A QUIZ APPRENTICE

by

Claude Queant

A thesis
presented to the University of Ottawa
in partial fulfillment of the requirements for
the Degree of
Master of Computer Science (M.C.S.)

University of Ottawa
Ottawa, Ontario, 1986

ISBN 0-315-36536-6

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Abstract

Given knowledge of basic programming techniques and the ability to assimilate application domain concepts, a programmer's apprentice is able to understand a user's program on a very fundamental level and therefore cooperate with him effectively in the design, implementation, and maintenance of the program.

Starting from the general idea of an apprentice, the object of this thesis is to build a Quiz Apprentice. Quiz is a report writer, that is a tool for database users facilitating the production of reports, labels, form letters, etc.

The first step of the work consists of defining the role and possibilities of the Quiz Apprentice. Its main features are the ability of checking a user's program, debugging the user's code, and explaining any intervention made to correct the program. The second step is the development of a knowledge acquisition module. Given a problem-solution pair, this module generates a general rule which corresponds to the class of problem-solution pairs containing the given pair.

A prototype is built to prove the feasibility of the Quiz Apprentice. As this prototype is performing well, we think it may be a good starting point to build a real Quiz Apprentice and perhaps a more general apprentice.

This thesis is dedicated to my wife Brigitte and my little girl Dorothy. I want to thank them for their support and patience.

Acknowledgements

First of all, I would like to thank my advisor, Stanislaw Matwin, for the support and advice that he has given me during the development and the writing of this thesis. Secondly, I would like to thank everyone in the "AI group" of the University of Ottawa for allowing me to participate in their work. Finally, I would like to thank the ENST and CIMSA-SINTRA for making my stay at the University of Ottawa possible.

Table of contents

Introduction

1.1 - Objectives

When it is not possible to construct a fully automatic system for a task, it is often possible to construct a system that can assist a user in the task. This approach defines a new class of knowledge-based consultants, referred to as **apprentices**. The goal of this thesis is to create such a system; one which can serve as an apprentice in the domain of programming in Quiz.

Suppose that a programmer wants to write a program in Quiz. His main goal is to have a Quiz program which satisfies certain specifications. The role of the Quiz Apprentice is to provide assistance to the programmer without preventing him from doing simple things in the ordinary way. In order to achieve its objectives, a Quiz Apprentice ought to meet the following minimum criteria :

- Ability to check and correct a user's program in order to produce a runnable Quiz program.

- Ability to understand the user's specifications, translate them, and integrate them into the current user program.

It is assumed that the user will not be able to delegate to the Quiz Apprentice all the work that needs to be done. The key issue is rather division of labor. The user will have to make the hard decisions about what should be done and the assistant can take over some of the mundane programming tasks.

A user must have the possibility of giving any specification he requires for the program he wants to write. It may happen, however, that the Quiz Apprentice is unable to solve a particular user problem. The Quiz Apprentice must then have the capacity to learn new knowledge as it needs to understand unknown user specifications. The second objective of this thesis is then to investigate alternative methods of learning for an Apprentice, based on a simple learning algorithm.

We believe that an intelligent apprentice has to meet the above specifications. However, meeting them is only a necessary, not a sufficient condition for the system to be considered intelligent.

There are certain constraints on reaching these objectives. The first of these has to do with the nature of the project. An apprentice is interactive by nature; the Quiz Apprentice should be able to answer the user and correct his program as fast as possible if it is to be called an interactive system. The system developed for this thesis is a prototype implemented in Prolog. Since Prolog is not designed for real time processing, we consider that an acceptable response time should to be less than fifteen seconds.

A second constraint is time; the Quiz Apprentice had to be implemented in the limited time given for a Master's thesis. Building a really strong Quiz Apprentice would take a much longer time. Strictly speaking, it is as yet impossible to finish such a system because Quiz is not even a completed language. We were then obliged to impose some restrictions on the Quiz Apprentice described in this thesis.

1.2 - General report

In this thesis, we demonstrate that the goals of the Quiz Apprentice are attainable. Though this Apprentice may not be considered an intelligent system, the work it performs provides excellent evidence that the thesis is sound. In fact, in our research, we have proven the feasibility of the Quiz Apprentice; now we have a strong base on which to build a real Quiz Apprentice.

The first objective of the Quiz Apprentice is to understand the user's specifications and to correct his program. We needed a representative set of user's requirements in order to verify that this goal has been reached. We used a representative sample of user's questions, built for the **Quiz Advisor** project.

A certain number of the questions used in that research were directly rejected for this project because their subject was not Quiz. An example of this type of question would be :

Is there a default limit of output to disk ?

The remaining questions, however, were well suited for our purpose. Of these, the Quiz Apprentice is unable to answer some. These are questions referring to side-effects in Quiz, keywords unknown by the Apprentice, formats and types, and some questions are simply too difficult to answer. However, the questions to which the Quiz Apprentice has an acceptable answer -- to which it responds with a Quiz program which respects the user's requirements -- represent 70% of the usable set of user questions.

The constraints governing the construction of the Quiz Apprentice have been respected and the resulting Apprentice is efficient; the action which requires the longest processing time is the correction of the program. Even so, every correction that has to be done on the program takes less than ten seconds. Despite the simplicity of the Quiz Apprentice, the work developed for this part of the thesis finds its justification in this result.

The second objective of this thesis was to investigate alternative methods of learning for the Quiz Apprentice. The goal was for the system to have the ability to learn everything generated in precedent steps. The type of learning we chose to build into the Apprentice is that of learning by example, and the result of the research is the development of two simple learning algorithms. Though these algorithms may be considered elementary, they allow the Quiz Apprentice to learn about 95% of all the knowledge necessary. The work developed for this second aspect of the Quiz Apprentice, then, also finds its justification in the results obtained.

1.3 - Thesis overview

The presentation of this thesis is decomposed as follows :

Chapter 2

This chapter gives a presentation of the general problem of apprentices and learning apprentices. A review of the literature is made to place the Quiz Apprentice in the context of the current research.

Chapter 3

This chapter describes how the first objective of the Quiz Apprentice was reached. After a brief description of the difficulties encountered, some formalizations are made for building the Quiz Apprentice. These formalizations are useful for representing a user program and the knowledge needed to correct it. From these formalizations, a description of the implementation is given. Finally, some examples are given to explain the work performed by the Quiz Apprentice.

Chapter 4

This chapter describes the work done in developing the learning module. After a presentation of the purpose of this module, the knowledge required for the Quiz Apprentice is studied and two algorithms are built to generate this knowledge. The implementation of these algorithms is then described and some examples are given to illustrate the work of the learning module.

Chapter 5

This chapter gives an evaluation of the Quiz Apprentice. After an estimation of the advantages and the drawbacks of the system, we give a description of the future work needed to build a strong Quiz Apprentice.

The reader who is interested in more technical details may refer to the appendices. They contain :

Appendix A : Description of Quiz

Appendix B : The Apprentice Code

Appendix C : The Apprentice Knowledge Base

A brief survey of the related literature

2.1 - Introduction

The material discussed in this section is often in the form of direct quotes from [Mitchell et al 85] and [Rich, Schrobe 79]. In some cases, the usual quotation marks have been omitted for readability.

2.1.1 - The software crisis

Since the time of the first computers, hardware improvements have increased dramatically. To make computers easier to program, different tools, such as compilers, assemblers, and operating systems, have been developed. Despite these tools, however, modern computers have become impossibly complex. This complexity is not only due to the size of the programs, but also to the fact that the number of interactions between modules grow much more quickly than the size of the modules. This problem is especially great in the field of artificial intelligence; programs are simply too large to be improved in their present form.

Many ways have been investigated to overcome this crisis in software engineering. Some designers have started a major branch of program verification research using formal mathematics. The goal of this research is to develop logical systems in which desired properties can be proved as theorems. Others have tried to use the computer itself to help reduce the difficulty of programming. The design of new programming languages and language processors is currently a very active field.

The ultimate form of the latter solution is Automatic Programming. The goal of this research is to create a system which will automatically generate correct, reasonably efficient code; code which would satisfy given high-level, application-oriented specifications.

2.1.2 - Apprentice systems

Unfortunately, it does not appear that the goal of Automatic Programming will be reached in the near future. A new approach to the software problem has recently been developed, however, which lies between language-oriented programming tools and automatic programming. This approach defines a new class of knowledge-based consultants, referred to as **apprentices**.

Given knowledge of basic programming techniques and the ability to assimilate application domain concepts, an apprentice is a computer system able to understand a user's program on a very fundamental level. It can therefore cooperate more effectively with the programmer in the design, implementation and maintenance of a program. An apprentice need not be capable of programming by itself, but can aid the expert programmer by checking his work in various ways [Rich, Schrobe 79].

The major theoretical problem in building an apprentice is that of developing a computer representation of programs and a knowledge base about programs which can facilitate smooth and natural interaction between the programmer and the apprentice.

An apprentice needs an understanding of the user program at different levels of abstraction. The lower level

contains a program written in the target language. This level is necessary for communication between the programmer and the apprentice. The higher level contains the logical structure of the program; with this tool, the apprentice can understand how and why the program works. This representation of the program is in fact a formal representation of the user's specifications. For the apprentice to understand a given program, it must connect all the levels of descriptions of the program.

Besides the usual knowledge required for language-oriented programming tools, an apprentice needs a major new kind of knowledge about the subject programs. Part of this knowledge is the same as that required for automatic programming, that is an understanding of the application domain and the way that parts of a program existing in the computer memory relate to concrete objects and operations in the real world. The other part of this knowledge is largely independent of the application domain. This consists mainly of the basic algorithms and data structuring techniques of programming.

2.1.3 - Learning Apprentice systems

It is by now well-recognized that a major impediment to developing knowledge-based systems is the knowledge acquisition bottleneck, the task of building up a complete enough and correct enough knowledge base to provide high-level performance. In an effort to reduce the cost and increase the level of performance of current knowledge-based systems, semi-automated tools have been developed for aiding in the knowledge acquisition process.

A new class of knowledge-based consultant systems has been proposed. These systems incorporate recently developed machine learning methods into their design in order to overcome the knowledge acquisition bottleneck and automate the acquisition of new rules. They are known as **Learning Apprentice systems**, and are defined as the class of interactive knowledge-based consultants that directly assimilate new knowledge. They learn by observing and analyzing the problem solving steps contributed by their users in normal use of the system [Mitchell et al 85].

In task domains for which Learning Apprentice systems are feasible, they will offer strong advantages over present architectures for knowledge based systems. A Learning Apprentice system distributed to a broad community of users could acquire a base of problem-solving experience much larger than that from which a human expert learns. Such a large experience base would offer the potential for acquiring a very strong knowledge base, provided effective learning methods could be developed.

2.2 - Review of the literature

The Apprentice systems represent a class of knowledge-based consultants which is studied with increasing interest in artificial intelligence research. Current research in machine learning is also very active. There are many systems which belong to these categories. Three projects which may be considered representative of the current research are The Programmer's Apprentice [Rich, Schrobe 79] [Waters 82], KLAUS [Haas, Hendrix 83] and LEAP [Mitchell et. al. 85]. We will also glance at very recent type of learning research which involves new kinds of rules: augmented rules and censors [Winston 86].

2.2.1 - The Programmer's Apprentice

The Programmer's Apprentice project uses the domain of programming as a vehicle for studying human problem solving behavior. Recognizing that it will be a long time before it is possible to fully duplicate an expert programmer's abilities, the project seeks to develop an intelligent system, the Programmer's Apprentice, which will help a programmer in various phases of the programming task.

The Programmer's Apprentice is considered a new, active agent in the programming environment. Some remarks about this new agent seem to be in order. First of all, it is not intended to replace the programming environment, but rather to augment it. The programmer can still communicate directly with the rest of the environment. He has always a "trap door", so to speak, so that he is not required to work exclusively through the Programmer's Apprentice.

More important is the philosophy behind the concept of the Programmer's Apprentice, the idea that there is a fundamental division of labour between the programmer and the Apprentice. In theory, the Programmer's Apprentice serves as a junior partner and critic, keeping track of details and assisting in the documentation, verification, debugging and modification of the program. The programmer does the really difficult tasks of design and implementation.

The main features for the design of the Programmer's Apprentice have been described in [Rich, Schrobe 78]. This initial Programmer's Apprentice proposal focuses on describing the basic AI ideas behind the system and explaining why they provide important leverage for various programming tasks. This proposal is weak, however, in specifics. Its

lack of specificity is probably due to the fact that, at the time the proposal was written, few attempts had been made to actually implement the Programmer's Apprentice.

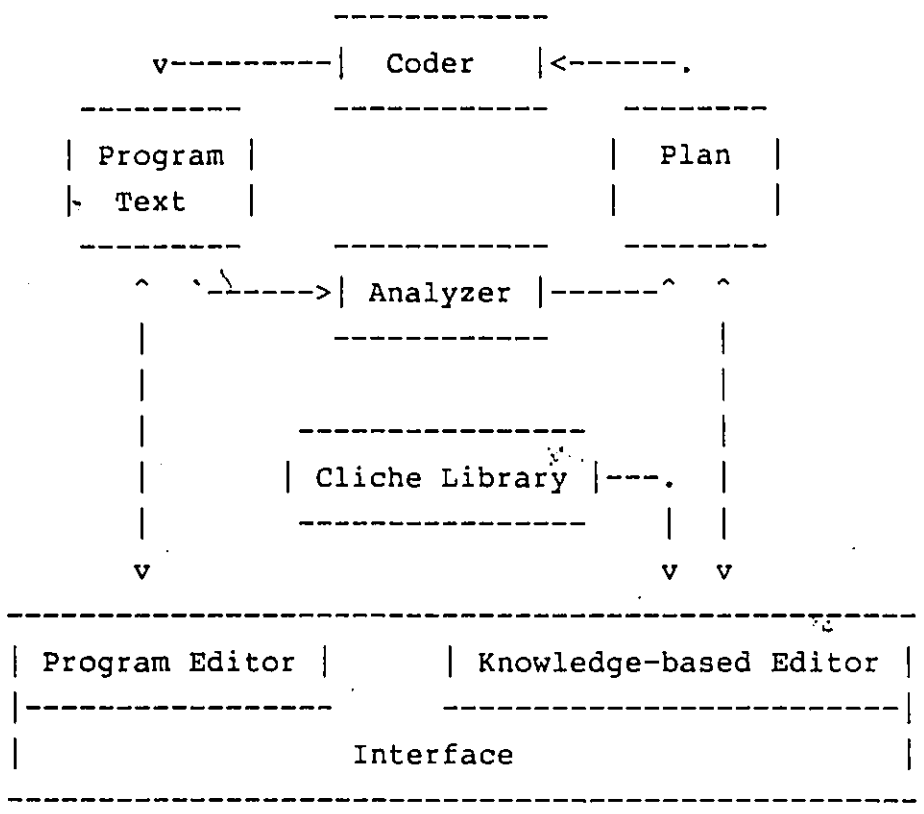

Fig 2.1 : The architecture of KBEmacs

An initial implementation of the Programmer's Apprentice has been done since the proposal. The Knowledge-Based Editor in Emacs, KBEmacs [Waters 86], is the culmination of a long effort to produce a running system which would exhibit some of the capabilities of the Programmer's Apprentice. KBEmacs is capable of acting as a semi-expert assistant to a person who is writing a program, even of taking over some parts of the programming task. The

architecture chosen for KBEmacs is shown in figure 2.1.

The user may choose between two editors to build his program. Depending on the editor, the program is translated into either a low level description such as program text, or a high level description such as a plan. The Analyzer and the Coder then translate the user program so that KBEmacs can understand it at every level of description. The library contains cliches which will be described a little farther on.

Three basic ideas define the approach taken and the capabilities of the KBEmacs system. The first idea is the **Assistant Approach.** Though presumably less knowledgeable than the user, the Assistant interacts with the tools in the environment in the same way the programmer does and is capable of helping the programmer do his job. The programmer will to make the hard decisions about what should be done and which algorithms should be used. However, much of the programming is mundane and can easily be done by an assistant. The keys to cooperation between the programmer and the assistant are effective communication and shared knowledge. KBEmacs does not necessarily make poor programmers better; its goal is to make expert programmers super-productive.

The second idea is the **Cliche.** A cliche consists of a set of roles embedded in an underlying matrix. Roles within a matrix may be thought of as slots in a frame. The matrix specifies how the roles interact in order to achieve the goal of the cliche as a whole.

Let us consider a cliche called "simple_report". The purpose of this cliche is to enumerate a sequence of items and print them out. The roles of this cliche are the "title" to be printed on a title page, the "enumerator" that

enumerates some sequences of items, the "print_item" to print out the information about each of the enumerated items and the "summary" to print out some summary information at the end of the report. The matrix of this cliche specifies different kinds of information. It specifies pieces of fixed computation, as, for instance, how to print out a title page. It also specifies the data flow and control flow that connect the roles. For example, the data flow will connect the output of the "enumerator" with the input of the "print_item". Finally, the matrix specifies various constraints on the roles. For example, the "print_item" is constrained to contain a computation appropriate for printing out the type of item. An essential part of the cliche concept is reusability. Once something has been thought out and given a name, it can be reused as a component in future thinking.

The third idea in the KBEmacs system is the **plan concept**. The plan formalism used in the system is designed to represent two basic kinds of information: the structure of a particular program and knowledge about cliches. This formalism includes not only the data flow and control flow relationships between parts of the program, but also goal-subgoal, prerequisites and other logical dependencies. For example, Fig 2.2 shows a diagram for a simple plan, the plan for the cliche "absolute_value".

