COSC 460
Honours Project Report
Department of Computer Science
University of Canterbury

An Expert System

for

Loan Decisions

By Lawrence Ang

1987

Supervisor : Dr. Wolfgang Kreutzer

Acknowledgements

I wish to thank the following persons : Dr Wolfgang Kreutzer, my supervisor, for his advice on various aspects of this project; Mr Robin Whitmore, of United, for creating this project and agreeing to provide the PC for development; Dr Graeme Black, of United, for providing technical assistance and liason; Mr Norman Farquhar, of United, for his patience in explaining the problems about mortgages and loans to me; and the staff of United Cashel Branch for allowing me to sit in at the loan interviews that they were conducting.

And finally but not least, I wish to thank my wife, Trindy, for her encouragement and love.

Table of Contents

1. Introduction

For more than 20 years, scientists working in the field of Artificial Intelligence have been developing computer programs that could solve problems in a way that would be considered intelligent if done by a human.

Emphasis has swayed from developing general-purpose programs to special-purpose ones because the domain of the problem for the first is just too large and difficult to formulate. Since the late 1970's, AI scientists came to realize that a large proportion of the problem-solving power of a program comes from the knowledge it processes, not just from the formalisms and inference schemes it employs. The main theme adopted by many AI researchers is :

> **To make a program "intelligent", provide it with lots of high quality, specific knowledge about some problem area.**

A particular class of programs called expert systems can be defined as computer programs that reproduce the behaviour of a human expert by using a pool of expert knowledge to attain high levels of performance in a narrow problem area. These programs typically represent knowledge symbolically, examine and explain their reasoning process, and often address problem areas that require long periods of special training and education for humans to master.

Research and development of expert systems have grown rapidly in recent years and may continue to do so even more dramatically in the years to come. Expert systems have been developed for many application areas and are beginning to saturate the software market; with prices ranging from less than a hundred dollars to thousands, and ones that run on personal computers to those that can only be run on large mainframes.

At present, there are very few expert systems developed for financial applications. The purpose of this project is to investigate the problem domain of loan-processing within the context provided by the United Building Society, a financial institution in New Zealand, and to develop an appropriate prototype expert system to address that problem.

2. Approach taken to the Project

A systematic approach for system development was taken to ensure that the whole project can be divided into various logical stages. This approach allows a large project to be broken down into various sub-tasks of temporal significance[26] such that at each stage, it is clearly specified what has to be done before going on to the next stage. The task involved at each stage is not done in isolation of the others. If some form of discipline is not adopted and adhered to, then it may well happen that at a later stage of the project, a certain concept or requirement is not understood and the whole project has to come to a standstill to rectify that problem. The relevent stages involved are :

1) Specification and Understanding of the Problem
2) Feasibility Study
3) Initial Design
4) Tool Selection
5) System Development
    . design
    . documentation
    . coding
    . testing and debugging
    . evaluation and feedback
6) Final Evaluation
7) Implementation
8) Report and Final Documentation

The format of discussion in the following sections of the report is roughly dictated by this approach to system design and development.

3. Specification of the Problem

It was essential to understand the nature of the problem before making any sort of deduction about the requirements and design of the system. This project was proposed by Mr R. Whitmore of the United Building Society and sponsored by the supervisor of this project, Dr W. Kreutzer. The primary objective was "To provide a PC based tool that can assist its user to choose between the range of Lending products provided by United".

The first stage of the project was to interact with the personnel at United to get a clear understanding of the problem. This group included the data processing personnel in the information services department, the product development manager as well as branch managers and their loan officers. Frequent appointments were made with some of them to discuss the policies of loan processing and the various requirements of the kinds of loans that United offers. A loan interview refers to the meeting of a client or potential client with a loan officer at any branch office to discuss the prospects and requirements of a loan application. I have attended a few of such interviews to get a better understanding of the information that was exchanged. This was necessary because if the eventual system was found to be credible, it will be initially introduced as a tool at the branch level during loan interviews.

The main problems that I have found can be broadly categorized as those of inconsistency and inflexibility. Quite often during a loan interview, neither the loan officer nor the customer knows what lending product or combination of products is best suited to the latter's needs. The choices involved are the types of loans and the duration of the loan while the parameters are the requirements of the customer, his income and liabilities, and many others. It is the job of the loan officer to guide the customer through a series of questions to establish his needs and hopefully to provide an appropriate recommendation. The only known procedure so far is just a list of facts that are required from the customer to satisfy some mathematically formulated constraints. The skill in achieving this is very much a personal attribute. The loan officer is free to employ his own methods to interrogate the customer. There is nothing wrong at all to allow these loan officers, being human, such form of individuality. However, because the interview is a highly interactive session, the loan officer who has to consider and evaluate a large amount of facts and rules, is exposed to various forms of human errors and inconsistency. Firstly, the answers given by the customer are usually written on a note-pad in a haphazard way that may lead to erroneous retrieval at a later stage.

Secondly, there is no procedure or consistency in the way questions are asked. Sometimes irrelevent ones are asked. Thirdly, the requirement to make some calculations on a hand-calculator may lead to unintended errors. It is the responsibility of the loan officer to make the customer aware of alternative loan products, should he fail to qualify for his first choice. However, factors like the appearance and dressing of the customer, his manners, the loan officer's mood, the environment, the time of the day and many others may affect the loan officer's capacity in recommending the customer an alternative loan. Although these events are unlikely, there is always a possibility for such situations which the management of United will want to avoid.

Besides being inconsistent, the current procedure also encourages inflexibility. To help in the manual calculations, the loan officers are given tables with some rates and figures for periods of multiples of 5 years. A customer who is retiring in 7 years may be recommended for a loan with a repayment period of 5 or 10 years although neither may be in his best interest. Also, a casual customer who only wants to enquire about the loan possibilities that he is eligible for may not be able to get a detailed listing of all of them with the current manual system.

There was an occurrence when a customer who supposedly did not qualify for a loan was offered one due to some errors committed at the branch level. He went away making all the necessary arrangements for the purchase of a property only to have his loan application rejected by the regional head office. Such incidents may tarnish the corporate image of United as a financial institution.

At present, a few constraints govern the lending criterion of each product. It was felt by the product development manager that the structure of these stand-alone constraints will not fully and adequately reflect the customer's ability to service or repay a loan. Suggestions have been made at some of the meetings with the personnel of United that heuristics or rules of thumb of some sort be employed at various points of the loan interview to test the customer's eligibility.

4. Initial Investigation (Feasibility Study)

The next stage of the project involved doing some research on existing expert systems to see if expert systems can be employed to handle the problems specified in the last section. As was mentioned in Section 1, the term 'expert system' is very broadly used to define programs that try to reproduce the behaviour of a human expert in solving a task within a particular problem domain. These tasks can vary from very simple ones to very complicated ones where the only way of solving them is to use some heuristics.

4. 1 Desirable Features of an Expert System

The most useful feature of an expert system is the high-level expertise it provides to aid in problem solving. This expertise can represent the best thinking of the top experts in a field, leading to considerably accurate and efficient solutions.

Another useful feature of an expert system is its provision of some institutional memory. If the knowledge base was developed through interactions with some key personnel, it represents their current policies, operating procedures or strategies. This compilation of knowledge becomes a consensus of opinion and may lead to a permanent record of the strategies and methods used by staff. Even when these personnel leave, their expertise is retained. An expert system can also be used to provide a training facility for new personnel since it already has the ability to explain its reasoning processes.

Another attractive feature of an expert system is its cost effectiveness. It can be quite costly to develop an expert system because it requires a substantial amount of research, investigation and development by the knowledge engineer and programmer. However, it is very cheap to operate an expert system, as the operating cost is just the nominal computer cost of running the program [1]. Human experts, on the other hand, are scarce and hence very expensive especially when the cost of employing them is recurring as compared to the one-time high developmental cost of expert systems.

The next point to be raised about expert systems is that they can perform consistently and can even "justify" their reasoning whereas humans can be unpredictable at times. Expert systems are also much more mobile than human experts. Transferring artificial expertise requires the trivial process of copying a program, while transferring knowledge from one human to another involves a lengthy and expensive process called education.

4.2 Drawbacks of Existing Expert Systems

Many people have welcomed the use of expert systems to solve problems quickly and more consistently. However, these systems should be only seen as tools to assist humans. There are many areas of application where expert systems cannot manage as well as humans.

Expert systems can only perform consistently within the boundary for which the knowledge base is defined. This is to say that an expert system is limited by the knowledge it has in its knowledge base about the real-world and in no capacity is it able to handle "abnormal" situations or problems from a different domain. People are much more creative and innovative. A human expert can reorganize relevent information and use it to synthesize new knowledge in unfamiliar situations.

Human can learn very quickly and adapt to changing conditions. To achieve this in programs is one of the major goals that AI is striving towards. Currently, new concepts are introduced to an expert system by changing or adding some rules; a practice which may or may not cause the new rules to interact with the existing rules in unexpected ways.

Human have commonsense knowledge that can be used to recognize invalid input while an expert system may need to do an exhaustive search of its knowledge base before it can make any conclusion. Expert systems may require the use of certain "heuristic" methods to reduce the amount of search required. Suppose someone was to be asked for the private phone number of the US president, he can tell straight away whether he knows it or not. An expert system may have to search its knowledge base for an answer, and not finding one may begin to ask the user some questions in order to draw some conclusion.

4.3 Pros and Cons of Expert Systems

Some advantages and disadvantages of using expert systems can be tabulated as follows :

	Expert System	Human Expert
Advantages	Consistent	Unpredictable
	Mobile	Difficult to transfer
	Affordable	Expensive
	Permanent	Perishable
Disadvantages	Uninspired	Creative
	Narrow focus	Broad focus
	Technical knowledge	Commonsense
	Rules to be changed	Adaptive and Learns

FIGURE 1

4. 4 Reviews of some Expert Systems

There are many expert systems which have been developed for a wide variety of applications. Most of them are different in many ways. They can be different in terms of the programming language or tool that they were built with, or the way that they handle uncertainty, or the method of representing knowledge and so on. A case study of some of them may lead to a better understanding of various implementation methods and the strategies used by their designers.

4.4.1 MYCIN (Shortliffe 1976)

MYCIN is one of the earliest and most well-known expert systems. It gives physicians consultative advice on the diagnosis and therapy for infectious diseases. The expert knowledge in MYCIN is represented in terms of production rules involving certainty factors, an ad hoc strategy of dealing with uncertainty. MYCIN is implemented in LISP. It employs a backward chaining control scheme. MYCIN constrains search by pruning from a set of hypotheses (sometimes all those in the knowledge base) before all the evidence is in. It is one of the first systems which provide explanations of the system's reasoning process. Studies indicate that the recommendations given by MYCIN compare favourably with the advice given by human experts in infectious diseases. An interesting conclusion made by Cendrowska (1984) is that much of the effectiveness of MYCIN is due to fine-tuning and ad hoc modifications, rather than the use of backward chaining and production rules. MYCIN was developed at Stanford University and reached the stage of a research prototype. [1] [2] [16]

4.4.2 XCON (Carnegie-Mellon University & DEC 1982)

R1/XCON (eXpert CONfigurer) configures VAX 11/780 computer systems, and is reported to save DEC (Digital Equipment Corporation) about $20M per year. It uses about 1200 rules and knows about 500 VAX components. From a customer's order it decides what components must be added to produce a complete operational system and determines the spatial relationships among all of the components. XCON applies its knowledge of the constraints on component relationships to standard procedures for configuring computers. It is non-interactive, is rule based, and uses a forward chaining control scheme (cf. backward chaining in MYCIN). XCON is implemented in OPS5 and is claimed to be the largest rule-based expert system in operation.[1] [16]

4.4.3 MUD (Kahn & McDermott 1985)

MUD is a drilling-fluid diagnostic and treatment consultant expert system recently developed at the Carnegie-Mellon University. It demonstrated a level of competence comparable to the expert "mud" engineers. It has a knowledge base of about 1600 rules. MUD uses an evidential rather than a causal approach to diagnosis. This means that instead of reasoning with a causal model, or an explicit representation of how hypothesized causes bring about symptoms, MUD explicitly represents the weighting of evidence related to its diagnostic conclusions in the program. MUD is said to have compiled diagnostic knowledge. It does not represent the intermediary steps in the causal path from the hypothesized problem to evidential consideration. The algorithm to diagnostic problem solving can be quite simply expressed as :

(1) Generate a set of plausible hypotheses
(2) Order the hypotheses for investigation
(3) For each hypothesis, determine the relevent evidential considerations
(4) Evaluate each hypothesis on the basis of the available evidence
(5) Accept or reject each hypothesis

While MYCIN employs pruning, the method used by MUD to reduce searching is to begin by passively collecting significant observations on the basis of which a relatively small set of initial hypotheses may appear as plausible candidates. MUD assumes that all data entered is certain. According to Kahn and McDermott [12], this approach has not degraded MUD's performance. [12]

4.5 What can be done for the Project

The review above showed three expert systems which employed different techniques in many ways and were used for different applications. Many others were reviewed but have not been presented in this report. Most of these systems were the products of years of hard work and research by some brilliant people in this field of AI. Some of these systems employ sophisticated methods of searching, knowledge representation and dealing with uncertainty.

The aim of the project has been specified and the problem domain has been studied. The next step is to decide what can be done. Is an expert system appropriate for this application ? There is a guideline given in [1] regarding this issue. If the problem took a human expert a few seconds to solve, then there is no point in developing a system to tackle such a trivial task; but if a problem required years of human effort to solve, then it is probably too difficult, at least in the context of this project, to develop an expert system for it, although it may not be impossible. The loan-processing problem in question usually takes a human expert tens of minutes to solve. This seems to qualify it as a reasonable area of application for an expert system.

Many expert systems like MYCIN reason with symbolic knowledge contained within facts or rules. However, for this project, the problem revolves around a financial realm which requires the manipulation of numerical as well as symbolic inputs.

