EXPERT SYSTEM FOR PREOPERATIVE ASSESSMENTS

INVESTIGATIONS IN THE DEVELOPMENT OF A WINDOWS-BASED EXPERT SYSTEM FOR PREOPERATIVE ASSESSMENTS

By

DEL ARCHER, B.Sc.

A Project
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree
Master of Science

McMaster University

MASTER OF SCIENCE (1993)
(Computer Science and Systems)

McMASTER UNIVERSITY
Hamilton, Ontario

TITLE: Investigations in the Development of a Windows Based Expert System for Preoperative Assessments

AUTHOR: Delbert Wayne Archer, B.Sc. (McMaster University)

SUPERVISOR: Professor N. Solntseff
Karl Langton

NUMBER OF PAGES: vii, 303

ABSTRACT

The Preop medical expert system (Langton et. al, 1990) was originally developed using the expert system tool, Nexpert Object, on a VAX computer. Nexpert Object creates an expert system specification which is executed by an interpreter within Nexpert Object. The original implementation, however, has several limitations, including:

1. lack of physical portability
2. requires Nexpert Object to run
3. crude user interface

In order to overcome the first limitation, Preop is implemented on a PC DOS portable computer. This project is addresses the other two limitations. Creating a compiled version of Preop eliminates the need for the Nexpert Object interpreter, and implementing it as a Microsoft Windows application provides a better user interface.

ACKNOWLEDGEMENTS

I would like to thank both K. Langton and N. Solntseff for all their guidance and patience. I would also like to thank my parents, Del and Sylvia, for all the support they have given me as I pursued my academic interest. Lastly, I would like to thank my wife, Lou-Ann. She has constantly encouraged and motivated me.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Introduction

For the past several years, the Health Information Research Unit (HIRU) of the Faculty of Health Sciences and the Department of Computer Science and Systems (CSS) at McMaster University have been collaborating on the development of the PREOP system. PREOP is an expert system for medical consultation. Before delving into the specifics of the project, it is necessary to explain a few terms that will be used throughout this paper. One should be familiar with the following key concepts: expert systems, NexpertObject, knowledge bases, MicroSoft Windows applications, and interpreters.

1.2 Terminology

An expert system is an artificial intelligence (AI) application that is created to solve problems related to a particular area of interest (Levine, Drang and Edelson, 1990). The term "expert" reflects two facts about such systems. For one, expert systems are programs that are very specific in their application (i.e., specialists or experts, not generalists). Secondly, such systems are generally developed to concentrate the knowledge of experts in a given area for use by others who are not experts.

Neuron Data's (1991) NexpertObject is an expert system development tool. The tool combines an object-oriented programming approach with rule based inference mechanisms. The next chapter will present a more detailed look at inference mechanisms, while chapter six will discuss the object-oriented paradigm.

NexpertObject is composed of three main parts; besides the main Nexpert kernel which governs the primary functionality of the tool, there is a graphical user interface, allowing developers to work in a point-and-click environment. In addition, there is an application program interface (API) which allows software developers to access NexpertObject functions from programs written in more conventional languages (C, Pascal, and FORTRAN).

A knowledge file is a text file produced by NexpertObject. This file represents the facts and rules that make up the expert knowledge used by the expert system (i.e., the knowledge base). Knowledge bases and the NexpertObject knowledge file will be discussed in greater detail later in this report.

A MicroSoft Windows application (or just Windows application) is an executable program that works with the

MicroSoft Windows operating system (MS Windows). In fact, such an application requires MS Windows for execution. Windows applications make use of the MS Windows interface objects to present the user with a graphical interface that is standard across all Windows applications. Chapters three and five will examine the subjects of user interfaces and Windows applications in further detail.

Another term that requires some explanation is interpreter. For the purposes of this project, an interpreter is a computer program that analyses a line from a source file and simulates, via the computer, the functions described by that line. This means that the interpreter must know the "meaning" of the functions described. That is, the interpreter must be able to translate the functions into machine-executable instructions. A more detailed discussion of interpreters will be presented later in this report.

1.3 PREOP

Having briefly reviewed the key concepts of this project, it is time to describe the PREOP application in more detail. PREOP is an expert system whose purpose is to provide clinicians with a tool to aid in the preoperative assessment of patients about to undergo emergent or urgent surgery (Holbrook et al, 1992).

PREOP is designed to provide medical staff with information about a given patient's chances of developing serious or severe post-operative complications. These probabilities are expressed in terms of a modified multifactorial index (MMI) score. MMI is a modification of an earlier multifactorial index (Langton et al., 1990). The index is a list of cardiac risk factors, each having an associated value. A patients MMI score is the sum of values of all applicable risk factors (Detskey et al, 1986). In addition to the MMI score, the PREOP system also makes recommendations regarding the management (around the time of surgery) of any medications that the patient requires.

To develop the PREOP system, Neuron Data's expert system development shell, NexpertObject (sometimes referred to as simply Nexpert) was used (Langton, 1990). Initially, PREOP was implemented on a VAX station and then later transferred to a Toshiba 3100-SX laptop computer (Holbrook et al, 1992). PREOP has five logical steps, as seen in **Figure 1.1**. The first three gather information from either the clinician or a database of patient information, if the patient's data has already been entered before. The last two steps represent the output of the PREOP system.

(1) History and demographics
(2) Medical status
(3) Medication profile
(4) Probability of an adverse cardiac event
(5) Recommendations

Figure 1.1: Five Steps of PREOP

After a satisfactory prototype of PREOP had been developed, focus shifted from expert system design to a search for a more practical implementation of a working system. That is, the version of PREOP developed using NexpertObject was limited in several ways, and an enhansed version was needed. Foremost among these limitations were:

(a) the system required the NexpertObject software to run.
(b) the system had a very limited user-interface.

The fact that NexpertObject was required for execution of PREOP was less than desirable for two reasons. First, each copy of PREOP would require a copy of NexpertObject, which would have to be paid for. Secondly, packaging PREOP with NexpertObject would require the user to become familiar with NexpertObject, and would allow the user to examine (and possibly alter) PREOP in ways he/she was not intended to.

Equally troubling, however, is the extremely limited user-interface that NexpertObject provides for applications. In particular, once an entry has been made, the user may not go back and change it. For example, suppose a user is prompted for the patients age and he enters it and then proceeds to the next entry prompt. Later, the user discovers that he made a mistake in entering the age value. It is not possible for the user to go back and alter it.

It has been documented that doctors do not favour computerized tools (Covel, Uman, and Manning, 1965; Abate et al, 1989). It follows then that they (physicians) would be even less likely to use a tool that is unwieldy. In the instance of expert systems, many users have particularly high expectations, perhaps because the software falls into the realm of artificial intelligence. In any case, the human-computer interface plays a major role in the acceptability of the software, from a user perspective.

E. Hoogendoorn (1991) developed a PC DOS-based interface using the WINDOWS Toolbox library of functions (Goodwin, 1989). While Hoogendoorn's work addressed some of the problems associated with the original interface, it still was not completely independent of the NexpertObject software. In addition, the PREOP executable file was very large (465K). The high memory requirements could cause problems (in the DOS environment) if subprocesses are spawned from the main program. One can compensate for this by using a product, such as MicroSoft Windows (sometimes simply called Windows or MS Windows), which allows the use of memory beyond 640K. However, if one is using Windows, why not create a Windows application?

1.4 General Outline

This project is a continuation of the expert systems research being carried out by the Health Information Research Unit of the Faculty of Health Sciences and the Department of Computer Science and Systems at McMaster University. It looks to address the two limitations mentioned above. In order to "free" PREOP from the NexpertObject system, an investigation was conducted into the development of an interpreter that interprets a NexpertObject knowledge-base (a fuller explanation is found in subsequent chapters). This investigation led to the implementation of a prototype interpreter that handles a subset of the NexpertObject functions. At the same time, the human-computer interface issue was addressed by implementing the interpreter as a Windows application, taking advantage of the user-friendly, graphic interface provided.

This document describes the work above. Chapter one outlines the purpose of the project, its key concepts, and relates this project to the PREOP research. Chapter two is a discussion of expert systems, especially inference mechanisms and knowledge representation. Both are crucial issues in the development of an expert system.

Chapter three presents an in-depth look at the issues involved in designing and implementing a human-computer interface. For a user-oriented application, the human-computer interface is very important. Design issues related to interpreters are discussed in Chapter four. Specifically, the design of a hybrid interpreter and an abstract machine is examined.

Chapters five and six are closely related to each other. The former is an overview of MicroSoft Windows and the issues related to creating applications for this operating system. In chapter six, the programming paradigm of object-oriented programming is explored, especially how one such language aids in developing Windows applications.

Chapters seven, eight and nine present descriptions of the three programs produced to implement the knowledge file interpreter, NexParse, SGroup and NexMach. And finally, Chapter ten is a discussion of the project in general, as well as an offering of possibilities for the future.

CHAPTER 2

EXPERT SYSTEMS

2.1 Introduction

Among the different branches of artificial intelligence (AI) technology, one branch in particular has proven to be very successful in real world applications: the expert system technology. Although a quick survey of the literature will provide several different definitions for the term, expert system, the following definition will suffice for the purposes of this paper. An expert system is an AI system created to solve problems in a particular domain (Levine, Drang, and Edelson, 1990).

The title, expert system, tends to conjure up ideas of an extremely "intelligent" system. However, the <u>expert</u> in expert system has slightly humbler connotations (Bench-Capon, 1990). In the early days of AI research, the impetus was to develop a "general problem solver". That is, early AI applications were intended to be able to solve problems of any description. Not surprisingly, these attempts met with very limited success. When a human being (a very good "general problem solver") tackles a problem, she draws from past experience and uses **common sense**; the latter has proven to be very difficult to encode in a computer program. Because of this stumbling-block, some researchers put their efforts into the development of AI applications that had very focused areas of purpose. That is, the application was intended to act as a specialist or expert, if you will. In addition, the fact that AI techniques are not generally employed for trivial situations, led to expert systems being utilised in situations where only a handful of people had expertise. The systems were (and are) intended to act as stand-ins for the human experts; this re-enforced the use of <u>expert</u> in the term expert system.

2.2 Components of an Expert System

Figure 2.1 illustrates the various components of a generic expert system. Excluding the user, the most important, and absolutely necessary, components are the user-interface, the knowledge base and the inference engine.

The user-interface serves to manage the interaction between the human user and the other components of the expert system. As alluded to earlier, the importance of the user-interface lies in that it greatly influences the user's opinion of the entire expert system, and this plays a key role in determining whether or not the user will make use of the expert system. The design of a human-computer interface will be investigated in a later chapter.

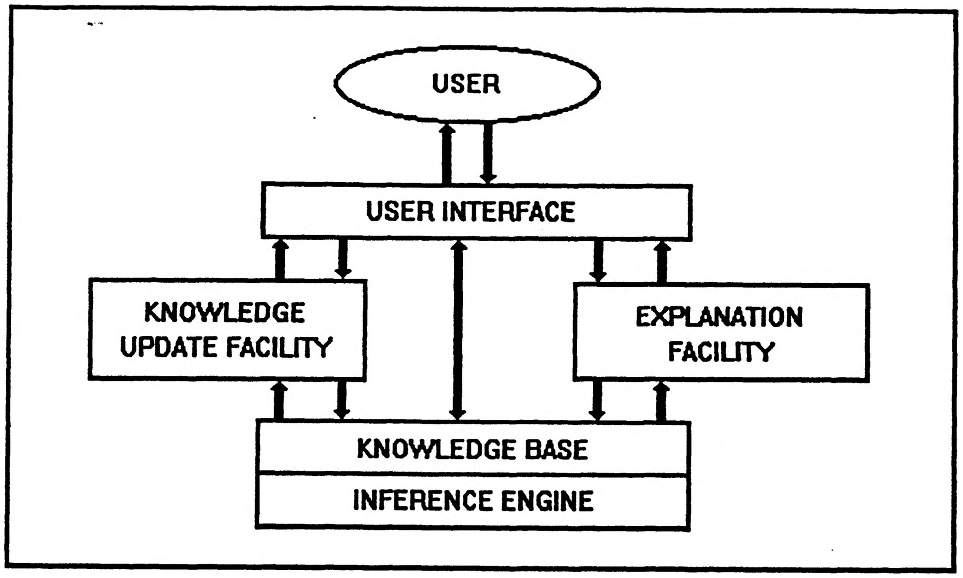

Figure 2.1: A Generic Expert System

An inference is a process of drawing a conclusion through reasoning. The inference engine, then, is the part of the expert system that manipulates the knowledge base to arrive at some conclusion, which is the solution to a given problem. The strategy used to infer (or problem solve) depends on the knowledge representation scheme. Section 2.5 will examine some common inference mechanisms.

In chapter one, the term, "knowledge base", was briefly introduced. Essentially, it is a collection of facts and rules defining relationships between facts. The facts and rules must be acquired from an expert (or experts), and this is the job of a knowledge engineer. Much of the issues surrounding knowledge acquisition falls into the realm of cognitive psychology and is beyond the scope of this paper.

After the knowledge engineer has gathered all the necessary knowledge, it must be translated into a form that is machine useable.

2.3 Knowledge Representation

Knowledge representation is the set of syntactic and semantic conventions that allow for a machine-useable description of facts, processes that change facts, objects related to the facts, and relationships between objects (Bench-Capon, 1990). When seeking a knowledge representation one must ensure that is meets certain criteria. Trevor Bench-Capon (1990) refers to these as **criteria of adequacy**; they are required properties. The first of these is metaphysical adequacy. A representation is metaphysically adequate if, and only if, it does not allow contradictions to exist between the facts that are to be represented. For example, consider the knowledge required for an expert system such as PREOP. In such a system, it is necessary to be able to represent a great deal of information about patients. If the knowledge representation allowed a single patient to have more than one age, for example, it would **not** be metaphysically adequate.

The second criterion for adequacy is epistemic adequacy. To satisfy this criterion, the knowledge representation must allow for the expression of all the facts that are required. Again, using the PREOP expert system as an example, a representation that did not allow for the description of a given patients medications would **not** be epistemically adequate.

Bench-Capon (1990) lists heuristic adequacy as another criterion of adequacy. However, this implies that the knowledge representation must be capable of expressing the reasoning that was used to reach a conclusion. It is doubtful that this is required for a knowledge representation to be useful, and so it will not be listed a required property in this paper.

The final property that a knowledge representation requires is computational tractability. A knowledge representation is computationally tractable if one is able to manipulate it efficiently within a computer system. Because this property depends heavily on the state of hardware and software technology, it is quite possible that representations that do not currently meet this standard, will in the future.

In addition to the criteria of adequacy, there are several other factors that one may wish to consider when choosing a knowledge representation. However, these factors do not represent required properties, but rather, properties that are desirable features. Interested readers are directed to Knowledge Representation by T. Bench-Capon (1990) for a detailed examination of such factors.

2.4 Production Systems

In **Figure 2.2** one will find definitions for the leading knowledge representation paradigms. Because they are one of the representations used by NexpertObject, production rules are of particular importance for this discussion. A production rule is comprised of two parts. There is a left hand side, which consists of one or more conditions, and a right hand side, which is made up of one or more actions (or results) (Hu, 1987).

Paradigm	Description
Production Rules	Knowledge is represented as a set of condition/action pairs (Hu,1987).
Semantic Networks describing	Knowledge is represented as a collection of objects with links between objects,
	relationships between objects (Findler,
1979).	
First Order Logic a	Knowledge is represented as a set of axioms, which are expressed in a formal language, and
	set of inference rules (Rolston, 1988).

Figure 2.2: Knowledge Representation Paradigms

A production system is a system that makes use of production rules to solve problems. Production systems are comprised of three components (Bench-Capon, 1990)

(1) working memory
(2) production memory
(3) rule interpreter

The working memory is a "scratch pad" of sorts. Initially, it contains the initial facts and the desired goal(s). As processing takes place, the working memory is updated with new facts and, perhaps, new goals. This means that the working memory is dynamic. It is important to note that in order for the production system to keep track of progress, it is necessary for the action portion of rules to update the working memory section.

The area of memory used to store the production rules is called production memory. Unlike working memory, production memory is static (i.e., does not change).

The final component of the production system, the rule interpreter, applies the rules stored in production memory to the current facts and updates working memory. The rule interpreter is also responsible for "deciding" when the final conclusion has been reached. The application of rules to facts is not a haphazard affair. On the contrary, a specific inference strategy (or strategies) is employed. The next section will examine this issue further.

2.5 Inference Mechanisms

Because in this project production rules are used to represent knowledge, inference mechanisms will be examined in the context of production rules. Essentially, conclusions are reached (or inferred) by searching through the knowledge base until all pertinent knowledge has been examined. The two inference mechanisms that will be examined here are forward chaining and backward chaining.

Forward chaining is a data-driven search of the knowledge base (Bench-Capon, 1990). The search starts with a set of initial facts, and the production rules are searched for a rule whose condition will be satisfied by the facts. Once such a rule is found, the right hand side or result of the rule is evaluated. This will produce a new set of facts. The search then continues for another rule and so on, until no more rules will apply to the set of facts (Levine, Drang, and Edelson, 1990). An example will give one a better understanding of forward chaining.

Rule 1: If person has sneezed, then person has a cold.

Rule 2: If person has a cold, then person stays in bed.

Rule 3: If person stays in bed, then person won't be at work.

NB: person is a variable representing any human being.

Figure 2.3: A Simple Knowledge Base

Examine the small set of rules found in **Figure 2.3.** These rules define an imaginary expert's knowledge about colds. Let's assume that we wish to know what will happen if the person, Del, sneezes. Our starting set of facts is simply, "Del has sneezed". Searching through the set of rules, it is found that the left hand side of the first rule is satisfied. Evaluating the right hand side of the rule yields the new set of facts, "Del has sneezed" and "Del has a cold". The knowledge base is searched for a rule satisfied by this new set. Of course rule one is still satisfied, but, it has already been evaluated. The second rule is also satisfied. After evaluating its right hand side, the set of facts is "Del has sneezed", "Del has a cold", and "Del stays in bed". If processing continues, the final set of facts will be:

"Del has sneezed"
"Del has a cold"
"Del stays in bed"
"Del won't be at work"

So, because Del sneezed, Del will not be at work. **Figure 2.4** summarizes the forward chaining that took place in this example.

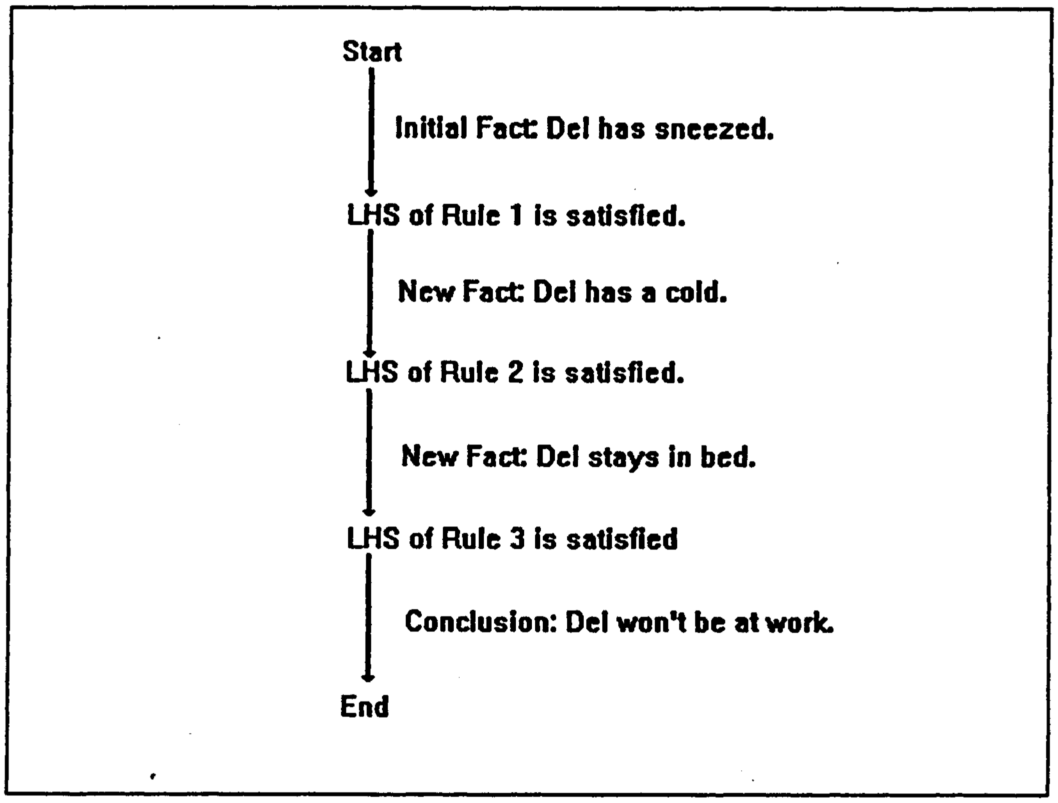
Start
Initial Fact: Del has sneezed.
LHS of Rule 1 is satisfied.
New Fact: Del has a cold.
LHS of Rule 2 is satisfied.
New Fact: Del stays in bed.
LHS of Rule 3 is satisfied
Conclusion: Del won't be at work.
End

Figure 2.4: An Example of Forward Chaining

It should now be apparent why the term used to describe this process is forward chaining. Chaining describes the chain of rules that results; forward is used because the process "moves" from an initial state to a final result state.

As one might guess, backward chaining is similar to forward chaining except that the process "moves" in the opposite direction (i.e., from final result to initial state). For this reason, backward chaining can be considered to be a goal-driven search of the knowledge base (Bench-Capon, 1990). The process begins with a final result (or goal). A search is made for a rule whose right hand side or result matches the final result; it is then hypothesized that the facts, necessary for the condition portion of the rule to be satisfied, are true. The process is then repeated using these facts as new results. This continues until there are no more rules that apply to the results. Again, an example will be enlightening.

Once more the knowledge base form **Figure 2.3** will be used. Let's suppose that the result (fact) "Del won't be at work" is known. This is a result; it is the cause that is sought. Scanning the rules reveals that the last rule has a result that matches the given final result. Based on the conditions of this rule, it is hypothesized that "Del stays in bed". Next a rule is sought that has "Del stays in bed" as a result; this would be rule two. The conditions of rule two lead to the inference "Del has a cold". Continuing will lead to the conclusion that "Del won't be at work" because "Del has sneezed". **Figure 2.5** summarizes the backward chaining that took place to reach this conclusion.

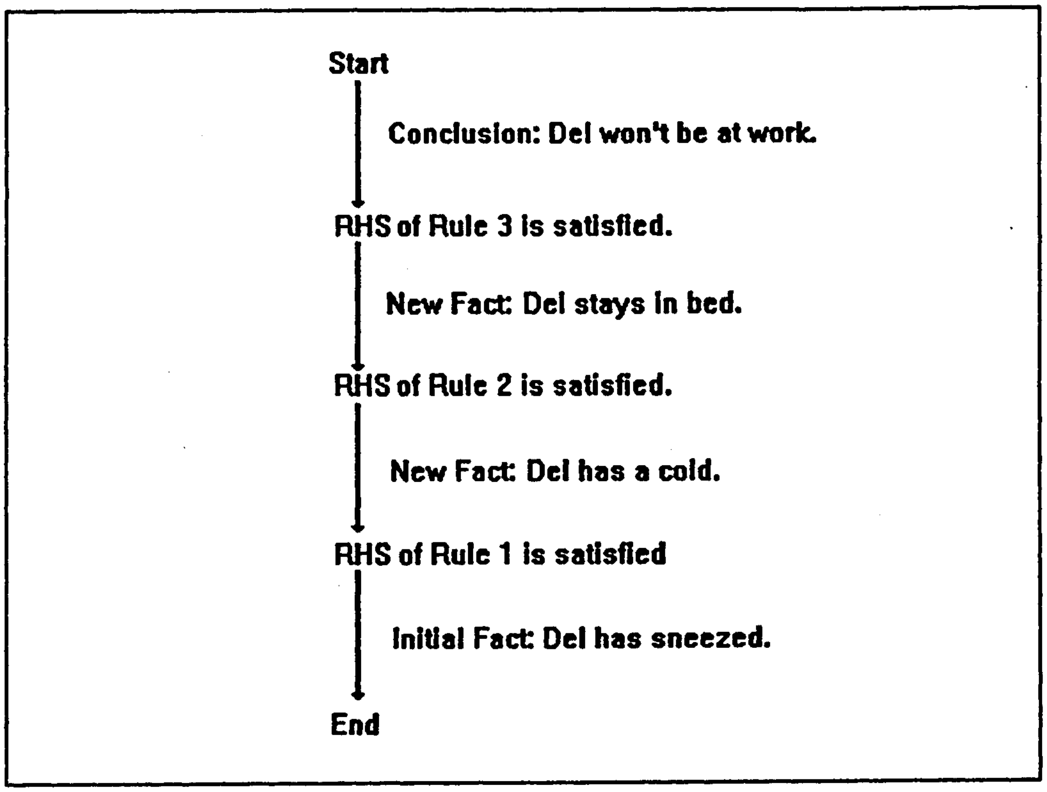
Start
Conclusion: Del won't be at work.
RHS of Rule 3 is satisfied.
New Fact: Del stays in bed.
RHS of Rule 2 is satisfied.
New Fact: Del has a cold.
RHS of Rule 1 is satisfied
Initial Fact: Del has sneezed.
End

Figure 2.5: An Example of Backward Chaining

CHAPTER 3

HUMAN - COMPUTER INTERFACES

3.1 Interfaces

An interface is a means of interaction between two systems. In the case of a human-computer interface, a person and a computer are the two systems. As computers became ever more pervasive in society, the frequencies of interaction between people and computers has increased. This, in turn, has meant that there has been more emphasis on how communication between the two is handled. In this context, the interface is extremely important.

In many cases, the interface determines whether or not people will use a particular computer system. If a computer system is not "user-friendly" (i.e., does not have a good human-computer interface), it is not likely to be used if there is a viable alternative. In fact, this may be especially true of artificial intelligence applications, as users may expect such systems to exhibit human characteristics. From the human perspective, there are two key aspects to be considered in interface design:

1) how information is displayed
2) how information is gathered

This chapter will explore the various issues involved in these two factors.

It is also important to consider what type of user (i.e., level of subject and computer literacy) and what kind of computer system are interacting. These elements can influence the design as well.

3.2 User and Computer Profiles

In the design of a computer system, it is necessary to have some sort of idea as to who the users will be. This is particularly true when developing the human-computer interface. Just as one must keep one's audience in mind when writing a report, one must keep in mind who the intended client is when creating a user interface. In technical terms, this "idea of the intended client" is called the user profile.

Similarly, there is a system profile. This is the way that the users of the system view it. Note, the way that the system is precieved is not necessarily close to reality; however, it may be easier for the client to make use of the system if he or she is able to draw some sort of analogy between what the computer system is doing and some human activity. Perhaps an example will make this more clear.

Consider the automated bank teller (ABT) as an illustration. ABTs allow users to perform a set of standard banking transactions without having to interact with a bank teller and, perhaps, without having to go to a bank. The user profile would be that of a typical bank customer. That is, the system is designed for users who will have banking transactions that will fall within the predefined standard set. As for the system profile, users of the ABTs view the system as working in roughly the same way that a bank teller does. You "tell" the ABT (teller) which transaction to perform and then withdraw or deposit your money accordingly.

In the past, the human-computer interface design has been dominated by the computer, due to hardware and software constraints. Slow processing speeds, primitive output devices, and limited software tools, severely limited the types of interfaces that could be developed. Typically, early interfaces were simple and restrictive (Cleal and Heaton, 1988); the user was restricted to a predetermined set of program states. Movement from one state to another was rigidly defined (Norton and Yao, 1992).

Hardware advances have produced faster processors and rapidly refreshed graphical displays. These advances, coupled with sophisticated software tools, allow the development of more advanced human-computer interfaces. Improving the interface involves more than simply enhancing the visual presentation of the interface. The ideal interface would be one that is tailored for a given user, responding to the needs of that user.

Of course it is not feasible to create a separate, custom interface for every possible user of a computer system. The solution lies in making a single interface that adjusts to different users. One approach would be to devise different interface states to accomodate different classes of users (Cleal and Heaton, 1988). For example, users may be categorized as novice, intermediate, or advanced. The interface would provide a different help system, a different method of gathering information, etc., depending on the level of the user. The problem with this approach is that there is a limited number of user categories. Even if the number of categories were increased, the transition from one class to another would still be an "all-or-nothing" change. There is no accounting for the gradual change in knowledge as users learn more about the system.

It has been proposed (Cleal and Heaton, 1988) that the aforementioned problem could be solved by employing artificial intelligence technology in the interface design. An expert system could be developed that accumulates data about users and acts according to a set of production rules. The development of such an interface is, however, beyond the scope of this paper.

3.3 Display Considerations

For many computer applications, information is displayed via text on a CRT. It should then be a primary concern of the interface designer(s) to present the text in a form that is most readable and, hopefully, most pleasing to the user. After all, if the user is unable to gather needed information from the computer system (especially an expert system), the system is unlikely to be used.

Muter, Latremouille, Treurniet, and Beam (1982) have clearly shown that, with respect to readability, text displayed on a CRT is very different from text displayed on paper. In a study involving two groups of readers, one group reading text from books and the other reading text from a CRT, it was demonstrated that text from a CRT is much more difficult to comprehend. On average, the group reading from the CRT read 28.5% slower. The reason: reduced legibility of CRT text.

There are several factors that contribute to the reduced legibility. One is the length of the text line (Kolers, Duchnicky, and Ferguson, 1981). Experiments have indicated that the length of the line can have an affect on reading speed. For example, a passage of text with forty characters per line is read 17% slower than the same passage with eighty characters per line. In the test conducted by

Muter et al., the CRT displayed thirty-nine characters per line, while the book had sixty characters per line.

Related to the length of the line is size of the text, or size of type. It is customary to measure the size of the type by points, with one point being roughly equal to 1/72 of an inch. The most common type sizes are nine, ten, eleven and twelve points. All these sizes have been found to be equally legible (Hulme, 1984). However, one should keep in mind that the size of type used will depend a great deal on the distance the viewer will be from the display.

Another factor is the method of contrast. Typically, books employ negative contrast. That is, the page (background) is light (usually white) and the text is dark (usually black). The effect is to increase the overall luminance of the reading material. CRT displays, on the other hand, often use positive contrast. The text is bright, and the the background is dark. Studies have shown that people prefer negative contrast (Radl, 1980).

Finally, it is worth noting that the case of the printing is also critical. On many older computer displays, all text was printed in uppercase. The shape of a word is an important clue to the reader (Hulme, 1984), aiding in the recognition of the word. Because upper case printing gives words uniform shape (see **Figure** 3.1), reading uppercase passages is more difficult.

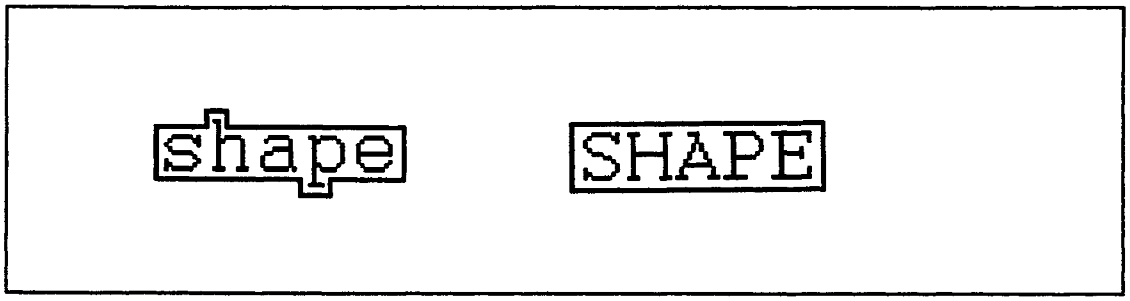

Figure 3.1: Case and Word Shape

The illustration in **Figure 3.1** makes it apparent that lowercase words have a great deal of shape, where as uppercase words appear to be rectangular.

The above has been a discussion of the physical form of displayed information. It is also important to pay attention to the methods used to represent the information. Specifically, the layout of the display can greatly influence a user's impression of the human-computer interface. There are several rules of thumb that should be employed when planning the layout of the display (Reid, 1984). These are summarized in **Table 3.1**.

One should particularly note that it is unwise to display too much information on the screen. It is important to keep this in mind because there is a strong tendancy to feel that the screen display should be filled with data, much as a text book would be. However, such displays will overwhelm users. Display only the information that is needed at any given time; keeping the screen display simple will make it easier for users to see what is truly important.

Some of the other rules deal with attracting a users attention. Although it has been discovered that the upper-right quarter of the screen is best place to display messages, many commercial software packages choose to use other areas of the screen. In some cases, it is known that the user will be focused on a particular part of the screen when the message will be displayed. In such an instance, it is best to present the message in this area of focus. Another factor to consider is the type of user that will be operating the software. For example, let us compare users who are skilled typists versus users who are two-finger typists. When the former are operating a computer terminal, they will likely be focused on the screen, where as the latter, will likely be concentrating on the keyboard, as they hunt down the next key to be pressed. Therefore, in the case of the expert typist, important messages should be placed in the centre of the display. On the other hand, messages should positioned at the bottom of the screen for two-fingered typists, since this is the portion of the display that is visible when one is focused on the keyboard.

Another way to draw the notice of a user is to present a message in blinking text. However, care should be taken not to employ this method too often. If the screen is filled with

Table 3.1: Summary of Display Layout Rules

Rule	Description
Do not overfill the screen	Do not use more than 25% of the display area. Beyond this limit, readers are unable to pick out information easily.
Use the upper-right quadrant for exception reporting	When display exceptional information (e.g., error messages, warnings, notices) the upper-right quarter of the screen. Viewers are most sensitive to changes in this area of the display. Of course, this assumes that the viewer is not focused on any particular part of the display.
Use mixed case words	As discussed earlier, readers can more easily recognize lower case words.
Allow data to flow naturally	The layout should be designed so that the user's eyes fall naturally on the next item of information.
Use blinking for important messages	Flashing messages will draw the readers attention.

flashing messages, the user is apt to become desensitized to this technique. It is important to note that these rules only refer to the visual display. One should remember that there are other ways to notify the viewer of an important event. Specifically, one can use an audible cue, such as a beep.

3.4 Window Displays

One method of creating a better display, and, therefore, better human-computer interface, is to divide the screen into windows. Dividing the screen has several advantages. For instance, each window could be used to represent a distinct task that a user is expected to complete; the operator simply moves to a specific window when she wishes to tackle a particular chore. This provides a level of organization to the display. Another example would be to employ overlapping windows as a way to preserve a history of what has already been accomplished. An operator could jump to a previous state by moving to that window.

A windowed display also helps in addressing some of the issues brought up by the rules found in **Table 3.1**. If the windows are resizable (i.e., the windows can be enlarged or reduced), then a user, who feels overwhelmed by the amount of information displayed on the screen, can reduce those windows that he feels are unnecessary at the moment.

If the windows are moveable (i.e., the windows can be relocated on the screen), users may reposition any window that is currently being used to a position on the screen that allows them to notice any messages that may be displayed during operation. This can prevent important information from going unnoticed. Returning to the earlier example of the users of two different typing skill levels, the expert typist may chose to position an active window in the centre of the screen. The two-fingered typist may be best served by positioning the active window near the bottom of the screen.

Before the advent of operating systems, such as MicroSoft Windows (MS Windows), which support graphical user interfaces (GUIs), providing a windowed interface required a great deal of effort on the part of software developers. MicroSoft Windows makes the task easier, however, by providing many functions for implementing windows. In addition, the resulting interface, in many cases, surpasses what many programmers are capable of, or have the time to, produce without the MS Windows functions. Besides the window, MS Windows provides many other interface elements. These will be discussed in further detail in later chapters, but one of these is the menu.

3.5 Menu Driven Systems

Menus provide a way for users to communicate with the computer system by selecting an item from a list of available options, which are meaningful in the context of the current state of the system. For example, the system may be in a state that requires the user to answer the question, "What is the sex of the patient?". A menu offering the choices, "(1) Male (2) Female", can make it simpler for the user to communicate to the system the correct response.

While menus are a valuable interface element, they are not appropriate in all situations. Knowing when to employ them is part of the skill of system design. A menu driven system can benefit occasional or novice users by limiting the amount of information to remember. Furthermore, because the number of responses are limited, it prevents nonsense responses from novice level users. On the other hand, the limited nature of menu response prevents their use in situations where all possible responses can not be anticipated. Advanced users may become frustrated with a menu driven system, finding that it takes too long to reach a particular option or that it is too awkward to navigate around the system (Reid, 1984). The latter is especially true of very complex systems.

Through careful design, it is possible to eleviate some of the navigation problems. A "Where Am I" facility, which will inform the user as to the current state of the system, is helpful. Coupled with this can be a "backup" and "return to start" function, that allows the operator to go back one level or return to the top level. Ideally, a map of the entire system could be presented, and the user would simply point to the system state that she wishes to go to (although this goes beyond menuing).

Special attention should be paid to the content and ordering of the menu items. It is imperative that these be logical to the end user. If this is done, the user will have an intuitive idea as to what he or she should do as the operator. Also, when there will be a processing delay when a selection is made, confirming the selection visually will allow the user to know that the selection was made correctly and that the delay is simply part of program execution.

CHAPTER 4

INTERPRETERS

4.1: Introduction

There are two methods to achieve execution of programs written using a high level language. One way is to compile the source code using a compiler. The compiler software converts the source code into equivalent machine code, which can be executed by the computer. The alternative method is to use an interpreter.

A computer language interpreter behaves much like a human language interpreter. A human interpreter listens to (or reads) each statement (or partial statement) and translates it. If the statement is repeated, the translation process is repeated. Likewise, a computer language interpreter analyses a source code statement and simulates the functions specified therein (i.e., evaluates the statement). If a statement is repeated, it is interpreted each time it is encountered (Parker, 1989). **Figure 4.1** shows a fragment of BASIC source code. Consider the statement:

```
LET S = S + I
```

This statement will be interpreted with each repetition of the **FOR..NEXT** loop.

```
     .
     .
     .
     100  INPUT N
     110  LET S = 0
     120  FOR I = 1 TO N
     130  LET S = S + I
     140  NEXT I
     .
     .
     .
```

Statement	Description
100	Accept an integer input from the user.
110	Store a value of zero in the variable, S.
120	Set the counter I to one.
130	Assign S the value of S plus I.
140	Add one to I; if I is less than N, go to statement 140.

Figure 4.1: A Fragment of BASIC Code

The analysis and evaluation performed by the interpreter can be decomposed into four distinct processes (Zarrella, 1982):

(1) determine the statement type
(2) decompose the statement into operators and operands
(3) search the symbol table for the address of any variable operands
(4) call subroutines to process the operators

The systematic nature of the analysis and evaluation lends itself well to automation. The list closely resembles the type of processing that a person goes through when she reads an algorithm and simulates the computations.

Although they generally result in program execution speeds that are slower than those achieved by compiled programs, interpreters are usually considered to be "friendlier" than compilers. This is because problems, both in development and execution, are reported immediately.

4.2 Types of Interpreters

A pure interpreter is one in which the source text is stored exactly as entered by the program developer. While this makes it easy to display the source in the way it was inscribed, it has some major disadvantages. For one, memory is needed to store blanks that are embedded in the text. Because of the styles employed in writing source code, this may require a large portion of available storage. In addition, the repeated scanning and analyzing of the source code can be very time consuming. Because of these issues, pure interpreters are seldom used.

To overcome the disadvantages of pure interpreters, hybrid interpreters have been developed. Such interpreters alter the original source text. For example, extra blanks are removed, operators are represented by numeric codes, and a symbol table is utilised to keep track of variables and statement labels. The consequence of all this is that the original source code cannot be regenerated from the altered version (Zarrella, 1982).

4.3 The Abstract Machine

Execution speed and memory requirements can be further improved by implementing an ideal, abstract machine (Zarrella, 1982). The abstract machine has an instruction set that closely resembles the source code, yet can be easily implemented on existing systems. The source code is converted to the abstract machine code, also known as intermediate code or pseudocode. Because the source code closely resembles the intermediate code, the conversion is a rapid process. The instruction set of the intermediate code must be designed carefully, taking into account the following requirements (Zarrella, 1982):

1) Conciseness: it must be much more succinct than the source code.

2) Easily simulated: the host machine must be able to easily simulate the intermediate instruction set; a complex simulator requires a larger interpreter and more memory.

3) Complexity: it must handle complex data and control structures, built-in functions, etc.

4) Speed: execution speed must be reasonable.

4.4 Interactive Interpreters

An interactive interpreter is one in which there is a continuous exchange of information between the interpreter and the user of the interpreter. Because this interaction aids in the development process, most interpreters fall into this category. For example, when one enters a line of source code, it is checked for syntax immediately, and any errors are reported. In addition, many interpreters possess an integral debugger. When a run-time error occurs, the debugger reports the location and type of error to the programmer.

Although they serve as powerful development tools, interpreters do have two major disadvantages, when compared to compilers. For one, the execution speed of interpreted applications is slower than that of compiled applications; a compiler would be more appropriate, then, for applications for which execution speed is critical. Another drawback for interpreters is that any program written for an interpreter, requires the interpreter software to be present for its execution. This means that distribution of the program requires distribution of the interpreter.

CHAPTER 5

MICROSOFT WINDOWS

5.1 MicroSoft Windows Introduction

The MicroSoft Windows (Windows) operating system is an extension of the MicroSoft DOS operating system. Moreover, it is a graphical extension of MS DOS. The extensive use of graphics to represent operating system and program commands is intended to make Windows easier for end users to perform operating system commands and to use application programs (MicroSoft, 1992).

Windows extends DOS in three ways (Norton and Yao, 1992); it provides:

1) the ability to run multiple programs at the same time.
2) high level graphics
3) a standard user interface

In addition, Windows is designed to respond to user-initiated events. All these factors contribute to the creation of applications that are optimized for user interaction. Therefore, applications that involve a great deal of user interaction benefit the most from the Windows environment.

5.2 Benefits of Windows

Windows provides benefits for two distinct groups: users and developers. For users, Windows supplies:

1) a standard interface
2) inter-application communications
3) multitasking
4) no need to set up devices or drivers
5) access to more memory

Among these, the first three are of particular importance to the average user. A standard interface means that all Windows applications "look and feel" the same. This, in turn, means that once a user has learnt to use the interface of one Windows application, she has learnt to use the interface of all Windows applications. This is far superior to the situation that exists with MS DOS, where users are often required to learn a new interface for each application they may wish to use.

Often, users will want to transfer the results of one program to another program. For example, one may wish to bring a graph created by a spreadsheet program into a word-processor document. Windows support of inter-application communication makes this an easier task. Screen data of any type can be transferred from one application to another by use of a buffer called the Clipboard. In addition, applications can communicate directly with one another by passing messages. These messages can contain data items.

Because PC computers have a single CPU, it is not possible to have true concurrency. However, Windows provides a method for multitasking. Many operating systems provide multitasking by dividing the CPU time into slices. Each application is processed only during its time slice; since the time slices are very short, users have the illusion of concurrency. The Windows approach is somewhat different, though. While it is true that the CPU time is divided up into time slices, applications only use their time slice if there is "something to do". In most cases "something to do" means that there is a message to respond to. If there is no such message, then the application surrenders its time slice.

For developers, Windows provides the following benefits (Borland International, 1991):

1) device independent graphics
2) support for a wide range of devices
3) a library of graphics routines
4) support of interface objects (menus, icons, bitmaps, etc.)

The Graphics Device Interface (GDI) is central to the concept of device independent graphics. The GDI provides a standard set of functions that can be used to create graphics displays. The programmer need not be concerned as to which device the display is being sent to. The GDI determines which device it is and takes care of the details. From the programmers point of view, the function calls are the same, regardless of the device. In addition, Windows readily supports a great number of devices. These devices fall into four broad categories:

1) display screen
2) hard-copy (e.g., laser printer)
3) bitmaps
4) metafiles

The latter two are actually pseudo-devices, which provide a method of storing graphic images in RAM (or on disk) and a method of sharing such images between applications.

The built-in graphics library functions furnish developers with routines for creating geometric figures for display purposes. This eliminates the need to develop the actual code to do the drawing. Related to this is the built-in support for many user-interface objects. Such objects are used to communicate with users. Again, time is saved because the code to implement such objects is already written.

5.3 Requirements

The aforementioned benefits are not without cost. The cost can be divided into two categories; there is a hardware cost and a software development cost. From the hardware standpoint, Windows requires "bigger" and faster computers. Bigger means more RAM, and faster means a more sophisticated CPU with a higher clock speed than is required for MS DOS. According to the Windows manual (Microsoft Corporation, 1992), version 3.1 of Windows requires at least 1Mb of RAM and a

80286 processor or better. Although not required, it is highly recommended that one use a VGA colour monitor and a mouse, as well. It has been the experience of this author, however, that a 80386 with 4Mb RAM is a more realistic minimum system.

In the area of software development, Windows programmers are required to take a new approach to application programming. Unlike most DOS applications, which are written in a sequential, procedure-driven manner, Windows applications are generally **message driven.**

Sequential, procedure-driven programs (also called modal programs) have a distinct beginning, middle, and end; the program is composed of distinct modes. A mode is a program state in which user actions are interpreted in a certain way and produce a specific result (Norton and Yao, 1992). The chief drawback to the modal programming approach is that users are limited in the way that they can move from one mode to another. The flow of control is precisely defined by the programmer. Also, the user is often required to remember which mode he is in, without being supplied with good visual clues. It should be noted, however, that modal programs are usually easier to write, since each mode can be written and tested separately.

In the context of message driven programming, a message signals an event that may or may not need to be responded to. A Windows message is an integer, which represents some change in the user interface. It is up to the developer to decide which messages to respond to and which messages to ignore. The flow of program control is governed by the actions of the user. The advantages of this approach to application programming is that it is easy for users to change program state, and the user is given good visual clues as to which state the program is in at any given point in time. For the developer, though, message driven programs are generally more work. The programmer must first identify all messages that should be responded to and then the code to execute the response must be developed.

In addition to the complexities of writing message-driven applications, Windows software developers must become accustomed to Windows method of output. All output from a Windows application is graphical, and all graphics are high-level graphics. This contrasts with the character-based output of most DOS applications. The emphasis on high-level graphical output means that it is quite easy to produce geometric figures. This is particularly true of the graphics that Windows supports with callable subroutines. However, because it is treated as a graphical object, text is more difficult to produce in Windows than in DOS. Instead of positioning text on the screen by character cell references, text must be positioned by pixel reference.

Finally, software developers must learn to use the user-interface objects employed by Windows applications. The most important interface objects are:

1) Window: provides a view into the application
2) Menu: provides a selection mechanism for users
3) Dialog Box: allows users to enter complex data elements
4) Cursor: indicates where in the window the user is currently pointing
5) Scroll Bar: allows users to scroll through a graphic display
6) Icon: provides a visual representation of some command, program, or data

Each of these interface objects is supported by Windows. This means that it is not necessary for programmers to write the code to implement them, as this has already been done. Instead, the challenge to developers is that they must understand how each is used.