The arrows "--->" represent the data flow and the arrows ">>>>" the control flow. A plan is composed of basic units of computation called segments. A segment can either correspond to a primitive computation, such as "-", or contain a subplan that describes the computation performed by the segment, as for example the segment "abs" in Fig 2.2.

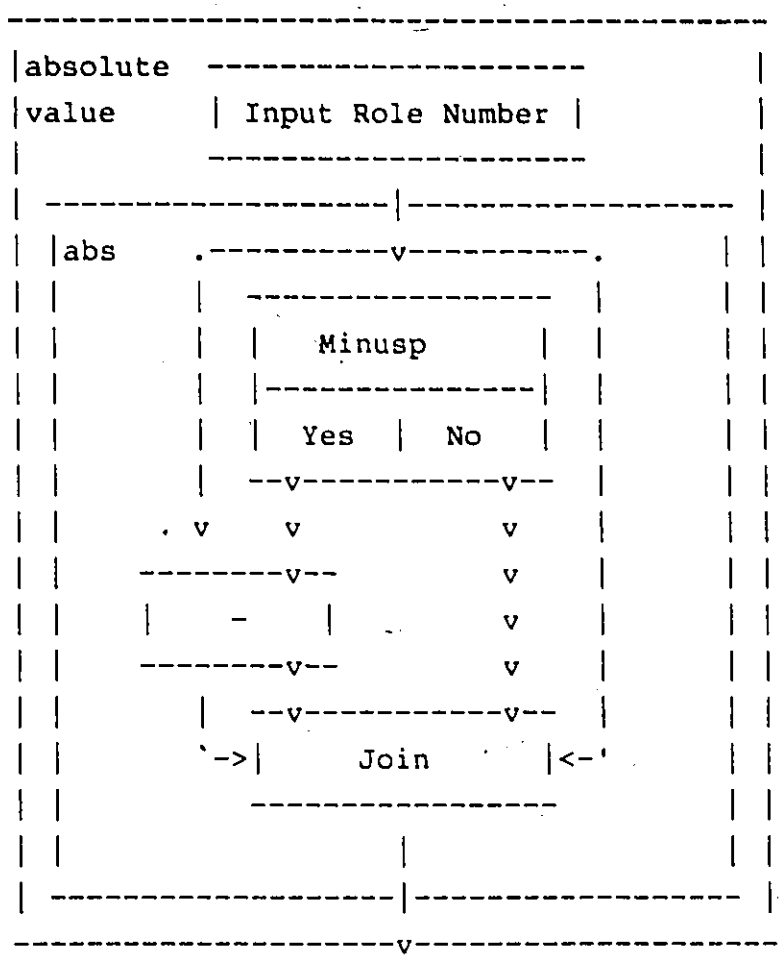

Fig 2.2 : A plan for the cliche "absolute value"

A key feature of the plan formalism is that it abstracts away from the syntactic features of programming languages and directly represents the semantic features of a program. For instance, a plan does not represent data flow in terms of the way it could be implemented in any particular language; it keeps only the features that are essential to the actual algorithm.

KBEmacs is a research experiment. The three ideas which led to KBEmacs are essentially the source of all of its power. KBEmacs has been tested on Lisp programs and ADA

programs. Its power can be illustrated by the order of magnitude between the size of the set of commands and the size of the program. But this system is still incomplete. It handles only 50% of ADA. It is also very slow; the processing time of a knowledge-based operation can take more than 5 minutes. As a result, no attempt will be made to turn KBEmacs in a practical tool. The project will continue to focus on the fundamental research issues associated with the Programmer's Apprentice.

2.2.2 - KLAUS

KLAUS (Knowledge Learning And Using Systems) is not, strictly speaking, an apprentice in the sense introduced above. It is, however, the first system in which learning is based upon the structure of a certain environment. In that respect, KLAUS is relevant to the topic of this thesis.

The core idea of a KLAUS system is that of a machine which can hold a conversation with a user in English about his specific domain of interest, subsequently retrieve and display information conveyed by the user, and apply various types of external software systems to solve user problems. Such software would include data base management systems, report generators, planners, simulators and statistical packages.

Because a user may provide new knowledge in an incremental and incomplete manner, a KLAUS system must keep track of what it has already been told so that it can deduce the existence of missing information and explicitly ask the user to supply it. Moreover, a KLAUS system must carefully distinguish what it does not know from what it knows to be false. This means that KLAUS systems do not make the "closed

world" assumption which states that if something is not said to be true, then it is considered false. A KLAUS system must also have the capacity to learn structured knowledge about concepts and make deductions based on its knowledge from past and present learning experiences. A KLAUS system must be able to learn new concepts, and at the same time the linguistic constructions used to express them. It must acquire domain-specific language expertise, not only to understand natural language statements formulated by the user, but also to generate natural language response to the user request.

A pilot KLAUS called NANOKLAUS has already been developed and tested. The principal components of NANOKLAUS are a natural language processing module, a formal deduction module, and a number of support procedures. The formal deduction module operates on a data-base of well-formed formulas in first order logic. The support procedures associated with the system aid in assimilating knowledge about new subject domains and in maintaining the data base. NANOKLAUS essentially creates a type hierarchy in a domain. It acquires structural knowledge about a domain, establishes subsumption relationships between types and provides objects with attributes of types already contained in the knowledge base.

As an example, consider this imaginary session with NANOKLAUS. It already knows that a carrier is a ship. It is possible to carry on the following dialogue with it, quoted from [Haas, Hendrix 83]:

user : A Kittyhawk is a carrier.
NK : I understand Kittyhawk.
[The user introduces a new object and NANOKLAUS establishes the relationship with the already existing type]

user : A Forrestal is a carrier.
NK : You're saying that anything that is a Forrestal is also a carrier.
Is Kittyhawk a proper subclass of Forrestal ?
user : No.
NK : Is Forrestal necessarily composed of entirely different members from Kittyhawk ?
user : Yes.
NK : Do Kittyhawk and Forrestal span the set of all carriers ?
user : No.
NK : Ok, I understand Forrestal.
[When a new subset of an existing set is created, NANOKLAUS queries the user to determine how it relates to that set, as well as to other previously introduced subsets of that set]

user : The JFK is a Kittyhawk.
NK : I understand JFK.
[A new object is entered]

user : The JFK is a Forrestal.
NK : Sorry, JFK is an individual of the type Kittyhawk and it is in conflict with the type Forrestal.
[NANOKLAUS used its knowledge of subset relationships to discover and then report an inconsistency in its user's assertions]

This dialogue shows how NANOKLAUS works with relationships between types and objects, but it does not show its capability of learning linguistic constructions.

NANOKLAUS unfortunately does not reach beyond this structural level of knowledge representation. For instance, it cannot give meaning to concepts by expressing them as

results of generic operations on other concepts already known to the system. Many other major limitations may be found in NANOKLAUS. For the most part, it is limited to learning about very concrete types of objects and their interrelations. And even in the domain of concrete objects, the system has no capacity to deal with other types of learning. Learning by analogy, for instance, is beyond its capabilities. NANOKLAUS is best described as a fragile, proof-of-concept system that was built to establish the feasibility of achieving the broader KLAUS goals.

A more sophisticated KLAUS, called MICROKLAUS, is now being implemented. It will cover a broader range of English constructions and will feature a more efficient deduction system. This project is not limited to the study of knowledge acquisition, either. It is rather a step towards the main goals of the KLAUS project, which are to provide technology for a system that combines a knowledge of how to use various software packages with an ability to learn facts about new domains.

2.2.3 - LEAP

LEAP [Mitchell et. al. 85] is a LEarning APprentice system currently being constructed as an augmentation to a knowledge-based VLSI design assistant called VEXED. VEXED provides interactive aid to the user in implementing a circuit. Given the functional specifications of the circuit, VEXED is capable of suggesting and carrying out possible refinements to the design. A large part of its knowledge about circuit design is composed of a set of implementation rules, each of which suggests some legal method for refining a special function. The other part of its knowledge base is composed of a set of control rules, each of which selects

the preferred implementation from among several workable options. This division of rule types is important because it eliminates the need to adjust existing implementation rules when a new one is added.

A fundamental feature of LEAP is that it embeds a learning component within an interactive problem-solving consultant. This allows it to collect training examples that are closely suited for the refinement of its rule base. The user needs to intervene in problem solving only when the system is missing knowledge relevant to the task at hand; the resulting training examples therefore focus specifically on this missing knowledge.

The first of the two parts of LEAP's knowledge base has the convenient property of logically independent rules; when one adds a new implementation rule characterizing a new implementation method, the correctness of the existing implementation rules is not affected in any way. To date, LEAP only considers the learning of this type of knowledge, because the complexity of explaining training examples for control rules is too great for the system to handle.

LEAP's learning component tries to make the system understand a solution to a problem by verifying that the solution correctly solves the problem. This implies that verification is done by rewriting the solution into a form involving only concepts already understood by the system.

The first step in the acquisition of a new circuit implementation rule is the generalization of the preconditions of the rule. This procedure transforms the preconditions of the rule so that they characterize the general class of input signals for the given circuit.

A further step is to generalize the user's circuit as well as the function it implements. To generalize, LEAP first verifies for itself that the circuit works for the example. It then generalizes from this example by retaining only those features of the signals that were mentioned in the verification. This set of signal features characterizes the class of input signals which will generate the desired function of the circuit. Then LEAP produces a corresponding generalization of the original functional specification. LEAP's explain-then-generalize method, based on having an initial domain theory for constructing the explanation of the example, allows LEAP to produce justifiable generalizations from single training examples.

This system limits "new" problems to such questions as can be expressed using functions and relations already understood by the system. LEAP can not cope with problems which belong to an entirely new class. Another limitation of this system is that it must be provided with a set of powerful rewriting rules. Such rules must be acquired from experts; this type of acquisition may be much more difficult to implement in a less structured domain than VLSI design.

2.2.4 - Learning by augmenting rules

This work is essentially based on the Macbeth system [Winston 84] which concentrates on analogy in rule acquisition. The theory developed for this work addresses the analogy process at work when we exploit past experience in fields such as management, political science, economics, medicine, and everyday life.

The first step of the work was to remark that rules generated by classic learning systems can be misapplied. For

instance, if we want a system to learn the story of Macbeth, it will first generate this type of rule:

IF there is a greedy lady
and there is a weak noble
and the noble is married to the lady
THEN the noble may want to be king

The rule is ready to be applied to another situation. Suppose we have a similar situation with a man and a lady and we know that the man does not like the lady. The situation is in fact different because we know that it is difficult for a person to influence someone who does not like him or her. The problem with the first rule is that it implies more than is written. Nothing is indicated about how causal relations tie everything together.

The extension to the existing theory is based on two ideas, the **blocking principle** and the **prima facie conjecture**. The blocking principle works in this way:

> Suppose that we have a rule, derived from a precedent, which seems to apply to a problem. We first consider all the relations in that part of the precedent's causal structure involved in forming the rule. If any such relation corresponds to a relation that is either false or manifestly implausible in the problem situation, then the rule based on the precedent does not apply.

The prima facie conjecture suggests that:

> A relation is manifestly implausible if its negation can be shown by a direct, one step inference from relations already in place.

Several solutions apply the blocking principle on existing rules. The one we have as representative adds an "unless" part to a normal "if-then" rule. This addition means that each rule is augmented at the time it is generated with entries that correspond to negations of all the relations in the transferred causal structure. Those relations that become the "if" and "then" parts of the rule are of course excepted from this procedure. The augmented entries constitute the "unless" part of the rule. According to the blocking principle, if any entry in the "unless" part of the rule corresponds to something that is manifestly true, then the rule does not apply.

Let us go back to the story of Macbeth as an example. The rule shown above may be written now:

IF there is a greedy lady
and there is a weak noble
and the noble is married to lady
THEN the noble may want to be the king
UNLESS the lady does not persuade the noble to want to be the king
or the lady is not able to influence the noble.

The blocking rule may be written:

IF there is a person X who does not like a person Y
THEN Y is not able to influence X

A rule becomes a censor when it blocks the application of another rule. Because censors look just like any other rules, they can be learned, stored, and retrieved in the same way. We can also find cases where censors block other censors. For instance, the last rule may be written:

IF there is a person X who does not like a person Y
THEN Y is not able to influence X
UNLESS X trusts Y

and the censor of that censor may be written:

IF there is a person X who is able to convince another person Y
THEN Y trusts X

Since the censors are just rules used in a special way, they can be learned like any other rule. This may be done by direct telling, or by precedent. But the nicer way is by near miss. To illustrate this type of learning, we give the following example:

A teacher tells the system that a bachelor is an unmarried, adult man. The system will produce an impoverished definition of bachelor:

IF there is a man
and the man is not married
and the man is an adult
THEN the man is a bachelor

The teacher complains when the system identifies a Catholic priest as a bachelor. The system notices that the only difference between the priest and other people who are correctly identified as bachelors is that the priest is not able to be married. The system guesses that bachelors must be able to be married. It then modifies the existing rule and adds a censor:

IF there is a man
and the man is not married
and the man is an adult
THEN the man is a bachelor
UNLESS the man is not able to be married

IF the man is a priest
THEN the man is not able to be married

This is a particularly simple situation. It is not known how difficult it would be to identify the right difference in general. The only clear result given by the authors of this theory is that if it is difficult to identify the right difference, a learning system should simply give up, waiting for more transparent examples to come along.

The number of censors developed for the system is sufficient to do more than surface-scratching experiments, however. The work is still in progress and the authors claim that the extended theory improves performance in those domains subject to problem solving by analogy. The long range goal lies further toward learning by discovery.

It must be noted, however, that learning by exploiting causal links between the objects involved in a certain process seems very useful to the kind of application of learning we are pursuing in this thesis. It may potentially provide a much more powerful generalization method than the one we use in chapter 4.

The Quiz Apprentice

3.1 - An overview of Quiz

The Apprentice constructed during research for this thesis has been especially designed to deal with Quiz programs. An overview of Quiz is presented here in order to review the basic notions of this language; this will help the reader to understand some features of the Apprentice before going on to finer details. Those interested in more details about Quiz may refer to Appendix A, where a more complete description of the language is given.

3.1.1 - Presentation

Developed by Cognos Inc., Quiz is a report writer which, at its more advanced levels, can produce reports that involve difficult manipulations of data and satisfy some complex processing requirements.

In recent years, Quiz has been incorporated into Powerhouse, Cognos's most widely installed fourth generation application development language for minicomputers.

3.1.2 - Main statements

A Quiz program consists of a set of statements that represent a specification of the desired report. Here is a description of the main statements:

Access Each program starts with an **access** statement that names the file(s) to be used, and states where the data to be reported comes from. The **access** statement corresponds to the join operator introduced in the relational database theory.

Select The **select** statement specifies the data selection criterion. It corresponds to the select operator introduced in the relational database theory.

Sort The **sort** statement declares by which items the data must be sorted. The sort items are called keys.

Report The **report** statement defines what is to be reported. It corresponds to the project operator introduced in the relational database theory.

Go The **go** statement means that the report has been defined and, at the same time, starts the process which generates the report.

The user begins a Quiz program with an **access** statement and terminates it with a **go** statement. During the phase in which the user defines the specifications of his program, the declared statements are consulted by the parser and no other actions are performed. When the **go** statement is entered, the main processing cycle begins. This includes a data access process, a report process, and a delivery process. These operations refer to any Quiz statement without observing the order of the statements. This reflects the non-procedural nature of Quiz; the Quiz program represents a specification of the desired report.

The existence of the processing cycle implies the notion of **pass**. A **pass** represents the processing cycle that

treats a Quiz program. By extension, a pass represents any Quiz program beginning with an access statement and ending with a go statement.

Quiz has more than a dozen statement types, but the ones mentioned above are sufficient for the creation of a large number of useful reports.

3.1.3 - Examples

In our examples, we work with an imaginary database which describes a company that has offices in several cities. The first example reports the employee number, the lastname, and the city for every employee in the company. The second example reports the same information, but only for the employees working in Boston. This second report is sorted on the lastname of the employees.

```
Ex 1 :
access employees
report employee lastname city
go
```

```
Report 1 :
employee        lastname        city
0001            Smith           boston
0002            Anderson        boston
0003            Bone            toronto
0004            Auger           toronto
0005            Child           boston
0006            Rojas           seattle
```

Ex 2 :

```
access employees
select if city="boston"
sort on lastname
report employee lastname city
go
```

Report 2 :

employee	lastname	city
0002	Anderson	boston
0005	Child	boston
0001	Smith	boston

In the last example, we want a report containing the name of the salesmen represented by the value of the Quiz variable "salesman" and the total billing for each salesman. The items to be reported must be sorted on the total billing. But the total billing is not in any file. It has to be calculated from the billing in a previous pass and stored in a subfile. In the main program, an access to the subfile is performed and the report can be processed. The code then represents the two-pass program:

```
access salesmen
define total-billing numeric = billing
sort on salesman
report summary salesman total-billing subtotal
set subfile at salesman name temp
go

access temp link to salesmen
sort on total-billing
report count salesman total-billing
go
```

These programs give basic reports and may be easily written by standard users, but they are not entirely representative of the possibilities of Quiz.

3.2 - Definition of the Quiz Apprentice

3.2.1 - Definition

The Quiz Apprentice is an attempt to apply the concept of an apprentice system to the domain of programming in Quiz. As such, the Quiz Apprentice has to apply the general principles as described in section 2.1 to a knowledge base of Quiz.

We can therefore define the Quiz Apprentice as a computer system capable of understanding Quiz programs, i.e. of transforming them into an internal interpretable form, and of cooperating with the user in designing, implementing and maintaining Quiz programs.