MUD and XCON can be considered as quite small systems with less than 2000 rules. Our loan-processing problem involves the use of even fewer explicit rules. There is however many implicit information embedded within numerical tables. For instance, tables of chargeable interest rates against loan durations and loan types. The problem here is much simpler, quite different, more commercial-based and mathematical. However, if we cease to compare ourselves with the achievements of the creators of MYCIN, XCON or other sucessful expert systems that took years to build, and concentrate on the definition (in Section 1) that "an expert system is a computer program that reproduces the behaviour of a human expert by using a corpus of knowledge to attain high levels of performance in a narrow area", then we can be justified in building a system which we may call an **Expert System** if it can reproduce the behaviour of the human expert, perform its task more quickly, accurately and consistently and avoid the problems mentioned earlier. The actual design and implementation of this expert system will be discussed in the following sections of this report.

5. Initial Design of LOANEX

This section looks at the basic requirements and design of an expert system called LOANEX (LOAN EXpert). It is very important that this be done before any actual development begins, because by specifying the goals and methods first, the actual system development will be made easier with goals to steer towards and methods to adhere to [28]. The following features were considered.

5.1 Interaction between Personnel

Communication was a vital and on-going part of this project. A diagram of the form of interactions involved between various personnel and tools is shown below. I had to serve as both knowledge engineer and programmer. The supervisor gives advice on various aspects of the project, the information services personnel provide systems support and advice, the domain expert provides his expert knowledge about loans and the branch staff conduct the loan interviews.

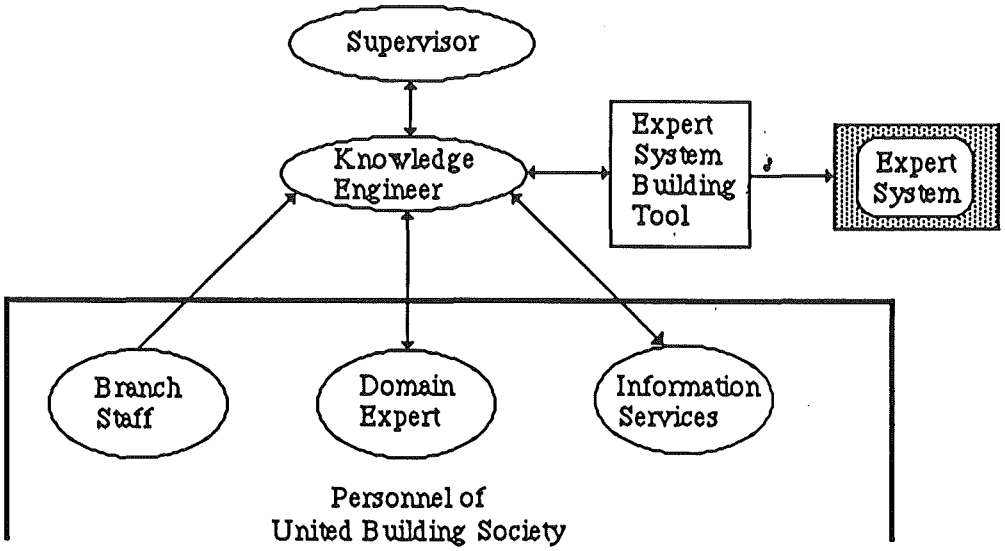


FIGURE 2

5.2 Requirements of the System

The system should behave like a loan consultant. It should serve customers who want to apply for a loan and those who merely want to find out how much they can borrow. It should be able to determine the needs of the customer and recommend a loan. Recommending a loan means to list a schedule of all the possibities, in terms of various

lengths of repayment period, to the customer and leave it to him to decide. When a loan application fails, the system should be able to provide reasons and perhaps to recommend other alternatives or to suggest the maximum amount that can be borrowed based on the information provided. (Refer to Appendix A)

5.3 Representing Knowledge

It has been stressed that the heart of an expert system is its corpus of knowledge. When building an expert system, it is important to choose an appropriate method of knowledge representation because of all the known methods, each provides the system with certain benefits, such as making it more efficient, more easily understood, or more easily modified [1]. The three most widely used methods are rules (the most popular), semantic nets, and frames. The latter two methods seem to be more popular for systems which employ some object-oriented technique of programming.

It was decided that the knowledge in LOANEX be represented in the form of **rules**. Rules are simple enough to be easily explainable. They incorporate human knowledge with a minimum of formalism. Another advantage claimed for rules is that of modularity. Rules can be easily added or deleted without affecting the rest of the system's operation. The skill and domain of the application of the system can be gradually expanded by enlarging the collection of rules in its knowledge base.The concept of rules is supported by many programming languages and tools.

The general form of a rule is :

IF antecedent THEN consequent

Antecedent is a list of one or more conditions which may themselves be the consequents from other rules. The interpretation of a rule is that if the antecedent is satisfied, the consequent can be too. If that consequent specifies an action, that action is executed; if it defines a conclusion, the effect is to infer the conclusion. A danger that is almost equivalent to an endless loop lies in specifying two or more interdependent rules where neither of the consequents can be inferred from any other rules in the knowledge base. This must be avoided :-

IF event_A THEN event_B
IF event_B THEN event_A

The antecedent of a rule (eg event_A) is the consequent of another rule whose antecedent is the consequent of it (ie event_B).

The LOANEX problem can be expressed by rules in a natural form as follows :

IF (it_is_a_residential_loan) AND
(it_is_a_first_mortgage) AND
((the_loan-to-value_ratio) < 85) AND
((the_repayment-to-income_ratio) < 35) AND
((the_sum_of_these_two_ratios) < 110) THEN (eligible_for_a_loan)

The antecedent above consists of 5 clauses which can be classified into 2 broad categories. The first two clauses are of the 'simple' type and can be inferred directly by satisfying some other rules or facts. Such clauses can either return a boolean value or numerical value. The remaining three clauses are of the 'conditional' type. Each involves evaluating an expression which contains operands of the simple type. Such clauses are known as 'conditions' in EXSYS, an expert system shell which will be discussed in Section 6.1. The above 2 types of clauses (simple and conditional) are almost equivalent to the text and mathematical types of conditions in EXSYS respectively [27].

The structure of a production rule system has the form illustrated in Figure 3. The knowledge extracted from the domain expert are formulated and kept in the **production memory** of the knowledge base. The **working memory**, as the name suggests, is temporary and holds information that pertains only to a transaction or enquiry session. The whole knowledge base consists of facts (data) and rules that use those facts as a basis for decision making.

The problem-solving knowledge that enforces the control strategy is called the **inference engine**. The advantage here, which is claimed for production rules systems, is that the domain knowledge is kept separate from the control strategy. The inference engine has an interpreter that decides how to apply the rules in the knowledge base to infer new knowledge or conclusions. It is also responsible for resolving conflicts by deciding which rule to fire next if more than one rule match the current conditions.

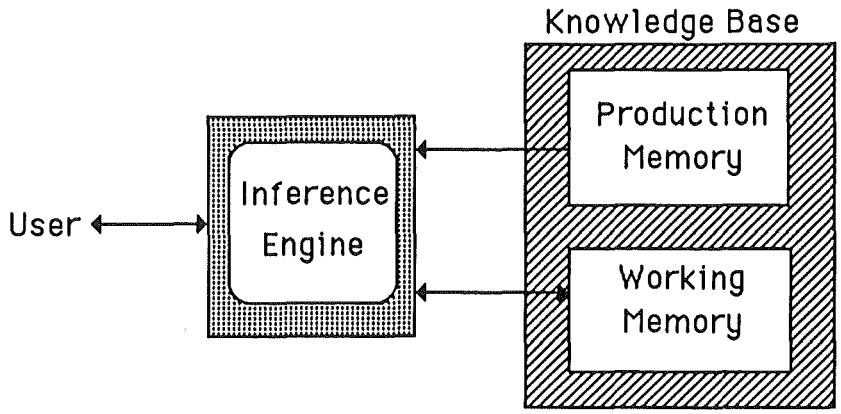


FIGURE 3 - Production Rule System

Inference engines generally adopt one or even both of these control strategies :-
1) Forward chaining (or data-driven, or event-driven)
2) Backward chaining (or goal-driven)

Forward chaining is used for generating theorems from a pool of knowledge. The process involves searching exhaustively through the rule base trying to fire some rule. This method is not appropriate for systems with a large knowledge base. OPS5, a famous rule-based system employs forward chaining. This method can be appropriately applied to 'synthetic' problems in which the solution process is constructive like systems configuration (as in XCON), where not all the solutions need to be enumerated. Backward chaining or goal-driven strategies involve searching through the rule base striving towards a goal. It is suitable for 'analytic' or diagnostic problems in which there is a finite number of candidate solutions which can be characterised in advance. In consideration of this, backward chaining is more appropriate for LOANEX which has a specific goal; that goal is to find out if the customer is eligible for a particular loan, and if so which among the available options is best for him.

5.4 Modelling the Expert System

LOANEX should be able to reproduce the behaviour of a loan officer, in terms of asking the customer for some relevent facts regarding a loan application or enquiry and offering him some recommendation or advice. In doing so, the system will probably be engaged in some form of dialogue with the user. This dialogue is termed **knowledge acquisition** because the system is receiving input or knowledge from some external source. Although the system will initially be used by the loan officer instead of the customer, the dialogue should be designed as though the system is directly communicating with the customer. There is a two-fold advantage in doing so. Firstly, the loan officer can just act as a 'middle-man' in the entire process; he can simply ask the

customer questions that appear on the screen and enter the customer's reply at the keyboard. Of course, the loan officer's job is not as monotonous as that. Being creative and adaptive, as highlighted in Section 4, the loan officer can at various points in time, offer constructive suggestions and guidance in answering those questions. The second advantage of designing a dialogue directed towards the customer is that LOANEX can be used without much alteration to serve customer's loan enquiries at ATM-like* terminals instead of only within a branch office (like the current manual system) if and when LOANEX is found to be credible. The above discussion shows that a **consultation** program is an essential part of LOANEX.

* ATM - Automatic-Teller Machines that are available for 24-hour banking transactions.

5.4.1 Scope of the System

A session that spans from the start of a loan application or loan enquiry to the end of it when a recommendation is provided is termed a **transaction**. The knowledge which is acquired during a transaction is stored in the working memory (see Section 5.2). The system uses this knowledge in conjunction with the permanent knowledge it holds in its production memory to make decisions about a transaction. The former knowledge is removed at the end of a transaction. Although this approach is not realistic in accordance to the real-world situation, it is nevertheless used for simplicity and in order to limit the size of this project. To be realistically practical, this system should have access to some database about customers. When a loan application is successful, some update operations will be done on the database system.

5.4.2 Structuring the Dialogue

It is not practical for LOANEX to expect only 'Yes-No' responses from the user as is the case for a number of existing expert systems. LOANEX should be capable of accepting numerical and symbolic input from the user. From the loan interviews that I have attended and the discussions that I had with the domain expert, I realized that there are so many methods to adopt to achieve the same goals. The objective at this stage was to reconcile all these differences and to produce a general procedure or order of asking questions so that no obviously redundant questions are asked and that LOANEX can decide at the earliest possible stage what branch of action to take or what recommendation to give. LOANEX should find out as soon as possible if a product is not appropriate for the customer and choose from one of the alternatives, if any. To achieve this, I had to play the parts of both the customer and the loan officer and do lots of loan enquiries with different loan requirements, cash input, income and so on, to see what questions should best be asked by the loan officer at each stage.

5.5 User Interface

The final product of any practical software development will eventually be used or assessed by some end-users. In this case, if the project is a success, it will be introduced to the staff at the branch level or even to the customer. Being commercial people, these end-users will probably only want to use any computer systems or tools without getting into too much technical detail and trouble. Quite often, besides functionality, the user interface can determine how much acceptance a system will gain from its users. As such, it is important to design a friendly user interface for a system that will provide the user with what he wants in an easy-to-understand, familiar and acceptable format.

One interesting notion to consider about humans is that they can be quite resistent to changes in their work environment, especially when the change is drastic, or requires training, or threatens their job. In view of this, LOANEX must be functional and yet be easy-to-use and familiar in approach, and be able to instill user-confidence in the system.

5.5.1 Easy to Use

There are many things that can be incorporated into a system to make it easy to use. Firstly, a system whose interface resembles an existing one will require less training for the user before knowing how to use it. Most of the existing systems used by the potential users have colour, windows, menus and so on. It was thought that in order for LOANEX to impress the users at the interface level, it should have the following interface features :

1) Menu Selection
2) Windows
3) Colour
4) English-like dialogue
5) Instructions and Error messages
6) Error correction mechanism
7) Session abortion mechanism

The first 5 features are very common and self-explanatory. The last 2 features were thought to be desirable in allowing the users the flexibility to change some previously entered input or to abort the whole session or transaction totally. Surprisingly, many computer systems and expert systems do not offer such facilities.

5.5.2 Encourage User-Confidence in the System

A desirable feature about expert system which was mentioned in Section 4, and is also found in MYCIN, is the ability to explain the system's reasoning process. As in MYCIN, instead of simply offering the physician, who is an expert in his own right, just a piece of advice, it was felt by MYCIN's implementors that it would be more convincing to the physician if the advice or diagnosis is accompanied by further explanations.

Currently, many expert systems offer some explanations facility by allowing the user to ask questions of the forms : How's and Why's. 'How' is a form of help facility that can be used to ask the system for directions in answering a question. There are 2 types of 'Why' questions. The first type of 'Why' question is motivation-oriented and allows the user to find out the system's motivation in asking the user the latest question. The system will normally respond by dumping a load of rules that govern the consultation program on the screen. The other type of 'Why' question is justification-oriented and can only be asked usually when a goal fails; the system is required to explain why that goal has failed.

The need to cater for the 'How' questions may be made unnecessary by designing questions that are easy to understand and answer. In addition, if any user's response was wrong, the use of meaningful and helpful error messages will provide guidance in answering the questions; there is no need for LOANEX therefore to answer the 'How' question if it is equipped with good error messages.