CHAPTER 6

OBJECT ORIENTED PROGRAMMING

6.1 The Evolution of Object Oriented Programming Languages

Before beginning a discussion of object-oriented programming, it is necessary to first review the history of the development of such languages (i.e., the motivation for their conception). Originally, the justification for inventing programming languages was to remove many of the tedious details of programming at the machine level. That is, in order to save on development time, the ease with which a program could be written was increased and the probability of making an error was decreased by automating many of the tasks that had to be performed manually during machine level programming. It was this line of thinking that led to the invention of pseudocode (MacLennan, 1987).

Of course, the first real high-level language was FORTRAN, but the motivation for its development was essentially the same as that of the aforementioned pseudocode. However, FORTRAN went a step further, in that there was a conscious effort to allow programmers to be able to represent arithmetic equations in a format similar to that used by mathematicians. It is also worth noting that there was a great effort put forward to produce a compiler (pseudocode was executed by an interpreter) that would produce fast code; to this end FORTRAN was successful. In fact, FORTRAN represents an incredible first step, but it hardly encourages the creation of clear and easily understood programs (Schildt, 1991).

As the complexity of the problems that computer programmers were required to solve increased, it became apparent that first-generation languages, such as FORTRAN, were not sufficient. Eventually, new programming paradigms were proposed to fill the gap. One of these was the structured programming method. This method took the best of previous models and built on them. The resulting approach to programming allowed programmers to tackle even more complex problems successfully. Unfortunately, as had happened in the past (and will very likely, continue to happen in the future), the size and complexity of programming projects continued to grow, and many of the modern day programming tasks are now approaching a level of complexity that can no longer be handled sufficiently by the structured programming approach. Once again, there is a call for a novel programming paradigm.

One possible solution that has been put forward is the object-oriented approach to programming. In essence, this methodology borrows the best from structured programming, but makes two major additions, encapsulation and inheritance (Snyder, 1986).

6.2 Encapsulation:

Encapsulation is the binding into a single unit of the data and the code pertaining to that data. This unit is called an object. The code forms what are called methods, which are functions or procedures bound to the object. Some, if not all of the code, serves as an external interface by which external code is able to access the data within the object. In other words, the only way that the data within the object can be "seen" is through the external interface, at least in theory. External code makes use of this interface by sending messages to the object. The object "reads" these messages and takes appropriate action. Of course, code within the object can manipulate the data directly, without having to go through a message system. In practice, many object oriented programming languages (e.g., TurboPascal for Windows) allow data within an object to be accessed directly. When using such languages, one should resist the temptation to access object data directly. One creates an object by first creating a class of that object type. Classes are object types, and objects are instances of classes (Wegner, 1987). Therefore, classes merely respond to calls for instantiation of objects. For this reason, the data declarations within a class are often referred to as instance variables, as the data only comes into existence when an object is instantiated. **Figure 6.1** shows an example of a class declaration of an object in object-oriented Pascal, as implemented by TurboPascal for Windows. The class (or object type) is called Point. It consists of two data elements, X and Y, and three procedural elements, Init, SetXY, and GetXY. The Init procedure is a special type of procedure called a constructor. A constructor is called automatically when an object is instantiated.

```
Type Point = object
               x: integer;
               y: integer;

               constructor Init(InitX, InitY: integer);
               procedure SetXY(SomeX, SomeY: integer);
               procedure GetXY(var SomeX, SomeY: integer);
```

Figure 6.1: An Object Declaration

The encapsulation of code by object-oriented languages exemplifies three of the principles of programming languages (MacLennan, 1987). First, code that relates to a single abstract data type can be abstracted out and kept within a single object. This is the Abstraction Principle in action. Also, the fact that code pertaining only to the access and manipulation of a single abstract data type is kept hidden within an object, as well as the fact that the data itself is hidden within an object, is an example of the Information Hiding Principle. Finally, the third principle obeyed by encapsulation is the Manifest Interface Principle, which states that all interfaces should be apparent in the language syntax. Well, in the case of objects, the interface is just the set of messages that it responds to.

Of course, it would mean very little to obey the aforementioned axioms if there were not some sort of payoff. Well, there is. Abstracting data and hiding the implementation details of the abstracted data type means that coupling (interdependencies) among separately written modules is greatly reduced (Snyder, 1986). Coupling is a measure of how tightly one module is related to or dependent on another; if the coupling is high, then changes in one module will affect the other module. Obviously this is undesirable, since it will mean more time is required for program maintenance. In many situations, the majority of the time spent on a program is spent maintaining it (Dr. Franek, personal communication). Therefore, steps should be taken to reduce maintenance requirements. The object-oriented approach, with its encapsulation, presents an effective method for reducing interdependencies; all would be fine if not for the added feature of inheritance.

6.3 Inheritance:

Inheritance is a means for the sharing of data and methods (code) between different classes of objects (Wegner, 1987). There are two types of inheritance, single inheritance and multiple inheritance. In the case of single inheritance, a subclass inherits from only one immediate parent or base class. The base class may itself inherit from another base class, and so on. By contrast, multiple inheritance involves inheriting from more than one base class.

Earlier, it was implied that inheritance complicates the object-oriented paradigm. It does so by introducing a new type of customer for classes. As described above, classes responded to commands invoking the instantiation of an object; with the addition of inheritance, classes must also respond to commands calling for inheritance of class features (Snyder, 1986). The bottom line is that a second interface is needed for communication between a parent class and its child classes (subclasses). (Recall that the first interface was between a class and anything outside that class.)

While it is true that inheritance complicates object-oriented languages, this is counter-balanced by the fact that inheritance reduces software development effort by allowing developers to reuse code that has already been written, by having classes inherit common code from other classes. In this way redundant coding is eliminated. For example, in the development of the Windows application of Preop, a specialized stack called a conclusion stack (see Chapter 9 for more details) was needed. A conclusion stack is a type of stack and, therefore, requires all the functions that a simple stack requires. As this author had already created a "stack class", none of the code required for the basic stack functions had to be rewritten. The "conclusion stack class" simply inherited from the "stack class".

6.4 TurboPascal for Windows with ObjectWindows

Obviously developing Windows applications puts new demands on programmers. TurboPascal for Windows (Borland, 1991) is an object oriented version of Pascal, with a library of objects developed to aid programmers in facing these new challenges. The library is suitably named ObjectWindows.

ObjectWindows serves to provide a better application interface (API) to Windows than is available with Windows alone. Interface elements (menus, icons, etc.) are represented by objects. Such objects contain Windows information and provide for the abstraction of Windows functions. By encapsulating the API in objects, many of the details of the Windows interface are removed from the programmer. It is important to note, however, that these library objects only define the behaviour, attributes, and data storage of the various interface elements; the physical implementation (i.e., the actual screen appearance) is handled by Windows. A library object is logically linked to the actual interface element by an integer handle.

The Windows API consists of approximately 600 different functions. Each of these functions requires many parameters of many different types. Keeping track of all these functions and parameters can be a daunting task. ObjectWindows simplifies this situation. As stated earlier, the objects in the ObjectWindows library store Windows information. Therefore, many of the values needed for function parameters are stored in objects, and are passed to the appropriate functions automatically. Also, related functions are grouped together (abstraction) into a single method, requiring developers to remember less.

A typical Windows application must deal with hundreds of messages. ObjectWindows makes the processing of these messages easier by providing a mechanism for automating message response. Instead of having to write code to process the various messages, it is only necessary to produce the code needed to respond to a given message. The responding code will be automatically invoked when the message is received.

In summary, ObjectWindows aids developers of Windows applications by:

1) providing a simplified interface through use of object oriented programming principles
2) simplifying function calls to the Windows API
3) automating message response

CHAPTER 7

NEXPARSE

7.1 Requirements

NexpertObject is a tool for building and running expert system applications. While this product is good for prototyping such applications, the presentation of data and the method of acquisition of data is not satisfactory. With the NexpertObject run-time user interface (called NORT), the presentation is limited to text mode displays; however, what is more problematic is the linear approach to data acquisition. Once a value has been entered, the user cannot go back and change it.

In addition, NexpertObject does not produce an executable file as its end product, but rather, creates a knowledge base file, which is an ASCII file containing the information needed to run the application, within NexpertObject. Therefore, each user of the expert system would also require NexpertObject. This situation is far from satisfactory.

The goal of this project, therefore, is to develop a set of tools that will allow one to take the knowledge base file, containing all the information needed for NexpertObject to run an expert system, and produce a stand-alone, executable file, that will be the final expert system product. Moreover, this final product will be a MicroSoft Windows application, providing users with an easy-to-use interface.

There are two approaches that can be taken in creating a stand-alone expert system. Either a system, that compiles the knowledge base file into an executable file, can be implemented, or a system can be built that will interpret the knowledge base file. In this project, the latter option was chosen, and NEXPARSE is the first step in achieving this goal. Briefly, NEXPARSE is the first component of a three-component hybrid interpreter. It takes as its input a knowledge base file and produces several files, rearranging and categorizing the information contained in the input file. However, before going into this further, it is necessary to explain some of the terminology used.

7.2 The Knowledge Base File

A knowledge base file consists of collections of some or all of the following: PROPERTIES, CLASSES, OBJECTS, META-SLOTS, RULES, and GLOBALS. CLASSES are the facility by which users represent real world entities. That is, CLASSES allow users to describe real world entities by listing the characteristics of these entities. These characteristics are listed as PROPERTIES. A PROPERTY consists of a name, which is made up of alphanumeric characters and underscores, but no spaces, and a type, which is either Integer, Float, Boolean, String, Date, or Time. Within the CLASS, only the PROPERTY name is listed. The name is bound to a type in a separate PROPERTY declaration. Additionally, CLASSES can contain subclasses. The use of subclasses allows CLASSES to inherit the PROPERTIES of an already declared CLASS, without having to relist all those PROPERTIES; instead, the name of the previously defined CLASS, which is syntactically similar to a PROPERTY name, is listed as a subclass of the new class.

While CLASSES are intended to describe (or represent) real world entities, OBJECTS serve to represent actual instances of these real world entities. For this reason, OBJECTS are often said to be instances of CLASSES. For example, one may define a CLASS, AUTOMOBILE, which describes automobiles. AUTOMOBILE would then list the PROPERTIES of automobiles in general. Now, the OBJECTS, MY_CAR and YOUR_CAR would be instances of the CLASS, AUTOMOBILE, and would represent actual real world entities. They would possess the same list of attributes that AUTOMOBILE does, but they would each have values for these PROPERTIES, corresponding to the values of the real world entities they represent.

Every OBJECT has a name, which is similar to a PROPERTY name. An OBJECT may also contain a list of CLASSES to which it belongs. If an OBJECT belongs to a CLASS, it is an instance of that CLASS and inherits all of the PROPERTIES of that CLASS. With NexpertObject, it is possible for an OBJECT to inherit from more than one CLASS (multiple inheritance). Also, an OBJECT can have a list of associated PROPERTIES. These PROPERTIES are in addition to PROPERTIES inherited from declared CLASSES and are defined in the same way as PROPERTIES of CLASSES. As a special case, it is also possible to associate one elementary data value directly with the OBJECT itself. To do this, one must define a special PROPERTY with the reserved name, VALUE.

Closely linked to both CLASSES and OBJECTS is the notion of a META-SLOT. META-SLOTS are used to define attributes of PROPERTIES (i.e., properties of PROPERTIES), that are associated with CLASSES or OBJECTS. These attributes determine the behaviour of the PROPERTY in its interaction with the rest of the system. For example, using META-SLOTS, a user of NexpertObject can determine how the value of a PROPERTY is determined, or how a PROPERTY is inherited from CLASSES to subclasses.

While CLASSES and OBJECTS (and their composite PROPERTIES) are used to describe real world entities, the expert knowledge is stored in RULES. Each RULE consists of one or more conditions, exactly one hypothesis, and zero or more actions. If all the conditions of a RULE are found to be true, the hypothesis, or conclusion, associated with the RULE is also true; if there are any actions associated with this RULE, then each of these actions is performed, only if all the conditions have been satisfied. This describes the components of a RULE. The execution of an expert system involves the evaluation of the appropriate RULE at the appropriate time. A RULE is evaluated either when its hypothesis is suggested as a goal to be investigated, or when one of its conditions is proposed to be true. The former is called backward chaining, and the latter is referred to as forward chaining.

Finally, GLOBALS are variables which dictate how the system will run. A GLOBAL consists of a global variable name and a boolean value (true or false) indicating the state of the variable.

7.3 Program Description

As previously mentioned, a knowledge base file contains instances of some or all of the aforementioned terms. For a precise specification of the knowledge base file, see "The Knowledge Base File" in Appendix A.

NEXPARSE produces six file types from its single input file. They are (1) PROPERTY files (*.prp), which are listings of PROPERTY names and their types; (2) CLASS files (*.cls), which are lists of CLASSES; (3) OBJECT files (*.obt), which are listings of OBJECTS; (4) META-SLOT files (*.slt), which contain lists of META-SLOT data; (5) RULE files (*.rul), which are listings of RULES; (6) GLOBAL files (*.gbl), which are lists of global variable names and their boolean values. Breaking the one data file into several subfiles makes managing the data easier, but the subfiles serve an additional purpose.

The data in the knowledge base file, as described above, is embedded among unnecessary information and is encoded in a less than useful format. For example, **Figure 7.1** shows a typical PROPERTY declaration.

```
(@PROPERTY=     age     @TYPE=Integer;)
```

Figure 7.1: A Typical PROPERTY Declaration

If a file contains only PROPERTIES, then it is not necessary to state that it is a PROPERTY that is being declared (@PROPERTY=). Also, since all PROPERTIES must have a type, it is wasteful to identify the type with @TYPE; one can determine it is a type by its position. Another problem with the declaration in **Figure 7.1** is that in order to determine the PROPERTY's name and type, one must parse this information from the line. **Figure 7.2** shows a file format that addresses all these issues.

```
age
Integer
```

Figure 7.2: A New PROPERTY Format

This simplistic approach makes use of position (within the file) to identify the PROPERTY components. The PROPERTY name always precedes the type. Since all PROPERTIES must have these two items, every pair of lines, within the file, comprises a PROPERTY definition. Also, note that extracting the information is quite easy, as one must simply read a line to obtain the name or type.

NEXPARSE works by first reading the knowledge base file one line (up to 80 characters) at a time. Each line is then searched for the keywords: @PROPERTY, @CLASS, @OBJECT, @SLOT, @RULE, and @GLOBAL. Lists, containing records for holding data related to each of the keywords, are maintained. When one of the keywords is encountered, a subroutine for handling that particular keyword is called, and the subroutine creates a new element of the appropriate list and extracts, from the input file, the necessary information. For example, when "@PROPERTY" is encountered, the PROPERTY name and type are extracted from the file and stored in a record. Upon reaching the end of the of the input file, the data, in the lists, are written to the appropriate output files, in a format consistent with the principles discussed earlier.

CHAPTER 8

SGROUP

8.1 Requirements

NexParse is only one step towards the final goal of an expert system independent of NexpertObject. Recall that one of the main goals of this project is to provide a system that allows physicians to enter the patient information and then allows them to go back at any point and modify it. It was decided that the best way to facilitate this in a Windows application would be to create a pull-down menu **Figure 8.1** provides an example of one such pull-down menu. The menu contains a list of "information categories". When a category is selected, the physician is presented with questions pertaining to that category. One can return to a category and alter the answers at any point, simply be re-selecting it.

The menu seen in **Figure 8.1** is an example of a Windows <u>resource</u>. A resource is a data file containing information descripting one or more of the MS Windows graphical elements (Borland International, 1991). These include:

1. bitmaps
2. cursors
3. dialog boxes
4. icons
5. keyboard accelerators
6. menus

The resource defines the visual aspects of these elements, not their functionality. For example, a menu resource would describe the screen appearance of the menu (height, width, contents) but would not describe what to do when a given menu option was selected.

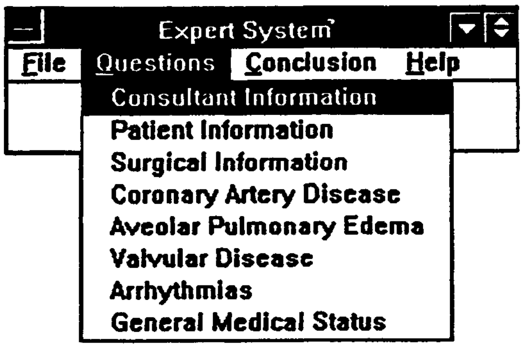

Figure 8.1: A PullDown Menu

Resources are stored externally of the executable files that use them. This means that they must be loaded at run-time. This has two great advantages. For one, a single resource can be used by many programs, introducing an element of reusability. Also, a resource can be altered without affecting the executable program that uses it. This makes applications using resources more maintainable.

There are two considerations when <u>creating</u> and <u>using</u> a menu resource. First, in order to create the pull-down menu, one must have a list of the menu options. For this project, the list is a list of "information categories". Since NexpertObject does not have a menu system, there is no equivalent list stored in the knowledge base file. Therefore, there is a problem in obtaining this list.

Also, once the pull-down menu has been constructed, the application program must be coded to respond to menu selections. It is possible to write routines to handle each of the menu selections specifically, but this would mean that any change in the content of the menu resource would require a similar change in the application program. Obviously, this eliminates the advantage of easier maintenance that was gained by making use of a Windows resource in the first place.

The SGroup utility was developed to address these issues. Written in the <u>TurboPascal for Windows</u> language (Borland International, 1991), it also provided an introduction to MicroSoft Windows programming concepts.

SGroup provides the expert system designer with a mechanism for entering the "information categories" that make up the menu selections. Each "information category" represents a group. Each group, in turn, contains a list of questions to be asked when the menu option it represents has been selected. The designer specifies which questions belong in a group by selecting them from a list derived for the NexpertObject META-SLOTS.

Recall that a META-SLOT is a NexpertObject entity that functions to define attributes of a PROPERTY (see chapter seven for more details). One of the attributes of a PROPERTY can be how its value is determined. In cases where the value of a PROPERTY is entered by the user of the expert system, NexpertObject defines a META-SLOT containing a PROMPT element. The PROMPT is an alphanumeric string representing the question that is to be directed at the user in order to retrieve the value for the PROPERTY. The list of META-SLOTS presented by SGroup is comprised of those that have a PROMPT element.

As output, SGroup produces an ASCII file containing one or more group names ("information categories"). Each group name is followed by a list of all the META-SLOTS that are to be included in that group. (Note: each META-SLOT has a unique label, that is the same as the PROPERTY to which it is associated). Because there is a data file with this information in it, it is possible to develop an algorithm to handle a general menu selection and use the data file to fill in the specifics at run-time. If the contents of the menu change, only the data file needs to be altered; the executable program that uses it remains unchanged. Chapter nine will examine the general menu handling algorithm in more detail.

8.2 Program Description

Unlike NexParse, SGroup is a MS Windows application. As such, it needs to be able to manipulate Windows' interface objects and respond to Windows' messages. A large part of the complexity of dealing with these issues is alleviated by using the library of Windows objects that comes with TurboPascal for Windows (TPW). The following discussion highlights the most important constructs of the SGroup program and how it makes use of the TPW Windows objects.

Like all interactive Windows applications, SGroup needs to be able to display and manipulate Windows' interface objects. Foremost among these are the windows. A window defines the display space for a given application (see **Figure 8.2**). All windows have some common functionality (Resize, Close, Move, etc.). The TPW object, TWindow, provides methods that implement these basic window functions. If one looks at the SGROUP.PAS code in Appendix A, one will notice that the TSGroupWin object defines the main window for SGroup. This object inherits its basic window functions from TWindow and then defines methods specific for this application. The ability to inherit the methods of TWindow means that programmers do not have to re-invent the basic window with each new application program.

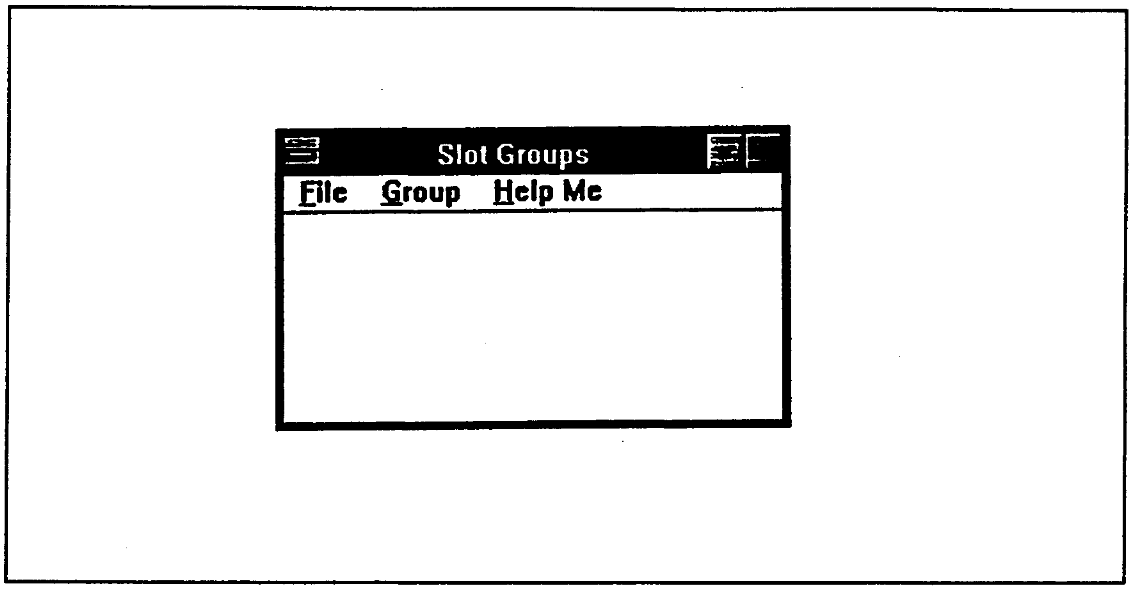

Figure 8.2: A Typical Window

As discussed in an earlier chapter, the MS Windows environment is an event-driven environment. Therefore, programs that are developed to operate interactively in this environment must respond to the events that are happening. Events are related to an application program through messages. Programs will ignore all messages except those that they have been coded to act upon. **Figure 8.3** shows the declaration of a method that handles the selection of the "Help" option from the menu bar of SGroup. The message sent when the selections in made has the identifier: **cm_First + cm_Help**. The method is linked to the message by putting the message identifier after the method declaration. The result is that when the "Help" selection is made, the Help method is invoked.

```
     procedure Help(var Msg: TMessage);
          virtual cm_First + cm_Help;
```

Figure 8.3: A Method Responding to a Message

In order to facilitate the creation of groups and their corresponding list of META-SLOTS, the SGroup application implements functions for <u>creating</u> and <u>deleting</u> groups and for <u>adding</u> and <u>removing</u> META-SLOTS to and from the groups. Each of these functions is implemented as a pull-down menu option of the Groups main menu option. Selecting any of these options invokes methods that allow the user to make selections from list boxes. A list box is an interface object that presents a window containing a list of elements. In the case of SGroup, these elements are groups and META-SLOTS. The list boxes are implemented via the TPW object, <u>TListBox</u>. The use of list boxes provides a very visual (and intuitive) way for the expert system designer to make selections regarding groups.

Once all groups have been formed, the designer should create the output file by selecting the "Save" option. This will cause the group name, followed by the name of each META-SLOT associated with it, to be written to an ASCII file, with the extension, ".SGP". All names are separated by carriage returns, and groups are further separated from one another by a carriage return; the order of output is the order of appearance in the list boxes. **Figure 8.4** displays part of an SGroup output file. "Consultant Information" and "Patient Information" are the groups. The other labels represent META-SLOTS.

```
Consultant Information
consultant.name
consultant.referring_physician
consultant.service

Patient Information
drug
patient.age
patient.id
patient.name
patient.sex
```

Figure 8.4: A Partial Output File

CHAPTER 9

NEXMACH

9.1: Requirements

The last two chapters were dedicated to the explanation of two utility programs that are distinct steps toward the overall goal of this project. NexParse parses a NexpertObject knowledge base file into several files, which reorganize the expert system information into a format that is easier to use. The SGroup utility provides a mechanism for grouping the questions, that are to be posed to the expert system user, into distinct categories. These categories will be menu options in the final product.

The NexMach program represents the final product. NexMach is the last step in achieving the project goal. To recap, the goal of this project is to investigate methods of improving the PREOP expert system (developed using the NexpertObject expert system development software) in three ways. First and foremost, PREOP needs to be independent of NexpertObject; all NexpertObject expert systems require the NexpertObject software to function. Secondly, it is desirable to improve the user interface of PREOP. As it exists now, it is quite primitive, especially in comparison to current PC software. Finally, the PREOP system needs to be changed so that the end-user can make alterations to responses that have already been entered. In other words, one should be able to go back to any previous step and change one's input.

There are several ways one could set out to remove PREOP from the NexpertObject development system. The most obvious of these is to recode PREOP in some conventional language; in fact, this was done (Ho et al., 1992). However, the production rules for PREOP were hard coded. If there were to be any changes to PREOP, the source code would have to be modified and recompiled, causing increased maintenance costs. In addition, any designing and developing that would be done in NexpertObject would have to be manually transcribed into the source code of PREOP. Or, any changes could be made directly in the source code, without going back to the original NexpertObject design. This would mean, however, that all the design and rapid development features of NexpertObject would be lost.

It was decided that the best approach would be to create a system that given a NexpertObject knowledge base file as input, would mimic the functionality of NexpertObject in the execution of the expert system. In other words, the system would be a knowledge base interpreter.

In chapter seven, the functionality of the NexParse program was discussed. NexParse reorganizes the raw knowledge base file into several files that were more readable or more easily interpreted. This represents the first step of a hybrid interpreter (the input text is modified); NexMach is the second step, in that it interprets the results.

In order to improve the human-computer interface of PREOP, a decision was made to explore the possibility of making PREOP a MicroSoft Windows application. The emphasis was on using menus in an effort to make the whole system more intuitive. The end-user can make menu selections using either the keyboard or the mouse. Once a selection is made, if more detailed information is needed, dialog windows are used to retrieve it. A dialog window is a window object that poses a question and provides an input area for the response (i.e., conducts a dialog; see **Figure 9.1** for an example). Dialog windows provide an effective way to focus the users attention on a subset of the expert system.

The overall principle that governed the design of the interface was Clarity of Presentation. All test is presented in clear type and as dark print on a light background. This is for readability. Dialog windows and messages are placed on the screen in locations that attract the users attention. This is for clear operation. Also, no information is displayed or prompted for until it is needed, keeping the display uncluttered. This avoids user confusion. Following the basic principle of clarity resulted in an interface that

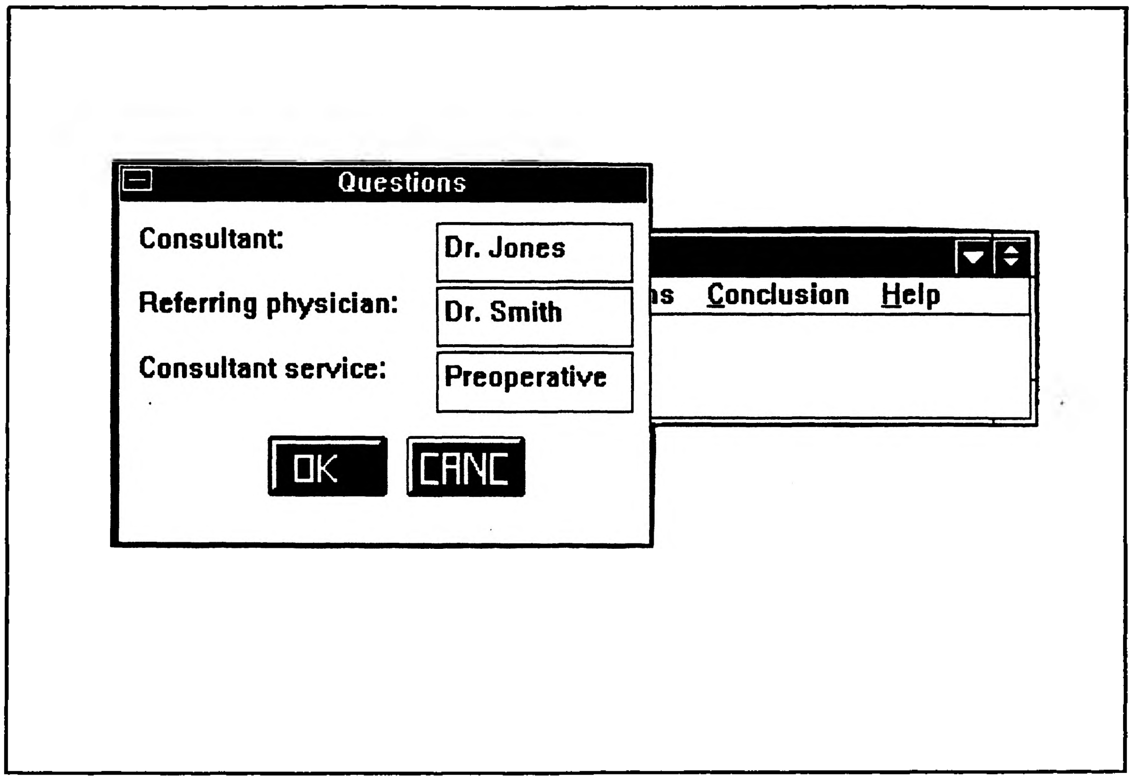

Figure 9.1: A Typical Dialog Box

is not only attractive but is also easy to use.

The implementation of PREOP as a MS Windows application also provided a way to address the issue of altering previously entered data at any time. Since information is grouped into categories, and these categories are represented by menu selections, from the user's perspective, altering previously entered information is simply a matter of re-selecting the appropriate menu option. When this is done, a dialog window, containing the information, will appear. To alter the data, edit the dialog box entry. This change, however, is only to the value of the entry. At the program level, it is necessary to re-evaluate the current state of the expert system, in light of this most recent change. This process will be dealt with in more detail later in this chapter.

9.2: Program Description

When the NexMach program initially starts up, its first task is to retrieve the knowledge base information that was outputted by NexParse and SGroup. This information is stored in several files: one for PROPERTIES, OBJECTS, CLASSES, META-SLOTS, RULES, GLOBALS, and Slot Groups (see chapters seven and eight for more details). The contents of each data file is imported into NexMach and stored in a list. Each list is an object inheriting from the TCollection object (Borland International, 1991).

The TCollection object is a dynamic array that can store any type of object passed to it. Each of the lists adds to the basic TCollection functionality, the ability to import data that is stored on disk. An object is defined for each of the different NexpertObject entities (PROPERTIES, OBJECTS, etc.), and as each of the data files is read, the appropriate object is created and stored in the appropriate list.

Once the initial knowledge base information has been imported, NexMach must begin inferring based on the rules and information at hand. Because the main inference mechanism employed in the initial design of PREOP is backward chaining, NexMach begins with backward chaining. See chapter two from a more detailed description of backward chaining.

Figure 9.2 shows a subset of the PREOP rules as a tree diagram. The figure shows graphically how the evaluation of one rule may require that one first evaluate another rule, which may, in turn, require the evaluation of another rule (i.e. chaining). That is, one rule may contain a condition that is the hypothesis of another rule. If the value of the hypothesis is unknown, then the other rule must be evaluated before the first rule. A problem arises, though, in knowing which rule to return to when a segment of the chain has been evaluated. For example, in evaluating rule 13 in **Figure 9.2**, rule 3 must be evaluated first. In addition, the evaluation of rule 3 requires that other rules be evaluated. Once rule 3 has been evaluated, how do we know to return to rule 13?

To overcome this difficulty, NexMach implements a <u>Conclusion Stack</u>. A Conclusion Stack is an object consisting of a stack, knowledge base entities (PROPERTIES, OBJECTS, CLASSES, RULES and META-SLOTS), and an initial <u>suggestion</u>. The suggestion is the conclusion or hypothesis that is initially suggested to be true in order to start the backward chaining. The stack is used to hold <u>Conclusions</u>. A Conclusion is an object that points to a rule to be evaluated

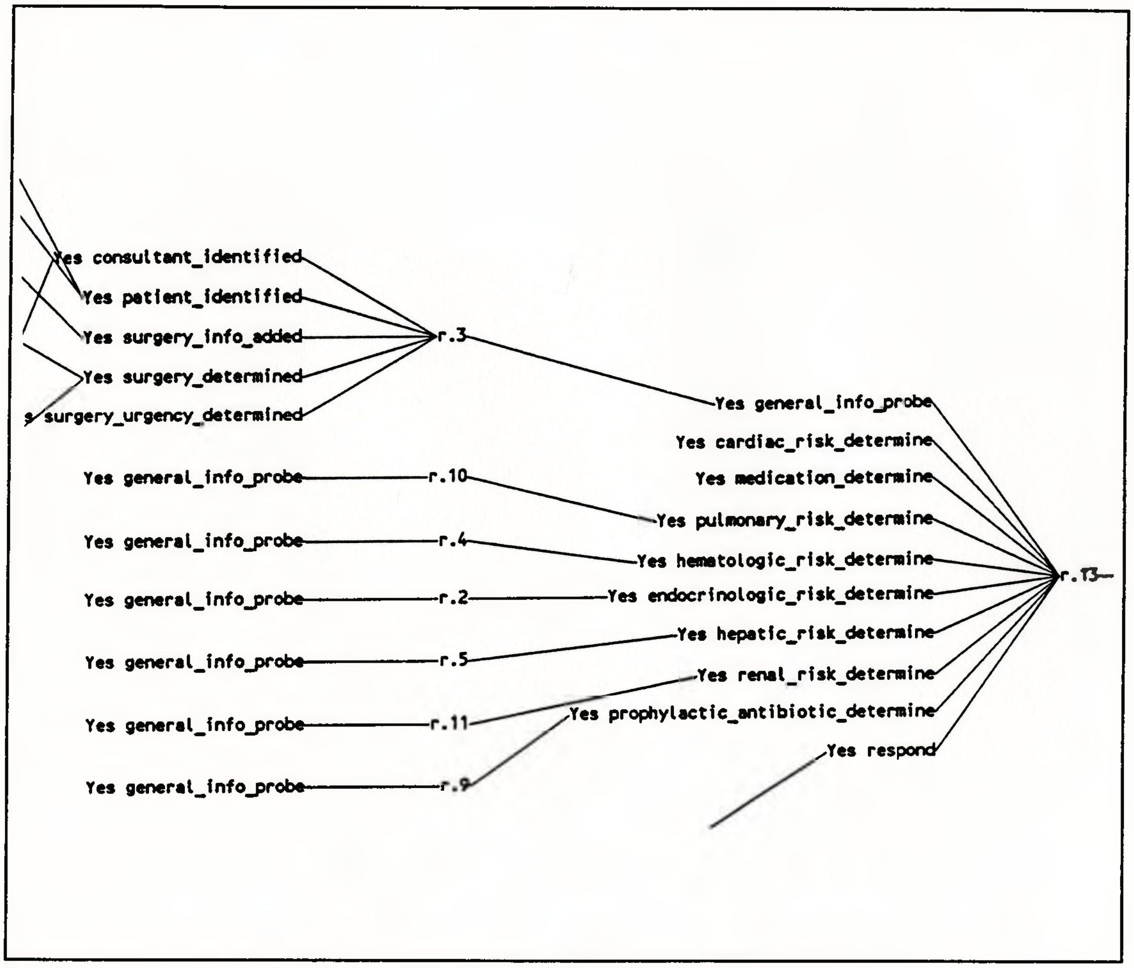
Yes consultant_identified
Yes patient_identified
Yes surgery_info_added
Yes surgery_determined
surgery_urgency_determined
r.3
Yes general_info_probe
r.10
r.4
r.2
r.5
r.11
r.9
Yes cardiac_risk_determine
Yes medication_determine
Yes pulmonary_risk_determine
Yes hematologic_risk_determine
Yes endocrinologic_risk_determine
Yes hepatic_risk_determine
Yes renal_risk_determine
Yes prophylactic_antibiotic_determine
Yes respond
r.13

Figure 9.2: A Subset of the PREOP Rules

and indicates at which clause evaluation is currently at. A clause is a condition or action of a rule. (Recall, RULES can have one or more conditions and one or more actions; the clause is used to keep track of what condition or action is currently being determined.)

When NexMach first starts, the Conclusion Stack is initialized by determining what rule contains the initial hypothesis and pushing a Conclusion object for this rule onto the stack. From here, chaining begins. For each rule that must be evaluated, a Conclusion object is pushed onto the stack. Whenever an unresolved condition is encountered, the rule that will resolve it is found and a Conclusion object for it is pushed onto the stack. Whenever a rule is completely evaluated, its Conclusion object is pulled from the stack and processing continues with the Conclusion on the top of the stack. Once the Conclusion stack is empty, backward chaining is complete and a conclusion or inference can be drawn about the original suggestion.

Figure 9.3 depicts graphically what occurs to the Conclusion Stack during backward chaining. (For a description of the Conclusion Stack and Conclusion object, see the BakChain Unit in Appendix A). The example starts with the evaluation of rule 13. A Conclusion object for rule 13 is placed on the top of the stack (TOS); the clause indicator shows that clause 1 of rule 13 (**Yes general_info_probe**) is to be determined. At this point **general_info_probe** is unresolved; however, the hypothesis (or conclusion) of rule 3 is **general_info_probe**. Therefore, evaluating rule 3 will resolve the hypothesis. So, a Conclusion object for rule 3, with a clause indicator of 1, is pushed onto the stack.

Processing will continue until the last clause of rule 3 (clause 5: **Yes surgery_urgency_determined**) is determined. When this is done, the rule 3 hypothesis, **general_info_probe** will be resolved (either true or false), the Conclusion object for rule 3 is pulled from the stack. Processing will continue with rule 13, clause 2.

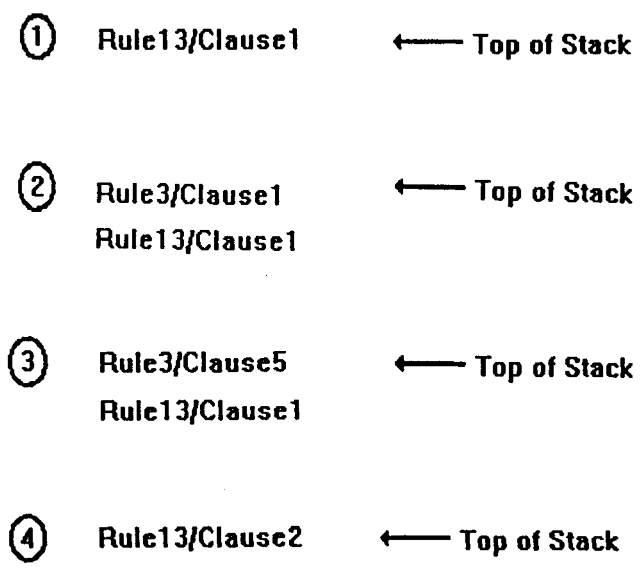

Figure 9.3: The Conclusion Stack During Backchaining

Supporting backward chaining is only half the battle for NexMach. It is also necessary to have a means of obtaining information to infer upon. This information is obtained from the user of the expert system. As stated previously, the key mechanism for obtaining input from the user is the dialog window. Also, recall that the prompting for, and accepting of, information from the user is to be as general as possible. This is the reason for the SGroup utility and its output, the slot-group data file.

The groupings created in SGroup comprise the options of the Questions option of the main menu. When a selection is made, the group name is used to determine which META-SLOTS are to be used and which PROPERTIES will store the information. Each META-SLOT describes prompting information (i.e. the question to be posed). Based on the number of prompts to be displayed in the window and the length of the longest prompt, the size of the dialog window is calculated.

The dialog window is then constructed by defining a window using the dimensions just calculated. Each prompt is added to the window as a static string (i.e. unchanging). To accept the user's input, edit boxes are added after each prompt. An edit box is a rectangular area whose borders are marked with a line. The user's entry appears inside the rectangular area. See the QuesWind Unit in Appendix A for a description of the source code required to implement the dialog window.

The PROPERTY descriptions provide details on the type of data that the user is to input. Currently type checking exists for only three types: real, integer, and string. In some cases, a list of allowable data values is also supplied, but currently, NexMach does not make use of this information. As a future enhancement, this information could be used to present a list of possible values, and validation could be performed to ensure that the data entered is among the allowable values.

Once the information has been entered, the user press an "OK" button in the dialog window to signal that the information should be processed. NexMach then retrieves the information from the edit box, performs type checking, and stores it in the appropriate PROPERTY. If more than one entry was made, this process is performed for each one.

The last component that NexMach required was a NexpertObject command processor. NexpertObject has its own set of operators, which are used in the conditions and actions of RULES. These operators are used to evaluate expressions, to evaluate the state of the expert system, to import data from external files, etc. For example, the **YES** operator will determine if the value of a hypothesis is true or not. For this project, the NexMach program only handles a subset of all the NexpertObject operators, as the knowledge base file used to develop this prototype did not make use of all the available operators.

All RULES are stored as objects, having a hypothesis, a left-hand side and possibly a right-hand side; the left-hand side consists of one or more conditions, and the right-hand side consists of one or more actions. To evaluate a RULE, each condition is evaluated first. If all conditions hold, each action is then performed. The left-hand side and right-hand side are similar in that they are both composed of an operator and at least one operand. Based on the value of the operator, a Pascal case statement is used to call the appropriate procedure to imitate the behaviour of the NexpertObject operator.

CHAPTER 10

CONCLUSIONS

10.1 Conclusions

One of the goals of this project was to investigate the possibility of developing an interpreter of the NexpertObject knowledge file. To this end, the project was successful. The NexMach program can successfully interpret any knowledge file that sticks to the subset of NexpertObject operators that have been included in the NexMach instruction set. This is definitely an avenue worth pursuing. If the set were expanded to include all NexpertObject operands, then it would be possible to create a stand-alone version (i.e., a "NexpertObject-independent" version) of any expert system developed with NexpertObject.

Another project goal was to improve the human-computer interface of PREOP. This has been done in that NexMach translates knowledge file functions to produce a Windows application. That is, the interpretation produces machine-instructions that use the MS Windows interface. However, the NexMach instruction set must be expanded to handle all the NexpertObject operands that PREOP utilizes.

10.2 Future Directions

As previously mentioned, the work done in this project is intended to be investigative. As such, there is a great deal of work that can be picked up from this point. For instance, the set of operators implemented in NexMach could be expanded to include all of the operators supported by NexpertObject.

Similarly, NexMach uses the inference mechanism of backward chaining; however, NexpertObject also supports forward chaining. This too could be added to NexMach. With these sorts of enhancements, NexMach becomes a tool that can interpret any NexpertObject knowledge base file, not just the PREOP expert system.

As for the human-computer interface, more work could be done in improving the overall appearance of the interface. One feature that might prove very valuable is the addition of a tool (or accelerator) bar that allows users to press a button to quickly access menu options.

Finally, with respect to the PREOP application itself, a future direction that could be explored is the development of a system that actually "learns" from past results and modifies itself accordingly. A database of past results would need to be maintained and actions taken based on the contents of this database. Because all the knowledge required for the expert system is stored externally of the executable program, it is possible for the system to modify the knowledge in these files and, thereby, modify itself. This is would truly be a versatile system.

REFERENCES

1. Abate, M.A, Jacknowitz, A.I., and Shumway, J.M., 1989, "Information Sources Utilized by Private Practice and University Physicians", Drug Info Journal, 23, pp. 309-319.

2. Bench-Capon, T., 1990, Knowledge Representation: an Approach Artificial Intelligence. Academic Press, Toronto, Ontario.

3. Borland International, 1991, Turbo Pascal for Windows (Windows Refernce Guide). Borland International, Scotts Valley, California.

4. Cleal, D. and Heaton, N., 1988, Knowledge-based Systems: Implications for Human-Computer Interfaces. Halsted Press, Toronto, Ontario.

5. Covell, D.G., Uman, G.C., and Manning, P.R., 1985, "Information Needs in Office Proctice: Are They Being Met?", Ann. Intern. Med., 103, pp. 596-599.

6. Detskey, A.S., Abrams, H.B., McLaughlin, J.R., Drucker, D.J. Sasson, A., Johnston, N., Scott, J.G., Forbath, N., and Hilliard, J.R., 1986, "Predicting Cardiac Complications in Patients Undergoing Non-cardiac Surgery", J. Gen. Int. Med., 1, pp. 211-219.

7. Goodwin, M., 1988, User Interfaces in C. Programmer's Guide to State-of-the-Art Interfaces. Management Information Source Inc., Portland, Oregon.

8. Ho, J., Leung, S., and Zohrri, N., 1992, "The Development of a Windows User Interface for PREOP, a Medical Expert System", Computer Science 3MP6 Project, McMaster University, Hamilton, Ontario.

9. Holbrook, A., Langton, K.B., Haynes, R.B., Mathieu, A., and Cohen, S., 1992, "Development of an Evidence-Based Expert System to Assist with Preoperative Assessments", _Proceedings of the Fifteenth Symposium on Computer Applications in Medical Care_, Wasington D.C., pp. 669-673.

10. Hoogendoorn, E., 1991, "Investigations into the Development of Run-Time Interface for Nexpert", M.Sc. Thesis, Department of Computer Science and Systems, McMaster University, Hamilton, Ontario.

11. Hu, David, 1987, _Programmers Reference Guide to Expert Systems_. H.W. Sams, Indianapolis, Indiana.

12. Hulme, Charles, 1984, Reading: "Extracting Information from Printed and Electronically Presented Text", _in:_ Monk, Andrew (editor), _Fundamentals of Human Computer Interaction_, Academic Press, London, Ontario, pp. 35-47.

13. Kolers, P.A., Dachnicky, R.L. and Ferguson, D.C., 1981, "Eye Movement Measurement of Readabililty of CRT Displays", _Human Factors_, _23_, pp. 517-527.

14. Langton, K.B., Ramsden, M.F., Montaxemi, A.R., Solntseff, N. and Haynes, R.B., 1990, "PREOP: An Expert System for Preoperative Assessment", _Proceedings of the Second IASTED International Symposium_, M.H. Hamza (editor), pp. 167-170.

15. Levine, Drang, and Edelson, 1990, _AI and Expert Systems: a Comprehensive Guide_, C Language. McGraw-Hill, New York.

16. MacLennan, Bruce, 1987, _Principles of Programming Languages_. 2nd ed., Holt, Rinehart, and Winston, Inc., Toronto, Ontario.

17. MicroSoft Corporation, 1991, _MicroSoft Windows Version 3.1_. MicroSoft Corporation, USA.

18. Muter, P., Latremouille, S.A., Treurniet, W.C., and Beam, P., 1982, "Extended Reading of Continuous Text on Television Screens", Human Factors, 24, pp. 501-508.

19. Neuron Data, 1991, NexpertObject, Introduction Manual. Neuron Data, Palo Alto, California.

20. Norton, Peter and Yao, Paul, 1992, Borland C++ Programming for Windows. Bantam Books Inc., Toronto, Ontario.

21. Parker, C.S., 1989, Management Information Systems. McGraw-Hill, Inc., Toronto, Ontario.

22. Radl, G.W., 1980, "Experimental Investigations for Optimal Presentation-Mode and Colours of Symbols on the CRT-Screen", in: Grandjean, E. and Vigliani, E. (editors), Ergonomic Aspects of Visual Display Terminals, Taylor and Francis, London.

23. Reid, P., 1984, "Work Station Design, Activities and Display Techniques", in: Monk, Andrew (editor), Fundamentals of Human-Computer Interaction, Academic Press, London, Ontario, pp. 107-126.