3.2.2 - Goals

It is important to note that an apprentice need not be capable of programming by itself. It just has to understand the user's objectives in order to give him a runnable Quiz program. The standard user therefore needs some knowledge of Quiz to use the Quiz Apprentice.

A user's requirement is a sentence that represents user's intentions for solving problems using Quiz. A user's requirement may be a Quiz requirement or a non-Quiz requirement, but it must not be interpreted as a Quiz error since it may be translated into Quiz code.

The first obligation for a Quiz Apprentice is its acceptance of a Quiz program. In a more general way, the Apprentice has to understand user specifications. If a user's specification is a Quiz specification, the Apprentice remains inactive. In the contrary case, the Apprentice must notice the non-Quiz specification, warn the user, and translate it into Quiz code if the user desires it.

3.2.3 - Description of a session

In order to explain the concept of the Quiz Apprentice in more detail, we present an imaginary session which traces the design of a Quiz program. The only feature which is not part of this thesis is the use of free English dialogue.

We start in a state where the code already given by the user is the following:

```
access employees link to billings
report project lastname billing
```

and the dialogue begins:

```
user : I want to sort on project
app  : access employees link to billings
       sort on project
       report project lastname billing
   ok
```

[The Apprentice recognizes the part which is in Quiz code and inserts it in the program. The statements of the program are then displayed in the order the Quiz parser treats them]

user : Instead of report billing, I want
       report billing subtotal
app : access employees link to billings
       sort on project
       report project lastname billing subtotal
  I suppose you mean billing subtotal at project

[The Apprentice accepts the user requirement, but it finds an ambiguity. It then warns the user about it]

user : No, I want subtotal at employee
app : So, I have to change the sort statement
       access employees link to billings
       sort on project on employee
       report project lastname billing subtotal at employee
  ok

[The Apprentice finds that "employee" is not a key . It then adds "employee" into the sort statement to make it a key and notifies the user]

user : Now, I want to select if billing subtotal is bigger
       than 1000
app : It is impossible to select with an item which is not
       already calculated. You may calculate it in a first
       pass.
user : I calculated this item named billsum in the file temp
app : access employees link to billings link to temp
       select if billsum > 1000
       sort on project on employee
       report project lastname billsum
  ok

[The Apprentice finds that the user requirement is a non-Quiz requirement and it offers a solution to fix the code.

The user gives it some information and then the Apprentice rewrites the code according to the new information]

The session continues with the Apprentice looking over the user's shoulder.

3.3 - Formalization

Before trying to implement the Quiz Apprentice, it is necessary to do some formalizations to better understand the problem and to define a framework in which we will shape a solution to solve the problem of the Quiz Apprentice.

3.3.1 - Difficulties of formalizing

In order to have an accurate knowledge representation, we first have to formalize the knowledge of Quiz and of the user's requirements. As we will see, such knowledge belongs to real life and is therefore essentially ambiguous and vague.

3.3.1.1 - Difficulties due to Quiz

The first difficulty comes from the non-procedural nature of Quiz. This is a problem common to many fourth generation languages, and it will still remain if the Apprentice is generalized to accept a large set of fourth generation languages. Because of its non-procedural nature, it is impossible to take inspiration from the work already done on other apprentices. Some concepts, such as segments, have an entirely different meaning; others, such as control flow or data flow, become completely useless.

The second difficulty is due to the representation of a Quiz program. Some attributes may be omitted when writing a Quiz program, and the main processing cycle of a simple Quiz program therefore implies the setting of all uninstantiated control attributes. For instance, the report of a simple item implies the recognition of all attributes given by the user and possibly the use of some default values for other attributes.

The problem of uniqueness within the representation of a Quiz program poses another problem. The main statements given in section 3.1.2 must be unique if they are present in a Quiz program. Many other useful statements, however, may have several occurrences. The problem is to build a knowledge representation which respects the uniqueness of some statements and the non-uniqueness of others.

The last difficulties appear because the language is a very recent one. Quiz has been developed and updated several times over the last few years to answer the demand of Cognos's clients. At the present time, no conceptual model for the true relationships between objects in Quiz has been developed. The existing Quiz terminology fails to distinguish inconsistencies in terminology.

3.3.1.2 - Difficulties due to the user's requirements

A user's requirement is by nature vague and ambiguous. It is necessary to understand that requirement, that is, to translate it into an internally interpretable form, so that the Apprentice will be capable of answering the user. This general problem belongs to the set of problems in natural language processing, but, despite its interesting

complexities, it is not part of this project. It may be addressed in a future development of the Apprentice.

The difficulty of defining a user's requirement has been addressed in another project, "The Quiz Advisor". This study scrutinized sample questions asked by Cognos's clients, sorting them into six different categories. The main category consists of "How" questions, and represents 50% of the questions examined. The second largest category consists of "Why" questions and represents 20% of the total.

Given this division of user's requirements, the difficulty is to find a language in which every requirement is to be expressed. Such a language would have to be precise enough to avoid the problem of natural language processing, but vague enough to accept a large number of requirements; defining specifications that are too strict will only lead to another representation of Quiz. Rather than attempting to solve this difficult problem generally, we have opted in this thesis for a simple, ad-hoc notation to describe user's requirements. Its simplicity may not mandate the use of the term "language", but we would like to retain it for the sake of generality of discussion.

3.3.2 - Restrictions

After studying the main difficulties, it seems that the construction of a language to represent the knowledge of Quiz and the knowledge of the user's requirements is too difficult to realize directly. It is therefore necessary to make some restrictions and thus simplify the problems in order to build a prototype of the Apprentice. But we need not forget that the prototype is only a step towards solving the general problem and building a real apprentice.

3.3.2.1 - Restrictions on Quiz

It seems impossible to formalize all the knowledge on Quiz in a simple way, even for a limited question answering system. We therefore choose a subset of the language according to certain criteria:

- We include the most useful statements in order to represent as many Quiz programs as possible.

- We exclude all statements which describe an interaction with the hardware (Set printer ...).

- We maximize the number of user's requirements which can be expressed with the selected statements.

In order to facilitate the choice, we make several assumptions:

- There exists a coherent conceptual model of Quiz.

- All inconsistencies are removed.

These assumptions are not restrictive for the Apprentice because people in Cognos are working in that direction and these assumptions will be facts very soon.

These criteria lead to a subset of Quiz that includes all the statements beginning with the following keywords:

Access
Define
Select
Sort
Heading
Report
Footing
Set subfile
Go

This subset of Quiz is considered representative because a large majority of programs uses a subset of these statements and more than 70% of the user's requirements are related to them.

3.3.2.2 - Restrictions on user's requirements

The choice of restrictions on the user's requirements is naturally closely related to the subset of Quiz. We must reject any requirement which is related to a Quiz keyword not included in the selected subset. We also have to accept any other requirement in order to have the Quiz Apprentice as helpful as possible. An accepted requirement is then a requirement which refers to any selected keyword.

However, the choice for the attributes of the keywords in the accepted requirements is much more vague. We assume that the standard user of the Apprentice must have some knowledge of Quiz because an apprentice is not capable of programming by itself; its function is to help the user in the design of his program. We choose to accept any Quiz-like syntax for the attributes, knowing all the ambiguity this choice implies.

To be more precise, a user's requirement has to begin with a keyword chosen from the above and anything in a Quiz-like syntax is accepted to represent the list of attributes. We say that a requirement is in Quiz-like syntax if its syntax is either Quiz syntax, or "looks like" Quiz syntax. The user has to write a program which he thinks it is a Quiz program. The Quiz Apprentice then looks after the user to correct any non-Quiz requirement.

For instance, if the user asks:
I want to select every third name in a file

he has to give:
select every 3 name.

These restrictions roughly correspond to the user's requirements in the "How questions".

For instance, the question:
How do I report a name if the corresponding billing is over 1000 ?

may be translated by the user into:
report name if billing > 1000

After having made all these restrictions, we find that all the requirements that may still be described represent more than two thirds of the interesting question samples. We can conclude that, despite these restrictions, the prototype built reproduces quite faithfully the behavior of a real Quiz Apprentice.

3.3.3 - Formal representation

3.3.3.1 - Representation of a Quiz program

The framework of the problem being defined, we can now work on the representation of a Quiz program. The work done on the restrictions leads to an important point. The Apprentice accepts any user's requirement beginning with a Quiz keyword and the list of attributes of this keyword is anything written in Quiz-like syntax. If the attributes do not correspond to any Quiz code, the Apprentice has to find out the non-Quiz requirement and has to change it into Quiz code.

The Quiz statement can then be represented as a pair:

< Keyword , List of attributes >

But a problem still remains. In section 3.3.1.1 , we discussed the problem of uniqueness. The keywords

access
select
sort
report
set subfile
go

must be unique if they occur in a Quiz program. On the other hand, the keywords

define
heading
footing

may have several occurrences. This problem is solved with a pseudo-uniqueness approach by doing the following:

- We may define several new variables in a program, each one having a unique definition.

- We may have several heading or footing statements in a program, each heading or footing of an item being unique.

To sum up, a Quiz program is a list of statements represented by a pair

< Keyword , List of attributes >

and the list of attributes may possibly be empty. Each keyword -- including define, heading, and footing -- for which the first element of the list of attributes is required, defines a unique pair to represent a unique statement of that type in the program.

This representation is very simple and the structure of the Apprentice and its implementation naturally takes advantage of it.

3.3.3.2 - Structure of the Apprentice

The representation of a statement has two parts that may have two different meanings. The keyword may represent the part of real Quiz code and the list of attributes the part of the user's requirements. It seems natural to have a structure which is directly derived from this representation. The Apprentice is thus divided into two main parts: an interpreter and an analyzer as shown in Fig 3.1.

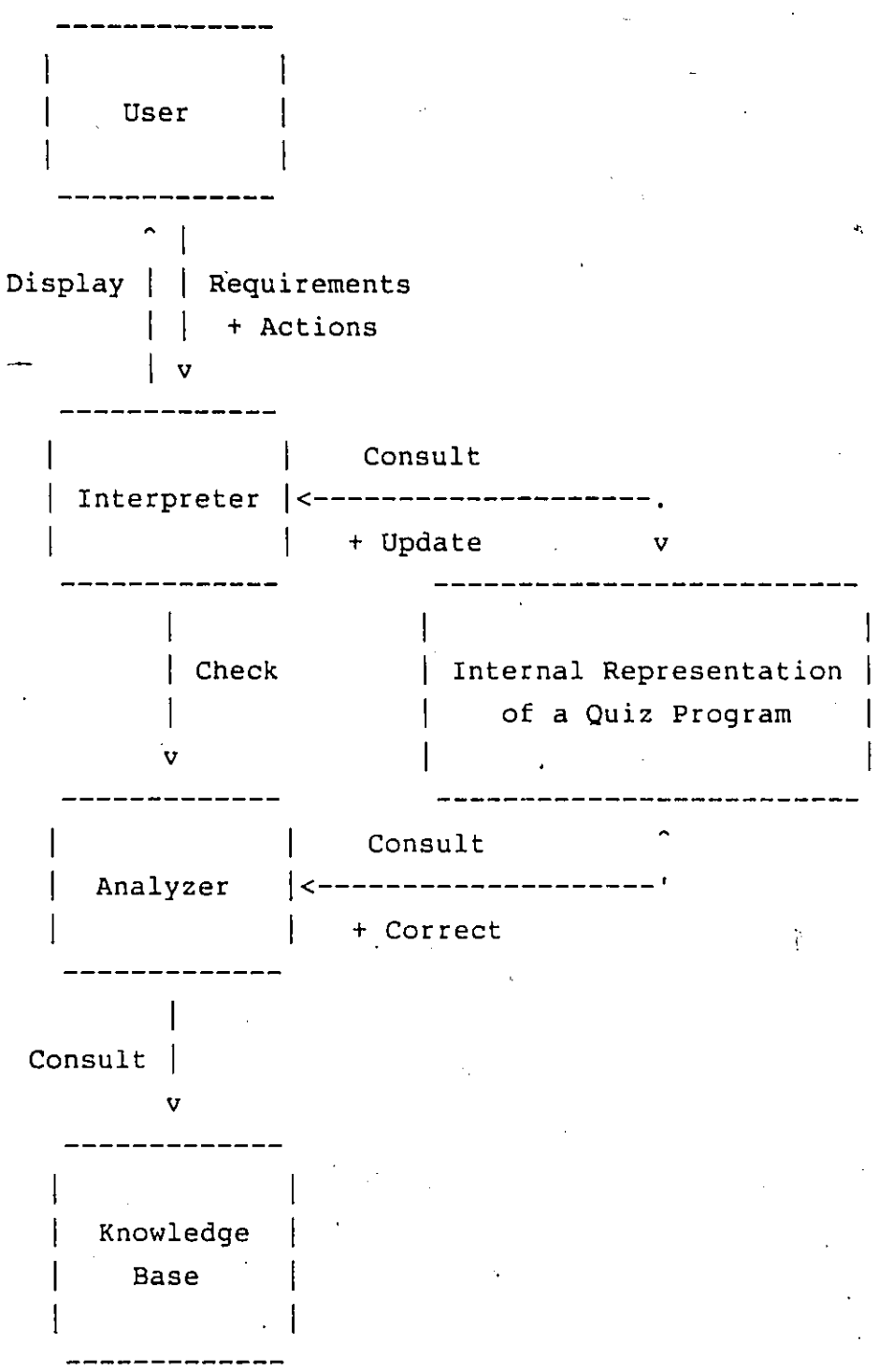

Figure 3.1 : Structure of the Apprentice

The interpreter:

The work of the interpreter is essentially related to the keyword part of a Quiz statement. It works as follows:

- It prompts the user for any requirement.
- It checks the first word of the requirement
- If this first word is a Quiz keyword, it accepts the requirement as a Quiz sentence and it updates the internal representation of the program according to this sentence. In particular, it preserves the pseudo-uniqueness.
- If the first word is not a Quiz keyword, it assumes that the requirement is an action to be performed on the program. This action is, for instance, a request for a check or a correction of the program. It then tries to perform this action.
- It then returns to the first step of this algorithm.

The role of the interpreter is to reproduce any action the Quiz parser does on a given Quiz statement. For instance, when the user enters an **access** statement, the quiz parser erases all other statements because it assumes that the user wants to work on a new program. The interpreter therefore erases all the statements when it receives an **access** statement from the user.

The only exception is for the **go** statement. Instead of producing the report defined by the program, we want the Apprentice to check the program and correct it before

accepting the user's requirements as real Quiz code.

The analyzer:

The work of the analyzer is exclusively related to the "list of attribute" part of the Quiz statements. The analyzer is activated by the interpreter or by the user through the interpreter. Once activated, it acts as follows:

- It matches the program with the knowledge base of the analyzer.

- If a non-Quiz specification is found, it generates an action, performs it to correct the code and sends a warning to the user. Once the action is performed, it returns to the first step of this algorithm.

- If there are no non-Quiz specifications, the analyzer stops.

The analyzer terminates its work when no more non-Quiz specifications are found, i.e. when the Apprentice finds that the program contains only Quiz specifications, according to the knowledge base.

These two modules require knowledge of Quiz programs for them to be helpful to the user. We will see that the nature of that knowledge varies depending on the module.

3.3.3.3 - Knowledge representation

Knowledge for the interpreter:

The interpreter must have knowledge of a Quiz parser because it has to perform the same actions. This knowledge has to be based on the Quiz keywords because the actions to be executed by the interpreter depend on them. The interpreter does not need to know more about Quiz because its role is not to analyze, but to execute. The knowledge of the interpreter is then rather restricted and is not subject to major changes. We call it "static knowledge". It does not need a special representation and can even be coded inside the interpreter.

Knowledge for the analyzer:

The analyzer is in fact the major part of the Apprentice. The choice of its knowledge is paramount for the Apprentice to be maximum benefit to the user.

Two choices may be made to build the knowledge base:

- The knowledge base may contain knowledge on the syntax and semantics of Quiz. When this knowledge is matched against the user program, the match fails each time a non-Quiz requirement occurs. The problem is to find a way to generate an action to fix the code from the failure of the matching procedure. For instance, it is very simple to detect a non-Quiz requirement as "select every 5 name", but it is impossible to transform it into Quiz code if the analyzer has no knowledge on the non-Quiz word "every".

- The knowledge base may contain knowledge about non-Quiz requirements. When the matching procedure starts, it tries to find any non-Quiz requirement in the program and the program is considered written in Quiz code if this procedure fails. If a non-Quiz requirement is found, it becomes easy to generate an action to fix the code because the non-Quiz code must have been described in the knowledge base.

We have chosen the second solution to represent the knowledge of the analyzer and that of the Apprentice in general. The knowledge base therefore contains a set of triples described as follows:

< Description of the non-Quiz requirement,
Action to perform to fix the code,
Narrative to be sent to the user as a warning >

The description of the non-Quiz requirement <P, A, N> is in fact the description of a general problem. The triple means:

For the general problem P given by the description,
The way to fix the code is to perform the general action A,
And the narrative N has to be sent to the user.

When the user wants his program to be checked, the analyzer tries to find a part of code which is an instantiation P' of the general problem P. The analyzer then sends an instantiation A' of the action A to be performed by the interpreter and warns the user with an instantiation N' of the narrative N.

At the beginning, the Quiz Apprentice has restricted knowledge. As a consequence, some non-Quiz requirements are

not recognized by the analyzer and they are therefore considered as Quiz code. We implemented a grammar to check if any syntax error was remaining. We know this is a weak test as soon as it does not test any causal relationships between the different statements of the user program. However, a constant update of the knowledge base is performed as the Quiz Apprentice is used. At a certain time, the knowledge base will be complete enough and the analyzer will not only correct a large number of user's requirements, but also detect the syntax errors in the program. The grammar will therefore be a useless repetition.