The first type of 'Why' questions was thought to be unnecessary for this area of application. Consider if this question was asked :

How much do you want to borrow ? $_

There is no reason for the user to have to ask 'Why'. LOANEX's purpose for asking this question should be obvious enough.

To justify the system's action, the second type of 'Why' question should be incorporated in LOANEX. In doing so, a customer will at least be made aware of the reasons why his loan application was rejected. Therefore, LOANEX will have an **explanation** program to handle this type of 'Why' questions.

5.6 Flexibility of the System

One of the advantages of expert systems is that of flexibility in changing the knowledge base. Although expert systems cannot learn in the literal sense, they should at least allow someone (the knowledge engineer, domain expert or end-user) to modify their knowledge bases so that they can adapt to the changing conditions and ever-changing real world if they (the expert systems) were to continue to remain as 'experts'. In designing LOANEX, the knowledge base must be modifiable. However, the ease with which to modify the knowledge base depends to a large extent on the development tool and implementation methods used. Since the actual tool and methods have not been finalised at this design stage yet, we cannot say how easy it is to achieve that. What can be done, at least, is to strive towards developing a program that is easy-to-modify at the programmer's level. Eventually, it is aimed that LOANEX will allow the end-users and not just the programmer, to update its knowledge base with the latest real world 'expert' knowledge. Therefore, desirably, LOANEX should have a **rule-acquisition** program.

5.7 Components of the System

The discussion from the last 3 sub-sections have shown that the LOANEX requires 3 distinct components :-

1) Consultation program
   - to acquire knowledge from the user during a transaction.

2) Explanation program
   - to provide reasons and recommendations when a loan application is rejected.

3) Rule acquisition program
   - to allow modifications to the knowledge base. These modifications should include various rules and heuristics as well as data like interest rates and so on.

The above only suggests three distinct logical components of the system. It does not mean that they are isolated from each other. In fact, they are all incorporated together as a single program with the code for the first two components hardly separable.

6. Selection of Tool

Many expert systems that have been studied were developed in some high level languages or expert system building tools. An expert system building tool, or sometimes known as an expert system shell, is usually built on top of some programming language and is specially designed to support the development of expert systems by offering facilities to represent knowledge and features like built-in inference engine, explanation facilities and so on. A survey in [1] reveals that the most frequently used languages for implementing expert systems are Lisp and Prolog. These are either used directly as development tools, or as the base languages of expert system shells.

The LOANEX problem requires that the tool be developed for an IBM PC or compatible machine because that is the type of machine used by United. Presently, there are far too many advertisements on expert system shells, various Lisp systems and Prolog systems, and many other relevent tools for the PC. Advertisements can be very misleading; there are systems or tools that differ greatly in terms of price, claiming to offer similar facilities for building expert systems. As this project only requires the development of a prototype, it was thought that there should not be any commitment to the purchase of any tool only to find out later that it is not appropriate for the problem.

A few tools were made available to me : MicroProlog; Prolog 86; Turbo Prolog; and EXSYS (of EXSYS Inc. USA). PC Scheme is also considered because it is similar to MacScheme which is available in the department. The selected tool must provide facilities to implement the features and requirements which were specified in the initial design in Section 5.

6.1 EXSYS - good but not good enough

EXSYS is an expert system shell implemented in C. It is rule-based, supports both forward and backward chaining, and has explanation facilities. It allows probabilities to be attached to its knowledge. It supports the English-like IF .. THEN rules illustrated in Section 5.2 with provisions for text conditions and mathematical conditions[27]. It is estimated that EXSYS can support up to 5000 rules. It seemed at first sight to be the best candidate for developing LOANEX. However, after some simple experiments with it, I decided that it is not that appropriate after all.

EXSYS is a general-purpose expert system shell that is easy to use and has many desirable properties. It can be used by someone like the domain expert of United with no programming experience to build an expert system. I suspect that it can be used to solve the LOANEX problem, to a certain extent at least. In that case, the job of the project is just to formulate whatever knowledge as IF .. THEN rules. Every thing else seems to be taken care of by EXSYS. System development will be quite speedy and the whole project may seem to be quite 'small' and simplistic.

Good as it may seem, using an expert system shell for development will cause the final product to lose much flexibility and imagination. The entire user-interface will be determined by the rules entered into the knowledge base. A 'Why' type of question by the user will cause the system to produce a list of relevent rules in their original form. There is **no scope to customize** the questions to ask, the explanations and recommendations to give, and the type of error messages to provide. In fact, most of the features required of LOANEX at the user-interface level (listed in Section 5.4) cannot be implemented with EXSYS and most other shells. Another problem is that of representing tables. There are many tables for numerical data like interest rates that are difficult to represent in the pure form of IF .. THEN rules. Also, loan schedules cannot be produced for the successful applicants. Most of the features and reports provided by the implemented version of LOANEX shown in Appendix A cannot be produced by general purpose expert system shells.

6.2 LISP-based Tools

Shneider, in his paper sent via the UUCP (March 1987), did a review of PC Scheme and other Lisp's. Scheme is a small language that has evolved from Lisp. It is becoming a standard teaching language in many computer science departments in American universities. Scheme is one of the first programming languages to treat procedures as first-class citizens. This means that procedures are similar to numbers or strings and can be passed as parameters to other procedures. Procedures and data values are known as objects in Scheme. They are dynamically allocated in a heap and are automatically deallocated (a process called garbage collection) when they are no longer needed. Scheme has many features desirable for AI application. It was suggested that only Golden Common Lisp and PC Scheme are suitable for the PC because of their relative simplicity and smaller size. Both of these are not available in the department. However, since MacScheme is available, I did some experiments with it. The language is good, but appears to be quite slow and unable to support the user-interface features specified in Section 5. There is a package called "ToolSmith" which MacScheme can use to support the desired screen and user-interface features; but it was not available when I started this project and could not be assessed.

6.3 Turbo Prolog

Prolog (**Pro**gramming in **Log**ic) is quite different from other conventional languages. It is a declarative language with a very short and simple syntax. It uses facts and rules to construct relations between objects. Prolog is very suitable for symbolic processing; it uses a process called unification to match variables with values of any types. Prolog has a built-in inference engine that controls execution automatically. It is a powerful language that is capable of generating all possible values that will satisfy a given goal.

This paragraph is a brief discussion of why Turbo Prolog was selected as the development tool for LOANEX and is in no way trying to promote this product. Turbo Prolog has quite a different syntax from the 'standard' Prolog defined by Clocksin and Mellish [20] but it has the common Prolog property of providing a backward chaining inference engine. It is acclaimed to be one of the fastest versions of Prolog and is the first implementation of Prolog for the IBM PC. It allows rapid development of prototypes with a built-in editor and debugger. There are windowing and graphics facilities. It implements a dynamic relational database which seems appropriate for the working memory of LOANEX. Turbo Prolog allows direct access to the operating system's commands and provides complete access to the computer's I/O ports. This is desirable for printing loan schedules and other reports. Programs developed in Turbo Prolog can be linked to programs implemented in Assembly Language, C, Pascal and FORTRAN. As such, any features which are difficult to implement in Turbo Prolog can be done in one of these 4 conventional languages. Turbo Prolog allows modular programming by offering the facilities to link separately produced modules. Unlike other interpreted Prolog's, Turbo Prolog is a compiler and is capable of generating stand-alone executable programs. This means that LOANEX can be run just like a system command with no necessity for the Turbo Prolog system to be visible at all.

These features together with its easy availability and low cost have prompted me to use Turbo Prolog to implement LOANEX. During the course of the project (especially the later part), there were some features which were found lacking in Turbo Prolog, or any Prolog for that matter. These will be discussed in Section 7 & 8. However, this tool did provide most of what it has promised, and its restrictions did not affect the functionality of LOANEX as a whole.

7. LOANEX - the implementation

This entire section covers what has been achieved and the methods used for implementation. Where weaknesses of the implementation methods are recognized, possible improvements are suggested.

7.1 The LOANEX System

The system consists of a main menu from which three different modules of functionality can be selected. They are :

1) Loan enquiry
2) Data Update
3) Rules Update

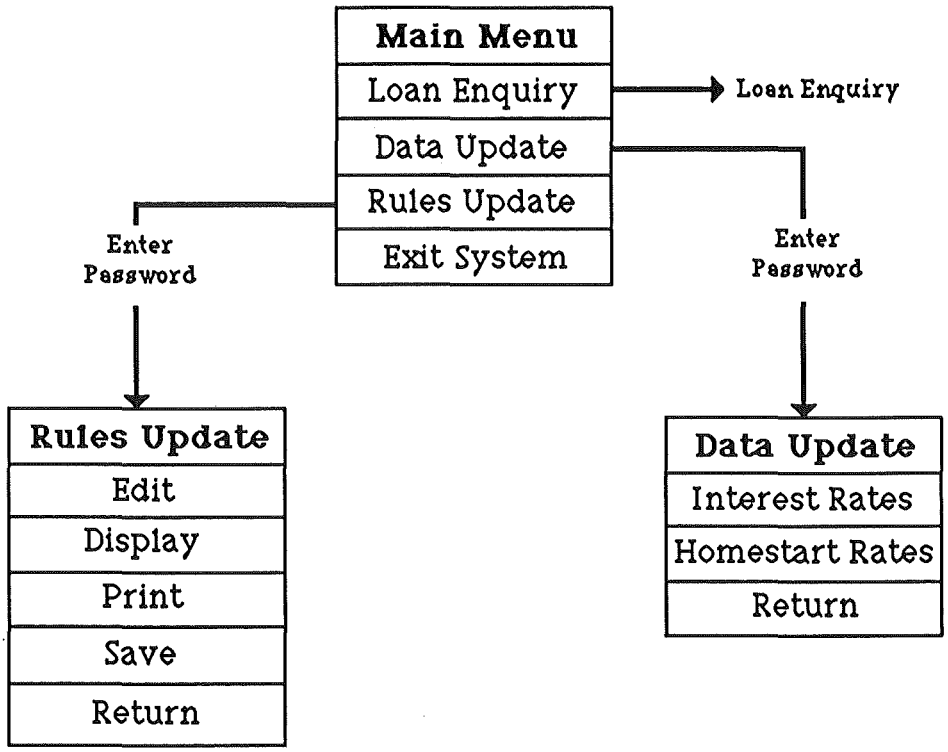
This section only provides a general description of the whole LOANEX system. More details are presented in the following sub-sections and also in Appendix A and B.

Option 1 handles a normal loan enquiry. The system guides the user through a series of questions and provides a recommendation in a tabular form if the customer is eligible for a loan. The questions consist of the yes-no types, menu types and numerical-valued types. At any point of the enquiry, the user can terminate the transaction by pressing the <ESC> key. There is also a mechanism to allow the user to correct the previous answer by entering the backslash (\) character. LOANEX can detect all forms of invalid answers and provide accurate error messages. Even if the customer is not eligible for the requested loan amount based on the given input, LOANEX will still provide advice like what is the maximum amount that can be borrowed. This can provide the customer with at least the choice of buying a cheaper property or of raising more money elsewhere. Reasons are also provided as to why the customer has failed to satisfy any lending criterion.

Option 2 and 3 can only be used by certain 'authorized' users with valid passwords. Option 2 allows alteration of data like the interest rates, homestart statistics and so on while Option 3 allows browsings of and alterations to the rule base that governs the credit evaluation of the customer. When any rule is modified within Option 3, LOANEX will check for the integrity of the rule base to ensure that this change will not interact with other rules to cause detrimental results. The user is not allowed to quit from the Option 3 sub-menu until the proper changes are made to attain integrity of the rule base.

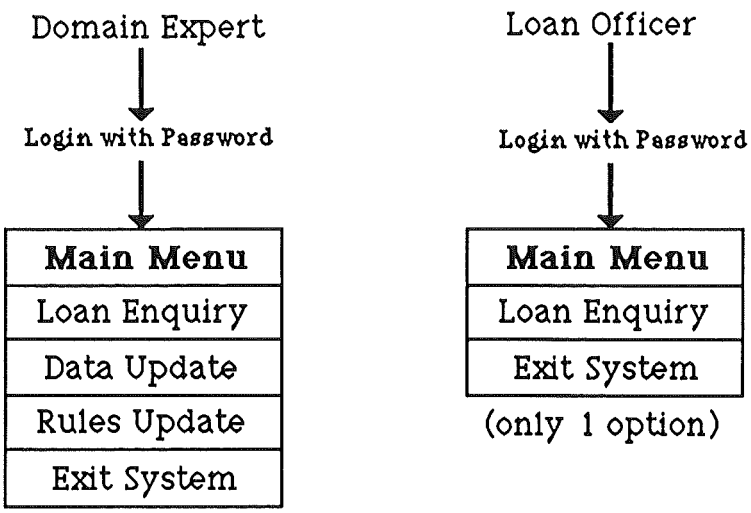
7.2 The Menu System

It is essential to restrict the use of Main Menu Option 2 and 3 to some authorized users. It is unimaginable how much 'harm' is done to the company if someone irresponsible changes the interest rate to 1%. As such, LOANEX has a very simplistic approach of requesting 'passwords' before allowing access to these 2 menu options. The implementation only attempts to reflect the necessity to restrict such update operations: it does not prevent repeated attempts of different passwords and does not use any encryption methods. One good method used by a large anonymous New Zealand company to restrict menu selections is by not showing the restricted ones at all on the screen. This requires an initial login into the system to identify the user and his access rights which are then used to determine the menu options to be offered to him (see Figure 4b). This is not done by LOANEX at present.