24. Schildt, Herbert, 1991, C++: The Complete Reference. Osbourne McGraw-Hill, Toronto, Ontario.

25. Snyder, Alan, 1986, "Encapsulation and Inheritance in Object-Oriented Programming Languages", OOPSLA '86 Conference Proceedings, 21:11, pp. 38-45.

26. Turbo Pascal for Windows: Windows Programming Guide. 1991, Borland International, Scotts Valley, California.

27. Wegner, Peter, 1987, "Dimensions of Object-Based Language Design", OOPSLA '87 Conference Proceedings, pp. 168-182.

28. Zarrella, John, 1982, Language Translators. MicroComputer Applications, Suisun City, California.

APPENDIX A

NEXPARSE

<u>NEXPARSE.PAS</u>

```
Program Nexparse (input, output);

Uses Dos, Decode, RuleTree;

Const
  FILE_NOTFOUND_MESS = 'Error: Input file not found.';
  NUM_PARAMETERS     = 2;

Var
  rule_tree : PRule_Node;
  input_filename, output_filename : TFilename;
  lists : TLists;

Procedure Display_Usage;
{-----------------------------------------------------------
Display_Usage: information pertaining to the usage of this
program is displayed on the screen.  The program then
halts.
Called by: Process_Commandline
------------------------------------------------------------}

  begin
    { Display usage information.}
    writeln;
    writeln('USAGE: ',paramstr(0),' <input_file_name>
      <output_file_name>');
    writeln;
    writeln('  where input_file_name is the full file name,
      including path,');
    writeln('  of the file to be parsed, and
      output_file_name is the prefix');
    writeln('  for the file name, including path, of the
      files in which the');
    writeln('  parsed data is to be written to.  A file name
      prefix is the');
    writeln('  file name without the extension.');
    writeln;
    writeln('  eg. ',paramstr(0),' preop.tkb testrun');
    writeln;
    writeln('  The file preop.tkb will be parsed and the
      following output');
    writeln('  files will be produced: ');
    writeln;
    writeln('    testrun.prp       testrun.cls
      testrun.obt');
    writeln('    testrun.slt       testrun.rul
      testrun.gbl');

    { Stop execution.}
    halt(1);

  end; {Display_Usage}

Procedure Process_Commandline(var input_filename,
                              output_filename: TFilename);
{----------------------------------------------------------
Process_Commandline: check the commandline for the proper
number of parameters; if the number is correct, retrieve the
parameters into the string variables, INPUT_FILENAME and
OUTPUT_FILENAME.  Otherwise, usage information is displayed.
Called by: Main
-----------------------------------------------------------}

  var num_params : integer;

  begin
    num_params := paramcount;

    { If there aren't 2 parameters on the command line,
      remind the user how this program is suppose to be
      used.}
    if (num_params <> NUM_PARAMETERS) then Display_Usage

    { Otherwise, retrieve the parameters.}
    else
      begin
        input_filename  := paramstr(1);
        output_filename := paramstr(2);
      end; {else}

  end; {Process_Commandline}

FUNCTION FNAME_CHK (fname: TFilename): boolean;
{ ---------------------------------------------------------
Fname_Chk: Using the FSEARCH() function of TurboPascal,
FNAME_CHK scans the drive specified by DRV for the file
whose name matches FNAME.  If the file is found (ie. P which
holds the full file name, including the path, is not empty)
then the function returns TRUE, otherwise, it returns FALSE.
Called by: Open_Files
----------------------------------------------------------}

  var   p : TFilename;
        i : byte;

  begin
    p := fsearch(fname, '');
    i := length(p);
    fname_chk := i <> 0;

  end; {Fname_Chk}

Procedure Open_Files(filename: TFilename);
{ -------------------------------------------------------
Open_Files: the input file is opened, using the name given
on the command line.
Called by: Main
----------------------------------------------------------}

  begin
    { Check to see if the input file really does exist.}
    if (Fname_Chk(filename)) then
      begin
        { Open the input file.}
        assign(target_file, filename);
        reset(target_file);
      end {if}

    { If the input file doesn't exist, display an error
message and halt.}
    else
      begin
        writeln(FILE_NOTFOUND_MESS);
        halt(1);
      end; {else}
  end; {Open_Files}

Procedure Close_Files;
{ ---------------------------------------------------------
Close_Files: both the input and the output file are closed.
Called by: Main
-----------------------------------------------------------}
  begin
    { Close all files.}
    close(target_file);
  end; {Close_Files}
```

```
Procedure Create_Rule_Tree(var rule_tree: PRule_Node; var
rules: TRules; var globals: TGlobals);
{ -----------------------------------------------------------
Create_Rule_Tree: a tree (RULE_TREE) is constructed from the
rule data extracted from the input file.
Called by: Main
Calls: Process_Line
-----------------------------------------------------------}

  begin
    old_hypos.first := nil;
    rule_tree := Build_Rtree(rules,globals);

  end; {Create_Rule_Tree}

Procedure Parse_Input(var lists: TLists);
{ ----------------------------------------------------------
Parse_Input: the input file is read in, one line at a time.
Each line is then sent to the line processing routine.
Called by: Main
Calls: Process_Line
----------------------------------------------------------}

  var line : TLine;

  begin
    { Discard the first line.}
    readln(target_file, line);

    { Process each line until the end of file marker.}
    while (not eof(target_file)) do
      begin
        { Process the next line.}
        readln(target_file, line);
        Process_Line(line,lists);
      end; {while}

  end; {Parse_Input}

{ ----------------------------------------------------------
Main:  first, the input and output files are opened.
Following this, the lists which will hold the different
elements of the input file, once it has been processed, are
initialized.  Each line of the input file is then read and
processed.  Lastly, the aforementioned lists are written to
the output file, and all files are closed.
Calls: Process_Line, Open_Files, Init_Lists, Display_Lists
----------------------------------------------------------}
```

```
Begin
  { Check the command line parameters.}
  Process_Commandline(input_filename, output_filename);

  { Open the input and output files.}
  Open_Files(input_filename);

  { Initialize all lists.}
  Init_Lists(lists);

  { Process the input file.}
  Parse_Input(lists);

  { Display all lists.}
  Display_Lists(lists, output_filename);

  { Create the rule tree.}
  Create_Rule_Tree(rule_tree, lists.rules, lists.globals);
  if (rule_tree <> nil) then
      Preorder_Rtree(rule_tree);

  { Close all files.}
  Close_Files;
End. {Main}
```

<u>DECODE.UNT</u>

```
Unit Decode;

Interface
  Const
    FILENAME_LENGTH = 255;
    MAX_LINE_LENGTH = 255;  { Max len of an output line. }
    PROMPT_MAX      = 255;  { Max len of a slot prompt. }
    NUM_MAX         = 5;    { Max digits in a inference }
                            { catagory number. }
  Type
    TFilename = string[FILENAME_LENGTH];
    TLine     = string[MAX_LINE_LENGTH];
    TIndex    = 0..MAX_LINE_LENGTH+1;
    TPrompt   = string[PROMPT_MAX];

    { Types associated with the PROPERTIES list.}
    PProperty = ^TProperty;
    TProperty = record
                  property_name : TLine;
                  property_type : TLine;
                  next          : PProperty;
                end; {record}
    TProperties = record
                    first : PProperty;
                    last  : PProperty;
                  end; {record}

    { Types associated with the CLASSES list.}
    PClass_Props = ^TClass_Props;
    TClass_Props = record
                     cp_name : TLine;
                     next    : PClass_Props;
                   end; {record}
    PClass_Subclass = ^TClass_Subclass;
    TClass_Subclass = record
                        cs_name : TLine;
                        next    : PClass_Subclass;
                      end; {record}
    PClass = ^TClass;
    TClass = record
               class_name    : TLine;
               class_props   : PClass_Props;
               last_cp       : PClass_Props;
               class_subclass: PClass_Subclass;
               last_cs       : PClass_Subclass;
               next          : PClass;
             end; {record}
```

```
TClasses = record
             first : PClass;
             last  : PClass;
           end; {record}

{ Types associated with the OBJECTS list.}
PObject_Class = ^TObject_Class;
TObject_Class = record
                  oc_name : TLine;
                  next    : PObject_Class;
                end; {record}
PObject_Props = ^TObject_Props;
TObject_Props = record
                  op_name : TLine;
                  next    : PObject_Props;
                end; {record}
PObject = ^TObject;
TObject = record
            object_name     : TLine;
            object_val      : boolean;
            object_val_type: TLine;
            object_classes  : PObject_Class;
            last_oc         : PObject_Class;
            object_props    : PObject_Props;
            last_op         : PObject_Props;
            next            : PObject;
          end; {record}
TObjects = record
             first : PObject;
             last  : PObject;
           end; {record}

{ Types associated with the SLOTS list.}
PExpr = ^TExpr;
TExpr = record
          operator : TLine;
          operand1 : TLine;
          operand2 : TLine;
          next     : PExpr;
        end; {record}
PContext = ^TContext;
TContext = record
             context_name : TLine;
             next         : PContext;
           end; {record}
PStrat = ^TStrat;
TStrat = record
           strat_lhs : TLine;
           strat_rhs : TLine;
```

```
            next      : PStrat;
          end; {record}
PSlot = ^TSlot;
TSlot = record
           slot_name    : TLine;
           slot_prompt  : TPrompt;
           slot_format  : TPrompt;
           slot_source  : PExpr;
           last_source  : PExpr;
           slot_context: PContext;
           last_context: PContext;
           slot_strat   : PStrat;
           last_strat   : PStrat;
           next         : PSlot;
        end; {record}
TSlots = record
            first : PSlot;
            last  : PSlot;
          end; {record}

{ Types associated with the RULES list.}
PRule = ^TRule;
TRule = record
           rule_name   : TLine;
           rule_infcat: TLine;
           rule_lhs    : PExpr;
           last_lhs    : PExpr;
           rule_rhs    : PExpr;
           last_rhs    : PExpr;
           rule_hypo   : TLine;
           next        : PRule;
        end; {record}
TRules = record
            first : PRule;
            last  : PRule;
          end; {record}

{ Types associated with the GLOBALS list.}
PGlobal = ^TGlobal;
TGlobal = record
             global_lhs : TLine;
             global_rhs : TLine;
             next       : PGlobal;
           end; {record}
TGlobals = record
                first : PGlobal;
                last  : PGlobal;
              end; {record}
```

```
  { The record structure holding all lists.}
  TLists = record
             properties : TProperties;
             classes    : TClasses;
             objects    : TObjects;
             slots      : TSlots;
             rules      : TRules;
             globals    : TGlobals;
           end; {record}

Var
  target_file : text; { This file accessed globally.}

Function Next_Word(line: TLine; i,j: TIndex): TIndex;
Procedure Parse_Word(line: TLine; delimiter: char;
                     var pword: TLine; var i,j: TIndex);
Function Extract_Name(line: TLine; i,j: TIndex): TLine;
Procedure Init_Properties(var properties: TProperties);
Procedure Add_Properties(pname, ptype: TLine; var
   properties: TProperties);
Procedure Display_Properties(properties: TProperties;
   prefix: TFilename);
Procedure Init_Classes(var classes: TClasses);
Procedure Add_Classes(cname: TLine; var classes:
   TClasses);
Procedure Add_Class_Props(cpname: TLine; var classes:
   TClasses);
Procedure Add_Class_Subclass(csname: TLine; var classes:
   TClasses);
Procedure Display_Classes(classes: TClasses; prefix:
   TFilename);
Procedure Init_Objects(var objects: TObjects);
Procedure Add_Objects(oname: TLine; var objects:
   TObjects);
Procedure Add_Object_Classes(ocname: TLine; var objects:
   TObjects);
Procedure Add_Object_Props(line,opname: TLine; var i,j:
   TIndex; var objects: TObjects);
Procedure Display_Objects(objects: TObjects; prefix:
   TFilename);
Procedure Init_Slots(var slots: TSlots);
Procedure Add_Slots(sname: TLine; var slots: TSlots);
Procedure Process_Prompt_Format(line: TLine; var prompt:
   TPrompt; var i,j: TIndex);
Procedure New_Expr(var p: PExpr; optr, oprd1, oprd2:
   TLine);
Procedure Process_Expr(var operator, operand1, operand2:
   TLine);
Procedure Process_Source(var slots: TSlots);
```

```
  Procedure Add_Context(cname: TLine; var slots: TSlots);
  Procedure Process_Context(var slots: TSlots);
  Procedure Process_Strat(strat_var,line: TLine; i,j:
     TIndex; var slots: TSlots);
  Procedure Display_Slots(slots: TSlots; prefix: TFilename);
  Procedure Init_Rules(var rules: TRules);
  Procedure Add_Rules(rname: TLine; var rules: TRules);
  Procedure Process_Infcat(line: TLine; i,j: TIndex; var
     rules: TRules);
  Procedure Process_Lhs(var rules: TRules);
  Procedure Process_Rhs(var rules: TRules);
  Procedure Process_Hypo(line: TLine; i,j: TIndex; var
     rules: TRules);
  Procedure Display_Rules(rules: TRules; prefix: TFilename);
  Procedure Init_Globals(var globals: TGlobals);
  Procedure Add_Globals(lhs, rhs: TLine; var globals:
     TGlobals);
  Procedure Display_Globals(globals: TGlobals; prefix:
     TFilename);
  Procedure Parse_Fn(line: TLine; var fn: TLine;
     var i,j: TIndex);
  Procedure Do_Property(line: TLine; var i,j: TIndex;
     var properties: TProperties);
  Procedure Do_Class(line: TLine; var i,j: TIndex; var
     classes: TClasses);
  Procedure Do_Object(line: TLine; var i,j: TIndex; var
     objects: TObjects);
  Procedure Do_Slot(line: TLine; var i,j: TIndex; var slots:
     TSlots);
  Procedure Do_Rule(line: TLine; var i,j: TIndex; var rules:
     TRules);
  Procedure Do_Global(var globals: TGlobals);
  Procedure Init_Lists(var lists: TLists);
  Procedure Process_Line(line: TLine; var lists: TLists);
  Procedure Display_Lists(lists: TLists; prefix: TFilename);

Implementation

Function Next_Word(line: TLine; i,j: TIndex): TIndex;
{ -----------------------------------------------------------
Next_Word: This function moves the line index, I, to point
to the next non-white-space character in the line, LINE, or
point to the end of the line, J.
Called by: Do_Property, Do_Class, Do_Object,
Add_Object_Props, Do_Slot, Process_Expr, Process_Hypo
 -----------------------------------------------------------}

  begin
    { Increment I until a non-white-space character is found
```

```
      or the end of the line is reached.}
    while ((i <= j) and ((line[i] < '!') or (line[i] >
      '~'))) do i := i+1;
    Next_Word := i;

  end; {Next_Word}

Procedure Parse_Word(line: TLine; delimiter: char;
                     var pword: TLine; var i,j: TIndex);
{ -----------------------------------------------------------
Parse_Word: the line index, I, is used to procede character
by character along the line, LINE, from the current position
of I until the delimiter, DELIMITER, is found or a
white-space character is encountered, adding each character
to the string variable, PWORD.  If neither of the previous
conditions are met, the parsing stops when the end of the
line (J) is reached.
Called by: Parse_Fn, Do_Property, Do_Class, Do_Objects,
      Add_Object_Props, Do_Slot, Process_Expr,
           Process_Hypo, Process_Infcat
 -----------------------------------------------------------}

  var quit : boolean;

  begin
    pword := '';

    { Add each character to PWORD until the DELIMITER is
      found or the end of the line is reached or a
      white-space character is encountered.}
    quit := false;
    while ((i <= j) and (not quit)) do
      begin
        if ((line[i] = delimiter) or (line[i] < ' ') or
          (line[i] > '~')) then
          quit := true
        else
          begin
            pword := pword + line[i];
            i := i+1;
          end; {else}
      end; {while}

  end; {Parse_Word}

Function Extract_Name(line: TLine; i,j: TIndex): TLine;
{ -----------------------------------------------------------
Extract_Name: LINE is parsed for the first word in it; this
word (NAME) is then returned.
```

```
Called by: Do_Class, Do_Object, Do_Slot, Do_Rule
Calls: Next_Word, Parse_Word
 ---------------------------------------------------------}

  var name : TLine;

  begin
    i := i+1;
    i := Next_Word(line,i,j);
    Parse_Word(line,' ',name,i,j);
    Extract_Name := name;

  end; {Extract_Name}

Procedure Init_Properties(var properties: TProperties);
{ ---------------------------------------------------------
Init_Properties: the linked list that holds the list of
properties is initialized to the empty state.
Called by: External
 ---------------------------------------------------------}

  begin
    { Set up an empty list.}
    properties.first := nil;
    properties.last  := nil;

  end; {Init_Properties}

Procedure Add_Properties(pname, ptype: TLine; var
                         properties: TProperties);
{ ---------------------------------------------------------
Add_Properties: a new PROPERTIES list element is created (P)
and the values PNAME and PTYPE are inserted in it.  If the
list is empty, it is added to the front of the list;
otherwise, the new element is added to the end of the list.
Called by: Do_Property
 ---------------------------------------------------------}

  var p : PProperty;

  begin
    { Create a new property list element.}
    new(p);
    p^.property_name := pname;
    p^.property_type := ptype;
    p^.next          := nil;

    if (properties.first = nil) then
      begin
```

```
        { New element is the first of the list.}
        properties.first := p;
        properties.last  := p;
      end
    else
      begin
        { New element is added to the end of the list.}
        properties.last^.next := p;
        properties.last := p;
      end;

  end; {Add_Properties}

Procedure Display_Properties(properties: TProperties;
                             prefix: TFilename);
{ ------------------------------------------------------------
Display_Properties: the linked list that holds the list of
properties is displayed from start to end.  P is used as a
temporary pointer to traverse the list.
Called by: External
 -----------------------------------------------------------}

  var p : PProperty;
      file_name : TFilename;
      prop_file: text;

  begin
    file_name := concat(prefix, '.prp');
    assign(prop_file, file_name);
    rewrite(prop_file);
    writeln(prop_file,'PROPERTIES');

    p := properties.first;

    { Display the attributes of each property in the list.}
    while (p <> nil) do
      begin
        writeln(prop_file, p^.property_name);
        writeln(prop_file, p^.property_type);
        writeln(prop_file);
        p := p^.next;
      end; {while}

    close(prop_file);

    end; {Display_Properties}
```

```
Procedure Init_Classes(var classes: TClasses);
{ -----------------------------------------------------------
Init_Classes: the linked list that holds the list of classes
is initialized to the empty state.
Called by: External
-----------------------------------------------------------}

  begin
    { Set up an empty list.}
    classes.first := nil;
    classes.last  := nil;

  end; {Init_Classes}

Procedure Add_Classes(cname: TLine; var classes: TClasses);
{ -----------------------------------------------------------
Add_Classes: a new class element (C) is created and added to
the list of classes.  Initially, CNAME is inserted in the
element, and the list of properties associated with this
class (CLASS_PROPS) is set to the empty state.
Called by: Do_Class
----------------------------------------------------------}

  var c : PClass;

  begin
    { Create a new class list element.}
    new(c);
    c^.class_name  := cname;
    c^.class_props := nil;
    c^.last_cp     := nil;
    c^.class_subclass := nil;
    c^.last_cs        := nil;
    c^.next           := nil;

    if (classes.first = nil) then
      begin
        { New element is the first of the list.}
        classes.first := c;
        classes.last  := c;
      end
    else
      begin
        { New element is added to the end of the list.}
        classes.last^.next := c;
        classes.last := c;
      end;

  end; {Add_Classes}
```

```
Procedure Add_Class_Props(cpname: TLine; var classes:
TClasses);
{ ----------------------------------------------------------
Add_Class_Props: a new element is created for the list of
properties, associated with the current class
(CLASSES.LAST^).  CPNAME is inserted into this element.  As
before, an empty list is treated as a special case.
Called by: Do_Class
---------------------------------------------------------}

  var cp : PClass_Props;

  begin
    { Create a new element for the list of properties for
      the current class.}
    new(cp);
    cp^.cp_name := cpname;
    cp^.next    := nil;

    { New element is the first in the list.}
    if (classes.last^.class_props = nil) then
      begin
        classes.last^.class_props := cp;
        classes.last^.last_cp     := cp;
      end
    { Add element to the end of the list.}
    else
      begin
        classes.last^.last_cp^.next := cp;
        classes.last^.last_cp := cp;
      end;

  end; {Add_Class_Props}

Procedure Add_Class_Subclass(csname: TLine; var classes:
TClasses);
{ ----------------------------------------------------------
Add_Class_Subclass: a new subclass element (CSNAME) is added
to the list of subclasses associated with the current class.
Called by: Do_Class
-----------------------------------------------------------}

  var cs : PClass_Subclass;

  begin
    { Create a new element for the list of properties for
      the current class.}
    new(cs);
    cs^.cs_name := csname;
    cs^.next    := nil;

    { New element is the first in the list.}
    if (classes.last^.class_subclass = nil) then
      begin
        classes.last^.class_subclass := cs;
        classes.last^.last_cs        := cs;
      end
    { Add element to the end of the list.}
    else
      begin
        classes.last^.last_cs^.next := cs;
        classes.last^.last_cs := cs;
      end;

  end; {Add_Class_Subclass}

Procedure Display_Classes(classes: TClasses; prefix:
TFilename);
{ --------------------------------------------------------
Display_Classes: the linked list that holds the list of
classes is displayed from start to end.  C is used as a
temporary pointer to traverse the list, and CP is used as to
traverse the list of class properties within each class.
Called by: External
----------------------------------------------------------}

  var c  : PClass;
      cp : PClass_Props;
      cs : PClass_Subclass;
      file_name  : TFilename;
      class_file: text;

  begin
    file_name := concat(prefix, '.cls');
    assign(class_file, file_name);
    rewrite(class_file);
    writeln(class_file,'CLASSES');

    c := classes.first;

    { Display the attributes of each class in the list.}
    while (c <> nil) do
      begin
        writeln(class_file, c^.class_name);
        cs := c^.class_subclass;
        cp := c^.class_props;

        { Display the contents of the subclass list for the
```

```
          current class.}
        while (cs <> nil) do
          begin
            writeln(class_file, 'SC ',cs^.cs_name);
            cs := cs^.next;
          end; {while}

        { Display the contents of the property list for the
          current class.}
        while (cp <> nil) do
          begin
            writeln(class_file, 'PR ',cp^.cp_name);
            cp := cp^.next;
          end; {while}

        writeln(class_file);
        c := c^.next;
      end; {while}

      close(class_file);

    end; {Display_Classes}

Procedure Init_Objects(var objects: TObjects);
{ -------------------------------------------------------
Init_Objects: the linked list that holds the list of objects
is initialized to the empty state.
Called by: External
---------------------------------------------------------}

  begin
    { Set up an empty list.}
    objects.first := nil;
    objects.last  := nil;

  end; {Init_Objects}

Procedure Add_Objects(oname: TLine; var objects: TObjects);
{ -------------------------------------------------------
Add_Objects: a new class element (O) is created and added to
the list of objects.  Initially, ONAME is inserted in the
element, and the list of classes associated with this object
(OBJECT_CLASSES), and the list of properties associated with
this object (OBJECT_PROPS) are set to the empty state.
Called by: Do_Object
---------------------------------------------------------}

  var o : PObject;
  begin
    { Create a new object list element.}
    new(o);
    o^.object_name:= oname;
    o^.object_val  := false; { Assume no value declaration.}
    o^.object_val_type:= '';
    o^.object_classes:=nil; { All associated lists}
    o^.last_oc := nil;      { are empty.          }
    o^.object_props   := nil;
    o^.last_op        := nil;
    o^.next           := nil;

    if (objects.first = nil) then
      begin
        { New element is the first of the list.}
        objects.first := o;
        objects.last  := o;
      end
    else
      begin
        { New element is added to the end of the list.}
        objects.last^.next := o;
        objects.last := o;
      end;

  end; {Add_Objects}

Procedure Add_Object_Classes(ocname: TLine; var objects:
TObjects);
{ --------------------------------------------------------
Add_Object_Classes: a new element is created for the list of
classes associated with the current object (OBJECTS.LAST^).
OCNAME is inserted into this element.  As before, an empty
list is treated as a special case.
Called by: Do_Object
---------------------------------------------------------}

  var oc : PObject_Class;

  begin
    { Create a new element for the list of classes for the
      current object.}
    new(oc);
    oc^.oc_name := ocname;
    oc^.next    := nil;

    { New element is the first in the list.}
    if (objects.last^.object_classes = nil) then
      begin
```

```
        objects.last^.object_classes := oc;
        objects.last^.last_oc        := oc;
      end
    { Add element to the end of the list.}
    else
      begin
        objects.last^.last_oc^.next := oc;
        objects.last^.last_oc := oc;
      end;

  end; {Add_Object_Classes}

Procedure Add_Object_Props(line,opname: TLine; var i,j:
                     TIndex; var objects: TObjects);
{ ---------------------------------------------------------
Add_Object_Props: a new element is created for the list of
properties, associated with the current object
(OBJECTS.LAST^).  OPNAME is inserted into this element.  As
before, an empty list is treated as a special case.
Called by: Do_Object
---------------------------------------------------------}

  var op    : PObject_Props;
  var vtype : TLine;

  begin
    { Check if the current object has a value.}
    if (opname = 'Value') then
      begin
        { Find the type of this value.}
        { Skip @TYPE declaration.}
        i := Next_Word(line,i,j);
        Parse_Word(line,'=',vtype,i,j);
        i := i+1;

        { Fetch the type and store it in the current object
          record.}
        Parse_Word(line,';',vtype,i,j);
        objects.last^.object_val      := true;
        objects.last^.object_val_type := vtype;
      end {if}

    { Otherwise, this is just an ordinary property.}
    else
      begin
        { Create a new element for the list of properties
          for the current object.}
        new(op);
        op^.op_name := opname;
        op^.next    := nil;

        { New element is the first in the list.}
        if (objects.last^.object_props = nil) then
          begin
            objects.last^.object_props := op;
            objects.last^.last_op      := op;
          end {if}
        { Add element to the end of the list.}
        else
          begin
            objects.last^.last_op^.next := op;
            objects.last^.last_op := op;
          end; {else}
      end; {else}

  end; {Add_Object_Props}

Procedure Display_Objects(objects: TObjects; prefix:
TFilename);
{ ---------------------------------------------------------
Display_Objects: the linked list that holds the list of
objects is displayed from start to end.  O is used as a
temporary pointer to traverse the list, and OC and OP are
used to traverse the list of object classes and object
properties within each object.
Called by: External
----------------------------------------------------------}

  var o  : PObject;
      oc : PObject_Class;
      op : PObject_Props;
      file_name: TFilename;
      obj_file : text;

  begin
    file_name := concat(prefix, '.obt');
    assign(obj_file, file_name);
    rewrite(obj_file);
    writeln(obj_file, 'OBJECTS');

    o := objects.first;

    { Display the attributes of each object in the list.}
    while (o <> nil) do
      begin
        writeln(obj_file, o^.object_name);
        if (o^.object_val) then
          writeln(obj_file,'VA ',o^.object_val_type);
        oc := o^.object_classes;
        op := o^.object_props;

        { Display the classes associated with this object.}
        while (oc <> nil) do
          begin
            writeln(obj_file, 'OC ',oc^.oc_name);
            oc := oc^.next;
          end; {while}

        { Display the properties associated with this
          object.}
        while (op <> nil) do
          begin
            writeln(obj_file, 'OP ',op^.op_name);
            op := op^.next;
          end; {while}

        writeln(obj_file);
        o := o^.next;
      end; {while}

      close(obj_file);

    end; {Display_Objects}

Procedure Init_Slots(var slots: TSlots);
{ -------------------------------------------------------
Init_Slots: the linked list that holds the list of slots is
initialized to the empty state.
Called by: External
---------------------------------------------------------}

  begin
    { Set up an empty list.}
    slots.first := nil;
    slots.last  := nil;

  end; {Init_Slots}

Procedure Add_Slots(sname: TLine; var slots: TSlots);
{ -------------------------------------------------------
Add_Slots: a new SLOTS element is created (S) and the values
SNAME is inserted in it.  If the list is empty, it is added
as the first of the list; otherwise, the new element is
added to the end of the list.
Called by: Do_Slot
---------------------------------------------------------}
```

```
  var s : PSlot;

  begin
    { Create a new slot list element.}
    new(s);
    s^.slot_name    := sname;
    s^.slot_prompt  := '';
    s^.slot_format  := '';
    s^.slot_source  := nil;
    s^.last_source  := nil;
    s^.slot_context:= nil;
    s^.last_context:= nil;
    s^.slot_strat   := nil;
    s^.last_strat   := nil;
    s^.next         := nil;

    if (slots.first = nil) then
      begin
        { New element is the first of the list.}
        slots.first := s;
        slots.last  := s;
      end
    else
      begin
        { New element is added to the end of the list.}
        slots.last^.next := s;
        slots.last       := s;
      end;

  end; {Add_Slots}

Procedure Process_Prompt_Format(line: TLine; var prompt:
                    TPrompt; var i,j: TIndex);
{ -------------------------------------------------------
Process_Prompt_Format: the PROMPT or FORMAT operator was
encountered, and this procedure extracts the string
associated with this operator.  The string starts and ends
with a double quote (").
Called by: Do_Slot
---------------------------------------------------------}

  var quit : boolean;

  begin
    prompt := '';     { Initialize the prompt string. }
    i := i+2;         { Skip the opening double quote.}

    { Continue adding characters to the prompt string until
      a closing double quote is encountered.}
    quit := false;
    while ((not quit) and (not eof(target_file))) do
      begin
        if (i <= j) then  { Check for " and ; }
          quit := ((line[i] = ';') and (line[i-1] = '"'));

        { Case of end of current line.}
        if (i > j) then
          begin
            readln(target_file, line);
            i := 1; j := length(line);
            prompt := prompt + ' ';
          end {if}

        else if (not quit) then
          begin
            prompt := prompt + line[i];
            i := i+1;
          end; {else}
      end; {while}

    { Drop the ending quote.}
    j := length(prompt);
    delete(prompt,j,1);

  end; {Process_Prompt_Format}

Procedure New_Expr(var p: PExpr; optr, oprd1, oprd2: TLine);
{ --------------------------------------------------------
New_Expr: creates a new expression record, putting OPTR,
OPRD1, OPRD2 in the appropriate slots.
Called by: Process_Lhs, Process_Rhs, Process_Source
---------------------------------------------------------}

  begin
    new(p);
    p^.operator := optr;
    p^.operand1 := oprd1;
    p^.operand2 := oprd2;
    p^.next     := nil;

  end; {New_Expr}

Procedure Process_Expr(var operator, operand1, operand2:
                       TLine);
{ --------------------------------------------------------
Process_Expr: the next line in the data file will contains
an expression.  This expression will contain an operator and
one or two operands.  This procedure parses these out.
Called by: Do_Slot, Process_Lhs, Process_Rhs
Calls: Next_Word, Parse_Word
----------------------------------------------------------------}

  var i,j        : TIndex;
      line, temp : TLine;
      quit       : boolean;

  begin
    { The next line contains the expression data.}
    readln(target_file,line);
    i := 1; j := length(line);

    { Parse the expression data.}
    { Parse out the operator name.}
    i := Next_Word(line,i,j);
    i := i+1;
    Parse_Word(line,' ',operator,i,j);

    if (operator <> '') then
      begin
        { Parse out the 1st operand.}
        i := Next_Word(line,i,j);
        i := i+1;
        quit := false;
        while (not quit) do
          begin
            Parse_Word(line,')',temp,i,j);
            operand1 := operand1 + temp;
            quit := ((line[i+1] <= ' ') or (line[i+1] > '~')
                or (i+1 = j));
            if (not quit) then begin
              inc(i);
              operand1 := operand1 + line[i];
            end; {if}
          end; {while}

        { Parse out the 2nd operand, if there is one.}
        i := i+1;
        i := Next_Word(line,i,j);
        if (i <= j) then
          begin
            i := i+1;
            Parse_Word(line,chr(13),temp,i,j);  { Go to the
                                                 end of the line.}
            operand2 := operand2 + temp;

            while (line[j] = '\') do
              begin
```

```
                delete(operand2,length(operand2),1);
                readln(target_file, line);
                i := 1; j := length(line);
                Parse_Word(line,chr(13),temp,i,j);
                operand2 := operand2 + temp;
              end; {while}

            { The last 2 characters will be closing
               parentheses.}
            delete(operand2,length(operand2)-1,2);
          end {if}
        else delete(operand1,length(operand1),1);
      end; {if}

  end; {Process_Expr}

Procedure Process_Source(var slots: TSlots);
{ -------------------------------------------------------
Process_Source: this procedure extracts the source
expressions for the from the input file, that are associated
with the current SLOT.  Each expression (OPERATOR, OPERAND1,
OPERAND2) is added to the list of source expressions
(SLOT_SOURCE) associated with the current SLOT.
Called by: Do_Slot
Calls: Process_Expr, New_Expr
---------------------------------------------------------}

  var quit : boolean;
      p    : PExpr;
      operator, operand1, operand2 : TLine;

  begin
    quit := false;
    while ((not quit) and (not eof(target_file))) do
      begin
        { Fetch the expression into the temporary
          variables.}
        operator := ''; operand1 := ''; operand2 := '';
        Process_Expr(operator,operand1,operand2);

        { Check for end of this SOURCE declaration.}
        quit := operator = '';

        { If not over, add the expression to the SOURCE list
          of the current slot.}
        if (not quit) then
          begin
            New_Expr(p,operator,operand1,operand2);
            { Case of an empty list.}
            if (slots.last^.slot_source = nil) then
              begin
                slots.last^.slot_source := p;
                slots.last^.last_source := p;
              end {if}

            { Otherwise, add it on to the end.}
            else
              begin
                slots.last^.last_source^.next := p;
                slots.last^.last_source       := p;
              end; {else}
          end; {if}
      end; {while}

  end; {Process_Source}

Procedure Add_Context(cname: TLine; var slots: TSlots);
{ -------------------------------------------------------
Add_Context: a new context element is created and added to
the list of contexts, belonging to the current slot.
Called by: Process_Context
---------------------------------------------------------}

  var p : PContext;

  begin
    { Create a new context element.}
    new(p);
    p^.context_name := cname;
    p^.next := nil;

    { If the context list for the current slot is empty, add
      this new element to the list.}
    if (slots.last^.slot_context = nil) then
      begin
        slots.last^.slot_context := p;
        slots.last^.last_context := p;
      end {if}

    { Otherwise, add it to the end of the list.}
    else
      begin
        slots.last^.last_context^.next := p;
        slots.last^.last_context       := p;
      end; {else}

  end; {Add_Context}
```

```
Procedure Process_Context(var slots: TSlots);
{ ---------------------------------------------------------
Process_Context: this procedure extracts from the input file
all context names (NAME) until a closing parenthesis is
found.  Each name is then inserted into the list of
contexts, belonging to the current slot.
Called by: Do_Slot
Calls: Extract_Name, Add_Context
---------------------------------------------------------}

  var line,name : TLine;
      i,j       : TIndex;

  begin
    { Extract each context name.}
    repeat
      begin
        readln(target_file,line);
        i := 1; j := length(line);
        name := Extract_Name(line,i,j);

        { Add it to the list of contexts.}
        if (name <> ')') then Add_Context(name,slots);
      end; {repeat}
    until (name = ')');

  end; {Process_Context}

Procedure Process_Strat(strat_var,line: TLine; i,j: TIndex;
                         var slots: TSlots);
{ ---------------------------------------------------------
Process_Strat: the current LINE contains a strategy
variable, which has already been parsed out (STRAT_VAR), and
a value for this variable, which has not been parsed out.
This procedure parses out this value and creates a new
strategy element and adds it to the strategy list belonging
to the current slot.
Called by: Do_Slot
Calls: Parse_Word
---------------------------------------------------------}

  var s : PStrat;

  begin
    { Create a new strategy element.}
    new(s);
    s^.strat_lhs  := strat_var;
    s^.next := nil;
    { Get the next word in the line.}
    i := i+1;
    Parse_Word(line,';',s^.strat_rhs,i,j);

    { If the list is empty, make S the first element.}
    if (slots.last^.slot_strat = nil) then
      begin
        slots.last^.slot_strat := s;
        slots.last^.last_strat := s;
      end {if}
    else
      begin
        slots.last^.last_strat^.next := s;
        slots.last^.last_strat       := s;
      end; {else}

  end; {Process_Strat}

Procedure Display_Slots(slots: TSlots; prefix: TFilename);
{ -------------------------------------------------------
Display_Slots: the linked list that holds the list of slots
is displayed from start to end.  S is used as a temporary
pointer to traverse the list and SRC and CON are used to
traverse the source lists and context lists associated with
each slot.
Called by: External
---------------------------------------------------------}

  var s  : PSlot;
      src: PExpr;
      con: PContext;
      str: PStrat;
      file_name : TFilename;
      slot_file : text;

  begin
    file_name := concat(prefix, '.slt');
    assign(slot_file, file_name);
    rewrite(slot_file);
    writeln(slot_file,'SLOTS');

    s := slots.first;

    { Display the attributes of each slots in the list.}
    while (s <> nil) do
      begin
        writeln(slot_file,s^.slot_name);
        if (s^.slot_prompt <> '') then
          writeln(slot_file, 'PR ',s^.slot_prompt);
        if (s^.slot_format <> '') then
          writeln(slot_file, 'FR ',s^.slot_format);

        { Display the list of source expressions.}
        src := s^.slot_source;
        while (src <> nil) do
          begin
            writeln(slot_file, 'SR ',src^.operator);
            writeln(slot_file, src^.operand1);
            if (src^.operand2 <> '') then
                writeln(slot_file, src^.operand2)
            else
                writeln(slot_file);
            src := src^.next;
          end; {while}

        { Display the list of context names.}
        con := s^.slot_context;
        while (con <> nil) do
          begin
            writeln(slot_file, 'CN ',con^.context_name);
            con := con^.next;
          end; {while}

        { Display the list of strategy variables and their
          values.}
        str := s^.slot_strat;
        while (str <> nil) do
          begin
            writeln(slot_file, 'ST ',str^.strat_lhs);
            writeln(slot_file, str^.strat_rhs);
            str := str^.next;
          end; {while}

        writeln(slot_file);
        s := s^.next;
      end; {while}

      close(slot_file);

    end; {Display_Slots}

Procedure Init_Rules(var rules: TRules);
{ ------------------------------------------------------
Init_Rules: the linked list that holds the list of rules is
initialized to the empty state.
Called by: External
---------------------------------------------------------}
```

```
  begin
    { Set up an empty list.}
    rules.first := nil;
    rules.last  := nil;

  end; {Init_Rules}

Procedure Add_Rules(rname: TLine; var rules: TRules);
{ ---------------------------------------------------------
Add_Rules: a new RULES element is created (R) and the values
RNAME is inserted in it.  If the list is empty, it is added
as the first of the list; otherwise, the new element is
added to the end of the list.
Called by: Do_Rules
-----------------------------------------------------------}

  var r : PRule;

  begin
    { Create a new slot list element.}
    new(r);
    r^.rule_name   := rname;
    r^.rule_infcat := '';
    r^.rule_lhs    := nil;
    r^.last_lhs    := nil;
    r^.rule_rhs    := nil;
    r^.last_rhs    := nil;
    r^.rule_hypo   := '';
    r^.next        := nil;

    if (rules.first = nil) then
      begin
        { New element is the first of the list.}
        rules.first := r;
        rules.last  := r;
      end
    else
      begin
        { New element is added to the end of the list.}
        rules.last^.next := r;
        rules.last       := r;
      end;

  end; {Add_Rules}

Procedure Process_Infcat(line: TLine; i,j: TIndex; var
                         rules: TRules);
{ ---------------------------------------------------------
Process_Infcat: an Inference Catagory is an integer ranging
from -32000 to +32000.  This procedure extracts and stores
this value in the current rule record.
Called by: Do_Rule
Calls: Parse_Word
---------------------------------------------------------}

  begin
    { The number is the next 'word' in the line.}
    i := i+1;
    Parse_Word(line,';',rules.last^.rule_infcat,i,j);

  end; {Process_Infcat}

Procedure Process_Lhs(var rules: TRules);
{ -------------------------------------------------------
Process_Lhs: the next one or more lines contain expressions
that are part of the LHS of the current rule.  Each of these
expressions is retrieved from the input file, parsed, and
added to a list of expressions associated with the LHS of
the current rule.
Called by: Do_Rule
Calls: Parse_Word, New_Lhs_Rhs, New_Expr
---------------------------------------------------------}

  var quit : boolean;
      p    : PExpr;
      operator, operand1, operand2 : TLine;

  begin
    quit := false;
    while ((not quit) and (not eof(target_file))) do
      begin
        { Fetch the expression into the temporary
          variables.}
        operator := ''; operand1 := ''; operand2 := '';
        Process_Expr(operator,operand1,operand2);

        { Check for end of LHS.}
        quit := operator = '';

        { Otherwise, add the expression to the LHS of the
          current rule.}
        if (not quit) then
          begin
            New_Expr(p,operator,operand1,operand2);

            { Case of an empty list.}
            if (rules.last^.rule_lhs = nil) then
              begin
```

```
                rules.last^.rule_lhs := p;
                rules.last^.last_lhs := p;
              end {if}

            { Otherwise, add it on to the end.}
            else
              begin
                rules.last^.last_lhs^.next := p;
                rules.last^.last_lhs       := p;
              end; {else}
          end; {if}
      end; {while}

  end; {Process_Lhs}

Procedure Process_Rhs(var rules: TRules);
{ --------------------------------------------------------
Process_Rhs: the next one or more lines contain expressions
that are part of the RHS of the current rule.  Each of these
expressions is retrieved from the input file, parsed, and
added to a list of expressions associated with the RHS of
the current rule.
Called by: Do_Rule
Calls: Parse_Word, New_Lhs_Rhs, New_Expr
---------------------------------------------------------}

  var quit : boolean;
      p    : PExpr;
      operator, operand1, operand2 : TLine;

  begin
    quit := false;
    while ((not quit) and (not eof(target_file))) do
      begin
        { Fetch the expression into the temporary
          variables.}
        operator := ''; operand1 := ''; operand2 := '';
        Process_Expr(operator,operand1,operand2);

        { Check for end of RHS.}
        quit := operator = '';

        { Otherwise, add the expression to the RHS of the
          current rule.}
        if (not quit) then
          begin
            New_Expr(p,operator,operand1,operand2);

            { Case of an empty list.}
          if (rules.last^.rule_rhs = nil) then
            begin
              rules.last^.rule_rhs := p;
              rules.last^.last_rhs := p;
            end {if}

          { Otherwise, add it on to the end.}
          else
            begin
              rules.last^.last_rhs^.next := p;
              rules.last^.last_rhs       := p;
            end; {else}
        end; {if}
    end; {while}

  end; {Process_Rhs}

Procedure Process_Hypo(line: TLine; i,j: TIndex; var rules:
                       TRules);
{ --------------------------------------------------------
Process_Hypo: the hypothesis is the next word in the current
line, delimited by a closing parenthesis.  It is parsed out
and stored in the current rule record.
Called by: Do_Rule
Calls: Parse_Word, Next_Word
----------------------------------------------------------}

  begin
    { The next 'word' in the current line is the
      hypothesis.}
    i := i+1;
    i := Next_Word(line,i,j);
    Parse_Word(line,')',rules.last^.rule_hypo,i,j);

  end; {Process_Hypo}

Procedure Display_Rules(rules: TRules; prefix: TFilename);
{ --------------------------------------------------------
Display_Rules: the linked list that holds the list of rules
is displayed from start to end.  R is used as a temporary
pointer to traverse the list.
Called by: External
----------------------------------------------------------}

  var r : PRule;
      lhs, rhs : PExpr;
      file_name: TFilename;
      rule_file: text;
  begin
    file_name := concat(prefix, '.rul');
    assign(rule_file, file_name);
    rewrite(rule_file);
    writeln(rule_file, 'RULES');

    { Start with the first element of the list.}
    r := rules.first;

    { Process each rule until the list is empty.}
    while (r <> nil) do
      begin
        writeln(rule_file, r^.rule_name);
        if (r^.rule_infcat <> '') then
          writeln(rule_file,'IC ', r^.rule_infcat);

        { Process all LHS expressions.}
        lhs := r^.rule_lhs;
        while (lhs <> nil) do
          begin
            writeln(rule_file,'L1 ',lhs^.operator);
            writeln(rule_file,lhs^.operand1);
            writeln(rule_file,lhs^.operand2);
            lhs := lhs^.next;
          end; {while}

        { Process all RHS expressions.}
        rhs := r^.rule_rhs;
        while (rhs <> nil) do
          begin
            writeln(rule_file,'R1 ',rhs^.operator);
            writeln(rule_file,rhs^.operand1);
            writeln(rule_file,rhs^.operand2);
            rhs := rhs^.next;
          end; {while}

        if (r^.rule_hypo <> '') then
          writeln(rule_file,'HY ', r^.rule_hypo);

        writeln(rule_file);
        r := r^.next;
      end; {while}

    close(rule_file);

  end; {Display_Rules}

Procedure Init_Globals(var globals: TGlobals);
{ -------------------------------------------------------
```

```
Init_Globals: the linked list that holds the list of global
variables is initialized to the empty state.
Called by: External
---------------------------------------------------------}

  begin
    { Set up an empty list.}
    globals.first := nil;
    globals.last  := nil;

  end; {Init_Globals}

Procedure Add_Globals(lhs, rhs: TLine; var globals:
                      TGlobals);
{ -------------------------------------------------------
Add_Globals: a new element is added to the GLOBALS list.
Again, an empty list is a special case.
Called by: Do_Global
---------------------------------------------------------}

  var g : PGlobal;

  begin
    { Create a new global list element.}
    new(g);
    g^.global_lhs := lhs;
    g^.global_rhs := rhs;
    g^.next       := nil;

    if (globals.first = nil) then
      begin
        { New element is the first of the list.}
        globals.first := g;
        globals.last  := g;
      end
    else
      begin
        { New element is added to the end of the list.}
        globals.last^.next := g;
        globals.last := g;
      end;

  end; {Add_Globals}

Procedure Display_Globals(globals: TGlobals; prefix:
                          TFilename);
{ -------------------------------------------------------
Display_Globals: the linked list that holds the list of
properties is displayed from start to end.  G is used as a
```

```
temporary pointer to traverse the list.
Called by: External
---------------------------------------------------------}

  var g : PGlobal;
      file_name : TFilename;
      gbl_file  : text;

  begin
    file_name := concat(prefix, '.gbl');
    assign(gbl_file, file_name);
    rewrite(gbl_file);
    writeln(gbl_file,'GLOBALS');

    g := globals.first;

    { Display the attributes of each global variable in the
      list.}
    while (g <> nil) do
      begin
        writeln(gbl_file, g^.global_rhs);
        g := g^.next;
      end; {while}

    close(gbl_file);

  end; {Display_Globals}

Procedure Parse_Fn(line: TLine; var fn: TLine;
                   var i,j: TIndex);
{ -------------------------------------------------------
Parse_Fn: the current line, LINE, is parsed for the next
word.
Called by: Process_Line
Calls    : Parse_Word
---------------------------------------------------------}

  begin
    { Initialize function name string.}
    fn := '';
    i := i+1;

    { Fetch the function name.}
    Parse_Word(line,'=',fn,i,j);

  end; {Parse_Fn}
```

```
Procedure Do_Property(line: TLine; var i,j: TIndex;
                      var properties: TProperties);
{ -------------------------------------------------------
Do_Property: the current line, LINE, is parsed for the
property name, PNAME, and the property type, PTYPE.  Both of
these values should be found on the current line.
Called by: Process_Line
Calls    : Next_Word, Parse_Word, Add_Properties
---------------------------------------------------------}

  var pname, ptype : TLine;

  begin
    { Find the property's (variable's) name.}
    i := i+1;
    i := Next_Word(line,i,j);
    Parse_Word(line,' ',pname,i,j);

    { Find the type of this property.}
    { Skip @TYPE declaration.}
    i := Next_Word(line,i,j);
    Parse_Word(line,'=',ptype,i,j);
    i := i+1;

    { Fetch the type.}
    Parse_Word(line,';',ptype,i,j);

    { Add these property attributes to the list.}
    Add_Properties(pname, ptype, properties);

  end; {Do_Property}

Procedure Do_Class(line: TLine; var i,j: TIndex; var
                   classes: TClasses);
{ -------------------------------------------------------
Do_Class: first, the current line, LINE, is parsed for the
class name, CNAME.  A new class list element is then added
to the class list.  Next, lines are read from the input
file, TARGET_FILE, and each line is parsed for a single
property name.  Each of these names is added to the list of
class properties associated with the current class.  This
continues until a line containing a closing parenthesis is
encountered.
Called by: Process_Line
Calls    : Next_Word, Parse_Word, Add_Class_Props,
           Add_Class_Subclass
---------------------------------------------------------}

  var temp1, temp2 : TLine;
```

```
      quit          : boolean;

  begin
    { Find the class's name.}
    templ := Extract_Name(line,i,j);
    Add_Classes(templ, classes);

    quit := false;
    while ((not quit) and (not eof(target_file))) do
      begin
        readln(target_file, line);
        i := 1; j := length(line);
        i := Next_Word(line,i,j);
        i := i+1;
        Parse_Fn(line,templ,i,j);

        quit := i > j;
        if (not quit) then
          begin
            while ((not quit) and (not eof(target_file))) do
              begin
                readln(target_file, line);
                i := 1;  j := length(line);
                i := Next_Word(line,i,j);
                Parse_Word(line,' ',temp2,i,j);

                { A closing parenthesis marks the end of the
                  section.}
                if (temp2 = ')') then quit := true
                else if templ = 'PROPERTIES' then
                  Add_Class_Props(temp2, classes)
                else if templ = 'SUBCLASSES' then
                  Add_Class_Subclass(temp2, classes);
              end; {while}
              quit := false;
          end; {else}
      end; {while}

  end; {Do_Class}

Procedure Do_Object(line: TLine; var i,j: TIndex; var
                    objects: TObjects);
{ -------------------------------------------------------
Do_Object: first the name of the object is extracted from
the current line.  The subsequent lines contain the
components of the object; these are either PROPERTIES or
CLASSES.  Each component is enclosed in parenthesis, and
this is fact is used to extract each component and add it to
the appropriate list associated with the current object.
```

```
Called by: Process_Line
Calls    : Next_Word, Parse_Word, Add_Object_Props,
           Add_Object_Classes
---------------------------------------------------------}

  var oname, templ, temp2 : TLine;
      quit                : boolean;

  begin
    { Find the object's name.}
    oname := Extract_Name(line,i,j);
    Add_Objects(oname, objects);

    { Retrieve the component type.}
    quit := false;
    while ((not quit) and (not eof(target_file))) do
      begin
        readln(target_file, line);
        i := 1; j := length(line);
        i := Next_Word(line,i,j);
        i := i+1;
        Parse_Fn(line,templ,i,j);

        quit := i > j;
        if (not quit) then
          begin
            while ((not quit) and (not eof(target_file))) do
              begin
                readln(target_file, line);
                i := 1;  j := length(line);
                i := Next_Word(line,i,j);
                Parse_Word(line,' ',temp2,i,j);

                { A closing parenthesis marks the end of the
                  section.}
                if (temp2 = ')') then quit := true
                else if templ = 'PROPERTIES' then
                  Add_Object_Props(line,temp2,i,j,objects)
                else if templ = 'CLASSES' then
                  Add_Object_Classes(temp2,objects);
              end; {while}
              quit := false;
          end; {else}
      end; {while}

  end; {Do_Object}
```

```
Procedure Do_Slot(line: TLine; var i,j: TIndex; var slots:
                  TSlots);
{ -------------------------------------------------------
Do_Slot:   a SLOT consists of one or more subcomponents
(meta-slots), separated by blank lines.  The first line
contains the name of the slot and the subsequent lines
contain the meta-slots.  After the name is parsed out, a new
element of the slots list is created, and the meta-slots are
then processed.
Called by: Process_Line
Calls    : Next_Word, Parse_Word, Add_Slots,
           Process_Prompt_Format, Process_Source,
           Process_Context
----------------------------------------------------------}

  var sname,fn : TLine;
      quit     : boolean;

  begin
    sname := Extract_Name(line,i,j);
    Add_Slots(sname, slots); { Create a new slot entry.}

    { Extract and process each of the meta-slots associated
      with this slot.}
    quit := false;
    while ((not quit) and (not eof(target_file))) do
      begin
        readln(target_file,line); {Fetch meta-slot name.}
        i := 1; j := length(line);
        i := Next_Word(line,i,j) - 1;
        Parse_Fn(line,fn,i,j);

        if fn = '' then quit := true  { Slots are separated
                                        by empty lines.}
        else if (fn = '@PROMPT') then

Process_Prompt_Format(line,slots.last^.slot_prompt,i,j)
        else if (fn = '@FORMAT') then

Process_Prompt_Format(line,slots.last^.slot_format,i,j)
        else if (fn = '(@SOURCES') then
            Process_Source(slots)
        else if (fn = '(@CONTEXTS') then
            Process_Context(slots)
        else if (fn <> ')') then
            Process_Strat(fn,line,i,j,slots);
      end; {while}

  end; {Do_Slot}
```

```
Procedure Do_Rule(line: TLine; var i,j: TIndex; var rules:
TRules);
{ -------------------------------------------------------
Do_Rule:   each rule consists of at least a LHS and a
HYPOthesis; in addition, there may be a RHS and an INFerence
CATagory.  This routine determines the subcomponents of the
current rule.  Each subcomponent is labelled, and this
procedure acts according to the labels it finds.
Called by: Process_Line
Calls    : Next_Word, Parse_Word, Add_Rules
--------------------------------------------------------}

  var fn   : TLine;
      rname: TLine;
      quit : boolean;

  begin
    rname := Extract_Name(line,i,j);    { Determine the
                                          rules name.}
    Add_Rules(rname, rules);            { Create a new rule
                                          entry. }

    { Process each subcomponent of the current rule.}
    quit := false;
    while ((not quit) and (not eof(target_file))) do
      begin
        readln(target_file,line);
        i := 1; j := length(line);
        i := Next_Word(line,i,j);
        quit := line[i] = ')';
        if (not quit) then
          begin
            i := i+1;
            Parse_Word(line,'=',fn,i,j);
            if (fn = 'INFCAT')     then
Process_InfCat(line,i,j,rules)
            else if (fn = '@LHS')  then Process_Lhs(rules)
            else if (fn = '@HYPO') then
Process_Hypo(line,i,j,rules)
            else if (fn = '@RHS')  then Process_Rhs(rules)
          end; {if}
      end; {while}

  end; {Do_Rule}

Procedure Do_Global(var globals: TGlobals);
{ -------------------------------------------------------
Do_Global: each line, until a closing parenthesis is
encountered, contains the name of a global variable as well
```

```
as the value for this variable (in the form GV=Val).  This
procedure parses the line, extracting and storing the name
and its value in a list of global variables.
Called by: Process_Line
Calls    : Next_Word, Parse_Word, Add_Globals
---------------------------------------------------------}

  var quit    : boolean;
      line    : TLine;
      lhs,rhs: TLine;
      i,j     : TIndex;

  begin
    { Process all globals.}
    quit := false;
    while ((not quit) and (not eof(target_file))) do
      begin
        readln(target_file,line);
        i := 1; j := length(line);
        i := Next_Word(line,i,j);
        Parse_Word(line,'=',lhs,i,j);

        { A closing parenthesis marks the end of the global
          declarations.}
        quit := lhs = ')';
        if (not quit) then
          begin
            i := i+1;
            Parse_Word(line,';',rhs,i,j);
            Add_Globals(lhs,rhs,globals);
          end; {if}
      end; {while}

  end; {Do_Global}

Procedure Init_Lists(var lists: TLists);
{ -------------------------------------------------------
Init_Lists: all the lists for the different file elements
are initialized to the empty state.
Calls: Init_Properties, Init_Classes, Init_Objects,
       Init_Slots, Init_Rules, Init_Globals
---------------------------------------------------------}

  begin
    { Initialize all lists.}
    with lists do
      begin
        Init_Properties(properties);
        Init_Classes(classes);
        Init_Objects(objects);
        Init_Slots(slots);
        Init_Rules(rules);
        Init_Globals(globals);
      end; {with}

  end; {Init_Lists}

Procedure Process_Line(line : TLine; var lists: TLists);
{ -----------------------------------------------------
Process_Line: the first word of LINE is parsed out.  This
word should be a function name; depending on the value of
this function name, the appropriate action is taken.  I and
J are line indices used for parsing.  I represents the
current position in the line and J represents the length of
the line.
Called by: External
Calls    : Parse_Fn, Do_Property, Do_Class, Do_Object,
           Do_Slot
-------------------------------------------------------}

  var i,j : TIndex;
      fn  : TLine;

  begin
    j := length(line);
    i := 1;

    { Find the opening parenthesis.}
    while (i <= j) and (line[i] <> '(') do i := i+1;

    { Determine the function name.}
    Parse_Fn(line,fn,i,j);

    { Take the appropriate action for the given function.}
    with lists do
      if (fn = '@PROPERTY') then
        Do_Property(line,i,j,properties)
      else if (fn = '@CLASS') then
        Do_Class(line,i,j,classes)
      else if (fn = '@OBJECT') then
        Do_Object(line,i,j,objects)
      else if (fn = '@SLOT') then
        Do_Slot(line,i,j,slots)
      else if (fn = '@RULE') then
        Do_Rule(line,i,j,rules)
      else if (fn = '@GLOBALS') then
        Do_Global(globals)
      else if fn <> '' then
```

```
        writeln('Unknown function name, ',fn);

  end; {Process_Line}

Procedure Display_Lists(lists: TLists; prefix: TFilename);
{ ---------------------------------------------------------
Display_Lists: all the lists for the different file elements
are displayed.
Calls: Init_Properties, Init_Classes, Init_Objects,
       Init_Slots, Init_Rules, Init_Globals
---------------------------------------------------------}

  begin
    { Display all lists.}
    with lists do
      begin
        Display_Properties(properties, prefix);
        Display_Classes(classes, prefix);
        Display_Objects(objects, prefix);
        Display_Slots(slots, prefix);
        Display_Rules(rules, prefix);
        Display_Globals(globals, prefix);
      end; {with}

  end; {Display_Lists}

End.
```