For now, knowledge acquisition is achieved by an expert having direct access to the knowledge base. Because it needs constant updating, we call this knowledge "dynamic". Automatic knowledge acquisition is described in the next chapter.

3.4 - Implementation

With the framework of the problem clearly defined, it becomes possible to build an implementation of the Quiz Apprentice. We have chosen Quintus Prolog to write the code.

3.4.1 - Internal representation of a Quiz program

Given the formal representation of a Quiz program, translation into a Prolog predicate becomes easy. The general form to represent a Quiz statement uses the ternary predicate "quiz" as follows:

```
quiz( file, keyword, list_of_attributes ).
```

The representation of a Quiz statement may be translated into a binary predicate, but we add the first argument that represents the file in which the report created by the program is located. These files are not real files. This permits the generation of two-pass reports. The program on which the user works is in the file "user"; if the user wants to save his program in a file, he may take any name but "user". A Quiz program is then represented with a set of instantiations of this predicate.

The keywords which may appear in the "quiz" predicate are the selected keywords given in section 3.3.2.1. We add a new non-Quiz keyword, **fpass**. It means that the variable defined in the **fpass** statement must be calculated in a first pass. For example:

```
fpass sales = billing subtotal at name
```

means that sales is an item to be calculated in a first pass with the given definition. This statement is especially a reminder and the analyzer may generate it when correcting a program.

The design, implementation and maintenance of a program being a dynamic action, we declare that the predicate "quiz" is a dynamic one and that the update of the program is done by asserting and retracting instantiations of that predicate. If this way of programming is rather offhand in Prolog, the advantage it gives is a better readability of the implementation.

Finally, we give two examples of Quiz programs with their respective implementations to illustrate the representation of a Quiz program.

```
Ex 1 :
access employees
select if city="boston"
sort on lastname
report employee lastname city
go

Representation of Ex 1 :
quiz(user, access, [employees]).
quiz(user, select, [if,city,=,"boston"]).
quiz(user, sort,   [on,lastname]).
quiz(user, report, [employee,lastname,city]).
quiz(user, go,     []).

Ex 2 :
access salesmen link to invoices.
define ytd_sales numeric = invoice_amount.
sort on salesman.
report summary salesman ytd_sales subtotal.
set subfile at salesman name ytd.
go.

access ytd link to salesmen.
sort on ytd_sales d.
report count salesman ytd_sales.
go.

Representation of Ex 2 :
quiz(ytd,  access, [salesmen,link,to,invoices]).
quiz(ytd,  define, [ytd_sales,numeric,=,invoice_amount]).
quiz(ytd,  sort,   [on,salesman]).
quiz(ytd,  report, [summary,salesman,ytd_sales,subtotal]).
quiz(ytd,  set,    [subfile,at,salesman,name,ytd]).
quiz(ytd,  go,     []).
```

```
quiz(user, access, [ytd,link,to,salesmen]).
quiz(user, sort,   [on,ytd_sales,d]).
quiz(user, report, [count,salesman,ytd_sales]).
quiz(user, go,     []).
```

An advantage of this representation is that it is very easy to have information on the current state of the program. For instance, if we want to know the content of the file "foo", we just have to ask

```
quiz(foo,Keyword,Attributes).
```

Another advantage comes with the pseudo-uniqueness of the list of attributes for a given predicate. If we look for a list of attributes given the file and the keyword, we need a single search to have it. This search either succeeds or fails.

3.4.2 - The Apprentice interpreter

The first task of the interpreter is to accept any requirement from the user and interpret it. We translate this interaction with the following predicate:

```
apprentice :- repeat,
                  readuser(L),
                  interpret(L),
                  display,
              L=[end],
              write('Nice meeting you !').
```

The interpreter reads the user's requirement, interprets it and displays the current state of the program. Once terminated, it reprompts the user for another requirement

until the user declares the session out with the requirement "end". The function "readuser" is an adaptation of the function "read_in" found in [Clocksin, Mellish 81]. It reads a sentence character by character and transforms it into a list of words.

A similar interaction loop is build for experts. The interpreter first prompts for the password. If the password is correct, the interpreter asserts the predicate "exp" which is considered as a flag for actions of restricted use. This interaction is:

```
apprentice :- write('Enter passwd'),
              readuser([passwd]),
              assert(exp),
              repeat,
                  readuser(L),
                  interpret(L),
                  display,
              L=[end],
              retract(exp),
              write('Nice meeting you !').
```

The interpreter may understand the user requirement as a Quiz statement or as an action. Depending on the circumstance, it has either to update the program or to perform the action. In addition, the interpreter always accepts the requirement "end". Finally, if the requirement is not recognized, it warns the user. The code is then:

```
interpret([end]).
interpret(L) :- update(L),!.
interpret(L) :- action(L),!.
interpret(_) :- write('non admissible requirement'),
                nl,!.
```

The presence of the cut operator means that, once an interpretation is found, no other interpretation is possible.

If the requirement is an action, the interpreter tests if the user is an expert. In the positive case, the action is immediately performed. If the user is not an expert, the interpreter performs or refuses to perform the action depending on whether this action is authorized. The corresponding code is:

```
action(L)      :- exp,
                  A=..L,
                  A.
action([X|L])  :- actionfilter(X),
                  A=..[X|L],
                  A.
action(_)      :- write('non available requirement'),nl.
```

The actions authorized for the user are actually:

```
actionfilter(generate).
actionfilter(rewrite).
actionfilter(readfile).
actionfilter(expert).
actionfilter(correct).
actionfilter(check).
actionfilter(grammar).
```

In the case of a Quiz sentence, the interpreter mimics the behavior of the Quiz parser. If any action of the parser is related to the keyword of the sentence, this action is executed by the interpreter. Once the action is performed, it adds the statement into the representation of the program.

An exception is made for the keyword "go". When the user enters "go", the interpreter erases the previous "go" statement if it existed, copies the program to another file if a set subfile statement is found, adds the go statement to the program, calls the analyzer to correct the entire program, and checks the syntax of the resulting program.

The procedure update is then:

```
update([access|L])        :- clear(user),
                             add(access,L,_,L).
update([define,X|L])      :- add(define,[X|L],[X|_],L).
update([select|L])        :- add(select,L,_,L).
update([sort|L])          :- clear(user,heading),
                             clear(user,footing),
                             add(sort,L,_,L).
update([report|L])        :- add(report,L,_,L).
update([heading,at,X|L])  :- add(heading,[at,X|L],
                                 [at,X|_],L).
update([footing,at,X|L])  :- add(footing,[at,X|L],
                                 [at,X|_],L).
update([set,subfile|L])   :- add(set,[subfile|L],
                                 [subfile|_],L).
update([go])              :- clear(user,go),
                             ifsubfile,
                             assert(quiz(user,go,[])),
                             correct,
                             grammar.
```

The meaning of the arguments of "add" is:

1st : keyword
2nd : list of attributes
3rd : argument associated to the keyword for the pseudo-uniqueness
4th : rest of the list of attributes

The procedure "add" checks if the fourth argument is empty. If this appears to be the case, the quiz parser understands that the user wants to erase the corresponding statement. So does the interpreter. If the fourth argument is not empty, the interpreter erases the previous statement and adds the new one. In both cases, the interpreter erases the "go" statement, indicating that a future report has to be done. The procedure "add" is then:

```
add(Kw,_,Arg,[])    :- clear(user,Kw,Arg),
                       clear(user,go).
add(Kw,List,Arg,_) :- clear(user,Kw,Arg),
                       clear(user,go),
                       assert(quiz(user,Kw,List)).
```

The main functions of the interpreter can be described in this way. But it is also important to note that all the knowledge of the Quiz syntax which we choose to represent is in the procedures "update" and "add". The procedure "update" handles specific actions on a given keyword and the procedure "add" preserves the pseudo-uniqueness important to the Apprentice.

3.4.3 - The Apprentice analyzer

Before designing the analyzer, we need a description of the knowledge required to find the non-Quiz specifications.

3.4.3.1 - Description of the knowledge base

We know that the knowledge base contains a set of triples. A triple is naturally represented with the ternary predicate "rule":

```
rule( Condition , Action , Narrative ).
```

The knowledge base therefore contains a set of instantiations of this predicate, each instantiation giving the condition by which to find the non-Quiz code, the action to perform, and the narrative to send to the user as a warning.

The "condition" part of a rule must describe why the code is not Quiz code. It contains the names of the keywords and the sublists of the lists of attributes in which the non-Quiz specification occurs. In the sublist, constants appear with their names and each variable appears with a Prolog variable and the Quiz type of the variable. Here are some examples to fix the ideas:

The sublist [every,n,item] found in the "select" statement is not Quiz code. The condition is then written:

```
[select,[every,X,numeric,Y,item]]
```

If we have the sublist [subtotal,at,item] in the "report" statement and we do not have the sublist [on,item] in the "sort" statement, we have an error. The condition is expressed as follows:

```
[report,[subtotal,at,X,item],not,sort,[on,X,item]]
```

The action part of the rule is a set of actions directly understandable by the interpreter; these actions may contain variables which occur in the condition.

The narrative is a list of words describing the error, and likewise may contain variables which occur in the condition.

An example of a complete rule is:

```
rule([select,[every,X,numeric,Y,item]],
     rewrite([every,X,Y],[Y,count,mod,X,=,0]),
     [rewrite,the,expression,every,X,Y]).
```

This means that, in the "select" statement, the sublist [every,X,Y] where X is a numeric and Y is an item, corresponds to non Quiz code. The way to fix the code is to execute the function "rewrite" with the given arguments. An appropriate narrative is then sent to the user.

If the user enters the statement:

```
select if billing > 1000 and every 3 name
```

the analyzer finds the sublist [every,3,name] and the Prolog unification gives X=3, Y=name. The analyzer performs the action

```
rewrite([every,3,name],[name,count,mod,3,=,0])
```

and the user is warned with the message "rewrite the expression every 3 name".

3.4.3.2 - Code of the analyzer

The analyzer may be activated for two different purposes. The user either wants a simple check of his program or he wants his program to be corrected; this implies two different modes for the analyzer.

In the "check mode", the analyzer searches a rule in the knowledge base and tries to match the condition with the current program. If the matching procedure fails, it searches another rule and retries the matching procedure with the new condition. If the matching procedure succeeds, the narrative is sent to the user. The procedure goes on until all rules have been called. The procedure "check" is then:

```
check :- check1,
         fail.
check.

check1 :- rule(C,_,N),
          try(C),
          writelist(N).
```

In the "correct mode", the analyzer performs the same search, but when it finds a match between a condition and the program, it sends the narrative to the user and performs the associated action to fix the code. Once the code is fixed, the analyzer begins a new search and tries every

rule in the knowledge base again. This is because every action changes the code; a condition of a rule which did not match the first time may possibly match the second time. The code of the procedure "correct" is as follows:

```
correct :- correct1,
           correct.
correct.

correct1 :- rule(C,A,N),
            try(C),
            writelist(N),
            A.
```

The role of the procedure "try" is to decompose the condition into pairs {keyword,sublist} or possibly into triples {not,keyword,sublist}. For each keyword, it calls the Quiz statement corresponding to the keyword and scans the list of attributes until a match is found between the sublist and the list of attributes. The procedure "append" acts here as a decomposing cell. The code is as follows:

```
try([]).
try([not,Prim,List|Rest]) :- not(try([Prim,List])),
                             try(Rest).
try([Prim,List|Rest])     :- quiz(user,Prim,Att),
                             append(_,Scan,Att),
                             match(List,Scan),
                             try(Rest).
```

The procedure "match" simply tries to match the sublist of the condition and the part of the Quiz statement according to the structure of the condition. If the match succeeds, the variables in the condition are unified with the corresponding words in the code. Then the corresponding

narrative and action are automatically instantiated through the Prolog unification. The procedure "match" is:

```
match([],_).
match([Var,Type|L],[Word|M]) :- matchl([Var,Type],Word),
                                match(L,M).
match([Cons|L],[Cons|M])     :- match(L,M).

matchl([X,key],Y)     :- item(Y), X=Y.
matchl([X,item],Y)    :- item(Y), X=Y.
matchl([X,numeric],Y) :- number(Y), X=Y.
matchl([X,summary],Y) :- summary(Y), X=Y.
matchl([X,string],Y)  :- string(Y), X=Y.
```

3.5 - Example

In this section, we present a commented dialogue between a user and the Quiz Apprentice. The knowledge base of the Quiz Apprentice contains all the rules described in Appendix C. However, in this dialogue, we only use a few rules to correct the user's requirements. We give here this subset of rules for a better comprehension of the dialogue:

```
rule([report,[at,X,key],not,sort,[on,X,key]],
     (writelist([enter,a,new,sort,statement]),
      readuser(R),interpret(R)),
     [X,has,to,be,a,sort,key])
```

Meaning : If the report statement contains a key X which is not declared in the sort statement, a new sort statement containing X must be entered.

```
rule([select,[every,X,numeric,Y,item]],
     rewrite([every,X,Y],[Y,count,mod,X,=,0]),
     [rewrite,the,expression,every])
```

Meaning : If the expression "every numeric item" is found in the select statement, it must be rewritten with a "modulo" operation.

```
rule([select,[X,item,Y,summary,at,Z,key]],
     generate([X,Y,at,Z]),
     [no,Y,operation,in,select])

rule([select,[X,item,Y,summary]],
     generate([X,Y]),
     [no,Y,operation,in,select])
```

Meaning : If a summary operation is found in the select statement, a new variable must be generated and its definition is given by the expression containing the summary operation.

We begin the dialogue in a state where the representation of the user program is empty.

```
| ?- apprentice.

|: access employees link to billings.

access employees link to billings
```

[The user calls the Quiz Apprentice and gives his first requirement. The Quiz Apprentice accepts it and displays the current state of the user program.]

```
|: repot project month billing subtotal at project.
Non available requirement

access employees link to billings
```

[The user wanted to add a report statement, but he made a mistake. The Quiz Apprentice does not recognize the requirement and refuses it.]

```
|: report project month billing subtotal at project.

access employees link to billings
report project month billing subtotal at project
```

[The user gives the report statement. It is added to the current program.]

```
|: select if billing subtotal at project > 1000
|: and every 3 month.

access employees link to billings
select if billing subtotal at project > 1000
       and every 3 month
report project month billing subtotal at project
```

[The user gives the select statement. It is also added to the current program. At this point, only the interpreter of the Quiz Apprentice worked. It checked every requirement and updated the user program.]

```
|: check.
project has to be a sort key
```

```
rewrite the expression every
no subtotal operation in select
no subtotal operation in select

access employees link to billings
select if billing subtotal at project > 1000 and
       every 3 month
report project month billing subtotal at project
```

[The user wants his program to be checked. The interpreter accepts the requirement and calls the analyzer to check the program. The analyzer matches the program against the knowledge base and finds three non-Quiz requirements. The last error has been found twice because two conditions match this error. The interpreter then displays the current user program.]

```
|: correct.
project has to be a sort key
enter a new sort statement
|: sort on project on month.

access employees link to billings
select if billing subtotal at project > 1000 and
       every 3 month
sort on project on month
report project month billing subtotal at project
```

[The user wants his program to be corrected. The analyser finds the first error and discovers that the sort statement is missing. It prompts the user for it, displays the new state of the program, and searches another error.]

```
rewrite the expression every

access employees link to billings
select if billing subtotal at project > 1000
       and month count mod 3 = 0
sort on project on month
report project month billing subtotal at project
```

[The analyzer finds that "every" is not a Quiz word and then translates the "every" expression into the "modulo" expression.]

```
no subtotal operation in select
give a new name to redefine billing subtotal at project
|: sales.

access employees link to billings
select if sales > 1000 and month count mod 3 = 0
sort on project on month
report project month sales

fpass sales = billing subtotal at project
```

[The subtotal in the select statement is a non-Quiz requirement. The analyzer prompts for a new name to redefine the expression containing the subtotal, replaces the expression with the new name everywhere in the program, and declares that the new name has to be calculated in a first pass.]

```
no count operation in select
give a new name to redefine month count
|: numb_month.
do you want month count at month ?
|: yes.

access employees link to billings
select if sales > 1000 and numb_month mod 3 = 0
sort on project on month
report project month sales

fpass sales = billing subtotal at project
fpass numb_month = month count at month
```

[The same type of non-Quiz requirement occurs with the count in the select statement. The only difference is that the analyser does not know at which key the user wants to count. It proposes the last key of the sort statement. No other non-Quiz requirements are found. The Quiz Apprentice prompts the user for another requirement.]

```
|: go.

access employees link to billings
select if sales > 1000 and numb_month mod 3 = 0
sort on project on month
report project month sales
go

fpass sales = billing subtotal at project
fpass numb_month = month count at month
```

[The user enters the go statement. The interpreter accepts the statement, tries to perform a correction and a syntatic

check of the program. Nothing has been found. The program is considered as a Quiz program, according to the knowledge base.]

|: end.

```
access employees link to billings
select if sales > 1000 and numb_month mod 3 = 0
sort on project on month
report project month sales
go

fpass sales = billing subtotal at project
fpass numb_month = month count at month
```

Nice meeting you !

[The user terminates the session by entering the "end" requirement.]

The learning module

4.1 - Presentation of the learning module

4.1.1 - Purpose of the learning module

A major impediment to developing knowledge-based systems is the knowledge acquisition bottleneck, that is the difficulty of building up a complete enough and correct enough knowledge base. In the Quiz Apprentice, this problem is handled by a special component, the learning module. This device incorporates learning machine methods to automate the acquisition of new knowledge. This knowledge, represented by general concept descriptions, is acquired by analyzing examples of problems.