The LOANEX menu system and method of restriction

FIGURE 4 (a)



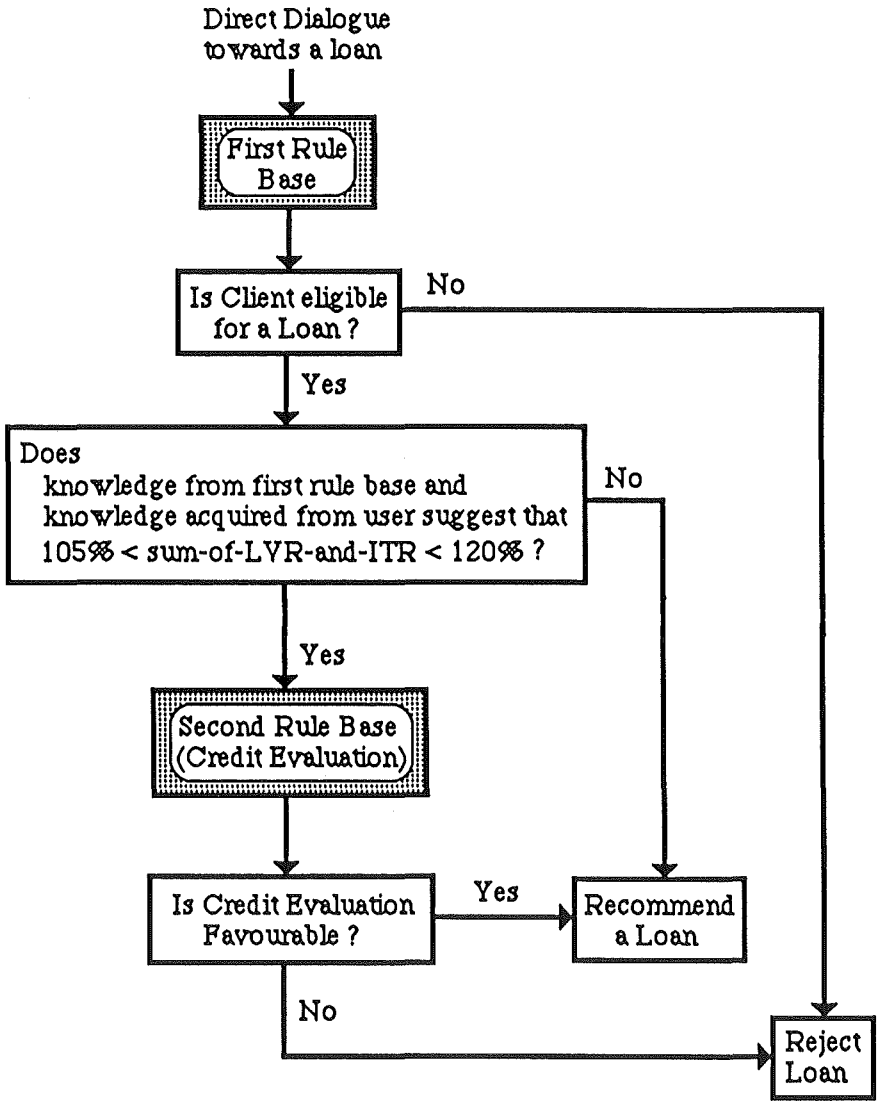
Determining access rights to menu operations during login time

FIGURE 4(b)

7.3 Two Distinct Rule Bases

Although Prolog provides a built-in inference engine that implements backward chaining, this was found to be insufficient for LOANEX. Two separate inference engines have been designed to drive two separate rule bases.

LOANEX has one rule base used to control the dialogue and to direct the system towards a goal. We will call it the **goal-oriented** rule base. This knowledge is stored in the production memory or the static database of the Turbo Prolog system. With this knowledge, LOANEX is able to identify a 'potentially risky' customer. It then calls upon the second **evaluation-oriented** rule base to apply some heuristic methods to determine the customer's credit worthiness before deciding whether to offer a loan or not. A 'potentially risky' customer is one who satisfies the lending criteria of a loan but falls into a predefined 'grey' or risky area. The knowledge for this second set of rules are loaded into the dynamic database from an external file. This is because this knowledge can be modified by some users using Option 3 of the main menu, and Turbo Prolog does not allow its code (or knowledge) to be self-modifying (an inherent weakness of Turbo Prolog which only allows facts to be asserted). So, what LOANEX does is to store this knowledge in a file and read (assert) it into the working memory at the start-up time of the system. However, this knowledge remains for as long as the system is up and not only for a single transaction. The inference engine for the second rule base resides in the static database because the rules representing its structure are stable and remain the same all the time.



How the rule bases work together to process a loan application

FIGURE 5

7.4 The Goal-oriented Rule Base

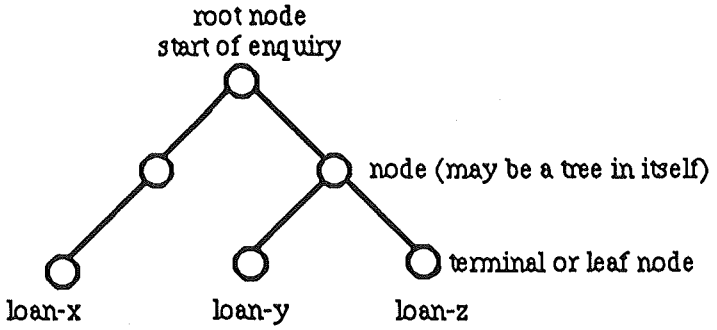
The main task of this rule base is to direct the dialogue towards the goal of selecting an appropriate loan or rejecting it. The structure of the dialogue (way to knowledge acquisition) is modelled like a decision tree with no probabilities attached to the branches. There are many good reasons why this is done. (Refer to Section 7.4.1 on the next page)

7.4.1 Actual Knowledge Representation

Each type of loan is a goal that is represented as a leaf node on the decision tree. The problem mentioned in Section 3 is that no expert uses a standard procedure in reaching each goal. That however does not mean that there cannot be one. After studying the problem for some time, I found that it is possible to introduce some procedure by organizing the dialogue in relation to the data flow of an actual transaction. For a residential loan, the normal human approach is to start from some basic questions and then begin narrowing down to specific ones where necessary. That process is analogous to traversing a decision tree. Each node in the tree can either represent a simple task of asking a question (yes-no, menu or numerical type) and performing some action based on the user's response or a compound task that involves many simple ones. As such, a compound task or node is also like a tree (see Figure 6b). A symbolic name is given to each of these nodes. A branch on the tree is represented by a list of these symbolic names and is also given a name. By using Prolog's way of inferring clauses as either true or false and asserting some flags into the database, LOANEX can decide whether to tranverse a left or right branch.

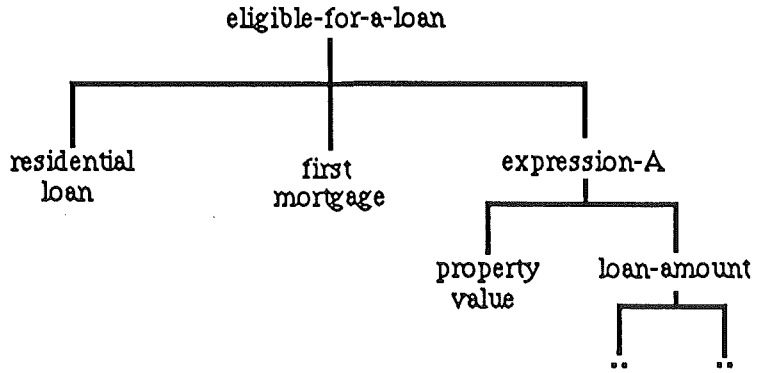
The Turbo Prolog equivalent of the IF .. THEN rule presented in Section 5.3 for the LOANEX problem is given below. These are Prolog facts. The first argument is the consequent or goal of the rule while the second argument is a list of simple and conditional clauses representing the antecedent of the rule. Each clause in the antecedent can itself be a consequent in another rule, as shown below by 'expression-A' and 'loan-amount'. An inference engine is designed that will use these rules to ask questions from the user and make inferences. Note that when rule 2 is fired, the first sub-goal is to satisfy the node 'homestart-not-qual'. This will check other rules in the production memory to see if the customer qualifies for a homestart loan or not. If that sub-goal succeeds (ie does not qualify), the inference engine will go on to satisfy the next sub-goal (property-value) in the list. If this sub-goal fails, a fact is asserted into the dynamic database to signal the inference engine to fire, say, rule 4.

1) rule (eligible-for-loan, [residential-loan, first-mortgage, expression-A, . . .]).
2) rule (expression-A, [homestart-not-qual, property-value, loan-amount, . . .]).
3) rule (loan-amount, [expression-L, . . .]).
4) rule (homestart, [zone, loan-amount, . . .]).



A decision tree consisting of nodes which may themselves be trees.
A node can also be seen as a goal which consists of sub-goals.

Consider a node called 'eligible-for-a-loan'.



A node (or goal) on the tree may have sub-goals

FIGURE 6 (a) and 6 (b)

7.4.2 "Hard-Wired" Knowledge

The feature of rule-acquisition has been implemented for the evaluation-oriented rule base and not for the first goal-oriented rule base. I thought that the knowledge in the first rule base which governs the handling of a loan enquiry is quite stable and does not fluctuate as much as interest rates and other statistics which can be modified via Option 2 of the main menu. This knowledge is the backbone of the main dialogue and had to be the first to be constructed. As Prolog, unlike Lisp or Scheme, does not easily allow its code to be self-modifying, the task of implementing this rule-acquisition feature which requires the modification of code is seen to be a formidable task of the project. Since the dialogue engaged in by the second evaluation-oriented rule base is simpler and more regular, it was decided to attempt implementing such a feature there first (see Section 7.5). If that can be done successfully, the same idea can then be transferred to the goal-oriented rule base. To cater for the inclusion of that feature, the code in this rule base is designed to be as general and modifiable as possible while maintaining its functionality.

There was indeed success in allowing rules to be modified in the evaluation-oriented rule base, but the idea cannot be transferred to the goal-oriented rule base because of various technical problems with the tool and limitations of the host computer (see Section 8.1).

7.4.3 Knowledge as Facts

There is a lot of numerical data in tabular forms for the LOANEX problem. These can be represented in a natural form as facts or conditionless rules in Prolog. A collection of such facts is equivalent to a relational database where each set of facts with the same fact-name represents a relation, and each single fact is a tuple of that relation. Prolog uses the process called unification to match the necessary facts in the database.

```
interest( 1, 2, 18.0).
interest( 1, 3, 18.5).
interest( 2, 5, 20.0).
interest( 2, 10, 21.5).
interest( 3, 5, 19.5).
```

Knowledge as facts

loan-type	loan-length	interest-rate
1	2	18.0
1	3	18.5
2	5	20.0
2	10	21.5
3	5	19.5

Knowledge in a relational database

FIGURE 7

7.4.4 Conflicts from Requirements

The inference engine that is implemented to infer from this goal-oriented rule-base is basically backward chaining. The single requirement of steering towards a leaf or goal node is quite straight forward and is inherent in backward chaining otherwise known as goal-driven. However, conflicts and complications arise when the specified requirements, namely mechanisms to terminate transactions and to correct errors, have to be incorporated.

Backtracking is done automatically in Prolog. To provide a mechanism to terminate a session, such backtracking must be prevented. Pressing the <ESC> key will take the user back to the main menu from the middle of a transaction; it will not re-ask the previous question, a process which backtracking will cause. This is useful for someone who wants to end the transaction abruptly. Such mechanism reflects a 'sensitive' nature of the system which all human experts should have; that is stop asking when the user or client asks you to. Surprisingly, many existing interactive expert systems do not have this feature and will only terminate a session when a conclusion of some sort is reached. The Prolog cut should be used to prevent backtracking and bring about this feature.

However, backtracking is desirable to allow a user to correct a previously entered answer. LOANEX requires a user to enter the '\' key to do that. It is desirable for a practical system to have such a facility to correct errors immediately; otherwise the user may not have confidence in the system for the rest of the transaction. He may assume that the system is processing a figure or input that is incorrect. Another reason is that the dialogue (questions asked) will be different if a 'yes' is entered instead of a 'no'; LOANEX will be tranversing a different branch in the tree.

The conflict caused by these two requirements was overcome by implementing what is called a 'conditional gate'. This gate always allows flows in the forward direction like the cut. When either the termination mechanism or the error-correction mechanism is invoked by the user pressing the appropriate key, a flag (or fact) is asserted into the dynamic database (working memory) and the current sub-goal fails. Control will attempt to backtrack through the gate in the opposite direction. However, the gate will check the asserted flag and will only allow the an error-correction option to backtrack to the previous question. The termination option is stopped at the gate and the main goal (ie enquiry) fails, returning control back to the main menu. The following is a simplified version of the code that implements the backward chaining inference engine and the 'gate' that resolves the above mentioned conflict.

```
backward_chaining ( [ First_Goal | Rest_of_Goals ] ) :-
    backward_chaining ( [ First_Goal ] ), !,     /* solve first goal before the */
    backward_chaining ( Rest_of_Goals ).         /* rest of the goals in the list */

backward_chaining ( [ Simple_Goal ] ) :-
    gate ( Simple_Goal ),                        /* placement of the gate */
    solve_this ( Simple_Goal ).                  /* satisfy a goal, may backtrack */

gate ( _ ) :-
    not ( flag ( termination ) ),      /* always succeed in the first instance */
    not ( flag ( correction ) ).       /* as these flags are not set           */

gate ( _ ) :-
    flag ( termination ), !, fail.     /* a cut to stop backtracking and fail the goal */

gate ( Current_Goal ) :-
    flag ( correction ),               /* allow correction of the previous goal */
    find ( Previous_Goal, Current_Goal ),
    do_something . . .
```

7.5 The Evaluation-oriented Rule Base

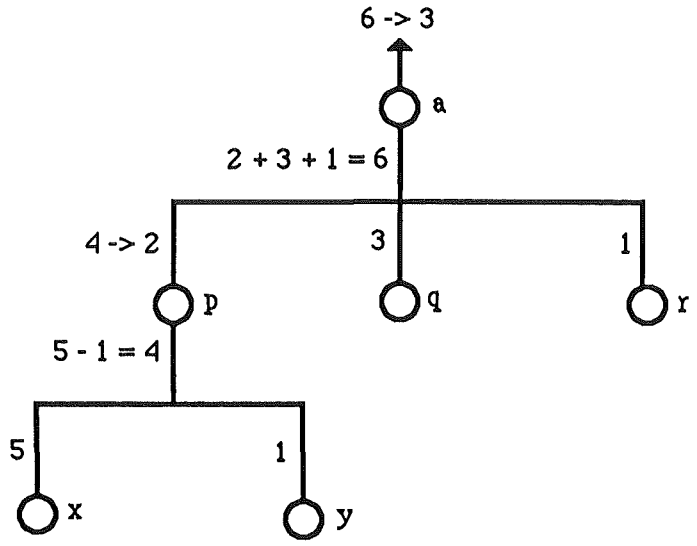
This is the part of the knowledge base that performs a credit evaluation of a customer who is identified to be 'potentially risky'. It consists of a shell which uses some heuristic rules to infer from the user-acquired knowledge. The heuristic rules are modifiable via Option 3 of the main menu. The inference engine for this shell also enforces backward chaining and is very simple (less than 100 lines of code in all). This is because the specific knowledge for credit evaluation is contained entirely in those heuristic rules. The inference engine, knowing the basic structure, simply uses the knowledge in these rules and performs some simple operation.

