ADVISOR

An Expert System Shell

Written in Prolog.

A Dissertation Presented in Fulfilment

of the requirement for the M.Sc Degree

in Computer Applications.

August 1990

Paul Powell B.Sc.

School of Computer Applications

Dublin City University

Dublin 9.

Supervisor : Mr M. Ryan.

DECLARATION

This dissertation is based on the author's own work. It has not previously been submitted for a degree at any academic institution.

Paul Powell

Paul Powell
August 1990.

ACKNOWLEDGEMENTS

I would like to thank Polydata Ireland Ltd., for their generous scholarship which made it possible for me to pursue this research. I would like to thank the staff at Polydata, especially Mr Tadgh Dennahy and Dr. Irwin Bauer for their input in building up a prototype rule set. I would also like to thank my supervisor, Mr. Michael Ryan for his support, direction and patience. Finally I would like to acknowledge the support of my colleagues, friends and family who have kept me sane.

Table of Contents.

Table of Contents.

List of Figures

Abstract

The purpose of the research reported here is to investigate the requirements of a system which will aid a designer in moving from the functional specification of a new product to a detailed technical specification of the polymer required, and to develop a system for this purpose. A very wide range of products can be made from polymers and over 5000 polymers are commercially available. The designer usually interacts with a polymer expert and together they try to specify the polymer properties required. Over 150 parameters are involved. Once values have been assigned to these parameters existing databases can be used to identify a suitable material. Polymer experts are scarce and there is a need for a system which will allow the designer to specify the parameter values dictated by the functional requirements of the design.

A number of problems are identified in this area. Some of these are of a terminological and communication nature and arise from the wide range of the application areas and large numbers of parameters to be specified. Others are due to the knowledge engineering problems in formalising the knowledge of the polymer expert.

An investigation of these problems lead to the specification of a system which combines a flexible quasi-English natural language with a rule based paradigm. A prototype system called Advisor is built. Advisor has the unique ability of allowing the expert to create not only the rules but also the vocabulary with which these rules are constructed and with which the designer builds up a description of the application. Thus all the vocabulary used within the system is familiar to both the expert and the designer. The underlying language for the implementation of Advisor is Arity Prolog.

The prototype validates the basic design decisions. Analysis of it's performance and suggestions for further refinements and improvements are given. The prototype system which is being put into use by a commercial plastic design company is currently being evaluated by polymer design experts in Aachen in Germany.

Chapter 1

The Problem Domain

Chapter 1 The problem domain.

1.1. The Design process.

Many aspects of design are difficult to support using computers because of the vagueness and ambiguity of the concepts and the creative nature of the process involved in transforming them into the detailed specifications necessary to select suitable materials. This problem is heightened in the area of design using polymers where a wide range of applications, everything from forks and other disposable cutlery to bumpers for cars, can be produced. Each application produced can be constructed from a wide range of different polymer materials. Over five thousand grades of polymers are commercially available, each of which has a large number of material properties.

Apart from the difficulty of specifying the design objectives in a clear and unambiguous manner, design using polymers is complicated because the properties they have are so numerous and varied. Indeed Lovrich and Tucker [LT861] state that "Polymers offer a wider range of material properties than any other class of materials; it is this range that makes them suitable for so many applications". However they go on to say that "Expert designers with years of experience are quite capable of making good design decisions..." but the problem is that "these specialists make up a very small percentage of the designers working with polymers".

An example of the wide range of properties is given by Charles MacDermott [CD79]. "A perfect example of the proliferation of candidate materials is the nylon family. Within this family, there are 6, 6/6, 6/10, 6/12 nylons as well as numerous copolymers of these basic polyamides. Furthermore, most of them are available in glass-reinforced, flame-retarded, mineral-filled, etc., versions. To this must be added nylons blended or grafted with

toughening agents, such as elastomers and polyesters, and again there are filled and glass-reinforced versions of these nylons".

As can be seen from this example of the nylon family, design experts must have an engineering background which enables them to understand the polymer properties and how polymers can be affected by their functionality and their environment. They must also know about the various processing techniques which can be used to manufacture the application. However the majority of designers are either being introduced to the area of polymer design having previously designed in more conventional material (e.g. steel, wood etc.) or are designers with no engineering background at all. Most material designers faced with the myriad of possible polymers tend just to design in the particular materials with which they are familiar. This can mean that they miss out on polymers which have additional desirable properties to the ones which are essential to the design, or which are more cost effective.

The ideas expressed by Lovrich and Tucker [LT861,LT862] were given further support in the course of the work described here and through meetings with polymer design experts from Aachen, Germany. It was discovered from these meetings that most designers would typically have no idea of the properties that polymers possess. Designers think in terms of the functionality, geometry and the appearance of an application. They use terms which describe the shape of an object, the environment in which it is to be used. The ordinary designer has to convey these ideas to an expert in polymer design who must study them to discern what characteristics the suitable polymer should have. In the past the polymer expert would then have consulted brochures and polymer specification sheets which have been compiled and distributed by producing companies. He would then extract a few suitable polymers and report his findings to the designer.

In recent years the problem of actually choosing a suitable polymer based on a

detailed technical profile of the polymer has been computerised by Polydata Ireland Ltd., who are the sponsors of this research. Polydata have developed a database system called CAPS (Computer Aided Polymer Selection) which contains an extensive and constantly updated list of known polymers together with all the technical properties they possess. An experienced polymer designer can submit a detailed technical specification to the CAPS system and the system will speedily return a list of all the qualifying grades of polymers which meet this specification, together with the name of the manufacturer of the polymer and other important information.

The problem still remains of identifying appropriate technical specification for use as input to CAPS. This is made worse by the shortage of technical expertise in the area coupled with designers' inexperience in designing applications using polymers. As it stands the designer produces a functional description of the application to be designed. Within this description would be terms which would describe such attributes as the operating environment, the geometry of the object, the expected working life of the application, it's functionality, etc. This information is then considered by the polymer design expert, who has a polymer engineering background as well as a knowledge of design concepts. The polymer expert produces a technical specification for this application. This in turn leads to a suitable plastic being selected. This selection process is carried out with the aid of a database system of known polymers and their properties (such as C.A.P.S which is used here) or else by considering other information which the expert has compiled over time.

In response to a polymer specification the CAPS system returns with a list of suitable polymers together with the supplier names and the values for the properties that the polymer possesses. The whole design selection process is outlined in Figure 1 and as can be seen a number of intermediary steps may be involved between the design stage and the selection stage.

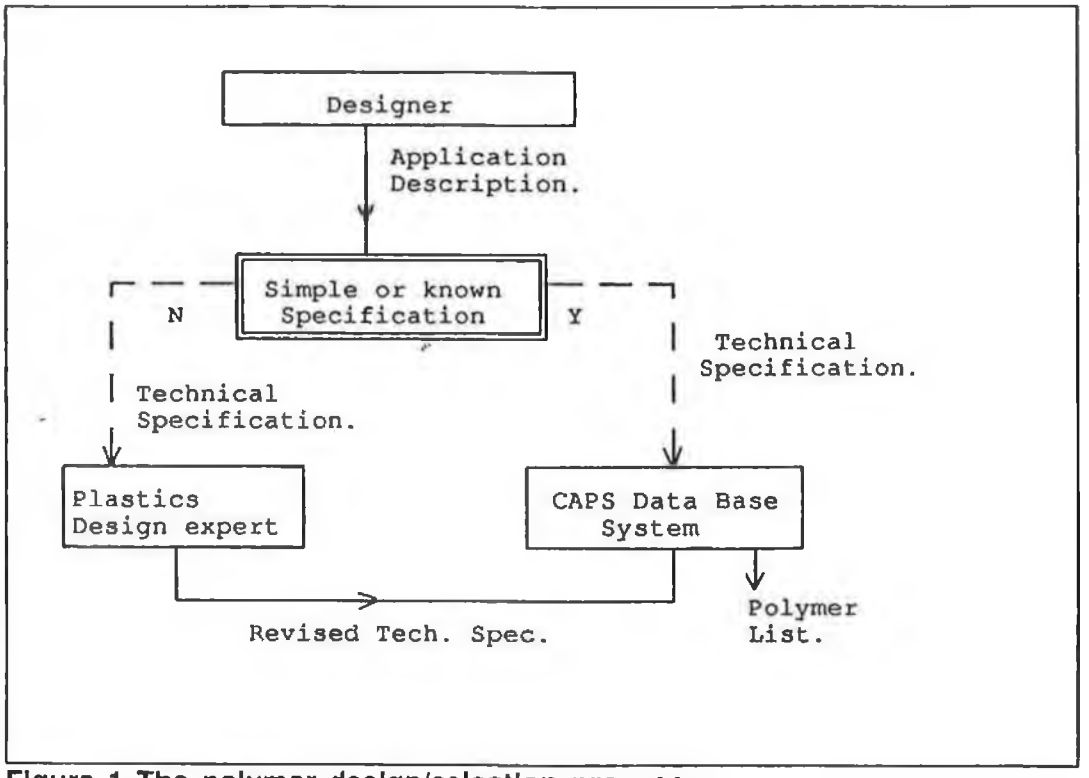

Figure 1 The polymer design/selection process.

1.2. The Aim of the research.

The aim of the research reported here is to explore the possibility of building a flexible, user friendly computer system to

(i) Enable an inexperienced polymer designer to specify the important aspects of a design in unambiguous terms which are familiar to him/her and
(ii) based on this description to help identify the required technical polymer properties together with appropriate range values.

Chapter 2

The Foreseen Problems

Chapter 2 The foreseen problems .

In considering a computer system to aid in the process of polymer specification a number of problems arise.

2.1. The knowledge engineering problem.

Formalising the knowledge of the expert poses difficulties. An important issue is the loss of knowledge due to the third party intervention of the knowledge engineer as a link between the expert and the computer system. For this reason a close interaction between the expert and the computer system is sought. Ideally the human knowledge engineer should be replaced by the computer system providing tools which are easy to understand and use. This would allow the polymer expert to program the system directly and transform the design/selection process shown in Figure 1 into a simplified version of the design/selection process as outlined in Figure 2.

The problem of knowledge engineering has been recognised since the early days of expert systems. The three main topics within knowledge engineering are knowledge acquisition, communication skills, and a sound understanding of the available programming tools. The classical definition of knowledge engineering is "the process of working with an expert to map what he or she knows into a form suitable for an expert system to use..."[BS84]. In the context of the system developed here it is the mapping of the experts knowledge in moving from a general functional description of an application onto a technical specification necessary to select a suitable polymer.

In order for the knowledge engineer to carry out his task he must become familiar

enough with the terminology and structure of the subject matter or domain to ensure that his or her questions directed at the expert are meaningful and relevant. The knowledge engineering process had to be performed during the development of the Advisor prototype. This involved meetings with polymer design experts from Aachen in Germany who were working in partnership with the Polydata team in Pearse St. Enterprise Centre, Dublin. The reasons for working with these experts from Aachen were not only that they were readily available for consultation but also because according to plastics engineers in Polydata there are very few such people in Ireland. It was also mentioned that companies would not readily commit their expert resources or indeed would not be willing to give information due to time, monetary and security reasons.

2.2. The Communication problem.

Two different types of people are involved in the process of design. The designer has a clear idea of what has to be designed, but needs help in selecting the best polymer for the design. The expert in polymer design has a good knowledge of the relationship between the polymer properties and particular aspects of a design. The problem which then exists is one of accurate communication between the designer who thinks strictly in design concepts and the expert who thinks in terms of the effects certain design characteristics have on the technical properties of a polymer. The parts both the expert and the designer play in the selection process can be seen in Figure 1.

The vagueness of the designer's description can lead to ambiguity because of the different terms used by different designers to describe similar concepts. Added to this is the problem of the designer not knowing what the various polymer properties mean and the effects the values assigned to them could have on the designed application.

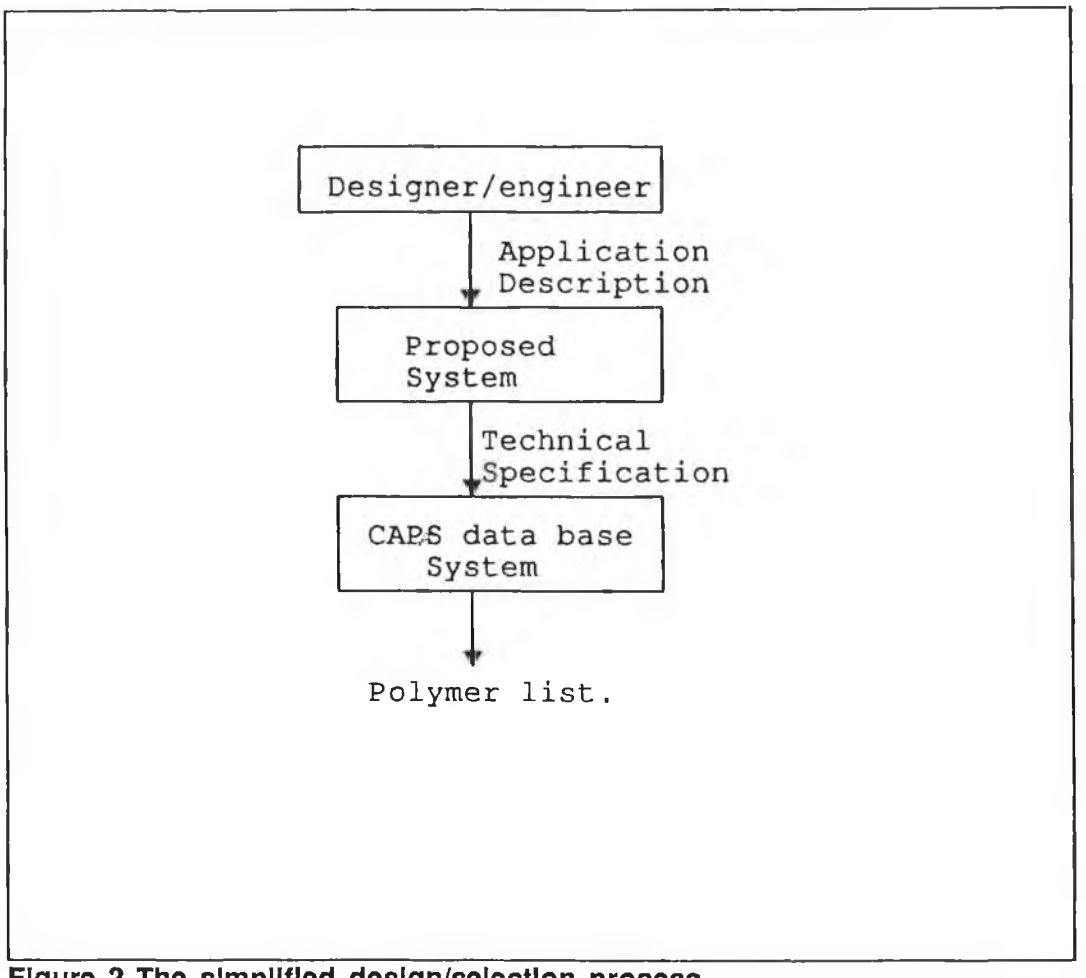

Figure 2 The simplified design/selection process

The polymer design experts visited Polydata periodically but their time was scarce as they were involved in other projects which Polydata were conducting at the time. This is one of the common problems in knowledge engineering- the availability of the expert is always restricted. However it was possible to meet them on enough occasions to allow their expertise in relation to the design of one class of product to be formalised. The chosen product class was what they referred to as "dishes", though in fact we would probably use the word "crockery", as the objects include all types of cups, saucers, plates and other eating devices. This difference in terminology further highlights one of the other issues which one is faced with when attempting the task of knowledge engineering. That is the problem of communication. Sometimes not only a technical language but also a natural

language barrier exists between the expert and the knowledge engineer. This can lead to much confusion and wasted development time.

From this example of crockery developed during these meetings 20 rules were extracted (See Figure 3) which could be coded and entered into a system prototype.

```
if apptype is dishes then scratch_res is yes and chemical_res is
detergents and food_approved is yes and chemical_res is
boiling_water and chemical_res is alcohol and antistatic is yes and
flame_retardant is yes .
if price is low and transparancy is yes then material is ps .
if apptype is dishes and instance is glass and usage is disposable
then heat_destortion > 60 and price is low .
if apptype is dishes and hardness is high then hardness > 26 and
hardness < 27 .
if hardness is high then hardness >= 20 and hardness <= 30 .
if apptype is dishes and used_in is microwave then microwave_res
is yes.
if apptype is dishes and thickness is thin then viscosity is low .
if apptype is dishes and usage is disposable then thickness is thin
.
if apptype is dishes and environment is petrol_station then
chemical_res is oil .
if apptype is dishes and environment is ship then density < 1 .
if apptype is dishes and temperature is low then impact_strength
is high.
if apptype is dishes and country is america then fda_test is yes .
if apptype is dishes and instance is glass then transparancy is yes
.
if apptype is dishes and specific_properties is dishes then
heat_destortion >= 100 .
if apptype is dishes and instance is cup then chemical_res is
alcohol .
if apptype is dishes and instance is glass then chemical_res is
alcohol .
if instance is plate then hardness is high .
if material is ps then hardness is high .
if instance is glass then high_gloss is yes .
if apptype is dishes and instance is glass then unfilled is yes .
```

Figure 3. The prototype Rule Set.

2.3. Knowledge acquisition.

One of the key issues within the field of knowledge engineering is knowledge acquisition. Knowledge acquisition is " the transfer and transformation of the problem-solving expertise from some knowledge source to a program. There are many sources which might be turned to, including human experts, textbooks, data bases..." [BS84].

It was necessary prior to talking to the experts, to research extensively material which would serve to identify some of the questions which need to be asked in order to begin the selection process, to become familiar with some of the technical terms used by polymer engineers and to identify the key issues which need to be addressed in reaching a technical specification. The key issues as identified by experts in filling in polymer design specifications (notably Charles P. MacDermott [CM82]) are:

General information

What is the function of the part?
How does the assembly operate?
Are there space and weight limitations?
What is the required service life?
Can several functions be combined in a single part?
Can the assembly be simplified?
What are the consequences of part failure?

Codes and specifications

Are acceptance codes (e.g. Society of Automotive Engineers, etc) required?

Environmental Considerations

What is the operating temperature?
Is the application exposed to sunlight and weathering?
Is there a chemical environment?
Is the application affected by humidity?

Costs

What are the cost and pricing limitations of the part?

Mechanical considerations

How is the part stressed in service?
What is the magnitude of the stress?
What is the stress versus time relationship?
What is the maximum deformation that can be tolerated?
What are the affects of friction and wear?
What tolerances are required?

Electric Considerations

What type of Voltage is the part to be subjected to?
What are the insulation requirements?

Appearance Considerations

What is the style of the part?
What is the shape of the part?
Does the part need to be coloured?
Does the part need a special surface finish?

This served as a basic knowledge for interviewing the expert with a view to reaching a set of rules on which a prototype system could be based. It was also beneficial to examine the selection process used by the CAPS polymer database system. This proved to be useful in identifying technical terms used by the expert and the kind of realistic values which could be assigned to them. This was done by sitting down with the expert while he used the system to build up a specification and selected suitable polymers for the chosen prototype example of crockery. It was also necessary to examine the database system from the point of view that the terms used would form a major part of the vocabulary provided for use within the developed prototype. These terms also form the link between the system developed and the CAPS database system.

While it was beneficial during the early stages to carry out the knowledge engineering process, I felt it was only useful in that it enabled me to identify the key issues involved in the domain. Reading around the subject matter enabled me to talk to the expert in his own terms and also to gather enough information to construct a good working prototype system.

2.4. The desired system qualities.

It was clear from an early stage that the approach required for building the system would have to be along "Expert System Shell" lines. This is because expert system shells are most suitable for encoding the expert's knowledge and flexible enough to allow this to be done iteratively. Also an expert system approach usually provides an inference mechanism with which to manipulate the expert's knowledge in an efficient and productive way.

It was the intention from the early stages of discussion with the expert to construct the system in such a way as would allow the expert to input their knowledge themselves at later stages in the development. This is in keeping with a common view which is that knowledge engineering is a thing of the past and the trend is moving towards a closer interaction between the expert and the expert system shell. Indeed Shaw and Gaines [SG86] believe that " The role of the knowledge engineer as an intermediary between the expert and the technology is being questioned not only on the cost grounds but also in relation to its effectiveness- knowledge may be lost through the intermediary and the expert's lack of knowledge of the technology may be less of a detriment than the knowledge engineer's lack of domain knowledge ".

Information must also be elicited from the product designer. There are a number of design decisions which are essential to building a complete functional specification of an application to be designed. Because the polymer expert uses this functional description to fill in the values of the polymer properties which are effected by these design considerations they should be as complete as possible. However the designer according to the expert frequently omits some of the essential characteristics of the design due to ignorance of their importance or negligence. Hence a problem exists in that a designer might not provide sufficient design information to enable an expert to build up a technical

specification and thus select a suitable polymer. So a need for questioning the designer about certain aspects of the design arises. The reverse is also true in that the expert might forget to include some important functional aspect of the design which the designer might find important. This leads to a need for the system to allow the polymer expert to iteratively redefine the expert system to be used by the designer.

If the designer can see the reasoning behind each fact established by the system it will enable him/her to work more efficiently with the system the next time it is used. Therefore the need exists to explain how and why a certain polymer specification list has been established for a given functional description. Also a need exists to explain the significance of a certain value being assigned to a certain polymer property in the final specification reached.

Chapter 3

The Current Expert System Technology

Chapter 3 The current expert system technology.

In seeking solutions to the problems vagueness, accurate communication, formalising the expert's knowledge, explanation and questioning identified in Chapter 2, existing literature and commercially available software in the area of expert systems were extensively researched. These included the concepts of user interface design; the important issues in current expert system technology (CEST) such as explanation facilities and questioning facilities; fast prototyping languages for expert systems development i.e. Lisp and Prolog; commercially available PC based expert system shells i.e. Crystal, Leonardo, Goldworks, Nexpert Object etc; and currently available expert systems in polymer selection.

3.1. Considerations for PC-based expert system shells.

The decision of implementing an expert system shell on a PC rather than on a large main frame or mini computer is governed by the available resources of a large number of small to medium sized companies involved in polymer design. A number of issues need to be considered when addressing the subject of PC-based expert system shells and existing expert system technology in the field of polymer design.

1. Processing power and speed.

These two commodities are limited in common PC technology and systems should provide speedy solutions to the type of non-standard problems that expert systems are built to deal with.

2. Finding a suitable shell.

Many expert system shells are available on the market today and each one has it's own special features which might make it more suitable to a particular domain.

3. The size of the Knowledge Base supported.

The knowledge base is one of the key components in any expert system and as such it should be easy to create and amend the knowledge base using the shell. The system should allow programming and maintenance of large knowledge bases efficiently and easily.

4. User interface.

The tools or functions provided by the system should be clearly presented to the user of the shell in order to be easy to use and the interface should cater for any unpredictable input from the user.

5. Training requirements.

Again shells should be judged on the level of training necessary to become proficient enough to construct an expert system using the shell. Documentation on the product should be extensive, clear and well organised.

6. Connectivity.

The shell should be able to interface with languages and/or database technology which can provide features which the shell itself cannot provide which will allow the maximum amount of flexibility in the system.

3.2. Existing PC-based expert system technology.

" Hitherto the PC has not been widely regarded as a powerful enough platform to develop and deliver practical expert systems"[DT87]. However this is now changing with advances being made in the software and hardware associated with PCs. As an example of the advances in software especially AI software (which in the past has been slow and cumbersome), programs developed in Arity Prolog can be compiled to give standard executable code for IBM PCs and compatibles which can run as fast as compiled applications in a standard programming language such as C. Lisp is now available on PCs where previously it was only available on mini or main frame computers or special Lisp work stations.[1]

In addition to this hardware on PCs is improving all the time. The Intel 80486 chip is now available together with expanded memory capabilities of up to 64 megabytes of RAM (e.g. Olivetti EISA CP486). Disk sizes up to 600 megabytes and optical media are also available for storage of large amounts of data. These advances are reflected in the myriad of commercially available PC-based shells available on the software market today (see [HMM87]) of varying degrees of complexity.

One approach to producing expert systems using current expert system technology is to buy a shell off the shelf. This has the advantage of allowing quick development, provided the shell fits the requirements of the problem domain and the shell is well written and easy to use. But it can be expensive depending on the level of complexity of the system and the shell. Typically shells which provide powerful AI techniques are at the high end (in terms of expense and complexity) of the market. These are usually hybrid or mixed

---

[1] However it is still necessary to have a Lisp environment in order to execute Lisp programs.

paradigm systems which can combine rules with data types such as frames, object-oriented programming facilities, and include properties such as inheritance. These shells are in the form of a library of tools and require a lot of time and effort to learn to use effectively. These of course are unsuitable for the polymer expert to use as some prior knowledge of AI techniques and programming would be necessary. Examples of such shells would include GOLDWORKS [GW89,RH90], KEE [ABCS89,WS89] and Nexpert Object. At the low end of the market there are the stand-alone shells which usually consist of one large software program. They are easier to use, but usually do not possess the powerful AI techniques mentioned above. These are usually simple rule based systems which support backward or forward chaining. Examples of these types of shell can be found in the likes of CRYSTAL, LEONARDO (see [BG89] for a review of these two in light of a system developed in both, see also RW88). The packages at the lower end of the market seem to be geared towards a particular domain and do not achieve the domain independence which has been a desired and expected property of current expert system technology. This is treated by David Tong in [DT87] who discusses an approach for the initial selection of a shell based on matching problem requirements with the shell features.

The ability to build and maintain a knowledge based system is an important consideration when evaluating PC-based expert systems shells. David Tong [DT89] says some "PCs currently are limited in random-access memory (RAM) so that if the ES shell requires the entire knowledge base to reside in RAM, it's size will be severely restricted". But he goes on to say that "Other shells have adopted an architecture whereby the knowledge base is modularised and kept mostly in mass storage. Knowledge modules are loaded into RAM only when needed during inference. In this way, very large knowledge bases can be accommodated". Modularization of knowledge bases should be encouraged as it prompts structured development. Similar to large conventional programs, large knowledge bases should be designed and developed in blocks so that testing and validation are manageable. This idea should be incorporated into the proposed system. The user should

be encouraged to build and save modules of rules, which can be loaded into the rule base.

The alternative to buying off the shelf products is to develop a shell either in a conventional programming language such as C, Pascal, etc. or a recognised AI programming language such as Prolog or Lisp. This approach is taken by companies which have the programming skill and experience to build such systems and in such cases it can be cheaper than buying and researching commercially available shells. It has the advantage of being geared specifically towards the task for which it is developed. It also has the added advantage that the person or persons creating the system will have a ready understanding of the problem domain having developed systems in that domain before.

There are expert systems available which deal with certain aspects of polymer selection and processing [LT861, LT862 MW86, PP86]. But they suffer from the fact that (as in the case of [LT861] above) the proposed system is implemented on a mini computer (OPS5 on a VAX 11-750) and mini computers are not within the price range of most small companies. They also suffer from the fact that they are question based and only allow yes or no answers (as in the case of [MW86] above). Most expert systems are based on the question and answer concept where the user is usually presented with a question in the form of text and is usually prompted to answer yes or no. Excessive questioning can alienate the user of an expert system as the user very quickly gets lost in the vast amount of text being presented.

The currently available expert system shells and expert systems in polymer design could not be used due to various deficiencies which they were found to have in the context of the problems outlined in Chapter 2 :

1. In most cases it is necessary to learn a predefined language in order to program the shell efficiently. Examples of this would be Goldworks, Crystal, Leonardo, and Nexpert Object.

It would be unacceptable for an expert to have to learn such skills in order to create an expert system as he/she has little experience in the use of computers.

2. Some of the tools provided within the shell can be quite cumbersome to use as is the case with using the rule editor in Leonardo, Crystal, and Nexpert Object.

3. Crystal was not readily available at the time of starting this research.

4. Goldworks suffers from the problems that firstly it is necessary to know the underlying language Lisp in order to program it effectively and secondly a PC with expanded memory and large amounts of disk space is needed to use it. Thus it is not applicable to most PCs on the market today.

3.3. Constructing an open-ended Expert system shell.

Because the available expert system shell technology is really geared to be used by people experienced in computer programming it was decided to adopt an alternative approach and to construct a system which would provide a suitable user interface and aids to enable the polymer expert to program the system directly.

In order to make the proposed system more accessible to the expert the main feature of the system is a unique programmable user interface which is formed using a combination of a simple quasi-English natural language interface, which is easily programmed by the expert using terms which are familiar to both the designer and the expert, together with a clear and concise graphical representation for this interface.

A rule based paradigm is chosen as the representation of the expert's knowledge (i.e the knowledge base) due to the fact that it provides a ready source of explanation and is easy for the programmer of the rule base (the polymer expert) and the user of the eventual expert system (the designer) to understand and use.

3.3.1 The knowledge Base

The central part of any expert system is the knowledge base. There are many different types of knowledge base representation mechanisms including frames, object oriented mechanisms and "If Then" rule structures. There are a number of attributes which a knowledge base representation should possess, among these are :

1. The knowledge base should be easily maintainable. The very nature of an expert system allows for the fact that the information contained in the knowledge base may not be correct and may be amended many times before the system is stable. Thus it should be easy for a person to add and retract information in the system without

knock on effects to the current information content of the knowledge base. This can only be achieved if the knowledge base is structured in such a way that each unit of knowledge is totally independent of other units.

2. Leading on from this last point the knowledge representation formalism should allow for incremental growth. There should be no detrimental effects in adding new pieces of knowledge as they are discovered in the domain in question.

3. The knowledge in the knowledge base should be readable. It should be possible for a human (e.g. the expert) to read through the knowledge base to check it's correctness. Also if a knowledge base is inherently understandable it can be used as an explanation of the systems actions.

4. The execution speed of the knowledge base is important. The user of the expert system should not have to wait for long lengths of time before the expert system responds to a query.

5. The system should be predictable in that it should give the right result when given the right information. This can be best achieved if the knowledge is constructed on sound proven principles as is the case with logic rule based systems as will be shown.

Rules are by far the most popular form of knowledge representation, especially for PC-based expert systems (See survey results in [HMM87] p208 - p215). They have the advantage of being easy to read and to follow while some of the more complicated paradigms such as object-oriented expert systems are not so easy to comprehend. Rule based systems are especially useful for encoding information about technical domains such as polymer selection, indeed Allen, Boarnet, Culbert and Savely [ABCS87] state that "IF-

THEN" statements that represent causal relations are especially applicable for encoding engineering knowledge". They go on to say that "For derivation type problems, rule based systems or hybrid systems with a rule based slant, are a natural selection (since cause and effect relationships are easily expressible as rules).", which also applies in this case as the proposed system will derive a technical specification from the supplied functional description.