4.1.2 - Underlying principles

The learning module becomes useful when the user gives a set of specifications, asks the Apprentice to check the program according to its knowledge base, and finds that the resulting program appears to have some non-Quiz specifications. The Apprentice needs more knowledge, and an expert must begin dialogue with the learning process in order to update the knowledge base.

An expert is a person who has sufficient knowledge of Quiz and sufficient knowledge of the Apprentice to supervise the learning process. He also has more privileges than a normal user and has direct access to every part of the Apprentice.

The learning module is based on problem-solution pairs. In the knowledge base, the rules contain two essential parts: a condition and an action. The condition represents a general description of a problem P, for example, a general description of a non-Quiz requirement, and the action represents the solution S of the problem, that is the action which transforms the non-Quiz requirement into Quiz code. The role of the learning module is to create a general problem-solution pair from a particular problem-solution pair given by the expert. When the expert wants to update the knowledge base, the learning process starts and follows this general procedure:

- The expert enters the set of requirements which describes the particular problem P0.

- The expert enters the particular solution S0 of the problem P0.

- The process discriminates those attributes in the lists of attributes of the requirements that seem to be the cause of the problem P0.

- The process generalizes the problem P0 to a problem P and the solution S0 to a general solution S according to the selected attributes.

- The expert enters the narrative.

- The process generalizes the narrative according to the selected attributes and creates a new rule to be inserted in the knowledge base.

When the Apprentice corrector is later confronted with a problem P' which is an instance of the general

problem P, it may copy the relationship between P and P' to produce a solution S' of the problem P', based on the solution S. The figure 4.1 sums up this general procedure.

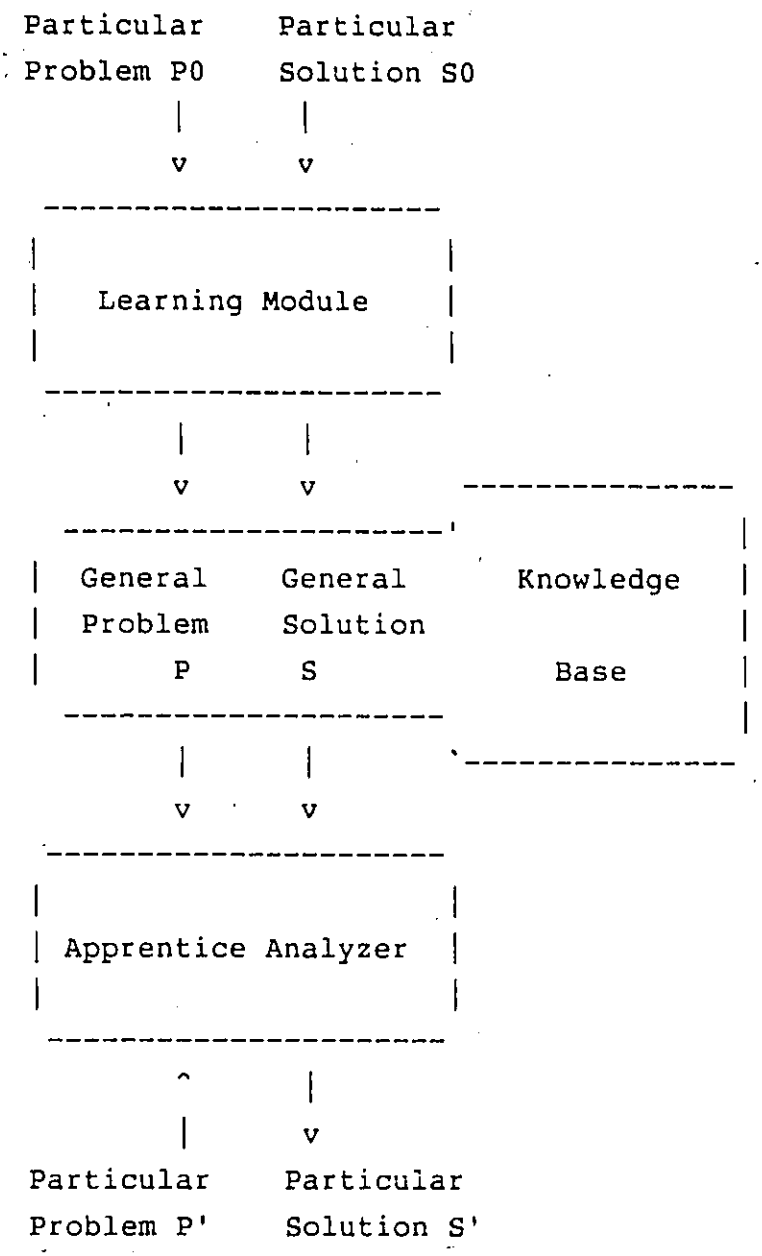

Figure 4.1 : Integration of the learning module in the Quiz Apprentice

4.2 - Development of the learning module

The problem now is to find an algorithm or a set of algorithms to generalize the particular problem-solution pairs. In order to be successful in this, we must first study in detail the set of problems for which generalization is required.

4.2.1 - Study of rules

4.2.1.1 - Set of problem-solution pairs

The first part of the work is to have as exhaustive a set of user's questions as possible. This work as been accomplished in the study cited in part 3.3.1.2 of this thesis. From this set of questions, we selected those which may be represented in Quiz-like syntax. These would begin with a selected Quiz keyword and end with a list of attributes. Half of these selected questions have an implicit solution when they are represented in the input form. For instance, a general question may be represented in the user's requirement as in the following example:

Question : Can I define values conditionally ?

Requirement : define item = value_1 if <condition>
else value_2

This is the general form in which to define values conditionally in Quiz. We should note that in fact the user will never write <condition>, but rather the specific condition which corresponds to his problem. If the Quiz Apprentice had a natural language interface, this interface would translate the general question into an acceptable requirement. The

question above would then find an answer during natural language processing and the Apprentice would just accept the statement.

In the other half of the set of questions, four main groups were formed, each one representing a general problem encountered by the users. The characteristics of each group will be defined later in this thesis. From each group, we choose a problem-solution pair that we consider representative. These problem-solution pairs are the following:

1 - Problem : How do I put a heading in a subfile ?

Solution : Impossible, no headings are put in a subfile.

2 - Problem : How do I subtotal at control break ?

Solution : You declare the control break key in the sort statement and you report the subtotal at key.

3 - Problem : How do I select every n item ?

Solution : You select if item count mod n equals 0.

4 - Problem : How do I select with a subtotal ?

Solution : You calculate your subtotal in a previous pass, affect it to a new variable and select with this new variable in the main pass.

Each question has been translated manually into an accepted user's requirement and, for each non-Quiz requirement, a rule has been created. The rules show us the formal representation which the learning module has to generate.

The requirements and the rules are:

```
1 -     heading at project skip page
        report subfile employee billing
        set subfile at name keep

        rule([heading,[],set,[subfile],report,[subfile]],
             clear(user,heading),
             [no,headings,in,subfile])
```

Interpretation : if there is a heading statement, a set subfile statement and if the report will go to a subfile, erase all headings in the program.

```
2 -     sort on branch on month
        report month billing subtotal at project

        rule([report,[at,X,key],not,sort,[on,X,key]],
             (show(sort),write('enter new sort'),
                        readuser(L),update(L)),
             [X,has,to,be,a,sort,key])
```

Interpretation : if the expression [at,X] is found in the report statement and the expression [on,X] is not in the sort statement, then show the existing sort statement, read the new sort statement and update the program. The question on the subtotal has an implicit answer because "report billing subtotal" is actually Quiz code. The rule solves the problem on the control break.

```
3 -     select if billing > 1000 and every 10 name

        rule([select,[every,X,numeric,Y,item]],
             rewrite([every,X,Y],[Y,count,mod,X,=,0]),
             [rewrite,the,expression,every])
```

Interpretation : if the expression [every,X,Y] is found in a select statement, rewrite this expression into [Y,count,mod,X,=,0].

4 - select if billing subtotal > 1000

```
rule([select,[X,item,Y,summary]],
     generate([X,Y]),
     [no,Y,operation,in,select])
```

Interpretation : if the expression [X,Y] is found in a select statement, generate a new variable which will have the value of item summary.

In fact, a richer set of about 40 rules has been developed for the Apprentice, but the four rules above are considered representative for the study of the development of the learning module. The remaining rules are shown in Appendix C.

4.2.1.2 - Decomposition into classes

The first two rules for non-Quiz requirements involve several Quiz statements and the last two involve unique Quiz statements. This remark seems very superficial, but the differences between the groups are more fundamental than first appears.

In the first group, we note that it is impossible to predict the action to perform. In example 1, the expert chooses to erase every heading statement because he thinks that the main idea is to save the report in a subfile. But he may have chosen either to erase the set subfile statement or to leave the choice to the user and simply warn him of

the problem. Either possibility is correct; the choice depends on the context of the use of the Quiz Apprentice. This implies an inability on the part of the learning module to decide the action to perform or to choose the statement on which the action has to be performed.

The learning module can, however, decide the action to perform for rules in the second group. Knowing that the non-Quiz requirement occurs in the list of attributes of the statement, the only possibilities for correcting the code are rewriting one expression into another or redefining a subset of attributes into a new variable. This naturally implies two subclasses of rules, the "rewrite" rules as rule 3 and the "generate" rules as rule 4.

The different classes of rules are then:

- "several statements" rules
- "unique statement" rules
  - "rewrite" rules
  - "generate" rules

4.2.2 - Algorithms

Starting from these four examples and the rules they imply, the next step is to develop algorithms that create general rules from particular problem-solution pairs. We naturally have two different algorithms corresponding to the two main classes of rules.

4.2.2.1 - Algorithm for the generation of "multiple statements" rules

We start the construction of the algorithm for "multiple statement" rules with the requirements corresponding to the particular problems to be treated by rule 1 and rule 2. These examples are P1 and P2:

P1 heading at project skip page
report subfile month billing
set subfile at name keep

P2 report month billing subtotal at project
not sort on project

"Not sort on project" means that "on project" does not appear in the sort statement.

The problem to solve comes from the conjunction of the requirements; we therefore ignore any problem which may occur in a single requirement. Every requirement has its list of attributes. For instance, "at project" and "skip page" are the attributes of the first requirement of P1. We then check every word which occurs in at least two lists of attributes. The analysis of the resulting set of words has good chances of revealing the origin of the problem. This is the case for 80% of our complete set of rules. We then declare that the resulting set of words certainly indicates the origin of the problem. All other words, except keywords, are dropped. The examples are then transformed into PP1 and PP2:

```
PP1     heading at
        report subfile
        set subfile at
set of common words : {at, subfile}

PP2     report project
        not sort project
set of common words : {project}
```

At this point, we must perform the generalization according to the set of common words. This set contains two types of words, Quiz variables and Quiz keywords. Quiz variables are generalized to their Quiz types. For instance, "project" is generalized to the type "key". Quiz keywords are considered as constants (i.e. "subfile") and are not generalized; exceptions are made for the Quiz keywords "at" and "on" because of their special meaning in Quiz. The examples are finally transformed into P1' and P2':

```
P1'     heading
        report subfile
        set subfile

P2'     report [key]
        not sort [key]
```

The problem is nearly solved. Only the narrative and the action to perform are lacking for the construction of a complete rule. The general algorithm for building "multiple statements" rules is then:

```
1       Enter the first statement

2       Enter another statement
```

3 Compute the set of common words

4 Return to step 2 if the last statement has not been entered

5 Reduce the statements according to the set of common words

6 Generalize variables to their types for the selected words

7 Enter the narrative and the action to build the general rule

The set of common words is computed for each statement entered. This operation is simple because it can be computed by induction as follows:

Let Wn be the set of common words for n statements.
Let Sn be the n-th statement entered.
Then

$$Wn = \bigcup_{1=<i,j=<n} (Si \hat{} Sj)$$

Wn may also be written

$$Wn = (Sn \hat{} (\bigcup_{1=<i<n} Si)) \cup (\bigcup_{1=<i,j<n} (Si \hat{} Sj))$$

Or

$$Wn = (Sn \hat{} (\bigcup_{1=<i<n} Si)) \cup Wn-1$$

Let Un be the union of the first n statements.
We can then write

$$Wn = (Sn \hat{} Un-1) \cup Wn-1$$

$$Un = Sn \cup Un-1$$

The start of induction may begin at step 1 with

$$W1 = \text{empty set}$$

$$U1 = S1$$

4.2.2.2 - Algorithm for the generation of "unique statement" rules

As in the preceding section, we start the development of the algorithm with examples corresponding the particular problems to be treated by rules 3 and 4. These examples are P3 and P4:

P3 select if billing > 1000 and every 5 name

P4 select if billing subtotal > 1000

We know that the non-Quiz specification occurs in the list of attributes. It is then natural to give the Quiz statement which implements the non-Quiz specification. The statements corresponding to the examples are Q3 and Q4:

Q3 select if billing > 1000 and name count mod 5 = 0

Q4 select if total_billing > 1000

The non-Quiz specifications that occur in the statement P are found by checking the difference between the statements P and Q. This check returns a pair of lists which represent the non-identical parts of P and Q, respectively. Those lists consist of words, where a word means a sequence of non-blank characters. Comparison of P3 and Q3 gives:

E3 "every 5 name"
"name count mod 5 = 0"

Comparison of P4 and Q4 gives:

E4 "billing subtotal"
"total_billing"

It is interesting to observe the words common to both lists. In example E3, we have "5" and "name"; in the second example, there are no common words. We distinguish two categories of rules, then, depending on the intersection of the set of words from the lists. When the intersection is not empty, the general solution consists of rewriting one list into another. This describes the "rewrite" rules. In the contrary situation, the general solution consists of replacing the first list by a new variable. This describes the "generate" rules.

Generation of the "unique statement" rule is performed according to a set of common words, as in the algorithm for "multiple statements" rules. For the "rewrite" rules, this set is obviously the set of words common to the lists describing the problem and its solution. For the "generate" rules, we choose as the set of words the list describing the problem. The generalization has the same characteristics as in the last section: Quiz variables are generalized to their Quiz types and Quiz words are

considered as constants.

At this point the only information we lack for building the rule is the narrative. The algorithm to build the "unique statement" rules is then:

1 Enter the requirement

2 Enter the correct Quiz statement

3 Find the lists representing the difference between the requirement and the statement

4 If the intersection of the two lists is empty, the rule is a "generate" rule and the set of words for the generalization contains the words in the requirement

5 If their intersection is not empty, the rule is a "rewrite" rule and the set of words for the generalization is the intersection found in 3 above

6 Generalize variables to their types according to the selected words

7 Enter the narrative to build the general rule

The generation of the rules which is based on a single problem-solution pair uses a set of words constructed during the analysis of the particular problem. This generalization is in fact very basic because only Quiz variables are generalized and the only degree of generalization is that of type. More complex generalizations are possible, of course. One could, for instance, investigate the causal relationship between a problem and its solution, and attempt

to generalize along this axis. Given the limited time and resources of this project, however, such a study is not possible. This simple generalization may be taken as an example and as a first step to a more complex generalization.

4.3 - Implementation

The generalization of rules being the same for both classes of learning, we first implement it to see how it fits into the translation of the algorithms.

4.3.1 - Implementation of the generalization part

Both types of learning generalizes rules according to a set of words. This set of words must be transformed into a new set, called the **reference**, before generalization can be performed. Quiz words are removed from the reference set, leaving only Quiz variables. Each variable is given two attributes, one a free Prolog variable and the other its Quiz type. The reference set, then, is made up of triples. Every generalization must be performed according to this reference.

We present this example to clarify the idea:

If the set of words is:

[billing,subtotal,at,name]

The only Quiz word is "at" and the reference is then:

[[billing,X,item],[subtotal,Y,summary],[name,Z,item]]

where X, Y, Z are Prolog variables. If the sentence:

no subtotal in a select

has to be generalized according to the above reference, the result is:

no Y in a select

The procedure "makeref" gives attributes to the Quiz variables and builds the reference. This procedure checks every word of the list of words. If a word has a Quiz type, the procedure adds it to the reference. Otherwise, the word is removed. The code is then:

```
makeref([X|K],[[X,_,numeric]|L]) :- number(X),makeref(K,L).
makeref([X|K],[[X,_,summary]|L]) :- summary(X),makeref(K,L).
makeref([X|K],[[X,_,string]|L]) :- string(X),makeref(K,L).
makeref([X|K],[[X,_,item]|L]) :- item(X),makeref(K,L).
makeref([_|K],L) :- makeref(K,L).
```

When a sentence has to be generalized, every word of the sentence which occurs in the reference will be replaced by the associated free Prolog variable. In this way, every sentence has the same free variables for the same words of the reference. The procedure which replaces words with variables in the reference uses three arguments, the input sentence, the output sentence, and the reference. Its code is:

```
replvar([],[],_).
replvar([X|K],[Y|L],R) :- member([X,Y,_],R),replvar(K,L,R).
replvar([X|K],[X|L],R) :- replvar(K,L,R).
```

We need an extra function which generalizes sentences as "replvar", but instead of just replacing a word

occurring in the reference by its associated free Prolog variable, we replace it by the free Prolog variable and the Quiz type given in the reference. This function is especially useful in constructing general conditions for the rules. For instance, given the expression and the reference:

```
every 10 name
[[10,X,numeric],[name,Y,item]]
```

the generalization of the expression according to the reference is:

```
every X numeric Y item
```

This extra function uses the same arguments as the preceding function and its code is the following:

```
replvartype([],[],_).
replvartype([at,X|K],[at,Y,key|L],R) :- member([X,Y,item],R),
                                        replvartype(K,L,R).
replvartype([on,X|K],[on,Y,key|L],R) :- member([X,Y,item],R),
                                        replvartype(K,L,R).
replvartype([X|K],[Y,T|L],R) :- member([X,Y,T],R),
                                replvartype(K,L,R).
replvartype([X|K],[X|L],R) :- replvartype(K,L,R).
```

4.3.2 - Implementation of "several statements" rules algorithm

The "learn" procedure begins the dialogue by prompting the user for the first two statements:

```
learn(multiple) :- write('enter the statement'),
                   readuser(P),
                   write('enter the statement'),
                   readuser(Q),
                   multstat(Q,[P],P,[]).
```

The "multstat" procedure deals with multiple statements and reproduces the "several statements" algorithm. Its arguments are:

First argument : The last statement entered.