RULETREE.UNT

```
Unit RuleTree;

Interface

  Uses Decode;

  Type
    { Types associated with the construction of the RULE
      TREE.}
    PRule_Node = ^TRule_Node;
    TRule_Node = record
                   rule_name : TLine;
                   child     : PRule_Node;
                   sibling   : PRule_Node;
                 end; {record}

    { Types associated with the list of hypotheses already
      processed.}
    POld_Hypo = ^TOld_Hypo;
    TOld_Hypo = record
                  rule : PRule;
                  next : POld_Hypo;
                end; {record}
    TOld_Hypos = record
                   first : POld_Hypo;
                   last  : POld_Hypo;
                 end;

  Var
    old_hypos : TOld_Hypos; { List is accessed globally.}

  Procedure Add_Old_Hypos(rule: PRule);
  Function Seen_Hypo(hypo: TLine): boolean;
  Function Add_Rule(rtree: PRule_Node; r: PRule):
      PRule_Node;
  Procedure Find_Hypo(var r,q: PRule; operand: TLine);
  Procedure Add_to_Tree(rule: PRule; rtree: PRule_Node;
      rules: TRules);
  Function First_Rule(rules: TRules; globals: TGlobals):
      PRule;
  Function Build_Rtree(rules: TRules; globals: TGlobals):
      PRule_Node;
  Procedure Preorder_Rtree(rule_tree: PRule_Node);

Implementation

Procedure Add_Old_Hypos(rule: PRule);
```

```
{ ------------------------------------------------------
Add_Old_Hypos: a new element is added to the list of old
hypotheses (ie. hypotheses which have already been
encountered).  If this list (OLD_HYPOS) is empty, then the
new element is put at the front of the list; otherwise, it
is added to the end of the list.
Called by: Find_Hypo
--------------------------------------------------------}

  var oh : POld_Hypo;

  begin
    new(oh);
    oh^.rule := rule;
    oh^.next := nil;

    if (old_hypos.first = nil) then
      begin
        old_hypos.first := oh;
        old_hypos.last  := oh;
      end {if}
    else
      begin
        old_hypos.last^.next := oh;
        old_hypos.last       := oh;
      end; {else}

  end; {Add_Old_Hypos}

Function Seen_Hypo(hypo: TLine): boolean;
{ ------------------------------------------------------
Seen_Hypo: this function returns a boolean indicating
whether or not HYPO is in the list OLD_HYPOS.
Called by: Add_to_Tree
--------------------------------------------------------}

  var h : POld_Hypo;

  begin
    h := old_hypos.first;

    { Scan the list of old hypotheses for a match.}
    while ((h <> nil) and (h^.rule^.rule_hypo <> hypo)) do
      h := h^.next;
    Seen_Hypo := h <> nil;

  end; {Seen_Hypo}

Function Add_Rule(rtree: PRule_Node; r: PRule): PRule_Node;
```

```
{ ---------------------------------------------------------
Add_Rule: this function creates a new node (P) and inserts
as a child of the rule node pointed to by RTREE.  There are
two scenarios for insertion.  (1) the inserted node is the
1st child (2) the inserted node is the 2nd or more child.
Each of these cases is handled separately.
Called by: Add_to_Tree
---------------------------------------------------------}

  var p,q : PRule_Node;

  begin
    { Create a new node.}
    new(p);
    p^.child     := nil;
    p^.sibling   := nil;
    p^.rule_name := r^.rule_name;

    { Rule has no children yet.}
    if (rtree^.child = nil) then rtree^.child := p

    { Rule already has one child.}
    else
      begin
        q := rtree^.child;

        { Find the end of the sibling chain.}
        while (q^.sibling <> nil) do q := q^.sibling;
        q^.sibling := p;
      end; {else}

    { Return a pointer to the new node.}
    Add_Rule := p;

  end; {Add_Rule}

Procedure Find_Hypo(var r,q: PRule; operand: TLine);
{ ---------------------------------------------------------
Find_Hypo: the list of rules is searched until the end of
the list is reached or a rule is found whose hypothesis
matches OPERAND.
Called by: Add_to_Tree
Calls: Add_Old_Hypos
---------------------------------------------------------}

  begin
    { Scan all rules until the end or a match is made.}
    while ((r <> nil) and (r^.rule_hypo <> operand)) do
      r := r^.next;
    if (r <> nil) then Add_Old_Hypos(r);
    q := r;

  end; {Find_Hypo}

Procedure Add_to_Tree(rule: PRule; rtree: PRule_Node; rules:
TRules);
{ -----------------------------------------------------
Add_to_Tree: this is a recursive procedure which builds a
tree consisting of rule data.  Rules are linked together by
a PARENT - CHILD relationship, although a parent only points
to its first child; the rest are indirectly pointed to via
sibling pointers starting at the first child.  Children of
any given rule are those rules whose hypothesis matches a
hypothesis on the LHS of the parent rule.  All hypotheses on
the LHS of RULE are used to find children of the current
rule tree node (RTREE).
Called by: Add_to_Tree
Calls: Find_Hypo, Add_Rule, Add_to_Tree
---------------------------------------------------------}

  var p   : PExpr;
      q,r : PRule;
      new_rtree : PRule_Node;

  begin
    { Start at the front of the list of LHS records for
      current RULE.}
    p := rule^.rule_lhs;

    { Process all LHS records.}
    while (p <> nil) do
      begin
        { Hypotheses are indicated by a 'Yes' operator.}
        if (p^.operator = 'Yes') then
          if (not Seen_Hypo(p^.operand1)) then
            begin
              r := rules.first;

              { Find all rules that have this hypothesis as
                their RULE_HYPO component.}
              repeat
                begin
                  Find_Hypo(r,q,p^.operand1);
                  if (q <> nil) then
                    begin
                      { Add this new rule to the current
                        rule tree (RTREE), producing a new
                        rule tree (NEW_RTREE).}
                    new_rtree := Add_Rule(rtree,q);
                    { Recursively call this procedure with
                      the new rule tree.}
                    Add_to_Tree(q,new_rtree,rules);
                  end; {if}
                r := r^.next;
              end; {repeat}
            until (r = nil);
          end; {if}
      { Get the next LHS record.}
      p := p^.next;
    end; {while}

  end; {Add_to_Tree}

Function First_Rule(rules: TRules; globals: TGlobals):
PRule;
{ -------------------------------------------------------
First_Rule: this function returns a pointer to the 1st (and
only) rule whose hypothesis is the hypothesis in the
SUGgestion LIST, found in the list of global variables.
Called by: Build_Tree
Calls: Find_Hypo
---------------------------------------------------------}

  var g   : PGlobal;
      p,q : PRule;

  begin
    { Search the list of globals for the SUGgestion LIST.}
    g := globals.first;
    while ((g^.global_lhs <> '@SUGLIST') and (g <> nil)) do
      g := g^.next;

    { Find the rule whose hypothesis matches the one in the
      suggestion list.}
    if (g <> nil) then
      begin
        p := rules.first;
        Find_Hypo(p,q,g^.global_rhs);
      end {if}
    else q := nil;

    First_Rule := q;

  end; {First_Rule}

Function Build_Rtree(rules: TRules; globals: TGlobals):
PRule_Node;
```

```
{ -------------------------------------------------------
Build_Rtree: this function returns a pointer to the rule
tree.  The first rule is found and becomes the root node of
this tree, and then ADD_TO_TREE is called to add the rest of
the rules to this single node tree, in the appropriate
order.
Called by: Main
Calls: First_Rule, Add_to_Tree
-------------------------------------------------------}

  var root_rule : PRule;
      rule_tree : PRule_Node;

  begin
    { Find the first rule for the rule tree.}
    root_rule := First_Rule(rules, globals);

    { Create a one node tree, consisting of this 1st rule.}
    if (root_rule <> nil) then
      begin
        new(rule_tree);
        rule_tree^.child     := nil;
        rule_tree^.sibling   := nil;
        rule_tree^.rule_name := root_rule^.rule_name;
        { Add the rest of the tree to this root.}
        Add_to_Tree(root_rule,rule_tree,rules);

        Build_Rtree := rule_tree;
      end {if}
    else
      Build_Rtree := nil;
  end; {Build_Tree}

Procedure Preorder_Rtree(rule_tree: PRule_Node);
{ -------------------------------------------------------
Preorder_Rtree: a postorder traversal of the rule tree,
RULE_TREE, is performed by recursive calls to this
procedure.
Called by: Main
-------------------------------------------------------}
  begin
    if (rule_tree <> nil) then                { Base Case.}
      begin
        writeln(rule_tree^.rule_name);
        Preorder_Rtree(rule_tree^.child);
        Preorder_Rtree(rule_tree^.sibling);
      end; {if}
  end; {Preorder_Rtree}
End. {RuleTree}
```

SGROUP

SGROUP.PAS

```
program Sgroup(input,output);

uses Strings, WinTypes, WinProcs, WinDos, WObjects, StdDlgs,
     SGLoad, PrmptObj, Group, GrpObj, GDWind, GAWind,
GRWind;

{$R SGROUP.RES}

const
  MAIN_MENU    = 100;    { Main menu id.}

  cm_Open      = 101;    { These constants define the id
numbers for the}
  cm_Save      = 102;    {   various menu selections.
       }
  cm_GrpCreate = 201;
  cm_GrpDelete = 202;
  cm_GrpAdd    = 203;
  cm_GrpRemove = 204;
  cm_Help      = 301;

type
  TSGroupApp = object(TApplication)
    { Methods}
    procedure InitMainWindow; virtual;
  end;

  { The Main window.}
  PSGroupWin = ^TSGroupWin;
  TSGroupWin = object(TWindow)
    { Attributes}
    Modified: Boolean;
    FileName: TFilename;
    Prompts : PCollection;
    Groups  : PCollection;

    { Methods}
    constructor Init(AParent: PWindowsObject; ATitle:
                 PChar);
    destructor Done; virtual;
    function CanClose: Boolean; virtual;
    procedure FileOpen(var Msg: TMessage);
      virtual cm_First + cm_Open;
    procedure FileSave(var Msg: TMessage);
      virtual cm_First + cm_Save;
    procedure GrpCreate(var Msg: TMessage);
      virtual cm_First + cm_GrpCreate;
    procedure GrpDelete(var Msg: TMessage);
      virtual cm_First + cm_GrpDelete;
    procedure GrpAdd(var Msg: TMessage);
      virtual cm_First + cm_GrpAdd;
    procedure GrpRemove(var Msg: TMessage);
      virtual cm_First + cm_GrpRemove;
    procedure Help(var Msg: TMessage);
      virtual cm_First + cm_Help;
  end;

{-----------------------------------------------}
{ TSGroupWin's method implementations:          }
{-----------------------------------------------}

constructor TSGroupWin.Init(AParent: PWindowsObject; ATitle:
PChar);
{----------------------------------------------------------
Init: this is the constructor for the TSGroupWin object.  It
simply calls the TWindow constructor, loads the main menu
(100), initializes the list of slots with prompts (PROMPTS)
and the list of GROUPings, and sets MODIFIED, which
indicates whether or not the GROUPings have modified, to
false.
-----------------------------------------------------------}
begin
  TWindow.Init(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, PChar(MAIN_MENU));
  Init_Prompts(Prompts);
  Init_Groups(Groups);
  Modified := false;

end; {TSGroupWin.Init}

destructor TSGroupWin.Done;
{----------------------------------------------------------
Done: this is the destructor for the TSGroupWin object.  All
lists are deallocated.
----------------------------------------------------------}
begin
  TWindow.Done;
  Prompts^.FreeAll;
  Groups^.FreeAll;
  Dispose(Prompts, done);
  Dispose(Groups, done);

end; {TSGroup.Done}

function TSGroupWin.CanClose: Boolean;
{-----------------------------------------------------------
CanClose: If the current data has not been saved since the
last change, the user is asked if he/she wishes to save
before exiting, or cancel the exit command.
-----------------------------------------------------------}
var
  Reply: Integer;

begin
  CanClose := true;

  { Has the data been modified?}
  if (MODIFIED) then
    begin
      { Create a message box.}
      Reply := MessageBox(HWindow, 'Do you want to save?',
        'Output has changed', mb_YesNoCancel or
          mb_IconQuestion);

      { Check the REPLY.}
      if (Reply = id_Yes) then Print_Groups(groups,
          FileName)
      else if (Reply = id_Cancel) then CanClose := false;
    end; {if}

end;

procedure TSGroupWin.FileOpen(var Msg: TMessage);
{-----------------------------------------------------------
FileOpen: When OPEN is selected from the FILE menu
selection, this procedure presents the user with a
file-dialog box in order to prompt the user for the DOS name
of the file to import.  The file is then read (via
FETCH_SLOTS) and all slots with prompts are added to the
collection of slots, PROMPTS.
-----------------------------------------------------------}
var SlotFile : text;

begin
  { Get the file name.}
  if Application^.ExecDialog(New(PFileDialog,
    Init(@Self, PChar(sd_FileOpen), StrCopy(FileName,
      '*.SLT')))) = id_Ok
    then
      begin
        { Retrieve the appropriate slots.}
        Open_Files(SlotFile, FileName);
        Fetch_Slots(SlotFile, Prompts);
        Close_Files(SlotFile);
      end; {if}

end; {TSGroupWin.FileOpen}

procedure TSGroupWin.FileSave(var Msg: TMessage);
{----------------------------------------------------------
FileSave: As long as the grouping data has been altered
since the last save (MODIFIED = TRUE), this procedure saves
all GROUP data and resets the MODIFIED flag.
----------------------------------------------------------}
begin

  if (Modified) then
    begin
      Print_Groups(groups, FileName);
      Modified := false;
    end; {if}

end; {TSGroupWin.FileSave}

procedure TSGroupWin.GrpCreate(var Msg: TMessage);
{----------------------------------------------------------
GrpCreate: This procedure is invoked when the user selects
the CREATE option of the GROUP menu selection.  A dialog box
is used to prompt the user for the name of the new group to
be created and then creates a new element in the GROUP
collection, with this name (NOM).
----------------------------------------------------------}
var nom : TGName;

begin
  if (not Modified) then Modified := true;
  StrPCopy(nom,'group1');
  if Application^.ExecDialog(New(PInputDialog, Init(@Self,
      'Create Group','Enter Group Name:', nom,
      SizeOf(nom)))) = id_OK then
      Add_Group(@nom, Groups);

end; {TSGroupWin.GrpCreate}

procedure TSGroupWin.GrpDelete(var Msg: TMessage);
{----------------------------------------------------------
GrpDelete: When the DELETE option of the GROUP menu
selection is chosen, this procedure produces a popup window
that prompts the user for the name of the group to be
deleted.  The group by that name is then removed from the
collection of groups
---------------------------------------------------------}
var GrpDelWnd : PWindow;

begin
  if (groups^.Count > 0) then
    begin
      if (not Modified) then Modified := true;
      GrpDelWnd := new(PGrpDelWnd, Init(@Self, 'Group
           Delete', groups));
      Application^.MakeWindow(GrpDelWnd);
    end; {if}

end; {TSGroupWin.GrpDelete}

procedure TSGroupWin.GrpAdd(var Msg: TMessage);
{---------------------------------------------------------
GrpAdd: This procedure is activated when the user selects
the ADD option of the GROUP selection menu.  A popup window
is generated, GRPADDWND, which allows the user to add slots
to a selected group.
---------------------------------------------------------}
var
  GrpAddWnd : PWindow;

begin
  if ((groups^.Count > 0) and (prompts^.Count > 0)) then
    begin
      if (not Modified) then Modified := true;
      GrpAddWnd := new(PGrpAddWnd, Init(@Self, 'Add Slots',
           groups, prompts));
      Application^.MakeWindow(GrpAddWnd);
    end; {if}

end; {TSGroupWin.GrpAdd}

procedure TSGroupWin.GrpRemove(var Msg: TMessage);
{---------------------------------------------------------
GrpRemove: If the user selects the REMOVE option of the
GROUP menu, a popup window is created that prompts the user
to select which group to remove slots from, and
subsequently, which slots to remove.
---------------------------------------------------------}
var
  GrpRmvWnd : PWindow;
```

```
begin
  if ((groups^.Count > 0) and (prompts^.Count > 0)) then
    begin
      if (not Modified) then Modified := true;
      GrpRmvWnd := new(PGrpRmvWnd, Init(@Self, 'Remove
           Slots', groups, prompts));
      Application^.MakeWindow(GrpRmvWnd);
    end; {if}

end; {TSGroupWin.GrpRemove}

procedure TSGroupWin.Help(var Msg: TMessage);
{-------------------------------------------------------------
Help: Selecting the HELP option from the main menu activates
the help system.
-------------------------------------------------------------}
begin
  MessageBox(HWindow, 'Feature not implemented', 'Help',
      mb_Ok);

end; {TSGroupWin.Help}

{------------------------------------------------------}
{ TSGroupApp's method implementations:                 }
{------------------------------------------------------}

procedure TSGroupApp.InitMainWindow;
{-------------------------------------------------------------
InitMainWindow: A primary window is created with the title
"Slot Groups"
-------------------------------------------------------------}
begin
  MainWindow := New(PSGroupWin, Init(nil, 'Slot Groups'));

end; {TSGroupApp.InitMainWindow}

{-------------------------------------------------------------
Main program: The application, with id SGROUP, is started.
-------------------------------------------------------------}

var
  SGroupApp : TSGroupApp;

begin
  SGroupApp.Init('SGroup');
  SGroupApp.Run;
  SGroupApp.Done;

end. {SGroup}
```

<u>SGLOAD.UNT</u>

```
Unit SGLoad;

{$V-} { Turn off type checking for strings.}

Interface
  Uses WObjects, WinDos, Strings, PrmptObj;

  Const
    LINE_MAX        = 255;
    TYPE_MAX        =   2;
    NEWLINE         = chr(13);
    MAX_PROMPTS     = 100;
    PROMPTS_OVERFLOW=  25;

  Type
    TLine     = string[LINE_MAX];
    TLineType = string[TYPE_MAX];
    TLineIndex= 0..LINE_MAX+1;
    TFilename = array [0..fsPathName] of char;

  Procedure Parse_Word(line: TLine; delimiter: char;
              var pword: TLine; var i,j: TLineIndex);
  Procedure Open_Files(var target_file: text; var filename:
              TFilename);
  Procedure Close_Files(var target_file: text);
  Procedure Init_Prompts(var prompt_list: PCollection);
  Function Process_Prompt(line: TLine; i,j: TLineIndex):
              TLine;
  Procedure Add_Prompt(AName, APrompt: TLine; var prompts:
              PCollection);
  Procedure Fetch_Slots(var slot_file: text; var
              prompt_list: PCollection);

Implementation

Procedure Parse_Word(line: TLine; delimiter: char;
                     var pword: TLine; var i,j: TLineIndex);
{ -----------------------------------------------------
Parse_Word: the line index, I, is used to procede character
by character along the line, LINE, from the current position
of I until the delimiter, DELIMITER, is found or a
white-space character is encountered, adding each character
to the string variable, PWORD.  If neither of the previous
conditions are met, the parsing stops when the end of the
line (J) is reached.
Called by: Fetch_Slots, Process_Prompt, Open_Files
---------------------------------------------------------}
```

```
  var quit : boolean;

  begin
    pword := '';

    { Add each character to PWORD until the DELIMITER is
      found or the end of the line is reached or a
      white-space character is encountered.}
    quit := false;
    while ((i <= j) and (not quit)) do
      begin
        quit := ((line[i] = delimiter) or (line[i] < ' ') or
           (line[i] > '~'));
        if (not quit) then
          begin
            pword := pword + line[i];
            inc(i);
          end; {if}
      end; {while}

  end; {Parse_Word}

Procedure Open_Files(var target_file: text; var filename:
TFilename);
{ ---------------------------------------------------------
Open_Files: the input file is opened, using the name,
FILENAME.
Called by: External
Calls: Parse_Word
----------------------------------------------------------}

  var i,j : TLineIndex;

  begin
    { Open the input file.}
    assign(target_file, filename);
    reset(target_file);

  end; {Open_Files}

Procedure Close_Files(var target_file: text);
{ ----------------------------------------------------------
Close_Files: both the input file is closed.
Called by: External
-----------------------------------------------------------}

  begin
    { Close all files.}
    close(target_file);
  end; {Close_Files}

Procedure Init_Prompts(var prompt_list: PCollection);
{ ----------------------------------------------------------
Init_Prompts: the list of prompts to be grouped,
PROMPT_LIST, is set to the empty state.
Called by: External
----------------------------------------------------------}

  begin
    { Initialize the list to the empty state.}
    prompt_list := new(PCollection,
      Init(MAX_PROMPTS,PROMPTS_OVERFLOW));

  end; {Init_Prompts}

Function Process_Prompt(line: TLine; i,j: TLineIndex):
TLine;
{ ----------------------------------------------------------
Process_Prompt: the remainder of the input LINE contains the
prompt for current slot.  This function parses out the
prompt and returns it.
Called by: Fetch_Slots
----------------------------------------------------------}

  var APrompt : TLine;

  begin
    { Extract the remainder of the line.}
    inc(i);
    Parse_Word(line,NEWLINE,APrompt,i,j);

    Process_Prompt := APrompt;

  end; {Process_Prompt}

Procedure Add_Prompt(AName, APrompt: TLine; var prompts:
PCollection);
{ ----------------------------------------------------------
Add_Prompt: a new prompts element is created with the slot,
ANAME, and the prompt, APROMPT.  This new element is then
added to the list of prompts that may be grouped.  The name
and prompt passed to this procedure are Pascal strings,
which must be converted to null-terminated strings for
storing.  BSLOT and BPROMPT act as buffers for this
conversion.
Called by: Fetch_Slots
----------------------------------------------------------}
  var slot, prompt: PChar;
```

```
    Bslot, Bprompt: array [0..LINE_MAX] of char;

begin
  { Convert the Pascal strings to null-terminated
    strings.}
  slot   := StrPCopy(Bslot,AName);
  prompt := StrPCopy(Bprompt,APrompt);

  { Create a new prompt entry.}
  prompts^.insert(new(PPrompt, Init(slot,prompt)));

end; {Add_Prompt}
```

```
Procedure Fetch_Slots(var slot_file: text; var prompt_list:
                      PCollection);
{ -----------------------------------------------------------
Fetch_Slots: each slot containing a prompt entry (indicated
by a value of 'PR' in LINE_TYPE) is added to the list of
slots that may be grouped.  LINE, I, and J are used to hold
the current input line and parsing data.  The slots are
contained in the file, SLOT_FILE.
Called by: External
Calls: Parse_Word, Process_Prompt, Add_Prompt
-----------------------------------------------------------}

  var line        : TLine;
      i,j         : TLineIndex;
      slot_name   : TLine;
      slot_prompt : TLine;
      line_type   : TLineType;

  begin
    { Discard the first line.}
    readln(slot_file);

    while (not eof(slot_file)) do
      begin
        { Fetch the name of the slot.}
        readln(slot_file, line);
        i := 1; j := length(line);
        Parse_Word(line,' ',slot_name,i,j);

        { Fetch the line-type of the next line in the file.}
        readln(slot_file, line);
        i := 1; j := length(line);
        Parse_Word(line,' ',line_type,i,j);

        { If the line-type is PRompt, then extract the
          prompt and add this slot to the list of prompts
          that may be grouped.}
        if (line_type = 'PR') then
          begin
            slot_prompt := Process_Prompt(line,i,j);
            Add_Prompt(slot_name, slot_prompt, prompt_list);
          end; {if}

        { Advance to the next record or end of the file.}
        while ((not eof(slot_file)) and (line <> '')) do
          readln(slot_file,line);
      end; {while}
  end; {Fetch_Slots}
End. { SGLoad}
```

<u>PRMPTOBJ.UNT</u>

```
Unit PrmptObj;

Interface
  uses WObjects, WinTypes, WinProcs, Strings;

  Type
    PPrompt = ^TPrompt;
    TPrompt = object(TObject)
      { Attributes}
      Slot      : PChar;
      Prompt    : PChar;
      Available : boolean;

      { Methods}
      constructor Init(ASlot, APrompt: PChar);
      destructor  Done; virtual;
      function    GetSlot   : PChar; virtual;
      function    GetPrompt: PChar; virtual;
      function    GetAvail : boolean; virtual;
      Procedure   SetSlot(ASlot: PChar); virtual;
      Procedure   SetPrompt(APrompt: PChar); virtual;
      Procedure   SwitchAvail; virtual;
    end; {TPrompt}

Implementation
Constructor TPrompt.Init(ASlot, APrompt: PChar);
{-------------------------------------------------------
TPrompt.Init: This is the constructor for the TPrompt
object.  All attributes are assigned values.  The group
indicator is set to the default value of 0.
-------------------------------------------------------}
  begin
    Slot   := StrNew(ASlot);
    Prompt := StrNew(APrompt);
    Available := true;

  end; {TPrompt.Init}

Destructor TPrompt.Done;
{-------------------------------------------------------
TPrompt.Done: This is the destructor for the TPrompt object.
The string attributes are disposed of.
-------------------------------------------------------}
  begin
    StrDispose(Slot);
    StrDispose(Prompt);
  end; {TPrompt.Done}
```

```
Function TPrompt.GetSlot: PChar;
{-----------------------------------------------------------
TPrompt.GetSlot: Returns the Slot value for the TPrompt
object.
-----------------------------------------------------------}
  begin
    GetSlot := Slot;

  end; {TPrompt.GetSlot}

Function TPrompt.GetPrompt: PChar;
{-----------------------------------------------------------
TPrompt.GetPrompt: Returns the Prompt value for the TPrompt
object.
-----------------------------------------------------------}
  begin
    GetPrompt := Prompt;

  end; {TPrompt.GetPrompt}

Function TPrompt.GetAvail: boolean;
{-----------------------------------------------------------
TPrompt.GetAvail: Returns the GROUP value for the TPrompt
object.
-----------------------------------------------------------}
  begin
    GetAvail := Available;

  end; {TPrompt.GetAvail}

Procedure TPrompt.SetSlot(ASlot: PChar);
{-----------------------------------------------------------
TPrompt.SetSlot: The Slot value for the TPrompt object is
assigned the value, ASLOT.
-----------------------------------------------------------}
  begin
    Slot := StrNew(ASlot);

  end; {TPrompt.SetSlot}

Procedure TPrompt.SetPrompt(APrompt: PChar);
{-----------------------------------------------------------
TPrompt.SetPrompt: The Prompt value for the TPrompt object
is assigned the value, APROMPT
-----------------------------------------------------------}
  begin
    Prompt := StrNew(APrompt);
  end; {TPrompt.SetPrompt}
```

```
Procedure TPrompt.SwitchAvail;
{-----------------------------------------------------------
TPrompt.SwitchAvail: The AVAILABLE flag of the TPrompt
object is toggled.
 ----------------------------------------------------------}
  begin
    Available := not Available;

  end; {TPrompt.SwitchAvail}

End.
```

<u>GROUP.UNT</u>

```
Unit Group;

Interface

  uses Strings, WinDos, WinProcs, WObjects, GrpObj;

  const
    MAX_GROUPS      = 25;
    GROUPS_OVERFLOW = 10;

Procedure Init_Groups(var group_list: PCollection);
Procedure Add_Group(name: PChar; var groups: PCollection);
Procedure Add_Slot(ASlot: PChar; var AGroup: PGroup);
Procedure Print_Groups(groups: PCollection; FileName:
PChar);

Implementation

Procedure Init_Groups(var group_list: PCollection);
{ -----------------------------------------------------------
Init_Groups: the list of groups, GROUP_LIST, is initialized
to the empty state.
Called by: External
-----------------------------------------------------------}

  begin
    { Initialize the list to the empty state.}
    group_list := new(PCollection,
Init(MAX_GROUPS,GROUPS_OVERFLOW));

  end; {Init_Groups}

Procedure Add_Group(name: PChar; var groups: PCollection);
{ -----------------------------------------------------------
Add_Group: A new group item is added to the collection of
groups, giving it the name, NAME.
Called by: External
-----------------------------------------------------------}

  begin
    { Create a new prompt entry.}
    groups^.insert(new(PGroup, Init(name)));

  end; {Add_Group}
```

```
Procedure Add_Slot(ASlot: PChar; var AGroup: PGroup);
{ -----------------------------------------------------------
Add_Group: A new group item is added to the collection of
groups, giving it the name, NAME.
Called by: External
------------------------------------------------------------}

  begin
    { Create a new prompt entry.}
    AGroup^.AddSlot(ASlot);

  end; {Add_Group}

Procedure Print_Groups(groups: PCollection; FileName:
                       PChar);
{ -----------------------------------------------------------
Print_Groups: All group names and the list of slots
associated with them are written to the text file, OUTFILE.
Called by: External
------------------------------------------------------------}
  var OutFile: text;
      PasFileName: string[fsPathName];
      dot : integer;

  procedure PrintOut(g: PGroup); far;

    begin
      g^.WriteGrp(OutFile);
    end;

  begin
    { Derive the output file name from the input file name.}
    PasFileName := StrPas(FileName);
    dot := Pos('.', PasFileName);

    if (dot <> 0) then
      PasFileName := Copy(PasFileName, 1, dot) + 'sgp'
    else
      PasFileName := PasFileName + '.sgp';

    { Open the output file.}
    assign(OutFile, PasFileName);
    rewrite(OutFile);
    { Print each group.}
    groups^.ForEach(@PrintOut);
    { Close the output file.}
    close(OutFile);
  end; {Add_Group}
End.
```

GRPOBJ.UNT

```
Unit GrpObj;

Interface
  uses Strings, WinTypes, WinProcs, WObjects;
  const
    GNAME_MAX      =  50;
    MAX_SLOTS      =  25;
    SLOTS_OVERFLOW =  10;

  type
    TGName  = array[0..GNAME_MAX] of char;
    PPGroup = ^PGroup;
    PGroup  = ^TGroup;
    TGroup  = object(TObject)
      { Attributes}
      name  : PChar;
      slots : PCollection;

      { Methods}
      constructor Init(AName: PChar);
      destructor  Done; virtual;
      function    GetName : PChar; virtual;
      procedure   SetName(AName: PChar); virtual;
      procedure   AddSlot(ASlot: PChar); virtual;
      procedure   DeleteSlot(Idx: Integer); virtual;
      procedure   WriteGrp(var OutFile: text);
    end; {TPrompt}

Implementation

Constructor TGroup.Init(AName: PChar);
{---------------------------------------------------------
TGroup.Init: This is the constructor for the TGroup object.
All attributes are assigned values.  The collection of slots
is set to the empty state.
--------------------------------------------------------}
  begin
    { Set the group name.}
    name := StrNew(AName);

    { Initialize the list to the empty state.}
    slots := new(PCollection,
                Init(MAX_SLOTS,SLOTS_OVERFLOW));

  end; {TGroup.Init}
```

```
Destructor TGroup.Done;
{-----------------------------------------------------------
TPrompt.Done: This is the destructor for the TGroup object.
The string attribute, NAME, is disposed of.
-----------------------------------------------------------}
  begin
    StrDispose(name);

  end; {TGroup.Done}

Function TGroup.GetName: PChar;
{-----------------------------------------------------------
TGroup.GetName: Returns the group NAME for the TGroup
object.
-----------------------------------------------------------}
  begin
    GetName := name;

  end; {TGroup.GetName}

Procedure TGroup.SetName(AName: PChar);
{-----------------------------------------------------------
TGroup.SetName: Sets the NAME attribute of the TGroup
object.
-----------------------------------------------------------}
  begin
    name := StrNew(AName);

  end; {TGroup.SetName}

Procedure TGroup.AddSlot(ASlot: PChar);
{-----------------------------------------------------------
TGroup.AddSlot: Adds a new slot, ASLOT, to the collection of
slots.
-----------------------------------------------------------}
  begin
    slots^.insert(StrNew(ASlot));

  end; {TGroup.AddSlot}

Procedure TGroup.DeleteSlot(Idx: integer);
{-----------------------------------------------------------
TGroup.DeleteSlot: Removes the slot indexed by, IDX, from
the collection of slots.
-----------------------------------------------------------}
  begin
    slots^.AtDelete(Idx);

  end; {TGroup.DeleteSlot}
```

```
Procedure TGroup.WriteGrp(var OutFile: text);
{---------------------------------------------------------
TGroup.WriteGrp: The name of the group is written to the
file, OutFile.
---------------------------------------------------------}
  var OutName : string[GNAME_MAX];

  procedure PrintSlot(s: PChar); far;

    var
      SlotName : string[255];

    begin
      SlotName := StrPas(s);
      writeln(OutFile, SlotName);

    end; {PrintSlot}

  begin
    { Write the group name.}
    OutName := StrPas(name);
    writeln(OutFile, OutName);

    { Write each slot name associated with this group.}
    slots^.ForEach(@PrintSlot);

    { Write a separator line.}
    writeln(OutFile);

  end; {TGroup.DeleteSlot}

End.
```

<u>GDWIND.UNT</u>

```
Unit GDWind;

Interface

  uses WObjects, WinTypes, WinProcs, GrpObj;

  const
    id_LB1 = 301;
    id_BN1 = 302;
    id_BN2 = 303;

    GRPDEL_XPOS   = 100;
    GRPDEL_YPOS   = 100;
    GRPDEL_WIDTH  = 300;
    GRPDEL_HEIGHT = 200;

  type
    PGrpDelWnd = ^TGrpDelWnd;
    TGrpDelWnd = object (TWindow)
      { Attributes}
      LB1    : PListBox;
      GrpList: PCollection;

      { Methods}
      constructor Init(AParent: PWindowsObject; ATitle:
                       PChar; AGrpList: PCollection);
      procedure SetupWindow; virtual;
      procedure IDBN1(var Msg: TMessage); virtual id_First +
                      id_BN1;
      procedure IDBN2(var Msg: TMessage); virtual id_First +
                      id_BN2;
    end; {TGrpDelWnd}

Implementation

Constructor TGrpDelWnd.Init(AParent: PWindowsObject; ATitle:
                PChar; AGrpList: PCollection);
{-------------------------------------------------------
Init: this is the constructor for the TGrpDelWnd object.  It
creates a pop-up window containing a list box and two
buttons.  Also, the GRPLIST attribute is assigned the list
passed as a parameter, AGRPLIST.
-------------------------------------------------------}
  const
    LB_X     = 20;   BN1_X     = 220;  BN2_X     = 220;
    LB_Y     = 30;   BN1_Y     =  20;  BN2_Y     =  70;
    LB_WIDTH = 150;  BN1_WIDTH =  60;  BN2_WIDTH =  60;
    LB_HEIGHT = 100;  BN1_HEIGHT =  30;  BN2_HEIGHT =  30;

  var
    Btn : PButton;

  begin
    TWindow.Init(AParent, ATitle);
    DisableAutoCreate;

    { Set the attributes for this popup window.}
    Attr.Style := ws_PopupWindow or ws_Caption or
           ws_Visible;
    Attr.X := GRPDEL_XPOS;
    Attr.Y := GRPDEL_YPOS;
    Attr.W := GRPDEL_WIDTH;
    Attr.H := GRPDEL_HEIGHT;

    { Create the list box.}
    LB1 := new(PListBox, Init(@Self, id_LB1, LB_X, LB_Y,
           LB_WIDTH, LB_HEIGHT));
    LB1^.Attr.Style := LB1^.Attr.Style and not lbs_Sort;

    { Create the two buttons (OK and CANCEL).}
    Btn := new(PButton, Init(@Self, id_BN1, 'OK', BN1_X,
           BN1_Y, BN1_WIDTH, BN1_HEIGHT, true));
    Btn := new(PButton, Init(@Self, id_BN2, 'CANCEL', BN2_X,
           BN2_Y, BN2_WIDTH, BN2_HEIGHT, false));

    { Assign the group list.}
    GrpList := AGrpList;

  end; {TGrpDelWnd.Init}

Procedure TGrpDelWnd.SetupWindow;
{-------------------------------------------------------
SetupWindow: This procedure sets up the list box by adding
all the elements of the collection, GRPLIST, to the list box
list of selections.
-------------------------------------------------------}
  procedure AddList(g: PGroup); far;

    begin
      { Add the group name to the list of selections.}
      LB1^.AddString(g^.name);

    end; {AddList}

  begin
    TWindow.SetupWindow;
    { Add each element of GRPLIST collection to list box.}
    GrpList^.ForEach(@AddList);

  end; {TGrpDelWnd.SetupWindow}

Procedure TGrpDelWnd.IDBN1(var Msg: TMessage);
{---------------------------------------------------------
IDBN1: If the user presses the 'OK' button, then the current
selection in the list box (SELINDEX) is used as an index to
the collection of of groups (GRPLIST), and the indexed group
is deleted.  The window then closes.
---------------------------------------------------------}
  var
    SelIndex : integer;
  begin
    SelIndex := LB1^.GetSelIndex;
    GrpList^.AtDelete(SelIndex);
    CloseWindow;

  end; {TGrpDelWnd.IDBN1}

Procedure TGrpDelWnd.IDBN2(var Msg: TMessage);
{---------------------------------------------------------
IDBN2: If the 'CANCEL' button is pressed, the window is
closed and no other action is taken.
---------------------------------------------------------}
  begin
    CloseWindow;

  end; {TGrpDelWnd.IDBN2}

End. {GDWind}
```

