*, 26 (1983):832-843.

[Allen 84]
Allen, J. F., "Towards a General Theory of Action and Time", *Artificial Intelligencé* 23 (1984):123-154.

[ART 85]
ART Programming Tutorial (Version 2.0), Inference Corporation, Los Angeles, CA, 1985.

[Bennet, Hollander 81]
Bennet, J.S., and Hollander, C. R., "DART: An Expert System for Computer Fault Diagnosis", *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, vol. 2, 1981, pp.843-845.

[Billmers, Carifio 85]
Billmers, M. A., and Carifio, M. G., "Building Knowledge-Based Operating System Consultants", *Proceedings of the Second IEEE Conference on Artificial Intelligence Applications*, Miami Beach, Florida, 1985, pp.449-454.

[Blum 82]
Blum, R. L., "Induction of Causal Relationships from a Time-Oriented Clinical Database: An Overview of the RX Project", *Proceedings of the National Conference on Artificial Intelligence*, Pittsburgh, PA, 1982, pp.355-357.

[Brown 84]
Brown, J. S., "The Low Road, the Middle Road, and the High Road", in [Winston, Prendergast 84], pp.81-90.

[Buchanan, Shortliffe 84]
Buchanan, B. G. and Shortliffe, E. H., eds., *Rule-Based Expert Systems, The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley Publishing Company, 1984.

[Chandrasekaran, Mittal 82]
Chandrasekeran, and Mittal, "Deep versus Compiled Knowledge Approaches to Diagnostic Problem-Solving", *Proceedings of the National Conference on Artificial Intelligence*, Pittsburgh, PA, 1982, pp.349-354.

[Charniak, McDermott 85]
Charniak, E., and McDermott, D.; *Introduction to Artificial Intelligence*, Addison-Wesley Publishing Company, Reading, MA, 1985.

[Chin 83a]
Chin, D. N., "A Case Study of Knowledge Representation in UC", *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, 1983.

[Chin 83b]
Chin, D. N., "Knowledge Structures in UC, The UNIX Consultant", *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics*, MIT, 1983, pp.159-163.

[Clancey, Letsinger 81]
Clancey, W. J., and Letsinger, R., "NEOMYCIN: Reconfiguring a Rule-Based Expert System for Application to Teaching", *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, vol. 2, 1981, pp.829-836.

[Clancey, Shortliffe 84]
Clancey, W. J. and Shortliffe, E. H., eds., *Readings in Medical Artificial Intelligence, The First Decade*, Addison-Wesley Publishing Company, 1984.

[Constant et al. 87]
Constant, P., Matwin, S., and Szpakowicz, S., "Question-Driven Approach to the Construction of Knowledge-Based Software Advisor Systems", *Proceedings of the Third Conference on Artificial Intelligence Applications*, Orlando, Florida, February 1987, pp.29-37.

[Date 82]
Date, C. J., *An Introduction to Database systems*, 3rd ed., Addison-Wesley Publishing Company, Reading, MA, 1982.

[Davis 76]
Davis, R., "Applications of Meta-Level Knowledge in the Construction, Maintenance, and Use of Large Knowledge Bases", in [Davis, Lenat 80].

[Davis, Lenat 80]
Davis, R., and Lenat, D., *Knowledge-Based Systems in Artificial Intelligence*, McGraw-Hill, New York, 1980.

[Davis 83a]
Davis, R., "Diagnosis via Causal Reasoning: Paths of Interaction and the Locality Principle", *Proceedings of the National Conference on Artificial Intelligence*, Washington, D.C., 1983, pp. 88-94.

[Davis 83b]
Davis, R., "Reasoning from First Principles in Electronic Troubleshooting", *International Journal of Man-Machine Studies*, 1983, pp.403-423.

[Deering et al. 81]
Deering, M., Faletti, J. and Wilensky, R., "PEARL: An Efficient Language for Artificial Intelligence Programming", *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, 1981.

[Delisle 87]
Delisle, S., "A Natural Language Interface for an Expert Advisor System", M.C.S. thesis, Department of Computer Science, University of Ottawa, 1987.

[de Kleer 84]
de Kleer, J., "How Circuits Work", *Artificial Intelligence* 24 (1984):205-280.

[Douglass, Hegner 82]
Douglass, R., and Hegner, S., "An Expert Consultant for the UNIX System: Bridging the Gap Between the User and Command Language Semantics", *Proceedings of the Fourth National Conference of the Canadian Society for Computational Studies of Intelligence (CSCSI)*, Saskatoon, Saskatchewan, 1982, pp.92-96.

[Doyle 79]
Doyle, J., "A Truth Maintenance System", *Artificial Intelligence* 12, 1979, pp.231-272.

[Faletti 82]
Faletti, J., "PANDORA - A Program for Doing Commonsense Planning in Complex Situations", *Proceedings of the National Conference on Artificial Intelligence*, 1982, pp.185-188.

[Fikes, Kehler 85]
Fikes, R., and Kehler, T., "The Role of Frame-Based Representation in Reasoning, *Communications of the ACM* 28 (1985):904-920.

[Gorry et al. 78]
Gorry, G. A., Silverman, H., and Pauker, S. G., "Capturing Clinical Expertise: A Computer Program that Considers Clinical Responses to Digitalis", *The American Journal of Medicine* 64 (1978):452-460.

[Hayes-Roth et al. 83]
Hayes-Roth, F., Waterman, D., and Lenat, D., eds., *Building Expert Systems*, Addison-Wesley Publishing Company, Reading, MA, 1983.

[Houghton 84]
Houghton, R. C., "Online Help Systems: A Conspectus", *Communications of the ACM* 27 (February 1984):126-133.

[Johnson, Soloway 84]
Johnson, W. L., and Soloway, E.,,"Intention-Based Diagnosis of Programming Errors" *Proceedings of the National Conference on Artificial Intelligence*, 1984, pp.162-168.

[Johnson 85]
Johnson, W. L., "PROUST: A System Which Debugs Pascal Programs", *Proceedings of the Expert Systems in Government Symposium,* McLean, VA, October 1985, p.157.

[Kelly, Steinberg 82]
Kelly, V. E., and Steinberg, L. I., "The CRITTER System: Analyzing Digital Circuits by Propagating Behaviors and Specifications", *Proceedings of the National Conference on Artificial Intelligence*, Pittsburgh, PA, 1982, pp.284-289.

[Lewis et al. 78]
Lewis, P. M., Rosenkrantz, D. J., and Stearns, R. E., *Compiler Design Theory*, Addison-Wesley Publishing Company, 1978.

[Matwin et al. 86]
Matwin, S., Skuce, D., and Szpakowicz, S., "Question-Driven Approach to the Design of a Software Advisor System", TR-86-04, Department of Computer Science, University of Ottawa, April 1986.

[McCalla et al. 86]
McCalla, G. I., Bunt, R. B., and Harms, J. J., "Diagnosis of Non-syntactic Programming Errors in the SCENT Advisor", *Proceedings of the Fifth National Conference of the Canadian Society for Computational Studies of Intelligence (CSCSI)*, London, Ontario, 1984, pp.109-116.