Many people believe [MWL88,RW88,WS89] that more emphasis should be placed by expert systems on explaining their actions. A rule based paradigm provides a ready source of explanation. Explanations are used in different ways in expert systems. There are explanations in the form of answers to How? and Why? questions from the user of the expert system. The answer to How? is basically a trace of all the rules which have been used to come to the current conclusion. The answer to Why? is usually quoted as the rule which the system is attempting to solve at that time. In addition to this most expert system shells provide a means to customise explanations by providing some form of text which can be used to expand and clarify conclusions reached by the expert system. A customised explanation facility should be incorporated in the proposed system also.

Due to the flexible vocabulary and the rule based paradigm chosen to represent the expert's knowledge the resulting system is an open ended, rule based expert system shell, which allows the expert to build a system which guides an inexperienced polymer designer from a vague description of a design, towards the technical specification necessary to select a suitable polymer. The term open ended system is used to describe a system which is language independent in that the language to be used to express the rules and queries in the system is built up by the person programming the system.

3.3.2. The role of the polymer expert.

Being knowledgeable in both the areas of design and polymer engineering the expert knows both the design terms and the polymer property names to be used. Therefore the task of defining the general terms to be used to describe the functionality of the application to be designed and the technical terms of a polymer falls on the expert. These two sets of terms form the vocabulary for the quasi-English natural language interface. It will also be the task of the expert to construct the rule base which will determine which of the technical parameters of the polymers will be effected by certain design terms being assigned certain values.

The process of creating a rule based expert system is an iterative one. The builder of an expert system very seldom if ever gets it right first time. Therefore the system should allow for the iterative process of creating a working rule base and provide the means to test this rule base. In the proposed system this iterative process involves more than just making changes to the rule base. It involves the iterative process of producing the vocabulary to be used within these rules. An expert programming a rule base may not only recognise missing rules but may recognise missing terms which should be included in these rules and having recognised these missing terms may identify more rules. So the iterative process continues. This means that it must be easy for the expert who is constructing the expert system to create and amend both the rules and the vocabulary with which these rules are constructed. Also a rule tracing mechanism should be incorporated to allow the expert developing the rule base to view the rules as they are being activated by the system with a view to removing any possible mistakes in the logic of the rules.

The approach of allowing the expert to program the system reduces the problems encountered in the knowledge engineering process by allowing a closer interaction between the expert and system which eliminates the potential interference of the human knowledge

engineer. Because of this closer interaction between the expert and the system, the system must be expressive and adaptable enough to allow the expert to define both the terminology to be used within the system and the rules which assist the designer in reaching a technical specification for a polymer.

3.3.3. The role of the designer.

The role of the designer in the system is one of user of the system built using the expert's knowledge of how design concepts affect the properties of polymers. The designer builds up a functional description of an application using the terms defined by the expert. This forms a query which is to be solved using the rules present in the system. The solution which is presented to the designer is the technical specification for a suitable polymer, a list of all the properties that a polymer can possess with certain range values being set for certain parameters.

The designer is restricted to using the functional terms (as defined by the polymer design expert) to describe an application. This approach of having a flexible restricted vocabulary avoids the potential inadequacies of a system which restricts a designer to a predefined static vocabulary. At the same time it reduces the chances of ambiguity by restricting the number of terms which can be used to represent a particular design concept. Because these terms are entered into the system by the expert who is knowledgable in the field of design they should be familiar to the designer. If this is not the case the designer can access the meaning of the concepts represented by these terms by accessing information contained in certain rules present in the rules base.

3.3.4. The Language.

The terms defined by the expert form the basis for the quasi-English natural language interface used to interact between the developed expert system and the designer. These terms also form the vocabulary to be used by the expert in the construction of rules. The syntax of this quasi-English natural language is very simple and hence the interface is easy to use. The semantics of the terms involved, if they are not clear to the designer can be found within the structure of the rules within the rule base, as these rules are a function of the functional and technical terms defined by the expert.

The quasi-English language constructed within the proposed system is used for both rules and queries. As such it has to be easy to understand because it is being used by inexperienced computer users. "Every programming language has rules that prescribe the syntactic structure of well-formed programs" [ASU86]. These rules are known as the grammatical rules of the language. In the case of the quasi-English natural language proposed here a program can be considered as a set of rules entered by the expert or a description set up by the designer. Each sentence in the grammar is made up of clauses joined together by conjunctions. Each clause is made up of three "lexical items" [JA78] called the **descriptor (dsc)** the **conjugate/operator (cop)** and the **value (val)**. The general format for a clause is

dsc cop val [es][cop]

were es stands for an end of sentence (a full stop and a carriage return) and [cop] could be either an 'and' or an 'or' (see figure 4. for a diagrammatical structure of the grammar).

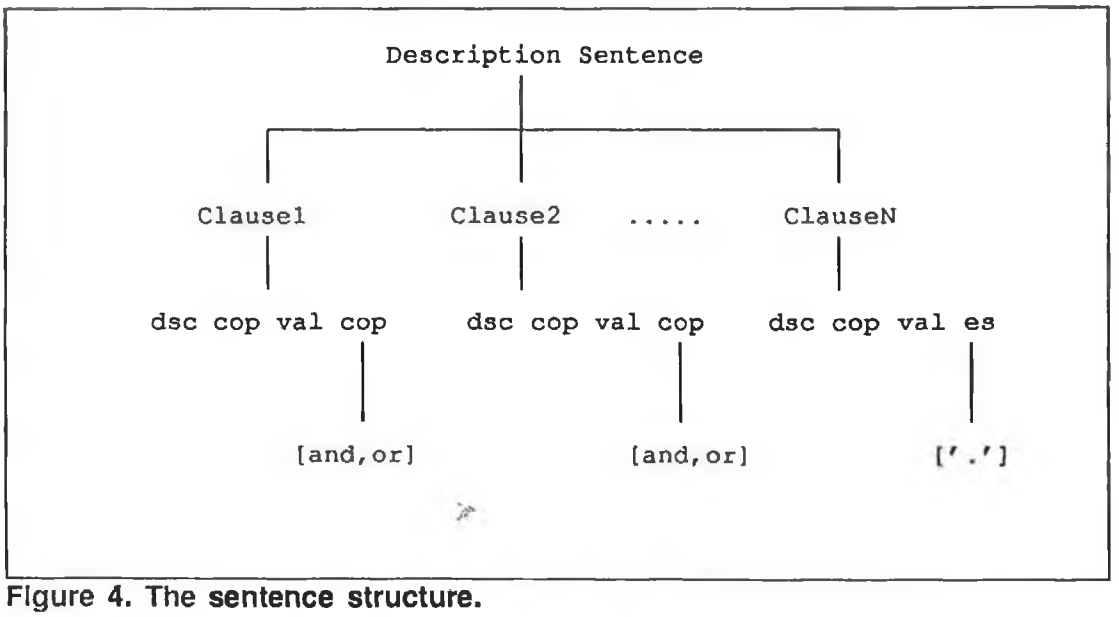

Figure 4. The sentence structure.

Many language constructs have "an inherently recursive structure that can be defined by context-free grammars" [ASU86]. The quasi-English natural language as can be seen from Figure 4 is recursive in it's definition and is defined by a context-free grammar. Once you have a simple grammatical structure you can automatically construct an efficient parser that determines if a statement is syntactically well formed. The parser which is constructed as a result of the above grammar is an ad-hoc top down recursive parser which produces a parse tree which is the same as that seen in Figure 4. This parser is discussed in section 5.3.1.1.

The grammar of the language used is simple and this fact makes it adaptable to the major European languages of English, French and German (see figure 5). This is partly due to the simplicity of the grammar of the language and partly due to the fact that these languages come from the same "parent language" (see [JA78] for details on language change).

descriptor	conjugate/operator	value
English		
temperature	is	high
French		
la_temperature	est	haute
German		
die_temperature	ist	hoch

Figure 5. general language structure.

3.4. An open ended system.

The proposed open ended system when developed should allow the expert to easily define and change both the functional terms and the polymer property terms to be used by the system as these terms make up the vocabulary to be used by the expert in building and testing the rules. A subset of this vocabulary is also used by the designer to build up a query to be fired on the rules. This vocabulary of terms to be used by the designer is defined iteratively by the expert as he/she is building the rule base.

This type of open ended expert system shell has a number of advantages.

(a). There is no need for the person programming the expert system to learn any special purpose programming language as the rules are constructed almost completely from terms which the person programming the system (i.e. the polymer design expert) has defined.

(b). The user of the developed expert system builds up a functional description of an application using design terms defined by the expert. These terms should be familiar to the designer using the system and thus there is no need for the designer to learn the syntax and semantics of a query language.

Chapter 4

Implementation Choices for an Open-Ended Expert System Shell

Chapter 4 Implementation Choices for an Open-Ended Expert System Shell.

Having established the qualities that an open ended expert system should possess it was necessary to construct a prototype system to test if these ideas could be implemented. Before constructing prototype system it is necessary to identify the functions that an open ended prototype expert system shell should possess. Having established these functions it is necessary to explore the possible paradigms with which to implement the prototype system.

4.1. Necessary system functions.

In looking at an open ended prototype expert system shell the following functions should be included in the system to allow both the polymer expert and the polymer designer to use the system effectively.

1. A function which will allow the expert to maintain the words which make up the vocabulary to be used in the system. This involves processing the three lists of words which as mentioned in Chapter 3 represent the valid words which can appear in a clause of the quasi-English natural language used in the system.

2. A full screen editor which will allow the expert to easily create and syntactically check a set of rules. This editor should allow such functions as string searching/replacing, quick cursor movement functions, fast key options for the more important functions, cut and paste facilities, etc, which are associated with the more sophisticated full screen editors

available today.

3. The system should allow the execution of the rules to be traced by the expert and the rules to be presented in a readable form to the expert with a view to allowing him/her to check the logical correctness of a particular rule set.

4. A descriptor tool must be provided which will allow the designer to readily view and use the vocabulary set up by the expert with a view to setting up a query to fired on a loaded rule base.

5. It should be easy for an inexperienced computer user (the expert or the designer) to compile, load and run a rule base with a view to initiating a query. Furthermore the designer should be given the choice of presenting information to the system using the descriptor tool mentioned above or allowing the system to initiate the process of constructing a polymer specification by questioning the user to establish the functional attributes of the design.

6. As mentioned earlier the need for questioning the designer about certain aspects of the design does arise and as such the questions should be presented in a clear and unambiguous manner so that the designer may understand them and give the appropriate reply. The designer should have the ability to question the decisions of the expert system when the system is asking questions of the designer. This quality should also be incorporated into the questioning facility which is presented to the designer on the screen.

7. During the course of reaching a specification the system should be able to explain it's actions in terms of the rules it has used to get to the current conclusion and in terms of the current rule which it is trying to solve. When the solution is finally reached the system should also present all the rules which have been used in order to reach the specified

solution as a further means of explaining it's actions. The system should also provide the expert with an easy to use tool which will enable the expert to set up a customised explanation which will enable the designer to further explore the significance of a certain value being set in the eventual technical specification for a polymer.

4.2. Implementation language choices.

In order to construct a prototype it is necessary to choose a language with which to implement the system which will facilitate the proposed main features of providing a flexible quasi-English natural language interface, a rule based paradigm and a suitable graphical user interface. This means that the language for implementation should possess database facilities to handle the potentially large vocabulary to be used in the quasi-English natural language interface; a data structure to represent a rule structure, together with a means to represent the designers query; a means of manipulating this data structure i.e. an inference mechanism; and some form of clear and concise graphics tool with which to represent the user interface. The table in Figure 6 outlines how effective the various paradigms which were researched were found to be with respect to the needs of the prototype system.

Type of Paradigm	Feature Supported			
	Rule based structure	Flexible vocabulary (database facilities)	Inference mechanism	graphics
C, Pascal etc.	2	3	2	10
Database Tech.	5	6	3	2
Lisp	6	6	6	8
Prolog	10	9	10	8

The numbers indicate on a scale of 1 to 10 the difficulty involved in supporting the property needed by an open ended system with the particular type of paradigm.

e.g. A low figure of 2 would indicate that a particular paradigm would have great difficult in representing that particular feature.

A high figure of 10 would mean that the particular paradigm would have little difficulty in supporting that particular feature.

Figure 6. Table of proposed paradigms and their relative effectiveness.

As can be seen from the above table the list of suitable paradigms shortens down to the two symbolic processing languages i.e. Lisp & Prolog.

4.2.1. Lisp.

Lisp is suitable for natural language processing as it is good at symbolic processing, which would allow it to support the proposed natural language. Also some versions of Lisp can incorporate graphics capabilities i.e. GClisp, but Lisp is discounted for a number of reasons.

Lisp is function evaluation driven (using the read-evaluate-print cycle) it has no built in inference mechanism with which to manipulate a rule structure. This is a major draw back with the language as it would be necessary to construct an inference mechanism.

Because Lisp is a functional language where programs are treated in a procedural light, there is no concept of a database therefore it does not readily support the database manipulation techniques which would be necessary to maintain the flexible vocabulary of the proposed system. It would be necessary to write special list processing functions to create the illusion of a database. It would be awkward to maintain the flexible vocabulary in such a way as to allow speedy access to particular terms contained in these lists as there are no database functions provided to deal with such necessities.

As of yet Lisp programs can only run in the Lisp environment and as such an application developed in Lisp needs a large amount of memory and disk space necessary to support this environment.

4.2.2. Prolog.

An alternative symbolic programming language Prolog was then evaluated as a language to implement the system.

The fact that Prolog is based on the well understood principles of predicate logic ensures that any expert system represented in Prolog will have the desirable properties of soundness, reliability, predictability. Alan Bundy in his paper [AB87] puts forward the theory that among a collection of " mostly unreliable, knowledge-engineering techniques, one family stand out as a model of respectability and reliability: the techniques of logic deduction used in automatic theorem proving and logic programming, e.g. resolution. " He believes that logic programming provides " a sound theoretical foundation " and that this will lead to more reliable expert systems by making them more robust, predictable and flexible. He argues that " An expert system rule or fact can be regarded as a formula of (predicate) logic ".

The Prolog language is made of facts and rules which are to be proved true and so provides a language in which the experts rules and the designers facts can be readily represented. The symbolic nature of Prolog together with the declarative reading of Prolog clauses ensures that the flexible natural language interface can be easily supported. A fact such as "application type is dishes" could be represented as "apptype(dishes)", where apptype is called the predicate or functor (see Clocksin & Melish) and the dishes is called the argument. The arguments can be atoms, integers, variables or indeed other facts. A rule consists of a head and a body. The body is made up of sub-goals which have to be proved true in order for the rule to be proved true. A Clause is the name given collectively to both facts and rules. There is an added complication that a predicate can also be defined by a group of clauses. For example the append predicate is defined by :

```
append([],L,L).
append([X|L1],L2,[X1|L3]) :- append(L1,L2,L3).
```

Prolog supports recursion as the above example shows and is very adept at handling lists. Lists are a very useful data type which are common to both LISP and Prolog. They can contain various different data types including other lists. You can treat lists in different ways :

1. You can consider lists as [a,b,c,[d,e],f] and manipulate it as a whole list [T, U, W, X, Y, Z] would return each element of the list in the associated variable.

2. You can consider lists as having a head and a tail [H|T] (much like the rules discussed earlier). This allows you to work with the top element of the list and allows you to process the list recursively with ease by using the same predicate to process the tail. The tail of a list is always a list itself.

3. The list as a whole can be unified with a variable. This allows you to use a list without having to break it up into it's head and tail, allowing it to be passed easily from predicate to predicate as in the case of the goal on the right hand side of the append rule above.

One problem with processing lists in Prolog is that the predicates used are highly recursive as in the case of the append predicate above. When processing very large lists such predicates can quickly use up the stack space and possibly cause the program to crash. A solution to this problem is to use difference lists which lend themselves to faster and more efficient implementations of the recursive predicates. Difference lists are yet another way of considering lists. The list is considered as being a list with a hole in it. This hole is to be filled in by some list operation. For example, a list [a,b] is considered the same

as a difference list [a,b|X] - X, were X is the hole in the list. As a result of this fact, the append predicate can be written in a non-recursive form by replacing recursion with matching. The append predicate could now be considered as

append(A - B, B - C, A - C).

Thus if A - B is [a,b|X] - X and B - C is [c,d|Y] - Y then the result using matching in the new form of the append predicate would be [a,b,c,d|Y] - Y, which is the desired result.

Prolog has an inference mechanism built into the language and this inference mechanism could easily be used on the rules constructed by the expert. The procedural meaning of Prolog is based on the resolution principle for mechanical theorem proving introduced by Robinson ([AJR65,AJR79]). Prolog uses a special strategy for resolution theorem proving called SLD, which incorporates matching (equivalent to unification), instantiation and backtracking. A variable is said to be instantatiated when the object for which that variable stands for is known. A variable is not instantiated when what the variable stands for is not yet known. The object in this case is usually an atom, string, integer or a structure. In practical programming terms instantiation means that the object is assigned perhaps temporarily to this variable. It is temporarily assigned because it can become uninstantiated during backtracking.

With matching the following rules apply :

1. An uninstantiated variable will match any object. As a result that object will be what the variable stands for.

2. Otherwise, an integer or atom or string will match only itself.

3. Otherwise, a structure will match another structure with the same functor and number of arguments, and all corresponding arguments must match.

The inference mechanism in Prolog is perhaps best explained using an example. Assume the following database where '%' represents comments.

```
% 1 john is a thief
thief(john).
% 2 mary likes food
likes(mary,food).
% 3 mary likes wine
likes(mary,wine).
% 4 john like X if X likes wine
likes(john,X) :- likes(X,wine).
% 5 X may steal Y if X is a thief and X likes Y and Y is valuable.
may_steal(X,Y) :- thief(X),likes(X,Y),valuable(Y).
```

In response to the question "what may john steal" i.e. may_steal(john,X) Prolog proceeds as follows:

1. First it searches through the database (top down) until it finds a fact or a rule to match the query. It finds it in the form of clause 5 which is a rule, marks this place in the database and X in the rule becomes instantiated to john. It then attempts to solve the sub-goals on the right hand side of the rule in order left to
right starting with thief(john) as X has been instantiated to john from the original query.

2. It initiates the search for the goal thief(john) from the top of the database and finds the fact thief(john). Prolog marks this place in the database also. It then attempts to satisfy the second goal in clause 5 which is effectively likes(john,Y).

3. The goal likes(john,Y) matches with the head of a rule (clause 4) The Y in the goal shares with the X in the head of clause 4, and both remain uninstantiated. To satisfy this rule, Prolog attempts to find a solution to the likes(X,wine) in clause 4.

4. The goal succeeds because it matches with likes(mary,wine) (clause 3) with X being instantiated to mary in clause 4 and Y being instantiated to mary in the second goal of clause 5 because X and Y share.

5. Having solved the first two goals in clause 5 it now attempts to solve the third and last goal which is effectively valuable(mary). But there is no fact to match this in the database and no rule to try and establish it so Prolog backtracks to try and find alternative solutions. During backtracking all variables which were previously instantiated become uninstantiated.

6. Prolog has kept track of all the places in the database were it has found solutions. It starts by trying to find an alternative solution to second goal likes(X,Y) in clause 5 which causes clause 4 to backtrack. But this too fails as likes(mary,wine) is the only fact that matches the right hand side of clause 4.

7. It then backtracks further to try and resatisfy thief(X) but this also fails causing the whole of clause 5 to fail. Since Prolog can find no other fact or rule to match the original query the query fails and Prolog returns with the answer "no".

Prolog has the added advantage of having ready access to a database in that a Prolog program consists of facts and rules contained in a program database. Facts can be added and retracted during the course of the execution of the program. This program database can be used to hold the rules as defined by the expert and the facts about the application to be designed, as supplied by the designer. It can also be used to hold the functional and polymer property terms defined by the expert and provides database manipulation predicates which can be used to maintain the flexibility of these terms and thus the quasi-English natural language interface.

Another useful feature of the Prolog language is it's ability to alter the structure of it's own programs during execution. This is done using the evaluable predicates (evaluable predicates are predicates which are built into the language) retract and assert (and variations on these predicates). The retract predicate allows you to remove a named predicate form the database, while assert allows you to add a new or changed predicate to a database. This ability to make changes to the program makes Prolog more flexible than any standard language such as C or Pascal which are static languages. The reason why this is so important in the context of the proposed prototype system, is due to the fact that a loaded knowledge base[2] can be treated as a Prolog program. Hence as new facts are established through questioning the user or through proof of a rule they are added to the existing known facts in the knowledge base. Thus altering the structure of the program. These facts are available to other rules which may be activated during Prologs search through the knowledge base. Such actions could not be efficiently supported using a standard programming language.

As the rules and the application descriptions are represented to the users in a quasi-English form but must be processed by the system in some internal form a need arises to translate from one form into the other. English-like rules translate quite easily into Prolog rules. The application description is easily translated into a set of Prolog facts.

Once a solution has been reached in the form of a technical parameter list a need might arise for the designer to query how a certain range value was reached. Prolog allows the system to easily group together and display all the rules which have lead to this range value being set.

[2] The knowledge base in the context of an expert system developed using Advisor consists of both facts about the current application under consideration and the rules which map a description onto a technical specification for a suitable polymer.

An added bonus obtained if the system is constructed in Prolog is that when the rules created by the expert and the facts provided by the designer are translated into Prolog code, they can be added to the code for the shell to form one complete Prolog Program. This facet of the language allows the rules to access parts of the system and use some of the functions incorporated in the system such as the graphical capabilities provided by the system.

4.2.3. Some Problems with Prolog.

However Prolog also has a number of disadvantages which have to be overcome in constructing the system. But these are problems which affect the person constructing the shell rather than the expert and the designer who will be using the shell.

As can be seen from the above thief example Prolog searches through the program database sequentially each time it attempts to solve a goal or sub-goal. If the program database was to be represented as an inverted tree structure with every possible solution shown as a path in this tree, then the search through the database would represent a depth first search of this tree structure. In a depth first search each particular branch in a tree is followed downward from left to right until the original goal is proved to be true or all the possible solutions are investigated. This method has the advantage that a solution will be found if it exists (provided no circular reasoning occurs) but has the disadvantage that for a large program database the search can be very time consuming if the solution lies to the right hand side of the tree and is several levels down.

Because the Prolog inference mechanism incorporates the facility to backtrack on failure to solve a goal the process of debugging a program can be very difficult as it is necessary to keep track of all the places in the program database where the previous solutions have been found because consequent searches start from this point in the program

database. It also necessary to take into account the fact that variables become uninstantiated during backtracking which further complicates the debugging process.

As Prolog is a highly recursive programming language it uses a large amount of stack space to remember the previous calls to the predicates and the values established. But the amount of stack space is limited and is only released during back tracking. This has implications on the structure of predicates and the amount of recursion which can be allowed within a Prolog program. As a rule a Prolog program should be made up of many concise predicates rather than a few large ones which can detract from the readability of a Prolog program. However some implementations of the Prolog language allow repeat and fail loops within a predicate which use the back tracking mechanism to reclaim the stack space used by the predicate. In addition to providing repeat and fail loops some versions of Prolog also support tail recursion. Tail recursion occurs when the last call in a recursive predicate is a call to the predicate itself. Any such recursive function can be replaced by a loop like structure and thus avoid using up the program stack. A good compiler will detect tail recursion and replace such predicates with a loop structure.

But overall Prolog was seen to be the most suitable language with which to implement the prototype system as it matched most closely the main features of the proposed open ended system.

4.2.4. Choosing a suitable version of Prolog.

The problem still remained of choosing the best form of Prolog for the task. A number of PC based versions of Prolog were researched. Products were considered in the light of graphical capabilities to be incorporated in the user interface and the quasi-English natural language interface, database capabilities both from a programming point of view and for manipulating the maintaining the flexible vocabulary, possession of a good development

environment and from the point of view of providing standard Prolog as defined by Clocksin and Mellish.

Turbo Prolog has good database capabilities both for manipulating the program database from a programming point of view and the capabilities to use the program database as a database in it's own right. It has good graphics capabilities through the use of external graphics packages. It has a good development environment. It also possesses a compiler for creating stand alone applications. However this compiler does not detect tail recursion. But the main problem with Turbo Prolog is that it is not standard Prolog in that you have to declare domains, predicates, databases, clauses, and goal types prior to executing a program which makes it awkward to use and makes the programs non-portable.

Prolog86 has standard Prolog predicates but was not considered as it only has a very primitive development environment and programs can only be used from within the interpreter. Also it provides no graphics capabilities. It too does not detect tail recursion.

Smalltalk V also has a version of Prolog incorporated into the Smalltalk Environment. As it is part of the Smalltalk environment it possessed good developmental tools and could access the graphics capabilities built into Smalltalk. It also detects tail recursion. But it suffered from the fact that it slanted towards an object oriented paradigm which is difficult for the polymer expert to understand, it only possessed a subset of standard Prolog and you need the Smalltalk environment to use it.

The Arity Prolog product proved to be the most impressive of the Prolog packages encountered. It provides a standard Prolog language base together with some useful enhancement features. It provides excellent database features both at the programming level and at a low level. The low level features allow the flexible vocabulary to be manipulated

quickly using specialised database manipulation and search facilities. It provides a good developmental environment in the form of an interpreter which incorporates a sophisticated debugger and good editing facilities. It also provides a good compiler to produce an executable version of a program developed in the interpreter. The compiler also detects tail recursion. Finally, incorporated within the language are the programmable features of dialog boxes, windows and pop-down menus which can be used to form a powerful graphical user interface. A full evaluation of the Arity Product can be found in Appendix C.

Chapter 5

Advisor

The Prototype System

Chapter 5 Advisor the prototype system.

Having established the functions that the proposed expert system shell should possess, explored all the available shell technology and found it wanting, decided to build a shell which would better provide the desired qualities and selected a suitable language with which to implement the prototype system it was then necessary to construct the prototype system. The resulting Prototype system is called Advisor.

5.1. Prototyping.

"The actual development of expert system begins in earnest when the knowledge engineer and the human expert work together to create the prototype system, a small working version of an expert system designed to test the assumptions about how to encode the facts, relationships, and inference strategies of the expert." [HMM87]. This section outlines the value of prototyping by outlining the first approach adopted in the construction of the first Advisor prototype which subsequently proved to be inept in certain areas and outlines the alternative approach which was adopted as a result.

5.1.1. The shell intentions.

The first prototype shell constructed included a module called the Application Descriptor which enables a user to build up a description of an application in an English like format using lists of words which were previously set up by the expert (In this case myself and the expert took the part of the user). The prototype shell also includes a module called the Rule Builder which was used to parse and compile the English like rules as seen in Figure 3 into a form shown in Figure 7.

```
r1([scratch_res(yes),chemical_res(detergents),food_approved(yes),
chemical_res(boiling_water),chemical_res(alcohol),antistatic(yes),
flame_retardant(yes)]) :- apptype(dishes) .

r2([material(ps)]) :- price(low),transparancy(yes).

r3([heat_distortion(>,60),price(low)]) :-
apptype(dishes),instance(glass),usage(disposable).

r4([hardness(>,26),hardness(<,27)]) :- apptype(dishes),hardness(high).

r5([hardness(>=,20),hardness(<=,30)]) :- hardness(high).

r6([microwave_res(yes)]) :- apptype(dishes),used_in(microwave).

r7([viscosity(low)]) :- apptype(dishes),thickness(thin).

r8([thickness(thin)]) :- apptype(dishes),usage(disposable).

r9([chemical_res(oil)]) :- apptype(dishes),environment(petrol_station).

r10([density(<, 1)]) :- apptype(dishes),environment(ship).

r11([impact_strength(high)]) :- apptype(dishes),temperature(low).

r12([fda_test(yes)]) :- apptype(dishes),country(america).

r13([transparancy(yes)]) :- apptype(dishes),instance(glass).

r14([heat_distortion(>=,100)]) :- apptype(dishes),specific_properties(dishes).

r15([chemical_res(alcohol)]) :- apptype(dishes),instance(cup).

r16([chemical_res(alcohol)]) :- apptype(dishes),instance(glass).

r17([hardness(high)]) :- instance(plate).

r18([hardness(high)]) :- material(ps).

r19([high_gloss(yes)]) :- instance(glass).

r20([unfilled(yes)]) :- apptype(dishes),instance(glass).
```

Figure 7. The compiled prototype rule set.

It was attempted to use a production rule inference mechanism in order to allow the rules to be more comprehendible to the person entering them. This involved controlling and monitoring a rule firing cycle by keeping track of rules fired and which rules

established new facts, together with all the facts established. This is discussed further below.

Uncertainty in systems is usually represented in the form of some numeric confidence factor. The prototype system did not provide a mechanism for supporting confidence factors for two reasons. " Many if not most applications do not require uncertainty" [RW88] and this is the case in the area of polymer design where the process of either assigning a value or not, is exact. Also it is widely accepted that confidence factors are usually erroneous and meaningless when used in systems unless they are used properly and are based on a sound theoretical basis [GL89]. Most systems allow the designer of a rule base to specify confidence factors or probability values, but these are assigned subjectively by the person constructing the rules and hence they are unreliable.

It was also necessary to devise a method to trace the activation of the rules as this is necessary to support an explanation facility and is necessary for checking if the logic of a rule base is correct.

5.1.2. Advisor the first approach.

Having carried out the knowledge engineering process and built the first prototype shell I was ready to test the Advisor prototype shell by inputting some rules which I obtained during my interviews with the experts concerned. Prototyping is very much an iterative process but necessary to test the principles on which the system is based to see if the current approach is a valid one. As with any computer system, errors should be found in the early stages of development as the cost of fixing errors increases over time [RD78],[BB81 p38-p43]. Prototyping in the instance of the Advisor system had to be considered in two contexts. Firstly the shell itself was a prototype and secondly the system developed based on the rules obtained was to be a prototype of the eventual expert system.

The way the inference mechanism in the first prototype system worked was as follows....

1. The application description was translated into facts and asserted in the rule base (program database). The rules in the loaded rule base were organised into groups. The groups being set up as the rules were being consulted by analyzing the contents of the left and right hand sides of the rules.

There were three groups of rules

(i) The general to general group. These were rules which mapped general properties onto general properties. Examples of these kind of rules can be seen in the likes of r1 and r2 in Figure 7.

(ii) The general to specific group. These were rules which mapped general terms onto specific terms. Specific terms were terms which had specific values assigned to them using arithmetic operators. Examples of these rules can be seen in r3 and r4 in Figure 7.

(iii) The third group of rules mapped specific terms onto specific terms. However in the experts' opinion this situation should never arise as technical parameters in CAPS are independent properties.

2. The rule base was activated by calling each compiled rule in turn starting with r1 or calling them by groups in the order (i,ii,iii) above. Using the groups one could firstly activate the general to general rules and gradually progress to the technical specification through each of the groups examining the results at each stage.

3. If the goals succeeded i.e. if all the facts on the right hand side of the rule were present in the program database, the facts in the list of the head of the rule would be added to the current contents of the program database and the next rule would be called.