Second argument : The list of statements entered so far, except the last one.

Third argument : The union of the statements.

Fourth argument : The set of common words.

The "multstat" procedure first checks the last statement entered. If this one is not empty, the user may have other statements to enter. The procedure then prompts the user for the next statement, sets the list of common words and goes through this procedure again until an empty statement is found. When an empty statement has been entered, the procedure reduces the statements according to the set of common words, builds the reference, and generalizes the statements to create the condition of the future rule. Then it prompts the user for the narrative, generalizes the narrative, prompts for the action to perform, and inserts the new rule in the knowledge base. The procedure is then:

```
multstat([],K,_,W) :- reduce(K,[],L,W),
                      makeref(W,R),
                      generalize(L,C,R),
                      write('text for the rule'),
                      readuser(T),
                      replvar(T,N,R),
                      write('action to perform'),
                      read(A),
                      asserta(rule(C,A,N)).
multstat(L,K,U,W) :- write('enter the statement'),
                     readuser(P),
                     setwords(L,U,W,Ul,Wl),
                     multstat(P,[L|K],Ul,Wl).
```

The setting of the list of common words uses the formulas given in section 4.2.2.1. Its arguments are the n-th statement, the union and the set of common words at step n-1 and the union and, the set of common words at step n. The procedure is:

```
setwords(Stn,U,W,Un,Wn) :- inter(Stn,U,T),union(W,T,Wn),
                           union(Stn,U,Un).
```

The "reduce" procedure takes each statement, reduces its list of attributes according to the set of common words, and builds a list which has the format of the condition of a rule. The output list it produces is reversed to respect the entry order of the statements. The arguments of this procedure are the list of statements in reverse order, the reduced list of statements in reverse order,the reduced list of statements in the entry order, and the set of common words. The procedure is:

```
reduce([],L,L,_).
reduce([[not,X|Y]|K],L,M,W) :- reducelist(Y,Y1,W),
                               reduce(K,[not,X,Y1|L],M,W).
reduce([[X|Y]|K],L,M,W) :- reducelist(Y,Y1,W),
                           reduce(K,[X,Y1|L],M,W).
```

The procedure "generalize" generalizes the list of reduced statements according to the reference. This list contains either Quiz keywords or lists of attributes. If a Quiz keyword is found, the procedure copies it. If a list of attributes is found, it is transformed through the last generalization function described. The procedure is:

```
generalize([],[],_).
generalize([[X|K]|L],[K1|L1],R) :- replvartype([X|K],K1,R),
                                   generalize(L,L1,R).
generalize([X|L],[X|L1],R) :- generalize(L,L1,R).
```

4.3.3 - Implementation of "unique statement" rules algorithm

When the expert wants to add a new rule for a new type of non-Quiz requirement, he may also want to add other rules which correct similar non-Quiz requirements. For instance, these two non-Quiz requirements are similar:

```
select every 10 name
select each 10 th name
```

It may be useful to build a rule for each statement at the same time. In such a case, we would add this small change to the previous algorithm.

The procedure "learn" begins the dialogue by prompting the user for the first non-Quiz statement:

```
learn(unique) :- write('enter the statement'),
                 readuser(P),
                 uniquestat(P,[]).
```

The "uniquestat" procedure reads every other statement as equivalent to the initial one. Once the user enters an empty statement, the procedure prompts the user for both the Quiz statement which represents the translation in Quiz of the non-Quiz statements and the narrative of the rules. The arguments of this procedure correspond to the last statement entered and the list of all statements entered except the last one. The procedure is then:

```
uniquestat([],L) :- write('give the Quiz statement'),
                    readuser(R),
                    write('give the text for the rules'),
                    readuser(T),
                    create_n(R,L,T).
uniquestat(K,L) :- write('give another equivalent form'),
                   readuser(R),
                   uniquestat(R,[K|L]).
```

The "create_n" procedure creates as many rules as there are non-Quiz statements. It takes every non-Quiz statement and tests it to determine whether it starts with the same keyword as the Quiz statement. If the statement does not begin this way, it is dropped from the procedure. If it does, however, the procedure finds the expressions that differ between the non-Quiz statement and the Quiz statement, finds the intersection of these expressions and creates a rule. The arguments of this procedure are the Quiz statement, the list of the non-Quiz statements, and the

narrative of the rules. The procedure is:

```
create_n(_,[],_).
create_n([X|K],[[X|L]|M],T) :- change(E1,E2,K,L),
                               inter(E1,E2,I),
                               create(X,E2,E1,I,T),
                               create_n([X|K],M,T).
create_n(K,[L|M],T) :- writelist(L),
     write('is not compatible with the Quiz statement'),nl,
                          create_n(K,M,T).
```

The procedure "change" finds the expressions that differ between two lists. It also works in the opposite way, changing one expression into another in a given list. The procedure is:

```
change(A,B,[X|K],[X|L]) :- change(A,B,K,L).
change(A,B,K,L) :- append(A,R,K),append(B,R,L).
```

The "create" procedure creates a rule given the Quiz keyword, the non-Quiz expression, the Quiz expression, the intersection of these two expressions, and the narrative. If the intersection is empty, the rule is a "generate" rule. The procedure builds the reference from the non-Quiz expression and generalizes the condition of the rule, the non-Quiz expression, and the narrative according to the reference. It then inserts the new rule into the knowledge base.

If the intersection is not empty, the rule is a "rewrite" rule. The procedure builds the reference from the intersection of the expressions and generalizes the condition of the rule, the non-Quiz expression, the Quiz expression, and the narrative according to the reference. It then inserts the new rule into the knowledge base. The procedure is then:

```
create(X,E,_,[],T) :- makeref(E,R),
                      replvartype(E,K,R),
                      replvar(E,L,R),
                      replvar(T,N,R),
                      asserta(rule([X,K],generate(L),N)).
create(X,E,QE,I,T) :- makeref(I,R),
                      replvartype(E,K,R),
                      replvar(E,L,R),
                      replvar(QE,M,R),
                      replvar(T,N,R),
                      asserta(rule([X,K],rewrite(L,M),N)).
```

4.4 - Example

In this section, we present a commented dialogue between a user and the Quiz Apprentice. The knowledge base of the Quiz Apprentice is empty. Any statement entered is then considered as a Quiz statement by the Quiz Apprentice. The Apprentice then has to learn rules in order to correct the user program. The dialogue begins in a state where the user already entered his program.

```
| ?- apprentice.

access employees link to billings
select if sales > 1000 and every 10 name
report project print at name name sales
```

[The Quiz Apprentice displays the current user program.]

check.

```
access employees link to billings
select if sales—> 1000 and every 10 name
report project print at name name sales
```

[The user wants his program to be checked. As the knowledge base is empty, no errors are found and the program is considered as a Quiz program.]

```
go.
syntax error in condition here
every 10 name

access employees link to billings
select if sales > 1000 and every 10 name
report project print at name name sales
```

[The user enters the go statement. The interpreter asks the analyzer to correct the program, but no errors are found. A syntactic check is performed and an error is discovered. The Quiz Apprentice does not know how to correct it and has to learn new rules.]

```
expert.
Enter passwd : passwd.
```

[The expert mode is entered.]

```
learn multiple.
enter the statement : report project print at name.
enter the statement : not sort on name.
```

```
enter the statement : .
narrative for the rule : name has to be a sort key.
action to perform : writelist([enter,a,new,sort,statement]),
                    readuser(R),interpret(R).

rule([report,[at,X,key],not,sort,[on,X,key]],
     (writelist([enter,a,new,sort,statement]),
          readuser(R),interpret(R)),
     [X,has,to,be,a,sort,key])
```

[The expert enters the learning module for the Quiz Apprentice to learn a "multiple statements" rule. The learning module prompts the user for the set of statements which describes the error, for the narrative, and for the action to perform. It then displays the rule it learns.]

```
learn unique.
enter the statement : select if sales > 1000
                              and every 10 name.
give another equivalent form : select if sales > 1000
                                      and each 10 th name.
give another equivalent form : select if sales > 1000
                                      and every 10 th name.
give another equivalent form : .
give the Quiz statement : select if sales > 1000
                                  and name count mod 10 = 0.
give the text for the rules : rewrite the expression
                               equivalent to every.

rule([select,[every,X,numeric,th,Y,item]],
     rewrite([every,X,th,Y],[Y,count,mod,X,=,0]),
     [rewrite,the,expression,equivalent,to,every])
```

```
rule([select,[each,X,numeric,th,Y,item]],
     rewrite([each,X,th,Y],[Y,count,mod,X,=,0]),
     [rewrite,the,expression,equivalent,to,every])
rule([select,[every,X,numeric,Y,item]],
     rewrite([every,X,Y],[Y,count,mod,X,=,0]);
     [rewrite,the,expression,equivalent,to,every])
```

[The user enters the learning module for the Quiz Apprentice to learn a "unique statement" rule. The learning module prompts the expert for the set of equivalent statements, each one describing a different expression of the same non-Quiz requirement. It prompts the expert for the narrative and then displays the set of rules it learns.]

```
end.

access employees link to billings
select if sales > 1000 and every 10 name
report project print at name name sales
```

[The user quits the expert mode and the interpreter displays the current program.]

```
correct.
rewrite the expression equivalent to every

access employees link to billings
select if sales > 1000 and name count mod 10 = 0
report project print at name name sales
```

[The user asks the interpreter for a correction of his program. The analyzer matches the new rules against the user program and finds a first non-Quiz requirement. The program

is then corrected.]

```
month has to be a sort key
enter a new sort statement
sort on project on month.
```

```
access employees link to billings
select if sales > 1000 and name count mod 10 = 0
sort on project on month
report project print at name name sales
```

[The analyzer finds a second error with the "multiple statements" rule. The program is corrected and the current state of the program is displayed. No other errors are found in the program. It is then considered a Quiz program.]

```
go.
```

```
access employees link to billings
select if sales > 1000 and name count mod 10 = 0
sort on project on month
report project print at name name sales
go
```

[The user enters the go statement. The interpreter accepts the statement, tries to perform a correction and a syntactic check of the program. No mistake has been found. The program is considered as a Quiz program.]

```
end.

access employees link to billings
select if sales > 1000 and name count mod 10 = 0
sort on project on month
report project print at name name sales
go

Nice meeting you !
```

[The user terminates the session by entering the "end" requirement.]

Conclusion

In this chapter, we present an evaluation of the Quiz Apprentice prototype, giving some comments on the main advantages and drawbacks. Finally, we try to detail the future work that can be done to enhance this prototype.

5.1 - Drawbacks of the prototype

One of the main drawbacks of the prototype is due to the restrictions initially imposed on the project. We gave reasons for these restrictions, but a real Quiz Apprentice should be able to help any user on any Quiz program. The purpose of this prototype was to prove the feasibility of such an Apprentice, and we think that the performance of this prototype is sufficient to prove it.

The Quiz Apprentice should be able to handle all the possibilities of Quiz. This is not an obvious task, even starting from the prototype, because Quiz is still in development and many inconsistencies in terminology may still be found in the language.

The Quiz Apprentice should also handle any kind of user's requirement. Expressing all his requirements in a strict format is quite limiting for a user. The best user interface would be a natural language one. A natural language interface would transform any form of a user's requirement into a given format; then the Quiz Apprentice could fully satisfy the user.

A major drawback to the Quiz Apprentice is that it is language dependent. The Apprentice is designed to handle only Quiz programs; we know that the Quiz Apprentice will never understand languages like Pascal or Lisp. The problem of a general apprentice has been studied for many years and the research on this subject is still moving in new directions. This Apprentice does not meet the criteria of a multi-language apprentice. However, we think that it certainly may be redesigned to handle languages similar to Quiz. The ultimate goal would be to have a general "fourth generation language" apprentice. The Quiz Apprentice would gain a new dimension if this goal appears in the future to be reachable.

5.2 - Advantages of the prototype

One of the first advantages of the Quiz Apprentice is its speed. Speed is a determining factor in deciding whether or not a system is interactive, and an apprentice is by nature an interactive system. The Quiz Apprentice satisfies the speed constraint, since when a user works with it, he considers its answer to be instantaneous. The task which requires the greatest amount of time is that done by the analyzer -- the correction of the program. The time spent on this task depends on the number of non-Quiz requirements found in the program, but the time spent to find a non-Quiz requirement and to translate it into Quiz code is less than five seconds. This may be considered an excellent time for a prototype implemented in Prolog.

Another advantage of the Quiz Apprentice is its simplicity. The formalizations described in Chapter 3 permitted the implementation of the Quiz Apprentice in a hundred lines of Prolog code. This simplicity has two results. The first

is speed; The Quiz Apprentice is considered fast. The second is that the Quiz Apprentice is easy to develop, starting from this initial implementation in Prolog. For instance, new keywords may easily be implemented inside the interpreter. The simplicity of the Quiz Apprentice also makes it easy to understand. It is possible that future development of the Quiz Apprentice may easily be done by any person having sufficient knowledge of Quiz and Prolog.

Another advantage of the Quiz Apprentice is that its simplicity does not interfere with its performance. To evaluate the performance of the Quiz Apprentice, we needed a representative set of user's requirements. We started with a representative set of user questions compiled for the project **A Quiz Advisor**. We immediately rejected some questions whose subject was not Quiz and transformed the remaining ones into a representation understandable by the Quiz Apprentice. We then were able to study the user's requirements and the way the Quiz Apprentice understands them. Our prototype was unable to understand some of these requirements, those referring to side-effects in Quiz, formats and types, and some which were simply too difficult to understand. The Quiz Apprentice does find an acceptable answer -- the Quiz program it generates respects the user's requirement -- for all others requirements. This set of understandable requirements represents 70% of the initial set of requirements. This result allows us to say that the performance of the Quiz Apprentice is noteworthy for its capability of helping the user in the implementation and the design of his program in most cases.

We tried to deepen the study of the set of understandable requirements and their associated questions. The goal was to find the knowledge necessary for the Apprentice to help the user. Some questions find an implicit answer

when they are translated into an understandable requirement. For instance, the question

How do I sort on a defined item ?

may be translated into the requirements

```
define X = ...
sort on X
```

which are simply inserted into the current program because they are actually written in Quiz code. Some questions require the static knowledge of the interpreter. For instance, the question

Is the "define" statement calculated after the "select" statement ?

finds its answer when the interpreter displays the user program with the "define" statement before the "select" statement. But the majority of the requirements need to be treated by the analyzer in order to be translated into Quiz code. We then studied the number of rules needed to correct the non-Quiz requirements. The result is the curve in Fig 5.1

It is not possible to make a definitive conclusion with such a small set of requirements, but we can affirm that the Quiz Apprentice needs to learn fewer rules as the number of requirements increases. It would be interesting to know if this curve has an asymptote. To answer this question, we would ask a certain number of users to work with the Quiz Apprentice. Each one would create and complete his own knowledge base. The concatenation of all the knowledge bases will be a nearly complete knowledge base which will

then give an approximation of the asymptote.

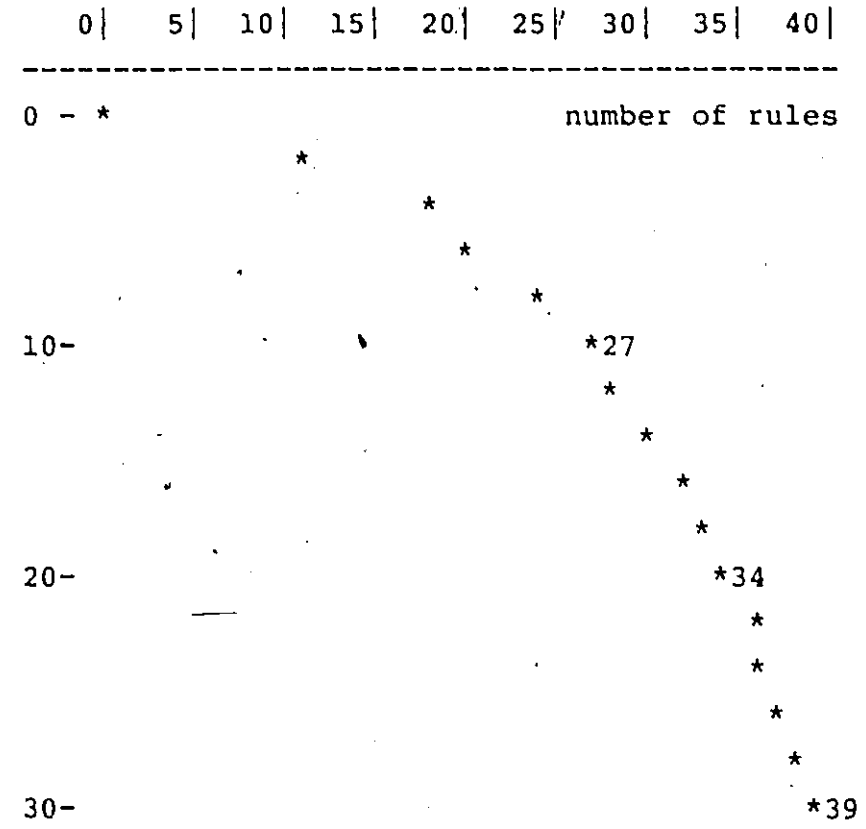

Fig 5.1 : Rules needed for the non-Quiz requirements

The last work we did was to have the Quiz Apprentice learn all the rules needed to correct the non-Quiz requirements. The algorithms are simple, and the Quiz Apprentice was able to learn 37 of the 39 rules of the knowledge base. The remaining rules were related to delicate questions on linking. Another advantage of the Quiz Apprentice, then, is the efficiency of its learning module.