[McDermott, Brooks 82]
McDermott, D., and Brooks, R., "ARBY: Diagnosis with Shallow Causal Methods", *Proceedings of the National Conference on Artificial Intelligence*, 1982, pp.370-372.

[Mitchell et al. 83]
Mitchell, T. M., Steinberg, L. I., Kedar-Cabelli S., Kelly, V. E., Shulman, J., and Weinrich, T., "An Intelligent Aid for Circuit Redesign", *Proceedings of the National Conference on Artificial Intelligence*, Washington, D.C., 1983, pp.274-278.

[Mitchell et al. 84]
Mitchell, T., Steinberg, L., and Shulman, J., "A Knowledge-Based Approach to Design", *Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems*, Denver, CO, December 1984.

[Myers 79]
Myers, G. J., *The Art of Software Testing*, Wiley, 1979.

[NRM 82]
NATURAL Reference Manual, Version 1.2. Software AG of North America, Inc., 1982.

[O'Shea, Eisenstadt 84]
O'Shea, T., and Eisenstadt, M., eds., *Artificial Intelligence: Tools, Techniques, and Applications*, Harper & Row publishers Inc., 1984.

[Patil 81]
Patil, R. S., "Causal Representation of Patient Illness for Electrolyte and Acid-Base Diagnosis", Ph.D. dissertation, MIT, Technical Report MIT/LCS/TR-267, 1981.

[Patil et al. 81]
Patil, R. S., Szolovits, P., and Schwartz, W. B., "Causal Understanding of Patient Illness in Medical Diagnosis", *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, vol. 2, 1981, pp.893-899.

[Patil et al. 82]
Patil, R. S., Szolovits, P., and Schwartz, W. B., "Information Acquisition in Diagnosis", *Proceedings of the National Conference on Artificial Intelligence*, Pittsburgh, PA, 1982, pp.345-348.

[Pressman 82]
Pressman, R. S., *Software Engineering: A Practitioner's Approach*, McGraw-Hill, New York, 1982.

[Pople 77]
Pople, H. E., "The Formation of Composite Hypotheses in Diagnostic Problem Solving: An Exercise in Synthetic Reasoning", *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977, pp.1030-1037.

[Puterman 77]
Puterman, Z. M., *The Concept of Causal Connection*, Philosophical Studies, University of Uppsala, Uppsala, Sweden, 1977.

[Quintus 86]
Quintus Prolog Reference Manual, Quintus Computer Systems Inc., 1986.

[QUIZ 85]
QUIZ version 5.01 User's Guide, Cognos Inc., 1985.

[Rich 83b]
Rich, E., *Artificial Intelligence*, McGrow-Hill, 1983.

[Rich, Shrobe 79]
Rich, C., and Shrobe, H. E., "Design of Programmer's Apprentice", in *Artificial Intelligence: An MIT Perspective*, edited by P. H. Winston and R. H. Brown, vol. 1, The MIT Press, Cambridge, MA, 1979, pp.137-175.

[Rieger 76]
Rieger, C., "An Organization of Knowledge for Problem Solving and Language Comprehension", *Artificial Intelligence* 7, 1976, pp.89-127.

[Rieger 77]
Rieger, C., "The Declarative Representation and Procedural Simulation of Causality in Physical Mechanisms", *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977, pp.250-256.

[Rolnick 73]
Rolnick, W. B., ed., "Causality and Physical Theories", *AIP Conference Proceedings*, Wayne State University, 1973.

[Rubinoff 85]
Rubinoff, R., "Explaining Concepts in Expert Systems: The CLEAR System", *Proceedings of the Second IEEE Conference on Artificial Intelligence Applications*, Miami Beach, Florida, 1985, pp.416-421.

[Schank 84]
Schank, R., "Intelligent Advisory Systems", in [Winston, Prendergast 84], pp.133-148.

[Schlegel 73]
Schlegel, R., "Historic Views of Causality", in [Rolnick 73], pp.3-22.

[Shibahara 85]
Shibahara, T., "On Using Causal Knowledge to Recognize Vital Signals: Knowledge-based Interpretation of Arrhythmias", *Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Los Angeles*, 1985, pp.307-314.

[Shibahara et al. 83]
Shibahara, T., Tsotsos, J. K., Mylopoulos, J., and Covvey, H. D., "CAA: A Knowledge-Based System Using Causal Knowledge to Diagnose Cardiac Rhythm Disorders", *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, vol. 1, 1983, pp.242-245.

[Shoham 85]
Shoham, Y., "Reasoning About Causation in Knowledge-Based Systems", *Proceedings of the Second IEEE Conference on Artificial Intelligence Applications*, Miami Beach, Florida, 1985, pp.629-634.

[Shortliffe 76]
Shortliffe, E. H., *Computer-Based Medical Consultations: MYCIN*, Elsevier, New York, 1976.

[Shrager, Finin 82]
Shrager, J., and Finin, T., "An Expert System that Volunteers Advice", *Proceedings of the National Conference on Artificial Intelligence*, 1982, pp.339-340.

[Sinnhuber, Hunter 84]
Sinnhuber, R. K., and Hunter, J., "Temporal Information for Medical Expert Systems: A Review of Recent Work", Report AIMG-5, Artificial Intelligence in Medicine Group, University of Sussex, 1984.

[Skuce 77]
Skuce, D., "Toward Communicating Qualitative Knowledge Between Scientists and Machines", Ph.D. dissertation, McGill University, Montreal, 1977.

[Skuce 83]
Skuce, D., "The Lesk Tutorial", TR-83-03, Department of Computer Science, University of Ottawa, July 1983.

[Skuce et al. 85a]
Skuce, D., Matwin, S., Tauzovich, B., Oppacher, F., and Szpakowicz, S., "A Logic-Based Knowledge Source System for Natural Language Documents", *Data and Knowledge Engineering* 1 (1985):201-231.

[Skuce et al. 85b]
Skuce, D., Matwin, S., Tauzovich, B., Szpakowicz, S., and Oppacher, F., "A Rule-Oriented Methodology for Constructing a Knowledge-Base from Natural Language Documents", *Proceedings of the Expert Systems in Government Symposium*, McLean, VA, Oct. 1985, pp.378-385.

[Skuce 86]
Skuce, D., "Natural Language Synthesis of a Database Reporting Language Using Commercial Expert System Technology", TR-86-24, Department of Computer Science, University of Ottawa, December 1986.

[Stanley 87]
Stanley, R., "Artificial Intelligence Applied to the HELP Function", *Proceedings of the HP 3000 International conference*, Vienna, Austria, March 1987.

[Suwa et al. 84]
Suwa, M., Scott, A. C., and Shortliffe, E. H., "Completeness and Consistency in a Rule-Based System", in [Buchanan, Shortliffe 84], pp.159-170.

[Swartout 81]
Swartout, W. R., "Explaining and Justifying Expert Consulting Programs", *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, 1981, pp.815-822.

[Swartout 83]
Swartout, W. R., "XPLAIN: a System for Creating and Explaining Expert Consulting Programs", *Artificial Intelligence* 21 (1983):285-325.