4. If any goal on the right hand side of the rule failed the next rule would be tried.

5. When the system had fired on all the rules it would begin again from the first rule. Using both the original facts and the newly established facts. The firing cycle stopped when no newly established facts were found.

6. The system kept track of :

(A) The rules which failed.

(B) The rules which succeeded.

(B) The original facts as presented in the functional description.

(C) The facts established to date as well as those established in that particular firing cycle.

All these were to be used to establish which facts would be fired in the next pass on the rules and provided a means of auditing the rules as they were activated.

This method had the advantage of being easy to understand for the person constructing the rules. The reading of the quasi-English version of the rules was similar to the way in which most people would think of rules as there was no need for meta rules which tend to complicate the issue. But the method suffered from a number of draw backs.

1. The system was not making the maximum use of Prologs built in inference mechanism. It was based more in the style of production systems such as XCON or R1.

2. It suffered from the fact that it was hard to interpret all the facts which were reported by the system and would have been difficult to construct why and how mechanisms.

3. It would also have been difficult to establish if the firing cycle would complete in a

reasonable period of time, especially for a larger rule base. This is due to the iterative nature of controlling rule activation during the firing cycle. For example during a firing cycle a single new fact may be established but the firing cycle would still continue. The worst case scenario being if one new fact is established during each firing cycle.

Taking all these disadvantages into consideration it was decided to make some changes in the approach to constructing the rules and to structure the inference mechanism so that it took full advantage of Prologs powerful inference mechanism. This new method proved to be most satisfactory and led to the system as it exists to date.

5.2. Advisor a prototype shell.

The construction of the open ended prototype system called Advisor gave rise to a number of problems which needed to be solved. The Advisor system is a large piece of computer software and as such it needs to constructed in a modular fashion. It is also necessary for the Advisor system to support the functions of an open ended system as stated in Chapter 4 and representing these in a manner which would allow them to be used easily by the expert and the designer. As it has been decided that the best model to represent the expert's knowledge, in the context of the polymer design domain, is a rule based model, the problem of controlling the search through a large rule set must be addressed. A method of allowing the system to elicit vital design information from the designer must be implemented. Also providing a designer with the means to question the actions of the system and a way of allowing the system to explain it's actions should be realised.

As the system was to be a large piece of software it was necessary to construct it in a modular fashion in order to allow ease of maintenance in the iterative process of prototype development.

5.2.1. The Modularity of Advisor.

Modularity and structured design are modern concepts of computer systems design which considerable attention has been drawn to over the last decade or so (notably by Jackson and Davis [MJ83],[WD83]). A good programming language should support these concepts. In this section the proposal is put forward that Prolog is a suitable programming language in that it has good modular programming qualities and the implications of this fact with respect to the development and implementation of the Advisor system are considered.

One of the advantages of using Arity Prolog to develop the Advisor shell was that the system could be constructed and tested in modules. The main modules which make up the Advisor Program can be seen in Figure 8. Modularity in program development is important as it reduces the problems of maintainability and debugging. Modularity also increases the readability of programs and leads to more structured programs. Prolog is a highly modularised programming language in that predicates are completely independent pieces of code. As there are no global variables in Prolog it also possesses the added advantage of having good data hiding features [MJ83] as predicates normally communicate through calls to each other and parameter passing.

The first modules to be created, tested and debugged in Advisor were the Application Descriptor and the Rule Builder. These were then presented to the Polymer expert as a prototype system and were subsequently amended to reflect new ideas which arose from Expert's views on what the system should do. The changes to the system were carried out with ease because of the modularity of the system. These two modules incorporated the natural language interface and the adaptable language construct which will be described in section 5.3. The other modules which represent additional tools for expert system development were added to the system at a later stage. These modules could use

predicates developed in the construction of the earlier two modules while remaining independent of them.

Prolog programs are very difficult to debug because of the backtracking mechanism which is incorporated (even with a sophisticated debugging tool). From a debugging point of view, modularity is especially important, as it allows one to localise the errors which occur and set the debugger to be activated in that module as described in the last section (using an inline call to trace). This is especially important when programs are large as was the case with the Advisor system.

The idea of modularity also has repercussions at the compilation stage of the development of Prolog applications (see Appendix C for a description of the compilation process of Arity applications). Only those modules which have been changed need to be recompiled and linked to the existing object modules.

It is the purpose of the Application descriptor module to allow the designer to build up a query or application description, translate this description into a set of Prolog facts and invoke the rules in the rule base to use this functional description to establish a technical parameter list.

The purpose of the rule builder is to allow the expert to iteratively define both the terms to be used in the system and to iteratively construct the rule base which maps the functional specification of an application onto the technical specification for a suitable polymer. It is also the function of the rule builder module to allow the expert to set up a customised explanation in the form of canned text to explain to the designer using the system the significance of a range value being assigned to a technical parameter in the solution to the functional description.

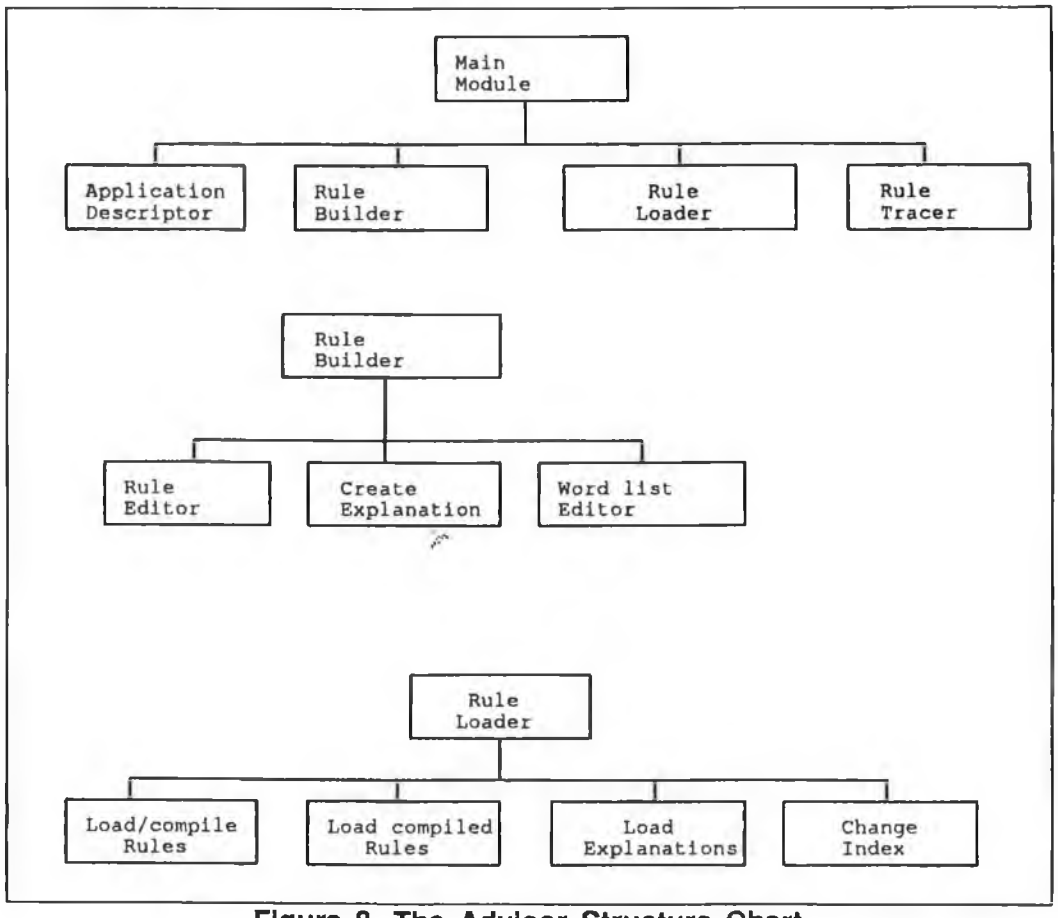

Figure 8. The Advisor Structure Chart

The rule loader module is used to handle loading and compiling of a rule base into the system. It is involved in loading the customised explanations set up by the expert in the rule builder. It is also responsible for changing the data structure which is used to represent the technical specification list.

5.3. The user interface and functions.

The user interface of any computer system is used to present information to the user of the system and to gather information from the user. " It has been estimated that half of any decent expert system should be devoted to communicating with the user,..." [RW88]. The problem which exists is that the users of the Advisor system have limited experience in the use of computers. So the functions of the tools provided by the shell must be inherently obvious. David Tong says [DT87] "the success of an expert system often depends on the acceptance of the end users..."and that "too often the end user interface is neglected at the prototype stage. While end user details may not be of paramount importance at that time, establishing the basic end user requirements will help avoid a later switch in shells". Arity Prolog proved to be an excellent choice for developing this interface as it provides facilities to construct customised dialog boxes which present information to the user of the system in a clear and easily assimilated form. Thus allowing the shell to be programmed and used by inexperienced computer users. David Tong also has views on this last point, he maintains that "... knowledge base maintenance is best conducted by the expert himself who is likely to be inexperienced in knowledge engineering. The ease of use of the shell goes far in making this possible and without extensive training of the expert" [DT87].

The user interface in the case of the Advisor prototype shell consists of the flexible quasi-English natural language interface together with a graphical representation of the quasi-English natural language interface and the system functions in the form of dialog boxes.

5.3.1. Representing the quasi-English natural language.

The quasi-English natural language as seen in Chapter 3 consists of sentences made up of clauses, where each clause consists of a descriptor followed by a an operator which is used to assign the third "lexical item" of the clause, the value, to the descriptor. These clauses can be joined together by a conjugate which is either an 'or' or an 'and' to form sentences. The end of a sentence is recognised by a full stop and a carriage return. These sentences are used by the designer in building up a functional description of an application. They are also used by the expert in building up the rules in the system.

The three elements of a clause have word lists associated with them. These word lists are set up iteratively by the expert and define the valid words that can appear in each slot in the clause. The function to set up and manipulate these lists is found within the Rule Builder. The word lists represent a vocabulary which is readily available to both the expert developing the expert system and the user of the designed expert system. The word lists allow the system to be as flexible as is possible as they can be edited to reflect the terminology to be used in a particular domain. The contents of these word lists are displayed through the use of list box controls described below. The three word lists can be seen displayed in the application descriptor dialog box in Figure 9 below. The designer building up a description of an application can choose words from these lists using predefined function keys and place these on the command line in order to construct the description. Thus ensuring that the vocabulary used to build up the query is correct.

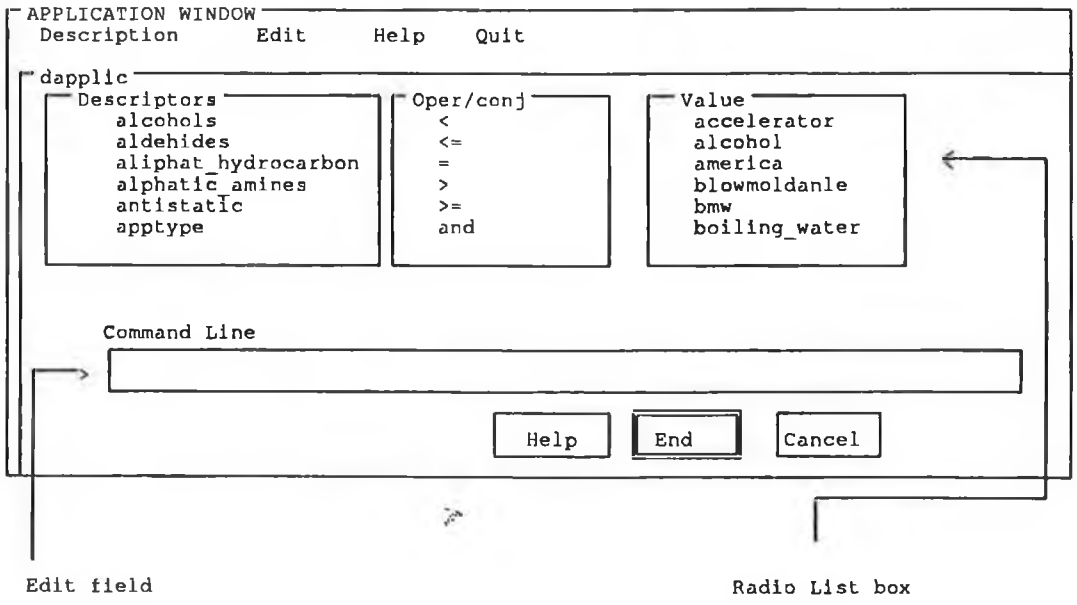

Figure 9. The Application descriptor dialog box.

Setting up these word lists is an elementary process for the expert with the aid of the add dialog box which allows the expert to add words to the three separate word lists and display the current list of words for the particular list with which he/she is working. The add dialog box can be seen in Figure 10. Words can also be deleted quite easily from these lists using the delete dialog box as seen in Figure 11.

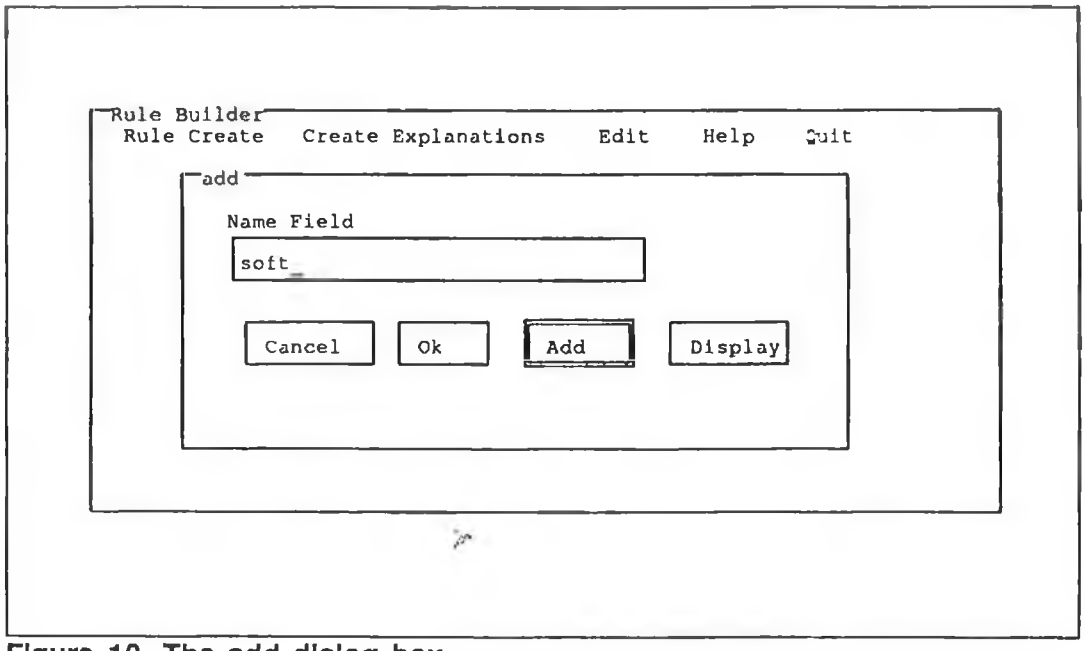

Figure 10. The add dialog box.

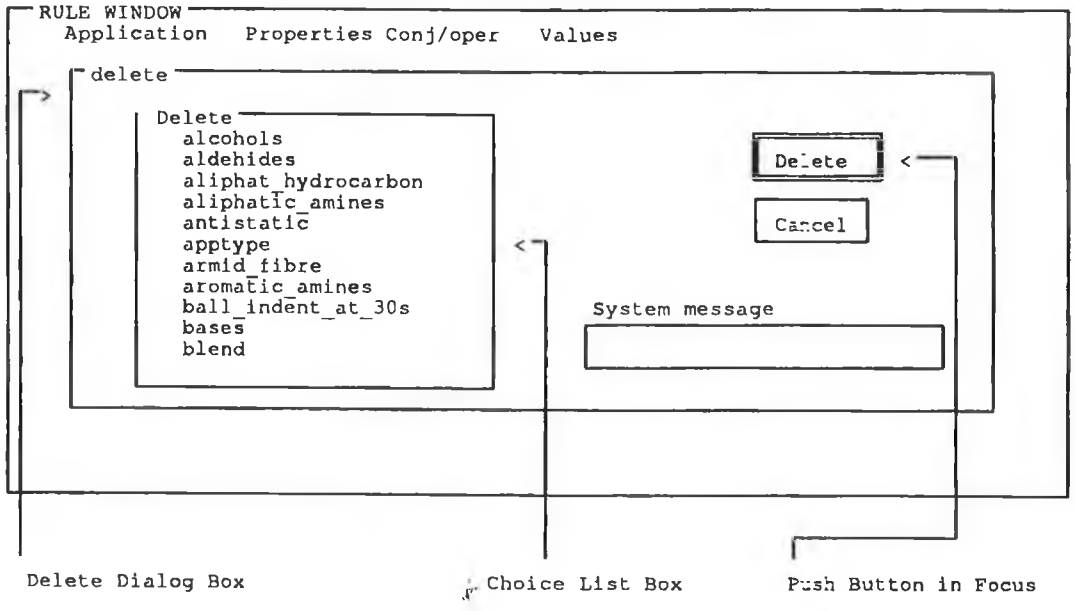

Figure 11. The Delete dialog box.

The reasons for making the expert set up the list of valid words prior to constructing rules are as follows....

1. A dictionary of terms used in the application domain is built up.

2. Once the word lists are set up they can be used with the special functions built into Advisor to choose words from default lists while constructing rules and queries.

3. The value word list provide the set of valid answers to questions asked by the system from which the user can choose an answer.

4. It encourages the expert to think about the domain before starting to code the rules. He identifies all the technical terms as well as the design terms which will be used in building rules and queries.

Within the descriptor word list there are two classes of words called general and specific. The **general class of words** are typically used by the user of the developed expert system to build a query and would represent his/her world in familiar terms. These terms would be used to describe the functionality and operating environment of the application. The **specific class of words** are the technical words and terms as used by the plastics engineers[3]. These terms are used to describe the polymer properties which the chosen plastic should possess.

The operators are also classed into to general and specific types. **General operators** (such as of, to, is, are, in, etc.) are used to assign values to general descriptors. Because

[3] These words would typically appear in the Index structure to be discussed in the next section.

of the technical nature of the specification which is passed on to the CAPS database system arithmetic symbols have been chosen to assign values to specific descriptors. These arithmetic operators are termed **specific operators**. Both numeric and string values can be assigned to specific descriptors. This would be awkward in a standard programming language but in Prolog it is quite acceptable to assign an atom, string or integer to the same variable as variables are not typed.

The valid **values** allowed in the system consist of all integers and symbols which are defined in the value class list of words.

Some of the descriptors and the operators have special semantics within the system. The specific class of descriptor words as well as being included in the word list are also included in the **Index** structure (Advisor's representation of the technical parameter list) which is Advisor's representation of the polymer technical parameter list and is the link between the expert system and the CAPS database. Values are assigned to the descriptors' slots during the consultation of the rule base. The Index structure is discussed in the next section. The specific operators i.e. the arithmetic operators also have a special semantic meaning within the system as they are used to assign values to the elements of the technical parameter list.

5.3.1.1. The Parsers.

As mentioned earlier (5.3.1. and 3.3.4.) the words present in the lists representing the three "lexical items" of a clause are used to build queries and rules. The fact that these queries and rules are to be translated into Prolog implies the need to exercise some form of parsing of the rules and queries to ensure the correctness and consistency of the translated form. "Parsing is the process of determining if a string of tokens can be generated by a grammar" [ASU86]. Such a parser should according to Aho, Seti and

Ullman [ASU86]

1. "... report any syntax errors in an intelligible fashion " and
2. "... recover from commonly occurring errors so that it can continue processing the remainder of it's input."

Some form of lexical analysis is also required prior to invoking the parser to split the sentence input into tokens to be processed by the parser.

There are in fact two parsers within the Advisor system, one for parsing the queries and one for parsing the rules. Both are similar using the clause structure as their "syntactic sugar" to produce the parse tree as seen in Figure 4 (see section 3.3.4). The difference between the two parsers stems from the rule parser's need to process the key words "if" and "then". The lexical analyzer for the rule parser consists of the split/1 predicate (see advsplit.ari in appendix A) together with the pass/1 predicate and the pass2/2 predicate (both present in advdsprl.ari in Appendix A). The lexical analyzer for the query parser only needs to use the split/1 predicate.

The resulting "intermediate code" [ASU86] produced by the two parsers is processed by the translator (the build/0 predicate in advappmu.ari for the query parser and the commit/2 predicate in advcomit.ari for the rule parser). The whole process of lexical analysis, parsing and translation can be seen in Figure 11a.

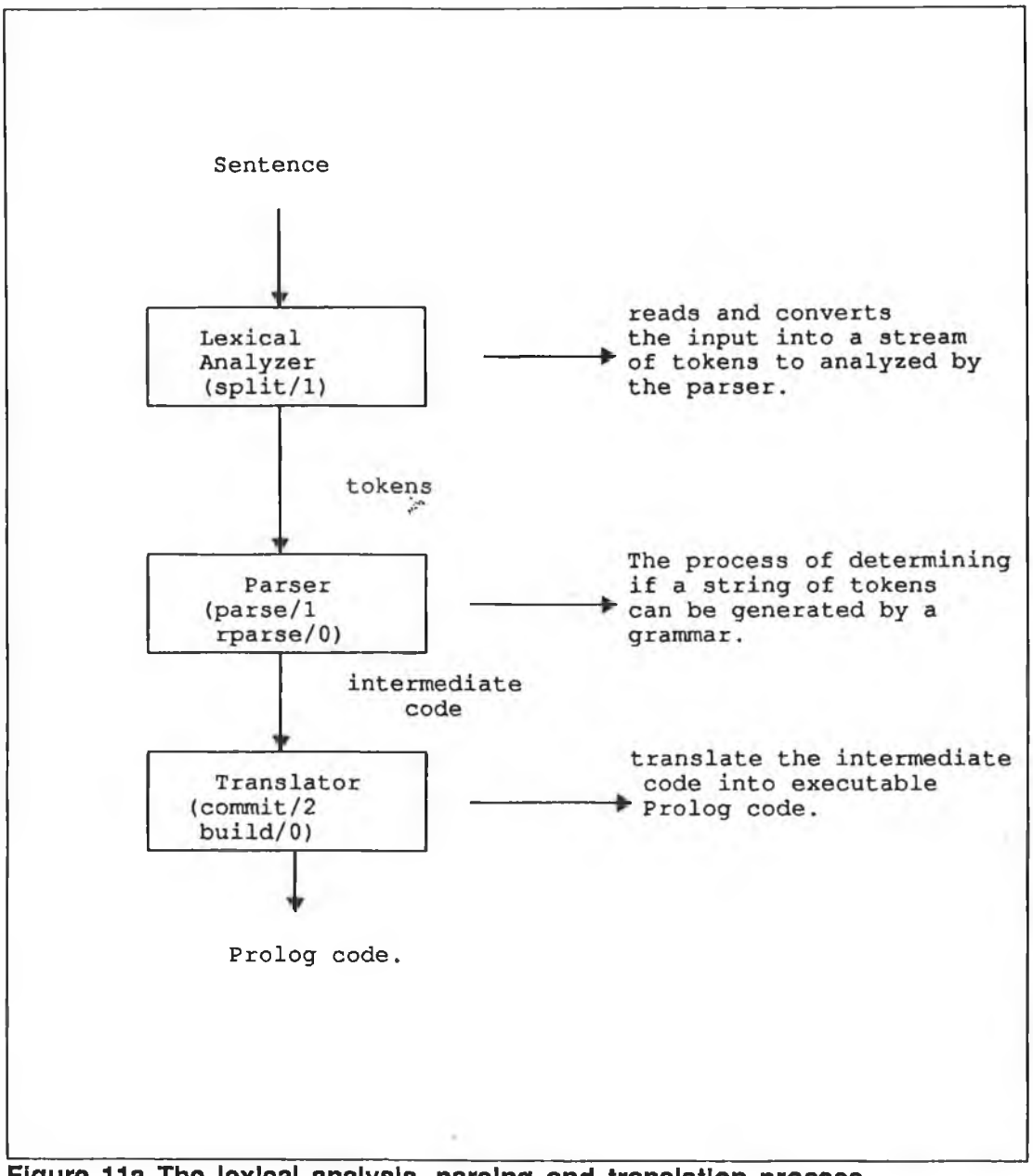

Figure 11a The lexical analysis, parsing and translation process.

5.3.2. The graphical interface.

The second part of the user interface determines the way in which the quasi-English natural interface and the system functions are presented to the users of the shell (both designers and experts alike). The system functions and the quasi-English natural language interface are presented to the user using a combination of pop-up windows, pop-down menus and dialog boxes.

5.3.2.1. Windows and Menus.

There are three important windows in the Advisor system each representing a suite of functions for a particular task. The three windows being the Main Window, the Application Window, and the Rule Window. The idea of associating windows with specific sets of tasks is done with a view to not cluttering up the Main Menu bar and visually separating the tasks. The idea of moving into a different part of the system for a different set of tasks reflects the underlying modularity of the system. This structuring of the system in this way helps the users to become familiar with the system as they are only forced to think about one part at a time.

The functions associated with these windows are invoked using the popdown menu system (see Figure 12) which is easy to operate using combinations of arrow keys and the return key. Most of the successful PC expert system shells are menu driven (e.g. Crystal, Leonardo, Experience).

As can be seen from Figure 12 the menus are organised in a hierarchial fashion which is easy for the user to assimilate. Sub functions of the main functions are represented in popdown mode. These are used when the function name outlines a broad concept which can be expanded upon e.g. an edit could be a addition or deletion or a change, or similarly the Load function could have sub-functions (see Figure 13, Figure 14).

At the top level there is the main menu on which the first two functions (from left to right) call the two remaining windows. The rest of the functions are to do with manipulating a selected knowledge base and obtaining system help. The advantage of menu driven systems is that they are organised and clearly show the functionality behind each choice.

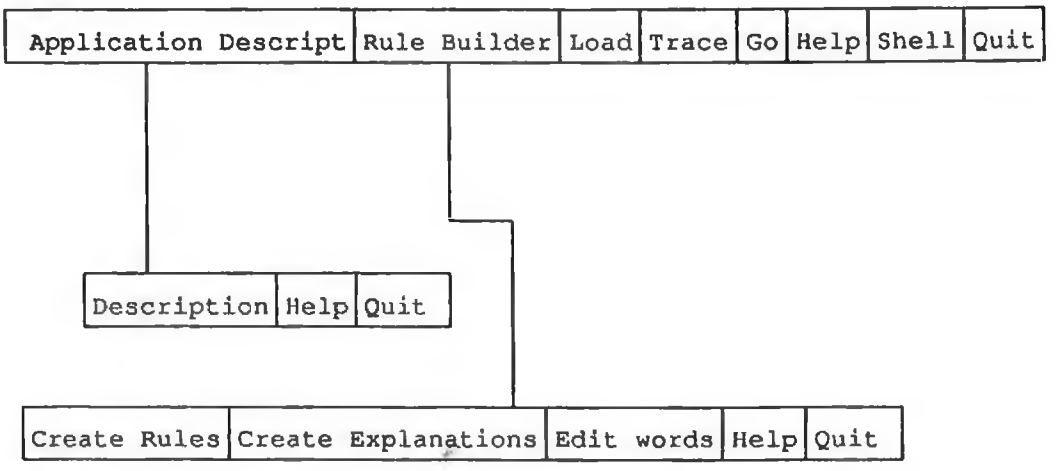

Figure 12. The menu system chart for Advisor.

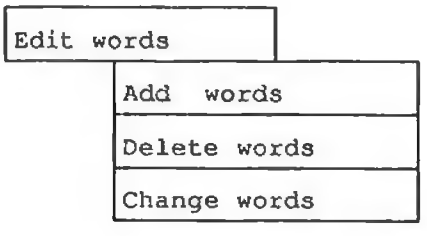

Figure 13. A popdown menu .

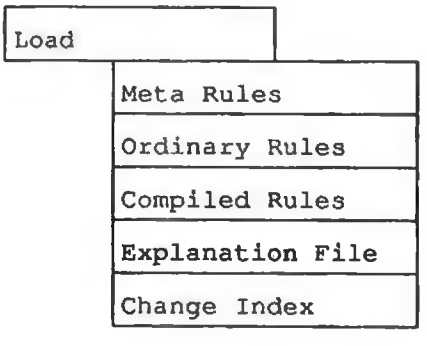

Figure 14. The Load Popdown Menu

5.3.2.2. Dialog Boxes.

All the important functions on the menu bars give rise to what are termed dialog boxes. Dialog boxes are a special type of pop-up window which have various different types of graphical controls present in them which are used to present and gather different types of information.

Dialog boxes are used to control dialog between the user and the Advisor Shell in keeping with ideas laid down by Alistair Sutcliffe on Human-computer interface design [AS88]. Dialog boxes such as the Application Descriptor dialog box (see Figure 9) which is used by the designer to build up a query in the quasi-English natural language, have several controls which act as aids to the designer using the tool. Different controls are used to present and gather different types of information. All dialog boxes in Advisor have standard actions both for moving from control to control within a dialog box and for carrying out certain actions within controls (which are predefined as per the Arity/Prolog Language Reference Manual [AP1]).

In all dialog boxes the TAB key moves forward from one control to another and Pressing Shift-TAB moves back from one control to the previous one. If a label on a control has a character outlined in a different colour then pressing ALT and that character will bring that control into focus. The various different types of controls used in Advisor and the standard actions defined for each control are outlined below.

Choice button :

Choice buttons are used for choosing one or more items within a dialog box, the choice of which can be programmed to affect some other control in the dialog box. For example in the display dialog box (see Figure 15a), choosing one of the list categories, displays the currently defined word list for this category in the list box control displayed.

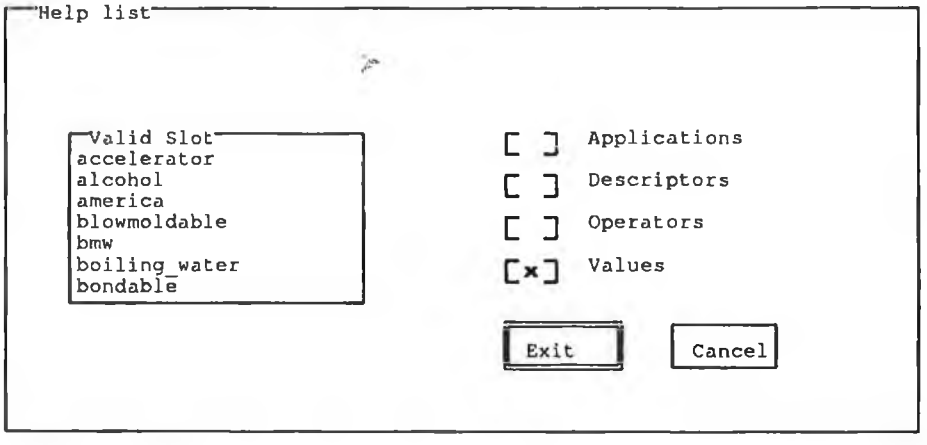

Figure 15a The Help list dialog box.