GAWIND.UNT

```
Unit GAWind;

Interface

  uses WObjects, WinTypes, WinProcs, GrpObj, PrmptObj,
       SGrpDS, Strings;

  const
    id_LB1 = 401;
    id_LB2 = 402;
    id_BN1 = 403;
    id_BN2 = 404;

    GRPADD_XPOS   = 100;
    GRPADD_YPOS   = 100;
    GRPADD_WIDTH  = 500;
    GRPADD_HEIGHT = 300;

  type
    PGrpAddWnd = ^TGrpAddWnd;
    TGrpAddWnd = object (TWindow)
      { Attributes}
      LB1       : PListBox;
      LB2       : PListBox;
      GrpList   : PCollection;
      PrmptList: PCollection;

      { Methods}
      constructor Init(AParent: PWindowsObject; ATitle:
          PChar; AGrpList: PCollection; APrmptList:
          PCollection);
      procedure SetupWindow; virtual;
      procedure IDBN1(var Msg: TMessage); virtual id_First +
           id_BN1;
      procedure IDBN2(var Msg: TMessage); virtual id_First +
           id_BN2;
    end; {TGrpAddWnd}

Implementation

{----------------------------------------------------------
TGrpAddWnd's method implementations.
----------------------------------------------------------}

Constructor TGrpAddWnd.Init(AParent: PWindowsObject; ATitle:
     PChar; AGrpList: PCollection; APrmptList: PCollection);
{----------------------------------------------------------
Init: this is the constructor for the TGrpAddWnd object.  It
creates a pop-up window containing a list box and two
buttons.  Also, the GRPLIST attribute is assigned the list
passed as a parameter, AGRPLIST, and the prompt list,
PRMPTLIST, is assigned the passed parameter, APRMPTLIST.
-------------------------------------------------------------}
  const
    LB1_X      = 20;   BN1_X      = 350;  BN2_X      = 350;
    LB1_Y      = 30;   BN1_Y      =  30;  BN2_Y      =  80;
    LB1_WIDTH  = 300;  BN1_WIDTH  =  60;  BN2_WIDTH  =  60;
    LB1_HEIGHT = 100;  BN1_HEIGHT =  30;  BN2_HEIGHT =  30;

    LB2_X      =  20;
    LB2_Y      = 150;
    LB2_WIDTH  = 460;
    LB2_HEIGHT = 100;

  var
    Btn : PButton;

  begin
    TWindow.Init(AParent, ATitle);
    DisableAutoCreate;

    { Set the attributes for this popup window.}
    Attr.Style := ws_PopupWindow or ws_Caption or
      ws_Visible;
    Attr.X := GRPADD_XPOS;
    Attr.Y := GRPADD_YPOS;
    Attr.W := GRPADD_WIDTH;
    Attr.H := GRPADD_HEIGHT;

    { Create the list boxes.}
    LB1 := new(PListBox, Init(@Self, id_LB1, LB1_X, LB1_Y,
      LB1_WIDTH,LB1_HEIGHT));
    LB1^.Attr.Style := LB1^.Attr.Style and not lbs_Sort;
    LB2 := new(PListBox, Init(@Self, id_LB2, LB2_X, LB2_Y,
      LB2_WIDTH, LB2_HEIGHT));
    LB2^.Attr.Style := LB2^.Attr.Style and not lbs_Sort
                         or lbs_MultipleSel;

    { Create the two buttons (OK and CANCEL).}
    Btn := new(PButton, Init(@Self, id_BN1, 'OK', BN1_X,
      BN1_Y, BN1_WIDTH, BN1_HEIGHT, true));
    Btn := new(PButton, Init(@Self, id_BN2, 'CANCEL', BN2_X,
      BN2_Y, BN2_WIDTH, BN2_HEIGHT, false));

    { Assign the group and prompt lists and the group
      selection.}
    GrpList := AGrpList;
    PrmptList := APrmptList;

  end; {TGrpAddWnd.Init}

Procedure TGrpAddWnd.SetupWindow;
{----------------------------------------------------------
SetupWindow: This procedure sets up the two list boxes, LB1
& LB2.  Into LB1, all group names in the list of groups,
GRPLIST, are inserted.  Into LB2, all prompts from the list
of prompts, PRMPTLIST, are inserted.
---------------------------------------------------------}
  procedure AddGrpList(g: PGroup); far;

    begin
      { Add the group name to the list of selections.}
      LB1^.AddString(g^.name);

    end; {AddGrpList}

  procedure AddPrmptList(p: PPrompt); far;

    var
      temp : array [0..255] of char;

    begin
      { Add the slot prompt to the list of selections.}
      if (p^.GetAvail) then LB2^.AddString(p^.Prompt)
      else
        begin
          StrCopy(temp, '*');
          StrCat(temp, p^.Prompt);
          LB2^.AddString(temp);
        end; {else}

    end; {AddPrmptList}

  begin
    TWindow.SetupWindow;

    { Add each element of the GRPLIST collection to the list
      box.}
    GrpList^.ForEach(@AddGrpList);
    LB1^.SetSelIndex(0);

    { Add each element of the PRMPTLIST collection to the
      list box.}
    PrmptList^.ForEach(@AddPrmptList);
  end; {TGrpAddWnd.SetupWindow}
```

```
Procedure TGrpAddWnd.IDBN1(var Msg: TMessage);
{----------------------------------------------------------
IDBN1: If the 'OK' button is pressed, the list of selections
made in the list box, LB2, which lists the prompts, is
transfered to a transfer buffer, TRANSBUF.  All selections
are then used as indices to locate the slot names
corresponding to the prompts selected.  These slot names are
then added to the list of SLOTS associated with the GROUP
that was selected in list box, LB1.  I & J are used as
temporary integer holders for traversing the array of
selections.  Also, after each selection is used to add the
slot names to the SLOTS list, the corresponding prompt is
deleted from the prompt list, PRMPTLIST.
---------------------------------------------------------}
  var
    TransBuf : PMultiSelXferRec;
    Group    : PGroup;
    Prompt   : PPrompt;
    temp     : array [0..255] of char;
    i,j      : integer;

  begin
    { Create a transfer buffer.}
    New(TransBuf);
    TransBuf^.List := New(PStrCollection, Init(100,25));
    TransBuf^.MultiSelRec := AllocMultiSel(LB2^.GetCount);

    { Perform the transfer.}
    LB2^.Transfer(TransBuf, tf_GetData);

    if (TransBuf^.MultiSelRec <> nil) then
    with TransBuf^ do
      begin
        { Determine the selected group.}
        Group := GrpList^.At(LB1^.GetSelIndex);
        i := MultiSelRec^.Count;
        j := 0;

        { Add the slot name corresponding to each selected
          prompt to list of slots associated with GROUP.}
        while (j < i) do
          begin
            Prompt :=
                PrmptList^.At(MultiSelRec^.Selections[j]);
            if (Prompt^.GetAvail) then
              begin
                Group^.AddSlot(Prompt^.GetSlot);
                Prompt^.SwitchAvail;
              end; {if}
            inc(j);
          end; {while}

      end; {with}

    { Free the memory required for the transfer buffer.}
    FreeMultiSel(TransBuf^.MultiSelRec);
    Dispose(TransBuf^.List, Done);
    Dispose(TransBuf);

    { Close the window.}
    CloseWindow;

  end; {TGrpAddWnd.IDBN1}

Procedure TGrpAddWnd.IDBN2(var Msg: TMessage);
{----------------------------------------------------------
IDBN2: If the 'CANCEL' button is pressed, then the window is
closed, and no further action is taken.
----------------------------------------------------------}
  begin
    { Close the window.}
    CloseWindow;

  end; {TGrpAddWnd.IDBN2}

End.
```

GRWIND.UNT

```
Unit GRWind;

Interface

  uses Strings, WObjects, WinTypes, WinProcs, GrpObj,
       PrmptObj, SGrpDS;

  const
    id_LB1 = 501;
    id_LB2 = 502;
    id_BN1 = 503;
    id_BN2 = 504;

    GRPRMV_XPOS   = 100;
    GRPRMV_YPOS   = 100;
    GRPRMV_WIDTH  = 500;
    GRPRMV_HEIGHT = 300;

  type
    PGrpRmvWnd = ^TGrpRmvWnd;
    TGrpRmvWnd = object (TWindow)
      { Attributes}
      LB1       : PListBox;
      LB2       : PListBox;
      GrpList   : PCollection;
      PrmptList: PCollection;

      { Methods}
      constructor Init(AParent: PWindowsObject; ATitle:
           PChar; AGrpList, APrmptList: PCollection);
      procedure SetupWindow; virtual;
      procedure IDLB1(var Msg: TMessage); virtual id_First +
           id_LB1;
      procedure IDBN1(var Msg: TMessage); virtual id_First +
           id_BN1;
      procedure IDBN2(var Msg: TMessage); virtual id_First +
           id_BN2;
    end; {TGrpRmvWnd}

Implementation

{-------------------------------------------------------
TGrpRmvWnd's method implementations.
-------------------------------------------------------}
```

```
Constructor TGrpRmvWnd.Init(AParent: PWindowsObject; ATitle:
      PChar; AGrpList, APrmptList: PCollection);
{----------------------------------------------------------
Init: this is the constructor for the TGrpRmvWnd object.  It
creates a pop-up window containing a list box and two
buttons.  Also, the GRPLIST attribute is assigned the list
passed as a parameter, AGRPLIST.
-----------------------------------------------------------}
  const
    LB1_X      = 20;    BN1_X      = 350;  BN2_X      = 350;
    LB1_Y      = 30;    BN1_Y      =  30;  BN2_Y      =  80;
    LB1_WIDTH  = 300;   BN1_WIDTH  =  60;  BN2_WIDTH  =  60;
    LB1_HEIGHT = 100;   BN1_HEIGHT =  30;  BN2_HEIGHT =  30;

    LB2_X      =  20;
    LB2_Y      = 150;
    LB2_WIDTH  = 460;
    LB2_HEIGHT = 100;

  var
    Btn : PButton;

  begin
    TWindow.Init(AParent, ATitle);
    DisableAutoCreate;

    { Set the attributes for this popup window.}
    Attr.Style := ws_PopupWindow or ws_Caption or
      ws_Visible;
    Attr.X := GRPRMV_XPOS;
    Attr.Y := GRPRMV_YPOS;
    Attr.W := GRPRMV_WIDTH;
    Attr.H := GRPRMV_HEIGHT;

    { Create the list boxes.}
    LB1 := new(PListBox, Init(@Self, id_LB1, LB1_X, LB1_Y,
      LB1_WIDTH, LB1_HEIGHT));
    LB1^.Attr.Style := LB1^.Attr.Style and not lbs_Sort;
    LB2 := new(PListBox, Init(@Self, id_LB2, LB2_X, LB2_Y,
      LB2_WIDTH, LB2_HEIGHT));
    LB2^.Attr.Style := LB2^.Attr.Style and not lbs_Sort
                          or lbs_MultipleSel;

    { Create the two buttons (OK and CANCEL).}
    Btn := new(PButton, Init(@Self, id_BN1, 'OK', BN1_X,
      BN1_Y, BN1_WIDTH, BN1_HEIGHT, true));
    Btn := new(PButton, Init(@Self, id_BN2, 'CANCEL', BN2_X,
      BN2_Y, BN2_WIDTH, BN2_HEIGHT, false));
    { Assign the group list.}
    GrpList   := AGrpList;
    PrmptList := APrmptList;

  end; {TGrpRmvWnd.Init}

Procedure TGrpRmvWnd.SetupWindow;
{----------------------------------------------------------
SetupWindow: This procedure adds all group names to list
box, LB1.  LB2, on the other hand, is left empty, until a
group is selected.
----------------------------------------------------------}
  procedure AddGrpList(g: PGroup); far;

    begin
      { Add the group name to the list of selections.}
      LB1^.AddString(g^.name);

    end; {AddGrpList}

  begin
    TWindow.SetupWindow;

    { Add each element of the GRPLIST collection to the list
      box.}
    GrpList^.ForEach(@AddGrpList);

  end; {TGrpRmvWnd.SetupWindow}

Procedure TGrpRmvWnd.IDLB1(var Msg: TMessage);
{----------------------------------------------------------
IDLB1: If a message is sent from list box, LB1, this
procedure intercepts it, and if the message indicates that
there has been a change in the users selection, the list
box, LB2, is cleared, and a list of slot names associated
with the group, whose name has been selected in LB1, are
displayed in list box, LB2.  I indicates which group name
has been selected; GRP is a pointer to the selected group.
----------------------------------------------------------}
  var
    i : integer;
    Grp : PGroup;

  procedure AddList(s: PChar); far;

    var
      i : integer;
      p : PPrompt;
      found : boolean;
    begin
      { Add the prompt, corresponding to the slot-name, to
        LB2.}
      i := 0; found := false;
      { Search for a match to the slot-name in the PRoMPT
        LIST.}
      while ((i < PrmptList^.Count) and (not found)) do
        begin
          p := PrmptList^.At(i);
          found := StrComp(p^.slot, s) = 0;
          if (not found) then inc(i);
        end; {while}

      if (found) then LB2^.AddString(p^.prompt);

    end; {AddList}

  begin
    { Is it a selection change?}
    if (Msg.lParamHi = lbn_SelChange) then
      begin
        { Find the index of the selected group.}
        i := LB1^.GetSelIndex;
        LB2^.ClearList;
        { Use the index to find the actual group.}
        Grp := GrpList^.At(i);
        { Add all slot names associated with this group to
          LB2.}
        Grp^.Slots^.ForEach(@AddList);
      end {if}

    else DefWndProc(Msg);

  end; {TGrpRmvWnd.IDLB1}

Procedure TGrpRmvWnd.IDBN1(var Msg: TMessage);
{----------------------------------------------------------
IDBN1: If the 'OK' button is pressed, the selections made in
list box, LB2, are transfered to the transfer buffer,
TRANSBUF.  These selections are then used as indices to
remove the selected slots from the list of slots associated
with the GROUP selected in list box, LB1.  I and J are used
to examine the selections, in the transfer buffer,
sequentially.
----------------------------------------------------------}
  var
    TransBuf : PMultiSelXferRec;
    Group    : PGroup;
    p        : PPrompt;
    i,j,k    : integer;
    found    : boolean;

  begin
    { Create a transfer buffer.}
    New(TransBuf);
{*} TransBuf^.List := New(PStrCollection, Init(100,25));
    TransBuf^.MultiSelRec := AllocMultiSel(LB2^.GetCount);

    { Perform the transfer.}
    LB2^.Transfer(TransBuf, tf_GetData);

    if (TransBuf^.MultiSelRec <> nil) then
    with TransBuf^ do
      begin
        { Determine the selected group.}
        Group := GrpList^.At(LB1^.GetSelIndex);
        i := MultiSelRec^.Count;
        j := 0;

        { Remove all the selected slots from the list of
          slots.}
        while (j < i) do
          begin
            k := 0; found := false;
            { Search for a match to the slot-name in the
              PRoMPT LIST.}
            while ((k < PrmptList^.Count) and (not found))
              do begin
                p := PrmptList^.At(k);
                found := StrComp(p^.slot, Group^.Slots^.At(
                         MultiSelRec^.Selections[j])) = 0;
                if (not found) then inc(k);
              end; {while}
            if (found) then p^.SwitchAvail;
            Group^.DeleteSlot(MultiSelRec^.Selections[j]);
            inc(j);
          end; {while}

      end; {with}

    { Free the memory required for the transfer buffer.}
    FreeMultiSel(TransBuf^.MultiSelRec);
    Dispose(TransBuf^.List, Done);
    Dispose(TransBuf);

    { Close the window.}
    CloseWindow;
  end; {TGrpRmvWnd.IDBN1}

Procedure TGrpRmvWnd.IDBN2(var Msg: TMessage);
{----------------------------------------------------------
IDBN2: If the 'CANCEL' button is pressed, then the window is
closed, and no further action is taken.
-----------------------------------------------------------}
  begin
    { Close the window.}
    CloseWindow;

  end; {TGrpRmvWnd.IDBN2}

End. {GRWind}
```

SGRPDS.UNT

```
Unit SGrpDS;

Interface

  Uses WObjects;

  Type
    PMultiSelXferRec = ^TMultiSelXferRec;
    TMultiSelXferRec = record
      List : PCollection;
      MultiSelRec : PMultiSelRec;
    end; {TMultiSelXferRec}

Implementation

End. {SGrpDS}
```

NEXMACH

EXPERT.PAS

```
program Expert(input,output);

uses
WinTypes, WinProcs, WObjects, Strings, TSlotGrp, SlotList,
PropList, ClassObj, ClsList, ObtList, RuleLoad, GblLoad,
GrpObj, QuesWind, SlotObj, BakChain, WinCrt;

{$R TEST.RES}

const
  MAIN_MENU    = 100;   { Main menu id.}
  MAX_GRPS     =  25;
  GRPS_OVRFLOW =   5;

  cm_Open = 101;   { These constants define the id numbers}
  cm_Save = 102;   {   for the various menu selections. }
  cm_Ques = 200;
  cm_Help = 300;

type
  TExpertApp = object(TApplication)
    procedure InitMainWindow; virtual;
  end;

  { The Main window.}
  PExpertWin = ^TExpertWin;
  TExpertWin = object(TWindow)
    { Attributes}
    Groups : PSlotGrps;
    Props  : PPropList;
    Classes: PClassList;
    Objts  : PObjtList;
    Slots  : PSlotList;
    Rules  : PCollection;
    Globals: TGlobal;
    SugList: string;
    ConclusionStack: PConclusionStack;

    { Methods}
    constructor Init(AParent: PWindowsObject; ATitle:
      PChar);
    destructor Done; virtual;
    function CanClose: Boolean; virtual;
    procedure WMCommand (var Msg: TMessage);
      virtual wm_First + wm_Command;
    procedure FileOpen(var Msg: TMessage);
      virtual cm_First + cm_Open;
    procedure FileSave(var Msg: TMessage);
      virtual cm_First + cm_Save;
    procedure Help(var Msg: TMessage);
      virtual cm_First + cm_Help;
  private
    procedure UnCheckQuestions; virtual;
    procedure EnableQuestions; virtual;
  end; {TExpertWin}

{-------------------------------------------------}
{ TExpertWin's method implementations:           }
{-------------------------------------------------}

constructor TExpertWin.Init(AParent: PWindowsObject; ATitle:
      PChar);
{-------------------------------------------------------------
Init: this is the constructor for the TExpertWin object.  It
simply calls the TWindow constructor, loads the main menu
(100), retrieves the slot-group data, the Property, Object,
Slot.
-------------------------------------------------------------}
begin
  TWindow.Init(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, PChar(MAIN_MENU));

  { Retrieve all data from disk files into the appropriate
     lists.}
  Groups := new(PSlotGrps, Init(MAX_GRPS, GRPS_OVRFLOW));
  Props  := new(PPropList, Init(MAX_PROPS, PROPS_OVRFLOW));
  Classes:= new(PClassList, Init(MAX_CLASS, CLASS_OVRFLOW));
  Objts  := new(PObjtList, Init(MAX_OBTS, OBTS_OVRFLOW,
     Props, Classes));
  Slots  := new(PSlotList, Init(MAX_SLOTS, SLOTS_OVRFLOW));
  FetchRules(Rules);
  FetchGlobals(Globals, SugList);
  UnCheckQuestions;
  ConclusionStack := new(PConclusionStack, Init(SugList,
      Props, Classes, Objts, Rules, Slots));
  if (not ConclusionStack^.BackChain) then EnableQuestions;

end; {TExpertWin.Init}
```

```
destructor TExpertWin.Done;
{-------------------------------------------------------
Done: this is the destructor for the TExpertWin object.  All
lists are deallocated.
-------------------------------------------------------}
begin
  TWindow.Done;
  dispose(ConclusionStack, done);
  dispose(Groups, done);
  dispose(Props, done);
  dispose(Classes, done);
  dispose(Objts, done);
  dispose(Slots, done);
  dispose(Rules, done);

end; {TSGroup.Done}

function TExpertWin.CanClose: Boolean;
{-------------------------------------------------------
CanClose: If the current data has not been saved since the
last change, the user is asked if he/she wishes to save
before exiting, or cancel the exit command.
-------------------------------------------------------}
var
  Reply: Integer;

begin
  CanClose := true;

  { Create a message box.}
  Reply := MessageBox(HWindow, 'Do you want to save?',
    'Output has changed', mb_YesNoCancel or
      mb_IconQuestion);

  { Check the REPLY.}
  if (Reply = id_Yes) then CanClose := false
  else if (Reply = id_Cancel) then CanClose := false;

end;

procedure TExpertWin.WMCommand(var Msg: TMessage);
{-------------------------------------------------------
WMCommand: Every time the user selects a menu item, this
procedure is invoked.  It checks the command id (Msg.wParam)
to see if it is in the range that has been reserved for the
Question submenu.  If it is within this range, a Question
popup window is created which will display the prompts
associated with the selected group.
-------------------------------------------------------}
```

```
var
  QuesWnd : PWindow;
  Grp : PGroup;

begin
  TWindowsObject.WMCommand(Msg);

  { Is the message id in the Question range?}
  if ((Msg.wParam > cm_Ques) and (Msg.wParam < cm_Ques+100))
    then begin
      { Fetch the group selected.}
      Grp := Groups^.At(Msg.wParam - 201);

      { Create the Question Window.}
      QuesWnd := new(PQuesWindow, Init(@Self, 'Questions',
           Grp,Props,Objts,Rules,Slots,ConclusionStack));
      Application^.MakeWindow(QuesWnd);
      if (not ConclusionStack^.Backchain) then
           EnableQuestions;
    end; {if}

end; {TExpertWin.WMCommand}

procedure TExpertWin.FileOpen(var Msg: TMessage);
{----------------------------------------------------------
FileOpen:
---------------------------------------------------------}

begin
  MessageBox(HWindow, 'Feature not implemented', 'FileOpen',
      mb_Ok);
end; {TExpertWin.FileOpen}

procedure TExpertWin.FileSave(var Msg: TMessage);
{----------------------------------------------------------
FileSave:
---------------------------------------------------------}

begin
  MessageBox(HWindow, 'Feature not implemented', 'FileSave',
      mb_Ok);

end; {TExpertWin.FileSave}

procedure TExpertWin.Help(var Msg: TMessage);
{----------------------------------------------------------
Help: Selecting the HELP option from the main menu activates
the help system.
---------------------------------------------------------}
```

```
begin
  MessageBox(HWindow, 'Feature not implemented', 'Help',
mb_Ok);

end; {TExpertWin.Help}

procedure TExpertWin.UnCheckQuestions;
{---------------------------------------------------------
UnCheckQuestions: Each of the items in the Questions submenu
is unchecked.
---------------------------------------------------------}
var
  Idx : word;

begin
  Idx := 1;
  repeat
    begin
      CheckMenuItem(Attr.Menu, cm_Ques+Idx, mf_UnChecked);
      inc(Idx);
    end; {repeat}
  until (Idx > Groups^.Count);

end; {TExpertWin.UnCheckQuestions}

procedure TExpertWin.EnableQuestions;
{---------------------------------------------------------
EnableQuestions: For each group in Groups, all slots are
checked to see if any are "active" (ie. are accepting
input).  If any are, then the group menu selection is
enabled.
---------------------------------------------------------}
var
  i,j : integer;
  Flag: boolean;
  Slot: PSlot;
  Grp : PGroup;

begin
  for i := 0 to Groups^.Count-1 do
  begin
    Grp := Groups^.At(i);
    Flag := false;
    j := 0;

    while ((not Flag) and (j < Grp^.SlotCount)) do
    begin
      Slot := Slots^.FindMatch(StrPas(Grp^.GetSlot(j)));
      if (Slot <> nil) then Flag := Slot^.IsActive;
      inc(j);
    end; {while}

    if (Flag) then
      EnableMenuItem(Attr.Menu,i+cm_Ques+1,mf_ByCommand +
           mf_Enabled);
  end; {for}

end; {TExpertWin.EnableQuestions}

{---------------------------------------------------}
{ TExpertApp's method implementations:             }
{---------------------------------------------------}

procedure TExpertApp.InitMainWindow;
{---------------------------------------------------------
InitMainWindow: A primary window is created with the title
"Expert System"
---------------------------------------------------------}
begin
  MainWindow := New(PExpertWin, Init(nil, 'Expert System'));

end; {TExpertApp.InitMainWindow}

{---------------------------------------------------------
Main program: The application, with id EXPERT, is started.
---------------------------------------------------------}

var
  ExpertApp : TExpertApp;

begin
  ExpertApp.Init('Expert');
  ExpertApp.Run;
  ExpertApp.Done;

end. {Expert}
```

<u>GRPOBJ.UNT</u>

```
Unit GrpObj;

Interface
  uses Strings, WinTypes, WinProcs, WObjects;
  const
    GNAME_MAX      =  50;
    MAX_SLOTS      =  25;
    SLOTS_OVERFLOW =  10;

  type
    PSlotName = ^TSlotName;
    TSlotName = object(TObject)
      {Attributes}
      Slot : PChar;
      {Methods}
      constructor Init(ASlot: PChar);
    end; {TSlotName}

    TGName  = array[0..GNAME_MAX] of char;

    PGroup  = ^TGroup;
    TGroup  = object(TObject)
      { Attributes}
      Name  : PChar;
      Slots : PCollection;

      { Methods}
      constructor Init(AName: PChar);
      destructor  Done; virtual;
      function    GetName : PChar; virtual;
      procedure   SetName(AName: PChar); virtual;
      function    GetSlots: PCollection; virtual;
      function    GetSlot(Idx: Integer): PChar; virtual;
      procedure   AddSlot(ASlot: PChar); virtual;
      procedure   DeleteSlot(Idx: Integer); virtual;
      function    SlotCount: Integer; virtual;
    end; {TPrompt}

Implementation

Constructor TGroup.Init(AName: PChar);
{-------------------------------------------------------
TGroup.Init: This is the constructor for the TGroup object.
All attributes are assigned values.  The collection of slots
is set to the empty state.
-------------------------------------------------------}
```

```
  begin
    { Set the group name.}
    Name := StrNew(AName);

    { Initialize the list to the empty state.}
    Slots := new(PCollection,
Init(MAX_SLOTS,SLOTS_OVERFLOW));

  end; {TGroup.Init}

Destructor TGroup.Done;
{----------------------------------------------------------
TPrompt.Done: This is the destructor for the TGroup object.
The string attribute, NAME, is disposed of.
----------------------------------------------------------}
  begin
    StrDispose(Name);
    dispose(Slots, done);

  end; {TGroup.Done}

Function TGroup.GetName: PChar;
{----------------------------------------------------------
TGroup.GetName: Returns the group NAME for the TGroup
object.
----------------------------------------------------------}
  begin
    GetName := Name;

  end; {TGroup.GetName}

Procedure TGroup.SetName(AName: PChar);
{----------------------------------------------------------
TGroup.SetName: Sets the NAME attribute of the TGroup
object.
----------------------------------------------------------}
  begin
    Name := StrNew(AName);

  end; {TGroup.SetName}

Function TGroup.GetSlots: PCollection;
{----------------------------------------------------------
TGroup.GetSlots: Retrieves the slot list for the group.  The
list is a collection of pointers to pointers of null
terminated strings (PChar).
----------------------------------------------------------}
  begin
    GetSlots := Slots;
  end; {TGroup.GetSlots}

Function TGroup.GetSlot(Idx: Integer): PChar;
{-----------------------------------------------------------
TGroup.GetSlot: Retrieves the slot specified by the index
(IDX).
-----------------------------------------------------------}
  var
    s : PSlotName;
  begin
    s := Slots^.At(Idx);
    GetSlot := s^.Slot;

  end; {TGroup.GetSlot}

Procedure TGroup.AddSlot(ASlot: PChar);
{-----------------------------------------------------------
TGroup.AddSlot: Adds a new slot, ASLOT, to the collection of
slots.
-----------------------------------------------------------}
  begin
    Slots^.insert(new(PSlotName, Init(ASlot)));

  end; {TGroup.AddSlot}

Procedure TGroup.DeleteSlot(Idx: integer);
{-----------------------------------------------------------
TGroup.DeleteSlot: Removes the slot indexed by, IDX, from
the collection of slots.
-----------------------------------------------------------}
  begin
    Slots^.AtFree(Idx);

  end; {TGroup.DeleteSlot}

Function TGroup.SlotCount: Integer;
{-----------------------------------------------------------
TGroup.SlotCount: The number of slots in the SLOTS list is
returned.
-----------------------------------------------------------}
  begin
    SlotCount := Slots^.Count;

  end; {TGroup.SlotCount}
```

```
Constructor TSlotName.Init(ASlot: PChar);
{-------------------------------------------------------------
Init: The TSlotName object is initialized by allocating
space on the heap 4 the string pointed to by PChar, and
setting Slot to point to this new null terminated string.
-------------------------------------------------------------}
begin
  Slot := StrNew(ASlot);

end; {TSlotName.Init}

End.
```

PROPLIST.UNT

```
Unit PropList;

Interface
  Uses
    WinTypes, WinProcs, WObjects, PropObj, NexFile;

  Const
    MAX_PROPS     = 200;
    PROPS_OVRFLOW =  20;
    PROP_EXT = '.prp';

  Type
    PPropList = ^TPropList;
    TPropList = object(TCollection)
      {Methods}
      constructor Init(AMax, AnOvrFlow: integer);
    private
      procedure FetchProps; virtual;
    end; {TPropList}

Implementation

Constructor TPropList.Init(AMax, AnOvrFlow: integer);
{ -----------------------------------------------------------
InitProps: This is the constructor for the TProps object
which is a special collection object that holds a list of
properties.  The collection is initialized by calling the
ancestral constructor and then a routine loads the property
data from a file.
-----------------------------------------------------------}
begin
  TCollection.Init(AMax, AnOvrFlow);
  FetchProps;

end; {TPropList.Init}

Procedure TPropList.FetchProps;
{ -----------------------------------------------------------
FetchProps: Each pair of lines in the file, PropFile,
represents a PROPERTY.  This procedure reads each pair of
lines (Line1, Line2) and creates a new TProp object and
inserts it into the list of PROPERTIES that this object
holds.
-----------------------------------------------------------}
var
  PropFile : text;
  Line1, Line2 : string;
```

```
begin
  OpenFiles(PropFile, ConCat(NEX_FILE, PROP_EXT));

  { Discard the first line.}
  readln(PropFile);

  while (not eof(PropFile)) do
    begin
      readln(PropFile, Line1);
      if (not eof(PropFile)) then readln(PropFile, Line2);
      Insert(new(PProp, Init(Line1, Line2)));
      readln(PropFile);
    end; {while}

    CloseFiles(PropFile);

end; {TPropList.FetchProps}

End. {PropList}
```

<u>NEXFILE.UNT</u>

```
Unit NexFile;

{$V-} { Turn off type checking for strings.}

Interface

  Uses
    WinDos;

  Const
    LINE_MAX = 255;
    TYPE_MAX =   2;
    NEWLINE  = chr(13);
    NEX_FILE = 'test';

  Type
    TLine     = string[LINE_MAX];
    TLineType = string[TYPE_MAX];
    TLineIndex= 0..LINE_MAX+1;
    TFileName = string[fsPathName];

  Function NextWord(line: TLine; i,j: TLineIndex):
      TLineIndex;
  Procedure ParseWord(Line: TLine; delimiter: char;
                var pword: TLine; var i,j: TLineIndex);
  Function  ProcessComponent(Line: TLine; i,j: TLineIndex):
                TLine;
  Procedure OpenFiles(var TargetFile: text; filename:
                TFileName);
  Procedure CloseFiles(var TargetFile: text);

Implementation

Function NextWord(line: TLine; i,j: TLineIndex): TLineIndex;
{ -----------------------------------------------------------
Next_Word: This function moves the line index, I, to point
to the next non-white-space character in the line, LINE, or
point to the end of the line, J.
-----------------------------------------------------------}

  begin
    { Increment I until a non-white-space character is found
      or the end of the line is reached.}
    while ((i <= j) and ((line[i] < '!') or (line[i] >
      '~'))) do i := i+1;
    NextWord := i;
  end; {Next_Word}
```

```
Procedure ParseWord(Line: TLine; delimiter: char;
                     var pword: TLine; var i,j: TLineIndex);
{ -----------------------------------------------------------
ParseWord: The Line index, I, is used to procede character
by character along the Line, Line, from the current position
of I until the delimiter, DELIMITER, is found or a
white-space character is encountered, adding each character
to the string variable, PWORD.  If neither of the  previous
conditions are met, the parsing stops when the end of the
Line (J) is reached.
Called by: External
-----------------------------------------------------------}
  var
    quit : boolean;

  begin
    pword := '';

    { Add each character to PWORD until the DELIMITER is
      found or the end of the Line is reached or a
      white-space character is encountered.}
    quit := false;
    while ((i <= j) and (not quit)) do
      begin
        quit := ((Line[i] = delimiter) or (Line[i] < ' ') or
           (Line[i] > '~'));
        if (not quit) then
          begin
            pword := pword + Line[i];
            inc(i);
          end; {if}
      end; {while}

  end; {ParseWord}

Function ProcessComponent(Line: TLine; i,j: TLineIndex):
TLine;
{ -----------------------------------------------------------
ProcessComponent: The remainder of the input Line contains
the name of the component to be added to the current Object.
This function extracts and returns the name.
Called by: External
-----------------------------------------------------------}
  var
    AComponent : TLine;

  begin
    { Extract the remainder of the Line.}
    inc(i);
    ParseWord(Line,NEWLINE,AComponent,i,j);
    ProcessComponent := AComponent;

  end; {ProcessComponent}

Procedure OpenFiles(var TargetFile: text; filename:
TFileName);
{ -----------------------------------------------------------
OpenFiles: The input file is opened, using the name,
FILENAME.
Called by: External
------------------------------------------------------------}
  var
    i,j : TLineIndex;

  begin
    { Open the input file.}
    assign(TargetFile, filename);
    {$I-}
    reset(TargetFile);
    {$I+}
    { Check for an IO error.}
    if (IOResult <> 0) then
      begin
        writeln('File ',filename,' not found.');
        halt(1);
      end; {if}

  end; {OpenFiles}

Procedure CloseFiles(var TargetFile: text);
{ -----------------------------------------------------------
CloseFiles: Both the input file is closed.
Called by: External
------------------------------------------------------------}
  begin
    { Close the file.}
    close(TargetFile);

  end; {CloseFiles}

End. {NexFile}
```

CLASSOBJ.UNT

```
Unit ClassObj;

Interface

  Uses
    WinTypes, WinProcs, WObjects;

  Const
    MAX_PROPNAMES     = 25;
    PROPNAMES_OVRFLOW =  5;

  Type
    PPropName = ^TPropName;
    TPropName = object(TObject)
      {Attributes}
      Name : string;
      {Methods}
      constructor Init(AName: string);
      function    GetName: string; virtual;
    end; {TPropName}

    PClass = ^TClass;
    TClass = object(TObject)
      {Attributes}
      Name : string;
      PropNames : PCollection;
      {Methods}
      constructor Init(AName: string);
      destructor  Done; virtual;
      function    GetName: string; virtual;
      procedure   AddProp(AProp: string); virtual;
      function    GetProp(Idx: integer): string; virtual;
      function    PropCount: integer; virtual;
    end; {TClass}

Implementation

Constructor TClass.Init(AName: string);
{----------------------------------------------------------
Init: A TClass object is constructed by assigning Name the
value of AName.  Also, the list, PropNames, is initialized.
----------------------------------------------------------}
begin
  Name := AName;
  PropNames := new(PCollection, Init(MAX_PROPNAMES,
      PROPNAMES_OVRFLOW));
end; {TClass.Init}
```

```
Destructor TClass.Done;
{-----------------------------------------------------------
Done: The PropNames list is disposed, freeing the memory
allocated to it.
-----------------------------------------------------------}
begin
  dispose(PropNames, done);

end; {TClass.Done}

Function TClass.GetName: string;
{-----------------------------------------------------------
GetName: The string value of Name is returned.
-----------------------------------------------------------}
begin
  GetName := Name;

end; {TClass.GetName}

Procedure TClass.AddProp(AProp: string);
{-----------------------------------------------------------
AddProp: A new TPropName object is created and added to the
list of PropNames.
-----------------------------------------------------------}
begin
  PropNames^.Insert(new(PPropName, Init(AProp)));

end; {TClass.AddProp}

Function TClass.GetProp(Idx: integer): string;
{-----------------------------------------------------------
GetProp: As long as the index, Idx, is within range, the
TPropName object indexed by Idx, in the list PropNames, is
found and the string stored within it is returned.
-----------------------------------------------------------}
var
  p : PPropName;

begin
  if (Idx < PropNames^.Count) then
    begin
      p := PropNames^.At(Idx);
      GetProp := p^.GetName;
    end {if}
  else GetProp := '';

end; {TClass.GetProp}

Function TClass.PropCount: integer;
```

```
{-----------------------------------------------------------
PropCount: The number of items in the PropNames list is
returned.
-----------------------------------------------------------}
begin
  PropCount := PropNames^.Count;

end; {TClass.PropCount}

Constructor TPropName.Init(AName: string);
{-----------------------------------------------------------
Init: This constructor creates a new TPropName object and
stores in it the string, AName, in the Name attribute.
-----------------------------------------------------------}
begin
  Name := AName;

end; {TPropName.Init}

Function TPropName.GetName: string;
{-----------------------------------------------------------
GetName: The string held by Name is returned.
-----------------------------------------------------------}
begin
  GetName := Name;

end; {TPropName.GetName}

End. {ClassObj}
```

BAKCHAIN.UNT

```
Unit BakChain;

Interface

  Uses
    WObjects, ObtObj, RuleObj, SlotList, Stack, Operator,
Expr;

  Type
    TPerform = (Incomplete, CompleteTrue, CompleteFalse,
Wait);

    PConclusion = ^TConclusion;
    TConclusion = object(TObject)
      {Attributes}
      RuleIdx : integer;
      Rule    : PRule;
      Clause  : integer;
      {Methods}
      constructor Init(ARule: PRule; ARuleIdx, AClause:
           integer);
      function GetRule: PRule; virtual;
      procedure NuRule(ARule: PRule; ARuleIdx: integer);
           virtual;
      function GetClause: integer; virtual;
      function IncClause: boolean; virtual;
    end; {TConclusion}

    PConclusionStack = ^TConclusionStack;
    TConclusionStack = object(TStack)
      {Attributes}
      Start   : boolean;
      Suggest : string;
      Slots   : PSlotList;
      Props   : PCollection;
      Classes : PCollection;
      Objts   : PCollection;
      Rules   : PCollection;
      {Methods}
      constructor Init(ASuggestion: string; APropList,
           AClassList, AnObjtList, ARuleList: PCollection;
           ASlotList: PSlotList);
      function BackChain: boolean; virtual;
      procedure ClearStack; virtual;
    private
      function FindHypo(Idx: integer; AHypo: string):
           integer; virtual;
      function NextRule: boolean; virtual;
      function NuConclusion(AHypo: string): boolean;
           virtual;
      function PerformOp(AnExpr: PExpr): TPerform; virtual;
      procedure AdvTop; virtual;
    end; {TConclusionStack}

Implementation

Constructor TConclusion.Init(ARule: PRule; ARuleIdx,
                              AClause: integer);
{----------------------------------------------------------
Init: The TConclusion object is instantiated by setting Rule
and Clause equal to ARule and AClause, respectively.
----------------------------------------------------------}
begin
  Rule := ARule;
  RuleIdx := ARuleIdx;
  Clause := AClause;

end; {TConclusion.Init}

Function TConclusion.GetRule: PRule;
{----------------------------------------------------------
GetRule: This function serves to return a pointer to the
RULE that is stored in the TConclusion object.
----------------------------------------------------------}
begin
  GetRule := Rule;

end; {TConclusion.GetRule}

Procedure TConclusion.NuRule(ARule: PRule; ARuleIdx:
      integer);
{----------------------------------------------------------
NuRule: The current value of Rule is replaced by ARule and
the current value of RuleIdx is replaced by ARuleIdx.
----------------------------------------------------------}
begin
  Rule := ARule;
  RuleIdx := ARuleIdx;
  Clause := 0;

end; {TConclusion.NuRule}

Function TConclusion.GetClause: integer;
{----------------------------------------------------------
GetClause: This function serves to return the value of the
Clause indicator, that is stored in the TConclusion object.
```

```
---------------------------------------------------------}
begin
  GetClause := Clause;

end; {TConclusion.GetClause}

Function TConclusion.IncClause: boolean;
{---------------------------------------------------------
IncClause: If Clause is less than the number of LHS
expressions, then Clause is incremented by one, and the
function returns true; otherwise, Clause is left unchanged,
and the function returns false.
---------------------------------------------------------}
begin
  if (Clause < Rule^.LhsCount-1) then
    begin
      inc(Clause);
      IncClause := true;
    end {if}
  else
    IncClause := false;

end; {TConclusion.IncClause}

Constructor TConclusionStack.Init(ASuggestion: string;
APropList, AClassList,
            AnObjtList, ARuleList: PCollection; ASlotList:
PSlotList);
{---------------------------------------------------------
Init: The stack is initialized by calling the parent
constructor, and then the collection holders are assigned
the appropriate values.
---------------------------------------------------------}
begin
  TStack.Init;
  Start := true;
  Suggest := ASuggestion;
  Slots := ASlotList;
  Props := APropList;
  Classes := AClassList;
  Objts := AnObjtList;
  Rules := ARuleList;

end; {TConclusionStack.Init}

Function TConclusionStack.FindHypo(Idx: integer; AHypo:
      string): integer;
{---------------------------------------------------------
FindHypo: The list of RULES, Rules, is searched, starting at
```

```
the index Idx and proceding forward (+ve) through the list.
The search halts when a match (as indicated by Match) is
made between a RULE (r) hypothesis and AHypo, or when the
end of the list is reached.  If a  match is made, this
function returns the index (Idx) to the matching RULE.
Otherwise, a value of -1 is returned.
---------------------------------------------------------}
var
  Match : boolean;
  Count : integer;
  r     : PRule;

begin
  Match := false;
  Count := Rules^.Count;
  { Search for a match.}
  while ((Idx < Count) and (not Match)) do
    begin
      r := Rules^.At(Idx);
      Match := AHypo = r^.GetHypo;
      inc(Idx);
    end; {while}

  { If a match is found, return the index to the matching
      rule.}
  if (Match) then FindHypo := Idx-1
  { Otherwise, return -1.}
  else FindHypo := -1;

end; {TConclusionStack.FindHypo}

Function TConclusionStack.NextRule: boolean;
{---------------------------------------------------------
NextRule:
---------------------------------------------------------}
var
  c : PConclusion;
  RuleIdx : integer;
  Rule : PRule;

begin
  c := Top;
  RuleIdx := FindHypo(c^.RuleIdx+1, c^.Rule^.GetHypo);
  if (RuleIdx <> -1) then c^.NuRule(Rules^.At(RuleIdx),
      RuleIdx);
  NextRule := RuleIdx <> -1;

end; {TConclusionStack.NextRule}
```

```
Function TConclusionStack.NuConclusion(AHypo: string):
      boolean;
{-----------------------------------------------------------
NuConclusion: If the hypothesis, AHypo, is found in a RULE
in Rules, a new TConclusion object is created and pushed
onto the top of the conclusion stack and the function
returns true.  If AHypo is not found, then the function
simply returns false.
-----------------------------------------------------------}
var
  RuleIdx : integer;
  Rule    : PRule;

begin
  RuleIdx := FindHypo(0, AHypo);
  if (RuleIdx <> -1) then
    begin
      Rule := Rules^.At(RuleIdx);
      Push(new(PConclusion, Init(Rule, RuleIdx, 0)));
    end; {if}

  NuConclusion := RuleIdx <> -1;

end; {TConclusionStack.NuConclusion}

Function TConclusionStack.PerformOp(AnExpr: PExpr):
      TPerform;
{-----------------------------------------------------------
PerformOp:
-----------------------------------------------------------}
var
  TriAns : TriBool;
  p : TPerform;

begin
  p := CompleteTrue;
  case (AnExpr^.GetOperator) of
    _CreateObject:
      ProcCreateObject(Classes, Objts, Props,
      AnExpr^.GetOperand1, AnExpr^.GetOperand2);

    _Do:
      ProcDo(Objts,AnExpr^.GetOperand1,AnExpr^.GetOperand2);

    _Is:
      begin
        TriAns := ProcIs(Objts, AnExpr^.GetOperand1,
           AnExpr^.GetOperand2);
        if (TriAns = Unknown) then
      begin
        if (not Slots^.IsActive(AnExpr^.GetOperand1))
        then
          Slots^.ToggleSlot(AnExpr^.GetOperand1);
          p := Wait;
      end
    else if (TriAns = No) then p := CompleteFalse;
  end; {_Is}

_IsNot:
  begin
    TriAns := ProcIsNot(Objts, AnExpr^.GetOperand1,
       AnExpr^.GetOperand2);
    if (TriAns = Unknown) then
      begin
        if (not Slots^.IsActive(AnExpr^.GetOperand1))
        then
          Slots^.ToggleSlot(AnExpr^.GetOperand1);
        p := Wait
      end
    else if (TriAns = No) then p := CompleteFalse;
  end; {_IsNot}

_Name:
  if (ProcName(Objts, AnExpr^.GetOperand1,
  AnExpr^.GetOperand2) = Unknown) then
    begin
      if (not Slots^.IsActive(AnExpr^.GetOperand1)) then
        Slots^.ToggleSlot(AnExpr^.GetOperand1);
      p := Wait;
    end; {if}

_No:
  begin
    TriAns := ProcNo(Objts, AnExpr^.GetOperand1);
    if (TriAns = Unknown) then p := Incomplete
    else if (TriAns = No) then p := CompleteFalse;
  end; {_No}

_Retrieve:
  if (not ProcRetrieve(Classes, Objts, Props,
  AnExpr^.GetOperand1, AnExpr^.GetOperand2)) then
  p := CompleteFalse;

_Show:
  ProcShow(Objts, AnExpr^.GetOperand1,
  AnExpr^.GetOperand2);

_Yes:
      begin
        TriAns := ProcYes(Objts, AnExpr^.GetOperand1);
        if (TriAns = Unknown) then p := Incomplete
        else if (TriAns = No) then p := CompleteFalse;
      end; {_Yes}

  end; {case}

  PerformOp := p;

end; {TConclusionStack.PerformOp}

Procedure TConclusionStack.AdvTop;
{-------------------------------------------------------------
AdvTop: This is a recursive procedure that increments the
Clause indicator of the TConclusion object on the top of the
Conclusion Stack.  If incrementing this object results in
the Clause indicator surpassing the number of clauses in the
associated RULE, then the Conclusion is popped off the
stack, disposed of, and the procedure recursively calls
itself.
-------------------------------------------------------------}
var
  c : PConclusion;

begin
  if (not IsEmpty) then
    begin
      c := Top;
      {Increment the Clause indicator.}
      if (not c^.IncClause) then
        begin
          AssignValue(Objts,'TRUE',c^.Rule^.GetHypo);
          c := Pop;
          dispose(c, done);
          AdvTop;
        end; {if}
    end; {if}

end; {TConclusionStack.AdvTop}

Function TConclusionStack.BackChain: boolean;
{-------------------------------------------------------------
BackChain:
-------------------------------------------------------------}
var
  c : PConclusion;
  LHS : PExpr;
  Op : TPerform;
```

```
begin
  readln;
  Op := Incomplete;
  if (Start) then
  begin
    NuConclusion(Suggest);
    Start := false;
  end; {if}

  while ((not IsEmpty) and (Op <> Wait)) do
    begin
      c := Top;
      LHS := c^.Rule^.GetLHS(c^.Clause);
      Op := PerformOp(LHS);
      if (Op = Incomplete) then
        if (not NuConclusion(LHS^.GetOperand1)) then halt;
      else if (Op = CompleteTrue) then AdvTop
      else
        begin
          AssignValue(Objts, 'FALSE', c^.Rule^.GetHypo);
          if (not NextRule) then
            begin
              c := Pop;
              dispose(c, done);
            end; {if}
        end; {else}
  end; {while}

  BackChain := Op <> Wait;

end; {TConclusionStack.BackChain}

Procedure TConclusionStack.ClearStack;
{----------------------------------------------------------
ClearStack: The contents of the stack are popped off one at
a time and disposed.  In this way, the stack is cleared.
----------------------------------------------------------}
var
  c : PConclusion;
begin
  while (not IsEmpty) do
  begin
    c := Pop;
    dispose(c,done);
  end; {while}
  Start := true;
end; {TConclusionStack.ClearStack}
End. {BakChain}
```