[Szpakowicz et al. 86]
Szpakowicz, S., Matwin, S., and Skuce, D., "QUIZ Advisor: A Consultant for a Fourth Generation Software Package", *Proceedings of the Sixth International Workshop on Expert Systems*, Avignon, France, 1986, pp.155-168.

[Tauzovich et al. 85]
Tauzovich, B., Matwin, S., Oppacher, F., Skuce, D., and Szpakowicz, S., "An Expert Advisory System for Government Regulations: Knowledge Acquisition Methodology", in *Artificial Intelligence in Economics and Management*, edited by L. F. Pau, North Holland, 1986, pp.205-212.

[Waterman 86]
Waterman, D. A., *A Guide to Expert Systems*, Addison-Wesley Publishing Company, Reading, MA, 1986.

[Waters 85]
Waters, R. C., "The Programmer's Apprentice", *IEEE Transactions on Software Engineering* 7 (November 1985).

[Waters 86]
Waters, R. C., "KBEmacs: Where's the AI?", *The AI Magazine* 7 (1986):47-56.

[Weiss et al. 78]
Weiss, S. M., Kulikowski, C. A., Amarel, S., and Safir, A., "A Model-Based Method for Computer-Aided Medical Decision Making", *Artificial Intelligence* 11 (1978):145-172.

[Wilensky 84]
Wilensky, R., "Talking to UNIX in English: An Overview of an On-line UNIX Consultant", *The AI Magazine* (Spring 1984):29-39.

[Wilensky et al. 84]
Wilensky, R., Arens, Y. and Chin, D., "Talking to UNIX in English: An Overview of UC", *Communications of the ACM* 27 (June 1984):574-593.

[Wilensky et al. 86]
Wilensky, R., Mayfield, J., Albert, A., Chin, D., Cox, C., Luria, M., Martin, J., and Wu, D., "UC - A Progress Report", Report No. UCB/CSD 87/303, Computer Science Division, University of California, Berkeley, July 1986.

[Winston 84]
Winston, P. H., *Artificial Intelligence*, 2nd ed., Addison-Wesley Publishing Company, Reading, MA, 1984.

[Winston, Prendergast 84]
Winston, P. H., and Prendergast, K. A., eds., *The AI Business*, The MIT Press, Cambridge, MA, 1984.

APPENDIX A: QAUZ Knowledge Base

```
STATEMENT      ACCESS

PURPOSE        to access (= declare) a single file
               xor
               to access multiple files

SYNTAX/KINDS

     ACCESS  {first_file_declaration:?FILE/file_name}
             *[{Sub: n from 2} {LINK_TO_part:?FILE}] {Count:?NSF}
                 present: ACCESS with a LINK_TO_option,
                 absent:  ACCESS without a LINK_TO_option,
     {end}


PT-SEQUENCES

     first_file_declaration =
         file_declaration
             <t1>        --->  .FILE is a declared file,
             <t2>        --->  records of .FILE are required
                                          (= not optional),;

     file_declaration =
         {file:?DFILE/file_name}
         {Cond: .DFILE is described in the dictionary as a file,}
             <t3>     fail--->  error message ERRA1 is issued,
         [ALIAS_part]
             <t4>        --->  .ALIAS is specified,
             <t5>  absent--->  .DFILE has no alias,
         {Cond: .FILE is a unique file,}
             <t6>     fail--->  error message ERRA2 is issued,;

     ALIAS_part = ALIAS {name:?ALIAS/file_name};

     LINK_TO_part = LINK TO subsequent_file_declaration;

     subsequent_file_declaration =
         file_declaration
             <t7>        --->  .FILE is a declared file,
             <t8>        --->  .FILE is a subordinate file,
         [OPTIONAL_part]
             <t9>        --->  records of .FILE are optional,
             <t10> absent--->  records of .FILE are required,;

     OPTIONAL_part = OPTIONAL;
```

```
PH-RULES

     .FILE is the single file
          <h1>   --->  ?DDREC is set to
                       the dictionary_record of .DFILE,;

     .FILE is the primary file
          <h2>   --->  ?DDREC is set to
                       the dictionary_record of .DFILE,;

     .FILE is a subordinate file
          <h3>   --->  a link for .FILE is established through
                                    an internal procedure,;
          <h4>   --->  ?DDREC is augmented with
                       the dictionary_record of .DFILE,;

     .ALIAS is specified
          <h5>   --->  ?FILE is set to .ALIAS,;

     .DFILE has no alias
          <h6>   --->  ?FILE is set to .DFILE,;

     records of .FILE are optional
          <h7>   --->  ?OPT is set to 'YES',;

     records of .FILE are required
          <h8>   --->  ?OPT is set to 'NO',;


INTRO-PARAMS

     FILE[n]        is a file,
     DFILE[n]       is the dictionary_file of .FILE[n],
     ALIAS[n]       is the alias (= alternative_name) of .DFILE[n],
     OPT[n]         is the optional_records_flag of .FILE[n],
     DDREC          is the description of record_complexes,
     NSF            is the number of subordinate files,


ERROR-MESSAGES

     ERRA1          Expected: RECORD key file,
                    (the file was not found in the dictionary,)

     ERRA2          Invalid file name,
                    (file names are not unique - need aliasing,)

END
```

```
STATEMENT      ACCESS with a LINK_TO_option

PURPOSE        to access multiple files

SYNTAX/KINDS

     ACCESS  {first_file_declaration:?FILE}
             **[{Sub: n from 2} {LINK_TO_part}] {Count:?NSF}
     {end}

PT-SEQUENCES

             <t1>       --->  .FILE is the primary file,;

END
```

```
STATEMENT      ACCESS without a LINK_TO_option

PURPOSE        to access a single file

SYNTAX/KINDS

     ACCESS  {first_file_declaration:?FILE} {end}

PT-SEQUENCES

             <t1>       --->  .FILE is the single file,;

END
```

```
STATEMENT      SELECT

PURPOSE        to specify a selection_condition
                                    (= selection_criterion)
               xor
               to cancel (= disable) a selection_condition

SYNTAX/KINDS

     SELECT [file_part]
                 present:  SELECT with a file_option,
                 absent:   SELECT without a file_option,
            [IF_part]
     {end}


PT-SEQUENCES

     file_part =
         {file:?SFILE/file_name}
         {Cond: .SFILE is a declared file,}
            <t1>  fail--->  error message ERRS1 is issued,;

     IF_part =
         IF {condition:?COND}
         {Cond: the type of .CONT is equal to the type of .EXPR,}
            <t2>  fail--->  error message ERRS2 is issued,;

     condition = {content:?CONT}  operator  expression;

     content = item;

     operator = |[ EQ, NE, LT, LE, GE, GT]{Choice:?OP};

     expression = |[number, string, item]{Choice:?EXPR};

     number = {?N/number};

     string = {?STR/string};

     item = dictionary_item;

     dictionary_item =
         {item_name:?ITEM/item_name}
         {Cond: .ITEM is described in the dictionary as an item,}
            <t3>     fail--->  error message ERRS3 is issued,
         [item_qualifier:.QUAL]
            <t4>         --->  .QUAL is specified,
            <t5>  absent--->  a default for .QUAL is used,;
```

```
    item_qualifier =
        OF {file:?QUAL/file_name}
        {Cond: .QUAL is a declared file,}
           <t6>    fail--->  error message ERRS4 is issued,
        {Cond: .ITEM is described in the dictionary as
                                              an item of .QUAL,}
           <t7>    fail--->  error message ERRS5 is issued,;

PH-RULES

    a default for .QUAL is used
        <h1>   --->  ?QUAL is determined
                     through an internal procedure;

INTRO-PARAMS

    SFILE          is a file,
    COND           is a selection_condition,
    CONT           is the content of .COND,
    OP             is the operator of .COND,
    EXPR           is the expression of .COND,
    ITEM           is an item,
    QUAL           is the qualifier of .ITEM,
    N              is a number,
    STR            is a string

ERROR-MESSAGES

    ERRS1          Expected:  file  IF  eol,
                   (the file was not declared,)

    ERRS2          The data type is not the expected one,
                   (the data types in the condition do not agree,)

    ERRS3          Expected:  item,
                   (the item was not found in the dictionary,)

    ERRS4          Expected:  file  LT  LE  EQ  GE  GT  NE,
                   (the item qualifier .QUAL is not a declared file,)

    ERRS5          Cannot find the item .ITEM OF .QUAL,
                   (the qualified item was not found in
                                                the dictionary,)

END
```