Edit field :

This control proves to be most useful for processing lines of the natural language text were the user can make mistakes. The command line in Figure 9 is an example of an edit field. Edit fields can also be used to display messages issued by the system as is the case when errors occur in the Rule Parser. An edit field is used to show or get a single line of text. A full range of edit facilities are available within an edit field including string searching, search and replace, cut and paste facilities. Thus it is ideal for processing application description which if found erroneous can be changed quickly and easily.

Push Button :

Push buttons are used to represent commands or functions within dialog boxes. While in focus pressing the Enter Key or space bar will invoke the labelled command. A push button is in focus if it has a doubly lined boarder. Push Buttons are in evidence in most of the dialog boxes in the Advisor system and can be used to call other dialog boxes as is the case with the Help Push Button in Figure 9.

List box :

List boxes are used to display lists of similar items. In the Advisor system they are used extensively to display the word categories which are used to build up queries and rules. The actions which are defined within a list box are

There are two types of list boxes :

1. When in a **Choice list** box (e.g. in the delete dialog box Figure 11) pressing the spacebar toggles the choice marker. One can have multiple choices in the choice list box. Choice list boxes are useful in cases were one wants to choose and process several items in a list.

2. When in a **Radio list** box a selected item is indicated by a single marker which moves up and down with the up and down arrow keys. Examples of Radio list boxes can be seen in Figure 9.

Edit box :

Edit box controls are used for processing large amounts of text (larger than can be handled by an edit field). Edit boxes in compiled applications are treated the same as the editor in the Arity Interpreter Environment and as they such possess all the same functions as the editor. An example of an edit box can be seen in the Rule editor as seen in Figure 15, which is used to view, edit and check the syntax of rules in the Advisor system. Edit boxes provide all the facilities provided by modern full screen editors including a search and replace function and cut and paste facilities.

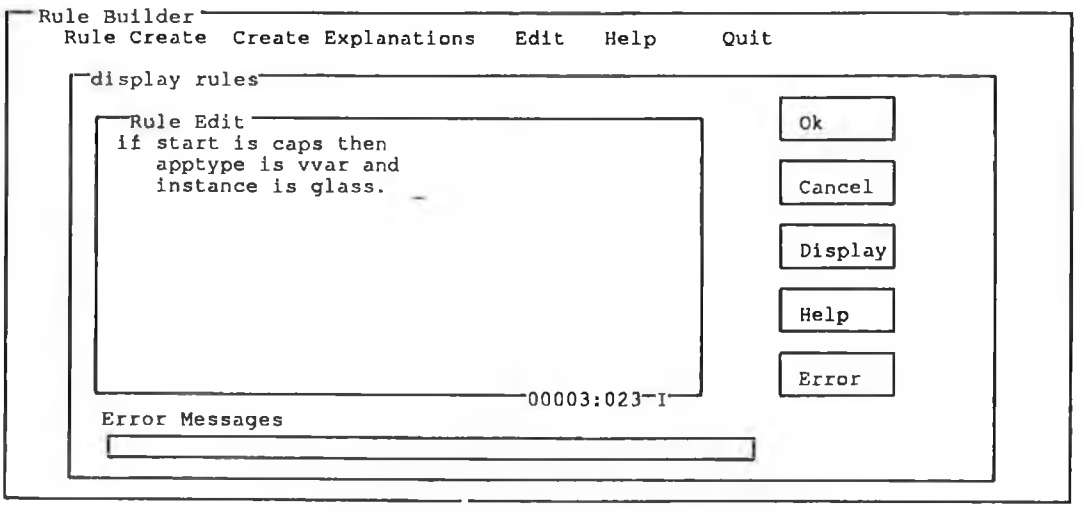

Figure 15b. The Rule Editor dialog box.

One of the main advantages to dialog boxes is that by combining the actions of the controls mentioned above together with special purpose actions programmed into the compiled application, one can have a very powerful and adaptable graphical interface which is easy to use.

Function keys can be redefined to have special actions within dialog boxes while certain types of controls are in focus. As an example of how beneficial this redefinition process can be, the Application descriptor dialog box (Figure 9) is considered. In the Application descriptor dialog box the function key F2 is used to select a highlighted item from a list box and place it on the command line. This saves an inexperienced typist the trouble of typing the command line while insuring that the spelling of the words is correct. This in turn leads to a more efficient method of building up queries to be processed by decreasing the number of typing errors.

5.4. The technical specification (index).

Before the problem of implementing the knowledge base could be addressed it was necessary to devise an appropriate data structure to represent the technical specification. One which could be presented easily to the expert who's task it would be to set up and manipulate this technical specification.

The **technical specification** or **index** is Advisor's link with the associated database. It consists of a set of attribute Range value pairs which can be considered as a table as seen in Figure 16a. These attribute names are terms which are common to both the expert system and the associated database. Although the attribute names may not have the exact name as in the associated database (as terms in the Advisor can only have a maximum length of 20 characters and must contain no capitals or spaces), they have the same order

and this same order is what is important. In the case of the CAPS database for which this system has been developed, the technical specification or index is a list of the properties as seen in Appendix B. The list is in the same order in both systems to ensure that parameter values are assigned to the right slots.

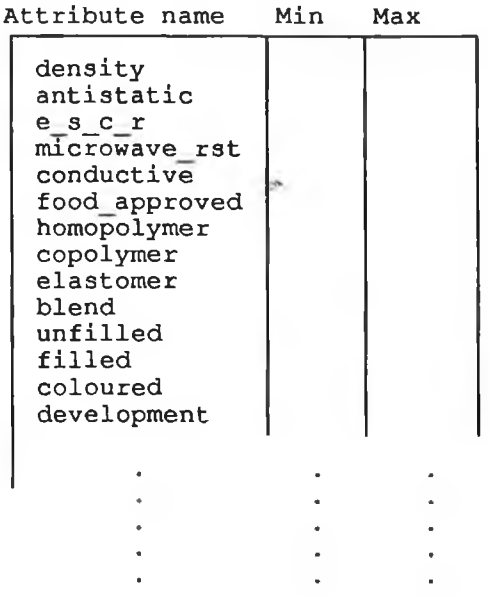

Figure 16a. The technical Specification table

Conceptually the Index structure can be thought of as being the same as a simple table consisting of three slots as seen Figure 16a. The first slot is the name of that particular polymer technical property. The second and third slot can be considered as the upper and lower range values which can be set for this particular technical property by the system as it is invoking the rules entered by the expert which affect this property. This table must be easy to create and amend by the expert and as such is presented to the expert together with all the associated actions which need in the form of a dialog box.

The index may be changed using the Change Index function on the Load popdown menu and it is the job of the expert to set up this index. When the expert invokes this

command he is presented with the **Change Index** dialog box as in Figure 16b. This dialog box enables him to insert items in the parameter list or detract items from the parameter list. There is also a function to allow him to change the name of an existing slot. All changes made to the index also affect the descriptor word list which appear in other dialog boxes such as the application descriptor. Again the function in the form of the Change Index dialog box is easy for the expert to assimilate and use.

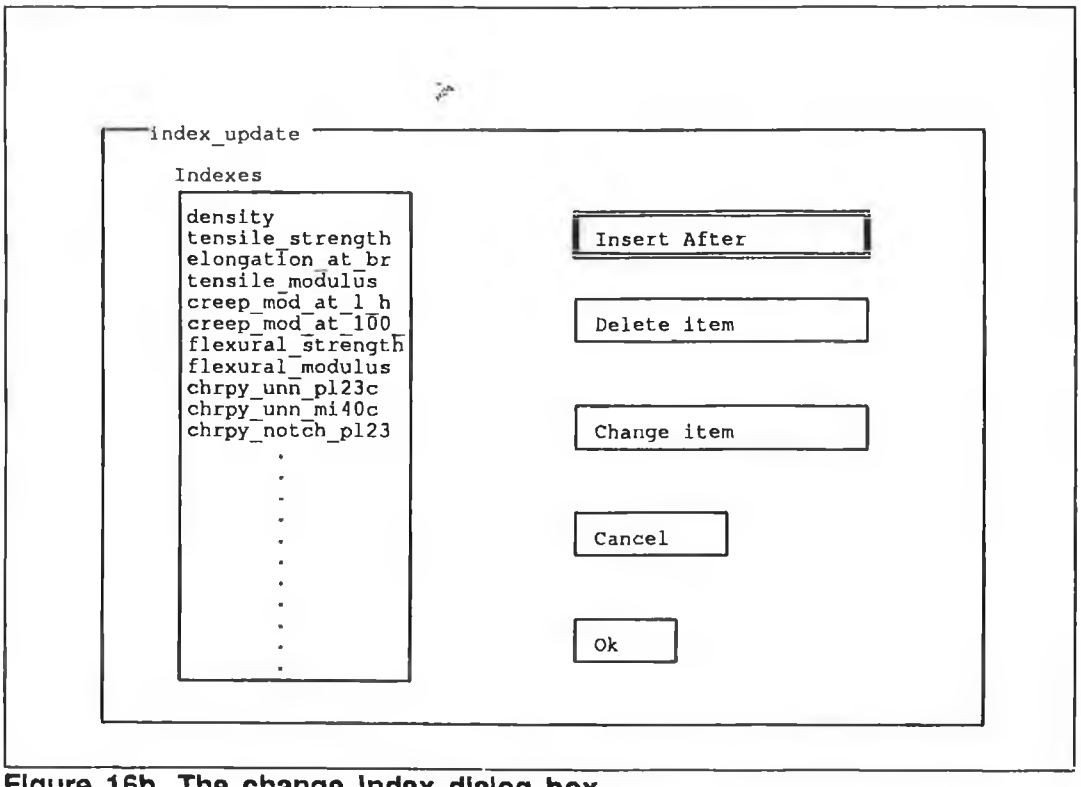

Figure 16b. The change index dialog box.

The expert can conceptually think of the index as a table of properties and their associated range values but in order to allow the system to process this list it must be represented internally in some form of data structure which Prolog can manipulate.

The Index structure in Figure 16a is actually a combination of two structures (

predicates) in Prolog the first is a predicate called index/2 (in file advindex.ari). The first argument of this predicate is the name which corresponds to the same attribute which appears in the associated database. The second argument is it's position in the index. This second argument is a numerical value which is used to find the appropriate slot in the second structure which makes up the Index. The index/2 predicate can be seen in Figure 17.

```
index(density,1).
index(tensile_strength,2).
index(elongation_at_br,3).
index(tensile_modulus,4).
index(creep_mod_at_1h,5).
        .
        .
        .
        .
        .
```

Figure 17. The Index/2 Predicate.

The second structure which makes up the Index is a clause called table/1. The only argument of the table clause is a list of lists where each inner list contains a range value argument pair as seen in Figure 18. The attribute number in the index/2 predicate is used to index into this table structure hence it's name.

```
table([A,B], [C,D], [E,F], [G,H], [I,J], [K,L], ...
      .....]).
```

Figure 18 The table/1 Clause.

5.5. Rule structure and format.

The knowledge base of an expert system usually consists of some from of rule structure. Humans find it easy to think in terms of 'If Then' rules and thus it was a suitable paradigm on which to base the Advisor shell. The fact that the users of the Advisor system in this case are inexperienced polymer designers and non-computer literate experts only serves to support this choice. Advisor's Rule Base is logic based, in that the rules and facts therein are in the form of Prolog clauses. Alan Bundy [AB87] believes that making expert systems logic based is only way to ensure that they are more correct and reliable as they are then based on the sound mathematical principles of resolution.

Because many rules could be involved in an expert system the problem of control had to be considered and dealt with in the prototype shell. This problem is greatly reduced by structuring the rule base into levels of rules and using information provided by the designer to guide the system in the right direction towards a solution. Four rule types are catered for. These are termed Super rules, Sub rules, Ordinary rules and Leaf rules. The Super rules are used to guide the inference mechanism towards a possible solution by using facts provided by the designer. These facts are provided either prior to rule activation or established by the responses of the designer to questions asked by the system about certain functional aspects of the design.

It has been attempted in the course of Advisor's construction to present the rule base to the user in an English like format, while translating these rules into Prolog to provide the powerful inference mechanism of resolution. The Prolog format of the rules is completely hidden from the expert programming the system who with the aid of the rule editor dialog box (as seen in Figure 15.) enters the rules using the quasi-English format and vocabulary which the expert has constructed. Translating these rules into Prolog rules is easy as the symbolic nature of Prolog allows the words entered by the expert to have

meaning by themselves. A rule structure is inherent in the Prolog language and this also allows easy translation of English-like rules. The built in idea of an 'is a' or 'is' concept into the reading of a Prolog clause also greatly alleviates the problem of translating these english like rules into Prolog rules.

A fact which emerged in the course of constructing a rule base was the need for levels of rules. The need for rule levels stems from the need to control Prolog's depth first search through the rules contained in the program database and from the recognised need for questioning the designer about certain functional aspect of the design.

From discussions with the polymer design experts in the early stages of development it was discovered that there would be a need to question an inexperienced designer using an expert system developed with Advisor, as the designer may unwittingly omit information which is vital to the selection of a suitable polymer (there are rules which govern the users interaction with any computer system which are to be found in [KF87] and which support this approach). So a rule level is incorporated to enable the system to gather as much information from the user as possible prior to activating the main body of the rule base.

There are four different types of rules in the Advisor system. The four different types of rules arise from the need to satisfy the following tasks:

1. To question the user to insure that all the information possible has been provided.
2. To take this information and decide which rules are appropriate for the given description.
3. To use the general information provided by the user to determine certain technical parameter values.

Both 2 and 3 above are to do with controlling the search through the rule base.

Two types of Meta rules[4] and two types of ordinary rules are provided. The use of Meta rules and ordinary rules gives rise to levels of rules which are discussed in Section 5.5.1. The rules themselves are written in an English-like format which is discussed in Section 5.5.2. These English type rules are translated into Prolog rules which are used to reach a technical specification. This is translation process is discussed in Section 5.5.3, together with the questioning facility which is built into the Meta rule layer and the trace facility which is built into all the rules in the system, during the translation process.

5.5.1. Rule Levels.

The way the rule base is structured gives rise to the idea of rule levels. The structure of the rule levels can be seen in Figure 19. At the top most level are the Super Rules. These Meta Rule types define which general design concepts are to be examined and the order in which they are to be considered. They also decide which of the Sub Rules will be called if any. The Sub Rules in turn determine which of the Ordinary Rules should be called. It should be noted that a Sub Rule can call a Leaf Rule[5] directly or else it can call another rule which will in turn call a Leaf Rule. One can have as many levels of rules inside the Ordinary Class of Rules as one wishes but the final rule in a rule chain must be a Leaf rule. Fracturing the rule base is recommended because the more fractured the rule base is the less updating one will have to do to add in new rules at later stages in the rule base development. It is also more efficient to fracture the ordinary rule level, as a large number of concise Prolog clauses leads to more efficient usage of the stack space.

By the time the Ordinary rule level is called the majority of the decisions about the

---

[4] Meta rules is a term used to describe rules about rules.

[5] A Leaf Rule is the term used to describe a rule which directly affects the Database parameter list which represented by the index structure.

functionality of the application have been made and the rule set is confined to the rules called as goals from the one Sub Rule. In this way the control problem is greatly reduced. The system only follows a predefined path through the rule base, based on the information provided by the designer. This information is provided in response to questions asked of the designer, by the system at the Super rule level. This may or may not be supplemented by facts provided in the course of describing the application using the application descriptor. This predefined path is established by the Meta rule layer (both the sub rules and the Super rules combined) and leads to a solution providing the expert has provided the appropriate Leaf rules. This method is preferable to just allowing the system to proceed on the facts presented to it as it avoids trying out every possible rule path which may go several levels down and then fail at one of the lower levels. Before the Sub Rule level can be called the Super rule level ensures the designer must provide enough information to call at least one Sub Rule.

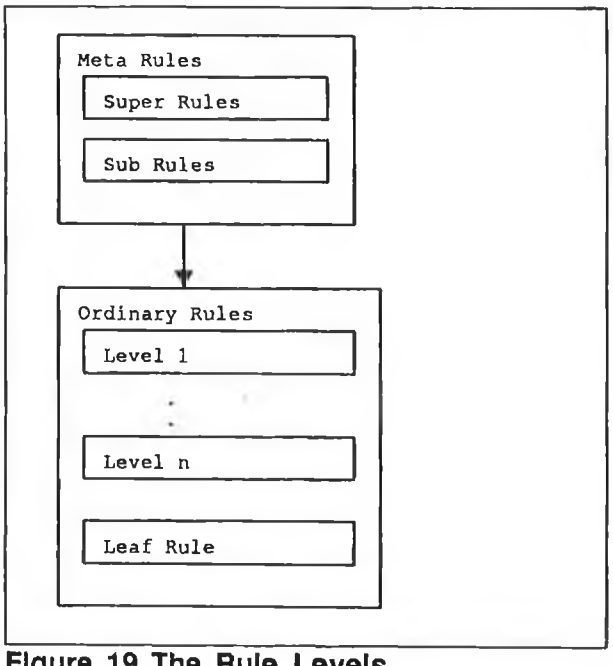

Figure 19 The Rule Levels.

Section 5.5.2 The Rule Format.

The vocabulary used in the rules is predefined by the expert with the exception of the key words 'if' and 'then' which are used to indicate where the head and body of the rule are to be found. Both the head and the body of the rules consist of clauses which have the format of the quasi-English natural language described earlier.

Attention is first turned to the Meta rule level. The two types of Meta Rules to be discussed here are termed **Super Rules** and **Sub Rules**. The Meta rule level serves two purposes. Firstly rules in the Meta rule level define which rules are to be activated at the lower levels for a given set of circumstances. This is the task of the Sub Rules. Secondly the Meta level rules serve as a means of trying to extract the maximum amount of information from the user prior to activating a line of reasoning. This is the task of the Super Rules. As the rules are translated into Prolog clauses order is important because of the sequential nature of Prolog's search through the program database. This fact affects the structure of the Meta rules.

Super Rules.

Super rules consist of a head and a body separated by the key word 'then'. Within the structure of the Head is just a single goal which identifies the rule as being a Super rule. It says that if the start is caps then the following properties should be considered and values should be established for these properties before proceeding further.

```
if start is caps then
    apptype is vvar and
    instance is vvar and
    usage is vvar and
    environment is vvar .
```

Figure 20 A Sample Super Rule

The body of a Super rule (see Figure 20) consists of one or more goals which have to be solved before the system can proceed further. The vvar stands for "variable value" and tells the shell when it is compiling the rule to create a unique Prolog variable value for this property. The value will be assigned to this property when the user of the expert system states the fact with the application descriptor or when a value is assigned to the property in response to a question asked by the system. Because of the sequential nature of the rules in the rule base it is necessary to enter the Meta rules in decreasing order with respect to the amount of information to be gathered. For example if the first rule in the rule base was that which is in Figure 20 above then the structure of the rest of the Super rules would be as seen in Figure 21.

```
if start is caps then apptype is vvar and
   instance is vvar and usage is vvar and
      environment is vvar .

if start is caps then apptype is vvar and
   instance is vvar and usage is vvar.

if start is caps then apptype is vvar and
   instance is vvar.

if start is caps then apptype is vvar.
```

Figure 21. The Super rule order.

This rule ordering is necessary to ensure that the questioning facility (which is built into the Super rules while they are being translated into Prolog rules) does not repeatedly ask the same question for a value which the user has specified he/she does not know. Also within the Super rule structure the person building the rules should ensure that the most important properties are considered first in each rule. This factor too can reduce the number of questions asked by the system at the Super rule level.

However the situation can still arise were the system will repeatedly ask for a value regardless of the structure of the Super rules. If the user says that he/she does not know one of the property values which appears in two consecutive rules then the system will attempt to satisfy this goal again in the second of the two rules. This is why those properties which the user should know should have precedence in each Super rule. For example in Figure 21 above the most important property is the application type (apptype) so this is placed first in every rule. If the user answers don't know to the question : what is the value for the application type ? in the first rule then the questioning mechanism will try again to establish the value when it tries the second Super rule. This is how Prolog's search mechanism works. If it cannot solve one of the goals on the right hand side of the rule it backtracks to try the next rule in the rule base which matches the rule head.

There would be nothing to stop the person programming the rule base from inserting more Super rules to deal with the case were the application type is not known. This would simply be done by extending the Super rule level to include rules were application type does not come into play. But this fact would not stop the system from repeatedly questioning the user about this important property's value. Thus there is a trade off between having a Meta rule level with a questioning mechanism and trying to establish a solution just based on the description given by the user of the system. The advantage of the non-questioning method is that the user is not continually asked questions (which would speed up the consultation process). But this method suffers from the fact that the system is more likely to return failure as the user can neglect to provide the proper information. The advantage of the method incorporating the questioning mechanism is that the system will attempt to establish more information from the user than he/she has provided.

However repeated questioning for a value which the user specifies as not known

is a problem which would be difficult to solve given the structure of the rule base as a whole. If the approach was taken were a "don't know" answer resulted in a Prolog variable remaining uninstantiated, then this variable would match any atom in the head of a Sub rule which could lead to a false technical parameter list being established. One possible solution to the problem would be to provide a 'dont_know' value in the value word list (from which the answers to the questions are selected) and provide a Sub rule to deal with that value being assigned to that slot.

Sub Rules.

The **Sub Rules** are a mirror image of the Super Rules above in that there should be at least one Sub Rule for every rule present in the Super Rule layer. It is the purpose of the Sub Rules to establish which rules should be fired based on the information provided by the user of the expert system. The structure of a Sub Rule can be seen in Figure 22.

```
if pvar is dishes and pvar is glass and pvar
  is disposable and pvar is ship then
  specific_properties of dishes and
  specific_properties of glass and
  specific_properties of disposable and
  environment is ship .
```

Figure 22 A Sample Sub Rule.

The head consists of one or more values being assigned to a variable 'pvar' (property variable). It is important that the values in the head match the order of it's associated Super Rule, otherwise the system will not find the Sub Rule when it has the values. These Sub Rules tell the system that given the specified values (which will have been established through the use of the matching Super Rule in the layer above), the following rules should be invoked. The rules to be called form the body of the rule. Given the Super Rule structure in Figure 21 above the Sub Rule structure should be the same as that seen in Figure 23.

One of the reason for having two types of Meta Rules is to increase the adaptability of the Meta Rule level. For example in one instance the application type might be dishes. In another instance an application type of say a kitchen appliances might be under consideration. In the Super Rule level one could define one Super rule to establish the value for application type and the parameters common to each. In the Sub Rule level there could be two rules which would stipulate which rules would be called given one of the specified application types. This in turn would determine which Ordinary Rules would called at the lower levels. So that once the application properties which need to be considered for all applications have been identified, it is just a matter of updating the Sub Rule level and the Ordinary rule levels for new application types.

```
if pvar is dishes and pvar is glass and pvar
 is disposable and pvar is ship then
 specific_properties of dishes and
 specific_properties of glass and
 specific_properties of disposable and
 environment is ship .

if pvar is dishes and pvar is glass and pvar
 is disposable then
 specific_properties of dishes and
 specific_properties of glass and
 specific_properties of disposable.

if pvar is dishes and pvar is glass then
 specific_properties of dishes and
 specific_properties of glass.

if pvar is dishes then
 specific_properties of dishes.
```

Fig 23 The Sub rule order

The introduction of a Meta rule level is in fact a way of allowing the Programmer of the rule base to direct Prolog's inference mechanism toward a solution using responses

to requests for the values from the system and/or the description of the application provided. Furthermore it ensures that an expert system's[6] ability to reach a solution degrades gracefully. Provided the rule base is programmed correctly then the system should be able to establish a solution even if only one fact is presented to it.

Ordinary Rules.

The next class of rules are the **Ordinary Rules**. The ordinary rules specify which of the CAPS parameters (as defined in the Index structure See section 4.2.) are affected by the functional characteristics and other properties of the application being described. There are some restrictions on Ordinary rules in the Advisor system. Firstly there can only be one condition on the left hand side of the rule. The reason for this restriction is to keep the rule semantics as simple as possible, when translating the rules into Prolog. In any case the compounding of conditions should have been dealt with within the Meta level rules. There can be many layers of Ordinary rules but a particular rule chain or line of reasoning must end in what is known as a **Leaf Rule**.

A Leaf Rule is a rule which directly affects a CAPS parameter list item as defined in the Index. Leaf Rules are distinguished from other ordinary rules by the use of arithmetic operators (e.g. '=' , '<' , '>' etc.) in clauses on the right hand side of the rules. See Figure 24 for an example of a Leaf Rule. The term 'leaf rule' arises from the fact that this rule is the terminal node in the depth first search tree which Prolog carries out as it searches through the rule base (Program database).

[6] That is an expert system developed using Advisor.

```
if specific_properties of glass then
  alcohols = yes and
  high_gloss = yes and
  contin_serv_temp > 100 and
  unfilled = yes .
```

Figure 24. A Sample Leaf Rule.

Other Ordinary Rules use other operators (e.g. 'is' , 'to', 'of' etc.) in clauses on the right hand side of rules to assign values to parameters which are not CAPS parameter list items but are in fact goals which call other rules. These intermediary rules define path to the Leaf Rules discussed above and only serve as a means of structuring the knowledge base. An example of such a rule chain can be seen in Figure 25. The first rule effectively calls the second rule which in turn sets the CAPS parameter easy_flow to a range between 1 and 5.

```
if specific_properties of disposable then
  price is low .

if price is low then
  easy_flow < 5 and
  easy_flow > 1 .
```

Figure 25. A sample Rule Chain.

The order of the Ordinary rules is not as important as in the Meta rule levels, but it can be used to the programmers advantage. By placing two rules (with the same head) consecutively in the rule base one can deal with cases were a rule does not apply (if the first rule fails then the second rule will be activated).

Section 5.5.3. Rule Translation.

Before the rules can be activated in the expert system they must be translated into their Prolog equivalent. The programmer defined predicate commit/2 (see listing of advcomit.ari in Appendix A) is used to translate the rules from their English format into a Prolog format. An example of the English format of the rules can be seen in Figure 26. The corresponding Prolog format can be seen in Figure 27 and Figure 28. All Rules at all levels in the translated form (Figure 27 and figure 28) of the rule base have two variable Parameters in common in the head of the rule, namely X0 and Xn. X0 denotes the list of technical parameters (to be passed to the associated database) before calling that particular rule. Xn denotes the list when the rule has been successfully completed (proved). The list initially consists of an empty (uninstantiated) slot for each attribute value pair of a property named in the index list or table. Each slot is itself a list of two elements which are initially uninstantiated.

The Super Rules (see Figure 27)

The caps/2 rule is the Prolog equivalent of a Super rule. The Rule caps/2 [7] is the entry point to the rule base. When it is being created the first goal is a call to the trace/2 predicate (this predicate is built into the Advisor system and can be seen in 'advrulgo.ari' in Appendix A). This goal indicates to the tracing mechanism that the rule is being called. It contains the English format of the compiled rule (the second argument). The trace predicate is used to record the current rule being fired and the rules which have been called so far. This information is used in response to Why? and How? questions from the user. It is also used by the Debugger mechanism when activated.

---

[7] caps/2 stands for the clause having functor caps and having two arguments.

The next group of goals are the properties for which values need to be established in order to successfully activate the Sub rule layer. As can be seen from Figure 27 and Figure 26 the vvar word is translated into a unique Prolog variable[8] (within that rule). These are simple goals which are solved by facts which the user has provided in the form of a application description or in the form of answers given in response to the questioning facility (described below) initiated by the system.

The call to the sub rule level is through the goal **'docaps'**. The docaps goal has a variable number of arguments which have been instantiated to values which have been provided by the previous goals. The last two arguments in the docaps goal are X0 and Xn (the list before and after the goal is called). Finally the second trace goal is added to indicate that the rule has been successful.

The Sub Rules (see Figure 27).

The docaps predicate is the Prolog representation of a Sub rule. The docaps predicate has a variable number of arguments which are atoms, and which must match those established in the corresponding Super rule in the rule level above. The first goal on the right hand side of the sub rule is again a call to the built in trace/2 function. The remaining goals are calls to the lower level Ordinary Rules.

Each successive goal processes the parameter list as returned from the previous goal to yield a new parameter list which has been affected by the Leaf Rules in the Ordinary Rule level. X0 is processed to yield X1, X1 is processed to yield X2 and so on, until Xn is finally established and the whole rule succeeds. These goals can be calls to Leaf rules directly or to rules which will in turn call other Ordinary rules (ultimately ending in a leaf

[8] Prolog variable are denoted as an atom beginning with an upper case letter.

rule, otherwise the list remains unchanged).

The Ordinary rule level (see Figure 28).

Ordinary rules have a head which consists of a functor which is a general descriptor name and three arguments. The first argument is the general value which has been assigned to a descriptor name. The next two arguments are the parameter list before and after the rule has been invoked i.e.

```
decriptor1(value1,X0,Xn) :-
        trace(call,.....),
        arithmetic_name(descriptor2,value2,X1,X2),
        descriptor3(value3,X2,Xn),
        trace(true,.....).
```

The trace goals are again added to the right hand side during construction of the rule. Two type of goals can be present on the right hand side of an ordinary rule. A general goal is just a call to another Ordinary rule at the next level down. A **Leaf Goal** on the other hand is a call to a predicate which actually built into the system. There are built in predicates (which perform actions on the parameter list) for each of the permissable arithmetic assignment operators. These predicates are contained in the file 'advtable.ari' which can be seen in Appendix A. As an example, taking the 'density = 1', this would be translated in to the Prolog goal 'eq(density,1,X1,X2)' were X1 would be the parameter list passed on by the previous goal and X2 would be the new parameter list which has been affected by the eq/4 predicate. The arithmetic predicates assign values to the parameter list slots based on number line arithmetic implemented as Prolog Rules (see 'advtable.ari' in appendix A). Number line arithmetic is used to deal with cases were conflicting range numeric values are being assigned to the same technical specification property. In such cases the system will select a numeric range which encompasses both numeric ranges if it is valid to do so (The predicates which operate this number line arithmetic can also be seen in 'advtable.ari').