CLSLIST.UNT

```
Unit ClsList;
{$V-} { Turn off type checking for strings.}

Interface
  Uses
    WObjects, ClassObj, NexFile;

  Const
    MAX_CLASS     = 25;
    CLASS_OVRFLOW =  5;
    CLS_EXT = '.cls';

  Type
    PClassList = ^TClassList;
    TClassList = object(TCollection)
      {Methods}
      constructor Init(AMax, AnOvrFlow: integer);
    private
      procedure ProcessSubClass(ClassLine: TLine; var
           AClass: PClass); virtual;
      procedure FetchClasses; virtual;
    end; {TPropList}

Implementation

Constructor TClassList.Init(AMax, AnOvrFlow: integer);
{ -----------------------------------------------------------
Init:
------------------------------------------------------------}
begin
  TCollection.Init(AMax, AnOvrFlow);
  FetchClasses;

end; {TClassList.Init}

Procedure TClassList.ProcessSubClass(ClassLine: TLine; var
      AClass: PClass);
{ -----------------------------------------------------------
ProcessSubClass: If a subclass has been encountered, then
this procedure is called to find all the PROPERTIES of the
subclass, and add them to the current CLASS.  Cls is used as
a pointer to find the subclass in the list of CLASSES.
Therefore, CLASSES must be defined before they can be used
as subclasses.  Once the subclass is found, i and j are
used, as an index and upper limit, to retrieve all the
PROPERTIES from the subclass (NB: the function Match is used
to find the subclass in the list of CLASSES.)
---------------------------------------------------------}
var
  Cls : PClass;
  i, j: integer;
  Prop : string;

function Match(c: PClass):boolean; far;
  begin
    Match := c^.GetName = ClassLine;
  end; {Match}

begin
  Cls := FirstThat(@Match);
  if (Cls <> nil) then begin
    i := 0;
    j := Cls^.PropCount;
    while (i < j) do
      begin
        Prop := Cls^.GetProp(i);
        if (Prop <> '') then AClass^.AddProp(Prop);
        inc(i);
      end; {while}
  end; {if}

end; {TClassList.ProcessSubClass}

Procedure TClassList.FetchClasses;
{ --------------------------------------------------------
FetchClasses: The CLASS file, ClassFile, is opened and the
CLASS information found in it is extracted and stored in
CLASS objects (c).  CLASSES are made up two different
elements, subclasses and PROPERTIES.  A subclass is a CLASS
that has already been defined and converted to a list of
PROPERTIES.  Therefore, when a subclass is encountered (as
indicated by LineType), the list of PROPERTIES for the
subclass is substituted, instead.  If a PROPERTY is found,
then it is simply added to the list PROPERTIES for the
current CLASS.
---------------------------------------------------------}
var
  ClassFile : text;
  c : PClass;
  i,j : TLineIndex;
  LineType : TLineType;
  Line, ClassLine : TLine;

begin
  OpenFiles(ClassFile, concat(NEX_FILE, CLS_EXT));
  { Discard the first Line.}
  readln(ClassFile);
  while (not eof(ClassFile)) do
    begin
      { Fetch the name of the class.}
      readln(ClassFile, Line);
      i := 1; j := length(Line);
      ParseWord(Line,' ',ClassLine,i,j);

      { Allocate a new class.}
      c := new(PClass, Init(ClassLine));

      while ((not eof(ClassFile)) and (length(Line) > 0)) do
        begin
          { Fetch the Line-type of the next Line in the
           file.}
          readln(ClassFile, Line);
          i := 1; j := length(Line);
          ParseWord(Line,' ',LineType,i,j);
          ClassLine := ProcessComponent(Line,i,j);

          { Act according to the LineType.}
          if (LineType = 'SC') then
                ProcessSubClass(ClassLine, c)
          else c^.AddProp(ClassLine);
        end; {while}
      Insert(c);
    end; {while}

  CloseFiles(ClassFile);

end; {TClassList.FetchClasses}

End. {ClsList}
```

<u>RULELOAD.UNT</u>

```
Unit RuleLoad;
{$V-} { Turn off type checking for strings.}

Interface

  Uses
    WObjects, RuleObj, NexFile;

  Const
    MAX_RULE     = 25;
    RULE_OVRFLOW =  5;
    RULE_EXT = '.rul';

  procedure FetchRules(var ARuleList: PCollection);

Implementation

Procedure InitRules(var ARuleList: PCollection);
{ --------------------------------------------------------
InitClasses: The list of rules, ARuleList, is initialized.
Called by: FetchRules
----------------------------------------------------------}
  begin
    { Initialize the list to the empty state.}
    ARuleList := new(PCollection, Init(MAX_RULE,
      RULE_OVRFLOW));

  end; {InitRules}

Procedure ProcessIC(AnIC: TLine; var ARule: PRule);
{ --------------------------------------------------------
ProcessIC: The string AnIC is converted to a real value.  If
the conversion is ok, the IC of ARule is assigned the
integer equivalent of the real.
Called by: FetchRules
----------------------------------------------------------}
var
  Err : integer;
  Result : real;

begin
  val(AnIC, Result, Err);
  if (Err = 0) then ARule^.SetIC(Trunc(Result));

end; {ProcessIC}
```

```
Procedure ProcessLhs(ALine: TLine; var RuleFile: text; var
      ARule: PRule);
{ ----------------------------------------------------------
ProcessLhs: The Operator line (ALine) of a Lhs expression
has been encountered and this procedure fetches the next two
lines (Operand1, Operand2) which contain the first and
second operands of the expression, from the input file,
RuleFile.  Then, the Lhs expression is added to the list of
such expressions in the RULE, ARule.
Called by: FetchRules
----------------------------------------------------------}
var
  Operator, Operand1, Operand2 : TLine;

begin
  Operator := ALine;
  readln(RuleFile, Operand1);
  readln(RuleFile, Operand2);
  ARule^.AddLhs(Operator, Operand1, Operand2);

end; {ProcessLhs}

Procedure ProcessRhs(ALine: TLine; var RuleFile: text; var
      ARule: PRule);
{ ----------------------------------------------------------
ProcessRhs: The Operator line (ALine) of a Rhs expression
has been encountered and this procedure fetches the next two
lines (Operand1, Operand2) which contain the first and
second operands of the expression, from the input file,
RuleFile.  Then, the Rhs expression is added to the list of
such expressions in the RULE, ARule.
Called by: FetchRules
----------------------------------------------------------}
var
  Operator, Operand1, Operand2 : TLine;

begin
  Operator := ALine;
  readln(RuleFile, Operand1);
  readln(RuleFile, Operand2);
  ARule^.AddRhs(Operator, Operand1, Operand2);

end; {ProcessRhs}

Procedure FetchRules(var ARuleList: PCollection);
{ ----------------------------------------------------------
FetchRules: The file, RuleFile, is read and the RULE data is
extracted from it and stored in RULE objects (r), which are
kept in a list of rules, ARuleList.  The integers, i and j,
are used to parse the lines of the file;  LineType is used
to store the line type, as indicated by the first two
characters on the input line.  Also, Line is used to hold
each line of the input file, while it is being analysed.
Called by: External
---------------------------------------------------------}
var
  RuleFile : text;
  r : PRule;
  i,j : TLineIndex;
  LineType : TLineType;
  Line, RuleLine : TLine;

begin
  InitRules(ARuleList);
  OpenFiles(RuleFile, concat(NEX_FILE, RULE_EXT));

  { Discard the first Line.}
  readln(RuleFile);
  while (not eof(RuleFile)) do
    begin
      { Fetch the name of the class.}
      readln(RuleFile, Line);
      i := 1; j := length(Line);
      ParseWord(Line,' ',RuleLine,i,j);

      { Allocate a new class.}
      r := new(PRule, Init(RuleLine));

      while ((not eof(RuleFile)) and (length(Line) > 0)) do
        begin
          { Fetch the Line-type of the next Line in the
            file.}
          readln(RuleFile, Line);
          i := 1; j := length(Line);
          ParseWord(Line,' ',LineType,i,j);
          RuleLine := ProcessComponent(Line,i,j);

          { Act according to the LineType.}
          if (LineType = 'IC') then ProcessIC(RuleLine, r)
          else if (LineType = 'L1') then
            ProcessLhs(RuleLine, RuleFile, r)
          else if (LineType = 'R1') then
            ProcessLhs(RuleLine, RuleFile, r)
          else if (LineType = 'HY') then
            r^.SetHypo(RuleLine);
        end; {while}
      ARuleList^.Insert(r);
    end; {while}
  CloseFiles(RuleFile);
end; {FetchRules}
End. {RuleLoad}
```

<u>OBTLIST.UNT</u>

```
Unit ObtList;

{$V-} { Turn off type checking for strings.}

Interface

  Uses
    WObjects, Strings, NexFile, ObtObj, ClassObj, PropObj;

  Const
    MAX_OBTS     = 100;
    OBTS_OVRFLOW =  20;

    OBT_EXT = '.obt';

  Type
    PObjtList = ^TObjtList;
    TObjtList = object(TCollection)
      constructor Init(AMax, AnOvrFlow: integer;
                       APropList, AClassList: PCollection);
    private
      procedure ProcessClass(Line: TLine; AClassList,
        APropList: PCollection; var Objt: PObjt); virtual;
      procedure ProcessProp(Line: TLine; APropList:
        PCollection; var Objt: PObjt); virtual;
      procedure InsertObjt(AnObjt: PObjt); virtual;
      procedure FetchObjects(APropList, AClassList:
        PCollection); virtual;
    end; {TObjtList}

Implementation

Constructor TObjtList.Init(AMax, AnOvrFlow: integer;
                APropList, AClassList: PCollection);
{ -----------------------------------------------------------
Init: This is the constructor for the TObjtList object,
which inherits from TCollection.  First the collection is
initialized by calling the parent constructor.  Then, the
OBJECT data is retrieved from disk by a call to
FetchObjects.
-----------------------------------------------------------}
  begin
    TCollection.Init(AMax, AnOvrFlow);
    FetchObjects(APropList, AClassList);

  end; {TObjtList.Init}
```

```
Procedure TObjtList.ProcessClass(Line: TLine; AClassList,
      APropList: PCollection; var Objt: PObjt);
{ ----------------------------------------------------------
ProcessClass: The parameter, Line, contains a CLASS name.
The list, AClassList, is searched for a CLASS having this
name.  If such a CLASS is found, all PROPERTIES of this
CLASS are added to the list of PROPERTIES associated with
the current OBJECT, Objt.
----------------------------------------------------------}
var
  PropName: string;
  i, j: integer;
  Cls: PClass;

function ClassMatch(c: PClass): boolean; far;
  begin
    ClassMatch := c^.GetName = Line;
  end; {ClassMatch}

function PropMatch(p: PProp): boolean; far;
  begin
    PropMatch := p^.GetName = PropName;
  end; {PropMatch}

begin
  Cls := AClassList^.FirstThat(@ClassMatch);
  if (Cls <> nil) then
    begin
      j := Cls^.PropCount;
      i := 0;
      while (i < j) do
        begin
          PropName := Cls^.GetProp(i);
          if (PropName <> '') then
            Objt^.AddProp(APropList^.FirstThat(@PropMatch));
          inc(i);
        end; {while}
    end; {if}

end; {TObjtList.ProcessClass}

Procedure TObjtList.ProcessProp(Line: TLine; APropList:
      PCollection;
                                   var Objt: PObjt);
{ ---------------------------------------------------------
ProcessProp: Line contains the name of a PROPERTY.  The
list, APropList, is searched (Match) for a PROPERTY with
this name.  If the search is successful, the PROPERTY is
added to the list of PROPERTIES associated with the current
OBJTECT, Objt.
------------------------------------------------------------}
var
  Prop : PProp;

function Match(p: PProp): boolean; far;
  begin
    Match := p^.GetName = Line;
  end; {Match}

begin
  Prop := APropList^.FirstThat(@Match);
  if (Prop <> nil) then Objt^.AddProp(Prop);

end; {TObjtList.ProcessProp}

Procedure TObjtList.InsertObjt(AnObjt: PObjt);
{------------------------------------------------------------
InsertObjt: This procedure adds the OBJECT, AnObjt, to the
collection of OBJECTS, as long as it is not already found in
the list.  If it is already in the list, then this procedure
has no effect.  To find the location to insert, a binary
search is used.
------------------------------------------------------------}
var
  First, Last, i : integer;
  Found : boolean;
  o : PObjt;

begin
  { Initialize search variables.}
  First := 0;
  Last  := Count -1;
  Found := false;
  { Search until successful or completed list.}
  while ((First <= Last) and (not Found)) do
    begin
      i := (First+Last) div 2;
      o := At(i);
      if (AnObjt^.GetName < o^.GetName) then Last := i-1
      else if (AnObjt^.GetName > o^.GetName) then
           First := i+1
      else Found := true;
    end; {while}

  { If the object is not already in the list, insert it.}
  if (First > Last) then AtInsert(First, AnObjt);

end; {TObjtList.InsertObjt}
```

```
Procedure TObjtList.FetchObjects(APropList, AClassList:
      PCollection);
{ -----------------------------------------------------------
FetchObjects: This procedure opens and reads the OBJECT
file, ObjectFile.  This file contains information on
OBJECTS.  An OBJECT consists of a name, which is read in and
used to instantiate a new OBJECT object (o), and one or more
subcomponents (being PROPERTIES from the list, APropList,
and CLASSES from the list, AClassList).  Each subcomponent
is read in, and is processed, according to its type
(LineType).  New OBJECTS are added to the list of OBJECTS,
held by TObjtList.  Line and ObjectLine are used as buffers
for the input file.
-----------------------------------------------------------}
var
  ObjectFile : text;
  o : PObjt;
  i,j : TLineIndex;
  LineType : TLineType;
  Line, ObjectLine : TLine;

begin
  OpenFiles(ObjectFile, concat(NEX_FILE, OBT_EXT));

  { Discard the first Line.}
  readln(ObjectFile);
  while (not eof(ObjectFile)) do
    begin
      { Fetch the name of the object.}
      readln(ObjectFile, Line);
      i := 1; j := length(Line);
      ParseWord(Line,' ',ObjectLine,i,j);

      { Allocate a new object.}
      o := new(PObjt, Init(ObjectLine));

      while ((not eof(ObjectFile)) and (length(Line) > 0))
        do begin
        { Fetch the Line-type of the next Line in the file.}
          readln(ObjectFile, Line);
          i := 1; j := length(Line);
          ParseWord(Line,' ',LineType,i,j);
          ObjectLine := ProcessComponent(Line,i,j);

          { Act according to the LineType.}
          if (LineType = 'VA') then o^.SetType(ObjectLine)
          else if (LineType = 'OC') then
            ProcessClass(ObjectLine,AClassList,APropList,o)
          else if (LineType = 'OP') then
```

```
            ProcessProp(ObjectLine,APropList, o);
          end; {while}
        InsertObjt(o);
      end; {while}

    CloseFiles(ObjectFile);

  end; {TObjtList.FetchObjects}

  End. {ObtList}
```

PROPOBJ.UNT

```
Unit PropObj;

Interface
  Uses
    WinTypes, WinProcs, WObjects;

  Const
    PROPERTY_SIZE = 255;

  Type
    TPropType = (PropInt, PropFloat, PropBool, PropStr,
      PropDate, PropTime, PropUnknown);

    PProp = ^TProp;
    TProp = object(TObject)
      {Attributes}
      Name : string[PROPERTY_SIZE];
      PropType : TPropType;
      Value : string;
      {Methods}
      constructor Init(AName: string; APropType: string);
      function    GetName: string; virtual;
      function    GetType: TPropType; virtual;
      function    GetValue: string; virtual;
      procedure   SetValue(AValue: string); virtual;
    end; {TProp}

  function  CalcType(APropType: string): TPropType;
  function  CalcIntValue(AValue: string): integer;
  function  CalcRealValue(AValue: string): real;
  function  CalcBoolValue(AValue: string): boolean;

Implementation

Constructor TProp.Init(AName: string; APropType: string);
{----------------------------------------------------------
Init: The TProp object is initialized by setting Name to
AName and fetching the TPropType corresponding to the
string, APropType.
----------------------------------------------------------}
begin
  Name := AName;
  PropType := CalcType(APropType);
  Value := '';

end; {TProp.Init}
```

```
Function TProp.GetName: string;
{-----------------------------------------------------------
GetName: The value of the string, Name, is returned.
-----------------------------------------------------------}
begin
  GetName := Name;

end; {TProp.GetName}

Function TProp.GetType: TPropType;
{-----------------------------------------------------------
GetType: The value of PropType is returned.
-----------------------------------------------------------}
begin
  GetType := PropType;

end; {TProp.GetType}

Function TProp.GetValue: string;
{-----------------------------------------------------------
GetValue: The string held by Value is returned.
-----------------------------------------------------------}
begin
  GetValue := Value;

end; {TProp.GetValue}

Procedure TProp.SetValue(AValue: string);
{-----------------------------------------------------------
SetValue: Value is assigned the string, AValue.
-----------------------------------------------------------}
begin
  Value := AValue;

end; {TProp.SetValue}

Function CalcType(APropType: string): TPropType;
{-----------------------------------------------------------
CalcType: This function returns a value of TPropType,
depending on the string, APropType.
-----------------------------------------------------------}
var
  i : TPropType;

begin
  if (APropType = 'Integer') then i := PropInt
  else if (APropType = 'Float') then i := PropFloat
  else if (APropType = 'Boolean') then i := PropBool
  else if (APropType = 'String') then i := PropStr
  else if (APropType = 'Date') then i := PropDate
  else i := PropTime;

  CalcType := i;

end; {CalcType}

Function CalcIntValue(AValue: string): integer;
{----------------------------------------------------------
CalcIntValue: This function attempts to convert the string,
AValue, to an integer value by use of the Val() function.
If the conversion fails (ie. ErrCode non-zero) then a value
of zero is returned.
---------------------------------------------------------}
var
  ErrCode: integer;
  Dest   : real;

begin
  val(AValue, Dest, ErrCode);
  if (ErrCode <> 0) then CalcIntValue := 0
  else CalcIntValue := Round(Dest);

end; {CalcIntValue}

Function CalcRealValue(AValue: string): real;
{----------------------------------------------------------
CalcRealValue: This function is similar to CalcIntValue,
except that if the conversion is successful, a real value is
returned.
----------------------------------------------------------}
var
  ErrCode: integer;
  Dest   : real;

begin
  val(AValue, Dest, ErrCode);
  if (ErrCode <> 0) then CalcRealValue := 0
  else CalcRealValue := Dest;

end; {CalcRealValue}

Function CalcBoolValue(AValue: string): boolean;
{----------------------------------------------------------
CalcBoolValue: If AValue is 'true' then the boolean, true,
is returned.  All other values of AValue result in false
being returned.
---------------------------------------------------------}
begin
```

```
  if (AValue = 'TRUE') then CalcBoolValue := true
  else CalcBoolValue := false;

end; {CalcBoolValue}

End. {PropLoad}
```

<u>OBTOBJ.UNT</u>

```
Unit ObtObj;

Interface

  Uses
    WinTypes, WinProcs, WObjects, Strings, PropObj;

  Const
    NAME_SIZE   = 255;
    MAX_OPS     =  15;
    OPS_OVRFLOW =   5;

  Type
    PObjtProp = ^TObjtProp;
    TObjtProp = object(TObject)
      {Attributes}
      Prop : PProp;
      {Methods}
      constructor Init(AProp: PProp);
    end; {TObjtProp}

    PObjt = ^TObjt;
    TObjt = object(TObject)
      {Attributes}
      Name : string[NAME_SIZE];
      ObjtType : ^TPropType;
      Value : ^string;
      Props  : PCollection;
      {Methods}
      constructor Init(AName: string);
      destructor  Done; virtual;
      procedure   SetType(AnObjtType: string); virtual;
      procedure   SetValue(AValue: string); virtual;
      procedure   AddProp(AProp: PProp); virtual;
      function    GetName: string; virtual;
      function    GetType: TPropType; virtual;
      function    GetValue: string; virtual;
      function    GetPropCount: integer; virtual;
      function    GetProp(Idx: integer): PProp; virtual;
    end; {TObjt}

  function StringType(APropType: TPropType): string;

Implementation
```

```
Constructor TObjt.Init(AName: string);
{----------------------------------------------------------
Init: The Name field is assigned the value, AName, the
ObjtType field gets set to nil, and each of the lists is
initialized.
-----------------------------------------------------------}
begin
  Name  := AName;
  ObjtType := nil;
  Value := nil;
  Props   := new(PCollection, Init(MAX_OPS, OPS_OVRFLOW));

end; {TObjt.Init}

Destructor TObjt.Done;
{----------------------------------------------------------
Done: All the lists of TObjt are destroyed, freeing the
memory.
-----------------------------------------------------------}
begin
  if (ObjtType <> nil) then dispose(ObjtType);
  if (Value <> nil) then dispose(Value);
  dispose(Props, done);

end; {TObjt.Done}

Procedure TObjt.SetType(AnObjtType: string);
{----------------------------------------------------------
SetType: ObjtType is given a value of TPropType, based on
the contents of AnObjtType.
----------------------------------------------------------}
begin
  new(ObjtType);
  ObjtType^ := CalcType(AnObjtType);
  SetValue('');

end; {TObjt.SetType}

Procedure TObjt.SetValue(AValue: string);
{----------------------------------------------------------
SetValue: Value is assigned the string, AValue.
----------------------------------------------------------}
begin
  new(Value);
  Value^ := AValue;

end; {TObjt.SetValue}
```

```
Procedure TObjt.AddProp(AProp: PProp);
{-----------------------------------------------------------
AddProp: A new element of the Props list is initialized and
added to the list.
-----------------------------------------------------------}
begin
  Props^.Insert(new(PObjtProp, Init(AProp)));

end; {TObjt.AddProp}

Function TObjt.GetName: string;
{-----------------------------------------------------------
GetName: The value of Name is returned.
-----------------------------------------------------------}
begin
  GetName := Name;

end; {TObjt.GetName}

Function TObjt.GetType: TPropType;
{-----------------------------------------------------------
GetType: The value of ObjtType is returned.
-----------------------------------------------------------}
begin
  GetType := ObjtType^;

end; {TObjt.GetType}

Function TObjt.GetValue: string;
{-----------------------------------------------------------
GetValue: The value of Value is returned.
-----------------------------------------------------------}
begin
  GetValue := Value^;

end; {TObjt.GetValue}

Function TObjt.GetPropCount: integer;
{-----------------------------------------------------------
GetPropCount: An integer indicating the number of TProp
objects in the collection, Props, is returned.
-----------------------------------------------------------}
begin
  GetPropCount := Props^.Count;

end; {TObjt.GetPropCount}
```

```
Function TObjt.GetProp(Idx: integer): PProp;
{---------------------------------------------------------
GetProp: A pointer to the OBJECT PROPERTY (from the list of
PROPERTIES for this OBJECT, Props, and indexed by Idx) is
returned.  If the Idx index is invalid (ie. out of range), a
nil pointer is returned, instead.
---------------------------------------------------------}
var
  p : PObjtProp;

begin
  if ((Idx >= 0) and (Idx < Props^.Count)) then
    begin
      p := Props^.At(Idx);
      GetProp := p^.Prop;
    end {if}
  else GetProp := nil;

end; {TObjt.GetProp}

Constructor TObjtProp.Init(AProp: PProp);
{---------------------------------------------------------
Init: The Prop field is assigned the value, AProp.
---------------------------------------------------------}
begin
  Prop := new(PProp, Init(AProp^.GetName,
      StringType(AProp^.GetType)));

end; {TObjtProp.Init}

Function StringType(APropType: TPropType): string;
{---------------------------------------------------------
StringType: This function returns a string value
corresponding to the TPropType value of APropType.
----------------------------------------------------------}
begin
  case (APropType) of
    PropInt: StringType := 'Integer';
    PropFloat: StringType := 'Float';
    PropBool: StringType := 'Boolean';
    PropStr: StringType := 'String';
    PropDate: StringType := 'Date';
    else StringType := 'Time';
  end; {case}
end; {StringType}
End. {ObtObj}
```

SLOTLIST.UNT

```
Unit SlotList;

{$V-} { Turn off type checking for strings.}

Interface

  Uses
    WObjects, NexFile, SlotObj;

  Const
    MAX_SLOTS     = 50;
    SLOTS_OVRFLOW = 20;

    SLOT_EXT = '.slt';

  Type
    PSlotList = ^TSlotList;
    TSlotList = object(TCollection)
      constructor Init(AMax, AnOvrFlow: integer);
      function FindMatch(AName: string): PSlot; virtual;
      function ToggleSlot(AName: string): boolean; virtual;
      function IsActive(AName: string): boolean; virtual;
    private
      function ProcessPrompt(Line: TLine; i,j: TLineIndex):
           TLine; virtual;
      procedure ProcessSource(Line: TLine; i,j: TLineIndex;
           var s: PSlot; var SlotFile: text); virtual;
      procedure ProcessStrategy(Line: TLine; i,j:
           TLineIndex; var s: PSlot; var SlotFile: text);
           virtual;
      procedure FetchSlots; virtual;
    end; {TSlotList}

Implementation

Constructor TSlotList.Init(AMax, AnOvrFlow: integer);
{ -------------------------------------------------------
Init: The TSlotList object holds a collection of META-SLOTS.
This collection is initialized by calling the parent
constructor and the FetchSlots procedure, which retrieves
the META-SLOT information from disk.
---------------------------------------------------------}
begin
  TCollection.Init(AMax, AnOvrFlow);
  FetchSlots;

end; {TSlotList.Init}
```

```
Function TSlotList.FindMatch(AName: string): PSlot;
{---------------------------------------------------------
FindMatch: The collection of SLOTS is searched sequentially
for a Slot whose name matches the string (AName); if such a
slot is found, this function returns a pointer to the Slot;
otherwise, a nil pointer is returned.
----------------------------------------------------------}
var
  i : integer;
  Match : boolean;
  s : PSlot;

begin
  i := Count -1;
  Match := false;
  while ((i >= 0) and (not Match)) do
    begin
      s := At(i);
      Match := s^.GetName = AName;
      dec(i);
    end; {while}

  if (not Match) then s := nil;
  FindMatch := s;

end; {TSlotList.FindMatch}

Function TSlotList.ToggleSlot(AName: string): boolean;
{ --------------------------------------------------------
ToggleSlot: The SLOT with the name, AName, is first searched
for in the collection.  If it exists, it has its activity
toggled and the function returns true; otherwise, the
function returns false, indicating that no such SLOT exists.
-----------------------------------------------------------}
var
  s : PSlot;
  Found : boolean;

begin
  Found := true;
  s := FindMatch(AName);
  if (s <> nil) then s^.SwitchActivity
  else Found := false;

  ToggleSlot := Found;

end; {TSlotList.ToggleSlot}

Function TSlotList.IsActive(AName: string): boolean;
```

```
{ -----------------------------------------------------------
IsActive: The SLOT by the name of AName is searched for in
the collection.  If it is found, the function returns its
"active" status.  If it is not found then false is returned.
-----------------------------------------------------------}
var
  s : PSlot;

begin
  s := FindMatch(AName);
  if (s <> nil) then IsActive := s^.IsActive
  else IsActive := false;

end; {TSlotList.IsActive}

Function TSlotList.ProcessPrompt(Line: TLine; i,j:
TLineIndex): TLine;
{ -----------------------------------------------------------
{ ProcessPrompt: The remainder of the input Line contains
the prompt for current slot.  This function parses out the
prompt and returns it.
-----------------------------------------------------------}
var
  APrompt : TLine;

begin
  { Extract the remainder of the Line.}
  inc(i);
  ParseWord(Line,NEWLINE,APrompt,i,j);
  ProcessPrompt := APrompt;

end; {TSlotList.ProcessPrompt}

Procedure TSlotList.ProcessSource(Line: TLine; i,j:
      TLineIndex; var s: PSlot; var SlotFile: text);
{ -----------------------------------------------------------
ProcessSource: The remainder of Line holds the Operand for
the Source.  Then, the next two lines in SlotFile hold the
Operand1 and Operand2, respectively.  Once all three values
are obtained, they are added to the current Slot, s.
-----------------------------------------------------------}
var
  Operator, Operand1, Operand2: TLine;

begin
  { Extract the remainder of the Line.}
  inc(i);
  ParseWord(Line,NEWLINE,Operator,i,j);
  readln(SlotFile, Operand1);
  readln(SlotFile, Operand2);
  { Add Source data to Slot.}
  s^.AddSource(Operator, Operand1, Operand2);

end; {TSlotList.ProcessSource}

Procedure TSlotList.ProcessStrategy(Line: TLine; i,j:
      TLineIndex; var s: PSlot; var SlotFile: text);
{ -----------------------------------------------------------
ProcessStrategy: The remainder of the Line holds the left
hand side (Lhs) of the Strategy, and the next line in the
file, SlotFile, holds the right hand side, Rhs.  These two
values are added to the current Slot, s.
----------------------------------------------------------}
var
  Lhs, Rhs: TLine;

begin
  { Extract the remainder of the Line.}
  inc(i);
  ParseWord(Line,NEWLINE,Lhs,i,j);
  readln(SlotFile, Rhs);
  { Add Strategy data to Slot.}
  s^.AddStrategy(Lhs, Rhs);

end; {TSlotList.ProcessStrategy}

Procedure TSlotList.FetchSlots;
{ -----------------------------------------------------------
FetchSlots: First, the NEXPARSE Slot File is opened for
reading.  Then, each line is read and its Line Type is
determined. Depending on the Line Type, the appropriate
procedure is called to extract the data from SlotFile and
added to the Slot, s, which is currently being formed.
Slots are divided by blank lines; if one of these is
encountered, s is added to the collection, and a new Slot is
dynamically allocated.
----------------------------------------------------------}

  var
    i,j      : TLineIndex;
    LineType : TLineType;
    s        : PSlot;
    SlotFile : text;
    SlotLine, Line: TLine;

  begin
    OpenFiles(SlotFile, ConCat(NEX_FILE, SLOT_EXT));
    { Discard the first Line.}
    readln(SlotFile);
    while (not eof(SlotFile)) do
      begin
        { Fetch the name of the slot.}
        readln(SlotFile, Line);
        i := 1; j := length(Line);
        ParseWord(Line,' ',SlotLine,i,j);

        { Allocate a new slot.}
        s := new(PSlot, Init(SlotLine));

        while ((not eof(SlotFile)) and (length(Line) > 0))
          do begin
        { Fetch the Line-type of the next Line in the file.}
            readln(SlotFile, Line);
            i := 1; j := length(Line);
            ParseWord(Line,' ',LineType,i,j);

            { Act according to the LineType.}
            if (LineType = 'PR') then
              begin
                SlotLine := ProcessPrompt(Line,i,j);
                s^.AddPrompt(SlotLine);
              end {if}
            else if (LineType = 'FR') then
              begin
                SlotLine := ProcessPrompt(Line,i,j);
                s^.AddFormat(SlotLine);
              end {else if}
            else if (LineType = 'CN') then
              begin
                SlotLine := ProcessPrompt(Line,i,j);
                s^.AddContext(SlotLine);
              end {else if}
            else if (LineType = 'SR') then
              ProcessSource(Line,i,j,s,SlotFile)
            else if (LineType = 'ST') then
              ProcessStrategy(Line,i,j,s,SlotFile)
            else if (length(LineType) = 0) then Insert(s);
          end;  {while}
      end;  {while}

    CloseFiles(SlotFile);

  end; {TSlotList.FetchSlots}

End. { SlotList}
```

<u>GBLLOAD.UNT</u>

```
Unit GblLoad;
{$V-} { Turn off type checking for strings.}

Interface

  Uses NexFile, PropObj;

  Const
    GBL_EXT = '.gbl';
    MAX_GLOBALS = 15;

  Type
    TGlobal = array[0..MAX_GLOBALS] of boolean;

  Procedure FetchGlobals(var AGlobalList: TGlobal; var
      ASugList: string);

Implementation

Procedure FetchGlobals(var AGlobalList: TGlobal; var
      ASugList: string);
{ -----------------------------------------------------------
FetchGlobals: The GLOBAL variables are stored in an array,
AGlobalList, and the Suggestion List is stored as a string,
ASugList.  This procedure reads a file (GlobalFile)
containing a list of boolean values (one for each GLOBAL
variable) in ASCII format.  These are converted to Pascal
booleans and are stored in AGlobalList.  The last line of
the input file holds the Suggestion List and is read and
stored in ASugList, as is.  Line and i are used for reading
the input file.
Called by: External
-----------------------------------------------------------}
var
  i : integer;
  GlobalFile : text;
  Line : TLine;

begin
  { Open the global file and discard the first line.}
  OpenFiles(GlobalFile, concat(NEX_FILE, GBL_EXT));
  readln(GlobalFile, Line);

  i := 0;
  { Retrieve each of the global variable values.}
  while ((not eof(GlobalFile)) and (i <= MAX_GLOBALS)) do
    begin
```

```
      readln(GlobalFile, Line);
      AGlobalList[i] := CalcBoolValue(Line);
      inc(i);
    end; {while}

  { Retrieve the suggestion list.}
  readln(GlobalFile, ASugList);

  CloseFiles(GlobalFile);

end; {FetchGlobals}

End. {GblLoad}
```

<u>QUESWIND.UNT</u>

```
Unit QuesWind;

Interface

  Uses WinTypes, WinProcs, WObjects, Strings, GrpObj,
       PropObj, ObtObj, SlotObj, SlotList, Expr, Bakchain;

  Const
    BN_HEIGHT = 30; { These determine the appearance }
    BN_WIDTH  = 60; {   of the Buttons in the window.}

    EC_MAX = 10; { These determine the appearance }
    EC_OVRFLOW =   5; { of the Edit Control Objects. }
    EC_WIDTH   = 100;
    EC_HEIGHT  =  30;
    EC_LEN     =  80;
    First_EC   = 201;

    TEXT_HEIGHT = 20; { These constants determine the look}
    TEXT_WIDTH  =  7; { of text displayed in the window.}
   TEXT_SPACE  = 12;
    SUBLINE_LEN = 60;

    INIT_X = 10;   { These constants determine the position}
    INIT_Y = 10;   { and size of the window.}
    TOP_MARGIN  = 10;
    SIDE_MARGIN = 10;

    id_BN1 = 298;       { The button id's.}
    id_BN2 = 299;

  Type
    PPEdit = ^TPEdit;
    TPEdit = object(TObject)
      {Attributes}
      Edit : PEdit;
      SlotNum : integer;
      {Methods}
      constructor Init(AnEdit: PEdit; ASlot: integer);
    end; {TPEdit}

    PPSlot = ^TPSlot;
    TPSlot = object(TObject)
      {Attributes}
      Slot : PSlot;
      FormatedPrompt : string;
      {Methods}
      constructor Init(ASlot: PSlot; AFormatedPrompt:
           string);
    end; {TPSlot}

    PQuesWindow = ^TQuesWindow;
    TQuesWindow = object(TWindow)
      {Attributes}
      Group: PGroup;
      Props: PCollection;
      Objts: PCollection;
      Rules: PCollection;
      Slots: PSlotList;
      GrpSlots: PCollection;
      EditControls: PCollection;
      ConclusionStack: PConclusionStack;

      {Methods}
      constructor Init(AParent: PWindowsObject; ATitle:
           PChar; AGrp: PGroup; APropList, AnObjectList,
           ARuleList: PCollection; ASlotList: PSlotList;
           AConStack: PConclusionStack);
      destructor  Done; virtual;
      procedure   IDBN1(var Msg: TMessage); virtual id_First
           + id_BN1;
      procedure   IDBN2(var Msg: TMessage); virtual id_First
           + id_BN2;
    private
      procedure   GetGrpSlots(var AGrpSlots: PCollection;
                              var NumLines: integer);
    end; {TQuesWindow}

  function  MaxPromptLen(AList: PCollection): word;
  function  NewStatic(Self: PWindowsObject; APrompt: string;
                x,y: integer): PStatic;
  function  NewEdit(Self: PWindowsObject; i,x,y: integer;
                    AValue: string): PEdit;
  procedure SetAttr(SlotCount, LineCount: integer; Len:
                     word; var AnAttr: TWindowAttr);
  function  FormatPrompt(APrompt: string): string;
  procedure AssignObjt(AnObjtList: PCollection; AnObjtName,
                    APropName, AValue: string);

Implementation

Constructor TQuesWindow.Init(AParent: PWindowsObject;
      ATitle: PChar; AGrp: PGroup; APropList, AnObjectList,
      ARuleList: PCollection; ASlotList: PSlotList;
      AConStack: PConclusionStack);
{---------------------------------------------------------
Init: This is the constructor for TQuesWindow.  The Group
and Slots list take on the lists, AGrp and ASlotList,
respectively, which indicate the group that was selected
(along with the slots contained in that group) and the list
of Slots associated with the expert system.  In order to
determine the dimensions of the TQuesWindow, the number and
maximum length of the lines in the prompts to be displayed
is calculated.  Then, each prompt is displayed on the screen
as a static control object.  In addition, for each prompt an
edit control is displayed so that users may enter data at
the prompt.  Finally, two buttons are added to the window.
The 'OK' button is intended to allow users to exit the
window and save any data they may have entered; the 'CANCEL'
button simply closes the window without saving.
----------------------------------------------------------}
var
  s : PPSlot;
  i,x,y,NumLines: integer;
  MaxLen : word;
  Static : PStatic;
  Button : PButton;

begin
  TWindow.Init(AParent, ATitle);
  DisableAutoCreate;

  { Initialize the lists.}
  Group := AGrp;
  Props := APropList;
  Objts := AnObjectList;
  Slots := ASlotList;
  Rules := ARuleList;
  ConclusionStack := AConStack;
  EditControls := new(PCollection, Init(EC_MAX,
EC_OVRFLOW));
  GrpSlots := new(PCollection, Init(MAX_SLOTS,
SLOTS_OVERFLOW));

  GetGrpSlots(GrpSlots, NumLines);
  MaxLen := MaxPromptLen(GrpSlots);
  if (MaxLen > SUBLINE_LEN) then MaxLen := SUBLINE_LEN;

  { Set the attributes for this popup window.}
  SetAttr(GrpSlots^.Count,NumLines, MaxLen, Attr);

  x := INIT_X;
  y := INIT_Y;
  for i:= 0 to GrpSlots^.Count-1 do
    begin
```

```
      s := GrpSlots^.At(i);
      if (s^.Slot^.IsActive) then
        begin
          Static := NewStatic(@Self,s^.FormatedPrompt,x, y);
          EditControls^.Insert(new(PPEdit, Init( NewEdit
            (@Self, i, x+(MaxLen*TEXT_WIDTH)+SIDE_MARGIN, y,
            FetchValue(s^.Slot^.GetName, Objts)), i)));
          y := y + TEXT_HEIGHT * ((length(s^.FormatedPrompt)
            div SUBLINE_LEN)+1) + TEXT_SPACE;
        end; {if}
    end; {for}

  y := y + TOP_MARGIN;
  x := x + (MaxLen*TEXT_WIDTH+EC_WIDTH+3*SIDE_MARGIN) div 2
       - (BN_WIDTH+SIDE_MARGIN);
  Button := new(PButton, Init(@Self, id_BN1, 'OK', x, y,
      BN_WIDTH, BN_HEIGHT, true));
  x := x + BN_WIDTH+SIDE_MARGIN;
  Button := new(PButton, Init(@Self, id_BN2, 'CANCEL', x, y,
      BN_WIDTH, BN_HEIGHT, false));

end; {TQuesWindow.Init}

Destructor TQuesWindow.Done;
{----------------------------------------------------------
Done: All lists associated with this popup window are
destroyed, freeing the allocated memory.
----------------------------------------------------------}
begin
  TWindow.Done;
  dispose(GrpSlots, done);
  dispose(EditControls, done);

end; {TQuesWindow.Done}

Procedure TQuesWindow.IDBN1(var Msg: TMessage);
{----------------------------------------------------------
IDBN1: If the user presses the 'OK' button, then each of the
control objects (e) is checked to see if it has been
modified.  If a change has occurred, the new value is
extracted (TextBuffer) and assigned to the Property or
Object (as determined by the value of j) that corresponds to
the Slot (s) for which this value has been entered.
----------------------------------------------------------}
var
  TextBuffer : array[0..EC_LEN] of char;
  i,j : integer;
  e : PPEdit;
  s : PPSlot;
  SlotName : string;

begin
  { Check each edit control.}
  for i := 0 to EditControls^.Count-1 do
    begin
      e := EditControls^.At(i);

      { If the edit control has been modified, process the
        new data.}
      if ((e^.Edit^.IsModified) and
        (e^.Edit^.GetText(@TextBuffer, EC_LEN) <> 0)) then
        begin
          s := GrpSlots^.At(e^.SlotNum);
          {Slot corresponding to the edit.}
          SlotName := s^.Slot^.GetName;
          j := pos('.',SlotName);    { Property names follow
                                       a '.'.}

          { Is it a Property or an Object?}
          if (j <> 0) then
            AssignObjt(Objts, Copy(SlotName, 1, j-1),
               Copy(SlotName,j+1,length(SlotName)),
               StrPas(TextBuffer))
          else
            AssignObjt(Objts, SlotName, '',
               StrPas(TextBuffer));
        end; {if}
    end; {for}

  CloseWindow;

end; {TQuesWindow.IDBN1}

Procedure TQuesWindow.IDBN2(var Msg: TMessage);
{-------------------------------------------------------
IDBN2: If the 'CANCEL' button is pressed, the window is
closed and any data that was entered is ignored.
-------------------------------------------------------}
begin
  CloseWindow;

end; {TQuesWindow.IDBN2}

Procedure TQuesWindow.GetGrpSlots(var AGrpSlots:
     PCollection; var NumLines: integer);
{-------------------------------------------------------
GetGrpSlots: This procedure creates a list of Slots
(AGrpSlots) that corresponds to the list of Slot names
(Group^.GetSlot(i)).  I is used as an index for traversing
the list of Slots associated with the Group (Group) and s
acts as a temporary pointer for pointing to the Slots to be
added to the list, and gs is used to create new elements of
AGrpSlots.
---------------------------------------------------------}
var
  i : integer;
  s : PSlot;

begin
  s := nil;
  NumLines := 0;
  for i := 0 to Group^.SlotCount-1 do
    begin
      { Find the Slot whose name matches the name of the
        slot in list of slots associated with the group.}
     s := Slots^.FindMatch(StrPas(Group^.GetSlot(i)));
      if (s <> nil) then
        begin
          AGrpSlots^.Insert(new(PPSlot, Init(s,
                            FormatPrompt(s^.GetPrompt))));
          NumLines := NumLines + (length(s^.GetPrompt)
                      div SUBLINE_LEN) + 1;
        end; {if}
    end; {for}

end; {TQuesWindow.GetGrpSlots}

Constructor TPSlot.Init(ASlot: PSlot; AFormatedPrompt:
string);
{---------------------------------------------------------
Init: The TPSlot object, which holds a pointer to a Slot, is
initialized by setting the pointer to the value of ASlot.
---------------------------------------------------------}
begin
  Slot := ASlot;
  FormatedPrompt := AFormatedPrompt;

end; {TPSlot.Init}

constructor TPEdit.Init(AnEdit: PEdit; ASlot: integer);
{---------------------------------------------------------
Init: The TPEdit object is initialized by setting the Edit
Control pointer (Edit) to the value of AnEdit and SlotNum to
ASlot.
---------------------------------------------------------}
begin
  Edit := AnEdit;
  SlotNum := ASlot;

end; {TPEdit.Init}

Function MaxPromptLen(AList: PCollection): word;
{----------------------------------------------------------
MaxPromptLen: The string length of each prompt in the list
of Slots (AList) is examined, and the maximum length is
returned.  I is used as an index to traverse AList, s is
used as a temporary Slot pointer, and Max is used to hold
intermittant maximum string lengths.
Called by: Init
---------------------------------------------------------}
var
  i : integer;
  s : PPSlot;
  Max : word;

begin
  i := AList^.Count -1;
  Max := 0;
  while (i >= 0) do
    begin
      s := AList^.At(i);
      if (Max < length(s^.Slot^.GetPrompt)) then Max :=
length(s^.Slot^.GetPrompt);
      dec(i);
    end; {while}

  MaxPromptLen := Max;

end; {MaxPromptLen}

Function NewStatic(Self: PWindowsObject; APrompt: string;
      x,y: integer): PStatic;
{----------------------------------------------------------
NewStatic: This function returns a pointer to a new Static
Control Object.  The Static will display APrompt at the
location x,y.
Called by: Init
---------------------------------------------------------}

var
  Prompt : array[0..PROMPT_SIZE] of char;
  LineLen, NumLines : integer;

begin
  NumLines := length(APrompt) div SUBLINE_LEN + 1;
  if (NumLines > 1) then LineLen := SUBLINE_LEN
  else LineLen := length(APrompt);
  NewStatic := new(PStatic, Init(Self, -1, StrPCopy(Prompt,
      APrompt), x, y, LineLen * TEXT_WIDTH, TEXT_HEIGHT *
      NumLines, LineLen));

end; {NewStatic}

Function NewEdit(Self: PWindowsObject; i,x,y: integer;
                 AValue: string): PEdit;
{----------------------------------------------------------
NewEdit: A new Edit Control Object is created, and a pointer
to this object is returned.  The Edit is placed at the
position x,y and is given the id, i+First_EC.
Called by: Init
----------------------------------------------------------}
var
  Value : array [0..EC_LEN] of char;

begin
  StrPCopy(Value, AValue);
  NewEdit := new(PEdit, Init(Self, i+First_EC, Value, x,
             y, EC_WIDTH, EC_HEIGHT, EC_LEN, False));

end; {NewEdit}

Procedure SetAttr(SlotCount, LineCount: integer; Len: word;
                  var AnAttr: TWindowAttr);
{----------------------------------------------------------
SetAttr: The attribute record (AnAttr) of the popup window
is initialized, dictating the size and style of the window.
Called by: Init
----------------------------------------------------------}
var
  i,j : integer;

begin
  AnAttr.Style := ws_PopupWindow or ws_Caption or
ws_Visible;
  AnAttr.X := INIT_X;
  AnAttr.Y := INIT_Y;

  i := LineCount * TEXT_HEIGHT + (SlotCount+1) * TEXT_SPACE;
  j := (SlotCount+1) * EC_HEIGHT;
  if (i < j) then i := j;

  AnAttr.H := i + 4 * TOP_MARGIN + BN_HEIGHT;

  i := 3 * SIDE_MARGIN + 2 * BN_WIDTH;
  AnAttr.W := (Len * TEXT_WIDTH) + (3*SIDE_MARGIN) +
      EC_WIDTH;
  if (AnAttr.W < i) then AnAttr.W := i;

end; {SetAttr}

Function FormatPrompt(APrompt: string): string;
{----------------------------------------------------------
FormatPrompt: If the prompt, APrompt, has a length greater
than SUBLINE_LEN, then it is broken into sublines which are
linked together but are separated by carriage return/line
feed characters.  The single string of concatenated sublines
is returned.
----------------------------------------------------------}
var
  i : integer;
  Result : string;

begin
  Result := '';
  { Break APrompt into sublines of no longer than
     SUBLINE_LEN.}
  while (length(APrompt) > SUBLINE_LEN) do
    begin
      i := SUBLINE_LEN;
      while ((i > 0) and (APrompt[i] <> ' ')) do dec(i);
      Result := concat(Result, copy(APrompt, 1, i)) +#13#10;
      APrompt := copy(APrompt, i+1, length(APrompt)-i + 1);
    end; {while}

  Result := concat(Result, APrompt);
  FormatPrompt := Result;

end; {FormatPrompt}

Procedure AssignObjt(AnObjtList: PCollection; AnObjtName,
      APropName, AValue: string);
{----------------------------------------------------------
AssignObjt: This procedure inserts AValue as the value of
the OBJECT, whose name is AnObjtName (ObjtMatch).  If the
OBJECT itself does not hold a value, but has PROPERTIES
instead, then the OBJECT PROPERTY matching APropName is
found (PropMatch) and AValue is stored in it.
----------------------------------------------------------}
var
  FoundObjt : PObjt;
  Prop : PObjtProp;

function ObjtMatch(o: PObjt): boolean; far;
  begin
```

```
    ObjtMatch := o^.GetName = AnObjtName;
  end; {ObjtMatch}

function PropMatch(p: PObjtProp): boolean; far;
  begin
    PropMatch := p^.Prop^.GetName = APropName;
  end; {PropMatch}

begin
  FoundObjt := AnObjtList^.FirstThat(@ObjtMatch);
  if (FoundObjt <> nil) then
    if (APropName = '') then FoundObjt^.SetValue(AValue)
    else
      begin
        Prop := FoundObjt^.Props^.FirstThat(@PropMatch);
        Prop^.Prop^.SetValue(AValue);
      end; {else}

end; {AssignObjt}

End. {QuesWind}
```