STATEMENT SELECT with a file_option

PURPOSE to specify a selection_condition on records of a file
xor
to cancel a selection_condition on records of a file

SYNTAX/KINDS

```
SELECT file_part
       [IF_part]
            present:  SELECT file IF,
            absent:   SELECT file,
{end}
```

END

STATEMENT SELECT file IF

PURPOSE to specify a selection_condition on records of a file

SYNTAX/KINDS

SELECT file_part IF_part {end}

PT-SEQUENCES

```
        <t1>  --->  there is a selection_condition
                                    on records of .SFILE,;
```

END

STATEMENT SELECT file

PURPOSE to cancel selection_condition on records of a file

SYNTAX/KINDS

SELECT file_part {end}

PT-SEQUENCES

```
        <t1>  --->  there is no selection_condition
                                    on records of .SFILE,;
```

END

```
STATEMENT      SELECT without a file_option

PURPOSE        to specify selection_condition on record_complexes
               xor
               to cancel selection_condition on record_complexes

SYNTAX/KINDS

     SELECT [IF_part]
                present:   SELECT IF,
                absent:    SELECT without file or IF,
     {end}

END
```

```
STATEMENT      SELECT IF

PURPOSE        to specify a selection_condition on record_complexes

SYNTAX/KINDS

     SELECT  IF_part  {end}

PT-SEQUENCES

          <t1>   --->  there is a selection_condition
                                        on record_complexes,;
END
```

```
STATEMENT      SELECT without file or IF

PURPOSE        to cancel selection_condition on record_complexes

SYNTAX/KINDS

     SELECT {end}

PT-SEQUENCES

          <t1>   --->  there is no selection_condition
                                        on record_complexes,;
END
```

```
STATEMENT     REPORT

PURPOSE       to report
              xor
              to cancel reporting

SYNTAX/KINDS

     REPORT [report_specification]
                 present:   REPORT with a specification,
                 absent:    REPORT without a specification,
     {end}


PT-SEQUENCES

     report_specification =
         ![report_group, ALL_option]{Choice:?RS};

     report_group =
         report_items
         [trailing_skip_part:?TRS]
             <t1>          --->  .TRS is specified,
         [RESERVE_LINES_part]
             <t2>          --->  .RESL is specified,
             <t3>  absent--->  no lines are reserved;

     report_items =
         **[{Sub: n from 1} {report_item:?RITEM}] {Count:?NRI};

     report_item =
         [SKIP_part:?SAR]
             <t4>          --->  .SAR[n] is specified,
         [TAB_part]
             <t5>          --->  .TAB[n] is specified,
             <t6>  absent--->  a default for .TAB[n] is used,
         item
             <t7>          --->  .ITEM[n] is specified,
         [format_part];

     ALL_option =
         ALL
         [SKIP_lines_part]
             <t8>          --->  .NL is specified,;

     trailing_skip_part = SKIP_part;

     SKIP_part =
         SKIP
         [SKIP_amount]
             <t9>  absent--->  a default for .SA is used,;
```

```
SKIP_amount = |[number_of_lines, PAGE] {Choice:?SA};

RESERVE_LINES_part =
    RESERVE {n:?RESL/number}
    [LINES_part];

LINES_part = LINES;

TAB_part = TAB {n:?TAB/number};

format_part =
    [HEADING_part]
        <t10>        --->  .HSTR is specified,
        <t11> absent--->  a default for .HSTR is used,
    [PICTURE_part]
        <t12>        --->  .PICT is specified,
        <t13> absent--->  a default for .PICT is used,;

HEADING_part = HEADING {string:?HSTR/string};

PICTURE_part = PICTURE {string:?PICT/picture_string};

SKIP_lines_part = SKIP number_of_lines;

number_of_lines = {n:?NL/number};
```

PH-RULES

```
a default for .SA is used
     <h1>   --->  ?SA is set to 1,;

a default for .HSTR is used
     <h2>   --->  ?HSTR is set to the dictionary_heading
                                                  of .ITEM,;

a default for .PICT is used
     <h3>   --->  ?PICT is set to the dictionary_picture
                                                  of .ITEM,;

no record_items are requested for reporting
     <h4>   --->  ?NRI is set to 0,;
```

```
INTRO-PARAMS

     RS              is a report_specification,
     RITEM[n]        is a report_item,
     SAR[n]          is the skip_amount of .RITEM[n],
     TAB[n]          is the tab_column of .RITEM[n],
     HSTR[n]         is the column_heading of .RITEM[n],
     PICT[n]         is the picture of .RITEM[n],
     NRI             is the number of report_items,
     SA              is a skip_amount,
     TRS             is the trailing skip,
     RESL            is the number of reserved lines,
     SL              is the leading skip,
     NL              is a number of lines

END
```

```
STATEMENT    REPORT with a specification

PURPOSE      to report

SYNTAX/KINDS :

     REPORT |[report_group, ALL_option]{Choice:?RS} {end}
              <-- REPORT with a report_group,
                            <-- REPORT with the ALL_option,

END
```

```
STATEMENT     REPORT with a report_group

PURPOSE       to report specified items

SYNTAX/KINDS

     REPORT report_group {end}

END
```

```
STATEMENT     REPORT with the ALL_option

PURPOSE       to report all items

SYNTAX/KINDS

     REPORT ALL_option {end}

PT-SEQUENCES

              <t1>        --->  all record_items are
                                    requested for reporting,;

END
```

```
STATEMENT      REPORT without a specification

PURPOSE        to cancel reporting

SYNTAX/KINDS

     REPORT {end}

PT-SEQUENCES

              <t1>        --->  no record_items are requested
                                              for reporting,;
END
```