Super Rules

```
if start is caps then  % Rule1
apptype is vvar and
instance is vvar and
usage is vvar and
environment is vvar .

if start is caps then  % Rule2
apptype is vvar and
instance is vvar and
usage is vvar .

if start is caps then  % Rule3
apptype is vvar and
instance is vvar .

if start is caps then  % Rule4
apptype is vvar .
```

Sub Rules

```
if pvar is dishes and pvar is glass and pvar is disposable and pvar is ship then  % Rule5
specific_properties of dishes and
specific_properties of glass and
specific_properties of disposable and
environment is ship .

if pvar is dishes and pvar is glass and pvar is disposable then  % Rule6
specific_properties of dishes and
specific_properties of glass and
specific_properties of disposable .

if pvar is dishes and pvar is glass then  % Rule7
specific_properties of dishes and
specific_properties of glass .

if pvar is dishes then                    % Rule8
specific_properties of dishes .
```

Ordinary Rules

```
if specific_properties of dishes then     % Rule9
alcohols = yes and
detergents = yes and
food_approved = yes and
boiling_water = yes and
antistatic = yes and
flame_retardant = yes .

if specific_properties of glass then      % Rule10
alcohols = yes and
high_gloss = yes and
contin_serv_temp > 100 and
unfilled = yes .

if specific_properties of disposable then  % Rule11
price is low .

if price is low then  % Rule12
easy_flow < 5 and
easy_flow > 1 .

if glass_sphere is yes then  % Rule13
transparent = yes .

if environment is ship then  % Rule14
density < 1 .

if environment is petrol_station then % Rule15
motor_oil = yes and
petrol = yes and
density = 1 .
```

Figure 26. The sample Rule base.

Super Rules

```
caps(X0,Xn) :-  % Rule1
[!trace(call,$if start is caps then apptype is vvar and instance is vvar and usage is vvar and environment is vvar
.$)!], [!apptype(X1)!],[!instance(X2)!],[!usage(X3)!],[!environment(X4)!],
docaps(X4,X3,X2,X1,X0,Xn),
trace(true,$if start is caps then apptype is vvar and instance is vvar and usage is vvar and environment is vvar .$).


apptype(X) :- default(apptype,X). % A Call to the default questioning mechanism

caps(X0,Xn) :- % Rule2
[!trace(call,$if start is caps then apptype is vvar and instance is vvar and usage is vvar .$)!],
[!apptype(X1)!],[!instance(X2)!],[!usage(X3)!],
docaps(X3,X2,X1,X0,Xn),
trace(true,$if start is caps then apptype is vvar and instance is vvar and usage is vvar .$).

instance(X) :- default(instance,X).

caps(X0,Xn) :- % Rule3
[!trace(call,$if start is caps then apptype is vvar and instance is vvar .$)!], [!apptype(X1)!],[!instance(X2)!],
docaps(X2,X1,X0,Xn),
trace(true,$if start is caps then apptype is vvar and instance is vvar .$).

usage(X) :- default(usage,X).

caps(X0,Xn) :- % Rule4
[!trace(call,$if start is caps then apptype is vvar .$)!],
[!apptype(X1)!],
docaps(X1,X0,Xn),
trace(true,$if start is caps then apptype is vvar .$).

environment(X) :- default(environment,X).
```

Sub Rules

```
docaps(ship,disposable,glass,dishes,X0,Xn) :- % Rule5
[!trace(call,$if pvar is dishes and pvar is glass and pvar is disposable and pvar is ship then specific_properties of
dishes and specific_properties of glass and specific_properties of disposable and environment is ship .$)!],
specific_properties(dishes,X0,X1),
specific_properties(glass,X1,X2),
specific_properties(disposable,X2,X3),
environment(ship,X3,Xn),
trace(true,$if pvar is dishes and pvar is glass and pvar is disposable and pvar is ship then specific_properties of
dishes and specific_properties of glass and specific_properties of disposable and environment is ship .$).

docaps(disposable,glass,dishes,X0,Xn) :- % Rule6
[!trace(call,$if pvar is dishes and pvar is glass and pvar is disposable then specific_properties of dishes and
specific_properties of glass and specific_properties of disposable .$)!], specific_properties(dishes,X0,X1),
specific_properties(glass,X1,X2),
specific_properties(disposable,X2,Xn),
trace(true,$if pvar is dishes and pvar is glass and pvar is disposable then specific_properties of dishes and
specific_properties of glass and specific_properties of disposable .$).

docaps(glass,dishes,X0,Xn) :- % Rule7
[!trace(call,$if pvar is dishes and pvar is glass then specific_properties of dishes and specific_properties of glass
.$)!], specific_properties(dishes,X0,X1),
specific_properties(glass,X1,Xn),
trace(true,$if pvar is dishes and pvar is glass then specific_properties of dishes and specific_properties of glass .$).


docaps(dishes,X0,Xn) :-  % Rule8
[!trace(call,$if pvar is dishes then specific_properties of dishes .$)!],
specific_properties(dishes,X0,Xn),
trace(true,$if pvar is dishes then specific_properties of dishes .$).
```

Figure 27. The Meta Rules (Prolog form).

Ordinary Rules

```
specific_properties(dishes,X0,Xn) :- % Rule9 (level1 rule and leaf rule)
[!trace(call,$if specific_properties of dishes then alcohols = yes and detergents = yes and food_approved = yes and
boiling_water = yes and antistatic = yes and flame_retardant = yes .$)!], eq(alcohols,yes,X0,X1),
eq(detergents,yes,X1,X2),
eq(food_approved,yes,X2,X3),
eq(boiling_water,yes,X3,X4),
eq(antistatic,yes,X4,X5),
eq(flame_retardant,yes,X5,Xn),
trace(true,$if specific_properties of dishes then alcohols = yes and detergents = yes and food_approved = yes and
boiling_water = yes and antistatic = yes and flame_retardant = yes .$).

specific_properties(glass,X0,Xn) :- % Rule10
[!trace(call,$if specific_properties of glass then alcohols = yes and high_gloss = yes and contin_serv_temp > 100
and unfilled = yes .$)!],
eq(alcohols,yes,X0,X1),
eq(high_gloss,yes,X1,X2),
grt(contin_serv_temp,100,X2,Xn),
eq(unfilled,yes,X3,Xn),
trace(true,$if specific_properties of glass then alcohols = yes and high_gloss = yes and contin_serv_temp > 100
and unfilled = yes .$).

specific_properties(disposable,X0,Xn) :- % Rule11
[!trace(call,$if specific_properties of disposable then price is low .$)!],
price(low,X0,Xn),
trace(true,$if specific_properties of disposable then price is low .$).

price(low,X0,Xn) :- % Rule12
[!trace(call,$if price is low then easy_flow < 5 and easy_flow > 1 .$)!], lst(easy_flow,5,X0,X1),
grt(easy_flow,1,X1,Xn),
trace(true,$if price is low then easy_flow < 5 and easy_flow > 1 .$).

glass_sphere(yes,X0,Xn) :- % Rule13
[!trace(call,$if glass_sphere is yes then transparent = yes .$)!],
eq(transparent,yes,X0,Xn),
trace(true,$if glass_sphere is yes then transparent = yes .$).

environment(ship,X0,Xn) :- % Rule14
[!trace(call,$if environment is ship then density < 1 .$)!],
lst(density,1,X0,Xn),trace(true,$if environment is ship then density < 1 .$).

environment(petrol_station,X0,Xn) :- % Rule15
[!trace(call,$if environment is petrol_station then motor_oil = yes and petrol = yes and density = 1 .$)!],
eq(motor_oil,yes,X0,X1),
eq(petrol,yes,X1,X2),
eq(density,1,X2,Xn),
trace(true,$if environment is petrol_station then motor_oil = yes and petrol = yes and density = 1 .$).
```

Figure 28. The Ordinary Rules (Prolog form).

Section 5.5.4. The Inference Mechanism.

It is important when constructing a rule base to keep in mind the inference mechanism employed by Prolog as this is also used by the Advisor shell. The inference mechanism is based on the principles of "resolution refutation" first introduced by Robinson [AJR65]. A full description can be found in [AJR79]. The Prolog inference mechanism incorporates a backward chaining method which requires the programmer (i.e. the expert) of a rule base to think of the "if-then" rules in a different light. The rules should be thought of as being of the from " IF the head of the rule is to be true THEN the following goals must be proved to be true". Such rules are to be found in expert systems which have been constructed using shells such as Crystal[9] [RW88] and EMYCIN[10] [BS86] (one of the first expert system shells to be developed). This way of thinking about rules of this form is different to the forward chaining approach (which is the way people would normally consider an "IF THEN" rule). If the inference mechanism was forward chaining then the rules would be considered as "IF the conditions are true THEN take the following actions". This approach is to be found in such systems as R1 [PJ86].

The inference mechanism in the context of Advisor works (See Figure 26 for sample rule base) as follows.....

1. The system starts by searching the rule base for the first Super Rule (Rule1 - Rule4) or starting rule. It finds it at rule1.

2. It then tries to solve each of the goals on the Right Hand Side (RHS) of the rule. If a

---

[9] In fact the Crystal Rule format is in the form "X is Y and... IF W is Z and..."

[10] EMYCIN came about as a result of generalising the principles on which the MYCIN diagnosis system was based.

value has not been supplied for one of the property names on the RHS of the rule then it will ask the user to specify a value. If the user does not know it will try the next rule down which would be rule2. If the user continues to say don't know in response to a question the system will eventually run out of Super Rules at which stage the system will return a message that it has failed to reach any conclusion.

3. As soon as all the values are established for the items on the RHS of a Super Rule the system moves down to the Sub Rule level (Rule5 - Rule8).

4. It tries to find a Sub Rule whose Left Hand Side (LHS) matches (this should always be the case if the rule base has been programmed correctly and completely) the values which have been assigned in the Super Rule level.

5. If it finds one it will attempt to solve the goals in the order left to right on the RHS of that rule, in attempting this solution it will call rules which will be found at the Ordinary Rule level (Rule9 - Rule15).

6. If any of the rules fail at any level the system will proceed in the following manner :

(i). First it tries to find another match for the rule at the current level.

(ii) If it cannot find a solution at the current level it will move up a level to the calling rule and try and find a alternative solution at that level.

(iii) If it keeps on failing it will keep moving back up levels an try new options at each level until it reaches the Super Rule level. If all the rules at the Super Rule level fail the whole query will fail.

This is a modified version of depth first search or backward chaining mechanism which is maintained by the system. For example given the rule base in Figure 26. If the user provides the Following information the system will follow the path shown in Figure 29.

Information provided :

1. apptype is dishes.
2. instance is glass.
3. usage is disposable.
4. Do not know the environment.

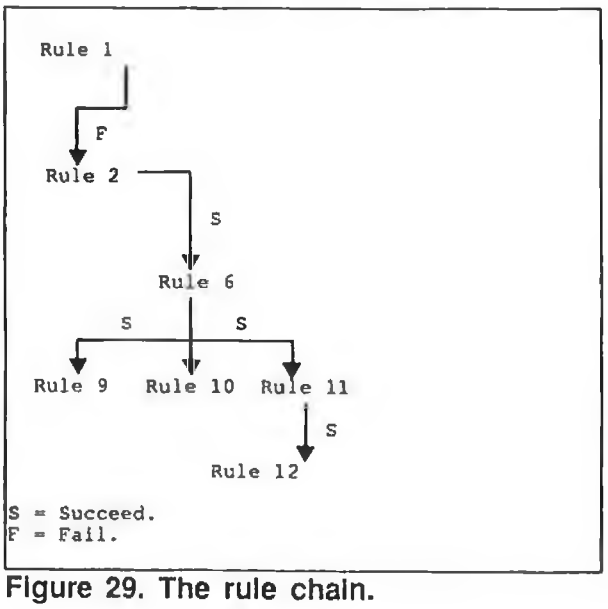

Figure 29. The rule chain.

5.6. Questions, Answers and customised explanations.

The problem of providing a facility which allows the system to question the designer about certain important aspects of the design and the problem of the system explaining it's actions needed to be addressed in the construction of the Advisor system. The need for the system to question the designer when it cannot proceed further with the facts presented to it and the need to explain certain aspects of the solution and how it is achieved are best solved by a system based on a logic rule based paradigm. This view was supported by the prototype system developed. The rules provide the source of the aspects the about which the system needs to question designer, if he/she has not provided the appropriate information. The ability of the system to question a designer about certain functional attributes of an application is built into the Super rule structure as it is being translated from it's quasi-English form into a Prolog equivalent. Because the translated rules form part of the overall Prolog based shell the questioning mechanism contained within the rule structure can use the graphical feature of dialog boxes built into the shell to present the question and the set of valid answers. This fact is important form the point of view of presenting the questions to the designer in a concise and unambiguous form. The rules also provide a ready means of explaining how and why a solution is reached by quoting all the rules which have been used to reach the solution. This ability to explain the systems actions is also incorporated into the rule structure as it is being translated from it's quasi-English form into it's Prolog equivalent. This is achieved by adding special trace predicates to the compiled form of the rules as has been seen. A need is also recognised to explain to the designer the significance of certain values being assigned to technical properties. This need is fulfilled through allowing the expert to set up customised explanations and allowing the designer to easily invoke these explanations when the eventual solution is reached.

5.6.1. The Questioning Facility.

For each new descriptor that the system comes across while constructing a Super rule, it automatically creates a rule which will call the **default questioning facility** to establish a value for that descriptor should the user fail to provide one in the application description. The format for this rule is

```
descriptor_name(X) :- default(descriptor_name,X).
```

The default/2 predicate is built into the system and can be seen in the program listing 'advrulgo.ari' in Appendix A. This default predicate runs the questioning dialog box as seen in Figure 30. which asks the user to choose the value to be assigned to the unestablished descriptor from a list of all the currently defined value words (again the word lists are used). This value is unified with the variable X and the appropriate variable in the calling goal in the Super rule. The fact which states that this value is assigned to the descriptor is also asserted in the program database. The order in which facts are added (after the application description has been given or a value has been supplied using the questioning dialog box) insures that this descriptor will not be questioned again if the current rule fails. For example if the value 'dishes' is assigned to the descriptor 'apptype' this fact will be asserted at the beginning of the clauses for the apptype e.g.

```
apptype(dishes).
apptype(X) :- default(apptype,X).
```

All the goals preceding the docaps goal in the translated Super rule are surrounded by snips. This ensures that if the docaps goal fails, (i.e. if the system cannot find an exact match for the given description) the system will not try to reestablish these facts during back tracking, and possibly ask for a value which the user has already provided.

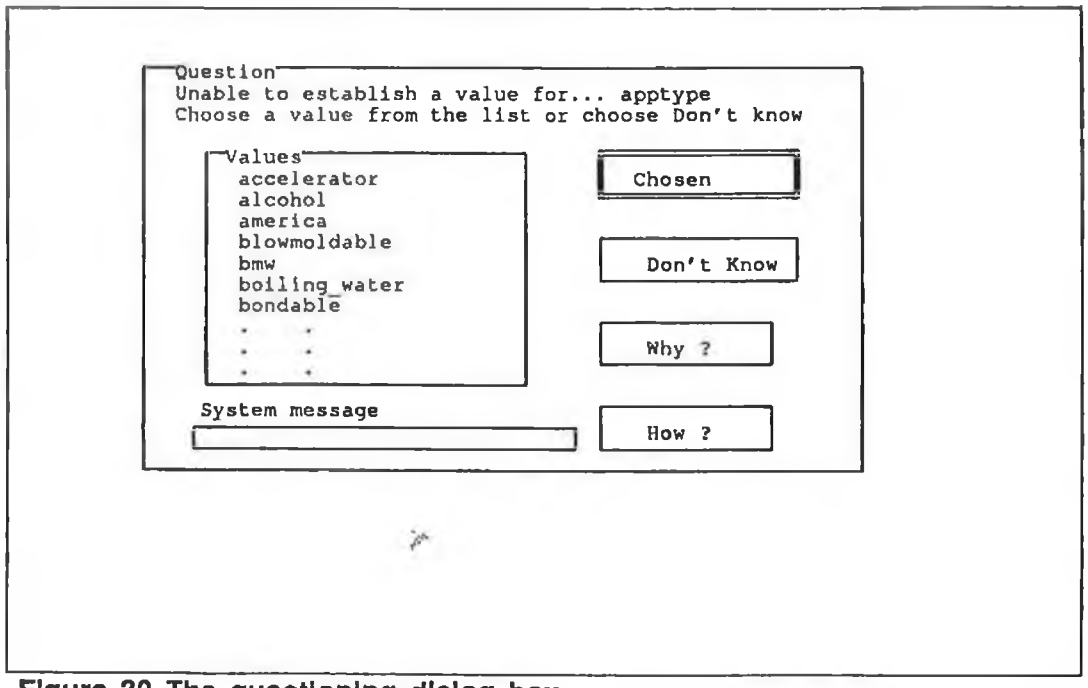

Figure 30 The questioning dialog box

5.6.2. Answers.

As well as questioning the designer about certain aspects of the design the system must give the designer the opportunity to question the decisions of the system in order to gain an insight into the polymer design process. This ability of the designer to question the actions of the system is incorporated into the question dialog box in the form of the How? and Why? push buttons in the question dialog box and in the solution dialog box (see Figure 31), in the form of the Because edit box and the explain push button. In this way the system can explain it's actions to the designer who should as a result gain an insight into the process of designing in polymers.

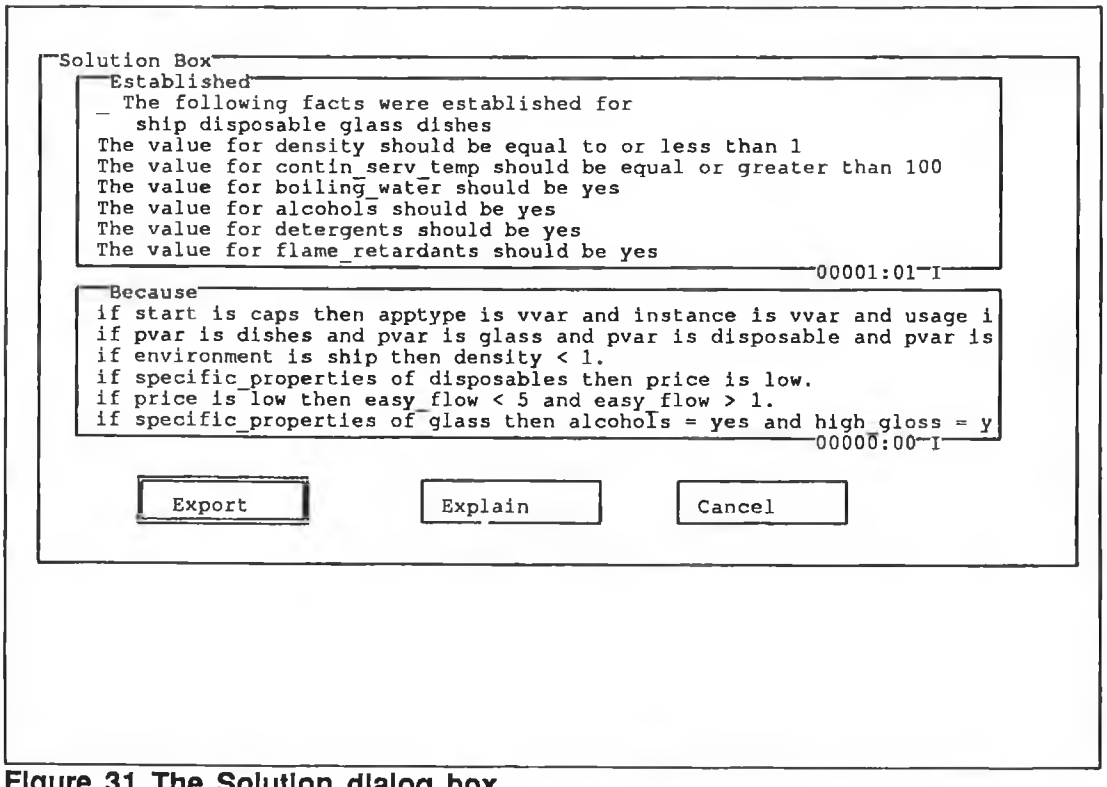

Figure 31 The Solution dialog box.

The information presented in response to the activation of the How?, Why? and in

the Because edit box is in the form of quoted rules. These rules are recorded using the trace predicate which is built into the system. The trace predicate is called as a goal from the right hand side of a rule. As it is ensured that the trace goal never fails it has no effect on the action of the rules and it's capacity is strictly one of reporting the systems actions.

In response to invoking the Why? push button in the question dialog box the designer is shown the rule which the system is currently trying to solve along with all the facts established so far. In response to the How? command the system responds again with the facts established so far together with all the rules used to get to the current state of the solution. The Why? and How? mechanisms can only be used if the system actually asks a question, which of course only occurs at the Super rule level. For this reason the trace mechanism is included as a further means of explaining the systems actions. On invoking the trace mechanism the designer can see each rule that the system calls and can determine which rules succeed and which rules fail. The trace mechanism allows all the rules in all the rule levels to be viewed as they are called.

The Because edit box in the solution dialog box is an expansion on the How? mechanism in the question dialog box. The contents of the Because edit box is a record of all the rules which have succeeded in order to reach the final solution which is displayed in the established edit box in the solution dialog box.

5.6.3. Customised explanations.

As well as a need to get the system to explain it's own actions a need exists to explain the significance of certain values being assigned to certain parameters (those mentioned in the established edit box) in the technical parameters. This facility is set up by the expert when he/she is setting up the rule base and the vocabulary with the aid of the explain index dialog box (see Figure 32).

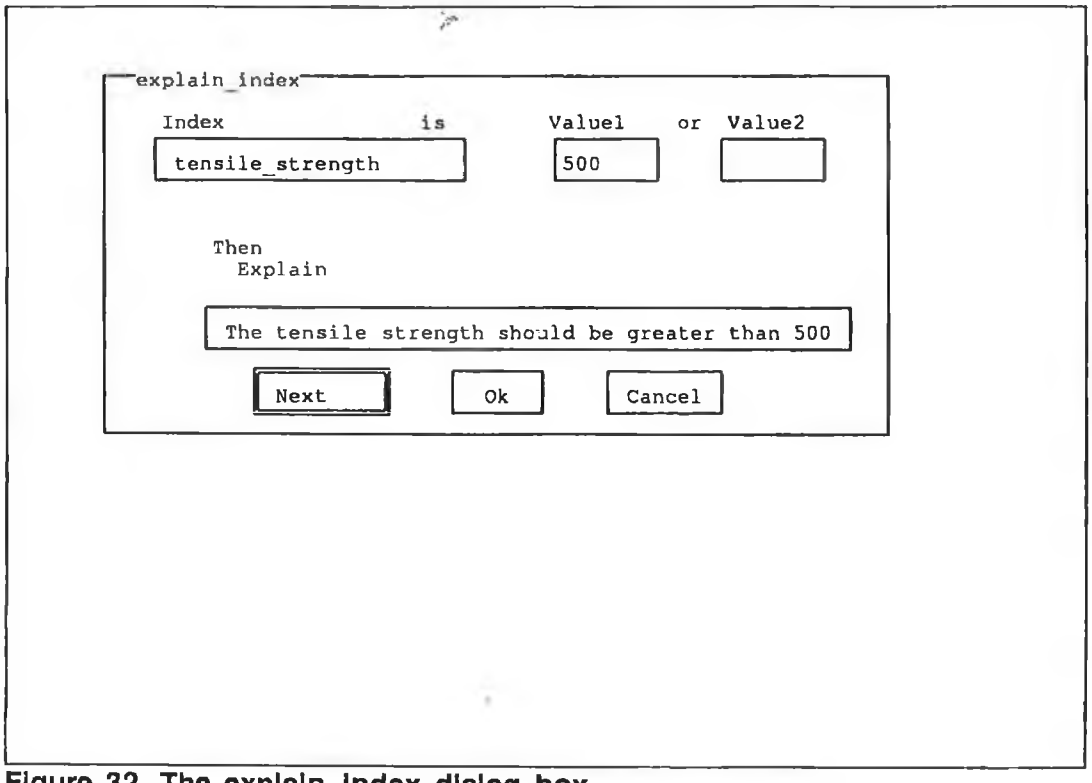

Figure 32. The explain_index dialog box

To set up customised explanations is simple to do using this explain_index dialog box. The edit fields labelled **Index, Value1, Value2** are used to stipulate the value range for a technical parameter for which the textual explanation is to be set up. The first value

slot Value1 represents the lower range value of the two values and the second value slot Value2 represents the upper range value (if two are provided).

A numeric value in the Value1 edit field and no value in the Value2 edit field indicates that the range value is greater or equal to this number in the Value1 edit field. Similarly if the Value1 edit field is empty and there is a number in the Value2 edit field this interpreted by the system less than or equal the Value2 number when the system is using the customised explanation information in the solution box. Equivalence is accounted for by having the same number in both the Value1 and Value2 edit fields. If either edit field contains a non numeric value then the it is taken to mean equal to this string constant.

The **Explain** edit field is used to enable the user to type in text to explain the significance of this value assignment to this Index Value. This line of text can be up to 250 characters in length and contain any sort of information which the expert might consider worth knowing.

The explanations are stored in the database in the form of Prolog fact which consists of the a slot for the name of the index attribute, two slots for the possible values to be assigned to this named attribute and a further slot which contains the text which explains the consequences of this value(s) being assigned to this attribute e.g.

explain(tensile_strength,500,_,$ The tensile strength should be greater than or equal to five hundred as the application will be subjected to large stress force, tensile strength is measured in Meters Per atmosphere.$).

These explanation are presented to the designer in response to the Explain command in the Solution dialog box and allow the designer to see the importance of the technical parameter values being set by quoting the parameter and the value assigned to it together

with the text which the expert has entered with the aid of the explain index dialog box.

5.7. Summary.

The Advisor system provides the means to support a flexible quasi-English natural language interface and presents this language to the user in the compact and efficient graphical form of dialog boxes. The application descriptions are built up easily by the designer in this quasi-English natural language using terms displayed in the lists present in the Application descriptor dialog box. The vocabulary being used by the designer having being previously defined by the expert.

The iterative process of building up the vocabulary to be used in the system and the rules is well supported. The expert can easily amend the vocabulary used in the system using the add and delete dialog boxes. He/she can also easily create and amend a rule base using the rule editor dialog box which incorporates a full screen editor and a rule parser to check the syntax of the rules contained therein.

The problem of controlling the search through the rule base is greatly alleviated by introducing the notion of rule levels. A rule can not proceed further than the top level unless certain essential facts about the design have been established. Once it does proceed to the next level the path through the rule base is already established as the next level decides which rules will be called given the facts presented at the level above.

The need to question the designer about certain aspects of the design is also included in the rule structure were just by the expert mentioning the important design aspects in the Super rules, the system automatically ensures that if the designer has not supplied a value for this important design concept then the system will question the designer about the concept.

The designer is able to examine the rule base while the system is working toward a solution and when it has reached a solution. This is done through the use of the How? and Why? mechanisms, the trace facility and the contents of the because edit box in the solution dialog box. Also the expert can build up customised explanations easily using the explain index dialog box. The designer can use these customised explanations to get a deeper insight into the consequences of values being assigned to technical parameters. These facilities fulfil the systems requirements to explain to the designer the process of reaching a technical specification.

In conclusion the Advisor system satisfactorily fulfils all the requirements of the proposed open-ended expert system as it allows the designer and the expert to work with the system using their own terms which represent concepts which are familiar to both and it allows the designer to describe an application in his/her own vague terms and use the expert knowledge encoded in the form of rules not only to reach a technical specification but also to gain a deeper insight into the process of design using polymers.

Chapter 6

Conclusions and

Future Work.

Chapter 6 Conclusions and Future work.

6.1. Conclusions.

Design using Polymers proved to be a suitable domain in which to apply current expert system technology (CEST) techniques. The Advisor prototype system was constructed in support of this fact.

A logic Rule based approach, such as the one incorporated in Advisor is more than sufficient to encode the knowledge of the expert in area of design using polymers. The need to provide explanations and to question the designer is also supported using a rule based paradigm.

The problem of controlling the search through the rule base gave rise to both production rule and backward chaining approaches being tried. The production rule approach proved to be inadequate (in terms of keeping track of the rules and controlling the rule firing cycle which is a feature of production rule systems) in the context of the design problem. The backward chaining approach proved to be more useful in terms of keeping track of the rules as they are activated and controlling the rule activation. This is mainly due to the fact that backward chaining is also the inference mechanism used by Prolog.

Advisor was constructed to overcome some deficiencies which currently available expert system shells possess. The major disadvantage being that they are not easily programmed by the expert. Expert systems should be constructed by the expert in keeping with the trend in CEST which is moving away from the knowledge engineer/expert relationship and more towards direct interaction between the expert and the expert system shell [SG86,DT87].

There is a need for expert system shells to be more open and flexible in order to allow a closer interaction between the expert and the system. The Advisor system allows a closer interaction between the expert and the shell by providing a quasi-English natural language interface, which is programmed by the expert. Thus the language used in rule construction and building functional descriptions consists almost entirely of terms which are familiar to the polymer expert and the designer.

Prolog is a suitable programming language for rule representation. This is due to the fact that Prolog is a language which consists nearly entirely of rules and facts which when asserted in a database form a program. Prolog has the added advantages that

1. It is based on the sound mathematical principles of resolution (as indicated by Bundy [AB87] in support of it as an expert system programming language) and thus any rule base constructed from Prolog facts and rules can be considered to be consistent and reliable[1].

2. Prolog has a built in inference mechanism which works well with it's own rule base structure. This inference mechanism is resolution theorem proving using backward chaining.

The need to present the rules to the designer and the expert in a quasi-English form and to use the same rules in their Prolog equivalent form gives rise to a need to translate the quasi-English rules into Prolog clauses. This is achieved with relative ease owing to the symbolic nature of Prolog. However having Prolog as the underlying representation for the rules means that the person programming the rule base must think of the rules in a

[1] The properties of Robustness, Predictability, Flexibility and Continuity as defined in [AB87] lead to this reliability.

backward chaining way which is contrary to the way in which rules are usually considered.

Arity Prolog is perhaps the best PC-based version of Prolog available on the market today[2]. Arity's usefulness stems from the fact that the developmental environment (i.e. the interpreter) is very well presented and easy to use[3]. Furthermore the Arity Product also incorporates a sophisticated compiler which can be used to produce executable applications.

---

[2] This is based on a comparison with Turbo Prolog, IBM Prolog (not PC-based) Prolog86 and SmallTalk Prolog. An exception to this would be Quintus Prolog (which is not PC-based) which is perhaps more powerful, while not possessing the graphical capabilities incorporated into the Arity language (i.e. Dialog Boxes).

[3] As apposed to the likes of Prolog86 which is very primitive and Smalltalk Prolog which is object oriented and thus difficult to understand.

6.2. The Future

Although the Advisor system is well suited to it's purpose some enhancements could be made to improve upon the final design.