<u>**EXPR.UNT**</u>

```
Unit Expr;
{$V-}  { Turn off type checking for strings.}

Interface

  Uses
    WObjects, Strings, NexFile, PropObj, ObtObj, Stack,
     ItemObj, Funcs;

  Type
    SOperators = set of char;

    PReal = ^Real;

  Const
    OperatorSet : SOperators = ['-', '+', '/', '*'];

  procedure InsertObjt(AnObjtList: PCollection; AnObjt:
      PObjt);
  function FindObjt(AnObjtList: PCollection; AnObjt:
      string): PObjt;
  procedure AssignValue(AnObjtList: PCollection; ASource,
      ADest: string);
  function FetchValue(AName: string; AnObjtList:
      PCollection): string;
  function FetchType(AName: string; AnObjtList:
      PCollection): TPropType;
  function UnQuoteString(AQuotedString: string): string;
  function Evaluate(AnExpr: string; AnObjtList:
      PCollection): PResult;
  function Interpretation(AnObjtList: PCollection;
      AnInterpret: string): string;
  function ExtractClass(AClass: string): string;
  procedure ExtractCreateLists(var CList, OList:
      PCollection; AList: string);
  function DoFuncs(AnObjtList: PCollection; AFnNo: integer;
                   AnOperandList: PCollection): PResult;

Implementation

Procedure InsertObjt(AnObjtList: PCollection; AnObjt:
PObjt);
{-----------------------------------------------------------
InsertObjt: This procedure adds the OBJECT, AnObjt, to the
collection of OBJECTS, AnObjtList, as long as it is not
already found in the list.  If it is already in the list,
then this procedure has no effect.  To find the location to
insert, a binary search is used.
------------------------------------------------------------}
var
  First, Last, i : integer;
  Found : boolean;
  o : PObjt;

begin
  { Initialize search variables.}
  First := 0;
  Last  := AnObjtList^.Count -1;
  Found := false;
  { Search until successful or completed list.}
  while ((First <= Last) and (not Found)) do
    begin
      i := (First+Last) div 2;
      o := AnObjtList^.At(i);
      if (AnObjt^.GetName < o^.GetName) then Last := i-1
      else if (AnObjt^.GetName>o^.GetName) then First := i+1
      else Found := true;
    end; {while}

  { If the object is not already in the list, insert it.}
  if (First>Last) then AnObjtList^.AtInsert(First, AnObjt);

end; {InsertObjt}

Function FindObjt(AnObjtList:PCollection;AnObjt:string):
      PObjt;
{----------------------------------------------------------
FindObjt: This function performs a binary search on the
list, AnObjtList, searching for an OBJECT with the name
AnObjt.  If successful, a pointer to the OBJECT is returned;
otherwise, a nil pointer is returned.  First, Last, i, o,
and Found are used in the search.
-------------------------------------------------------------}
var
  First, Last, i : integer;
  Found : boolean;
  o : PObjt;

begin
  { Initialize search variables.}
  First := 0;
  Last  := AnObjtList^.Count -1;
  Found := false;
  { Search until successful or completed list.}
  while ((First <= Last) and (not Found)) do
    begin
```

```
      i := (First+Last) div 2;
      o := AnObjtList^.At(i);
      if (AnObjt < o^.GetName) then Last := i-1
      else if (AnObjt > o^.GetName) then First := i+1
      else Found := true;
    end; {while}

  { If successful, return a pointer to the OBJECT;
otherwise, return nil.}
  if (First <= Last) then FindObjt := o
  else FindObjt := nil;

end; {FindObjt}

Procedure AssignValue(AnObjtList: PCollection; ASource,
ADest: string);
{----------------------------------------------------------
AssignValue: ADest holds the name of an OBJECT or OBJECT
PROPERTY in which to store the string, ASource.  The target
OBJECT is searched for in AnObjtList, and, if necessary, the
PROPERTY is found by invoking PropMatch.
----------------------------------------------------------}
var
  PropName : string;
  i : integer;
  o : PObjt;
  p : PObjtProp;

function PropMatch(p: PObjtProp): boolean; far;
  begin
    PropMatch := p^.Prop^.GetName = PropName;
  end; {PropMatch}

begin
  { Determine if it is an OBJECT or OBJECT PROPERTY.}
  i := pos('.', ADest);
  if (i = 0) then
    begin                  { Case OBJECT.}
      o := FindObjt(AnObjtList, ADest);
      if (o <> nil) then o^.SetValue(ASource);
    end {if}
  else
    begin                  { Case OBJECT PROPERTY.}
      o := FindObjt(AnObjtList, copy(ADest, 1, i-1));
      PropName := copy(ADest, i+1, length(ADest));
      p := o^.Props^.FirstThat(@PropMatch);
      p^.Prop^.SetValue(ASource);
    end; {else}
end; {AssignValue}
```

```
Function FetchValue(AName: string; AnObjtList: PCollection):
      string;
{-------------------------------------------------------
FetchValue: First, it is determined whether or not AName is
the name of an OBJECT or the name of an OBJECT.PROPERTY.  If
the former, then the OBJECT is found in AnObjtList and its
value is retrieved.  Otherwise, the OBJECT is found and then
the PROPERTY within that OBJECT.  Then, the PROPERTIES value
is retrieved.
-------------------------------------------------------}
var
  PropName,ObjtName : string;
  FoundProp: PObjtProp;
  FoundObjt: PObjt;
  i : integer;

function PropMatch(p: PObjtProp): boolean; far;
  begin
    PropMatch := p^.Prop^.GetName = PropName;
  end; {Match}

begin
  i := pos('.', AName);
  if (i <> 0) then
    begin
      ObjtName := copy(AName, 1, i-1);
      FoundObjt:= FindObjt(AnObjtList, ObjtName);
      if (FoundObjt <> nil) then
        begin
          PropName := copy(AName, i+1, length(AName));
          FoundProp :=
            FoundObjt^.Props^.FirstThat(@PropMatch);
          if (FoundProp <> nil) then
            FetchValue := FoundProp^.Prop^.GetValue
          else FetchValue := '';
        end {if}
      else FetchValue := '';
    end {if}
  else
    begin
      ObjtName := AName;
      FoundObjt := FindObjt(AnObjtList, ObjtName);
      if (FoundObjt <> nil) then
        FetchValue := FoundObjt^.GetValue
      else FetchValue := '';
    end; {else}

end; {FetchValue}
```

```
Function FetchType(AName: string; AnObjtList: PCollection):
TPropType;
{---------------------------------------------------------
FetchType: First, it is determined whether or not AName is
the name of an OBJECT or the name of an OBJECT.PROPERTY.  If
the former, then the OBJECT is found in AnObjtList and its
type is retrieved.  Otherwise, the OBJECT is found and then
the PROPERTY within that OBJECT.  Then, the PROPERTIES type
is retrieved.
---------------------------------------------------------}
var
  PropName,ObjtName : string;
  FoundProp: PObjtProp;
  FoundObjt: PObjt;
  i : integer;

function PropMatch(p: PObjtProp): boolean; far;
  begin
    PropMatch := p^.Prop^.GetName = PropName;
  end; {Match}

begin
  if ((AName[1] = #39) or (AName[1] = '\')) then
    AName := Interpretation(AnObjtList, AName);

  i := pos('.', AName);
  if (i <> 0) then
    begin
      ObjtName := copy(AName, 1, i-1);
      FoundObjt:= FindObjt(AnObjtList, ObjtName);
      if (FoundObjt <> nil) then
        begin
          PropName := copy(AName, i+1, length(AName));
          FoundProp :=
            FoundObjt^.Props^.FirstThat(@PropMatch);
          if (FoundProp <> nil) then FetchType :=
            FoundProp^.Prop^.GetType
          else FetchType := PropUnknown;
        end {if}
      else FetchType := PropUnknown;
    end {if}
  else
    begin
      ObjtName := AName;
      FoundObjt := FindObjt(AnObjtList, ObjtName);
      if (FoundObjt <> nil) then FetchType :=
FoundObjt^.GetType
      else FetchType := PropUnknown;
    end; {else}
```

```
end; {FetchType}

Function NuNumeric(ANumber: real): PReal;
{----------------------------------------------------------
NuNumeric: This function allocates memory to hold a real
variable and stores the real, ANumber, in this memory.  A
pointer to the allocated memory is then returned.
----------------------------------------------------------}
var
  r : PReal;

begin
  new(r);
  r^ := ANumber;
  NuNumeric := r;

end; {NuNumeric}

Function NuString(AString: string): PString;
{----------------------------------------------------------
NuString: This function allocates memory to hold a string
variable and stores the string, AString, in this memory.  A
pointer to the allocated memory is then returned.
----------------------------------------------------------}
var
  s : PString;

begin
  new(s);
  s^ := AString;
  NuString := s;

end; {NuString}

Function NuOperator(AnOperator: char): PChar;
{----------------------------------------------------------
NuOperator:
----------------------------------------------------------}
var
  c : PChar;

begin
  new(c);
  c^ := AnOperator;
  NuOperator := c;

end; {NuOperator}

Function UnQuoteString(AQuotedString: string): string;
```

```
{----------------------------------------------------------
UnQuoteString: The string passed to this function is
contained by double quotes.  The purpose of this function is
to strip away the double quotes and return the resultant
string.
----------------------------------------------------------}
var
  i, j : TLineIndex;
  UnQuotedString : string;

begin
  i := 2;
  j := length(AQuotedString);
  ParseWord(AQuotedString, '"', UnQuotedString, i, j);
  UnQuoteString := UnQuotedString;

end; {UnQuoteString}

Function CalcExpr(AnOperator: PChar; AnOperand1, AnOperand2:
PReal): real;
{----------------------------------------------------------
CalcExpr: Based on the operator character (*,/,+,-) stored
in memory location pointed to by AnOperator, an arithmetic
operation is performed on the two real numbers, pointed to
by AnOperand1 and AnOperand2.  The result of this operation
is returned.
----------------------------------------------------------}
begin
  case (AnOperator^) of
    '*' : CalcExpr := AnOperand2^ * AnOperand1^;
    { Notice that there is no division-by-zero check!}
    '/' : CalcExpr := AnOperand2^ / AnOperand1^;
    '+' : CalcExpr := AnOperand2^ + AnOperand1^;
    '-' : CalcExpr := AnOperand2^ - AnOperand1^;
    else  CalcExpr := 0;
  end; {case}

end; {CalcExpr}

Procedure AddOperator(AnOperator: PChar; var AStack, BStack:
PStack);
{----------------------------------------------------------
AddOperator: The operator character, pointed to by
AnOperator, is to be added to the operator stack, AStack.
However, if the operator currently on the top of AStack is
of higher priority than the new operator, the old operator
must be evaluated before the new one can be added.  To
evaluate an operator, CalcExpr is called with the old
operator and two operands (these come for the operand stack,
BStack).  The result is placed on the top of BStack.  The
process is repeated as long as the operator on top of AStack
is of higher priority than AnOperator.  In the end, however,
AnOperand is placed on the top of AStack.
---------------------------------------------------------}
var
  NuValue : PReal;
  Top : PChar;

begin
  { If the operator is not * or / and the operator stack is
    not empty, check the operator on the top of the stack to
    see if it is of higher priority than the new operator.}

  if ((AnOperator^<>'*') and (AnOperator^<>'/') and not
      (AStack^.IsEmpty)) then repeat begin
        Top := AStack^.Top;
        if ((Top^ = '*') or (Top^ = '/')) then
        { Higher priority ops.}
          begin
            new(NuValue);
            { Evaluate the operator on top of the stack.}
            NuValue^ := CalcExpr(AStack^.Pop, BStack^.Pop,
                BStack^.Pop);
            BStack^.Push(NuValue);
          end; {if}
      end; {repeat}
    until (((Top^ <> '*') and (Top^ <> '/')) or
(AStack^.IsEmpty));

  { Push the new operator onto the operator stack.}
  AStack^.Push(AnOperator);

end; {AddOperator}

Function Evaluate(AnExpr: string; AnObjtList: PCollection):
PResult;
{--------------------------------------------------------
Evaluate: The Evaluate function has two distinct components.
The first parses the expression, AnExpr, into its composite
operands and operators.  If the expression is a constant of
any type, the function ceases by returning the constant.
The expression may contain references to OBJECTS.  If this
is the case, the value of the OBJECT is retrieved, and this
is placed on the operand stack.  If,  the expression is an
arithmetic expression, then the OperandStack and perhaps the
OperatorStack will be non-empty.  In this case, the
expression is evaluated by popping the two stacks (2:1
OperandStack:OperatorStack) until both stacks are exhausted.
```

```
The final result is returned.
-------------------------------------------------------------------}
var
  IsReal : boolean;
  i,j,ELen,FLen : TLineIndex;
  OperandList : PCollection;
  SubExpr, StrResult, Func, Operand : string;
  ArrResult : array [0..255] of char;
  Num : real;
  Result : PReal;
  SResult : PChar;
  ErrCode : integer;
  OperandStack, OperatorStack: PStack;

begin
  {Pad the expression.}
  AnExpr := AnExpr + ' ';

  {Initialize the two stacks and the number flag.}
  IsReal := false;
  OperandStack := new(PStack, Init);
  OperatorStack := new(PStack, Init);

  {Parse the Expression.}
  ELen := length(AnExpr);
  i := 1;
  while (i < ELen) do
    begin
      { Extract subexpression.}
      ParseWord(AnExpr, ' ', SubExpr, i, ELen);
      val(SubExpr, Num, ErrCode);

      { Extract possible function name.}
      j := 1;
      FLen := length(SubExpr);
      ParseWord(SubExpr, '(', Func, j, FLen);

      if (ErrCode = 0) then
        begin
          IsReal := true;
          OperandStack^.Push(NuNumeric(Num));
        end {if}
      else if (SubExpr[1] in OperatorSet) then
        AddOperator(NuOperator(SubExpr[1]), OperatorStack,
           OperandStack)
      else if (SubExpr[1] = '"') then
        begin
          Evaluate := new(PResult,
           Init(UnQuoteString(SubExpr), 'String'));
        exit;
      end {elseif}
    else if ((SubExpr = 'TRUE') or (SubExpr = 'FALSE'))
      then begin
        Evaluate := new(PResult, Init(SubExpr,
         'Boolean'));
        exit;
      end {elseif}
    else if (IsFunc(Func) <> -1) then
      begin
        OperandList := new(PCollection, Init(MAX_OPERANDS,
          OPERAND_OVRFLOW));
        while (j < FLen) do
          begin
            ParseOperand(SubExpr, Operand, j);
            if (Operand <> '') then
              OperandList^.Insert(new(PItem,
                   Init(Operand)));
          end; {while}

        Evaluate := DoFuncs(AnObjtList, IsFunc(Func),
              OperandList);
        dispose(OperandList, done);
        exit;
      end
    else
      begin
        { Handle OBJECT references.}
        if ((SubExpr[1] = #39) or (SubExpr[1] = '\')) then
          StrResult := FetchValue(Interpretation
              (AnObjtList, SubExpr), AnObjtList)
        else StrResult := FetchValue(SubExpr, AnObjtList);

        val(StrResult, Num, ErrCode);
        StrPCopy(ArrResult, StrResult);
        if (ErrCode = 0) then
          begin
            OperandStack^.Push(NuNumeric(Num));
            IsReal := true;
          end {if}
        else OperandStack^.Push(StrNew(ArrResult));
      end; {else}

    inc(i);
  end; {while}

{If the expression was non-arithmetic, then return the
 string on the operand stack.}
if ((OperatorStack^.IsEmpty) and (not IsReal)) then
```

```
    begin
      SResult := OperandStack^.Pop;
      if (SResult <> nil) then
      begin
        Evaluate := new(PResult, Init(StrPas(SResult),
          StringType(FetchType(SubExpr, AnObjtList))))
      end
      else Evaluate := new(PResult, Init('', StringType(
           FetchType(SubExpr, AnObjtList))));
    end {if}
  else
    begin
      {If it's an arithmetic expression, process the
       operators and operands.}
      while not (OperatorStack^.IsEmpty) do
        OperandStack^.Push(NuNumeric (CalcExpr
          (OperatorStack^.Pop,OperandStack^.Pop,
           OperandStack^.Pop)));
      Result := OperandStack^.Pop;
      str(Result^, StrResult);
      Evaluate := new(PResult, Init(StrResult, 'Float'));
    end; {else}

end; {Evaluate}

Function Interpretation(AnObjtList: PCollection;
      AnInterpret: string): string;
{---------------------------------------------------------
Interpretation: This function takes as its input the string
AnInterpret, which is a NEXPERT OBJECT Interpretation.  The
function then derives the actual value of the Interpretation
by first decomposing the string into its composite parts:
(a) the root string (Prefix), (b) the attribute name
(Attribute), and (c) the suffix (Suffix).  Then the value
stored in the attribute with the name Attribute is
determined, by searching AnObjtList, and this value is
concatenated with the root string and the suffix to form the
end result.  Max and i are used in the parsing of the
Interpretation.
---------------------------------------------------------}
var
  Prefix, Suffix, Attribute : string;
  i, Max : TLineIndex;

begin
  i := 1;
  Max := length(AnInterpret);
  Prefix := '';
  { Parse the root string if there is one.}
  if (AnInterpret[1] = #39) then
    begin
      inc(i);
      ParseWord(AnInterpret, #39, Prefix, i, Max);
      inc(i);
    end; {if}

  { Parse the attribute name.}
  inc(i);
  ParseWord(AnInterpret, '\', Attribute, i, Max);

  { Parse the suffix.  If there isn't one, this will be an
     empty string.}
  inc(i);
  ParseWord(AnInterpret, #13, Suffix, i, Max);

  { Concatenate the subcomponents of the Interpretation.}
  Interpretation := concat(Prefix, FetchValue(Attribute,
      AnObjtList), Suffix);

end; {Interpretation}

Function ExtractClass(AClass: string): string;
{---------------------------------------------------------
ExtractClass: The name of a CLASS is stored in AClass; it is
enclosed by vertical lines (eg. ¦ClassName¦).  This function
parses out the name of the CLASS, temporarily storing it in
ClassName, and returns it.  Max and i are used in the
parsing of AClass.
---------------------------------------------------------}
var
  i, Max : TLineIndex;
  ClassName : string;

begin
  i := 2;
  Max := length(AClass);
  ParseWord(AClass, '¦', ClassName, i, Max);
  ExtractClass := ClassName;

end; {ExtractClass}

Procedure ExtractCreateLists(var CList, OList: PCollection;
      AList: string);
{---------------------------------------------------------
ExtractCreateLists: The string, AList, holds a list of CLASS
and OBJECT names.  This procedure parses out these names and
adds them to the lists, CList (for CLASS names) and OList
(for OBJECT names).  Note: as it is, this procedure does not
yet handle OBJECT names.
---------------------------------------------------------}
const
  MAX_LIST     = 10;
  LIST_OVRFLOW =  5;

var
  i, Max : TLineIndex;
  ClassName : string;

begin
  { Initialize the lists.}
  CList := new(PCollection, Init(MAX_LIST, LIST_OVRFLOW));
  OList := new(PCollection, Init(MAX_LIST, LIST_OVRFLOW));

  i := 1;
  Max := length(AList);
  { Parse the CLASS names out of the list.}
  while (i < Max-2) do
    begin
      if (AList[i] = '¦') then
        begin
          inc(i);
          ParseWord(AList, '¦', ClassName, i, Max);
          { Add the ClassName to the list.}
          CList^.Insert(new(PItem, Init(ClassName)));
        end; {if}
      if (i < Max-2) then i := i+2;
    end; {while}

end; {ExtractCreateLists}

Function DoFuncs(AnObjtList: PCollection; AFnNo: integer;
                 AnOperandList: PCollection): PResult;
{----------------------------------------------------------
DoFunc:
----------------------------------------------------------}
var
  i, Count, ErrCode : integer;
  Operand, Operand2 : PItem;
  Result, Result2   : real;
  ValueList: PCollection;
  Value    : PResult;

begin
  ValueList := new(PCollection, Init(MAX_OPERANDS,
OPERAND_OVRFLOW));
  i := 0;
  Count := AnOperandList^.Count;
  while (i < Count) do
    begin
      Operand := AnOperandList^.At(i);
      Value := Evaluate(Operand^.GetValue, AnObjtList);
      ValueList^.Insert(new(PItem, Init(Value^.GetValue)));
      inc(i);
    end; {while}

  case (AFnNo) of

    FTDay:
      begin
        Operand := ValueList^.At(0);
        DoFuncs := new(PResult,
Init(FnDay(Operand^.GetValue), 'String'));
      end; {FTDay}

    FTInt2Str:
      begin
        Operand := ValueList^.At(0);
        val(Operand^.GetValue, Result, ErrCode);
        DoFuncs := new(PResult,
Init(FnInt2Str(Round(Result)), 'String'));
      end; {FTInt2Str}

    FTNow: DoFuncs := new(PResult, Init(FnNow, 'Date'));

    FTStrCat:
      begin
        Operand := ValueList^.At(0);
        Operand2:= ValueList^.At(1);
        DoFuncs := new(PResult,
Init(FnStrCat(Operand^.GetValue,
                   Operand2^.GetValue), 'String'));
      end; {FTStrCat}

    FTSubString:
      begin
        Operand := ValueList^.At(0);
        Operand2:= ValueList^.At(1);
        val(Operand2^.GetValue, Result, ErrCode);
        Operand2:= ValueList^.At(2);
        val(Operand2^.GetValue, Result2, ErrCode);
        DoFuncs := new(PResult, Init(FnSubString
           (Operand^.GetValue, round(Result), round(
Result2)), 'String'));
      end; {FTSubString}
    else DoFuncs := new(PResult, Init('TRUE','Boolean'));
  end; {case}
end; {DoFuncs}
End. {Expr}
```

FUNCS.UNT

```
Unit Funcs;
{$V-}     { Turn off string type checking.}

Interface

  Uses
    WinDos, WObjects, NexFile, PropObj, ItemObj;

  Const
    NUM_FUNCS = 5;

    MAX_OPERANDS = 10;
    OPERAND_OVRFLOW = 5;

    MonthTbl : array[1..12] of integer = (
      31, 28, 31, 30, 31, 30,
      31, 31, 30, 31, 30, 31);

    FuncTbl : array[0..NUM_FUNCS-1] of string = (
      'DAY',
      'INT2STR',
      'NOW',
      'STRCAT',
      'SUBSTRING');

    FTDay       = 0;
    FTInt2Str   = 1;
    FTNow       = 2;
    FTStrCat    = 3;
    FTSubString = 4;

  Type
    PResult = ^TResult;
    TResult = object(TObject)
      {Attributes}
      Value: string;
      PropType: TPropType;
      {Methods}
      constructor Init(AValue, APropType: string);
      function    GetValue: string; virtual;
      function    GetType : TPropType; virtual;
    end; {TResult}

  function IsFunc(AFunc: string): integer;
  function FnDay(ADate: string): string;
  function FnInt2Str(AnInteger: integer): string;
  function FnNow: string;
  function FnStrCat(AString, AnotherString: string): string;
  function FnSubString(AString: string; AStart, ACount:
      integer): string;
  function NextParenthesis(AList: string; Idx: TLineIndex):
      TLineIndex;
  procedure ParseOperand(var AList, AnOperand: string; var
      Idx: TLineIndex);

Implementation

Constructor TResult.Init(AValue, APropType: string);
{----------------------------------------------------------
Init: The string, AValue, is stored in the attribute, Value.
The string APropType is used to calculate a TPropType
equivalent, which is stored in the PropType attribute.
----------------------------------------------------------}
begin
  Value := AValue;
  PropType := CalcType(APropType);

end; {TResult.Init}

Function TResult.GetValue: string;
{----------------------------------------------------------
GetValue: The Value string is returned.
----------------------------------------------------------}
begin
  GetValue := Value;

end; {TResult.GetValue}

Function TResult.GetType: TPropType;
{----------------------------------------------------------
GetType: The value of the PropType attribute is returned.
----------------------------------------------------------}
begin
  GetType := PropType;

end; {TResult.GetType}

Function IsFunc(AFunc: string): integer;
{----------------------------------------------------------
IsFunc: This function returns either -1 if the string AFunc
is not the name of a NEXPERT OBJECT function, or an integer
index to the NEXPERT OBJECT function name in the array of
such names, FuncTbl.
----------------------------------------------------------}
var
  i : integer;
```

```
begin
  i := 0;
  while ((i < NUM_FUNCS) and (FuncTbl[i] <> AFunc))
      do inc(i);
  if (i >= NUM_FUNCS) then IsFunc := -1
  else IsFunc := i;

end; {IsFunc}

Function FnDay(ADate: string): string;
{---------------------------------------------------------
FnDay: This Function extracts the Day value from the date
string, ADate.  If the Day position of the string does not
hold a numeric value, then a value of 0 is returned to
indicate an error.
---------------------------------------------------------}
var
  i, j : TLineIndex;
  Day  : string;
  ErrCode: integer;
  DayVal : real;

begin
  j := length(ADate);
  i := pos(' ', ADate);
  i := i + pos(' ', copy(ADate, i+1, j-i)) + 1;
  ParseWord(ADate, ' ', Day, i, j);

  val(Day, DayVal, ErrCode);
  if (ErrCode = 0) then FnDay := Day
  else FnDay := '0';

end; {FnDay}

Function FnInt2Str(AnInteger: integer): string;
{---------------------------------------------------------
FnInt2Str: This function returns the string value of the
integer, AnInteger.
---------------------------------------------------------}
var
  AString : string;

begin
  str(AnInteger, AString);
  FnInt2Str := AString;

end; {FnInt2Str}

Function FnNow: string;
```

```
{------------------------------------------------------------
FnNow: This function returns a date string containing the
current calendar date and time.
------------------------------------------------------------}
var
  Year, Month, Day, Hour, Minute, Second, Garbage : word;
  StrYr, StrMth, StrDay, StrHr, StrMin, StrSec : string;

begin
  GetDate(Year, Month, Day, Garbage);
  GetTime(Hour, Minute, Second, Garbage);

  str(Year, StrYr);
  str(Month, StrMth);
  str(Day, StrDay);
  str(Hour, StrHr);
  str(Minute, StrMin);
  str(Second, StrSec);

  FnNow := concat(StrYr, ' ', StrMth, ' ', StrDay, ' ',
      StrHr, ' ', StrMin, ' ', StrSec);

end; {FnNow}

Function FnStrCat(AString, AnotherString: string): string;
{------------------------------------------------------------
FnStrCat: The FnStrCat NEXPERT OBJECT function which
concatenates two strings, AString and AnotherString; this
function performs this operation by invoking the concat
Pascal function.
------------------------------------------------------------}
begin
  FnStrCat := concat(AString, AnotherString);

end; {FnStrCat}

Function FnSubString(AString: string; AStart, ACount:
      integer): string;
{------------------------------------------------------------
FnSubString: A substring of AString, beginning with the
character indexed by AStart and continuing for ACount
characters is returned.
------------------------------------------------------------}
begin
  FnSubString := copy(AString, AStart+1, ACount);

end; {FnSubString}

Function NextParenthesis(AList: string; Idx: TLineIndex):
      TLineIndex;
{---------------------------------------------------------
NextParenthesis: This is a recursive function that is called
when an opening parenthesis is found in the string, AList.
It increments Idx until the matching closing parenthesis is
found.  If another opening parenthesis is encountered, then
a recursive call is made to this function.
---------------------------------------------------------}
begin
  inc(Idx);
  while (AList[Idx] <> ')') do
    if (AList[Idx] = '(') then Idx := NextParenthesis(AList,
      Idx)
    else inc(Idx);
  inc(Idx);
  NextParenthesis := Idx;

end; {NextParenthesis}

Procedure ParseOperand(var AList, AnOperand: string; var
Idx: TLineIndex);
{---------------------------------------------------------
ParseOperand: This procedure parses the next operand from
the string AList and stores it it the string AnOperand.  The
integer, Idx, marks the starting character of AList at which
to begin parsing.
---------------------------------------------------------}
var
  OldIdx, Max : TLineIndex;

begin
  inc(Idx);
  OldIdx := Idx;
  Max := length(AList);

  { Keep incrementing Idx until a closing parenthesis is
    found or a separating comma is encountered. }
  while ((Idx <= Max) and (AList[Idx] <> ',') and
      (AList[Idx] <> ')')) do
    if (AList[Idx] = '(') then
      Idx := NextParenthesis(AList,Idx)
    else inc(Idx);
  { Copy the substring representing the operand.}
  AnOperand := copy(AList, OldIdx, Idx-OldIdx);
end; {ParseOperand}
End. {Funcs}
```

<u>TSLOTGRP.UNT</u>

```
Unit TSlotGrp;

Interface

  Uses
    GrpObj, Strings, WObjects, NexFile;

  Const
    SNameSize = 100;
    SGRP_EXT = '.sgp';

  Type
    TSName = array [0..SNameSize] of char;

    PSlotGrps = ^TSlotGrps;
    TSlotGrps = object(TCollection)
      {Methods}
      constructor Init(AMax, AnOvrFlow: integer);
    private
      procedure   FetchGrps; virtual;
    end; {TSlotGrps}

Implementation

Constructor TSlotGrps.Init(AMax, AnOvrFlow: integer);
{----------------------------------------------------------
Init: The special collection object, TSlotGrps, is
initialized by first calling the ancestral constructor, and
then calling a routine that loads slot group data from a
disk file.
---------------------------------------------------------}
begin
  TCollection.Init(AMax, AnOvrFlow);
  FetchGrps;

end; {TSlotGrps}

Procedure TSlotGrps.FetchGrps;
{----------------------------------------------------------
FetchGrps: This procedure reads the list of group names and
SLOT names, within each group, from the input file, AFile.
Each group is added to the list of groups that this object
holds.
---------------------------------------------------------}
  var
    p     : PGroup;
    AName : TGName;
    ASlot : TSName;
    AFile : text;

  begin
    { Open the input file.}
    OpenFiles(AFile, concat(NEX_FILE, SGRP_EXT));
    while (not eof(AFile)) do
      begin
        { Read the group name.}
        readln(AFile, AName);
        p := new(PGroup, Init(AName));

        repeat
          begin
            { Read the name of each SLOT in the group.}
            readln(AFile, ASlot);
            { A blank line separates groups.}
            if (StrLen(ASlot) > 1) then p^.AddSlot(ASlot);
          end; {repeat}
        until ((eof(AFile)) or (StrLen(ASlot) <= 1));

        Insert(p);
      end; {while}

      CloseFiles(AFile);

  end; {TSlotGrps.FetchGrps}

End. {TSlotGrp}
```

<u>ITEMOBJ.UNT</u>

```
Unit ItemObj;

Interface

  Uses WObjects;

  Const
    ITEM_SIZE = 255;

  Type
    PItem = ^TItem;
    TItem = object(TObject)
      {Attributes}
      Value : string[ITEM_SIZE];
      {Methods}
      constructor Init(AValue: string);
      function    GetValue: string; virtual;
    end; {TItem}

Implementation

Constructor TItem.Init(AValue: string);
{----------------------------------------------------------
Init: The attribute, Value, is assigned the string, AValue.}
---------------------------------------------------------}
begin
  Value := AValue;

end; {TItem.Init}

Function TItem.GetValue: string;
{----------------------------------------------------------
GetValue: The string, Value, is returned.
---------------------------------------------------------}
begin
  GetValue := Value;
 end; {TItem.GetValue}
End. {ItemObj}
```

<u>**SLOTOBJ.UNT**</u>

```
Unit SlotObj;

{$V-} { Turn off type checking for strings.}

Interface

  Uses WinTypes, WinProcs, WObjects, Strings;

  Const
    NAME_SIZE     = 255;
    PROMPT_SIZE   = 255; { These constants dictate the
string sizes for}
    FORMAT_SIZE   =  40; {   the slot components.
         }
    SOURCE_SIZE   = 100;
    CONTEXT_SIZE  = 255;
    STRATEGY_SIZE = 100;

    MAX_SRC=5; { These constants dictate the list size for}
    MAX_CON=5; { the Source, Context, and Strategy lists.}
    MAX_STRAT =  5;
    SRC_OVRFLOW   = 5;
    CON_OVRFLOW   = 5;
    STRAT_OVRFLOW = 5;

  Type
    PPrompt = ^TPrompt;
    TPrompt = string[PROMPT_SIZE];

    PFormat = ^TFormat;
    TFormat = string[FORMAT_SIZE];

    PSource = ^TSource;
    TSource = object(TObject)
      {Attributes}
      Operator : string[SOURCE_SIZE];
      Operand1 : string[SOURCE_SIZE];
      Operand2 : string[SOURCE_SIZE];
      {Methods}
      constructor Init(AnOperator, AnOperand1, AnOperand2:
          string);
    end; {TSource}

    PContext = ^TContext;
    TContext = object(TObject)
      {Attributes}
      Context : string[CONTEXT_SIZE];
      {Methods}
      constructor Init(AContext: string);
    end; {TContext}

    PStrategy = ^TStrategy;
    TStrategy = object(TObject)
      {Attributes}
      Lhs : string[STRATEGY_SIZE];
      Rhs : string[STRATEGY_SIZE];
      {Methods}
      constructor Init(ALhs, ARhs: string);
    end; {TStrategy}

    PSlot = ^TSlot;
    TSlot = object(TObject)
      {Attributes}
      Name    : String[NAME_SIZE];
      Active  : boolean;
      Prompt  : PPrompt;
      Format  : PFormat;
      Source  : PCollection;
      Context : PCollection;
      Strategy: PCollection;

      {Methods}
      constructor Init(AName: string);
      destructor  Done; virtual;
      procedure   AddPrompt(APrompt: string); virtual;
      procedure   AddFormat(AFormat: string); virtual;
      procedure   AddSource(AnOperator, AnOperand1,
AnOperand2: string);
                     virtual;
      procedure   AddContext(AContext: string); virtual;
      procedure   AddStrategy(ALhs, ARhs: string); virtual;
      function    GetName: string; virtual;
      function    IsActive: boolean; virtual;
      function    SwitchActivity: boolean; virtual;
      function    GetPrompt: string; virtual;
      function    GetFormat: string; virtual;
      procedure   GetSource(Idx: Integer; var AnOperator,
                  AnOperand1, AnOperand2: string); virtual;
      function    GetContext(Idx: Integer): string; virtual;
      procedure   GetStrategy(Idx: Integer; var ALhs, ARhs:
                  string); virtual;
      function    SourceCount: Integer; virtual;
      function    ContextCount: Integer; virtual;
      function    StrategyCount: Integer; virtual;

    end; {TSlot}
```

```
Implementation

Constructor TSlot.Init(AName: string);
{------------------------------------------------------------
Init: The Prompt and Format pointers are set to nil and each
list is initialized.  As well, the Active flag is set to the
default state, false.
------------------------------------------------------------}
begin
  Name   := AName;
  Active := false;
  Prompt := nil;
  Format := nil;
  Source   := new(PCollection, Init(MAX_SRC, SRC_OVRFLOW));
  Context  := new(PCollection, Init(MAX_CON, CON_OVRFLOW));
  Strategy:=new(PCollection,Init(MAX_STRAT, STRAT_OVRFLOW));

end; {TSlot.Init}

Destructor TSlot.Done;
{------------------------------------------------------------
Done: If they exit, the Prompt and Format memory is freed;
also, the lists are destroyed, freeing the allocated memory.
------------------------------------------------------------}
begin
  if (Prompt <> nil) then Dispose(Prompt);
  if (Format <> nil) then Dispose(Format);
  Dispose(Source, Done);
  Dispose(Context, Done);
  Dispose(Strategy, Done);

end; {TSlot.Done}

Procedure TSlot.AddPrompt(APrompt: string);
{------------------------------------------------------------
AddPrompt: Memory is allocated to hold a string of type
TPrompt.  APrompt is stored in this string and Prompt is set
to point at this string.
-----------------------------------------------------------}
var
  p : PPrompt;

begin
  p := new(PPrompt);
  p^:= APrompt;
  Prompt := p;

end; {TSlot.AddPrompt}
```

```
Procedure TSlot.AddFormat(AFormat: String);
{---------------------------------------------------------
AddFormat: Memory is allocated to hold a string of type
TFormat.  AFormat is stored in this string and Format is set
to point at this string.
---------------------------------------------------------}
var
  f : PFormat;

begin
  f := new(PFormat);
  f^:= AFormat;
  Format := f;

end; {TSlot.AddFormat}

Procedure TSlot.AddSource(AnOperator, AnOperand1,
      AnOperand2: string);
{---------------------------------------------------------
AddSource: A new Source element is dynamically created.  The
three parameters passed to this procedure are stored as
strings in this new element, and the new element is added to
the list, Source.
---------------------------------------------------------}
var
  s : PSource;

begin
  { Create a new Source element.}
  s := new(PSource, Init(AnOperator, AnOperand1,
      AnOperand2));

  Source^.Insert(s);  { Insert it in the list. }

end; {TSlot.AddSource}

Procedure TSlot.AddContext(AContext: string);
{---------------------------------------------------------
AddContext: A new Context string is dynamically created, and
AContext is stored in it.  The Context string is then added
to the Context list.
----------------------------------------------------------}
var
  c : PContext;

begin
  c := new(PContext, Init(AContext));    { Create a new
Context element.}
  Context^.Insert(c);                    { Add it to the
list.          }

end; {TSlot.AddContext}

Procedure TSlot.AddStrategy(ALhs, ARhs: string);
{---------------------------------------------------------
AddStrategy: Memory is allocated for a new Strategy element,
ALhs and ARhs are stored in it, and it is added to the
Strategy list.
----------------------------------------------------------}

var
  s : PStrategy;

begin
  s := new(PStrategy, Init(ALhs, ARhs)); { Create a new
                                          Strategy element.}
  Strategy^.Insert(s);  { Insert it in the list. }

end; {TSlot.AddStrategy}

Function TSlot.GetName: String;
{---------------------------------------------------------
GetName: This function returns a pointer to the string
containing the name of the slot.  If there is no string, a
nil pointer will be returned.
---------------------------------------------------------}
begin
  GetName := Name;

end; {TSlot.GetName}

Function TSlot.IsActive: boolean;
{---------------------------------------------------------
IsActive: This function returns a boolean value indicating
whether the Active flag is true or false.
---------------------------------------------------------}
begin
  IsActive := Active;

end; {TSlot.IsActive}

Function TSlot.SwitchActivity: boolean;
{---------------------------------------------------------
SwitchActivity: The boolean Active is given the opposite
value that it currently holds.  The resulting boolean value
is returned by the function.
---------------------------------------------------------}
begin
```

```
  Active := not Active;
  SwitchActivity := Active;

end; {TSlot.SwitchActivity}

Function TSlot.GetPrompt: string;
{-------------------------------------------------------------
GetPrompt: This function returns a string containing the
prompt.  If there is no prompt, then an empty string is
returned.
-------------------------------------------------------------}
begin
  if (Prompt <> nil) then GetPrompt := Prompt^
  else GetPrompt := '';

end; {TSlot.GetPrompt}

Function TSlot.GetFormat: string;
{-------------------------------------------------------------
GetFormat: Similar to GetPrompt, except that the string
returned contains the format string.
-------------------------------------------------------------}
begin
  if (Format <> nil) then GetFormat := Format^
  else GetFormat := '';

end; {TSlot.GetFormat}

Procedure TSlot.GetSource(Idx: Integer; var AnOperator,
      AnOperand1, AnOperand2: string);
{-------------------------------------------------------------
GetSource: A pointer, s, is used to point to the Source
element indexed by Idx. The strings contained in this
element are then assigned to An Operator, AnOperand1, and
AnOperand2.
-------------------------------------------------------------}
var
  s : PSource;

begin
  s := Source^.At(Idx);
  AnOperator := s^.Operator;
  AnOperand1 := s^.Operand1;
  AnOperand2 := s^.Operand2;

end; {TSlot.GetSource}

Function TSlot.GetContext(Idx: Integer): string;
{-------------------------------------------------------------
```

```
GetContext: A string is returned that contains the Context
element indexed by Idx.
---------------------------------------------------------}
var
  c : PContext;

begin
  c := Context^.At(Idx);
  GetContext := c^.Context;

end; {TSlot.GetContext}

Procedure TSlot.GetStrategy(Idx: Integer; var ALhs, ARhs:
      string);
{---------------------------------------------------------
GetStrategy: The parameters, ALhs and ARhs, are assigned the
values of the strings held by the Strategy element indexed
by Idx.
----------------------------------------------------------}
var
  s : PStrategy;

begin
  s := Strategy^.At(Idx);
  ALhs := s^.Lhs;
  ARhs := s^.Rhs;

end; {TSlot.GetStrategy}

Function TSlot.SourceCount: Integer;
{---------------------------------------------------------
SourceCount: The number of Source elements is returned.
---------------------------------------------------------}
begin
  SourceCount := Source^.Count;

end; {TSlot.SourceCount}

Function TSlot.ContextCount: Integer;
{---------------------------------------------------------
ContextCount: The number of Context elements is returned.
{---------------------------------------------------------}
begin
  ContextCount := Context^.Count;

end; {TSlot.ContextCount}

Function TSlot.StrategyCount: Integer;
{---------------------------------------------------------}
```

```
StrategyCount: The number of Strategy elements is returned.
----------------------------------------------------------}
begin
  StrategyCount := Strategy^.Count;

end; {TSlot.StrategyCount}

Constructor TSource.Init(AnOperator, AnOperand1, AnOperand2:
      string);
{---------------------------------------------------------
Init: The Source object is initialized by storing strings
containing AnOperator, AnOperand1, and AnOperand2 in it.
---------------------------------------------------------}
begin
  Operator := AnOperator;
  Operand1 := AnOperand1;
  Operand2 := AnOperand2;

end; {TSource.Init}

Constructor TContext.Init(AContext: string);
{---------------------------------------------------------
Init: The Context object is initialized by storing AContext
in it.
----------------------------------------------------------}
begin
  Context := AContext;

end; {TContext.Init}

Constructor TStrategy.Init(ALhs, ARhs: string);
{---------------------------------------------------------
Init: ALhs and ARhs are stored in the Strategy object.
----------------------------------------------------------}
begin
  Lhs := ALhs;
  Rhs := ARhs;

end; {TContext.Init}

End. {SlotObj}
```