TOPIC rphase

REP-RULES

```
        .FILE is a declared file
                <1>   --->  .FILE is an accessible file;

        ?RCITEM is a record_item of .FILE,
        .FILE is a declared file
                <2>   --->  .RCITEM is a declared record_item;

        ?RCITEM is a declared record_item
                <3>   --->  .RCITEM is an accessible record_item;

        .HSTR is established
                <4>   --->  .HSTR is printed above .RIVAL;

        .PICT is established
                <5>   --->  .RIVAL is formatted using .PICT;

        .TAB is established
                <6>   --->  .RIVAL is printed in .TAB;

        a default for .TAB is used
                <7>   --->  ?TAB is determined through an internal
                                                        procedure;

        .SL is established
                <8>   --->  .SL is skipped before the reporting of
                                                the next record_complex;

        .TRS is established
                <9>   --->  .TRS is skipped after the reporting of
                                                the next report_group;

        .SAR is established
                <10>  --->  .SAR is skipped before the reporting
                                                        of .RIVAL;
```

```
no record_items are requested for reporting
        <11>   --->  nothing is reported;


the length of .RIVAL is greater than the length of .PICT
        <12>   --->  the status of .RITEM is 'OVERFLOW';


the status of .RITEM is 'OVERFLOW',
the type of .ITEM/.RITEM is 'numeric'
        <13>   --->  ?FIELD is filled with
                                        the crosshatch_sign;


the status of .RITEM is 'OVERFLOW',
the type of .ITEM/.RITEM is 'string'
        <14>   --->  ?FIELD is truncated;


.RITEM[m] precedes .RITEM[n] in the REPORT statement
        <15>  --->  .FIELD[m] precedes .FIELD[n];


all record_items are requested for reporting,
?RCITEM is an accessible record_item
        <16>  --->   the data for .RCITEM are reported
                                                  in .FIELD;
        <17>  --->   ?HSTR is set to
                        the dictionary_heading of .ITEM;
        <18>  --->   ?PICT is set to
                        the dictionary_picture of .ITEM;

.WDL is greater than .WPL
        <19>  --->  there is a wraparound;


.RESL is established,
.SCP is less than .RESL
        <20>  --->  .REPC is printed on the next page,;


.SCP is less than .SREPC,
no lines are reserved
        <21>  --->  a part of .REPC is printed on
                                             the next page;


there is a wraparound
        <22>  incr--->  .SREPC;


there is a selection_condition on record_complexes
        <23>  decr--->  .NRC;
```

```
        there is no selection_condition on record_complexes
                <24>  incr--->  .NRC,


        there is a selection_condition on records of .FILE,
        .OPT is set to 'NO'
                <25>  decr--->  .NRC;


        there is a selection_condition on records of .FILE,
        .OPT is set to 'YES'
                <26>  incr--->  .NRC;

        .NSF
                <27>  prop--->  the range of available data
                                                for reporting;

        .TAB
                <28>  prop--->  the right_shift at .RITEM;


        .SAR
                <29>  prop--->  the down_shift at .FIELD;
                <30>  prop--->  .SREPC;


        .TRS
                <31>  prop--->  the down_shift at .REPC;

INTRO-PARAMS

        FIELD[n]        is the field of .RITEM[n],
        RIVAL[n]        is the value of .ITEM/.RITEM[n],
        REPC            is a report_complex,
        NRC             is the number of report_complexes,
        SREPC           is the size of .REPC,
        SCP             is the space on the current page,
        WDL             is the width of the detail_line,
        WPL             is the width of the print_line,
        RCITEM          is a record_item,

END
```

APPENDIX B: An Example of a QAUZ Session

```
| ?- qauz.
------------------------
System:    Quiz Advisor
Subsystem: QAUZ 1.1
------------------------
Author: B.Tauzovich

  (Type 'h' for help)

> h

    A QAUZ query consists of the following:

        Assumptions (optional)
        Quiz code (optional)
        A question

    Commands:
                a    update assumptions
                c    update QUIZ code
                s    show assumptions and code
                d    delete all assumptions and code
                q    enter a question
                m    show this menue
                h    help
                e    exit QAUZ

> q
    Enter your question:
    >> h

    The following types of questions are expected:

      Why ...?
      Why ... not ...?
      Why did I get the error message ...?
      What happens if I use ...?
      When is it true that ...?
      Syntax Questions:
         What kinds of statements are there?
         What are the different kinds of ... statement?
         What is the syntax of ...?
         What is the complete syntax of ...?
         What is ... a component of?
         Where is ... used?

    >> What is the syntax of ACCESS statement ?

         ACCESS first_file_declaration [LINK_TO_part]...

> What is the syntax of SELECT statement ?

         SELECT [file_part] [IF_part]

> Where is ALIAS used ?
```

'ALIAS' is

```
in--> ALIAS_part
      in--> file_declaration
            in--> first_file_declaration
                  in--> 'ACCESS' statement
            in--> subsequent_file_declaration
                  in--> LINK_TO_part
                        in--> 'ACCESS with a LINK_TO_option' statement
```

> What is the complete syntax of ACCESS statement ?

```
ACCESS
first_file_declaration
     file_declaration
          file
          [ALIAS_part]
               ALIAS
               name
[LINK_TO_part]...
     LINK
     TO
     subsequent_file_declaration
          file_declaration
               file
               [ALIAS_part]
                    ALIAS
                    name
          [OPTIONAL_part]
               OPTIONAL
```

> What kinds of statements are there ?

```
ACCESS
     ACCESS with a LINK_TO_option
     ACCESS without a LINK_TO_option

SELECT
     SELECT with a file_option
          SELECT file IF
          SELECT file
     SELECT without a file_option
          SELECT IF
          SELECT without file or IF

REPORT
     REPORT with a specification
          REPORT with a report_group
          REPORT with the ALL_option
     REPORT without a specification
```