The rule base language structure could be expanded upon to enable the programmer to include embedded Prolog code within the English like rule structure. This would allow the system to be more flexible by allowing the programmer of the rule base to access the underlying language. This is a feature which most expert system shells do not provide. A knowledge of Prolog would be necessary to use this enhancement feature. In the context of the Advisor program would involve creating two unique tokens, one to represent the start of the Prolog code and one to signify the end of the Prolog code. These tokens would be recognised as descriptors by the parser due to their position in a phrase in the command language. It also involves amending the parser to ensure that it skips over code contained within these symbols. The starting and terminating tokens would have to be symbols (atoms) which are not part of the Arity Prolog language nor likely to be used as a term in the expert system vocabulary. Symbols such as 'prolog{' and '}' could be used to represent the starting and ending tokens respectively. A sample rule might appear as

```
if specific_properties of dishes then
                flame_retardant = yes and
                prolog{ set_explain(X1,X2);
                        write('error in setting up explanation for')
                      } and
                tensile_strength is high.
```

During the rule translation it would be necessary to surround these Prolog goals with a left and right parentheses. Thus the rule above would be translated into the Prolog form

```
specific_properties(dishes,X0,Xn) :-
                eq(flame_retardant,yes,X0,X1),
                ( set_explain(X1,X2) ;
                 write('error in setting up explanation for')),
                tensile_strength(high).
```

These Prolog goals could be checked by writing them out to a temporary file and then using the consult/1 predicate to find errors if any in the Prolog code. This of course would mean making changes to the Advisor commit/2 predicate (see 'advcomit.ari' in appendix A) used to translate the rules into their Prolog form. This enhancement feature could be used by people experienced in using the system and in the use of Prolog, to improve the systems capabilities.

Another enhancement which would affect the natural language structure in Advisor is adapting the command or query language (as presented in the Application Descriptor dialog box) to allow the user to specify specific (technical) facts as well as general (design) facts. Also for reasons of flexibility the user of an expert system in Advisor should be allowed to fire a query on a loaded rule base based solely on the information provided by the user (without activating the questioning mechanism). Thus allowing an experienced user to see if the system could come up with a solution for a restricted set of facts.

Increasing importance is being placed on explanation features in expert systems, especially on an expert system's ability to explain it's own actions as apposed to just citing rules in response to a why question from the user. The incorporation of Prolog into the rule format in Advisor could be used explain it's findings more fully by activating some

customised explanation facility from within the rule itself, perhaps even calling some sort of dialog box from within the rule.

The system allows rules created by the expert to be compiled and linked to the shell program to create a special purpose expert system on design using polymers. To date this has not been done but can be done as soon as a stable rule set is written and tested. At some stage it is hoped that the rule base will become sufficiently complete to allow it to be actually compiled and linked to the Advisor object modules to form an expert system for Polymer selection. The main advantage of this would be one of speed. If new rules are found, following this incorporation of the rule base into the shell, these too could be compiled and linked to the existing expert system and thus the system could continue to grow in increments in it's executable form.

The need to guide and tutor in an expert system shell applies computer systems in general and has lead to much research being done into the relatively new area of user modelling [JT88], which while it applies to all human/computer interaction, equally applies to the interaction between an expert and expert system technology. It should be possible in the future to incorporate some form of user modelling into the Advisor system to enable the system to build a user profile of each user who logs onto the system. This profile should continually change based on the users ability or inability to use the system. This could have implications on such issues as context sensitive help (e.g. having stayup windows present on the screen to instruct the user in what to do next), explanations and presentation of information.

It would also be a good idea to further structure the vocabulary presented by the system (in the form of the descriptor, conjugate/operator and value word lists). This would involve separating design terms from technical terms within the vocabulary lists. The user profile mentioned above or some other method could be used to ensure that the designer

(as a novice user of the expert system) should be excluded from viewing the specific technical terms in the list boxes (used in the application descriptor and the question dialog box). This would enable the designer to concentrate on the design terms for the purpose of building an application description. On the other hand all the terms in the vocabulary should be available to an expert programming and testing a rule base as typically the expert would be knowledgeable about both the design and polymer domains.

Bibliography

Bibliography

[AAAI87] Rules for the implicit acquisition of knowledge about the user. AAAI 87 Vol 1 p295.

[AA89] DIFEAD A system to couple databases and Expert systems by A. Al-Zobaidie. Proceedings AI/CC - '88 UCD Vol 1 1988.

[ABCS87] The Nature and Evaluation of Expert System Tools for Engineering Applications by Allen, Boarnet, Culbert & Savely. Computers in engineering 1987 p143 - p155.

[AB87] How to improve the reliability of Expert systems by Allen Bundy. DAI University of Edinburgh Res Paper 336.

[AJR65] 'A machine-oriented logic based on the resolution principle'. Journal of the ACM, No. 12, p23-41.

[AJR79] Logic: Form and Function. Edinburgh: Edinburgh University Press.

[AP1] The Arity/Prolog Language Reference Manual. Arity Corporation 30 Domino Drive Concord, Masachusetts.

[AP2] Using the Arity/Prolog Interpreter and Compiler. Arity Corporation 30 Domino Drive Concord, Masachusetts.

[AS88] Human - Computer interface design by Alistair Sutcliffe. Macmillan Computer Science series 1988. ISBN 0-333-42898-6.

[ASU86] Compilers Principles, techniques, and tools by Alfred V.Aho, Ravi Sethi and Jeffrey D.Ullman ISBN 0-201-10194-7.

[BB81] Software Engineering Economics by Barry W.Boehm. Published by Prentice-Hall. ISBN 0-13-822122-7.

[BG89] Case Studies of expert systems development using microcomputer software packages by Tim Bodkin and Ian Graham. Expert Systems Feb. 1989 Vol 6 p12 - p15.

[BS86] Rule Based Expert Systems by Buchanan and Shortliffe. Addison-Wesley ISBN 0-201-10172-6.

[CM82] Selecting Thermoplastics for engineering applications by Charles P. MacDermott. ISBN 0-8247-7099-4.

Bibliography

[CM87] Programming in prolog by Clocksin & Mellish. Published by Springer-Verlag. ISBN 3-540-17539-3.

[DM84] Expert Systems in the Micro-electronic Age edited by Donald Michie for Edinburgh University Press ISBN 0 85224 493 2.

[DP871] The scope and limitations of First Generation Expert Systems by Derek Partridge. University of Exeter R150.

[DP871] Is Intuitive Expertise Rule Based? by Derek Partridge. University of Exeter R140.

[DT87] PC Expert Systems Prototyping and beyond by David W. Tong IEEE 1987 p184-p188.

[GK88] KNACK : Sample-driven Knowledge Acquisition for reporting systems by George Klinker. Carnegie and Mellen University CS-88-158.

[GL89] "On probabilistic Logic" by Jewin Guan & Victor R. Lesser presented at the second Irish Conference on Artificial Intelligence and Cognitive Science at DCU 1989.

[GW89] GOLDWORKS II User's Guide by Gold Hill Computers, Inc.

[HMM87] Expert Systems Tools and applications by Paul Harmon and Rex Maus and William Morrissey. Published by Wiley & Sons, Inc. ISBN 0-471-83951-5.

[IB86] Prolog programming for Artificial Intelligence by I. Bradko. Addison and Wesley. ISBN 0-201-14224-4.

[JA78] Teach Yourself Linguistics by Jean Aitchison. Published by David McKay & Co. Inc. 750 third avenue. NY 10017, U.S.A. ISBN 0-340-23106-8.

[JT88] A review and classification of User Modelling Systems and Methods by Jason Trenouth. Dept. of Computer Science, University of Exeter. R157

[JQ84] Fundamentals of the knowledge engineering problem by J.R. Quinlan, University of Sydney, Published in Introductory Readings in Expert Systems edited by Donald Mechie. ISBN 0-677-16350-9.

[KF87] Rules for the implicit acquisition of knowledge about the user by Robert Kass and Tim Finin. Department of Computer and Information Science, University of Pensylvania, Philadelphia, PA 19104-6389.

[KV88] Designing an expert shell for use in education by Karen Valley. DAI University of Edinburgh.

Bibliography

[KW86] Expert Systems Techniques, Tools and applications by Klahr and Waterman. Addison-Wesley ISBN 0-201-14186-8.

[LL88] Logic Approaches for Deductive Databases by Yim Yung Leung and Dik Lun Lee of Ohio State University. IEEE Expert Winter 1988 p64 - p74.

[LT861] Computer-Aided selection of polymeric materials and manufacturing Processes by Mark L. Loverich and Charles L. Tucker III. Dept. of mechanical and industrial engineering, University of Illinois.

[LT862] Computer program picks both resins and processes by Mark L. Loverich and Charles L. Tucker III. Plastics Engineering Oct. 86 p43 - p48.

[MB87] Specifying Meta Level Architectures for rule based systems by M. Beetz. University of Kaiserslautern sek1-sr-87-06.

[MJ83] System Development by Michael Jackson. Prentice-Hall international series in computer science. ISBN 0-13-880328-5.

[MS87] The design of the Postgres Storage system by M. StoneBaker. Memorandum No. UCB/ERL M87/6.

[MW86] Expert systems in polymer selection by Muserref Wiggins. ANTEC 86 p1393 - p1395.

[MWL88] Expert System Shells: Do they deliver what they promise? by Miller, Wilson & Lewis Air Products and chemicals, Inc., Chemical Engineering Progress Oct. 1988 p37 - p44.

[PH86] Logic Based Tools for building Expert and Knowledge based systems successes and failures of the technology by P. Hammond. Imperial College London 1986.

[PJ86] Introduction to Expert systems by Peter Jackson. Published by Addison-Wesley ISBN 0-20114223-6. p126-141.

[PP86] An Expert System for injection Moulding by Charles P. Paulson and Donald C. Paulson. ANTEC 86 p1396 - p1400.

[RD78] Hints for test selection: Help for the practising programmer by Richard A. DeMillo. IEEE catalog No. EHO 180-0.

[RH87] Building Expert systems using logic and meta-level interpretation by Han Reichgelt and Frank Van Harmelen. DAI University of Edinburgh Res Paper 303.

[RH90] Develop Advanced Expert systems (using Goldworks II) Byte january 1990 pages 219 - 224.

Bibliography

[RS87] The Postgres Data Model by L.A. Rowe and M.R.Stonebaker. Memorandum No. ECB/ERL M87/13.

[RV89] PC-Based expert system shells: some desirable and less desirable characteristics by Richard G. Vedder. Expert Systems Feb 1989, Vol 6 p28 - p41.

[RW88] Crystal By Ruth Wallsgrove. Personal Computer World November 1988 p172 - p175.

[SG86] Advances in interactive knowledge engineering by M. Shaw & B. Gaines Dept. Computer Science, University of Calgary, Alberta, Canada T2N 1N4.

[SHH86] The Design of the Postgres Rules System by M. Stonebraker, E. Hanson and C Hong. EECS Department, University of California Berkeley. Memorandum No. UCB/ERL M68/80.

[SL87] The Postgres Papers by M. Stonebaker and L.A. Rowe. Memorandum No. UCB/ERL M86/85.

[WD83] Systems analysis and design. A structured approach by William S. Davis. Addison and Wesley. ISBN 0-201-10271-4.

[WS89] An Explanation Facility for Today's Expert systems by Michael R. Wick and James R. Slagle. IEEE Spring 1989 p26 -p39.

Appendix A

Program Listings

```
% this predicate was used to load all the advisor prgoram modules into
% the interpreter program database during early stages of development.
%

tstload :-
[-adv],
[-advnewnx],
[-advloadr],
[-advappmu],
[-advrulmu],
[-advfopen],
[-advstpad],
[-append],
[-listdel],
[-member],
[-advhlsrn],
[-advfill],
[-advsplit],
[-advrprsr],
[-advsavrl],
[-listsub],
[-advdsprl],
[-advtable],
[-advrulgo],
[-advcomit],
[-advexpln].

% testrun just calls the main predicate.

tstrun :- main.
```

```
:- public main/0.
:- visible ton/0,toff/0,lmr/0,lrf/0,lcr/0,app_run/0,rulerun/0,mquit/0,mhelp/0,sysupdate/0.
:- visible mgo/0.
:- visible lex/0.
:- visible shl/0.
:- extrn load/0.
:- extrn get_files/4,put_files/4. % in advfopen.ari
:- extrn go/0:far.                % in advrulgo.ari
:- extrn abandon/0:interp.        % set when opening a file is abandoned.
:- extrn traceon/0:interp.        % set when the tracer is turned on.
:- extrn app_run/0:far.  % in advappmu.ari
:- extrn rulerun/0:far.  % in advrulmu.ari

% The following are the corresponding predicates for the Main Menu items.

% The two optins on the popdown help menu
mhelp :-
        load_key('advhelp.hlp',hkey),
        dialog_run(help_box),
        eraseall(hkey),
        expunge.

sysupdate :-
        load_key('advupdte.hlp',hkey),
        dialog_run(help_box),
        eraseall(hkey),
        expunge.

% the go option which allows the user to fire on the rule base without
% providing a description with the application descriptor
mgo :- asserta(start([])),
       asserta(applic_name($start$)),
       (go;true).

% The shell predicate allows the user to enter a dos shell.
shl :-
        delete_window(menu_win1),
        shell,
        define_window(menu_win1,'MAIN WINDOW',(0,0),(22,79),(120,113)),
        current_window(_,menu_win1).

% The quit option just consists of a cut which is always true.
mquit :- !.

% The following two predictes effectively turn the rule tracer on and off.
% by asserting the fact that it is on or retracting it.
ton :-
        ifthen((not(traceon)),
                assert(traceon)
            ).

toff :- abolish(traceon/0).
```

```
ad and compile the named meta rule file. The actual load predicate is
ined in advloadr.ari

 assert(ruletype(mrl)),
  recordz(fkey,$Enter the name for the META rule file to be loaded$,Ref1),
  (load;true),              % fails if file is abandoned
  retract(ruletype(mrl)),
  erase(Ref1),expunge.

d and compile the named ordinary rule file. The actual load predicate is
ned in advloadr.ari

ssert(ruletype(crf)),
  recordz(fkey,$Enter the name for the RULE file to be loaded$,Ref1),
  (load;true),              % fails if file is abandoned
  retract(ruletype(crf)),
  erase(Ref1),expunge.

  compiled meta rule file and compiled ordinary rule file.
nals : sfl, filename(Name), abandon ----- advsavrl.ari



  [!
  recordz(fkey,$Enter the name for the COMPILED META rule file$,Ref1),
  dialog_run(filename,sfl),
  ifthenelse((not(abandon)),        %if
            (                       %then
            retract(filename(Name)),
            ifthen( (not(Name == $$)),      %if
            (                       %then
                  concat(Name,$.cmr$,Fname),
                  fileerrors(_,off),
                  (reconsult(Fname);missing_file(Name)),
                  fileerrors(_,on)
                 )
              )                     % end if
            ),
           retract(abandon)     %else
           ),                   % end if
 rase(Ref1),expunge
 |
 ail. % fail it so that it will do the second clause in the predicate
```

```
% The Main predicate.
% The first main clause sets up the database by reading the dictionary
% of terms files and consulting the menu and dialog box definitions
% into the program database.
% The second main clause actually begins execution of the program by
% Activating the main program window and main menu.

main :-
        [!
        not(recorded(startup,true,_)),
        create_popup('Setting Up',(4 , 6),(19 , 71),(120 , 113)),
        tmove(8,8),
        write($Setting up the system.....$),
        tmove(10,8),
        write($Please wait a moment.....$),
        consult('advdefns.ari'),
        get_files('app.adv','dsc.adv','cop.adv','val.adv'),
        reconsult('advindex.ari'),
        recorda(startup,true,_),
        exit_popup
        !],fail.

main :-
% display authors name
        recordz(cpyr,$          Advisor          $,_),
        recordz(cpyr,$                           $,_),
        recordz(cpyr,$          Version II       $,_),
        recordz(cpyr,$                           $,_),
        recordz(cpyr,$   Written by Paul Powell $,_),
        recordz(cpyr,$                           $,_),
        recordz(cpyr,$   Press ESC to continue. $,_),
        dialog_run(author),
        eraseall(cpyr),expunge,

% create main window.
        define_window(menu_win1,'MAIN WINDOW',(0,0),(22,79),(120,113)),
        current_window(_,menu_win1),!,

% display message
        recordz(msg,$If you are using the system for the first time$,_),
        recordz(msg,$Then please choose the System Update sub function $,_),
        recordz(msg,$contained under the Help function on the next menu . $,_),
        dialog_run(message),
        eraseall(msg),
        expunge,

% keep activating main menu until the quit item is chosen.
        repeat,
        [! mainmenu(Rv) !],
        Rv == mquit,!,
```

```
% clean up by saving dictionary files and deleting main window.

        create_popup('Cleaning Up',(4 , 6),(19 , 71),(120 , 113)),
        tmove(8,8),
        write($Updating the Knowledge Base.....$),
        tmove(10,8),
        write($Please wait a moment.....$),
        put_files('app.adv','dsc.adv','cop.adv','val.adv'),!, %  in advfopen.ari
        eraseall(app),      % put into clear database after a quit
        eraseall(dsc),      % as it was causing duplicates if I ran
        eraseall(cop),      % it two times in row .
        eraseall(val),
        eraseall(startup),
        expunge,
        exit_popup,
        delete_window(menu_win1).

% activate chosen item Rv from the menu

mainmenu(Rv):-
        [! send_menu_msg(activate(mainm,(0,0)),Rv) !],
        Rv.
```

```
:- segment(far1).
:- public insert_index/0:far.
:- visible rep_index/2,getname/2.
:- visible insert_index/0.
:- extrn append/3.
:- extrn get_files/4:far.          % in advfopen.ari
:- extrn mark_deleted/2,data_add/2.     % in advappmu.ari
:- extrn first/1:interp.    % used to make sure F3 is pressed in dialog box
:- extrn index/2:interp.  % The list of names of technical parameters.
:- extrn table/1:interp.   % The table of range value slots associated with
                           % each of the index names.



% Insert_index is used to alter the parameter list which is Advisors link
% to the associated database. It does this by reconsulting the table list ( the
% index/2 predicates having been previously loaded ) and then calling the
% index_update dialog box.

insert_index :-
        [-advtabls],
        assert(first($dummy$)),
        dialog_run(index_update,rep_index).


%-----------------------------------------------------------------------
% The index_update box
%
% f3 fills in initial values for index_update box .

rep_index(Msg,index_update) :-
        first(Str),
        ( Msg = char(9,15) ;
          Msg = char(0,15) ;
          Msg = char(13,28) ;
          Msg = char(255,1) ),
          put(7).

% Pressing F3 fills up the list box with the parameter list names.

rep_index(char(0,61),index_update) :-
                first(Str),
                retract(first(Str)),
                repeat, % repeat and fail loop to fill up list box
                assertz(index(end,_)),
                index(A,_),
                [! atom_string(A,String),
                ifthen((not(A == end)),
                        send_control_msg(lb_add_string(String),2,index_update)
                    )
                !],
                A == end,
                send_control_msg(update,2,index_update),
                retract(index(end,_)).
```

```
% When the insert push button is pressed :
% 1. The system gets the current list item from the list box.
% 2. The user is asked for the name of the new string.
% 3. The new name is added to discriptor list of words.
% 4. The index data structure is updated with this new index name and slot.

rep_index(command(_,ins),index_update) :-
        send_control_msg(lb_set_index(Old,Old),2,index_update),
        send_control_msg(lb_get_text(Old,After),2,index_update),
        get_name(Str),
        data_add(Str,dsc),
        ifthen((not(Str == err)),
        (
        create_popup('Inserting ',(4 , 6),(19 , 71),(120 , 113)),
        tmove(8,8),
        write($Updating index numbers (this may take a while )$),
        tmove(10,8),
        write($Please wait a moment.....$),
        addindex(After,Str),
        exit_popup,
%       send_control_msg(lb_clear,2,index_update),
        send_control_msg(lb_insert_string(Str,Old),2,index_update),
        send_control_msg(update,2,index_update)
        )).

% A item is deleted from the the index list and from the decriptor list of
% valid words.

rep_index(command(_,dindx),index_update) :-
        send_control_msg(lb_set_index(C,C),2,index_update),
        send_control_msg(lb_delete_string(C,D),2,index_update),
        delindex(D),  % delete from index and table
        mark_deleted(D,dsc), % delete it from the database
        send_control_msg(update,2,index_update).
```

```
% The change function creates a strange effect in the list box . It does
% change the desired item but seems to scroll up the box as well as deleting
% the first letter from the list box label .
% The process of changing the name of an item involves deleting the old name
% and adding the new name to the list.

rep_index(command(_,cindx),index_update) :-
        send_control_msg(lb_set_index(C,C),2,index_update),
        send_control_msg(lb_get_text(C,D),2,index_update),
        get_name(Str),
        ifthen((not(Str == err)),
                (
                 changeindex(D,Str),
                 mark_deleted(D,dsc),
                 data_add(Str,dsc),
                 send_control_msg(lb_delete_string(C,D),2,index_update),
                 Cn is C - 1,
                 send_control_msg(lb_set_index(C,Cn),2,index_update),
                 send_control_msg(lb_insert_string(Str,Cn),2,index_update),
                 send_control_msg(lb_set_index(Cn,C),2,index_update),
                 send_control_msg(update,2,index_update)
                )
               ).

% If the list box is canceled then the two files that effectively make up
% The parameter list (advindex and advtabls) are reconsulted into the
% program database replacing the old clauses which where assumed unsafe.
% The dictionary of valid terms for each of the categories of words is also
% reloaded and the dialog box is exited. The clause is failed so that the
% default dialog function will clean up efficiently and correctly

rep_index(command(_,cancel),index_update) :-
        [!
        create_popup('Correcting ',(4 , 6),(19 , 71),(120 , 113)),
        tmove(8,8),
        write($Restoring Safe state (undoing all changes)...$),
        tmove(10,8),
        write($Please wait a moment.....$),
        [-advindex],
        [-advtabls],
        eraseall(app),
        eraseall(dsc),
        eraseall(cop),
        eraseall(val),
        get_files('app.adv','dsc.adv','cop.adv','val.adv'),
        exit_popup,
        send_control_msg(lb_clear,2,index_update),
        send_control_msg(update,2,index_update)
        !],fail .
```

```
% If the changes are concidered by the user to be correct then the index
% is saved to disk and the dialog box is exited. The clause is failed so that the
% default dialog function will clean up efficiently and correctly.

rep_index(command(_,ok),index_update) :-
        [!
        file_list('advindex.ari',index/2),
        file_list('advtabls.ari',table/1),
        send_control_msg(lb_clear,2,index_update),
        send_control_msg(update,2,index_update)
        !],fail .

% The default dialog box function.

rep_index(Msg,index_update) :-
        def_dialog_fn(Msg,index_update).

% get name and it's associated dialog box function are used to get the
% name of an item to be added or an item which a selected item is to be
% changed to.

get_name(Str) :-
        dialog_run(index_name,getname),
        ((retract(name(Str)),Str \= $$);Str = err).

getname(command(_,ok),index_name) :-
           [! send_control_msg(ef_set_text(Old,$$),2,index_name),
           send_control_msg(update,2,index_name),
           assert(name(Old))
           !],fail.

getname(Msg,index_name) :-
        def_dialog_fn(Msg,index_name).


% The addindex predicate is used to change the structure of the index
% by adding the new name after the current item in the index list of clauses
% reordering the index list of clauses to allow for the newly added item. A
% slot in the associated parameter table list is also created.

addindex(After,Name) :-
        atom_string(Name1,After),
        nl,write('Inserting after.....'),write(After),nl,
        insert_after(Name1,Name,Newindex), % insert new item after selected index
        write('Reordering.....'),nl,
        reorderup(Newindex),                % reorder after new index item
        write('Updateing table.....'),nl,
        update_table(a).
```

```
% delindex deletes the named from the index list of clauses and removes a
% slot from the parameter table list.

delindex(Name) :-
        delete_item(Name), % insert new item after selected index
        update_table(d).

% changeindex just finds the index clause which has the Name1 as it's
% first argument (Name1 being the old name) and replaces this with a clause
% which has the new name (Name2) as it's second argument. No reordering of
% the list is necessary.

changeindex(String1,String2) :-
        atom_string(Name1,String1),
        atom_string(Name2,String2),
        key(index(_,_),Key),
        nref(Key,Ref),
        find_db_pos(Name1,Ref,NameRef),
        index(Name1,N),
        replace(NameRef,index(Name2,N)),
        expunge.

% insert after finds the place in the program database where the clause
% with the specified name as it's first argument is and records a clause
% with the name to be inserted after this position.

insert_after(Name1,Str,Newindex) :-
        key(index(_,_),Key),
        nref(Key,Ref),
        find_db_pos(Name1,Ref,NameRef),
        index(Name1,N),
        Newnum is N + 1,
        int_text(Newnum,SN),
        concat([$index($,Str,$,$,SN,$)$],NewStr),
        string_term(NewStr,Newindex),
        record_after(NameRef,Newindex,X).


% The find_db_pos predicate searches through the index/2 clauses (treating
% them as terms to find the reference number (or pointer) to the clause
% with the specified name Name1.

find_db_pos(Name1,Ref,NameRef) :-
        instance(Ref,Term),
        Term =.. [index,Name1,Indexnum],
        NameRef = Ref.

find_db_pos(Name1,Ref,NameRef) :-
        nref(Ref,Nref),
        find_db_pos(Name1,Nref,NameRef).
```

```
% reorderup is used to reorder the index clauses' second parameter
% which is their number in the parameter ( necessary in other parts of
% the system ) list when a new index item is added.

reorderup(index(Name,N)) :-
        assertz(index(end,nonum)),
        repeat,
        index(NextName,No),
        [! ifthen(((not(NextName == end)),(not(NextName == Name)),No >= N),
                        (
                         retract(index(NextName,No)),
                         NewNum is No + 1,
                         assertz(index(NextName,NewNum))
                        )
          ),
       expunge
        !],
        NextName == end,
        retract(index(end,nonum)),
        expunge.


% reorderdown is used to reorder the index clauses' second parameter
% which is their number in the parameter ( necessary in other parts of
% the system ) list when an index item is deleted.


reorderdown(index(Name,N)) :-
        assertz(index(end,nonum)),
        repeat,
        index(NextName,No),
        [! ifthen(((not(NextName == end)),(not(NextName == Name)),No >= N),
                        (
                         retract(index(NextName,No)),
                         NewNum is No - 1,
                         assertz(index(NextName,NewNum))
                        )
          ),
       expunge
        !],
        NextName == end,
        retract(index(end,nonum)),
        expunge.

% delete_iem deletes a index clause from the program database and
% reorders the index number accordingly.

delete_item(String) :-
        atom_string(A,String),
        index(A,N),
        reorderdown(index(A,N)),
        retract(index(A,N)),
        expunge.
```

```
% The update_table/1 predicate is used to update the list which represents
% range slots for each of the index names. Used with parameter 'a' a slot
% is added to the end of the list. Used with parameter 'd' a slot is removed
% the list.

update_table(a) :-
        retract(table(L)),
        append(L,[[X99,Y99]],Newl),
        assert(table(Newl)),expunge.

update_table(d) :-
        retract(table(L)),
        del_slot(L,Newl),
        assert(table(Newl)),expunge.

del_slot([H|T],T).

% The list_is predicate is used to count how many slots are currently
% in the list.

list_is :-
        table(L),
        list_count(L,0).

list_count([],Y) :-
        write('Count is : '),write(Y),nl.

list_count([H|T],N) :-
        N1 is N + 1,
        list_count(T,N1).
```

```
:- public load/0.
:- public getlist/0:far.
:- extrn ruletype/1:interp. % used by read_rules and save_rules says what type of
                            % rule is being saved passed from lmr and lrf in adv.ari
:- extrn ferr/0:interp.     % flag set in pass1 and record_error_clause in advrprsr
:- extrn deflist/1:interp. % Used by commit
:- extrn commit/2:far.      % in advcomit.ari
:- extrn append/3:far.      % in append.ari
:- extrn read_rules/1:far. % in advsavrl.ari
:- extrn pass1/1:far,pass2/1:far,pass3/2:far. % in advdsprl.ari


% The rule loader is used to load and compile. It first of all reads the
% contents of the file and makes a list of rules. It then parses the rules
% and if no errors are found then the rules are commited to the database.


load :-
        eraseall(vkey),
        expunge,
        read_rules(vkey),
        retract(filename(Name)), % if read file abandoned then fail.
                                 % But if the file does not exist it is
                                 % still compiled and an empty file is produced
        assert(temp([])),
        gc,!,                    % temp is used by getlist to store the
        getlist,          % the list of individual rules
        retract(temp(List)),
% the bulk of these predicates are contained in advdsprl & advcomit
        assert(newrules([])),    % newrules is used by pass2 of the parser
        gc,!,
        tmove(8,8),
        nl,write('Loading/Compiling rules.....'),
        tmove(10,8),
        nl,write('Pass1.....'),!,
        pass1(List),!,
        tmove(10,8),
        nl,write('Pass2.....'),!,
        pass2(List),             % pass2 updates the newrules list
        retract(newrules(L)),!,
        tmove(10,8),
        nl,write('Pass3.....'),!,
        pass3(List,L),
        tmove(10,8),
        nl,write('                              '),!,
```

```
% Rules will only be commited if all of them are valid
% Otherwise ferr will be set in pass3 and the errors reported .

ifthenelse(ferr,
                (retract(ferr),
                 tmove(8,8),
                nl,write('                                    '),
                 recorda(rerr,$Use rule editor to correct errors....$,_),
                 recorda(rerr,$No rules have been added from this file..$,_),
                 recorda(rerr,$******* ATTENTION *********$,_),
                 dialog_run(error_box),
                 eraseall(rerr),
                 expunge
                ),
% else
        (
        tmove(8,8),
        nl,write('Adding to rule base.....'),
        ifthenelse(ruletype(mrl),Str = $.cmr$,Str = $.crl$),
        concat(Name,Str,Fname),
        create(F,Fname),
        assert(deflist([])), % used to keep track of the default clauses
                             % to avoid duplicates in the commit predicate
        commit(F,List),      % commit actually compiles the rules into there
                             % Prolog equivalent and writes the new rules out
                             % to the newly created file.
        retract(deflist(_)),
        tmove(8,8),
        nl,write('                                    '),
        close(F),
        reconsult(Fname)
        )
  ),                                 % end if ferr
        abolish(currentgoal/1),
        abolish(currentrules/1).


% Get list is used to the contents stored under the database key 'vkey' and
% construct a list of rules. which can be processed more easily by the rules
% parser.

getlist :-
        key(vkey,K),
        nref(K,R),
        go(R).
getlist.

go(R) :-
        instance(R,T),
        getline($$,T,Rule,R,Nr),
        retract(temp(L)),
        append(L,[Rule],Newl),
```

```
        assert(temp(Newl)),!,
        nref(Nr,Nref),
        go(Nref).

getline(PT,T,Rule,R,Nr) :-   % end of rule found
        string_search($.$,T,_),
        concat(PT,T,Rule),
        Nr = R.

getline(PT,T,Rule,R,Fr) :-   % not end of rule so continue search of db
        concat(PT,T,Newpt),
        nref(R,Nr),
        instance(Nr,Nt),
        getline(Newpt,Nt,Rule,Nr,Fr).
```

```
% This was the first Prolog program written for advisor and was used as a
% prototype for the eventual system.

:- segment(far1).
:- public add_all/2.
:- public app_run/0:far.
:- public db_search/4:far,find/5,data_add/2:far,mark_deleted/2:far.
:- visible support/2,adding/2,del_select/2. % DIALOG BOX MANAGEMENT PREDICATES
:- visible ass/0,delapp/0,deldsc/0,delcop/0. % MENU FUNCTIONS
:- visible apdsc/0,prdsc/0,cpdsc/0,vldsc/0.
:- visible quit/0,help/0,add/0,dele/0.
:- extrn split/1.          % in advsplit.ari
:- extrn member/2.         % in member.ari
:- extrn append/3.         % in append.ari
:- extrn showlist/0.       % in advhlsrn.ari
:- extrn dialog_fill/2,fill/3.  % in advfill.ari
:- extrn sentence/1:interp.     % used to store command line for parsing
:- extrn first/1:interp.    % Used to disable all keys until F3 is pressed
:- extrn del/1:interp.      % Used to say which category of words are to be deleted from
:- extrn parsed/1:interp.% Used to record all the tokens for the query parser
:- extrn add/1:interp.      % Used to record the list of items which have been added
:- extrn replace/1:interp.  % Used to store the last command entered using the
                            % application descriptor so that it can be redisplayed
                            % If they enter the application descriptor again
:- extrn type/1:interp.     % Used to record the word category that the add
                            % Predicate is going to be working with.
:- extrn index/2:interp.  % The index list which is checked to disallow
                            % the deletion of index items.
:- extrn go/0:far.          % In advrulgo.ari


% The following are the menu commands behind the word lists which
% appear when the delete option is chosen from the Rule Builder Edit option

delapp :-
        assert(del(app)),
       assert(first('Press F3 to continue.....')),
        dialog_run(delete,del_select),!.

deldsc :-
        assert(del(dsc)),
       assert(first('Press F3 to continue.....')),
        dialog_run(delete,del_select),!.


delcop :-
        assert(del(cop)),
       assert(first('Press F3 to continue.....')),
        dialog_run(delete,del_select),!.
```

```
delval :-
        assert(del(val)),
       assert(first('Press F3 to continue.....')),
        dialog_run(delete,del_select),!.


% The following are the menu commands behind the word lists which
% appear when the add option is chosen from the Rule Builder Edit option


apdsc :-
        assert(add([])),
        assert(type(app)),
        dialog_run(app_desc_add,adding),
        retract(add(L)),expunge.

prdsc :-
        assert(add([])),
        assert(type(dsc)),
        dialog_run(app_desc_add,adding),
        retract(add(L)),expunge.

cpdsc :-
        assert(add([])),
        assert(type(cop)),
        dialog_run(app_desc_add,adding),
        retract(add(L)),expunge.

vldsc :-
        assert(add([])),
        assert(type(val)),
        dialog_run(app_desc_add,adding),
        retract(add(L)),expunge.

% The quit menu option
quit :-  !.

% The help menu option gives help associated with the application descriptor

help :-
        load_key('advahlp.hlp',hkey),
        dialog_run(help_box),
        eraseall(hkey).

% This is the predicate behind the application descriptor option. It invokes
% the dapplic dialog box which enables the user to construct a description.

ass :-
      assert(first('Press F3 to continue.....')),
      (dialog_run(dapplic,support);
       (nl,write(' support failed '),get0(_))) ,
      abolish(first/1).
```

```
% This predicate sets up the Appliaction Window and calls the predicate appl(_) to
% set up and activate the menu and loads in the help file for this window. This
% predicate is called from the main menu activated in adv.ari

app_run:-
        define_window(menu_win2,'APPLICATION WINDOW',(0,0),(22,79),(120,113)),
        current_window(menu_win1,menu_win2),
        load_key('advahbox.hlp',hkey),
        appl(_),
        eraseall(hkey),
        expunge.

% appl activates the menu and keeps calling itself recursively until ESC is
% pressed or Quit is chosen from the menu bar in which case it returns to the
% main window.

appl(_):-
        menu_app(Rv),
        ifthen( (Rv = ass,sentence(N)) ,parse(_) ),
        ifthenelse((Rv = quit; Rv = cancel),
                   (
                    current_window(menu_win2,menu_win1),
                    delete_window(menu_win2)
                   )
                   ,(appl(_))).

menu_app(Rv):-
        [! send_menu_msg(activate(mapplic,(0,0)),Rv) !],
        Rv.

menu_app(Rv).

% activates the add menu
add :-
        [!send_menu_msg(activate(madd,(0,0)),Rv1) !] ,
        Rv1.
add.

% activates the delete menu
dele  :-
        [! send_menu_msg(activate(mdel,(0,0)),Rv2) !],
        Rv2.
dele.
```