7.5.1 Actual Knowledge Representation

The knowledge is loaded from a file at the start-up time of the system into Turbo Prolog's dynamic database. There are three types of rules which are called 'modules' in LOANEX. These are **rules, menus** and **questions**. The term 'shell' is used earlier because it is more or less general-purpose and is able to infer from any rule as long as it is defined as one of the above three types. All modules can be considered as functions that return some numerical value. The knowledge about **menus** includes the actual text for the menu itself, some range limits and some option ratings, while the knowledge about a **question** consist of the question text and the response range. A menu will return a predetermined rating of the menu option entered by the user. A question will return the same numerical value as entered by the user. A **rule** is used to structure the dialogue or the rule base. It is in fact an arithmetic expression in which the operands are names of either menus, questions or even sub-rules. With nested definitions of rules, the whole dialogue structure may look like the tree structure below. The four standard arithmetic operators (+, -, *, /) are defined.

Consider the following module definitions and notice how a rule-module causes branching in the tree. The order of evaluation of the tree in Figure 8 is by pre-order depth-first search.

Rule a = p + q + r (1 to 5 -> 2, 6 to 10 -> 3)
Rule p = x - y (1 to 2 -> 1, 3 to 5 -> 2)
Menu q (1 -> 1, 2 -> 3)
Menu y (1 -> 1, 2 -> 0)
Question r (1 to 5)
Question x (2 to 5)



Heuristic Tree Structure and Method of Evaluation
FIGURE 8

Rules **a** and **p** define the branching structure of the tree. The number of operands in the expression represents the number of branches. A rule definition requires the specification of an expression as well as a set of class-ratings. The class-ratings are required so that when the expression returns a value within a certain class or range, the rule will return a specified value to its ancestor. Notice how a value of 4 to Rule **p** is returned as 2 to Rule **a** (p's ancestor). A menu has a one-to-one mapping of the value received to the value returned (eg menu **y** returns 1 when option 1 is selected). A question simply returns the entered value if it falls within the specified range (eg question **r** returns 1 since it falls in the range 1 - 5).

7.5.2 Acquisition of Heuristic Knowledge

The heuristic knowledge in this rule base has been designed to quantify certain information about the customer. For a long time, it has been postulated that a person's credit-worthiness is affected by many factors besides his income. His age, stability of occupation, the kind of credit-cards he has, size of family, and others can also be as important. The domain expert with his years of experience has recognized these factors and has devised a set of questions for LOANEX to ask the customer in order to test his credit-worthiness or his ability to service a loan. Since these questions are just based on some heuristics, they are expected to be very volatile (and may even be inaccurate) and will change quite frequently. As such, a rule-acquisition feature is implemented for this rule base. Rule acquisition refers to the ability of the system to 'learn' by allowing the human user (not necessarily the knowledge engineer or programmer) to modify or add to its knowledge base.

The interface for modifying the rule base is very straight forward and simple. There are prompts and checks where necessary to assist the user in designing a rule, a menu or a question. For each menu option, a rating must be specified and for a question module, a range must be specified. LOANEX can simply use such knowledge to detect whether a user has entered a value beyond the range of acceptable answers or the range of valid menu options. An error message indicating the range of valid responses can then be shown on the screen. The error messages are produced from knowledge implicit in each module. Since during the dialogue for this part, the user is only required to provide numerical (integer) responses, all that the system has to do is to produce an error message indicating the acceptable range. The absence of symbolic input (like yes, no, why and so on) for these heuristic questions makes the input checking much easier.

The whole process of credit evaluation only returns a value to the first goal-oriented rule base which will decide whether to offer a loan or not. As such, this evaluation-oriented rule base does not offer any explanation facilities, only error messages.

The definition for a rule is slightly more complicated. For the example in Figure 8, there are two class-ratings for both rules. Notice that in both cases, the class-ratings are specified over a set of continuous discrete numbers (it is 1 to 10 for Rule **a** and 1 to 5 for Rule **p**). This ensures that the rule is completely defined over this range and that there will not be any ambiquity or error during the inferencing process. If the class-ratings for Rule **a** were defined as (1 to 5 -> 2, 7 to 10 -> 3), then the value of 6 returned by the expression (p + q + r) will cause a system error because the inference engine cannot find a value to return to Rule **a**'s ancestor (ie 6 is undefined for Rule **a**). LOANEX will ensure that this discipline of range continuity is followed during rule-acquisition. The system only deals with integers. During the class-rating definition for a rule, LOANEX ensures that the range is in increasing numerical order and that the lower bound (From) for one class is one more than the upper bound (To) of the previous class.

Class Ratings	From	To	Return-Value
1	1	2	0
2	3	5	4
3	6	10	2

A rule-module with these class-ratings is defined over the range 1 to 10 and may return a value in the set [0, 2, 4].

7.5.3 Simple Parsing and Evaluating

A rule is entered as an expression consisting of module names separated by any one of the 4 defined arithmetic operators and may look like this :

Rule age = actual_age + children * wrinkles

When this is entered, a simple parser will go through the expression, checking that it starts and ends with a module name and that names and operators alternate. There is no necessity to have expressions with brackets; in fact only the + operator needs to be implemented because the heuristics almost always involve adding some quantifiers to see what range the final sum falls within (-, * and / are not used). Anyway, these operators are catered for but no precedence rule is implemented and evaluation is always left to right. Evaluation of the expression will only require scanning through the list from the left, taking two modules which return a value each and performing an operation on these values based on the infix operator. There is no need to use stacks or reverse-polish to handle operators precedence. A correct rule is broken down or tokenised into separate symbols which are stored in a list as follows :

Rule age = actual_age + children * wrinkles
=> **rule** (age, [actual_age, +, children, *, wrinkles]).

7.5.4 Rule Existence Checks

When the above rule is defined, LOANEX will go on to search for the existence of those modules in the list, and request the user to define those that do not already exist in the database. This is done in a depth-first search manner; so if a new module is defined as a rule, then those modules specified within that rule (ie one lower level) are allowed to be defined first before the rest on the same level as this rule. LOANEX allows the user to specify whether he wants to save a definition or not. Suppose if the user does not want to save the definition for module 'children'; LOANEX will proceed to prompt for the definition of module 'wrinkles'. When the end of the list is reached, LOANEX does a second check to ensure that all modules in the list are defined. It continues prompting the user for redefinition until all the modules are defined and saved.

7.5.5 Rule Integrity Checks

Even if all the modules are defined, it may not mean that there is no cause for caution. Each rule is an expression defined over a specified range. A modification of any module below this rule in a hierarchy, may cause the expression to return a value that falls

outside this range. This is a serious problem because the system uses this range to map the possible values returned from the expression into other values that are returned to the next higher level. Any value that is not in this range will not have a mapping and will cause the system to crash. As such, there needs to be some form of range checks to ensure that any modifications to this rule base will leave its integrity intact. Every edit option will invoke such a check. Consider a modification of Question **x** in Figure 8. If the range for this question is changed from (2 - 5) to (1 - 5), then the range (1 - 2) specified by the next higher rule (Rule p) is violated. Rule **p** is represented by the expression x - y. The minimum value for x is now 1 while the maximum value for y remains 1. The resultant value of 0 is outside the range specified by Rule **p**. So, if someone answers 1 for Question **x** and 1 for Menu **y**, the system would fall over if there were no checks. Although a modification made to Question **x** has caused a problem, it will not be returned as the offending culprit. It is the rule (Rule p) which contains this question that is reported. LOANEX will not allow the user to quit from the current menu until the necessary changes are made to retain the integrity of the rule base.

The integrity check is done by evaluating all expressions in the rule base. Since it is possible for each module (rule, menu or question) to return a minimum value and a maximum value, these extremes are used to calculate the extremes of any expression. Which value to take is determined by the infix operator. In the above (x - y) expression, the minimum is derived by taking the minimum for x and subtracting the maximum of y (because of the - operator) from it.

7.6 Programming Techniques

This section provides a brief discussion of the Prolog programming techniques which I have employed while developing LOANEX.

There are three methods of execution control in Prolog [21]. They are backtracking through failing, recursion and the use of cuts.

Prolog does not provide any control structures like the IF, FOR and WHILE statements, which are available in most conventional programming languages. There are many parts of the LOANEX program which are procedural and require these control structures for easy coding. These control structures can be built by the procedurally oriented programming techniques suggested in [18]. Programming with such techniques required a fair amount of familiarity with the Prolog language. The IF..THEN..ELSE structure can be simulated by having an additional rule for a predicate so that when the first (THEN) rule fails, the backtracking mechanism will fire the next (ELSE) rule. However,

if the body of the first rule contains many other clauses (facts or rules), then Prolog will backtrack through those clauses first before trying the second rule. Prolog derives most of its inferencing power by this process of backtracking which is otherwise known as depth-first search or backward chaining.

Since there is no explicit form of loop structure in Prolog, recursion is one of the often used method. However, recursion demands stack space at the expense of execution time. Backtracking should be used instead of recursion to effect repetition whenever possible.

The inherent backward chaining nature of Prolog often causes unexpected backtrackings to irrelevent regions of the code. This problem is partially overcome with the proper use of 'green cuts'. The green cut is used to force binding to be retained once the right clause is reached [21]. These cuts are used to express determinism by preventing unnecessary backtrackings. The other type of cut is known as the red cut which is used to omit explicit conditions. This type of cut can often be replaced by the **not** predicate and should be avoided if possible.

The process of unification was used as often as possible. It has been recommended in [19] that Prolog's unification mechanism should do as much work as possible. Unification takes care of assigning values to uninstantiated variables and testing instantiated ones for equality. For example, to test two lists for equality, the first predicate definition below is preferable to the second in terms of efficiency.

(1) equal (X, X).

(2) equal ([], []).
equal ([H | T1], [H | T2]) :- equal (T1, T2).

Refer to Appendix C (program listing) for more details about various programming and implementation techniques.

8. Evaluation of LOANEX

LOANEX has been designed and developed under the assumption that it will be assessed and eventually be used by some commercial users. The quality of the user interface has been a highly emphasized feature of the system throughout the whole development process. Within the context of the first two menu options (namely Loan Enquiry and Data Update), the users may type <RETURN> to retain some default value and may use the go-back and termination mechanisms as specified in Section 5.5.1. While implementing the last menu option which offers the rule acquisition feature, I have concentrated on allowing the user to change rules and having checks on those rules to ensure that they will not cause any error. During the definition of such a rule, which may be an expression rule, a menu or a question, there are many different types of input required from the user. LOANEX has checks on these input values to ensure that they are correct, but due to the time constraint on this project, LOANEX does not offer any mechanisms for error correction and session termination within this menu option. These mechanisms would require the implementation of a method to resolve the conflict caused by the backtracking nature of Prolog as mentioned in Section 7.4.4. At present, a user may need to be moderately experienced in formulating rules used by LOANEX in order to modify the evaluation-oriented rule base.

8.1 Technical Problems (Hardware and Software)

LOANEX is now solely implemented in Turbo Prolog (version 1.1). It was mentioned in Section 6.3 that it is possible to generate an executable file so that the Turbo Prolog system need not be visible. This will make LOANEX portable and convenient to use (no compilation is required). I have spent many days trying to generate such an executable file by using various techniques recommended in [19]. The first method utilized Turbo Prolog's automatic linker to link all the necessary files and libraries after compilation, but at the current size of the LOANEX program, that process required more than the 640K RAM available on the PC. There was either insufficient memory for stack space or heap space. In fact, the current program for LOANEX cannot be expanded any further because of memory problems. This reflects Turbo Prolog's need for some virtual memory management schemes. The next method involved the division of the program into smaller modules, compiling them separately and linking them from within DOS, the operating system on the PC (ie outside Turbo Prolog). All linkings were successful and an executable file was generated. However, run-time errors started to occur when the system started to retrieve information from the dynamic database.

These issues reflect the need to spend more time on familiarisation with the technical aspects of the Turbo Prolog system. Various other difficulties about programming in Prolog were encountered. The techniques used to overcome them are discussed in Section 7.6.

8.2 Future Improvements and Directions

LOANEX is still quite far from completion regarding its objectives as a 'loan consultant'. Many features are yet to be implemented and there is definitely more room for future improvements and developments.

8.2.1 Expanding the System

At present, LOANEX can only handle four different types of transactions. It is desirable, of course, to increase its knowledge acquisition capability so that it is able to handle **all** the lending products offered by the United Building Society. The consultation program that directs the dialogue will require more rules. This means that the dialogue or tree structure in Figure 6 will grow in size and probably in complexity.