5.3 - Future work

The goal of the future research in this area is to build a powerful Quiz Apprentice which will be able to help the user in any situation. Much work remains to be done on the Quiz Apprentice because we have only implemented a prototype.

The first step would be to remove the drawbacks of the Quiz Apprentice. To do this, we must first enhance the Apprentice's knowledge of Quiz. We have already said that the Quiz Apprentice must understand any user's requirement about Quiz. Part of this knowledge is easy to implement. For instance, the interpreter can easily learn new keywords like "page heading". But other types of knowledge may be much more difficult to learn. For instance, the notion of linking, which is essential in Quiz, is considered very difficult to understand, even by Quiz experts. It will certainly not be any easier for the Apprentice's learning module to assimilate.

We also need to study the Quiz Apprentice's capacity to understand other fourth generation languages. Even though fourth generation languages work on the basic principle "minimum work, minimum skill, minimum maintenance for maximum results", they may be very different in nature. We then have to define a subset of fourth generation languages which are "similar" to Quiz, that is which would be good candidates for implementation in the Quiz Apprentice. From this point, we have to design a general representation of a user program and to build switch modules which can translate a program from a given language into this general representation and the reverse. This general representation may be compared to the notion of **deep plan** in the Programmer's Apprentice [Rich, Schrobe 79].

Another task would be building a natural language interface. This interface would authorize the user to express his requirements in any form he desires. The constraint of an input format will therefore be removed. But the problem of natural language processing is a very difficult one and is being studied in current research. Even for the limited domain of programming in Quiz, this type of interface is very difficult to design.

Some improvements also must be made on the knowledge base. We need to study many other user's requirements in order to enhance the present knowledge base. Instead of developing a very general and complete knowledge base, however, we think it is possible to build several knowledge bases for different purposes. An initial knowledge base may be developed for beginners. This would enable the Quiz Apprentice to help the user build basic programs; a lot of processing time will probably be spent in debugging. Another knowledge base may be developed for experts. In this case, the purpose of the Quiz Apprentice would be to help expert programmers be super productive; its main task would be understanding requirements. A specific knowledge base has already been developed; the analyzer of the Quiz Apprentice is presently used in the question-answering system of the **Quiz Advisor**. The knowledge base for this application is designed to work with formal knowledge of Quiz.

Finally, the learning module must be further developed. Though this module is very efficient for the prototype, some limitations will certainly appear when the Quiz Apprentice understands all the knowledge of Quiz. To enhance this module, we could, for instance, investigate causal relationships between a problem and its solution, and attempt to generalize along this axis.

The Quiz Apprentice is not in any sense a complete system. However, we have reached our goal of proving the feasibility of such a system. Our system is fast and efficient, although much work remains to be done before a real Quiz Apprentice can be built.

Appendices

A - Quiz

A.1 - Presentation

Quiz is a report writer developed by Cognos Inc. Its first version was released in 1979 for the Hewlett-Packard HP3000 Series of minicomputers. Quiz is a component of Power-House, a fourth generation application development language for minicomputers.

Quiz is a very versatile software program that is easy to learn and use. At its simplest level, it is a fast on-line reporting system that can be mastered by novice users in a matter of hours. At its most advanced level, it can be used by programmers and experienced users to interactively produce complex reports that involve complex manipulation of data and/or satisfy some complex manipulation requirements.

A.2 - Description of the language

A Quiz program consists of a set of statements that represent a specification of the desired report. Each program starts with an **access** statement and ends with a **go** statement. The order of the statements between **access** and **go** is irrelevant. This reflects the non-procedural nature of Quiz. The statements used for the production of a report are described after this paragraph. We do not give a full description of every statement. We just want to give a technical description of the most useful statements and options.

To specify input files and their logical relationships
ACCESS file1/subfile1[ALIAS name1]
[LINK [item] TO [key OF] file2/subfile2[ALIAS name2]
[OPTIONAL]
[AND/LINK [item] TO [key OF] file3/subfile3[ALIAS name3]
[OPTIONAL]] ...]

LINK specifies that the linkage is hierarchical
AND specifies that the linkage is parallel
OPTIONAL stipulates that the linkage is to continue even if a related record for this file does not exists.

To extract records from the primary file by key value
CHOOSE [key [value [TO value] [value [TO value]] ...]]
CHOOSE key PARM [PROMPT [string] [n TIMES]]

To assign a name to an expression
DEFINE name [type [* n]] [format] =

condition-expr/case-expr/PARM [PROMPT [string] [n TIMES]]

To display a message
DISPLAY string

To specify the content and format of the final footing
FINAL FOOTING [report-group]

[report-group] is described after this list of statements

To specify the content and format of control break footings
FOOTING AT sort-key [report-group]

To initiate the execution of a Quiz report
GO

To specify the content and format of control break headings
HEADING AT sort-key [report-group]

To specify the content and format of the initial heading
INITIAL HEADING [report-group]

To specify the content and format of a page footing
PAGE FOOTING [report-group]

To specify the content and format of a page heading
PAGE HEADING [report-group] [KEEP [COLUMN] [SKIP [n]]]

To specify the contents and format of report detail lines
REPORT [SUMMARY] [report-group [SKIP n]]

SUMMARY states that displayed output is not required. It is typically used with SET SUBFILE

To apply selection conditions to files and record complexes
SELECT [file] [IF condition]

To control output to subfiles
SET SUBFILE [AT sort-key] [NAME name] [KEEP] [APPEND]

To sort record complexes and define control breaks
SORT [ON item [D] [ON item2 [D]] ...]

Report-group
A report-group determines the content and format of details lines, footings, and headings. The general form is :

[report-item] ... [SKIP [n/PAGE]] [RESERVE n [LINES]]

Report-item
A report-item specifies what is to be reported, its position and format, and whether it is to be printed on every line or only at a control-break. The general form of a report-item is :

[SKIP [n/PAGE]][TAB n] item [summary][format][PRINT AT sort-key]

The summary operations are :

AVERAGE [AT sort-key] [RESET AT sort-key]

COUNT [AT sort-key] [INITIAL [VALUE] item] [RESET AT sort-key]

MAXIMUM [AT sort-key] [INITIAL [VALUE] item] [RESET AT sort-key]

MINIMUM [AT sort-key] [INITIAL [VALUE] item] [RESET AT sort-key]

PERCENT item [AT sort-key] [RESET AT sort-key]

RATIO item [AT sort-key] [RESET AT sort-key]

SUBTOTAL [AT sort-key][INITIAL [VALUE] item] [RESET AT sort-key]

B - The Quiz Apprentice code

B.1 - The interpreter

```
:- dynamic quiz/3, exp/0.

/* Main interaction loop for the expert */
expert :- write('Enter passwd'),
          readuser([passwd]),
          assert(exp),
          repeat,
              readuser(L),
              interpret(L),
              display,
              L=[end],
          retract(exp).

/* Main interaction loop for the user */
app :- repeat,
           readuser(L),
           interpret(L),
           display,
           L=[end],
       write('Nice meeting you !').

/* Interpretation of the user requirement */
interpret([end]) :- !.
interpret(L) :- update(L),!.
interpret(L) :- action(L),!.

/* Update of the user program */
update([access|L])        :- clear(user),
                             add(access,L,_,L).
update([define,X|L])      :- add(define,[X|L],[X|_],L).
update([fpass,X|L])       :- add(fpass,[X|L],[X|_],L).
update([select|L])        :- add(select,L,_,L).
update([sort|L])          :- clear(user,heading),
                             clear(user,footing),
                             add(sort,L,_,L).
update([report|L])        :- add(report,L,_,L).
update([heading,at,X|L])  :- add(heading,[at,X|L],[at,X|_],L).
update([footing,at,X|L])  :- add(footing,[at,X|L],[at,X|_],L).
update([set,subfile|L])   :- add(set,[subfile|L],[subfile|_],L).
update([go])              :- clear(user,go),
                             ifsubfile,
                             assert(quiz(user,go,[])),
                             correct,
                             grammar.
```

```
/* Update of the user program, taking care of the */
/*  pseudo-uniqueness                             */
add(K,_,A,[]) :- clear(user,K,A),
                 clear(user,go).
add(K,L,A,_) :- clear(user,X,A),
                clear(user,go),
                assert(quiz(user,K,L)).

/* If the user program contains a set subfile statement, copy */
/*  the program to the subfile                                 */
ifsubfile :- quiz(user,set,L),
             append(_,[name,X|_],L),
             copy(user,X).
ifsubfile :- quiz(user,set,_),
             copy(user,quizwork).
ifsubfile.

/* if the expert requirement is an action, perform the action */
/* if the user requirement is an action and the action is     */
/*  authorized, perform the action                            */
action(L) :- exp, A=..L, A.
action([X|L]) :- actionfilter(X), A=..[X|L], A.
action(_) :- write('non available requirement'), nl.

/* Actions authorized for the user */
actionfilter(generate).
actionfilter(rewrite).
actionfilter(readfile).
actionfilter(expert).
actionfilter(correct).
actionfilter(check).
actionfilter(gram).
```

B.2 - The matcher

```
/* Forward chaining of the rules to correct the user program */
correct :- correct1, correct.
correct.

/* Correction of the program for one rule */
correct1 :- rule(C,A,N),
            try(C),
            writelist(N),
            A.

/* Check the program for any non-Quiz requirement */
check :- check1, fail.
check.

/* Check the program with a single rule */
check1 :- rule(C,_,N),
          try(C),
          writelist(N).

/* Try to match the condition of the rule against the user */
/*  program                                                */
try([]).
try([not,Kw,Li|Rest]) :- \+ try([Kw,Li]),
                         try(Rest).
try([Kw,Li|Rest])     :- quiz(user,Kw,Att),
                         append(_,Scan,Att),
                         match(Li,Scan),
                         try(Rest).

/* Match a list of variables with a list of words */
match([],_).
match([Var,Ty|L],[Word|M]) :- match1([Var,Ty],Word),
                              match(L,M).
match([Cons|L],[Cons|M])   :- match(L,M).

/* Match a single variable taking care of its Quiz type */
match1([X,key],Y)     :- item(Y), X=Y.
match1([X,item],Y)    :- item(Y), X=Y.
match1([X,numeric],Y) :- number(Y), X=Y.
match1([X,summary],Y) :- summary(Y), X=Y.
match1([X,string],Y)  :- string(Y), X=Y.
```

B.3 - The learning

```
/* Learn a "multiple statement" rule */
learn(multiple) :- exp,
                   write('enter the statement'),
                   readuser(P),
                   write('enter the statement'),
                   readuser(Q),
                   multstat(Q,[P],P,[]).

/* In the case of a "multiple statement" rule, prompt the */
/*  expert for the statements and build the rule          */
multstat([],K,_,W) :- reduce(K,[],L,W),
                      makeref(W,R),
                      generalize(L,C,R),
                      write('text for the rule'),
                      readuser(T),
                      replvar(T,N,R),
                      write('action to perform'),
                      read(A),
                      asserta(rule(C,A,N)).
multstat(L,K,U,W) :- write('enter the statement'),
                     readuser(P),
                     setwords(L,U,W,U1,W1),
                     multstat(P,[L|K],U1,W1).

/* Reduce the list of statements according to a set of words */
reduce([],L,L,_).
reduce([[not,X|Y]|K],L,M,W) :- reducelist(Y,Y1,W),
                               reduce(K,[not,X,Y1|L],M,W).
reduce([[X|Y]|K],L,M,W) :- reducelist(Y,Y1,W),
                           reduce(K,[X,Y1|L],M,W).

/* Reduce a list of words according to a set of words */
reducelist([],[],_).
reducelist([at,X|K],[at,X|L],W) :- member(X,W),endreduce(K,L,W).
reducelist([on,X|K],[on,X|L],W) :- member(X,W),endreduce(K,L,W).
reducelist([X|K],[X|L],W) :- X\==at,X\==on,member(X,W),
                             endreduce(K,L,W).
reducelist([_|K],L,W) :- reducelist(K,L,W).

/* Finish the reduction of the list of words */
endreduce([at,X|K],[at,X|L],W) :- member(X,W),endreduce(K,L,W).
endreduce([on,X|K],[on,X|L],W) :- member(X,W),endreduce(K,L,W).
endreduce([X|K],[X|L],W) :- X\==at,X\==on,member(X,W),
                            endreduce(K,L,W).
endreduce(_,[],_).

/* Generalize the condition of the rule according to a */
/*  reference                                          */
generalize([],[],_).
generalize([[X|K]|L],[K1|L1],R) :- replvartype([X|K],K1,R),
                                   generalize(L,L1,R).
generalize([X|L],[X|L1],R) :- generalize(L,L1,R).
```

```
/* Learn a "unique statement" rule */
learn(unique) :- exp,
                 write('enter the statement'),
                 readuser(P),
                 uniquestat(P,[]).

/* In the case of a "unique statement" rule, prompt the */
/*  expert for any equivalent form and create the rules */
uniquestat([],L) :- write('give the Quiz statement'),
                    readuser(R),
                    write('give the text for the rules'),
                    readuser(T),
                    createn(R,L,T).
uniquestat(K,L) :- write('give another equivalent form'),
                   readuser(R),
                   uniquestat(R,[K|L]).

/* Create one rule for every form given */
createn(_,[],_).
createn([X|K],[[X|K1]|L],T) :- change(A,A1,K,K1),
                               inter(A,A1,I),
                               create(X,A1,A,I,T),
                               createn([X|K],L,T).
createn(R,[K|L],T) :- writelist(K),
                      write('Not compatible'),nl,
                      createn(R,L,T).

/* Given a statement, create one rule */
create(X,A,_,[],T) :- makeref(A,R),
                      replvartype(A,K,R),
                      replvar(A,L,R),
                      replvar(T,S,R),
                      asserta(rule([X,K],generate(L),S)).
create(X,A,B,C,T) :- makeref(C,R),
                     replvartype(A,K,R),
                     replvar(A,L,R),
                     replvar(B,M,R),
                     replvar(T,S,R),
                     asserta(rule([X,K],rewrite(L,M),S)).
```

```
/* Build a reference given a set of words */
makeref([X|K],[[X,_,numeric]|L]) :- number(X),makeref(K,L).
makeref([X|K],[[X,_,summary]|L]) :- summary(X),makeref(K,L).
makeref([X|K],[[X,_,string]|L])  :- string(X),makeref(K,L).
makeref([X|K],[[X,_,item]|L])    :- item(X),makeref(K,L).
makeref([_|K],L)                 :- makeref(K,L).

/* Generalize Quiz variables given the reference */
replvar([],[],_).
replvar([X|K],[Y|L],R) :- member([X,Y,_],R),replvar(K,L,R).
replvar([X|K],[X|L],R) :- replvar(K,L,R).

/* Generalize Quiz variables and attach the associated Quiz */
/*  type given the reference                                 */
replvartype([],[],_).
replvartype([at,X|K],[at,Y,key|L],R) :- member([X,Y,item],R),
                                        replvartype(K,L,R).
replvartype([on,X|K],[on,Y,key|L],R) :- member([X,Y,item],R),
                                        replvartype(K,L,R).
replvartype([X|K],[Y,T|L],R) :- member([X,Y,T],R),
                                replvartype(K,L,R).
replvartype([X|K],[X|L],R) :- replvartype(K,L,R).
```