%----------

```
% This is the user defined dialog box management predicate for the application
% descriptor dialog box.
% f3 updates the list boxes with new values previously added
% and also fills in the initial values .
%

support(char(0,61),dapplic):-
        first(_),
        retract(first(_)),
        ifthenelse(retract(replace(Com)),          % if there was an error
         (send_control_msg(ef_set_text(Oldtext,Com),5,dapplic),          % then
         send_control_msg(update,5,dapplic)
         ),
         (send_control_msg(ef_set_text(Oldtext,$$),5,dapplic), % else
         send_control_msg(update,5,dapplic)
         )
        ), % end if
        send_control_msg(lb_clear,1,dapplic),
        send_control_msg(update,1,dapplic),
        send_control_msg(lb_clear,2,dapplic),
        send_control_msg(update,2,dapplic),
        send_control_msg(lb_clear,3,dapplic),
        send_control_msg(update,3,dapplic),

        fill(dsc,dapplic,1) ,
        fill(cop,dapplic,2) ,
        fill(val,dapplic,3) .



% f1 clears the command line
support(char(0,59),dapplic) :-
         send_control_msg(ef_set_text(Oldtext,$$),5,dapplic),
        send_control_msg(update,5,dapplic),!,
         abolish(sentence/1).

% function key f2 selects items from the list box in focus and places them on the
% command line.

support(char(0,60),dapplic) :-
        !,
        which_control(N),
        send_control_msg(lb_set_index(l,l),N,dapplic),     % gets the index of the current choice
        send_control_msg(lb_get_text(l,Str),N,dapplic),
        send_control_msg(ef_set_text(Oldtext,Oldtext),5,dapplic),
        concat([Oldtext,Str,$ $],Newtext),
        send_control_msg(ef_set_text(Oldtext,Newtext),5,dapplic),
        send_control_msg(update,5,dapplic).



% on exiting the dialog box set an end of sentence marker
```

```
support(command(_,ok),dapplic) :-
        send_control_msg(ef_set_text(Oldtext,Oldtext),5,dapplic),
        concat([Oldtext,$ $,$es$],Text),
        split(Text),!,
        assert(rem(Oldtext)),
        exit_dbox(dapplic).

support(command(_,cancel),dapplic) :-
        ifthen(sentence(_),abolish(sentence/1) ),!,
        exit_dbox(dapplic).

support(command(_,help),dapplic) :-
        helprun.


support(Msg,dapplic) :-
        def_dialog_fn(Msg,dapplic).


helprun :-
        dialog_run(help_box).

%-----------------------------------------------------------------------
% When a name is added using the edit function add the string that is added
%  is also recorded under the appropriate key.
% The add predicate is made to fail so that the dialog box can be exited
%  properly by the default function which calls the appropriate control
% predicates.
% Values are added to a list as the user is entering them in the edit field
% they are not added to the database until the Ok push button is activated.
%-----------------------------------------------------------------------

adding(command(_,add),app_desc_add) :-
        [! send_control_msg(ef_set_text(Old,$$),2,app_desc_add),
           send_control_msg(update,2,app_desc_add),
        retract(add(L)),
        append([Old],L,Nl),
        assert(add(Nl))
         !],fail.

adding(command(_,ok),app_desc_add) :-
        [! add(L),
        retract(type(T)),
        add_all(L,T) !],fail.

adding(command(_,cancel),app_desc_add) :-
        [! retract(add(L)),
        retract(type(T)) !],fail.



adding(command(_,adp),app_desc_add) :-
        showlist.               % contained in advhlsrn.ari shows lists of
                                % currently defined words for the word lists
```

```
adding(Msg,app_desc_add) :-
        def_dialog_fn(Msg,app_desc_add).


% Adds all the items to the dictionary of valid terms for that particular
% category or Type of words.

add_all([],Type).
add_all([H|T],Type) :-
      data_add(H,Type),
        add_all(T,Type).

data_add(String,Dbkey):-        % checks for duplicates
        db_search(Dbkey,String,Flag,Ref),
        ifthen(Flag == notfound,
                (
                 instance(Ref,Old),
                 replace(Ref,String),
                 record_after(Ref,Old,_)
                )
              ),
        ifthen(Flag == first,recorda(Dbkey,String,_)),
        ifthen(Flag == last,recordz(Dbkey,String,_)).
```

```
%-----------------------------------------------------------------------
% This predicate will search the current database for a string Str stored
% under the database key Dbkey .
% It will fail if a match is not found .
%

db_search(Dbkey,Str,Flag,Ref1) :-
        [! key(Dbkey,Key) , nref(Key,Ref) !] ,
        instance(Ref,First),
        find(Str,First,Ref,Flag,Ref1).                    % found or not found


find(Str,First,Ref,Flag,NNref) :-
        instance(Ref,Term) ,
        (
         (
         [!
         (Str == Term,NNref = Ref,Flag = found);
         (Str @< First,NNref = Ref,Flag = first);
         (Str @< Term,NNref = Ref,Flag = notfound)
         !]
         );
           ([!nref(Ref,Nref)!],find(Str,_,Nref,Flag,NNref))
        ).

find(Str,Last,R,F,R) :-
        F = last.



%-----------------------------------------------------------------------
% The delete box
%
% f3 fills in initial values for delete box all other keys are disabled until
% that key is pressed.
% On pressing the Ok push button the system moves backwards through the list
% of words it is working with and deletes the ones which have been chosen from
% the choice list box in the delete dialog box.
% Descriptors which are also in the index list are not allowed to be deleted
% as they make up the interface to the associated database system.

del_select(Msg,delete) :-
        first(Str),
        ( Msg = char(9,15) ;
          Msg = char(0,15) ;
          Msg = char(13,28) ;
          Msg = char(255,1) ),
          send_control_msg(ef_set_text(Oldtext,Str),2,delete),
         send_control_msg(update,2,delete),
          put(7).
```

```
del_select(char(0,61),delete) :-
                first(Str),
                retract(first(Str)),
                del(Dbkey),
                send_control_msg(lb_clear,1,delete),
                send_control_msg(update,1,delete),
                send_control_msg(ef_set_text(Oldtext,$$),2,delete),
                send_control_msg(update,2,delete),
                dialog_fill(delete,Dbkey).

del_select(command(_,ok),delete) :-
        [! del(Dbkey),
        assert(deleted(Dbkey)),
        retract(del(Dbkey)),
        send_control_msg(lb_get_count(Count),1,delete),
        send_control_msg(ef_set_text(Oldtext1,$Deleting choices...$),2,delete),
       send_control_msg(update,2,delete),!,
        delete_choices(delete,Dbkey,Count),
        send_control_msg(ef_set_text(Oldtext2,$$),2,delete),
       send_control_msg(update,2,delete),
        send_control_msg(lb_clear,1,delete),
        send_control_msg(update,1,delete)
         !],
        fail.

del_select(command(_,cancel),delete) :-
        [! retract(del(Dbkey)) ,
        send_control_msg(ef_set_text(Oldtext,$$),2,delete),
       send_control_msg(update,2,delete),
        send_control_msg(lb_clear,1,delete),
        send_control_msg(update,1,delete)
        !],fail .

del_select(Msg,delete) :-
        def_dialog_fn(Msg,delete).

% delete choices goes through the list box of choices and deletes them from
% the dictionary of words if they are not index items.

delete_choices(Dbox,Dbkey,First) :-
        send_control_msg(lb_get_choices(First,Str),1,Dbox),
        atom_string(A,Str),
        (index(A,_);mark_deleted(Str,Dbkey)),!,
        Next is First - 1,!,
        delete_choices(Dbox,Dbkey,Next).

delete_choices(Dbox,Dbkey,0).

delete_choices(Dbox,Dbkey,First) :-
        Next is First - 1,!,
        delete_choices(Dbox,Dbkey,Next).
```

```
% The following predicates actually find the items in the program database
% and remove them.

mark_deleted(Str,Dbkey) :-
        key(Dbkey,Key),
        pref(Key,Ref),
        back_and_mark(Ref,Str,Dbkey).

mark_deleted(Str,Dbkey).

back_and_mark(Ref,Str,Dbkey) :-
        instance(Ref,String),
        String == Str,!,
        erase(Ref),
        expunge.


back_and_mark(Ref,Str,Dbkey) :-
        pref(Ref,Nref),
        back_and_mark(Nref,Str,Dbkey).
```

```
%-----------------------------------------------------------------------
% This section of the program is the query parser .
% It parses the sentence built up during the application description ,
%  that is the sentence that is on the command line when the End push button
%  is encountered .
% The end of sentence marker is taken as being a conjugate/operator
%  and is thus recorded under the key conj but only for the purposes of
%  parsing it deleted after parsing.
% The parser checks the syntax and semantics of a query in one go using a top
% down recursive ad-hoc compiler technique.
% If the parse is successful then the program builds up a prolog clause to
% represent the query and this is processed by the go predicate (in advrulgo.ari)
% which is used to fire the query on a loaded rule base.

parse(_) :-
        [! data_add($es$,cop) !] ,
          ifthenelse(lexical_syntax(dsc,1) ,
                        (dialog_run(success) ,
                         retract(rem(Com)),
%                        parsed(Name),
                         assert(applic_name($start$)), % starting fact for advrulgo
                         concat($start$,$([$,Head),  % starting fact
                         assert(applic(Head)),
                         build,
                          create(H,'advappl.rul'),
                         retract(applic(Strclause)),
                         write(H,Strclause),
                         nl(H),
                         close(H)
                        ) ,
                        true ),
          gc,
          reconsult('advappl.rul'),
          (go;true),
          ifthen(sentence(_),abolish(sentence/1) ),!,
          mark_deleted($es$,cop).

% application  check no longer used

lexical_syntax(app,N) :-
          [! sentence(Str),
          db_search(app,Str,Flag,_) !],
          Flag == found,
          assert(parsed(Str)),
          retract(sentence(Str)),!,
          lexical_syntax(dsc,N).

lexical_syntax(app,1) :-
          error_popup('Application name not found '),fail.

lexical_syntax(app,2) :-
          error_popup('Statement must contain valid application name'),fail.
```

```
% description check
lexical_syntax(dsc,N) :-
        [! sentence(Str),
        db_search(dsc,Str,Flag,_) !],
        Flag == found,
        assert(parsed(Str)),
        retract(sentence(Str)),!,
        lexical_syntax(cop,N).

lexical_syntax(dsc,1) :-
        error_popup('Description name not found '),fail.

lexical_syntax(dsc,2) :-
        error_popup('Statement must contain valid Description name').

% conjugate/operator check

lexical_syntax(cop,N) :-
        [! sentence(Str) !],
        Str = $es$,
        assert(parsed(Str)),
        retract(sentence(Str)).


lexical_syntax(cop,N) :-
        [! sentence(Str) ,
        db_search(cop,Str,Flag,_) !],
        Flag == found,
        assert(parsed(Str)),
        retract(sentence(Str)),!,
        ifthen(not(lexical_syntax(dsc,3)) , lexical_syntax(val,N)).

lexical_syntax(cop,1) :-
        error_popup('Conjugate/operator missing'),fail.

lexical_syntax(cop,2) :-
        error_popup('Conjugate/operator must be between two descriptors').

lexical_syntax(val,N) :-
        [! sentence(Str),
        db_search(val,Str,Flag,_)!],
        ((Flag == found);(int_text(I,Str))),
        assert(parsed(Str)),
        retract(sentence(Str)),!,
        lexical_syntax(cop,N).

lexical_syntax(val,1) :-
        error_popup('Value name not found'),fail.

lexical_syntax(val,2) :-
        error_popup('Value must be between conj/oper and ').
```

```
%-----------------------------------------------------------------------
% error box to display error messages for the parser.
%
%
error_popup(Derr_mess) :-
        retract(rem(Com)),
        ifthenelse(sentence(Str),
                concat([Derr_mess,$ $,Str],Full_err),
                Full_err is Derr_mess),
        recordz(ekey,Full_err,Ref),
        dialog_run(error),
        ifthen(parsed(_) , abolish(parsed/1) ),
        assert(replace(Com)),
        hard_erase(Ref).

%-----------------------------------------------------------------------
% This part of the program takes the parsed query and builds up a
% structure to represent the content of the information .
% This information will be used later to match up against the content stored
%  in the rules .
%
% It was necessary to use a repeat fail loop as the local stack could not
%  be relied upon to repeatidily call the build predicate recursively .


build :-
        repeat,
        retract(parsed(C)),
       [!
         case( [


% If this clause is called and there exists a current parameter list then
%  assert the current clause and start a new clause.
%
         ([!db_search(dsc,C,Flag,_)!],Flag == found)
        -> ( retract(applic(Str)),
             concat([Str,C,$($],Newstr),
             assert(applic(Newstr))
             ),  % end dsc      case

% A conjugate or an operator is appended to the end of the parameter list
% except for the special case of is which is fundemental and is contained
% in the meaning of the clause itself .
%
        ([!db_search(cop,C,Flag,_)!],Flag == found)
        -> (ifthen( (member(C,[$=$,$<$,$>$,$<=$,$>=$,$<>$]) ),
                        (retract(applic(Str)),
                         concat([Str,C,$,$],Newstr),
                         assert(applic(Newstr))
                )
                 ) ,
```

```
            ifthen(C == $and$ ,
                        ( retract(applic(Str)),
                          concat(Str,$,$,Nstr),
                          assert(applic(Nstr))
                        )
                   ),
            ifthen(C == $or$ ,
                        ( retract(applic(Str)),
                          concat(Str,$;$,Nstr),
                          assert(applic(Nstr))
                        )
                   ),

             ifthen(C == $es$ ,
                        ( retract(applic(Str)),
                          concat(Str,$]).$,Nstr),
                          assert(applic(Nstr))
                )
                   )
            ),      %end cop case

% A values is just appended to the list .
        ((([Idb_search(val,C,Flag,_)!],Flag == found));(int_text(I,C)))
        -> (retract(applic(Str)),
            concat([Str,C,$)$],Newstr),
            assert(applic(Newstr))
          )
           | (nl,write('Failing to Build '),write(C),nl,get0(_))])  % end case
          !]

        , C == $es$.
```

```
:- segment(far1).
:- public rulerun/0:far.
:- visible rhlp/0,rquit/0,cmr/0,crf/0.          % MENU CHOICES
:- extrn rcreat/0.          % in advdsprl.ari

% Cretaing meta rule files
cmr :-
        assert(ruletype(mrl)),
        (rcreat;true) ,             % fails if file abandoned
        abolish(ruletype/1).

% creating ordinary rule files.
crf :-
        assert(ruletype(crf)),
        (rcreat;true) ,
        abolish(ruletype/1).

% Get Help on this part of the system
rhlp :-
%       [-advhlpbx],
        load_key('advrhlp.hlp',hkey),
        dialog_run(help_box),
        eraseall(hkey).

% The quit option.
rquit :- !.

% The rulerun predicate is responable for setting up the Rule Window
% and activating the menu bar.

rulerun :-
        define_window(menu_win3,'RULE WINDOW',(0,0),(22,79),(120,113)),
        current_window(menu_win1,menu_win3),!,
        repeat,
        [! menu_rul(Rv) !],
        ((Rv == rquit);(Rv == cancel)),
        current_window(_,menu_win1),
        delete_window(menu_win3).

menu_rul(Rv):-
        [! send_menu_msg(activate(mrule,(0,0)),Rv) !],
        Rv.
```

```
:- public get_files/4:far,put_files/4.
:- extrn strip_blanks/2,padd/3.          % in advstpadd
:- extrn node_size/1:interp.             % The node size determines the maximum
                                         % size of the dictionary items to be written
                                         % to file.
% Opens and reads the contents of the named files for each of the word lists
% used in the system.

get_files(W,X,Y,Z):-                     % Arity can only handle two file pointers
        assert(node_size(20)),
        open(H1,W,r),!,                  % at the same time so do it all sequentially
        read_file(H1,app),               % when reading in
        close(H1),
        open(H2,X,r),!,
        read_file(H2,dsc),
        close(H2),
        open(H3,Y,r),!,
        read_file(H3,cop),
        close(H3),
        open(H4,Z,r),!,
        read_file(H4,val),
        close(H4).

% Saves the contents of each of the word lists to disk.

put_files(W,X,Y,Z):-
        open(H1,W,w),!,
        write_file(H1,app),
        close(H1),

        open(H2,X,w),!,
        write_file(H2,dsc),
        close(H2),

        open(H3,Y,w),!,
        write_file(H3,cop),
        close(H3),

        open(H4,Z,w),!,
        write_file(H4,val),
        close(H4).

read_files(H1,H2,H3,H4):-
        ifthen( (not(read_file(H1,app))), (error(1),close(H1)) ),!,
        ifthen( (not(read_file(H2,dsc))), (error(2),close(H2),fail) ),!,
        ifthen( (not(read_file(H3,cop))), (error(3),close(H3),fail) ),!,
        ifthen( (not(read_file(H4,val))), (error(3),close(H4),fail) ).


error(N) :-
        case([N = 1 -> (write('error reading file app.adv'),nl),
              N = 2 -> (write('error reading file dsc.adv'),nl),
              N = 3 -> (write('error reading file cop.adv'),nl),
              N = 4 -> (write('error reading file val.adv'),nl)]).
```

```
read_file(H,Dkey) :-
        seek(H,0,eof,Eof),    % if it's not empty then get the nodes
        not(Eof == 0),
        seek(H,0,bof,_),      % file pointer back to the begining of the file
        node_size(S),
        get_nodes(H,S,Eof,Dkey).

read_file(H,Dkey) :-          % if it is empty do nothing
        seek(H,0,eof,Empty),
        Empty == 0.


% get_nodes gets the words form the associated file and stores them
% under the appropriate key in the program database.

get_nodes(H,S,Eof,Dkey):-
        [! read_string(H,S,String1) !],
        not(String1 == $$),
        strip_blanks(String1,String2),
        recordz(Dkey,String2,_),
        get_nodes(H,S,Eof,Dkey).

get_nodes(H,S,Eof,Dkey).      % detect ond of file and stop

% write_file chains forward through the list of words as stored in the
% program database and padds them out to the node size if neccessary before
% writing them out to the specified file.

write_file(H,Dkey) :-
        key(Dkey,Key),
        nref(Key,Ref),
        write_and_go(Ref,H).
write_file(H,Dkey).

write_and_go(Ref,H):-
        instance(Ref,String1),
        padd(String1,String2,20),!,      % in advstpad.ari
        ifthen( not((string_search($+$,String2,Loc) ; string_search($-$,String2,Loc1)) ) ,
                write(H,String2) ),
        nref(Ref,Nref),
        write_and_go(Nref,H).
```

```
:- public padd/3,strip_blanks/2.

% This predicate will return a string padded with blanks to the specified
% length .

padd(String,String1,Padding_length):-
        string_length(String,Length),
        Len is (Padding_length - Length),
        padd1(String,String1,Len),
        string_length(String1,L).


padd1(S,S1,N):-
        N > 0,
        concat(S,$ $,S2),
        N1 is N - 1,
        padd1(S2,S1,N1).

% stopping condition
padd1(String,String1,0):-
        !,
        String1 = String.

% predicate to strip blanks once the node has been read in
strip_blanks(String1,String2) :-
        string_search($ $,String1,Locat),
        substring(String1,0,Locat,String2).
```

```
:- segment(far1).
:- public showlist/0.
:- visible hlst/2,update_cb/3.  % HELP LIST DIALOG PREDICATES
:- extrn fill/3.                % in advfill.ari
:- extrn grabbed/1:interp.      % used in an attempt to allow the user to
                                % return a selected list item to the rule
                                % editor.

showlist :-
        dialog_run(help_list,hlst).

% The following set the choice buttons appropriatily and fill the list box
% with the chosen word list.

hlst(char(0,30),help_list) :-       %ALT + A application word list
                [!
                    update_cb(6,7,8),
                    send_control_msg(lb_clear,3,help_list),
                    fill(app,help_list,3)!],fail.

hlst(char(0,32),help_list) :-       %ALT + D descriptor word list
                [!
                    update_cb(5,7,8),
                    send_control_msg(lb_clear,3,help_list),
                    fill(dsc,help_list,3) !],fail.

hlst(char(0,24),help_list) :-       %ALT + O conjugate/operator word lists
                [!
                    update_cb(5,6,8),
                    send_control_msg(lb_clear,3,help_list),
                    fill(cop,help_list,3) !],fail.

hlst(char(0,47),help_list) :-       %ALT + V value word list
                [!
                    update_cb(5,6,7),
                    send_control_msg(lb_clear,3,help_list),
                    fill(val,help_list,3) !],fail.
```