It is not possible, now, for the user to change the rules in the goal-oriented rule base. This has been made possible for the evaluation-oriented rule base, but the idea could not be transferred because the goal-oriented rule base's structure is more complicated and also because of the technical problems mentioned in Section 8.1. However, the idea of allowing someone, perhaps the domain expert, to specify new rules or modify existing ones should be pursued further.

8.2.2 Uncertainty in Knowledge

So far, LOANEX has been designed to handle knowledge that is certain. It does not attach any certainty factors to its knowledge like MYCIN or weightings to hypotheses like MUD (Section 4). In the context of the LOANEX problem, it is actually not very realistic sometimes to see things as either black or white, true or false. Especially for the current heuristic method of enumerating some knowledge concerning a person's credit worthiness, it may be more appropriate to use some form of plausibility factors or probabilities to represent some level of certainty (or doubt) concerning that knowledge. Such knowledge may be represented as :

```
IF ( current_accommodation_flat ) AND
   ( income_per_month > 1200 ) AND
   ( is_engaged ) THEN ( moving_within_next_2_years  IS  0.9 )

IF ( moving_within_next_2_years ) AND
   ( changed_3_jobs_in_previous_year ) THEN ( ability_to_repay IS 0.15 )
```

8.2.3 Option of Using Other Tools

Despite the problems mentioned in the Section 8.1, Turbo Prolog actually provides a good environment for developing prototypes. Since procedures written in C, Pascal, FORTRAN and Assembly Language can be called from Turbo Prolog, most of the procedural routines can be implemented in these languages. These possibilities have not been fully explored during the course of the project. There are a few software packages, like BridgeWare, that link (specifically) Turbo Prolog with database systems like DBase III, spreadsheet systems like MultiPlan and so on. Such possibilities may allow LOANEX to be integrated into a large information system. From the amount of commercial advertisements and literature available on Turbo Prolog, it appears that Turbo Prolog may become a popular software development tool, "especially for expert systems or expert system shells" according to [19], [21] and many other advertisements in computer magazines.

Expert system shells should again be considered in future. I have only experimented with EXSYS, a shell which according to the review in [17] is averagely priced and of moderate usefulness. Shells provide the environment to develop expert systems rapidly but lack an amount of flexibility as mentioned in Section 6.1. The writers of [17] and many other articles have recommended using shells first before attempting to develop a system from scratch. If there is any expert system shell that will allow its user to use some 'base' language or facilities to implement customized features like messages and reports generation, then the option of using it should be considered for LOANEX.

After reading more about Scheme, and developing some programs in it, I found that it is a good language to use. I am not surprised that Scheme, and various dialects of Lisp are so popular with AI researchers (see Section 6.2). If PC Scheme is available with additional features similar to those of Turbo Prolog (see Section 6.3 regarding features that will support user-interfaces), it could also be a very good candidate as a tool for implementing a later version of LOANEX.

9. Conclusion

LOANEX has reached a stage of being an assessible prototype expert system to address the loan-processing problem in the context of the United Building Society. Although it is able to solve only a subset of the problems, it should provide sufficient grounds for the loan experts to assess how useful a tool like LOANEX can be for that particular area of application. Three basic attributes of most expert systems are provided : the ability to consult, the ability to explain and the ability to acquire new knowledge. There is however, still much more to be done before LOANEX can "move into the real world".

During the course of this project, I have learnt a lot about Prolog and MS-DOS (the operating system for the PC that has been used). I have also read many interesting materials concerning expert systems which have truly intrigued me.

The major tasks involved in this project are communicating with people, understanding the problem and designing the system (as an analyst), formalising the knowledge (as a knowledge engineer), coding the program (as a programmer) and doing everything else (as a student). I am glad that I have done this project.

References

[1] Waterman, D. A.; *A Guide to Expert Systems.*
Addison Wesley, 1986.

[2] Hayes-Roth F., Waterman D. A. & Lenat D. B.; *Building Expert Systems.*
Addison Wesley, 1983.

[3] O'Shea, T. & Eisenstadt, M; *Artificial Intelligence: Tools, Techniques and Applications.*
Harper & Row, 1984.

[4] Winston, P. H.; *Artificial Intelligence.*
Addison Wesley, 1984.

[5] Weiss, S. & Kulikowski, C.; *A Practical Guide to Designing Expert Systems.*
Rowman & Allanheld, 1984.

[6] Gondran, M.; *An Introduction to Expert Systems.*
Mcgraw-Hill, 1986.

[7] Zualkernam I., Tsai W. T. & Volovik D.; "Expert Systems and Software Engineering",
IEEE Expert, Winter 1986, Vol. 1, No. 4.

[8] Elliot, L. B. & Scacchi, W.; "Towards a Knowledge-Based System Factory",
IEEE Expert, Winter 1986, Vol. 1, No. 4.

[9] Bahill, A. T. & Ferrell, W. R.; "Teaching an Introductory Course in Expert Systems",
IEEE Expert, Winter 1986, Vol. 1, No. 4.

[10] Williams, C.; "Expert Systems, Knowledge Engineering, and AI Tools",
IEEE Expert, Winter 1986, Vol. 1, No. 4.

[11] Myers, W.; "Introduction to Expert Systems",
IEEE Expert, Spring 1986, Vol. 1, No. 1.

[12] Kahn, G. & McDermott, J.; "The Mud System",
IEEE Expert, Spring 1986, Vol. 1, No. 1.

[13] Elliot, L. B.; "Analogical Problem-Solving and Expert Systems",
IEEE Expert, Summer 1986, Vol. 1, No. 2.

[14] Neale, M.; "Expert Systems in Prolog",
University of Canterbury, Honours Project, 1984.

[15] Andreae, D.; "Design and Development of a Simple Diagnostic Expert System",
University of Canterbury, Honours Project, 1986.

[16] Andreae, J. H.; "Man-Machine Studies", University of Canterbury,
Department of Electrical and Electronics Engineering, May 1986.

[17] Olsen B., Pumplin B. & Williamson M.; "The Getting of Wisdom : PC Expert System Shells",
Computer Language, March 1987,Vol. 4, No. 3.

[18] Munakata, T.; "Procedurally Oriented Programming Techniques in Prolog",
IEEE Expert, Summer 1986, Vol. 1, No. 2.

[19] *Turbo Prolog Owner's Handbook.* Borland International.

[20] Clocksin, W. F. & Mellish, C. S.; *Programming in Prolog.*
Springer-Verlag, 1981.

[21] Townsend, C.; *Introduction to Turbo Prolog.*
Sybex, 1987.

[22] Clark, K. L. & McCabe, F. G.; *Micro Prolog : Programming in Logic.*
Prentice Hall, 1987.

[23] Bratko, I.; *Prolog Programming for Artificial Intelligience.*
Addison Wesley, 1986.

[24] Dybvig, R. K.; *The Scheme Programming Language.*
Prentice Hall, 1987.

[25] Abelson, A. & Sussman, G. J.; *Structure and Interpretation of Computer Programs.*
The MIT Electrical Engineering and Computer Science Series,
MIT Press, 1985.

[26] Kroenke, D.; *Business Computer Systems : An Introduction.*
Mitchell, 1985.

[27] Huntington, D.; "EXSYS User's Manual",
EXSYS Inc., USA, 1984.

Appendix A : Sample Transactions

Messages enclosed in <<< >>> are comments that will not be shown to the user.

Example 1 :

Press <ESC> to return to main menu
Enter \' to re-answer previous question

(1) Residential
(2) Supplemental
(3) Sections
(4) Commercial
(5) Farms
What is the purpose of the loan ? **1**

Do you have a homestart loan (y/n) ? **n**

Do you
(1) Have a property and a loan amount in mind
(2) Want to know how much you can borrow
Enter 1 or 2 : **1**

What is the value of the property ? **$100000**

The maximum loan based on the maximum risk of 85% is $83333.00
How much do you wish to borrow (<RETURN> for maximum amount) ? **$83333.00**

It is assumed that you have a cash contribution of $16667.00
If there are other loans, please enter the repayments as liabilities later

How many income inputs will there be ? **1**

Enter information on Income #1
(1) weekly
(2) fortnightly
(3) monthly
(4) annually
What type is it ? **4**

Income amount : **$23000**

Total Monthly Income is $1916.67

How many other liabilities do you have (<RETURN> for none) ? **<RETURN>**

Are you buying the property through United Realty World (y/n) ? **n**

Sorry, you are not eligible for a loan
Press any key to continue or (w)hy for reasons : **w**

... continues on the next page

A person is eligible for a loan if
1) The loan-to-value ratio is no more than 85% and
2) The repayment-to-income ratio is no more than 35% and
3) The sum of these 2 ratios is no more than 110%

Your loan-to-value ratio of 85.00% is acceptable

Your monthly repayment over 20 years is $1621.75
Your current other liabilities amount to $ 0.00
Your monthly income is $1916.67
Your repayment-to-income ratio of 84.61% is NOT acceptable

Recommendation :
You wanted a loan of $83333.00
But based on the information you have supplied,
the maximum loan that you can make is $24622.00
You need more cash input - loan amount needs to be smaller

Press any key to continue : (control will return to the main menu)

Example 2 **:**

Press <ESC> to return to main menu
Enter '\' to re-answer previous question

(1) Residential
(2) Supplemental
(3) Sections
(4) Commercial
(5) Farms
What is the purpose of the loan ? **1**

Do you have a homestart loan (y/n) ? **n**

Do you
(1) Have a property and a loan amount in mind
(2) Want to know how much you can borrow
Enter 1 or 2 : **1**

What is the value of the property ? **$60000**

The maximum loan based on the maximum risk of 85% is $50000.00
How much do you wish to borrow (<RETURN> for maximum amount) ? **$50000.00**

It is assumed that you have a cash contribution of $10000.00
If there are other loans, please enter the repayments as liabilities later

How many income inputs will there be ? **1**

Enter information on Income #1
(1) weekly
(2) fortnightly
(3) monthly
(4) annually
What type is it ? **3**

Income amount : **$1800**

How many other liabilities do you have (<RETURN> for none) ? **<RETURN>**

Are you buying the property through United Realty World (y/n) ? **y**

Credit Evaluation Begins <<< grey area identified >>>

Age
1) Under 21
2) 21 - 25
3) 26 - 36
4) 37 - 50
5) Over 50
What age group do you belong to ? **4**

How many years have you stayed at your current address ? **10**

How many years have you stayed at your previous address ? **15**

Accommodation
1) Living with parents
2) Boarding
3) Renting
4) Ownership home with mortgage
5) Mortgage-free ownership home
What is your current type of accommodation ? **4**

Duration with your current employer
1) Under 1 yr
2) 1 - 2 yrs
3) 2 - 5 yrs
4) 5 - 10 yrs
5) Over 10 yrs
Which duration ? **5**

Amount of weekly take-home pay
1) Under $100
2) $100 - $150
3) $150 - $200
4) $200 - $ 250
5) Over $250
How much ? **5**

% of take-home pay committed to payments
1) Over 50%
2) 40 - 50%
3) 30 - 40%
4) 20 - 30%
5) Under 20%
What percentage ? **5**

Credit-cards
1) None
2) Bankcard / Visa
3) Amex / Diners
4) Goldcards / Premium
What type do you have ? **3**

Credit report findings
1) Poor
2) Average / Nothing adverse
3) Excellent
What type of finding ? **3**

Press any key to list loan schedule :

Monthly Repayment Schedule

Loan Period(Years)	Repayment($)	Total Ratio(%)	Interest/$1000
20	952.86	116.76	18.68
19	956.88	116.90	18.76
18	961.83	117.06	18.86
17	967.94	117.26	18.98
16	975.50	117.52	19.13
15	965.78	117.19	18.94
14	977.75	117.59	19.17
13	992.69	118.09	19.46
12	1011.48	118.72	19.83
11	1035.29	119.51	20.30
10	1048.33	119.94	20.56

Do you want to print this schedule (y/n) ? (control will return to the main menu)

Example 3 : (Homestart loan)

(1) Residential
(2) Supplemental
(3) Sections
(4) Commercial
(5) Farms
What is the purpose of the loan ? **1**

Do you have a homestart loan (y/n) ? **yes**

What is the amount of homestart loan that you have ? **$5000**

What zone is the property situated in (1, 2, 3) ? **1**

Do you
(1) Have a property and a loan amount in mind
(2) Want to know how much you can borrow
Enter 1 or 2 : **2**

How many income inputs will there be ? **1**

Are you a sole applicant (y/n) ? **y**
<<< note that 1 income input may not mean a sole applicant >>>

Enter information on Income
(1) weekly
(2) fortnightly
(3) monthly
(4) annually
What type is it ? **1**

Income amount : **$800**

Anomaly detected :
The income that you have given amounts to $41600.00 annually
According to the Housing Corporation, if your homestart loan is $5000
your annual income should be $22901 - $23900

Shall I
(1) Take the income range set by the Housing Corporation
(2) Take the income that was given
Enter 1 or 2 : **1**

Press any key to display loan amounts and requirements :

Loan Amount Eligibility (20 year period)

	Purchase Through United Realty World $	Purchase Through Other Sources $
Available homestart loan	5000.00	5000.00
Maximum UBS loan offered	31353.00	30702.00
Minimum cash contribution	5433.00	5335.00
Minimum house price	41786.00	41037.00
Maximum house price	90000.00	90000.00
Monthly repayment	598.00	590.00

The minimum cash contribution is based on 13% of the house price
For every dollar above the minimum house price,
you need an extra dollar of cash contribution

Do you want to print this (y/n) ? (control will return to the main menu)

Appendix B : Rule Acquisition

A simple example of Editing a Module in the Evaluation-oriented Rule Base :

<<< a list of all existing module names is displayed >>>
Select name of module : **Age**

Age is currently defined as a menu

Do you want to define Age as a
(1) Rule
(2) Menu
(3) Question
Select option : **1**

Defining Age as a rule ...
Enter expression for Rule Age
Rule Age = **Actual_Age - MakeUp**

How many class ratings are there for Rule Age ? **3**

Class	From	-	To	Rating	<<< user is required to supply >>>
1	**1**		**20**	**5**	<<< the last 3 numbers on each line >>>
2	**21**		**40**	**3**	<<< *cursor positioning is handled>>>
3	**41**		**120**	**2**	<<< by the LOANEX system >>>

Do you want to save this rule (y/n) ? **y**

<<< prompt for a module definition >>>
Please enter information on module 1 : Actual_Age of Rule Age

Do you want to define Actual_Age as a
(1) Rule
(2) Menu
(3) Question
Select option : **3**

Defining Actual_Age as a Question ...