OPERATOR.UNT

```
Unit Operator;
{$V-}     { Turn off string type checking.}

Interface

  Uses WObjects, WinDos, PropObj, ClassObj, ObtObj, NexFile,
ItemObj, Expr, Error, Funcs, Strings;

  Const
    MAX_ITEMS    = 10;
    ITEM_OVRFLOW =  5;

  Type
    TriBool   = (Yes, No, Unknown);
    TFileType = (UnknownFileType, NXPDB);
    TFill     = (NoFill, Add);

    PRetrieve = ^TRetrieve;
    TRetrieve = object(TObject)
      {Attributes}
      FileName: string;
      FileType: TFileType;
      Fill    : TFill;
      ObjtName: string;
      Create  : string;
      Props   : PCollection;
      Fields  : PCollection;
      Cursor  : string;
      {Methods}
      constructor Init(AName, TheRest: string);
      destructor  Done; virtual;
      function    GetFileName: string; virtual;
      function    GetFileType: TFileType; virtual;
      function    GetFill: TFill; virtual;
      function    GetObjtName: string; virtual;
      function    GetCreate: string; virtual;
      function    PropCount: integer; virtual;
      function    GetProp(AnIdx: integer): string; virtual;
    private
      function CalcFileType(AFileType: string): TFileType;
           virtual;
      function CalcFill(AFill: string): TFill; virtual;
      function CalcName(AName: string): string; virtual;
      function CalcProps(AList: string): PCollection;
           virtual;
      function CalcFields(AList: string): PCollection;
           virtual;
    end; {TRetrieve}

  function ProcYes(AnObjtList: PCollection; AVariable:
      string): TriBool;
  function ProcNo(AnObjtList: PCollection; AVariable:
      string): TriBool;
  function ProcIs(AnObjtList: PCollection; AnOperand1,
      AnOperand2: string): TriBool;
  function ProcIsNot(AnObjtList: PCollection; AnOperand1,
      AnOperand2: string): TriBool;
  function ProcEqual(AnObjtList: PCollection; AnOperand1,
      Anoperand2: string): boolean;
  function ProcNotEqual(AnObjtList: PCollection; AnOperand1,
      Anoperand2: string): boolean;
  procedure ProcReset(AnObjtList: PCollection; AnOperand:
      string);
  function ProcName(AnObjtList: PCollection; ASource, ADest:
      string): TriBool;
  procedure ProcCreateObject(AClassList, AnObjtList,
      APropList: PCollection;AnOperand1,AnOperand2: string);
  function  ProcRetrieve(AClassList, AnObjtList, APropList:
      PCollection; AnOperand1, AnOperand2: string): boolean;
  procedure ProcDo(AnObjtList: PCollection; ASource, ADest:
      string);
  procedure ProcShow(AnObjtList: PCollection;
      AFileName,AParamList: string);

Implementation

Constructor TRetrieve.Init(AName, TheRest: string);
{----------------------------------------------------------
Init: This is the constructor for the TRetrieve object.  The
parameter, AName, holds the DOS name of the file to retrieve
from.  However, this name is enclosed by double-quotes (")
and must be unquoted.  The string, TheRest, holds the other
parameters of the Retrieve operator.  These are parsed out,
one at a time and used to set the other attributes of the
TRetrieve object.
----------------------------------------------------------}
var
  i, Max : TLineIndex;
  Temp1, Temp2 : string;

begin
  { Unquote the file name.}
  FileName := UnQuoteString(AName);

  { Parse the parameters from TheRest.}
  i := 1; Max := length(TheRest);
  while (i < Max) do
    begin
      ParseWord(TheRest, '=', Templ, i, Max);
      inc(i);
      ParseWord(TheRest, ';', Temp2, i, Max);
      inc(i);

      { Depending on the contents of Templ, set the
        appropriate attribute.}
      if (Templ = '@TYPE') then
           FileType := CalcFileType(Temp2)
      else if (Templ = '@FILL') then Fill := CalcFill(Temp2)
      else if (Templ = '@NAME') then
           ObjtName := CalcName(Temp2)
      else if (Templ = '@CREATE') then Create := Temp2
      else if (Templ = '@PROPS') then
           Props := CalcProps(Temp2)
      else if (Templ = '@FIELDS') then
           Fields := CalcFields(Temp2)
      { NB: no Interpretations handled by @CURSOR!!!}
      else if (Templ = '@CURSOR') then Cursor := Temp2
      else writeln('Parameters not understood: ',Templ,'
           ',Temp2);
    end; {while}

end; {TRetrieve.Init}

Destructor TRetrieve.Done;
{-----------------------------------------------------------
Done: This is the destructor for the TRetrieve object.  The
two collection attributes were dynamically created;
therefore, their memory needs to be freed.
-----------------------------------------------------------}
begin
  { Deallocate the list memory.}
  dispose(Props, done);
  dispose(Fields, done);

end;

Function TRetrieve.GetFileName: string;
{-----------------------------------------------------------
GetFileName: This function simply returns the FileName
attribute.
------------------------------------------------------------}
begin
  GetFileName := FileName;

end; {TRetrieve.GetFileName}
```

```
Function TRetrieve.GetFileType: TFileType;
{----------------------------------------------------------
GetFileType: This function simply returns the FileType
attribute.
----------------------------------------------------------}
begin
  GetFileType := FileType;

end; {TRetrieve.GetFileType}

Function TRetrieve.GetFill: TFill;
{----------------------------------------------------------
GetFill: The value of the Fill attribute is returned.
-----------------------------------------------------------}
begin
  GetFill := Fill;

end; {TRetrieve.GetFill}

Function TRetrieve.GetObjtName: string;
{----------------------------------------------------------
GetObjtName: The string attribute, ObjtName, is returned by
this function.
-----------------------------------------------------------}
begin
  GetObjtName := ObjtName;

end; {TRetrieve.GetObjtName}

Function TRetrieve.GetCreate: string;
{----------------------------------------------------------
GetCreate: The string attribute, Create, is returned by this
function.
----------------------------------------------------------}
begin
  GetCreate := Create;

end; {TRetrieve.GetCreate}

Function TRetrieve.PropCount: integer;
{----------------------------------------------------------
PropCount: The number of items in the Props collection is
returned.
----------------------------------------------------------}
begin
  PropCount := Props^.Count;

end; {TRetrieve.PropCount}
```

```
Function TRetrieve.GetProp(AnIdx: integer): string;
{---------------------------------------------------------
GetProp: If the index, AnIdx, is within the bounds of the
collection, Props, the value held by the item indexed by
AnIdx is returned.  Otherwise, an empty string is returned.
---------------------------------------------------------}
var
  p : PItem;

begin
  if (AnIdx < Props^.Count) then
    begin
      p := Props^.At(AnIdx);
      GetProp := p^.GetValue;
    end {if}
  else GetProp := '';

end; {TRetrieve.GetProp}

Function TRetrieve.CalcFileType(AFileType: string):
TFileType;
{---------------------------------------------------------
CalcFileType: Based on the contents of the string parameter,
AFileType, a value of the type, TFileType, is returned.
---------------------------------------------------------}
begin
  if (AFileType = 'NXPDB') then CalcFileType := NXPDB
  else CalcFileType := UnknownFileType;

end; {TRetrieve.CalcFileType}

Function TRetrieve.CalcFill(AFill: string): TFill;
{---------------------------------------------------------
CalcFill: A value of the type, TFill, is returned, based on
the contents of the string, AFill.
---------------------------------------------------------}
begin
  if (AFill = 'ADD') then CalcFill := Add
  else CalcFill := NoFill;

end; {TRetrieve.CalcFill}

Function TRetrieve.CalcName(AName: string): string;
{---------------------------------------------------------
CalcName: The quoted-string, AName, is first unquoted, and
then all substrings enclosed by exclamation points (!) are
extracted and concatenated together.  The resulting
composite string is returned.
---------------------------------------------------------}
```

```
var
  Name, Templ, Temp2 : string;
  i, Max : TLineIndex;

begin
  Name := '';
  Templ := UnQuoteString(AName);
  i := 1; Max := length(Templ);
  while (Templ[i] = '!') do
    begin
      inc(i);
      ParseWord(Templ, '!', Temp2, i, Max);
      Name := concat(Name, Temp2);
      inc(i);
    end; {while}

  CalcName := Name;

end; {TRetrieve.CalcName}

Function TRetrieve.CalcProps(AList: string): PCollection;
{---------------------------------------------------------
CalcProps: This function takes a string containing a list of
comma separated PROPERTY names, and extracts the names from
it and stores them as TItem objects in the collection,
PropList.  Temp, i, Max are used in the extraction of the
names.
---------------------------------------------------------}
var
  PropList : PCollection;
  i, Max   : TLineIndex;
  Temp     : string;

begin
  PropList := new(PCollection, Init(MAX_ITEMS,
ITEM_OVRFLOW));
  i := 1; Max := length(AList);
  while (i < Max) do
    begin
      ParseWord(AList, ',', Temp, i, Max);
      if (Temp <> '') then PropList^.Insert(new(PItem,
           Init(Temp)));
      inc(i);
    end; {while}

  CalcProps := PropList;

end; {TRetrieve.CalcProps}
```

```
Function TRetrieve.CalcFields(AList: string): PCollection;
{----------------------------------------------------------
CalcFields: The string, AList, contains a list of comma
separated, quoted (") strings that represent record field
names.  This function parses out the quoted strings, strips
the quotes away, and adds them to the collection, FieldList,
which is returned as the result of the function.  Temp, i,
and Max are used in the extraction of the fields.
-----------------------------------------------------------}
var
  FieldList : PCollection;
  i, Max    : TLineIndex;
  Temp      : string;

begin
  FieldList := new(PCollection, Init(MAX_ITEMS,
ITEM_OVRFLOW));
  i := 1; Max := length(AList);
  while (i < Max) do
    begin
      ParseWord(AList, ',', Temp, i, Max);
      if (Temp <> '') then
        FieldList^.Insert(new(PItem,
           Init(UnQuoteString(Temp))));
      inc(i);
    end; {while}

  CalcFields := FieldList;

end; {TRetrieve.CalcFields}

Function ProcYes(AnObjtList: PCollection; AVariable:
string): TriBool;
{----------------------------------------------------------
ProcYes: The value corresponding to the variable, AVariable,
is determined, and then the value is converted into a return
value of the type, TriBool.
-----------------------------------------------------------}
var
  s : string;

begin
  s := FetchValue(AVariable, AnObjtList);
  if (s = '') then ProcYes := Unknown
  else if (CalcBoolValue(s)) then ProcYes := Yes
  else ProcYes := No;

end; {ProcYes}
```

```
Function ProcNo(AnObjtList: PCollection; AVariable: string):
TriBool;
{---------------------------------------------------------
ProcNo: Since the No operator is the logical negation of the
Yes operator, this function calculates the value of the No
operator by calling ProcYes and returning the opposite
result (Result).  The exception is in the case ProcYes
returns Unknown; if it does, then ProcNo also returns
Unknown.
---------------------------------------------------------}
var
  Result : TriBool;

begin
  { Invoke ProcYes.}
  Result := ProcYes(AnObjtList, AVariable);
  if (Result = Yes) then ProcNo := No
  else if (Result = No) then ProcNo := Yes
  else ProcNo := Unknown;

end; {ProcNo}

Function ProcIs(AnObjtList: PCollection; AnOperand1,
      AnOperand2: string): TriBool;
{---------------------------------------------------------
ProcIs: AnOperand1 must be an OBJECT or OBJECT PROPERTY
specification.  The value of this OBJECT or OBJECT PROPERTY
is retrieved from the list, AnObjtList, and stored in the
variable, Value.  AnOperand2 is either a boolean constant,
one or more string constants, or one of 'KNOWN', 'UNKNOWN',
or 'NOTKNOWN' (UNKNOWN and NOTKNOWN are treated the in the
same manner).  This function returns a TriBool value
indicating whether or not Value matches AnOperand2.  In the
case of 'KNOWN' and UNKNOWN (and NOTKNOWN), the TriBool
returned indicates whether Value contains any value or Value
contains no value, respectively.
---------------------------------------------------------}
var
  s : string;
  r : PResult;
  i, Max: TLineIndex;
  Temp : boolean;

begin
  r := Evaluate(AnOperand1, AnObjtList);
  if (AnOperand2 = 'KNOWN') then
    if (r^.GetValue <> '') then ProcIs := Yes
    else ProcIs := No
  else if ((AnOperand2 = 'UNKNOWN') or (AnOperand2 =
     'NOTKNOWN')) then
    if (r^.GetValue = '') then ProcIs := Yes
    else ProcIs := No
  else
    if (r^.GetValue = '') then ProcIs := Unknown
    else if (AnOperand2[1] <> '"') then
      if (r^.GetValue = AnOperand2) then ProcIs := Yes
      else ProcIs := No
    else
      begin
        Max := length(AnOperand2);
        i := 2;
        Temp := false;
        while ((i < Max-3) and (not Temp)) do
          begin
            ParseWord(AnOperand2, '"', s, i, Max);
            Temp := s = r^.GetValue;
            if (i < Max-3) then i := i+3;
          end; {while}

        if (Temp) then ProcIs := Yes
        else ProcIs := No;
      end; {else}

end; {ProcIs}

Function ProcIsNot(AnObjtList: PCollection; AnOperand1,
      AnOperand2: string): TriBool;
{----------------------------------------------------------
ProcIsNot: This function returns a boolean value which is
simply the logical negation of ProcIs, with the same
parameters.  If ProcIs returns Unknown, then ProcIsNot is
also Unknown.
----------------------------------------------------------}
var
  Result : TriBool;

begin
  Result := ProcIs(AnObjtList,AnOperand1,AnOperand2);
  if (Result = Yes) then ProcIsNot := No
  else if (Result = No) then ProcIsNot := Yes
  else ProcIsNot := Unknown;

end; {ProcIsNot}

Function ProcEqual(AnObjtList: PCollection; AnOperand1,
      Anoperand2: string): boolean;
{----------------------------------------------------------
ProcEqual: If the value of the attributes indicated by
```

```
AnOperand1 & AnOperand2 are known and equivalent, then this
function returns true; otherwise a false value is returned.
----------------------------------------------------------}
var
  v1, v2 : string;

begin
  v1 := FetchValue(AnOperand1, AnObjtList);
  { Is the value of AnOperand1 known?}
  if (v1 <> '') then
    begin
      v2 := FetchValue(AnOperand2, AnObjtList);     {Known}
      ProcEqual := v1 = v2;
    end {if}
  else
    ProcEqual := false;                            {Unknown}

end; {ProcEqual}

Function ProcNotEqual(AnObjtList: PCollection; AnOperand1,
      Anoperand2: string): boolean;
{---------------------------------------------------------
ProcNotEqual: The NotEqual operator is the counter-part of
the Equal operator.  As such, it is calculated by calling
ProcEqual and logically negating the boolean result.  This
negated result is returned by the function.
---------------------------------------------------------}
begin
  ProcNotEqual :=  not ProcEqual(AnObjtList, AnOperand1,
                     AnOperand2);

end; {ProcNotEqual}

Procedure ProcReset(AnObjtList: PCollection; AnOperand:
      string);
{---------------------------------------------------------
ProcReset: Invoking the Reset operator causes the attribute
represented by AnOperand to be reset to the unknown state
('').
----------------------------------------------------------}
begin
  AssignValue(AnObjtList, AnOperand, '');

end; {ProcReset}

Function ProcName(AnObjtList: PCollection; ASource, ADest:
      string): TriBool;
{---------------------------------------------------------
ProcName: Using the Name operator causes the value of the
```

```
NEXPERT OBJECT expression, ASource, to be assigned to the
attribute with the name ADest.  Of course, the two must be
type compatible.  AnObjtList is needed for finding the
destination attribute.  If ASource hasn't been defined yet,
the operation is not performed, and the function returns
Unknown.  Otherwise, the function returns Yes.
-------------------------------------------------------------}
var
  r : PResult;
  t : TPropType;

begin
  { Evaluate the source.}
  r := Evaluate(ASource, AnObjtList);

  if (r^.GetValue <> '') then
    begin
      { Retrieve the type of the destination.}
      t := FetchType(ADest, AnObjtList);

      { If the type is Integer, round off the Float result.}
      if (t = PropInt) then
        { Check if the result is indeed a Float.}
        if ((r^.GetType = PropFloat) or (r^.GetType =
           PropInt)) then begin
            str(CalcIntValue(r^.GetValue),ASource);
            AssignValue(AnObjtList, ASource, ADest)
          end {if}
        else TypeClash(ASource, ADest)

      { For all other types, check for type compatibility.}
      else if (t = r^.GetType) then
        AssignValue(AnObjtList, r^.GetValue, ADest)
      else TypeClash(ASource, ADest);
      ProcName := Yes;
    end {if}
  else ProcName := Unknown;

end; {ProcName}

Procedure ProcessClass(Line: TLine; AClassList, APropList:
      PCollection; var Objt: PObjt);
{ ------------------------------------------------------------
ProcessClass: The parameter, Line, contains a CLASS name.
The list, AClassList, is searched for a CLASS having this
name.  If such a CLASS is found, all PROPERTIES of this
CLASS are added to the list of PROPERTIES associated with
the current OBJECT, Objt.
 -------------------------------------------------------------}
```

```
var
  PropName: string;
  i, j: integer;
  Cls: PClass;

function ClassMatch(c: PClass): boolean; far;
  begin
    ClassMatch := c^.GetName = Line;
  end; {ClassMatch}

function PropMatch(p: PProp): boolean; far;
  begin
    PropMatch := p^.GetName = PropName;
  end; {PropMatch}

begin
  Cls := AClassList^.FirstThat(@ClassMatch);
  if (Cls <> nil) then
    begin
      j := Cls^.PropCount;
      i := 0;
      while (i < j) do
        begin
          PropName := Cls^.GetProp(i);
          if (PropName <> '') then
            Objt^.AddProp(APropList^.FirstThat(@PropMatch));
          inc(i);
        end; {while}
    end; {if}

end; {ProcessClass}

Procedure ProcCreateObject(AClassList, AnObjtList,
  APropList: PCollection; AnOperand1, AnOperand2: string);
{-------------------------------------------------------------
ProcCreateObject: This procedure performs the duties of the
CreateObject operand.  The operand takes one or two operands
(AnOperand1, AnOperand2).  If only one operand is used then
AnOperand2 is an empty string.  The first operand is the
name of an OBJECT to be created or a CLASS to be made a
subCLASS.  The second operand, if there is one, is a list of
CLASS and/or OBJECT names, for which the first operand is
made an instance or, subOBJECT of, or is made a subCLASS of.
CList and OList are used to hold the lists of CLASSES
(CList) and OBJECTS (OList) held by the second operand.
ClassName and ObjtName are used to hold the name indicated
by AnOperand1, depending on whether it is a CLASS or OBJECT
name.  (NB: both operands may contain patterns or interpret-
ations.  Note: as is, this procedure does not handle
```

```
subOBJECTS or patterns.
---------------------------------------------------------------}
var
  ClassName, ObjtName : string;
  Objt : PObjt;
  AClass : PItem;
  i, Count : integer;
  CList, OList : PCollection;

begin
  ObjtName := '';
  { Process the first operand.}
  if ((AnOperand1[1] = '\') or (AnOperand1[1] = #39)) then
    ObjtName := Interpretation(AnObjtList, AnOperand1)
  else if (AnOperand1[1] = '¦') then
      ClassName := ExtractClass(AnOperand1)
  else ObjtName := AnOperand1;

  { If the first operand is an OBJECT, create an OBJECT with
     this name.}
  if (ObjtName<>'') then Objt := new(PObjt, Init(ObjtName));

  if (AnOperand2 <> '') then
    begin
      { Process the second operand.}
      ExtractCreateLists(CList, OList, AnOperand2);

      i := 0;
      Count := CList^.Count;
      { Add each CLASS in the list to the new OBJECT.}
      while (i < Count) do
        begin
          AClass := CList^.At(i);
          ProcessClass(AClass^.GetValue, AClassList,
            APropList, Objt);
          inc(i);
        end; {while}
    end; {if}

  InsertObjt(AnObjtList, Objt);

end; {ProcCreateObject}

Function GetAttributes(AList: string): PCollection;
{------------------------------------------------------------
GetAttributes: This function extracts all the '¦' delimited
strings from the string AList.  Each substring is added to a
collection of strings, Attr.  When the end of AList is
reached, the collection is returned.
```

```
---------------------------------------------------------}
var
  Attr : PCollection;
  i, Max : TLineIndex;
  Temp : string;

begin
  { Initialize the new collection.}
  Attr := new(PCollection, Init(MAX_ITEMS, ITEM_OVRFLOW));
  i := 1; Max := length(AList);
  { Parse the string for '¦' delimited substrings.}
  while (i < Max) do
    begin
      i := NextWord(AList, i, Max);
      ParseWord(AList, '¦', Temp, i, Max);
      inc(i);
      Attr^.Insert(new(PItem, Init(Temp)));
    end; {while}

  { Return the new collection.}
  GetAttributes := Attr;

end; {GetAttributes}

Procedure AddAttributes(ARetrieve: PRetrieve; AnAttrList:
      PCollection; AnObjt: PObjt);
{---------------------------------------------------------
AddAttributes: The list of values held by AnAttrList are
assigned to the values held by the PROPERTIES of the OBJECT,
AnObjt.  If there are more values than PROPERTIES to hold
them, then an error message is displayed, and no assignment
takes place.
---------------------------------------------------------}
var
  i, Idx, Count : integer;
  Prop : string;
  p : PProp;
  a : PItem;

begin
  Idx := 0; Count := AnAttrList^.Count;

  { Check for too many values in the list.}
  if (Count > AnObjt^.GetPropCount) then
    begin
      writeln('Too many attributes to add to
           ',AnObjt^.GetName,'.');
      halt(1);
    end {if}
  else
    { Assign the values to the corresponding PROPERTIES.}
    while (Idx < Count) do
      begin
        p := AnObjt^.GetProp(Idx);
        i := 0; a := nil;
        while (i < ARetrieve^.PropCount) do
          begin
            Prop := ARetrieve^.GetProp(i);
            if (Prop = p^.GetName) then
              begin
                a := AnAttrList^.At(i);
                i := ARetrieve^.PropCount;
              end; {if}
            inc(i);
          end; {while}
        if (a <> nil) then p^.SetValue(a^.GetValue);
        inc(Idx);
      end; {while}

end; {AddAttributes}

Function GetNameIndex(AFieldList: PCollection; AName:
      string): integer;
{----------------------------------------------------------
GetNameIndex: This function searches the collection of
TItems, AFieldList, for one that matches the string, AName.
If a match is found (Found = TRUE), an integer index to the
matching item in the collection is returned.  Otherwise, a
value of -1 is returned.
----------------------------------------------------------}
var
  p : PItem;
  Idx, Count : integer;
  Found : boolean;

begin
  Idx := 0; Count := AFieldList^.Count;
  while ((Idx < Count) and (not Found)) do
    begin
      p := AFieldList^.At(Idx);
      Found := p^.GetValue = AName;
      if (not Found) then inc(Idx);
    end; {while}

  if (Found) then GetNameIndex := Idx
  else GetNameIndex := -1;

end; {GetNameIndex}
```

```
Function FileFound(AFileName: string): boolean;
{-------------------------------------------------------------
FileFound: A search of all directories in the current path
is made for the file AFileName.  If it is found, this
function returns TRUE; otherwise, the result is false.
-------------------------------------------------------------}
var
  s: array[0..fsPathName] of char;
  FileName: array[0..12] of char;

begin
  StrPCopy(FileName,AFileName);
  FileSearch(s, FileName, GetEnvVar('PATH'));
  FileFound := s[0] <> #0;
end; {FileFound}

Function ProcRetrieve(AClassList, AnObjtList, APropList:
     PCollection; AnOperand1, AnOperand2: string): boolean;
{-------------------------------------------------------------
ProcRetrieve: This function performs the operation of the
NEXPERT OBJECT Retrieve operator.  AnOperand1 holds the name
of the file to be read from, and AnOperand2 contains a list
of parameters for the retrieve.  A TRetrieve object is
created and is used to hold info indicating how the file
data is to be interpreted.  Each line (Line) of the file is
then read and the information contained therein is parsed
out and added to a list (Attr) of such data.  If new OBJECTS
are to be made (@FILL=ADD), then ProcCreateObjt is invoked
and the data in Attr is stored in the PROPERTIES of the new
OBJECT.
-------------------------------------------------------------}
var
  r : PRetrieve;
  a : PItem;
  InFile : text;
  Line : string;
  Attr : PCollection;
  i, Count, NameIdx : integer;

begin
  { Create a new TRetrieve object.}
  r := new(PRetrieve, Init(AnOperand1, AnOperand2));
  NameIdx := GetNameIndex(r^.Fields, r^.GetObjtName);
  if (FileFound(r^.GetFileName)) then
  begin
    OpenFiles(InFile, r^.GetFileName);
    { Skip the first two lines of a NXPDB file.}
    if (r^.GetFileType = NXPDB) then
      begin
```

```
        if (not eof(InFile)) then readln(InFile);
        if (not eof(InFile)) then readln(InFile);

        { Process the rest of the file.}
        while (not eof(InFile)) do
          begin
            readln(InFile, Line);
            if (Line[1] <> '*') then
              begin
                Attr := GetAttributes(Line);
                if (r^.GetFill = Add) then
                begin
                  a := Attr^.At(NameIdx);
                  ProcCreateObject(AClassList, AnObjtList,
                     APropList,a^.GetValue, r^.GetCreate);
                  AddAttributes(r, Attr, FindObjt(
                     AnObjtList, a^.GetValue));
                end; {if}
                dispose(Attr, done);
              end; {if}
          end; {while}
        ProcRetrieve := true;
      end {if}
    else
      ProcRetrieve := false;

    CloseFiles(InFile);
  end {if}
  else
    ProcRetrieve := false;

  dispose(r, done);
end; {ProcRetrieve}

Procedure ProcDo(AnObjtList: PCollection; ASource, ADest:
      string);
{-----------------------------------------------------------
ProcDo: The expression held by the parameter, ASource, is
evaluated and the result is assigned to the OBJECT indicated
by the parameter ADest.
------------------------------------------------------------}
var
  r : PResult;
  t : TPropType;


begin
  { Evaluate the source.}
  r := Evaluate(ASource, AnObjtList);
  { Retrieve the type of the destination.}
  t := FetchType(ADest, AnObjtList);

  { If the type is Integer, round off the Float result.}
  if (t = PropInt) then
    { Check if the result is indeed a Float.}
    if ((r^.GetType = PropFloat) or (r^.GetType = PropInt))
      then begin
        str(CalcIntValue(r^.GetValue),ASource);
        AssignValue(AnObjtList, ASource, ADest)
      end {if}
    else TypeClash(ASource, ADest)

  { For all other types, check for type compatibility.}
  else if (t = r^.GetType) then AssignValue(AnObjtList,
      r^.GetValue, ADest)
  else TypeClash(ASource, ADest);

end; {ProcDo}

Procedure ProcShow(AnObjtList: PCollection; AFileName,
      AParamList: string);
{-----------------------------------------------------------
ProcShow: Display the contents of the file AFileName.
------------------------------------------------------------}
var
  r : PResult;
  AFile : text;
  Line : string;

begin
  r := Evaluate(AFileName,AnObjtList);
  if (r^.GetValue <> '') then
    if (FileFound(r^.GetValue)) then
    begin
      OpenFiles(AFile, r^.GetValue);
      while (not eof(AFile)) do readln(AFile, Line);
      writeln(Line);
      CloseFiles(AFile)
    end; {if}

  dispose(r,done);

end; {ProcShow}

End. {Operator}
```

ERROR.UNT

```
Unit Error;

Interface

  Uses WinCrt;

  procedure TypeClash(ASource, ADest: string);

Implementation

Procedure TypeClash(ASource, ADest: string);
{----------------------------------------------------------
TypeClash: Report a type mismatch into a standard out
window.
-----------------------------------------------------------}
begin
  writeln;
  writeln('TYPE MISMATCH!');
  writeln('The error occurred when ', ASource,' was assigned
      to ', ADest);
  writeln('Consequently, the assignment has been ignored.');
  writeln;

end; {TypeClash}

End. {Error}
```

STACK.UNT

```
Unit Stack;

Interface

  Uses WObjects;

  Type
    PStackItem = ^TStackItem;
    TStackItem = record
      Item   : Pointer;
      Next   : PStackItem;
    end; {TStackItem}

    PStack = ^TStack;
    TStack = object(TObject)
      {Attributes}
      TOS : PStackItem;
      {Methods}
      constructor Init;
      function    Top: Pointer; virtual;
      function    Pop: Pointer; virtual;
      procedure   Push(AnItem: Pointer); virtual;
      function    IsEmpty: boolean; virtual;
    end; {TStack}

Implementation

constructor TStack.Init;
{----------------------------------------------------------
Init: The conclusion stack is set to the empty state by
assigning TOS the nil pointer.
----------------------------------------------------------}
begin
  TOS := nil;

end; {TStack.Init}

function TStack.Top: Pointer;
{----------------------------------------------------------
Top: A pointer to the item stored in the top element of the
stack is returned, but the top element is left intact and
the stack is not altered in any way.
-----------------------------------------------------------}
begin
  Top := TOS^.Item;

end; {TStack.Top}
```

```
function TStack.Pop: Pointer;
{-----------------------------------------------------------
Pop: A pointer (p) to the item stored in the first element
of the stack (TOS) is returned.  The top element is also
deallocated (via q), and the top of stack pointer (TOS) is
move downward by one element.
-----------------------------------------------------------}
var
  p : Pointer;
  q : PStackItem;

begin
  { Fetch the stack values.}
  p := TOS^.Item;

  { Dispose of the top item and update the stack.}
  q := TOS;
  TOS := TOS^.Next;
  dispose(q);
  Pop := p;

end; {TStack.Pop}

procedure TStack.Push(AnItem: Pointer);
{-----------------------------------------------------------
Push: A new stack item is dynamically created and AnItem is
stored in it.  The new item (p) is then inserted into the
linked list representing the stack, and the TOS pointer is
updated.
-----------------------------------------------------------}
var
  p : PStackItem;

begin
  { Create a new stack element and insert the appropriate
     values.}
  new(p);
  p^.Item := AnItem;

  { Insert p and update the Top Of Stack pointer.}
  p^.next := TOS;
  TOS := p;

end; {TStack.Push}
```

```
function TStack.IsEmpty: boolean;
{-------------------------------------------------------------
IsEmpty: If the conclusion stack is empty, then this
function returns true; otherwise, false is returned.
-------------------------------------------------------------}
begin
  IsEmpty := TOS = nil;

end; {TStack.IsEmpty}

End. {ConStack}
```

RULELIST.UNT

```
Unit RuleList;
{$V-} { Turn off type checking for strings.}

Interface

  Uses WinCrt, WObjects, RuleObj, NexFile;

  Const
    MAX_RULE     = 25;
    RULE_OVRFLOW =  5;
    RULE_EXT = '.rul';

  Type
    PRuleList = ^TRuleList;
    TRuleList = object(TCollection)
      constructor Init(AMax, AnOvrFlow: integer);
    private
     procedure ProcessIC(AnIC: TLine; var ARule: PRule);
       virtual;
     procedure ProcessLhs(ALine: TLine; var RuleFile: text;
                          var ARule: PRule); virtual;
     procedure ProcessRhs(ALine: TLine; var RuleFile: text;
                          var ARule: PRule); virtual;
     procedure FetchRules; virtual;
    end; {TRuleList}

Implementation

Constructor TRuleList.Init(AMax, AnOvrFlow: integer);
{ --------------------------------------------------------
Init: Construct TRuleList as a collection; then retrieve the
rules.
----------------------------------------------------------}
  begin
    TCollection.Init(AMax, AnOvrFlow);
    FetchRules;

  end; {TRuleList.Init}

Procedure TRuleList.ProcessIC(AnIC: TLine; var ARule:
     PRule);
{ --------------------------------------------------------
ProcessIC: The string AnIC is converted to a real value.  If
the conversion is ok, the IC of ARule is assigned the
integer equivalent of the real.
----------------------------------------------------------}
var
```

```
  Err : integer;
  Result : real;

begin
  val(AnIC, Result, Err);
  if (Err = 0) then ARule^.SetIC(Trunc(Result));

end; {TRuleList.ProcessIC}

Procedure TRuleList.ProcessLhs(ALine: TLine; var RuleFile:
      text; var ARule: PRule);
{ ----------------------------------------------------------
ProcessLhs: The Operator line (ALine) of a Lhs expression
has been encountered and this procedure fetches the next two
lines (Operand1, Operand2) which contain the first and
second operands of the expression, from the input file,
RuleFile.  Then, the Lhs expression is added to the list of
such expressions in the RULE, ARule.
-----------------------------------------------------------}
var
  Operator, Operand1, Operand2 : TLine;

begin
  Operator := ALine;
  readln(RuleFile, Operand1);
  readln(RuleFile, Operand2);
  ARule^.AddLhs(Operator, Operand1, Operand2);

end; {TRuleList.ProcessLhs}

Procedure TRuleList.ProcessRhs(ALine: TLine; var RuleFile:
     text; var ARule: PRule);
{ ----------------------------------------------------------
ProcessRhs: The Operator line (ALine) of a Rhs expression
has been encountered and this procedure fetches the next two
lines (Operand1, Operand2) which contain the first and
second operands of the expression, from the input file,
RuleFile.  Then, the Rhs expression is added to the list of
such expressions in the RULE, ARule.
-----------------------------------------------------------}
var
  Operator, Operand1, Operand2 : TLine;

begin
  Operator := ALine;
  readln(RuleFile, Operand1);
  readln(RuleFile, Operand2);
  ARule^.AddRhs(Operator, Operand1, Operand2);
end; {TRuleList.ProcessRhs}
```

```
Procedure TRuleList.FetchRules;
{ -----------------------------------------------------------
FetchRules: The file, RuleFile, is read and the RULE data is
extracted from it and stored in RULE objects (r), which are
kept in the collection of rules.  The integers, i and j, are
used to parse the lines of the file; LineType is used to
store the line type, as indicated by the first two
characters on the input line.  Also, Line is used to hold
each line of the input file, while it is being analysed.
 -----------------------------------------------------------}
var
  RuleFile : text;
  r : PRule;
  i,j : TLineIndex;
  LineType : TLineType;
  Line, RuleLine : TLine;

begin
  OpenFiles(RuleFile, concat(NEX_FILE, RULE_EXT));

  { Discard the first Line.}
  readln(RuleFile);
  while (not eof(RuleFile)) do
    begin
      { Fetch the name of the class.}
      readln(RuleFile, Line);
      i := 1; j := length(Line);
      ParseWord(Line,' ',RuleLine,i,j);

      { Allocate a new class.}
      r := new(PRule, Init(RuleLine));

      while ((not eof(RuleFile)) and (length(Line) > 0)) do
        begin
        {Fetch the Line-type of the next Line in the file.}
          readln(RuleFile, Line);
          i := 1; j := length(Line);
          ParseWord(Line,' ',LineType,i,j);
          RuleLine := ProcessComponent(Line,i,j);

          { Act according to the LineType.}
          if (LineType = 'IC') then ProcessIC(RuleLine, r)
          else if (LineType = 'L1') then
            ProcessLhs(RuleLine, RuleFile, r)
          else if (LineType = 'R1') then
            ProcessLhs(RuleLine, RuleFile, r)
          else if (LineType = 'HY') then
            r^.SetHypo(RuleLine);
        end; {while}
      Insert(r);
    end; {while}

  CloseFiles(RuleFile);

end; {TRuleList.FetchRules}

End. {RuleList}
```

RULEOBJ.UNT

```
Unit RuleObj;

Interface

  Uses WinTypes, WinProcs, WObjects;

  Const
    MAX_EXPRS    = 10;
    EXPR_OVRFLOW =  5;

    RNAME_SIZE   =   5;
    HYPO_SIZE    = 100;
    OPERAND_SIZE = 255;

  Type
    TOperator = (_Do, _Let, _CreateObject, _DeleteObject,
      _Retrieve, _Write, _Reset, _Strategy, _Show, _Execute,
      _LoadKB, _UnloadKB, _NoInherit, _InhMethod,
      _Interrupt, _InitValue, _RunTimeValue, _InhValueDown,
      _AskQuestion, _Backward, _InhValueUp, _Is, _IsNot,
      _Name, _No, _Yes);

    POperand = ^TOperand;
    TOperand = string[OPERAND_SIZE];

    PExpr = ^TExpr;
    TExpr = object(TObject)
      Operator : TOperator;
      Operand1 : TOperand;
      Operand2 : POperand;
      {Methods}
      constructor Init(AnOperator: string);
      destructor  Done; virtual;
      procedure   SetOperand1(AnOperand: TOperand); virtual;
      procedure   SetOperand2(AnOperand: TOperand); virtual;
      function    GetOperator: TOperator; virtual;
      function    GetOperand1: TOperand; virtual;
      function    GetOperand2: TOperand; virtual;
    end; {TExpr}

    PRule = ^TRule;
    TRule = object(TObject)
      {Attributes}
      Name : string[RNAME_SIZE];
      IC   : integer;
      Hypo : string[HYPO_SIZE];
      Lhs  : PCollection;
      Rhs   : PCollection;
      {Methods}
      constructor Init(AName: string);
      destructor  Done; virtual;
      procedure   SetIC(AnIC: integer); virtual;
      procedure   SetHypo(AHypo: string); virtual;
      procedure   AddLhs(AnOperator, AnOperand1, AnOperand2:
                    string); virtual;
      procedure   AddRhs(AnOperator, AnOperand1, AnOperand2:
                    string); virtual;
      function    GetName: string; virtual;
      function    GetIC: integer; virtual;
      function    GetHypo: string; virtual;
      function    GetLhs(Idx: integer): PExpr; virtual;
      function    GetRhs(Idx: integer): PExpr; virtual;
      function    LhsCount: integer; virtual;
      function    RhsCount: integer; virtual;
    end; {TRule}

  function CalcOperator(AnOperator: string): TOperator;

Implementation

Constructor TRule.Init(AName: string);
{---------------------------------------------------------
Init: The Name field is assigned the value, AName, IC is set
to 1 (default), and each of the lists is initialized.
----------------------------------------------------------}
begin
  Name := AName;
  IC := 1;
  Lhs := new(PCollection, Init(MAX_EXPRS, EXPR_OVRFLOW));
  Rhs := new(PCollection, Init(MAX_EXPRS, EXPR_OVRFLOW));

end; {TRule.Init}

Destructor TRule.Done;
{---------------------------------------------------------
Done: The Lhs and Rhs list are disposed of, freeing the
memory allocated to them.
---------------------------------------------------------}
begin
  dispose(Lhs, done);
  dispose(Rhs, done);

end; {TRule.Done}
```

```
Procedure TRule.SetIC(AnIC: integer);
{-----------------------------------------------------------
SetIC: The attribute, IC, is assigned the value of AnIC.
-----------------------------------------------------------}
begin
  IC := AnIC;

end; {TRule.SetIC}

Procedure TRule.SetHypo(AHypo: string);
{-----------------------------------------------------------
SetHypo: The attribute, Hypo, is assigned the value of
AHypo.
-----------------------------------------------------------}
begin
  Hypo := AHypo;

end; {TRule.SetHypo}

Procedure TRule.AddLhs(AnOperator, AnOperand1, AnOperand2:
      string);
{-----------------------------------------------------------
AddLhs: A new TExpr object (e) is dynamically created, with
AnOperator as an operator, AnOperand1 as operand1, and, if
AnOperand2 is not an empty string, AnOperand2 as operand2.
This new object is then inserted into the list, Lhs.
-----------------------------------------------------------}
var
  e : PExpr;

begin
  e := new(PExpr, Init(AnOperator));
  e^.SetOperand1(AnOperand1);
  if (AnOperand2 <> '') then e^.SetOperand2(AnOperand2);
  Lhs^.Insert(e);

end; {TRule.AddLhs}

Procedure TRule.AddRhs(AnOperator, AnOperand1, AnOperand2:
      string);
{-----------------------------------------------------------
AddRhs: A new TExpr object (e) is dynamically created, with
AnOperator as an operator, AnOperand1 as operand1, and, if
AnOperand2 is not an empty string, AnOperand2 as operand2.
This new object is then inserted into the list, Rhs.
-----------------------------------------------------------}
var
  e : PExpr;
```

```
begin
  e := new(PExpr, Init(AnOperator));
  e^.SetOperand1(AnOperand1);
  if (AnOperand2 <> '') then e^.SetOperand2(AnOperand2);
  Rhs^.Insert(e);

end; {TRule.AddRhs}

Function TRule.GetName: string;
{----------------------------------------------------------
GetName: The string value of Name is returned.
----------------------------------------------------------}
begin
  GetName := Name;

end; {TRule.GetName}

Function TRule.GetIC: integer;
{----------------------------------------------------------
GetIC: The integer value of IC is returned.
----------------------------------------------------------}
begin
  GetIC := IC;

end; {TRule.GetIC}

Function TRule.GetHypo: string;
{----------------------------------------------------------
GetHypo: The string value of Hypo is returned.
----------------------------------------------------------}
begin
  GetHypo := Hypo;

end; {TRule.GetHypo}

Function TRule.GetLhs(Idx: integer): PExpr;
{----------------------------------------------------------
GetLhs: As long as the index, Idx, is within range, then a
pointer to the expression (PExpr) representing the indexed
element of the Lhs list, is returned.  Otherwise, a nil
pointer is returned.
----------------------------------------------------------}
begin
  if ((Idx >= 0) and (Idx < Lhs^.Count)) then
      GetLhs := Lhs^.At(Idx)
  else GetLhs := nil;

end; {TRule.GetLhs}
```

```
Function TRule.GetRhs(Idx: integer): PExpr;
{-----------------------------------------------------------
GetRhs: As long as the index, Idx, is within range, then a
pointer to the expression (PExpr) representing the indexed
element of the Rhs list, is returned.  Otherwise, a nil
pointer is returned.
-----------------------------------------------------------}
begin
  if ((Idx >= 0) and (Idx < Rhs^.Count)) then GetRhs :=
Rhs^.At(Idx)
  else GetRhs := nil;

end; {TRule.GetRhs}

Function TRule.LhsCount: integer;
{-----------------------------------------------------------
LhsCount: This function returns the number of items in the
Lhs list.
-----------------------------------------------------------}
begin
  LhsCount := Lhs^.Count;

end; {TRule.LhsCount}

Function TRule.RhsCount: integer;
{-----------------------------------------------------------
RhsCount: This function returns the number of items in the
Rhs list.
-----------------------------------------------------------}
begin
  RhsCount := Rhs^.Count;

end; {TRule.RhsCount}

Constructor TExpr.Init(AnOperator: string);
{-----------------------------------------------------------
Init: The Operator attribute is assigned a value based on
the contents of AnOperator; Operand1 is made an empty string
and Operand2 is set to nil, since it is optional.
-----------------------------------------------------------}
begin
  Operator := CalcOperator(AnOperator);
  Operand1 := '';
  Operand2 := nil;

end; {TExpr.Init}
```

```
Destructor TExpr.Done;
{-----------------------------------------------------------
Done: If it has been allocated memory, the Operand2
attribute is disposed, freeing the memory.
-----------------------------------------------------------}
begin
  if (Operand2 <> nil) then dispose(Operand2);

end; {TExpr.Done}

Procedure TExpr.SetOperand1(AnOperand: TOperand);
{-----------------------------------------------------------
SetOperand1: The Operand1 attribute is assigned the value of
the string, AnOperand.
-----------------------------------------------------------}
begin
  Operand1 := AnOperand;

end; {TExpr.SetOperand1}

Procedure TExpr.SetOperand2(AnOperand: TOperand);
{-----------------------------------------------------------
SetOperand2: The Operand2 attribute is dynamically allocated
and assigned the value of the string, AnOperand.
-----------------------------------------------------------}
begin
  new(Operand2);
  Operand2^ := AnOperand;

end; {TExpr.SetOperand2}

Function TExpr.GetOperator: TOperator;
{-----------------------------------------------------------
GetOperator: The value of the Operator attribute is
returned.
-----------------------------------------------------------}
begin
  GetOperator := Operator;

end; {TExpr.GetOperator}

Function TExpr.GetOperand1: TOperand;
{-----------------------------------------------------------
GetOperand1: The value of the Operand1 attribute is
returned.
-----------------------------------------------------------}
begin
  GetOperand1 := Operand1;
end; {TExpr.GetOperand1}
```

```
Function TExpr.GetOperand2: TOperand;
{---------------------------------------------------------
GetOperand2: The value of the Operand2 attribute is
returned.
---------------------------------------------------------}
begin
  if (Operand2 <> nil) then GetOperand2 := Operand2^
  else GetOperand2 := '';

end; {TExpr.GetOperand2}

Function CalcOperator(AnOperator: string): TOperator;
{---------------------------------------------------------
CalcOperator: The string AnOperator is analysed and a
corresponding value of the type, TOperator, is returned.
---------------------------------------------------------}
begin
  if (AnOperator = 'Do') then CalcOperator := _Do
  else if (AnOperator = 'Let') then CalcOperator := _Let
  else if (AnOperator = 'CreateObject') then
     CalcOperator := _CreateObject
  else if (AnOperator = 'DeleteObject') then
     CalcOperator := _DeleteObject
  else if (AnOperator = 'Retrieve') then
     CalcOperator := _Retrieve
  else if (AnOperator = 'Write') then CalcOperator := _Write
  else if (AnOperator = 'Reset') then CalcOperator := _Reset
  else if (AnOperator = 'Strategy') then
     CalcOperator := _Strategy
  else if (AnOperator = 'Show') then CalcOperator := _Show
  else if (AnOperator = 'Execute') then
     CalcOperator := _Execute
  else if (AnOperator = 'LoadKB') then
     CalcOperator := _LoadKB
  else if (AnOperator = 'UnloadKB') then
     CalcOperator := _UnloadKB
  else if (AnOperator = 'NoInherit') then
     CalcOperator := _NoInherit
  else if (AnOperator = 'InhMethod') then
     CalcOperator := _InhMethod
  else if (AnOperator = 'Interrupt') then
     CalcOperator := _Interrupt
  else if (AnOperator = 'InitValue') then
     CalcOperator := _InitValue
  else if (AnOperator = 'RunTimeValue') then
     CalcOperator := _RunTimeValue
  else if (AnOperator = 'InhValueDown') then
     CalcOperator := _InhValueDown
  else if (AnOperator = 'AskQuestion') then
```

```
     CalcOperator := _AskQuestion
  else if (AnOperator = 'Backward') then
     CalcOperator := _Backward
  else if (AnOperator = 'Is') then CalcOperator := _Is
  else if (AnOperator = 'IsNot') then CalcOperator := _IsNot
  else if (AnOperator = 'Name') then CalcOperator := _Name
  else if (AnOperator = 'Yes') then CalcOperator := _Yes
  else if (AnOperator = 'No') then CalcOperator := _No
  else CalcOperator := _InhValueUp;

end; {CalcOperator}

End. {RuleObj}
```

CONSTACK.UNT

```
Unit ConStack;

Interface

  Uses WObjects, Stack;

  Type
    PConcl = ^TConcl;
    TConcl = record
      Rule  : integer;
      Clause: integer;
    end; {TConcl}

    PConStack = ^TConStack;
    TConStack = object(TStack)
      constructor Init
    private
      function FindHypo(ARuleList: PCollection; Idx:
           integer; AHypo: string):integer; virtual;
    end; {TConStack}

Implementation

Function FindHypo(ARuleList: PCollection; Idx: integer;
      AHypo: string):integer;
{--------------------------------------------------------
FindHypo: The list of RULES, ARuleList, is searched,
starting at the index Idx and proceding forward (+ve)
through the list.  The search halts when a match (as
indicated by Match) is made between a RULE (r) hypothesis
and AHypo, or when the end of the list is reached.  If a
match is made, this function returns the index (Idx) to the
matching RULE.  Otherwise, a value of -1 is returned.
--------------------------------------------------------}
var
  Match : boolean;
  Count : integer;
  r     : PRule;

begin
  Match := false;
  Count := ARuleList^.Count;
  { Search for a match.}
  while ((Idx < Count) and (not Match)) do
    begin
      r := ARuleList^.At(Idx);
      Match := AHypo = r^.GetHypo;
      inc(Idx);
    end; {while}

{ If a match is found, return the index to the matching
rule.}
  if (Match) then FindHypo := Idx-1
  { Otherwise, return -1.}
  else FindHypo := -1;

end; {FindHypo}

End. {ConStack}
```