> a

No assumptions
Update the assumptions:

```
1 >h
   * Options:  <lesk_statement> | del(X) | <cr>
     where X is 'all', a number, or a list of numbers
```

```
                    * Enter options one at a time
                  1 >'AMOUNT' is an item .
                  2 >'AMOUNT' is described in the dictionary as an item
                  3 >


> c
    No code
    Enter your QUIZ code:
                  >> h
                     * Enter one statement per line
                     * Use only capital letters in keywords
                     * To end type <cr> after the prompt
                     NOTE: another way to replace the code is to
                           enter the following type of assumption:
                           I used "<stmt1>;<stmt2>;..."
                  >> SELECT
                  >> ACCESS EMPLOYEES ALIAS STAFF LINK TO BILLINGS OPTIONAL
                  >> SELECT EMPLOYES
                  >> REPORT ALL
                  >> SELEC IF AMOUNT LE LIMIT
                     Not a statement (input ignored)
                  >> SELECT IF AMOUNT LE LIMIT
                  >> REPORT EMPLOYEE OF STAFF PICTURE "^^^^^" LASTNAME HEADING "Name"
                  >>


> What happens if I use the code ?

    SELECT
    ACCESS EMPLOYEES ALIAS STAFF LINK TO BILLINGS OPTIONAL
    SELECT EMPLOYES
    REPORT ALL
    SELECT IF AMOUNT LE LIMIT
    REPORT EMPLOYEE OF STAFF PICTURE "^^^^^" LASTNAME HEADING "Name"

    Analyzing: SELECT  (ST1)

     (1)  there is no selection_condition on record_complexes
     (2)  ST1 is a 'SELECT without file or IF' statement

    Analyzing: ACCESS EMPLOYEES ALIAS STAFF LINK TO BILLINGS OPTIONAL  (ST2)
    NOTE: All previous statements are cancelled!

     (1)  EMPLOYEES is described in the dictionary as a file (UNCERTAIN)
     (2)  STAFF is the alias of EMPLOYEES
     (3)  STAFF is a unique file (VERIFIED)
     (4)  STAFF is a declared file, therefore
     (5)       STAFF is an accessible file
     (6)  records of STAFF are required
          For file 2 :
     (7)     BILLINGS is described in the dictionary as a file (UNCERTAIN)
     (8)     BILLINGS has no alias
     (9)     BILLINGS is a unique file (VERIFIED)
    (10)     BILLINGS is a declared file, therefore
    (11)          BILLINGS is an accessible file
    (12)     BILLINGS is a subordinate file, therefore
    (13)          a link for BILLINGS is established through an internal procedure
```

```
(14)     records of BILLINGS are optional
(15)  STAFF is the primary file
(16)  ST2 is an 'ACCESS with a LINK_TO_option' statement

Analyzing: SELECT EMPLOYES  (ST3)

(17)  error message ERRS1 is issued:
       " Expected : file IF eol"
 (The rest of the statement is ignored)

Analyzing: REPORT ALL  (ST4)

(18)  all record_items are requested for reporting
(19)  ST4 is a 'REPORT with the ALL_option' statement

Analyzing: SELECT IF AMOUNT LE LIMIT  (ST5)

(20)  AMOUNT is described in the dictionary as an item (ASSUMED)
(21)  a default for the qualifier is used, therefore
(22)       the qualifier is determined through an internal procedure
(23)  LIMIT is described in the dictionary as an item (UNCERTAIN)
(24)  a default for the qualifier is used, therefore
(25)       the qualifier is determined through an internal procedure
(26)  the type of AMOUNT is equal to the type of LIMIT (UNCERTAIN)
(27)  there is a selection_condition on record_complexes
(28)  ST5 is a 'SELECT IF' statement

Analyzing: REPORT EMPLOYEE OF STAFF PICTURE "^^^^^" LASTNAME HEADING "Name"  (ST6)
NOTE: ST4 is cancelled!

      For report_item 1 :
(29)     a default for the tab_column is used, therefore
(30)          the tab_column is determined through an internal procedure
(31)     EMPLOYEE is described in the dictionary as an item (UNCERTAIN)
(32)     STAFF is a declared file (VERIFIED)
(33)     EMPLOYEE is described in the dictionary as an item of STAFF (UNCERTAIN)
(34)     STAFF is the qualifier of EMPLOYEE
(35)     EMPLOYEE is an item
(36)     a default for the column_heading is used, therefore
(37)          the column_heading is set to the dictionary_heading of EMPLOYEE, ther
(38)               the column_heading is printed above the value
(39)     "^^^^^" is the picture of report_item, therefore
(40)          the value is formatted using "^^^^^"
      For report_item 2 :
(41)     a default for the tab_column is used, therefore
(42)          the tab_column is determined through an internal procedure
(43)     LASTNAME is described in the dictionary as an item (UNCERTAIN)
(44)     a default for the qualifier is used, therefore
(45)          the qualifier is determined through an internal procedure
(46)     LASTNAME is an item
(47)     "Name" is the column_heading of LASTNAME, therefore
(48)          "Name" is printed above the value
(49)     a default for the picture is used, therefore
(50)          the picture is set to the dictionary_picture of LASTNAME, therefore
(51)               the value is formatted using the picture
(52)  no lines are reserved
(53)  ST6 is a 'REPORT with a report_group' statement
```

```
> What happens if I use "ACCESS EMPLOYEES ; SELECT EMPLOYEES" ?
                              ACCESS EMPLOYEES
                              SELECT EMPLOYEES

   ACCESS EMPLOYEES
   SELECT EMPLOYEES

   Analyzing: ACCESS EMPLOYEES  (ST1)

    (1)  EMPLOYEES is described in the dictionary as a file (UNCERTAIN)
    (2)  EMPLOYEES has no alias
    (3)  EMPLOYEES is a unique file (VERIFIED)
    (4)  EMPLOYEES is a declared file, therefore
    (5)       EMPLOYEES is an accessible file
    (6)  records of EMPLOYEES are required
    (7)  EMPLOYEES is the single file
    (8)  ST1 is an 'ACCESS without a LINK_TO_option' statement

   Analyzing: SELECT EMPLOYEES  (ST2)

    (9)  EMPLOYEES is a declared file (VERIFIED)
   (10)  there is no selection_condition on EMPLOYEES
   (11)  ST2 is a 'SELECT file' statement


> When is it true that records are required ?

     records of a file are required,
       CONSEQUENCE OF:
          using first_file_declaration
          in 'ACCESS' statement
        OR
          using subsequent_file_declaration without
          OPTIONAL_part
          in LINK_TO_part
             in 'ACCESS with a LINK_TO_option' statement

> d
    No assumptions
    No code

> Why did I get the error message "Cannot find the item ADDRESS OF BILLINGS" ?

    The message "Cannot find the item ADDRESS OF BILLINGS" is issued
    because the qualified item was not found in the dictionary.

> Why did I get the error message "Expected : file" ?

    "Expected : file" is not a valid error message.
        Was the message "Expected : RECORD key file"? (y/n) >> n
        Was the message "Expected : file IF eol"? (y/n) >> n
        Was the message "Expected : file LT LE EQ GE GT NE"? (y/n) >> n

    Enter your QUIZ code:
                >> ACCESS BILLINGS
                >> SELECT BILINGS
                >>
```

```
    You should have been issued error message:
         "Expected : file IF eol"
    because the file was not declared
    (Statement: SELECT BILINGS).


> c
    QUIZ CODE:
                    ACCESS BILLINGS
                    SELECT BILINGS

    Action?  (<cr>=keep, r=replace, d=delete)  > r
    Enter your QUIZ code:
                 >> ACCESS EMPLOYEES LINK TO BILLINGS
                 >> REPORT AMOUNT OF BILLINGS
                 >>


> Why did I get an error message ?

    Conditions that are potentially violated:

       EMPLOYEES is described in the dictionary as a file
            (causing error message:
             "Expected : RECORD key file"
             at ACCESS statement).

       BILLINGS is described in the dictionary as a file
            (causing error message:
             "Expected : RECORD key file"
             at ACCESS statement).

       AMOUNT is described in the dictionary as an item
            (causing error message:
             "Expected : item"
             at REPORT statement).

       AMOUNT is described in the dictionary as an item of BILLINGS
            (causing error message:
             "Cannot find the item AMOUNT OF BILLINGS"
             at REPORT statement).

    Does this explain your problem? (y/n) >> n

    The following code has been considered:
          ACCESS EMPLOYEES LINK TO BILLINGS
          REPORT AMOUNT OF BILLINGS
    Is this your entire code? (y/n) >> y

    Provided you supplied correct evidence,
    the error doesn't originate from your code!

> What happens if I use "REPORT X SKIP PAGE" ?