Enter Question Actual_Age : **What is your actual age ?** <<< type in a string >>>

Enter the range of valid responses (eg 1 - 5) : **1-100**

Do you want to save this question (y/n) ? **y**

<<< prompt for a module definition >>>
Please enter information on module 2 : MakeUp of Rule Age

Do you want to define MakeUp as
(1) Rule
(2) Menu
(3) Question
Select option : **2**

Defining MakeUp as a Menu ...

How many menu options are there for Menu MakeUp ? **3**

Enter the introductory line for menu : **How much make-up do you have ?**

Menu-Option	Rating	
1) **None**	**0**	<<< cursor positioning is handled >>
2) **Moderate**	**3**	<<< by the LOANEX system >>>
3) **Very much**	**9**	

Enter the prompt line for menu : **How much do you use ?**

Do you want to save this menu (y/n) ? y

OK, integrity check is completed; nothing adverse

(control will return to the sub-menu Rules Update)

Appendix C : Program Listing

The program listings provided herein only include the code used by LOANEX to handle the third menu option of rule acquisition and the code used as the inference engine for credit evaluation. The rest of the program code for LOANEX is found in the file "loanex.pro" which is contained in the diskette accompanying this report.

```
clauses

/*------------------------------------------------------------ *
 * Routines to handle the rule-acquisition part of the system *
 * ie Menu option 3 of the main menu                          *
 * Only authorised users are allowed to select this option    *
 * This is implemented by the need to enter 'passwords'       */

/* Check for the existence of a module */

   module_existence(Name) :-
      db(rules(Name, _, _, _, _, _)).

   module_existence(Name) :-
      db(to_ask(Name, _, _, _, _)).

/* Print the whole heuristics rule base */

   print_rule_base(_, []) :- nl, !.

   print_rule_base(N, [Str | T]) :-
      N mod 2 = 1,
      db(rules(Str, _, _, _, _, List)),
      display_it(Str),
      print_rule_base(1, List),
      N1 = N + 1,
      print_rule_base(N1, T).

   print_rule_base(N, [Str | T]) :-
      N mod 2 = 1,
      display_it(Str),
      N1 = N + 1,
      print_rule_base(N1, T).

   print_rule_base(N, [_ | T]) :-
      N1 = N + 1,
      print_rule_base(N1, T).

/* Display the contents of a module */

   display_it(Name) :-
      db(rules(Name, _, _, _, _, _)),
      write("\n                Rule-Base Structure for Rule ", Name),
      write("\n=============================================================\n"),
      display_rule(nested, 1, 1, [Name]).

   display_it(Name) :-
      db(to_ask(Name, 'm', Lo, Hi, Menu)),
      write("\n                 Rule-Base Structure for Menu ", Name),
      write("\n=============================================================\n"),
      write(Menu), nl, nl,
      display_rates(Lo, Hi, Name).

   display_it(Name) :-
      db(to_ask(Name, 'q', Lo, Hi, Quest)),
      write("\n                 Rule-Base Structure for Question ", Name),
      write("\n=============================================================\n"),
      write(Quest), nl, nl,
      write("Accepted response is between ", Lo, " to ", Hi), nl.
```

```
/* Display the ratings of a menu */

   display_rates(L, H, _) :- L > H.

   display_rates(L, H, Name) :-
      db(ratings(Name, L, Score)),
      write(" Option -> ", L, "    Score -> ", Score), nl,
      L1 = L + 1,
      display_rates(L1, H, Name).

/* Display a rule in the nested or indented form */

   display_rule(_, _, _, []) :- !.

   display_rule(nested, N, K, [Str | T]) :-
      K mod 2 = 1,
      db(rules(Str, _, _, _, _, List)),
      check_tab(N, K),
      write(Str, " (Rule)\n"),
      N1 = N + 1, K1 = K + 1,
      display_rule(nested, N1, 1, List),
      display_rule(nested, N, K1, T).

   display_rule(nested, N, K, [Str | T]) :-
      K mod 2 = 1,
      db(to_ask(Str, Ch, _, _, _)), !,
      str_char(Sc, Ch), opsign(Sc, Type),
      check_tab(N, K),
      write(Str, " (", Type, ")\n"),
      K1 = K + 1,
      display_rule(nested, N, K1, T).

   display_rule(nested, N, K, [Str | T]) :-
      K mod 2 = 0,
      opsign(Sign, Str),
      Tab = N * 8 - 2, tab(Tab),
      write(Sign), K1 = K + 1,
      display_rule(nested, N, K1, T).

/* Display all the modules in a columnar form */

   display_rule(column, N, K, [Str | T]) :-
      K mod 2 = 1,
      db(rules(Str, _, _, _, _, List)),
      write(' ', Str, " (Rule)"),  nl,
      K1 = K + 1,
      display_rule(column, N, 1, List),
      display_rule(column, N, K1, T).

   display_rule(column, N, K, [Str | T]) :-
      K mod 2 = 1,
      db(to_ask(Str, Ch, _, _, _)),
      str_char(Sc, Ch), opsign(Sc, Type),
      write(' ', Str, " (", Type, ")\n"),
      K1 = K + 1,
      display_rule(column, N, K1, T).

   display_rule(column, N, K, [_ | T]) :-
      K mod 2 = 0, K1 = K + 1,
      display_rule(column, N, K1, T).
```

```
/* Display a rule in the formula form */

   display_rule(formula, N, K, [Str | T]) :-
      K mod 2 = 0,
      opsign(Sign, Str),
      write(' ', Sign), K1 = K + 1,
      display_rule(formula, N, K1, T).

   display_rule(formula, N, K, [Str | T]) :-
      cursor(_, C), C < 70, write(' ', Str),
      K1 = K + 1,
      display_rule(formula, N, K1, T).

   display_rule(formula, N, K, [Str | T]) :-
      nl, write("                ", Str),
      K1 = K + 1,
      display_rule(formula, N, K1, T).

/* Do some indenting */

   check_tab(N, 1) :-
      Tab = N * 8, tab(Tab).

   check_tab(_, _) :- write(' ').

   tab(0) :- !.

   tab(N) :- write(' '), N1 = N - 1, tab(N1).

/* Edit a module */

   edit_proc :-
      makewindow(2, 23, 112, "Edit Modules", 1, 1, 23, 50), nl,
      display_rule(column, 1, 1, ["CreditEval"]),
      write("\n Select name of module : "),
      cursor(R, C), get_string(R, C, S),
      removewindow,
      edit_rule(S).

   edit_rule(Name) :-
      db(rules(Name, _, _, _, _, List)), !,
      write("\n ", Name, " is a Rule = "),
      display_rule(formula, 1, 1, List), nl,
      cursor(Row, Col),
      write("\n Do you want to\n 1) Redefine this rule\n"),
      write(" 2) Change the current class-ratings\n Enter [1 or 2] : "),
      cursor(R, C), get_int(R, C, Select),
      Select >= 1, Select <= 2,
      cursor(Row, Col), nl, nl, nl, nl, cursor(Row, Col),
      edit_type(Select, Name).

   edit_rule(Name) :-
      not(opsign(_, Name)), db(rules(_, _, _, _, _, List)),
      in_list(Name, List),
      db(to_ask(Name, 'm', _, _, Quest)),
      !, write("\n ", Name, " is currently defined as a menu :\n"),
      write(Quest), nl,
      write("=========================================================="),
      proc(Name).
```

```
   edit_rule(Name) :-
      not(opsign(_, Name)), db(rules(_, _, _, _, _, List)),
      in_list(Name, List),
      db(to_ask(Name, 'q', Lo, Hi, Quest)),
      !, write("\n ", Name, " is currently defined as a question :\n"),
      write(Quest), nl,
      write("Range of valid response is ", Lo, " to ", Hi), nl,
      write("============================================================"),
      proc(Name).

   edit_rule(S) :-
      write("\n Module ", S, " cannot be found in the rule base\n"),
      write("\n Press any key to try again : "), readchar(_),
      clearwindow, edit_proc.

/* Find out what type it is and do it */

   edit_type(1, Name) :-
      proc(Name).

   edit_type(2, Name) :-
      clearwindow,
      write("\n Redefinition of class-ratings for Rule ", Name),
      write("\n Current Class Ratings :\n"),
      write("           From  -  To     Rating"),
      show_ratings(Name),
      write("\n\n Redefine the Class Ratings :\n"),
      write(" How many class ratings will there be ? "),
      cursor(R, C), get_int(R, C, Num),
      write("\n Class     From  - To     Ratings\n"),
      class_ratings(1, Num, [], List, 0, 0, 0, 0, X1, X2, X3, X4),
      write("\n Do you want to save the changes (y/n) ? "),
      readchar(Ch), write(Ch),
      db(rules(Name, _, _, _, _, RuleList)),
      save_rule(Ch, Name, X1, X2, X3, X4, RuleList, List).

/* Show the current class ratings of a rule */

   show_ratings(Name) :-
      db(c_rates(Name, Lo, Hi, Rate)),
      nl, cursor(R, _),
      cursor(R, 12), write(Lo),
      cursor(R, 20), write(Hi),
      cursor(R, 29), write(Rate),
      fail.

   show_ratings(_).

/* Check if a module exists */

   in_list(H, [H | _]) :- !.

   in_list(H, [_ | T]) :- in_list(H, T).

/* Notify user of a bad rule that has violated the rule base */

   notice(N) :-
      range(Rule, Lval, Hval),
      db(rules(Rule, Lo, Hi, _, _, _)),
```

```
    Lval >= Lo, Hval <= Hi, !,
    retract(range(Rule, _, _)),
    notice(N).

notice(N) :-
    range(Rule, Lval, Hval),
    db(rules(Rule, Lo, Hi, _, _, _)),
    Lval < Lo, !, retract(range(Rule, _, _)),
    asserta(badrule(Rule, Lo, Hi, Lval, Hval)),
    print_notice(Rule, Lo, Hi, Lval, Hval),
    N1 = N + 1, notice(N1).

notice(N) :-
    retract(range(Rule, Lval, Hval)),
    db(rules(Rule, Lo, Hi, _, _, _)),
    Hval > Hi,
    asserta(badrule(Rule, Lo, Hi, Lval, Hval)),
    print_notice(Rule, Lo, Hi, Lval, Hval),
    N1 = N + 1, notice(N1).

notice(0) :-
    write("\n OK, integrity check is completed; nothing adverse\n").

notice(_) :- !.

print_notice(Rule, Lo, Hi, Lval, Hval) :-
    write("\n Rule ", Rule, " has no integrity : "),
    write("\n Specified range is ", Lo, " to ", Hi),
    write("       Possible range is ", Lval, " to ", Hval),
    write("\n Please make appropriate change to Rule ", Rule), nl.

/* Check the integrity of the whole rule base */

integrity_check([Rule], K, L, H, Low, High) :-
    K mod 2 = 1,
    db(rules(Rule, _, _, Lo, Hi, List)),
    retract(maths(What)),
    asserta(maths("add")),
    !, integrity_check(List, 1, 0, 0, Lval, Hval),
    asserta(range(Rule, Lval, Hval)),
    calc_bounds(low, What, L, Lo, Hi, Low),
    calc_bounds(high, What, H, Lo, Hi, High).

integrity_check([Head | Rest], K, L, H, Lv, Hv) :-
    not(Rest = []),
    integrity_check([Head], K, L, H, Lval, Hval), K1 = K + 1, !,
    integrity_check(Rest, K1, Lval, Hval, Lv, Hv).

integrity_check([S], K, Lo, Hi, Lo, Hi) :-
    K mod 2 = 0,
    asserta(maths(S)).

integrity_check([S], K, L, H, Lval, Hval) :-
    K mod 2 = 1,
    db(to_ask(S, Ch, Low, High, _)), !,
    bounds(S, Ch, Low, High, Lo, Hi),
    maths(What), retract(maths(_)),
    calc_bounds(low,  What, L, Lo, Hi, Lval),
    calc_bounds(high, What, H, Lo, Hi, Hval).
```

```
/* Get the limits of an expression with the given possibilities */

   bounds(_, 'q', L, H, L, H) :- !.

   bounds(S, 'm', L, H, Lo, Hi) :-
      get_bounds(S, L, H, 1000, -1000, Lo, Hi).

   get_bounds(_, L, H, Lo, Hi, Lo, Hi) :- L > H, !.

   get_bounds(S, L, H, Lo, Hi, Low, High) :-
      db(ratings(S, L, Val)),
      compare(min, Lo, Val, L1), compare(max, Hi, Val, H1),
      N = L + 1,
      get_bounds(S, N, H, L1, H1, Low, High).

   calc_bounds(low, "add", L, Lo, _, Lval) :- Lval = L + Lo.

   calc_bounds(low, "minus", L, _, Hi, Lval) :- Lval = L - Hi.

   calc_bounds(low, "times", L, Lo, _, Lval) :- Lval = L * Lo.

   calc_bounds(low, "div", L, _, Hi, Lval) :- Hi <> 0, Lval = L div Hi.

   calc_bounds(high, "add", H, _, Hi, Hval) :- Hval = H + Hi.

   calc_bounds(high, "minus", H, Lo, _, Hval) :- Hval = H - Lo.

   calc_bounds(high, "times", H, _, Hi, Hval) :- Hval = H * Hi.

   calc_bounds(high, "div", H, Lo, _, Hval) :- Lo <> 0, Hval = H div Lo.

   calc_bounds(_, _, Val, _, _, Val).

/* The valid operators of an expression */

   opsign("+", add).
   opsign("-", minus).
   opsign("*", times).
   opsign("/", div).
   opsign("add", add).
   opsign("minus", minus).
   opsign("times", times).