B.4 - The grammar

```
/* Take one statement at a time and analyze it */
gram:- quiz(user,X,Y),analyze(X,Y).
gram.

/* Analyze each statement */
analyze(access,L)  :- analaccess(L),!,fail.
analyze(define,L)  :- analdefine(L),!,fail.
analyze(fpass,L)   :- analfpass(L),!,fail.
analyze(select,L)  :- analselect(L),!,fail.
analyze(sort,L)    :- analsort(L),!,fail.
analyze(heading,L) :- analheading(L),!,fail.
analyze(report,L)  :- analreport(L),!,fail.
analyze(footing,L) :- analfooting(L),!,fail.
analyze(set,L)     :- analset(L),!,fail.
analyze(go,L)      :- analgo(L),!,fail.
analyze(X,L)       :- warn(X,L),!,fail.

/* Analyze the access statement */
analaccess([X]) :- item(X).
analaccess([X,link,to|L]) :- item(X),analaccess(L).
analaccess([X,and,to|L]) :- item(X),analaccess(L).
analaccess(L) :- warn(access,L).

/* Analyze the define statement */
analdefine([X|L]) :- item(X),append(K,[=|K1],L),
                     analdeftype(K),analdefvalue(K1).

/* Analyze the attributes of the defined item */
analdeftype([X|L]) :- type(X),analdefsize(L).
analdeftype(L) :- att(L,[]).
analdefsize([*,X|L]) :- number(X),att(L,[]).
analdefsize(L) :- att(L,[]).
analdefsize(L) :- warn(define,L).

/* Analyze the value of the defined item */
analdefvalue([X]) :- (item(X);number(X);string(X)).
analdefvalue([pack|_]).
analdefvalue([parm,prompt,X]) :- string(X).
analdefvalue([X,if|L]) :- (item(X);number(X);string(X)),
                          append(A,[else|B],L),
                          analcond(A),analdefvalue(B).
analdefvalue([X,if|L]) :- (item(X);number(X);string(X)),
                          analcond(L).
analdefvalue(L) :- warn(define,L).

/* Analyze the fpass statement */
analfpass([X,=,Y,S,at,K]) :- item(X),item(Y),summary(S),item(K).
```

```
/* Analyze the select statement */
analselect([if|L]) :- analcond(L).
analselect(L) :- warn(select,L).

/* Analyze a condition */
analcond(L) :- append(A,[and|B],L),analcond(A),analcond(B).
analcond(L) :- append(A,[or|B],L),analcond(A),analcond(B).
analcond(L) :- append(A,[X|B],L),(X='<';X='>';X='='),
               analexpr(A),analexpr(B).
analcond(L) :- warn(condition,L).
analexpr([X]) :- (item(X);number(X);string(X)).
analexpr(L) :- append(A,[X|B],L),
               (X='+';X='-';X='*';X='/';X=mod),
               analexpr(A),analexpr(B).
analexpr(L) :- warn(expression,L).

/* Analyze a sort statement */
analsort([]).
analsort([on,X,d|L]) :- item(X),analsort(L).
analsort([on,X|L]) :- item(X),analsort(L).
analsort(L) :- warn(sort,L).

/* Analyze a heading statement */
analheading([at,X|L]) :- item(X),analreport(L).
analheading(L) :- warn(heading,L).

/* Analyze a footing statement */
analfooting([at,X|L]) :- item(X),analreport(L).
analfooting(L) :- warn(footing,L).

/* Analyze a report statement */
analreport([]).
analreport([summary|L]) :- analreport(L).
analreport(L) :- analskip(L,M),analreport(M).
analreport(L) :- warn(report,L).

/* Analyze a report-group */
analskip([skip,X|L],M) :- (number(X);X=page),analtab(L,M).
analskip([skip|L],M) :- analtab(L,M).
analskip(L,M) :- analtab(L,M).
analtab([],[]).
analtab([tab,X|L],M) :- number(X),analitem(L,M).
analtab(L,M) :- analitem(L,M).
analitem([X|L],M) :- (item(X);string(X)),analsummary(L,M).
analitem(L,_) :- warn(reportitem,L).
analsummary([X,at,Y|L],M) :- (summary(X);X=print),
                             item(Y),att(L,M).
analsummary([X|L],M) :- summary(X),att(L,M).
analsummary(L,M) :- att(L,M).
```

```
/* Analyze optional attributes of an item */
att([picture,X|L],M) :- string(X),att(L,M).
att([fill,X|L],M) :- string(X),att(L,M).
att([float,X|L],M) :- string(X),att(L,M).
att(L,L).

/* Analyze a set subfile statement */
analset([subfile|L]) :- analsetatt(L).
analset(L) :- warn(setsubfile,L).

/* Analyze attributes of a set subfile statement */
analsetatt([]).
analsetatt([at,X|L]) :- item(X),analsetatt(L).
analsetatt([name,X|L]) :- item(X),analsetatt(L).
analsetatt([keep|L]) :- analsetatt(L).
analsetatt(L) :- warn(setsubfile,L).

/* Analyze the go statement */
analgo([]).
analgo(L) :- warn(go,L).

/* Warn the user */
warn(X,L) :- writelist([syntax,error,in,X,here]),
             writelist(L),clear(user,go).

type(numeric).
type(character).
type(date).
```

B.5 - Input/Output

```
/* The input functions are modified versions of those given */
/*  in [Clocksin, Mellish 81]                               */

/* Read the user requirement */
readuser(P) :- read_in(P,user_input,_),!.

/* Read a file */
readfile(X) :- open(X,read,S),
               repeat,
                   read_in(P,S,A),
                   interpret(P),
                   A= -1,!,
               close(S).

/* Read a list of characters and transform it into */
/*  a list of words                                */
read_in(P,S,A) :- initread(L,A,S),words(P,L),!.

/* Read character by character */
initread(U,A,S) :- get(S,K1),readrest(K1,U,A,S).

readrest(46,[],46,_).
readrest(-1,[],-1,_).
readrest(K,[K|U],A,S) :- get0(S,K1),readrest(K1,U,A,S).

/* Transform the list of characters into a list of words */
words([],[]).
words([X|L],L1) :- word(X,L1,L2),!,blanks(L2,L3),words(L,L3).

word(A,[X|L],L1) :- alphanum(X,X1),!,alphanums(Y,L,L1),
                    name(A,[X1|Y]).
word(A,[X|L],L) :- name(A,[X]).

alphanums([X1|Y],[X|L],L1) :- alphanum(X,X1),!,
                              alphanums(Y,L,L1).
alphanums([],L,L).

alphanum(46,46).
alphanum(K,K)  :- K>33,K<38.
alphanum(K,K)  :- K>47,K<58.
alphanum(K,K)  :- K>93,K<123.
alphanum(K,K1) :- K>64,K<91,K1 is K\/32.

blanks([X|L],L1) :- X=<32,!,blanks(L,L1).
blanks(L,L).
```

```
/* Display the current state of the program in the order */
/*  the Quiz parser treats them                          */
display :- show(access),
           show(define),
           show(select),
           show(sort),
           show(heading),
           show(report),
           show(footing),
           show(set),
           show(go),nl,
           show(fpass),!.

show(X) :- quiz(user,X,L),writelist([X|L]),fail.
show(_).

writelist([]).
writelist([X]) :- write(X),nl,!.
writelist([X|L]) :- write(X),put(32),writelist(L),!.
```

B.6 - Tools

```
/* Clear statements of the user program */
clear(X)     :- clear(X,_,_).
clear(X,Y)   :- clear(X,Y,_).
clear(X,Y,Z) :- allretract(quiz(X,Y,Z)).

allretract(X) :- retract(X),fail.
allretract(_).

/* Copy a program from a file to another */
copy(_,F) :- clear(F),fail.
copy(F1,F2) :- quiz(F1,X,Y),assert(quiz(F2,X,Y)),fail.
copy(_,_).

/* Given a list of words, prompt for a variable name,          */
/*  replace the list by the new name and define the new name */
/*  in the current program                                     */
generate(L) :- write('give a new name to redefine '),
               writelist(L),
               readuser([Z|_]),
               rewrite(L,[Z]),
               newvar(L,Z).

/* Replace everywhere in the current program a sublist */
/*  by another                                         */
rewrite(X,Y) :- quiz(user,C,L),
                change(X,Y,L,L1),
                retract(quiz(user,C,L)),
                interpret([C|L1]),
                rewrite(X,Y).
rewrite(_,_).

/* If the new variable contains a summary, create a fpass */
/*  statement, otherwise a define statement               */
newvar([X,Y,at,K],Z) :- interpret([fpass,Z,=,X,Y,at,K]).
newvar([X,Y],Z) :- summary(Y),
                   lastkey(A),
                   writelist([do,you,want,X,Y,at,A,?]),
                   readuser(P),
                   tofp(P,Z,[X,Y,at,A]).
newvar(K,Z) :- interpret([define,Z,=|K]).

/* If the key for the summary was missing, ask for it */
tofp([yes|_],Z,K) :- interpret([fpass,Z,=|K]).
tofp(_,Z,[X,Y|_]) :- write('give another key'),
                     readuser([A|_]),
                     tofp([yes],Z,[X,Y,at,A]).
```

```
/* Give the lastkey in the sort statement */
lastkey(X) :- quiz(user,sort,L),!,
              reverse(L,M),
              append(_,[X,on|_],M).
lastkey(X) :- writelist([you,need,a,sort,statement]),
              readuser(P),
              interpret(P),
              lastkey(X).

/* Given a list, replace a sublist by another */
change(A,B,[X|K],[X|L]) :- change(A,B,K,L).
change(A,B,K,L) :- append(A,R,K),append(B,R,L).

/* Compute the set of common words by induction */
setwords(L,U,I,Ul,Il) :- union(L,U,Ul),
                         inter(L,U,T),
                         union(I,T,Il).

/* Well known tools ... */
reverse(A,B) :- reverse(A,[],B).
reverse([],A,A).
reverse([X|K],L,M) :- reverse(K,[X|L],M).

append([],L,L).
append([X|K],L,[X|M]) :- append(K,L,M).

member(X,L) :- append(_,[X|_],L).

inter([],_,[]).
inter([X|K],L,[X|M]) :- member(X,L),inter(K,L,M).
inter([_|K],L,M) :- inter(K,L,M).

union([],L,L).
union([X|K],L,M) :- member(X,L),union(K,L,M).
union([X|K],L,[X|M]) :- union(K,L,M).

/* Prolog definition of Quiz types */
item(X) :- \+ reserved(X),X @>= a,X @< zzz.

string(X) :- name(X,[34|_]).

summary(count).
summary(subtotal).
summary(maximum).
summary(minimum).
summary(average).

reserved(at).
reserved(on).
reserved(if).
reserved(parm).
reserved(prompt).
reserved(skip).
reserved(tab).
```

C - Knowledge base

In this section, we give the typical user question, the associated requirement and the rule which corrects the requirement. For some questions, several rules have been written to correct similar cases.

```
How do I divide a subtotal ?
report item1 subtotal / item2
rule([report,[X,item,Y,summary,/,Z,item]],
     generate([X,Y,/,Z]),
     [no,operation,in,a,report])

How do I select with a subtotal ?
select if item subtotal < 1000
rule([define,[X,item,Y,summary]],
     generate([X,Y]),
     [no,Y,in,a,define])
rule([define,[X,item,Y,summary,at,Z,key]],
     generate([X,Y,at,Z]),
     [no,Y,in,a,define])
rule([select,[X,item,Y,summary]],
     generate([X,Y]),
     [no,Y,in,a,select])
rule([select,[X,item,Y,summary,at,Z,key]],
     generate([X,Y,at,Z]),
     [no,Y,in,a,select])
rule([sort,[X,item,Y,summary]],
     generate([X,Y]),
     [no,Y,in,a,sort])
rule([sort,[X,item,Y,summary,at,Z,key]],
     generate([X,Y,at,Z]),
     [no,Y,in,a,sort])
rule([heading,[X,item,Y,summary]],
     generate([X,Y]),
     [no,Y,in,a,heading])
rule([heading,[X,item,Y,summary,at,Z,key]],
     generate([X,Y,at,Z]),
     [no,Y,in,a,heading])

How do I report an item without the sign ?
report item without sign
rule([report,[X,item,without,sign]],
     generate([X,without,sign]),
     [use,an,absolute,in,a,define])
rule([define,[X,item,without,sign]],
     rewrite([X,whithout,sign],[absolute,X]),
     [use,absolute,function])

How do I select when an item matches another one ?
select if item1 match item2
rule([select,[X,item,match,Y,item]],
     rewrite([X,match,Y],[matchpattern,X,Y]),
     [use,matchpattern,function])
```

```
Why my headings are not reported to the subfile ?
heading at key1
report summary item
set subfile at key3
rule([heading,[],set,[],report,[summary]],
     clear(user,heading),
     [no,heading,in,a,subfile])
rule([footing,[],set,[],report,[summary]],
     clear(user,footing),
     [no,footing,in,a,subfile])

How do I link a subfile to another subfile ?
access subfile f1 link to subfile f2
rule([access,[subfile,X,item,link,to,subfile,Y,item],
     rewrite([subfile,X,link,to,subfile,Y],
             [X,link,to,record,1,of,Y]),
     [subfiles,must,be,linked,thru,records])

How do I report an item with a % sign after ?
report item with % after.
rule([report,[with,%,after]],
     rewrite([with,%,after],[picture,"^^^^^^^^^%"]),
     [put,%,in,the,picture,of,the,item])

How do I select every 10 th item ?
select every 10 th item
rule([select,[every,X,numeric,th,Y,item]],
     rewrite([every,X,th,Y],[Y,count,mod,X,=,0]),
     [rewrite,the,expression,every])
rule([select,[every,X,numeric,Y,item]],
     rewrite([every,X,Y],[Y,count,mod,X,=,0]),
     [rewrite,the,expression,every])
rule([select,[each,X,numeric,th,Y,item]],
     rewrite([each,X,th,Y],[Y,count,mod,X,=,0]),
     [rewrite,the,expression,each])

Why nothing is going to my subfile ?
[footing at key item]
set subfile at key1
rule([set,[],not,report,[]],
     interpret([set,subfile]),
     [only,items,in,report,go,to,the,subfile])

How do I select when no records exists in a file ?
select if no records exists in file
rule([select,[no,records,exists,in,X,item]],
     rewrite([no,records,exists,in,X],[not,record,X,exists]),
     [rewrite,no,records,expression])
rule([define,[no,records,exists,in,X,item]],
     rewrite([no,records,exists,in,X],[not,record,X,exists]),
     [rewrite,no,records,expression])
```

```
How do I keep a subfile permanent ?
set subfile keep permanent
rule([set,[keep,permanent]],
     rewrite([keep,permanent],[keep]),
     [keep,implies,permanent])

How do I report an item with leading zeroes ?
report item with leading zeroes
rule([report,[X,item,with,leading,zeroes]],
     rewrite([X,with,leading,zeroes],[X,significance,6]),
     [use,significance,to,have,zeroes])

How do I select with a parm prompt ?
select if item = parm prompt "enter item"
rule([select,[parm,prompt,X,string]],
     generate([parm,prompt,X]),
     [parm,must,be,in,a,define])

How do I select duplicate items ?
select if duplicate items
rule([select,[duplicate,X,item]],
     rewrite([duplicate,X],[X,count,>,1]),
     [you,must,count,to,have,duplicate])

How do I access a file several times ?
access file1 link to file1
rule([access,[X,item,link,to,X,item]],
     rewrite([X,link,to,X],[X,link,to,temp,alias,X]),
     [you,need,an,aliasing])

What is the default key for a subtotal ?
report item subtotal
rule([report,[X,item,Y,summary]],
     write('The default sort key is the last key'),
     [])

How do I report an item on a condition ?
report item if condition
rule([report,[X,item,if,Y,rest]],
     generate([X,if,Y]),
     [no,condition,in,report])
```

```
How do I report at control break ?
report item print at key
rule([heading,[at,X,key]],not,sort,[on,X,key]],
     (show(sort),readuser(L),interpret(L)),
     [X,has,to,be,a,sort,key])
rule([report,[at,X,key]],not,sort,[on,X,key]],
     (show(sort),readuser(L),interpret(L)),
     [X,has,to,be,a,sort,key])
rule([footing,[at,X,key]],not,sort,[on,X,key]],
     (show(sort),readuser(L),interpret(L)),
     [X,has,to,be,a,sort,key])
rule([set,[at,X,key]],not,sort,[on,X,key]],
     (show(sort),readuser(L),interpret(L)),
     [X,has,to,be,a,sort,key])

How do I sort on the first letter of an item ?
sort on first letter of item
rule([sort,[first,letter,of,X,item]]
     generate([first,letter,of,X]),
     [use,extract,in,a,define])
rule([define,[first,letter,of,X,item]]
     rewrite([first,letter,of,X],[X,'[',1,:,1,']']),
     [use,extract])

Why all my records are not reported ?
access file1 link to file2
define item = value if not record in file2 exists
rule([define,[not,record],not,access,[optional]],
     write('Need optional in access'),
     [])
rule([select,[not,record],not,access,[optional]],
     write('Need optional in access'),
     [])

How do I select if a string is in another string ?
select if string1 in string2
rule([select,[X,item,in,Y,item]],
     rewrite([X,in,Y],[0,>,index,Y,X]),
     [use,index,for,strings])

How do I report an item in column 5 ?
report item in column 5
rule([report,[X,item,in,column,Y,numeric]],
     rewrite([X,in,column,Y],[tab,Y,X]),
     [use,tab,to,report,at,a,fixed,place])
```

Bibliography

[Clocksin, Mellish 81]
Clocksin, W.F., Mellish, C.S., Programming in Prolog, Springer-Verlag Berlin Heidelberg, 1981.

[Haas, Hendrix 83]
Haas, N., Hendrix, G.G., Learning by being told: Acquiring knowledge for information management, in Machine Learning, Michalski, ed., pp. 405-428, 1983.

[Mitchell et. al. 85]
Mitchell, T.M., Mahadevan, S., Steinberg, L.I., LEAP: A Learning Apprentice for VLSI Design, procs. IJCAI'85, pp. 573-580, 1985.

[Quiz 85]Quiz version 5.01, User's Guide, Cognos Inc., 1985.

[Rich, Schrobe 78]
Rich, C., Schrobe, H.E., Initial Report on a LISP Programmer's Apprentice, in Interactive Programming Environment, Barstow, ed., pp. 443-463, 1984.

[Rich, Schrobe 79]
Rich, C., Schrobe, H.E., Design of a Programmer's Apprentice, in Artificial Intelligence: An MIT Perspective, Winston, ed., pp. 137-173, 1979.

[Waters 82]
Waters, R.C., The Programmer's Apprentice: Knowledge Based Program Editing, in Interactive Programming Environment, Barstow, ed., pp. 464-486, 1984.

[Waters 86]
Waters, R.C., KBEmacs: Where's the AI ?, The AI Magazine, pp. 47-56, Apr 1986.

[Winston 84]
Winston, P.H., Artificial Intelligence, second edition, Addison-Wesley, 1984.

[Winston 86]
Winston, P.H., Learning by augmenting rules and accumulating censors, in Machine Learning, second edition, Michalski, ed., pp. 45-61, 1986.