                              REPORT X SKIP PAGE

   REPORT X SKIP PAGE

   Analyzing: REPORT X SKIP PAGE  (ST1)
```

```
      For report_item 1 :
(1)      a default for the tab_column is used, therefore
(2)           the tab_column is determined through an internal procedure
(3)      X is described in the dictionary as an item (UNCERTAIN)
(4)      a default for the qualifier is used, therefore
(5)           the qualifier is determined through an internal procedure
(6)      X is an item
(7)      a default for the column_heading is used, therefore
(8)           the column_heading is set to the dictionary_heading of X, therefore
(9)                the column_heading is printed above the value
(10)     a default for the picture is used, therefore
(11)          the picture is set to the dictionary_picture of X, therefore
(12)               the value is formatted using the picture
(13)  PAGE is the trailing skip, therefore
(14)       PAGE is skipped after the reporting of the next report_group
(15)  no lines are reserved
(16)  ST1 is a 'REPORT with a report_group' statement
```

```
> Where is SKIP used ?

    'SKIP' is

        in--> SKIP_lines_part
              in--> ALL_option
                    in--> 'REPORT with the ALL_option' statement
        in--> SKIP_part
              in--> report_item
                    in--> report_items
                          in--> report_group
                                in--> 'REPORT with a report_group' statement
              in--> trailing_skip_part
                    in--> report_group
                          in--> 'REPORT with a report_group' statement
```

```
> What happens if I use "REPORT X SKIP PAGE Y" ?
                          REPORT X SKIP PAGE Y

   REPORT X SKIP PAGE Y

   Analyzing: REPORT X SKIP PAGE Y  (ST1)

      For report_item 1 :
(1)      a default for the tab_column is used, therefore
(2)           the tab_column is determined through an internal procedure
(3)      X is described in the dictionary as an item (UNCERTAIN)
(4)      a default for the qualifier is used, therefore
(5)           the qualifier is determined through an internal procedure
(6)      X is an item
(7)      a default for the column_heading is used, therefore
(8)           the column_heading is set to the dictionary_heading of X, therefore
(9)                the column_heading is printed above the value
(10)     a default for the picture is used, therefore
(11)          the picture is set to the dictionary_picture of X, therefore
(12)               the value is formatted using the picture
      For report_item 2 :
(13)     PAGE is the skip_amount of Y, therefore
(14)          PAGE is skipped before the reporting of the value
```

```
(15)      a default for the tab_column is used, therefore
(16)           the tab_column is determined through an internal procedure
(17)      Y is described in the dictionary as an item (UNCERTAIN)
(18)      a default for the qualifier is used, therefore
(19)           the qualifier is determined through an internal procedure
(20)      Y is an item
(21)      a default for the column_heading is used, therefore
(22)           the column_heading is set to the dictionary_heading of Y, therefore
(23)                the column_heading is printed above the value
(24)      a default for the picture is used, therefore
(25)           the picture is set to the dictionary_picture of Y, therefore
(26)                the value is formatted using the picture
(27)   no lines are reserved
(28)   ST1 is a 'REPORT with a report_group' statement

> What is the complete syntax of report_item ?

        [SKIP_part]
             SKIP
             [SKIP_amount]
                  number_of_lines
                       n
                  /
                  PAGE
        [TAB_part]
             TAB
             n
        item
             dictionary_item
                  item_name
                  [item_qualifier]
                       OF
                       file
        [format_part]
             [HEADING_part]
                  HEADING
                  string
             [PICTURE_part]
                  PICTURE
                  string

> What is file_declaration a component of ?

        first_file_declaration
        subsequent_file_declaration

> d
    No assumptions
    No code

> a
    No assumptions
    Update the assumptions:
                1 >'EMPLOYEES' is a file
                2 >
```

```
> c
    No code
    Enter your QUIZ code:
                >> ACCESS EMPLOYEES
                >> SELECT IF CITY NE "Toronto"
                >> SELECT EMPLOYEES
                >> REPORT EMPLOYEE PICTURE "^^^^^^"
                >>

> Why did not I get [ a large number of report complexes ] ?

     the number of report_complexes is small,
       CONSEQUENCE OF:
          there is a selection_condition on
          record_complexes,
            CONSEQUENCE OF:
               using 'SELECT IF' statement

> What are the different kinds of SELECT statement ?

        SELECT
             SELECT with a file_option
                  SELECT file IF
                  SELECT file
             SELECT without a file_option
                  SELECT IF
                  SELECT without file or IF

> Why is [ a default for the column heading ] used ?

     a default for the column_heading of a report_item
     is used,
       CONSEQUENCE OF:
          using format_part without HEADING_part
          in report_item
             in report_items
                in report_group
                   in 'REPORT with a report_group' statement

> Where is item used ?

    'item' is

        in--> content
              in--> condition
                    in--> IF_part
                          in--> 'SELECT file IF' statement
                          in--> 'SELECT IF' statement
        in--> expression
              in--> condition
                    in--> IF_part
                          in--> 'SELECT file IF' statement
                          in--> 'SELECT IF' statement
        in--> report_item
              in--> report_items
                    in--> report_group
                          in--> 'REPORT with a report_group' statement
```

```
> Why is [ a default for the picture ] used ?

    Your query is INCONSISTENT!

       Contradiction found regarding:
           using format_part without PICTURE_part

       The contradiction results from
           a default for the picture of a report_item is
           used  (question)
         and
           REPORT EMPLOYEE PICTURE "^^^^^^"  (code)

    Either your assumptions and/or QUIZ statements are incorrect
    or you did not formulate your question properly!

    Assuming the latter...


    The NEGATION of your question is TRUE
    because the following DOES NOT HOLD:
          using format_part without PICTURE_part
          in report_iten
             in report_items
                in report_group
                   in 'REPORT with a report_group' statement (f)

> When is it true that a picture is established ?

     the picture of a report_item is established,
       CONSEQUENCE OF:
          the picture of the report_item is specified in
          the code,
            CONSEQUENCE OF:
               using PICTURE_part
               in report_item
                  in report_items
                     in report_group
                        in 'REPORT with a report_group' statement
      OR
          the picture of the report_item is set to the
          dictionary_picture of an item,
            CONSEQUENCE OF:
               all record_items are requested for reporting,
                 CONSEQUENCE OF:
                    using 'REPORT with the ALL_option' statement
               AND
               a record_item is an accessible record_item,
                 CONSEQUENCE OF:
                    the record_item is a declared record_item,
                      CONSEQUENCE OF:
                         the record_item is a record_item of a file
                         AND
                         the file is a declared file,
                           CONSEQUENCE OF:
                              using first_file_declaration
                              in 'ACCESS' statement
                            OR
```

```
                                using subsequent_file_declaration
                                in LINK_TO_part
                                   in 'ACCESS with a LINK_TO_option' statement
                    OR
                      a default for the picture of the report_item is