   opsign("div", div).
   opsign("q", "Question").
   opsign("m", "Menu").

/* Allow the definition of a module */

   proc(Name) :-
      cursor(Row, Col),
      write("\n Do you want to define ", Name, " as a\n 1) Rule\n 2) Menu"),
      write("\n 3) Question\n    Option : "),
      cursor(R, C), get_int(R, C, I),
      I >= 1, I <= 3,
      cursor(Row, Col), nl, nl, nl,
      nl, nl, nl, cursor(Row, Col),
      !, proc1(I, I, Name).

   proc1(1, 1, Name) :-
      write("\n Defining ", Name, " as a Rule :\n Rule ", Name, " = "),
```

```
    cursor(R, C), get_string(R, C, S), token_rule(1, S, [], Rulelist),
    write("\n How many class ratings will there be ? "),
    cursor(R1, C1), get_int(R1, C1, Num),
    Num >= 1, Num <= 4,
    write("\n Class     From  -  To     Rating\n"),
    class_ratings(1, Num, [], List, 0, 0, 0, 0, X1, X2, X3, X4),
    write("\nDo you want to save this rule (y/n) ? "),
    readchar(Ch), write(Ch),
    proceed(Ch, 1, Name, Rulelist),
    save_rule(Ch, Name, X1, X2, X3, X4, Rulelist, List).

proc1(2, 2, Name) :-
    write("\n Defining ", Name, " as a Menu :\n"),
    write(" How many menu options are there for Menu ", Name, " ? "),
    cursor(R, C), get_int(R, C, Num),
    Num >= 1, Num <= 5,
    proc1(4, Num, Name).

proc1(3, 3, Name) :-
    write("\n Defining ", Name, " as a Question :\n"),
    write(" Enter Question ", Name, " : "),
    cursor(R1, C1), get_string(R1, C1, S1),
    write("\n Enter the range of valid response (eg 1 - 5) : "),
    cursor(R2, C2), get_range(R2, C2, Low, Hi),
    concat(S1, " ", Str),
    write("\n Do you want to save the question (y/n) ? "),
    readchar(Yesno), write(Yesno),
    save_quest(Yesno, Name, Str, Low, Hi).

proc1(4, Num, Name) :-
    write(" Enter the introductory line for menu : "),
    cursor(R, C), get_string(R, C, Str1),
    write("        Menu-Option                          Rating\n"),
    menu_options([], List, 1, Num, Str1, Str2),
    write("\n    Enter the prompt line : "),
    cursor(R1, C1), get_string(R1, C1, Str4),
    concat(Str2, "\n ", Str3),
    concat(Str3, Str4, Str5),
    concat(Str5, " ", Str6),
    write("\n Do you want to save this menu (y/n) ? "),
    readchar(Ch), write(Ch),
    to_save(Ch, Num, List, Name, Str6).

/* Allow user to choose between saving or not */

proceed('y', N, Name, List) :-
    query_update(N, Name, List),
    check_existence(1, Name, List).

proceed(_, _, _, _).

/* Ask user to specify modules contained in a rule which is
   expressed as a list of symbols -> use depth-first search */

query_update(_, _, []).

query_update(N, Name, [_ | Rest]) :-
    N mod 2 = 0,
    N1 = N + 1,
    query_update(N1, Name, Rest).
```

```
   query_update(N, Name, [Part | Rest]) :-
      N mod 2 = 1,
      M = (N + 1) div 2,
      not(db(to_ask(Part, _, _, _, _))),
      clearwindow,
      write("\n Please enter information of module ", M),
      write(" : ", Part, " of Rule ", Name),
      proc(Part), N1 = N + 1,
      query_update(N1, Name, Rest).

/* Module already existing, don't ask and skip to the next one */
   query_update(N, Name, [_ | Rest]) :-
      N1 = N + 1,
      query_update(N1, Name, Rest).

/* See which of the modules in a rule has not been defined */

   check_existence(_, _, []) :- !.

   check_existence(N, Rule, [First | Rest]) :-
      N mod 2 = 1,
      db(to_ask(First, _, _, _, _)), !,
      N1 = N + 1,
      check_existence(N1, Rule, Rest).

   check_existence(N, Rule, [First | Rest]) :-
      N mod 2 = 1,
      clearwindow,
      write("\n Module ", First, " of Rule "),
      write(Rule, " has not been defined\n"),
      proc(First), !,
      N1 = N + 1,
      check_existence(N1, Rule, Rest).

   check_existence(N, Rule, [_ | Rest]) :-
      N1 = N + 1,
      check_existence(N1, Rule, Rest).

/* Allow the definition of a menu */

   menu_options(L, L, N, Num, S, S) :-
      N > Num, !.   /* terminating condition */

   menu_options(List1, List2, N, Num, S, Str) :-
      write("   ", N, ") "),
      cursor(R, C), get_string(R, C, L),
      cursor(R, 41), get_int(R, 41, I),
      str_int(Sint, N),
      concat(Sint, ") ", T1), /* concatenate every thing together */
      concat(T1, L, T2),      /* into a single string             */
      concat("\n ", T2, T3),
      concat(S, T3, LS),
      N1 = N + 1,
      menu_options([I | List1], List2, N1, Num, LS, Str).

/* Comparison functions */

   compare(min, X, Y, X) :-
      X <= Y, !.
```

```
   compare(min, _, Y, Y).

   compare(max, X, Y, X) :- X >= Y, !.

   compare(max, _, Y, Y).

/* Allow specification of the class ratings for a rule */

   class_ratings(N, Num, L, L, W, X, Y, Z, W, X, Y, Z) :-
      N > Num, !.

   class_ratings(1, Num, L, List, _, _, _, _, W, X, Y, Z) :-
      write("   1) "), cursor(R, _),
      cursor(R, 12), get_int(R, 12, I1),
      cursor(R, 20), get_int(R, 20, I2),
      I1 <= I2,
      cursor(R, 29), get_int(R, 29, I3),
      class_ratings(2, Num, [class(I1, I2, I3) | L], List,
      I1, I2, I3, I3, W, X, Y, Z).

   class_ratings(N, Num, L, List, W1, X1, Y1, Z1, W, X, Y, Z) :-
      write("   ", N, ") "), cursor(Row, _),
      cursor(Row, 12), get_int(Row, 12, I1),
      I1 = X1 + 1,
      cursor(Row, 20), get_int(Row, 20, I2),
      I1 <= I2,
      cursor(Row, 29), get_int(Row, 29, I3),
      N1 = N + 1,
      compare(min, Y1, I3, Lo), compare(max, Z1, I3, Hi),
      class_ratings(N1, Num, [class(I1, I2, I3) | L], List,
      W1, I2, Lo, Hi, W, X, Y, Z).

/* Discard previous definitions of a module */

   throw(S) :-
      retract(db(to_ask(S, _, _, _, _))), fail.

   throw(S) :-
      retract(db(ratings(S, _, _))), fail.

   throw(S) :-
      retract(db(c_rates(S, _, _, _))), fail.

   throw(S) :-
      retract(db(rules(S, _, _, _, _, _))), fail.

   throw(_).

/* Save a menu definition */

   to_save('y', Num, List, Name, Str) :-
      throw(Name),
      assert(db(to_ask(Name, 'm', 1, Num, Str))),
      save_ratings(Num, List, Name).

   to_save(_, _, _, _, _).

   save_ratings(0, _, _) :- !.
```

```
   save_ratings(N, [H | T], Name) :-
      asserta(db(ratings(Name, N, H))),
      N1 = N - 1,
      save_ratings(N1, T, Name).

/* Save a rule definition */

   save_rule('y', Name, LR, UR, Lo, Hi, Rulelist, List) :-
      throw(Name),
      assert(db(rules(Name, LR, UR, Lo, Hi, Rulelist))),
      save_class(Name, List).

   save_rule(_, _, _, _, _, _, _, _).

   save_class(_, []).

   save_class(Name, [class(Lo, Hi, X) | Rest]) :-
      asserta(db(c_rates(Name, Lo, Hi, X))),
      save_class(Name, Rest).

/* Save a question definition */

   save_quest('y', S, Str, Low, Hi) :-
      throw(S),
      assert(db(to_ask(S, 'q', Low, Hi, Str))).

   save_quest(_, _, _, _, _).

/* A simple parser to parse an expression, break it down into tokens,
   and store them in a list */

   token_rule(N, Str, L, List) :-
      N mod 2 = 1,
      fronttoken(Str, Token, Rest),
      !, isname(Token),
      N1 = N + 1,
      token_rule(N1, Rest, [Token | L], List).

   token_rule(N, Str, L, List) :-
      N mod 2 = 0,
      fronttoken(Str, Token, Rest),
      !, opsign(Token, Sign),
      N1 = N + 1,
      token_rule(N1, Rest, [Sign | L], List).

   token_rule(N, _, L, List) :-
      N mod 2 = 0,
      reverse(L, [], List), !.

/* Reverse a list of symbols */

   reverse([], L, L).

   reverse([Head | Rest], Temp, List) :-
      reverse(Rest, [Head | Temp], List).

/* Get the range specification for a question */

   get_range(R, C, Low, Hi) :-
      get_string(R, C, S),
```

```
      fronttoken(S, Token, Rest),
      str_int(Token, Low),
      fronttoken(Rest, "-", Rest2),
      fronttoken(Rest2, Token2, _),
      str_int(Token2, Hi).

   get_range(R, C, Low, Hi) :-
      cursor(R, C), nl, cursor(R, C), !, get_range(R, C, Low, Hi).

/* Get input of string and integer types */

   get_string(_, _, S) :-
      readln(S).

   get_string(R, C, S) :-
      cursor(R, C), nl, cursor(R, C),  /* clear that line */
      !, get_string(R, C, S).

   get_int(_, _, I) :-
      readint(I).

   get_int(R, C, I) :-
      cursor(R, C), nl, cursor(R, C),  /* clear that line */
```

```
clauses

/* Inference Engine for the Heuristics Rules
   Backward chaining is employed */

   bchain([Rule], K, _, Rating) :-
      K mod 2 = 1,
      db(rules(Rule, _, _, _, _, List)),
      bchain(List, 1, 0, Temp),
      db(c_rates(Rule, Low, Hi, Rating)),
      Temp >= Low, Temp <= Hi.

   bchain([Head | Rest], K, N, Sum) :-
      not(Rest = []),
      bchain([Head], K, N, N1),
      K1 = K + 1, !,
      bchain(Rest, K1, N1, Sum).

   bchain([Subgoal], K, Sofar, Sofar) :-
      K mod 2 = 0,
      asserta(maths(Subgoal)).

   bchain([Subgoal], K, Sofar, Result) :-
      K mod 2 = 1,
      evaluate(Subgoal, N),
      maths(What),
      retract(maths(_)),
      resultant(What, Sofar, N, Result).

/* A simple expression evaluator */

   resultant("add", Sofar, N, Result) :-
      Result = Sofar + N.

   resultant("minus", Sofar, N, Result) :-
      Result = Sofar - N.

   resultant("times", Sofar, N, Result) :-
      Result = Sofar * N.

   resultant("div", Sofar, N, Result) :-
     Result = Sofar div N.

/* Evaluate a rule, a menu or a question */

   evaluate(A, Rating) :-
      db(to_ask(A, Ch, Low, Hi, Quest)),
      response(Quest, Answer, Row, Col), !,
      eval(Ch, A, Answer, Low, Hi, Row, Col, Rating).

/* Getting response from the user */

   response(Quest, Answer, R, C) :-
      nl, write(Quest),
      cursor(R, C),
      R1 = R + 5, C1 = C + 3,
      makewindow(4, 7, 0, "", R1, C1, 1, 3),
      readln(Answer), !, nl.

   response(_, "esc", 0, 0) :- removewindow.
```

```
/* Checking for validity of the user response */

   eval(_, _, "esc", _, _, _, _, _) :- !, fail.

   eval('m', Amod, Str, Low, Hi, R, C, Rating) :-
      str_int(Str, I),
      I >= Low, I <= Hi,
      db(ratings(Amod, I, Rating)), !,
      shiftwindow(2),
      clearwindow, shiftwindow(4),
      removewindow,
      cursor(R, C), write(Str), nl.

   eval('q', _, Str, Low, Hi, R, C, Rating) :-
      str_int(Str, Rating),
      Rating >= Low, Rating <= Hi, !,
      shiftwindow(2),
      clearwindow, shiftwindow(4),
      removewindow,
      cursor(R, C), write(Str), nl.

   eval(Ch, A, _, Low, Hi, R, C, Rating) :-
      shiftwindow(2), clearwindow,
      cursor(1, 1),
      write("Enter input in the range ", Low, " to ", Hi),
      shiftwindow(4),
      reentry(S), !,
      eval(Ch, A, S, Low, Hi, R, C, Rating).
```

Appendix D : Instructions on how to Install LOANEX

You need an IBM PC or a compatible machine with a least 640K RAM. The DOS (operating system) used should be at least of version 2.0 or later. Insert the diskette provided into the default drive and enter "prolog". This should start up the Turbo Prolog system. Press the space bar according to the prompt on the screen once the system is loaded into the computer's memory. You should see 4 windows on the screen. Press the "f" key to select the File pull-down menu and then press the "l" key to loading a file. Enter the name "loanex" to load the LOANEX system into the work space. The main program of LOANEX (ie loanex.pro) will appear in the editor window. Press the "r" key to run this program. Once this is done, sit back and wait awhile for the program to compile and execute. If there is no technical problem, you should be able to use the LOANEX system within a minute or so.

If an error message like "insufficient space for the heap" or "insufficient space for the stack" appears in the message window, then refer to any DOS manual for ways of reducing the operating system's demand for memory so that LOANEX can have more to run on.