                      used,
                        CONSEQUENCE OF:
                           using format_part without PICTURE_part
                           in report_item
                              in report_items
                                 in report_group
                                    in 'REPORT with a report_group' statement

> When there is a wraparound ?
   Illegal input - parsing failed
   SEQAP: error(illegal_question_form,[when,there,is,a,wraparound])

> When is it true that there is a wraparound ?

     there is a wraparound,
       CONSEQUENCE OF:
          the width of the detail_line is greater than the
          width of the print_line

> When is it true that a tab column is determined through an internal procedure ?

     the tab_column of a report_item is determined
     through an internal procedure,
       CONSEQUENCE OF:
          a default for the tab_column of the report_item
          is used,
            CONSEQUENCE OF:
               using report_item without TAB_part
               in report_items
                  in report_group
                     in 'REPORT with a report_group' statement

> Why is footing printed ?

    Sorry can't answer this question...

> Why is not 'EMPLOYEES' the primary file ?

    Because the following DOES NOT HOLD:
          using 'ACCESS with a LINK_TO_option' statement (f)

> a
    ASSUMPTIONS:
                  1  'EMPLOYEES' is a file
    Update the assumptions:
                  2 >all record itms are requested
                  Illegal input - parsing failed
                  SEQAP: Lexical error
                  (input ignored)
                  2 >all record items are requested
                  3 >no record items are requested
                  4 >
```

```
> c
    QUIZ CODE:
                    ACCESS EMPLOYEES
                    SELECT IF CITY NE "Toronto"
                    SELECT EMPLOYEES
                    REPORT EMPLOYEE PICTURE "^^^^^^"

    Action?  (<cr>=keep, r=replace, d=delete)  >


> Why is the field truncated ?

    Your query is INCONSISTENT!

       Contradiction found regarding:
           all record_items are requested for reporting

    Please examine your query...

> a
    ASSUMPTIONS:
                1  'EMPLOYEES' is a file
                2  all record_items are requested
                3  no record_items are requested
    Update the assumptions:
                4 >del(al)
                Invalid input, try again
                4 >del(all)
    No assumptions
                1 >


> Why is the field truncated ?

     the field of a report_item is truncated,
       CONSEQUENCE OF:
          the status of the report_item is OVERFLOW,
            CONSEQUENCE OF:
               the length of the value of the item of the
               report_item is greater than the length of the
               picture of the report_item
          AND
          the type of the item of the report_item is string

> a
    No assumptions
    Update the assumptions:
                1 >an alias is specified
                2 >


> Why was the column heading not established ?

    Your query is INCONSISTENT!

       Contradiction found regarding:
           using ALIAS_part
```

```
Please examine your query...

   The contradiction results from
       the alias of the dictionary_file of a file is
       specified in the code  (assumption)
     and
       ACCESS EMPLOYEES  (code)

> a
ASSUMPTIONS:
              1  an alias is specified
Update the assumptions:
              2 >del(1)
No assumptions
              1 >


> Why was the column heading not established ?

   Your query is INCONSISTENT!

      Contradiction found regarding:
          using format_part without HEADING_part

      The contradiction results from
          not true that the column_heading of a report_item is
          established  (question)
        and
          REPORT EMPLOYEE PICTURE "^^^^^^"  (code)

   Either your QUIZ statements are incorrect
   or you did not formulate your question properly!

   Assuming the latter...

    the column_heading of a report_item is
    established,
      CONSEQUENCE OF:
         the column_heading of the report_item is set to
         the dictionary_heading of an item,
           CONSEQUENCE OF:
              a default for the column_heading of the
              report_item is used,
                CONSEQUENCE OF:
                   using format_part without HEADING_part
                   in report_item
                      in report_items
                         in report_group
                            in 'REPORT with a report_group' statement

> Why is the column heading set to the dictionary heading ?

    the column_heading of a report_item is set to the
    dictionary_heading of an item,
      CONSEQUENCE OF:
         a default for the column_heading of the
         report_item is used,
           CONSEQUENCE OF:
```

```
                using format_part without HEADING_part
                in report_item
                   in report_items
                      in report_group
                         in 'REPORT with a report_group' statement

> a
    No assumptions
    Update the assumptions:
                  1 >I used "SELECT IF DATEINPUT EQ DATEENTER"
                            SELECT IF DATEINPUT EQ DATEENTER
                     (the above code is now in effect)
                  1 >'DATEINPUT' is an item
                  2 >the type of 'DATEINPUT' is string
                  3 >'DATEENTER' is an item
                  4 >the type of 'DATEENTER' is numeric
                  5 >


> Why did I get an error message@?

    You should have been issued error message:
         "The data type is not the expected one"
    because the data types in the condition do not agree.


> d
    No assumptions
    No code

> Why did I get [ a small number of report complexes ] ?

     the number of report_complexes is small,
       CONSEQUENCE OF:
          there is a selection_condition on
          record_complexes (?),
            CONSEQUENCE OF:
               using 'SELECT IF' statement (?)
        OR
          there is a selection_condition on a file (?),
            CONSEQUENCE OF:
               using 'SELECT file IF' statement (?)
          AND
          records of the file are required (?),
            CONSEQUENCE OF:
               using first_file_declaration
               in 'ACCESS' statement (?)
             OR
               using subsequent_file_declaration without
               OPTIONAL_part
               in LINK_TO_part
                  in 'ACCESS with a LINK_TO_option' statement (?)

> c
    No code
    Enter your QUIZ code:
                  >> SELECT IF ELEM NE "123199"
                  >>
```

```
> a
    No assumptions
    Update the assumptions:
                1 >'ELEM' is an item
                2 >the type of 'ELEM' is numeric
                3 >


> Why does the code not work ?

    You should have been issued error message:
         "The data type is not the expected one"
    because the data types in the condition do not agree.


> d
    No assumptions
    No code

> Why is nothing reported ?

     nothing is reported,
       CONSEQUENCE OF:
          no record_items are requested for reporting,
            CONSEQUENCE OF:
               using 'REPORT without a specification' statement

> a
    No assumptions
    Update the assumptions:
                1 >I used "ACCESS A LINK TO B LINK TO B"
                          ACCESS A LINK TO B LINK TO B
                   (the above code is now in effect)
                1 >


> Why did I get [ a small number of fields ] ?

    You should have been issued error message:
         "Invalid file name"
    because file names are not unique - need aliasing.


> d
    No assumptions
    No code

> a
    No assumptions
    Update the assumptions:
                1 >'X' is a report_item
                2 >the type of 'X' is numeric
                3 >


> Why is the field filled with the crosshatch sign ?
```

```
     the field of a report_item is filled with the
     crosshatch_sign,
       CONSEQUENCE OF:
          the status of the report_item is OVERFLOW,
            CONSEQUENCE OF:
               the length of the value of the item of the
               report_item is greater than the length of the
               picture of the report_item
          AND
          the type of the item of the report_item is
          numeric

> e

yes
| ?-
```