space bar is used to set a choice button do as above

```
(32,57),help_list) :-
!
vhich_control(N),
'N == 5;N==6;N==7;N==8),
ithen(N==5,(
                update_cb(6,7,8),
                send_control_msg(lb_clear,3,help_list),
                fill(app,help_list,3)
               )
        ),
then(N==6,(
                update_cb(5,7,8),
                send_control_msg(lb_clear,3,help_list),
                fill(dsc,help_list,3)
               )
        ),
then(N==7,(
                update_cb(5,6,8),
                send_control_msg(lb_clear,3,help_list),
                fill(cop,help_list,3)
               )
        ),
then(N==8,(
                update_cb(5,6,7),
                send_control_msg(lb_clear,3,help_list),
                fill(val,help_list,3)
               )
        ) !],fail.
nand(_,ok),help_list) :-

end_control_msg(lb_set_index(Old,Old),3,help_list),
end_control_msg(lb_get_text(Old,After),3,help_list),
sert(grabbed(After))
,fail.

nelp_list) :-
ef_dialog_fn(Msg,help_list).

oice buttons depending on which one has been chosen
o(C1,C2,C3) :-
        send_control_msg(cb_set_val(_,unchecked),C1,help_list),
        send_control_msg(update,C1,help_list),
        send_control_msg(cb_set_val(_,unchecked),C2,help_list),
        send_control_msg(update,C2,help_list),
        send_control_msg(cb_set_val(_,unchecked),C3,help_list),
        send_control_msg(update,C3,help_list).
```

```
:- segment(far1).
:- public split/1:far.
%-------------------------------------------------------------------------
% The split(Text) predicate is used to split the command line sentence up
% into individual words and assert them as parts of a sentence
%
%

split(Text) :-
        [! string_length(Text,Len) !],
        not(Len = 0 ),
        skip_blanks(Text,Text2,Newlen),
        string_search($ $,Text2,Loc),
        substring(Text2,0,Loc,Str),
        assert(sentence(Str)) ,
        End is Newlen - Loc,!,
        substring(Text2,Loc,End,Newtext),
        split(Newtext).

% default stoping condition . Can actually get along without it but just in
% case !!!!!!!!!! .

split(Text) :-
        string_length(Text,Len),
        Len = 0.


% stopping condition for adv programs  . Could just as easily have a stopping
% condition of '.' for real sentences .

split(Text) :-
        string_search($es$,Text,Loc),
        assert(sentence($es$)).

% finished
skip_blanks(Text,Newtext,Len) :-
        [! substring(Text,0,1,Ch) !],
        not( Ch = $ $),!,
        string_length(Text,Len),
        copy(Newtext,Text).

% skip over the blanks
skip_blanks(Text,Newtext,L) :-
        string_length(Text,Len),
        Newlen is Len - 1,
        substring(Text,1,Newlen,Text1),
        skip_blanks(Text1,Newtext1,Newlen),
        copy(Newtext,Newtext1),
        copy(L,Newlen).


copy(Text,Text).
```

```
:- segment(far1).
:- public rp_arse/0.
:- extrn db_search/4,data_add/2,mark_deleted/2. % in advappmu.ari
:- extrn split/1.                               % in advsplit.ari
:- extrn currentgoal/1:interp.     % set in pass3 of advdsprl.ari
:- extrn sentence/1:interp.      % used to hold tokens when sentence is split
:- extrn currentrule/1:interp.  % set in pass3 of advdspri.ari
:- extrn ferr/0:interp.          % error flag set if any error occurs while parsing
                                    % a rule.
:- extrn error/0:interp.         % another error flag.

%-----------------------------------------------------------------------
% This section of the program is the RULE PARSER .
% It parses the sentence built up during the RULE CREATION
% The end of sentence marker is taken as being a conjugate/operator
% and is thus recorded under the key conj but only for the purposes of
% parsing.
% The parser searches the word lists for each slot in the sentence. If an
% error is found the parser still continues but records the rule as being
% faulty.
%
rp_arse :-
        currentgoal(G),
        concat(G,$ es$,Sg),
        split(Sg),
       [! data_add($es$,cop) !] ,
        (rlex_syntax(dsc,1);true),
        abolish(sentence/1),
        mark_deleted($es$,cop),!,
        expunge.


% description check
rlex_syntax(dsc,N) :-
        [! sentence(Str),
        db_search(dsc,Str,Flag,_)!],
        Flag == found,
        retract(sentence(Str)),!,
        rlex_syntax(cop,N).

rlex_syntax(dsc,1) :-
        sentence(Str),
        record_error_clause('Invalid undefined description name : ',Str),
        retract(sentence(Str)),!,
        rlex_syntax(cop,N).


% conjugate/operator check

rlex_syntax(cop,N) :-
        [! sentence(Str) !],
        Str == $es$,
        retract(sentence(Str)),!.
```

```
:- segment(far1).
:- public save_rules/1.
:- public read_rules/1:far.
:- visible sfl/2.
:- extrn abandon/0:interp.    % set if the user presses ESC in the filename
                              % dialog box.
:- extrn ruletype/1:interp.  % Used to record which type of rule file is being
                               % saved or loaded.
:- extrn filename/1:interp.  % used to record the actual filename.


% Save rules recieves a list of the rules to be saved and saves them.
% Firstly it runs the dialog box filename which gets the filename from
% the user. It then sees which type of file it is dealing with to choose
% the appropriate extention to be added on to the file name. The rwrite_out
% predicate is then used to recursively write out the list of rules.

save_rules([]).
save_rules(L) :-
        dialog_run(filename,sfl),
        ifthenelse((not(abandon)),
                        (
                         retract(filename(Name)),
                         ifthen( (not(Name == $$)),
                                 (
                                  ifthen(ruletype(crf),
                                         concat(Name,$.rul$,Fname)
                                        ),
                                  ifthen(ruletype(mrl),
                                         concat(Name,$.mrl$,Fname)
                                        ),
                                  create(H,Fname),
                                  rwrite_out(L,H),
                                  close(H)
                                 )
                             )
                        ),
                        retract(abandon)
                    ).
% The read_rules predicate :
% 1. gets the filename.
% 2. checks the rule type.
% 3. loads the rules under the database key vkey if the file exists.

read_rules(Key) :-
        dialog_run(filename,sfl),
        ifthenelse((not(abandon)),        %if
        (
        filename(Name),           %then

        ifthenelse( (not(Name == $$)),  %if
        (
         ifthen(ruletype(crf),
```

```
                concat(Name,$.rul$,Fname)
            ),
            ifthen(ruletype(mrl),
                concat(Name,$.mrl$,Fname)
            ),
        load_key(Fname,Key)
        ),recordz(Key,$$,_)                %else

        )
        ),
        (retract(abandon),recordz(Key,$$,_))    %else
        ).

% the rwrite_out predicate writes out the rules to the specified file and as
% it is writing them out ensures that they are in a form which will enable
% them to be viewed in the edit box of the rule editor the next time they
% loaded. It does this by placing a return after every 'then', 'and', '.'
% in the rules.

rwrite_out([],H).
rwrite_out([Str|T],H) :-
        ifthenelse( (not(string_search($.$,Str,_))),
                        concat(Str,$.$,Ostr),
                        Ostr = Str
                ),
                split_write(H,Ostr),
        nl(H),nl(H),
        rwrite_out(T,H).

split_write(H,Str) :-
        string_search($then$,Str,Pos),
        Apos is Pos + 5,        %including space
        substring(Str,0,Apos,String),
        write(H,String),nl(H),
        string_length(Str,Len),
        Alen is Len - Apos,
        substring(Str,Apos,Alen,Substr),
        split_write(H,Substr).

split_write(H,Str) :-
        string_search($and$,Str,Pos),
        Apos is Pos + 4,        %including space
        substring(Str,0,Apos,String),
        write(H,String),nl(H),
        string_length(Str,Len),
        Alen is Len - Apos,
        substring(Str,Apos,Alen,Substr),
        split_write(H,Substr).

split_write(H,Str) :-
        string_search($.$,Str,Pos),
        Apos is Pos + 1,
        substring(Str,0,Apos,String),
        write(H,String).
```

```
% sfl is the user defined part of dialog box predicate manager for the
% filename dialog box.

sfl(command(_,ok),filename) :-
        [! send_control_msg(ef_set_text(Oldtext,Oldtext),2,filename),
        send_control_msg(update,2,filename),
        assert(filename(Oldtext)) !],
        fail.

sfl(command(_,cancel),filename) :-
        [! assert(abandon) !],fail.


sfl(Msg,filename) :-
        def_dialog_fn(Msg,filename).
```

```
:- segment(far1).
:- public rcreat/0,pass1/1:far,pass2/1:far,pass3/2:far.
:- visible drl/2.  % RULE EDITOR DIALOG BOX PREDICATE
:- extrn read_rules/1,save_rules/1.       % in advsavrl.ari
:- extrn split/1.                  % in advsplit.ari
:- extrn rp_arse/0.                       % in advrprsr.ari
:- extrn append/3,member/2.
:- extrn subst/4.                         % in listsub.ari
:- extrn showlist/0.                      % in advhlsr.ari
:- extrn getlist/0:far.                   % in advloadr.ari
:- extrn ferr/0:interp.                   % set in pass1 and rparse in advrprsr.ari
:- extrn abandon/0:interp.                % set if user abondons filename routine
:- extrn grabbed/1:interp.                % set if user grabs word from displayed
                                          % help word list dialog box.


% rcreat is used to read in a rule file if it exists and invoke the rule
% edit dialog box which serves as a means of viewing rules and also ammending
% the rules. On exiting the dialog boc the rules are saved to a file which
% the user specifies the name.

rcreat :-
        gc,
        recordz(fkey,$Enter filename to be created or ammended$,R),
        read_rules(vkey),
        erase(R),
        expunge,!,
        retract(filename(_)),
        assert(saverule($$)),  % used to hold the rule string that is all the
                               % rules in the edit box.
        assert(newrules([])),
        recordz(rerr,$Errors.....$,_),
        load_key('advrdisp.hlp',hkey),
        (dialog_run(display_rules,drl);write('failed')),
        abolish(saverule/1),
        ifthenelse(abandon,retract(abandon),
                   (
                    assert(temp([])),
                    getlist,
                    retract(temp(L)),
                    save_rules(L)
                   )  % save the rules to file
                  ),
        abolish(newrules/1),
        eraseall(hkey),
        eraseall(vkey),
        eraseall(rerr),
        expunge.
```

```
% F10 compiles the rules

drl(char(0,68),display_rules) :-
        eraseall(rerr),
        (retract(newrules(L));true), % for F10 pressed twice in succession
        assert(newrules([])),
        expunge,
        recordz(rerr,$Errors.....$,_),
        send_control_msg(ef_set_text(Old,
                        $Compiling rules please wait.....$),8,display_rules),
        send_control_msg(update,8,display_rules),
        assert(temp([])),
        getlist,        % convert the rules as recorded under the
                                % database key 'vkey' into a list of rules.
        retract(temp(Rules)),
        (rparse(Rules);true),!,
        ifthenelse(ferr,
                (send_control_msg(ef_set_text(Oldtext,
                        $There are errors...Press Alt + E to display them$),8,display_rules),
                 send_control_msg(update,8,display_rules),
                  retract(ferr)
                ),
                (send_control_msg(ef_set_text(Oldtext,
                        $All rules sucessfully compiled.....$),8,display_rules),
                 send_control_msg(update,8,display_rules)
                )
            ),
        abolish(ferr/0),  % clear error flags having
        assert(saverule($$)).

% Alt + R brings the edit box which contains the rules into focus.

drl(char(0,19),display_rules) :-
        [! send_control_msg(ef_set_text(Oldtext,
        $Press F10 to compile rules.....$),8,display_rules),
        send_control_msg(update,8,display_rules)!],fail.


drl(command(_,errs),display_rules) :-
        showerrs.                       % shows the errors if any occured


drl(command(_,help),display_rules) :-
        helprun.                        % displays help on rule format and
                                        % edit keys.
```

```
drl(command(_,disp),display_rules) :-   % Alt + D display defaults
        showlist,                       % contained in advhlsrn.ari
        ifthen(grabbed(Atom),           % This does not work satisfactorially
                (retract(grabbed(_)),
                 recordz(vkey,Atom,_),
                 send_control_msg(update,6,display_rules)
                )
              ).

drl(command(_,cancel),display_rules) :-
        [! assert(abandon) !],fail.

drl(Msg,display_rules) :-
        def_dialog_fn(Msg,display_rules).

showerrs :-
        dialog_run(error_box).

helprun :-
        dialog_run(help_box).

forward:-
        key(vkey,Key),
        nref(Key,Ref),
        goforward(Ref).
forward.

goforward(Ref) :-
        instance(Ref,Term),
        retract(saverule(Str)),
        concat(Str,Term,Nstr),
        assert(saverule(Nstr)),!,
        nref(Ref,Nref),
        goforward(Nref).
```

```
% rparse parses the list of rules passed down to it. It compiles the rules
% three passes. The first pass checks to see if the 'if' and 'then' and '.'
% tokens are in the correct positions. The second pass splits the rules into
% the form
%                   consequences,conditions.
%
% which is easier for the third pass to process. The third pass ensures that
% checks the semantics and syntax of the rules.
% One of the reasons for parsing in three passes is that as it is a recursive
% if I was to do it in one large predicate I would quickly run out of stack
% space.
%
rparse(List) :-
          gc,!,
          pass1(List),!,
          gc,!,
          pass2(List),              % pass2 updates the newrules list
          gc,!,
          retract(newrules(L)),!,
          pass3(List,L),
          abolish(currentgoal/1),
          abolish(currentrules/1),
          gc.

% Pass1 search all the rules for the 'if', 'then' '.' format.

pass1([]).
pass1([H|List]) :-
       ifthen((not(string_search($if$,H,_))),
                  (
                   concat([H,$ $,$missing <<if>>$,$ $],Err2),
                   recordz(rerr,Err2,_),
                   ifthen((not(ferr)),assert(ferr))
                  )
                ),
          ifthen((not(string_search($then$,H,_))),
                  (
                   concat([H,$ $,$missing <<then>>$,$ $],Err3),
                   recordz(rerr,Err3,_) ,
                   ifthen((not(ferr)),assert(ferr))
                  )
                ),
          pass1(List).
```

```
% pass2 : gather together the consequences and actions.
pass2([]).
pass2([H|List]) :-
        string_search($if$,H,Posi),
        string_search($then$,H,Post),
        Pos2 is Posi + 2,
        Pos3 is Post + 4,
        string_length(H,Len),
        N is Len - Pos3,
        End is N - 1,
        Pos22 is Post - 3,
        substring(H,Pos2,Pos22,Conditions),
        substring(H,Pos3,End,Consequences),
        concat([Consequences,$,$,Conditions],Trule),
        retract(newrules(L)),
        append(L,[Trule],Newl),
        assert(newrules(Newl)),
        expunge,
        pass2(List).


% pass3 : Two lists are passed to this predicate  The first list is rules
% in english form. This is used so that if an error occurs the rule can be
% cited together with the error which occurred. The second list is the
% equivalent of this rule in the form which has been constructed by pass2.
% rp_arse is the actual parsing predicate and is contained in advrprsr.ari.

pass3([],[]).
pass3([Rh|Rlist],[H|List]) :-

        string_search($,$,H,Pos1),
        substring(H,0,Pos1,Conditions),
        Pos2 is Pos1 + 1,
        string_length(H,Len),
        Pos3 is Len - Pos1,
        Pos33 is Pos3 - 1,
        substring(H,Pos2,Pos33,Consequences),
        assert(currentrule(Rh)),          % current rule in case of error
        assert(currentgoal(Conditions)),!,      % current goal which might be faulty
        rp_arse,                                % parse the Condition.
        retract(currentgoal(Conditions)),
        assert(currentgoal(Consequences)),!,
        rp_arse,                                % parse the Consequence.
        retract(currentgoal(Consequences)),
        retract(currentrule(_)),
        expunge,
        pass3(Rlist,List).
```

```
:- segment(far1).
:- visible leq/4,geq/4,eq/4,grt/4,lst/4.
:- extrn add_all/2.        % in advappmu.ari
:- extrn append/3.
:- extrn index/2:interp. % this is a table of the parameter list names.


% The following predicates are used to work out the range values to be
% assigned to the various slots in the table of range value pairs which
% are associated with the index list of technical paramaters names. The
% assignment is based on the number line principal.

leq(Index,Val,Xn,Xn1) :-          % [infin,Val]
        index(Xn,Xn1,Index,Pos),
        upper_place(Xn,Xn1,Pos,Val).

geq(Index,Val,Xn,Xn1) :-                  % [Val,infin]
        index(Xn,Xn1,Index,Pos),
        lower_place(Xn,Xn1,Pos,Val).

grt(Index,Val,Xn,Xn1) :-          % [Val,infin]
        index(Xn,Xn1,Index,Pos),
        lower_place(Xn,Xn1,Pos,Val).

lst(Index,Val,Xn,Xn1) :-          % [infin,Val]
        index(Xn,Xn1,Index,Pos),
        upper_place(Xn,Xn1,Pos,Val).

eq(Index,Val,Xn,Xn1) :-           % [Val,Val]
        index(Xn,Xn1,Index,Pos),
        place(Xn,Xn1,Pos,Val).


index(Xn,Xn1,I,Pos) :-
        index(I,Pos).

index(Xn,Xn1,I,Pos) :-
        nl,write('No such index '),write(I),!,fail.

place(Xn,Xn1,Pos,Val) :-
        ctr_set(1,1),
        pl(Xn,Xn1,Pos,Val).

pl([[A1,B1]|T1],[[A2,B2]|T2],Pos,Val) :-
        ctr_is(1,N),
        N == Pos,
        A2 = Val,I,
        B2 = Val,
        T2 = T1.

pl([H1|T1],[H2|T2],Pos,Val) :-
        ctr_inc(1,_),!,
        H2 = H1,
        pl(T1,T2,Pos,Val).
```

```
lower_place(Xn,Xn1,Pos,Val) :-
        ctr_set(1,1),
        lplace(Xn,Xn1,Pos,Val).

lplace([[A1,B1]|T1],[[A2,B2]|T2],Pos,Val) :-
        ctr_is(1,N),
        N == Pos,
        check_lower_bounds(A1,B1,A2,B2,Val),
        T2 = T1.

lplace([H1|T1],[H2|T2],Pos,Val) :-
        ctr_inc(1,_),
        H2 = H1,
        lplace(T1,T2,Pos,Val).


check_lower_bounds(A1,B1,A2,B2,Val) :-
        var(A1),var(B1),
        A2 = Val,B2 = infin.

check_lower_bounds(A1,B1,A2,B2,Val) :-
         B1 == infin,
        Val > A1,
        A2 = Val,
        B2 = B1.



check_lower_bounds(A1,B1,A2,B2,Val) :-
        B1 == infin,
        Val < A1,
        A2 = Val,
        B2 = B1.

check_lower_bounds(A1,B1,A2,B2,Val) :-
        A1 == infinminus,
        Val < B1,
        A2 = Val,
        B2 = B1.

check_lower_bounds(A1,B1,A2,B2,Val) :-
        B1 =\= infin,
        A1 =\= infin,
        (Val @> B1,Val @> A1),
        A2 = A1,
        B2 = Val.
```

```
check_lower_bounds(A1,B1,A2,B2,Val) :-
        B1 =\= infin,
        A1 =\= infin,
        (Val @< B1,Val @< A1),
        A2 = Val,
        B2 = B1.


check_lower_bounds(A1,B1,A2,B2,Val) :-
        B1 =\= infin,
        A1 =\= infin,
        (Val @> A1,Val @< B1),
        A2 = Val,
        B2 = B1.


upper_place(Xn,Xn1,Pos,Val) :-
        ctr_set(1,1),
        uplace(Xn,Xn1,Pos,Val).

uplace([[A1,B1]|T1],[[A2,B2]|T2],Pos,Val) :-
        ctr_is(1,N),
        N == Pos,
        check_upper_bounds(A1,B1,A2,B2,Val),
        T2 = T1.


uplace([H1|T1],[H2|T2],Pos,Val) :-
        ctr_inc(1,_),
        H2 = H1,
        uplace(T1,T2,Pos,Val).

uplace(Xn,Xn1,Pos,Val).

check_upper_bounds(A1,B1,A2,B2,Val) :-          % [-infin,B]   leq
        var(A1),var(B1),
        A2 = infinminus,B2 = Val.

check_upper_bounds(A1,B1,A2,B2,Val) :-          % [A,B] leq,leq
         A1 == infinminus,
        Val < B1,
        A2 = Val,
        B2 = B1.



check_upper_bounds(A1,B1,A2,B2,Val) :-          % [-infin,Val]     leq,leq
        A1 == infinminus,
        Val > B1,
        A2 = A1,
        B2 = Val.
```

```
check_upper_bounds(A1,B1,A2,B2,Val) :-          % [A,B] geq,leq
        B1 == Infin,
        Val > A1,
        A2 = A1,
        B2 = Val.

check_upper_bounds(A1,B1,A2,B2,Val) :-          % rearrange boundaries
        integer(A1),integer(B1),
        (Val < B1,Val < A1),
        A2 = Val,
        B2 = B1.

check_upper_bounds(A1,B1,A2,B2,Val) :-
        integer(A1),integer(B1),
        (Val > B1,Val > A1),
        A2 = A1,
        B2 = Val.

check_upper_bounds(A1,B1,A2,B2,Val) :-
        integer(A1),integer(B1),
        (Val > B1,Val < A1),
        A2 = A1,
        B2 = Val.
```

```
:- segment(far1).
:- public go/0:far.
:- visible quest/2,solution/2. % DIALOG BOX PREDICATES
:- visible default/2. % This clause is added to the meta rules and calls
                      % the default predicate defined in this file to
                      % invoke the question dialog box if a

:- visible trace/2.   % A trace clause is also added to all the compiled rules
                      % and this is used to call the trace predicate defined in
                      % this file.

:- extrn member/2,append/3,fill/3,subst/4.

:- extrn table/1:interp.   % contains the slots for the index items.
:- extrn values/1:interp.          % used to record the values that have
                                   % been used to reach the established facts.
:- extrn why/1:interp.             % records the rule which is currently being
                                   % proved to be displayed in response to a why
                                   % question by the user.

:- extrn caps/2:interp.            % The head of the first meta rule to be found
                                   % in the program database
:- extrn index/2:interp.   % contains the names of the technical parameters
                                   % in the associated database.
:- extrn traceon/0:interp.         % set by the debugger.
:- extrn explain/4:interp.% contains the customised explanations.
:- extrn skipcall/0:interp.        % used to disable the call mechanism of the
                                   % debugger.
:- extrn skipsuccess/0:interp.     % used to disable the success mechanism of the
                                   % debugger.


% this predicate is used to get the list of facts which make up the description
% as set up in the application descriptor.

get_app_desc(Str,L) :-
        atom_string(A,Str),
        Clause =.. [A,L],
        call(Clause).
```

```
% The go predicate gets the application description if one has been provided
% and then fires this description on the loaded rule base.
%
go :-
  assert(values([])), % used to record the values that have been used to
                      % reach the established facts
  retract(applic_name(Applic)),
  get_app_desc(Applic,L),
  establish_init_facts(L),
  (fire;
  (
  recordz(msg,$Not sufficient information to make conclusion$,_),
  recordz(msg,$Rule base might not be adaquit for particular choice...$,_),
  dialog_run(message),
  eraseall(msg),
  ifthen(traceon,exit_popup), % to close debugger window on failing to start
  expunge
  )),
  retract(start(Newl)),       % get the application description as changed
                              % by answers given by the user to the questions asked
                              % start is the name of the description
                              % clause allways as is set up in advappmu .
  erase_initial_conditions(Newl).

fire :-
        reconsult('advtabls.ari'),
        table(X0),              % get two copies of the table list
        table(Xn),!,            % The starting and finishing list.
        ifthen(traceon,
        create_popup('D E B U G G E R ',(1 , 1),(20 , 78),(120 , 113))),!,
        caps(X0,Xn),!,  % this calls the super meta rule which is the starting
                        % point for the rule activation. This rule has to be
                        % present in the program database.
        ifthen(traceon,exit_popup),
        format(Xn,1),           % default format to report findings established
                                % from choices .
        assert(save(Xn)),       % used to write out the list to file to be
                                % processed by a CAPS program.
        retract(values(Newl)),  % used to record the values that have been used to
                                % reach the established facts.
        list_string(Newl,Str),  % used to convert the list of values established
                                % into a sentence to be displayed in the
                                % solution dialog box.
        recorda(skey,$$,_),
        recorda(skey,Str,_),
        recorda(skey,$ The following facts where established for $ ,_),
        (dialog_run(solution_box,solution);(write('solution failed'),get0(_))),
        eraseall(skey),!,
        eraseall(rtrace),!,
        abolish(table/1),
        abolish(skipcall/0),
        abolish(skipsuccess/0),
        abolish(traceon/0),
        expunge.
```

```
% The default predicate is called if a value cannot be established for a
%  meta rule descriptor. This results in a question being asked of the user.

default(Name,X) :-
        atom_string(Name,SName),
        concat($Unable to establish value for...$,SName,Str1),
        recordz(mess,Str1,_),
        recordz(mess,$choose value or choose Don't Know.$,_),
        dialog_run(question,quest),
        eraseall(mess),
        expunge,
        !,
        retract(chosen(X)),
        retract(values(V)),
        ifthenelse(member(X,V),assert(values(V)),
                               (append([X],V,NV),
                                 assert(values(NV))
                                 )
                   ),
% assert newly known fact so that it is not asked for again in subsequent
% redos of the caps predicate.
% Replace old fact with new fact for the same descriptor.
        Oldclause =.. [Name,Old],
        (retract(Oldclause);true), % To allow for the use of the Go function
        Newclause =.. [Name,X],
        asserta(Newclause),!,
% update the start clause to reflect the new fact so that it can be
% deleted when the rule consult is finished .
        retract(start(L)),
        ifthenelse(member(Oldclause,L),
                   subst(Oldclause,L,Newclause,Newl),
                   append([Newclause],L,Newl)
                  ),
        asserta(start(Newl)),
        expunge.

% quest is The user defined dialog dialog box management predicate for the
% the question dialog box.

% F3 updates the lists
quest(char(0,61),question):-
%       first,
%       retract(first),
        send_control_msg(lb_clear,2,question),
        send_control_msg(update,2,question),
        send_control_msg(ef_set_text(Oldtext,$$),3,question),
       send_control_msg(update,3,question),
        fill(val,question,2) .
```

```
quest(command(_,ok),question) :-
        [! send_control_msg(lb_set_index(I,I),2,question),
        send_control_msg(lb_get_text(I,Str),2,question),
        atom_string(A,Str),
        assert(chosen(A))
        !],fail.

quest(command(_,why),question) :-
        create_popup('E X P L A N A T I O N',(1 , 1),(20 , 78),(120 , 113)),!,
        write('Trying to answer the following question'),nl,
        why(Rule),
        write(Rule),nl,
        get0(_),
        exit_popup.

quest(command(_,how),question) :-
                recordz(rtrace,$Facts established by answers to questions$,R1),
                recordz(rtrace,$ and original query $,R2),
               recorda(rtrace,$----------$,R5),
                values(NewI),
                list_string(NewI,Str),
                recordz(rtrace,Str,R3),
                recordz(rtrace,$----------$,R33),
                recorda(rtrace,$Rules Used so far to get this far $,R4),
                dialog_run(how_box),
                erase(R1),erase(R2),erase(R3),erase(R4),erase(R5),erase(R33),
                expunge.

quest(Msg,question) :-
        def_dialog_fn(Msg,question).
```

```
% format takes the values which have been assigned to slots in the table
% together with the associated index name and constructs english like sentences
% to report the findings of the system to the user in an easy to read manner.

format([],_).
format([[A,B]|T],Index) :-
        nonvar(A),nonvar(B),
        index(Name,Index),
        assert(save(Name,A,B)),                 % save all the established values
        atom_string(Name,String),       % to be used in diagnostic explanation
        ifthenelse((number(A),number(B)),
                        (int_text(A,Astr),
                         int_text(B,Bstr),
                            i f t h e n e l s e ( A = =
B,single_range_eq(String,Astr),double_range(String,Astr,Bstr))
                        ),
                        (
% all the following should be mutually exclusive .

                        ifthen((number(A),B == infin),
                                        (
                                         int_text(A,Astr),
                                         single_range_geq(String,Astr)
                                        )
                                ),
                        ifthen((number(B),A == infinminus),
                                        (
                                         int_text(B,Bstr),
                                         single_range_leq(String,Bstr)
                                        )
                                ),
                        ifthen((atom(A),atom(B),A == B),
                                        (
                                         atom_string(A,Astr),
                                         single_range_eq(String,Astr)
                                        )
                                )
                        )
                ),
        I is Index + 1,
        format(T,I).

format([H|T],I) :-
        Index is I + 1,
        format(T,Index).

single_range_eq(String,Rangeval) :-
        concat([$The value for $,String,$ should be $,Rangeval],Sentence),
        recordz(skey,Sentence,_).
```

```
single_range_geq(String,Rangeval) :-
        concat([$The value for $,String,$ should be equal or greater than $,Rangeval],Sentence),
        recordz(skey,Sentence,_).

single_range_leq(String,Rangeval) :-
        concat([$The value for $,String,$ should be equal or less than $,Rangeval],Sentence),
        recordz(skey,Sentence,_).

double_range(String,Rangeval1,Rangeval2) :-
        concat([$The value for $,String,$ should be between $,Rangeval1,$ and $,Rangeval2
],Sentence),
        recordz(skey,Sentence,_).


% assert the facts built up in the application descriptor into the database
% so that the goals in the rules will find them and succeed.

establish_init_facts([]).  % the initial facts are the description of the
establish_init_facts([H|T]) :-  % application type
        retract(values(L)),        % used to record the values that have been used to
                                   % reach the established facts
        H =.. [Name,Val],
        asserta(H),
        append([Val],L,Nl),
        assert(values(Nl)),
        establish_init_facts(T).

% erase all the facts that have been established when the rule consultation is
% finished so that they will not interfer with further rule consultations.

erase_initial_conditions([]).
erase_initial_conditions([H|T]) :-
                retract(H),
                erase_initial_conditions(T).

% used to convert a list of atoms to a string
list_string([],$$).
list_string([H|T],S) :-
        atom_string(H,S1),
        list_string(T,S2),
        concat([S1,$ $,S2],S).
```

```
% The following are used by the rule tracer to print out the calling and
% success of rules and goals.
% The trace predicate is built into each rule and is called from that rule
% However the rules are only displayed if the rule tracer is turned on otherwise
% they are not displayed but the tracer predicate still succeeds.

trace(call,Rule) :-
        traceon,
        twrite(calling,Rule).

trace(call,Rule) :-
        assert(why(Rule)).


trace(true,Rule) :-
        [! traceon,
        twrite(succeeded,Rule) !],fail.
% if trace is off still record successful rules as these are used in the
% success dialog box to show what rules where used to reach the current
% conclusions.

trace(true,Rule) :-
        recorda(rtrace,Rule,_).
% Catch all to catch cases where tracer is not turned on or
% Rule which is bieng traced fails
trace(M,Rule).


twrite(calling,Rule) :-
        ((skipcall);
        (nl,write(calling),write( ' : '),write(Rule),write(' : '),get0(C))),
        check_mess(C).


twrite(succeeded,Rule) :-
        ((skipsuccess);
        (nl,write(succeeded),write( ' : '),write(Rule),write(' : '),get0(C))),
        check_mess(C).


twrite(M,R).

% check if the user wants to change any of the settings of the rule tracer.
check_mess(C) :-
        ifthen((C == 99,not(skipcall)),assert(skipcall)), % c
        ifthen((C == 115,not(skipsuccess)),assert(skipsuccess)), % s
        ifthen(C == 108,(assert(skipcall),assert(skipsuccess))).  % l
```

```
solution(command(_,expln),solution_box) :-
        explain.

solution(command(_,ok),solution_box) :-
        [!
        retract(save(L)),
        abolish(save/3),
        expunge,
        spec_write_out(L)
        !],fail.

solution(command(_,cancel),solution_box) :-
        [!
        abolish(save/3),
        retract(save(L)),
        expunge
        !],fail.

solution(Msg,solution_box) :-
        def_dialog_fn(Msg,solution_box).


% explain displays the customised expalanations for values established if
% they have been loaded into the program database.
explain :-
        assertz(save(last,v,v)),
        repeat,
        [!
        retract(save(Name,Val1,Val2)),
        ifthen(not(Name == last),
                        (
                         (explain(Name,Val1,Val2,Cure), % the loaded
                                                        % explanation clauses.
                          expln(Name,Val1,Val2,Cure)    % set up the
                                                        % explanations.
                         );
                         nexplain(Name,Val1,Val2)
                        )
                )
        !],
        Name == last,!,
        dialog_run(explain_box),        % show the explanations.
        eraseall(exkey),expunge.

expln(I,V1,V2,Exp) :-
        V1 = V2,
        atom_string(I,S),
        ifthenelse(atom(V1),atom_string(V1,S2),int_text(V1,S2)),
        concat([$If $,S,$ is $,S2],S3),
        recordz(exkey,S3,_),
        recordz(exkey,Exp,_).
```

```
explin(I,V1,V2,Exp) :-
        V1 \= V2,
        atom(V1),atom(V2),
        atom_string(I,S),
        atom_string(V1,S1),
        atom_string(V2,S2),
        concat([$If $,S,$ is $,S1,$ or $,S2],S3),
        recordz(exkey,S3,_),
        recordz(exkey,Exp,_).

explin(I,V1,V2,Exp) :-
        V1 \= V2,
        integer(V1),integer(V2),
        atom_string(I,S),
        int_text(V1,S1),
        int_text(V2,S2),
        concat([$If $,S,$ is $,S1,$ or $,S2],S3),
        recordz(exkey,S3,_),
        recordz(exkey,Exp,_).

explin(I,V1,V2,Exp) :-
        V1 \= V2,
        atom(V1),integer(V2),
        atom_string(I,S),
        int_text(V2,S2),
        concat([$If $,S,$ is less than or equal to $,S2],S3),
        recordz(exkey,S3,_),
        recordz(exkey,Exp,_).

explin(I,V1,V2,Exp) :-
        V1 \= V2,
        integer(V1),atom(V2),
        atom_string(I,S),
        int_text(V1,S1),
        concat([$If $,S,$ is greater than or equal to $,S1],S3),
        recordz(exkey,S3,_),
        recordz(exkey,Exp,_).

nexplain(Name,Val1,Val2) :-
        atom_string(Name,Str),
        concat($There is no explantion loaded for $.,Str,Strn),
        recordz(exkey,Strn,_).
```

```
% write out the parameter list values established to a file to be processed
% by the CAPS database system.

spec_write_out(L) :-
        create_popup('O U T L I S T',(1 , 1),(20 , 78),(120 , 113)),!,
        create(F,'advout.dat'),
        spec_write(F,L,1),
        exit_popup,
        close(F).

spec_write(F,[],N):-
        nl,write('Finished.....'),nl,
        write('Press any key to continue.....'),get0(_).

spec_write(F,[[H1,T1]|T],N) :-
        var(H1),var(T1),
        index(Name,N),
        nl,
        write('Processing ..... '),write(Name),nl,
        write('Val1 '),nl,
        write('Val2 '),
        write(F,Name),write(F,'  '),
        nl(F),
        N1 is N + 1,
        spec_write(F,T,N1).


spec_write(F,[[H1,T1]|T],N) :-
        integer(H1),atom(T1),
        index(Name,N),
        nl,
        write('Processing ..... '),write(Name),nl,
        write('Val1 '),write(H1),nl,
        write('Val2 '),write(T1),
        write(F,Name),write(F,'  '),
        write(F,H1),write(F,' ++'),
        nl(F),
        N1 is N + 1,
        spec_write(F,T,N1).


spec_write(F,[[H1,T1]|T],N) :-
        atom(H1),integer(T1),
        index(Name,N),
        nl,
        write('Processing ..... '),write(Name),nl,
        write('Val1 '),write(H1),nl,
        write('Val2 '),write(T1),
        write(F,Name),write(F,'  '),
        write(F,'-- '),write(F,T1),
        nl(F),
        N1 is N + 1,
        spec_write(F,T,N1).
```

```
spec_write(F,[[H1,T1]|T],N) :-
        integer(H1),integer(T1),
        index(Name,N),
        nl,
        write('Processing ..... '),write(Name),nl,
        write('Val1 '),write(H1),nl,
        write('Val2 '),write(T1),
        write(F,Name),write(F,'  '),
        write(F,H1),write(F,' '),write(F,T1),
        nl(F),
        N1 is N + 1,
        spec_write(F,T,N1).


spec_write(F,[[H1,T1]|T],N) :-
        atom(H1),atom(T1),
        index(Name,N),
        nl,
        write('Processing ..... '),write(Name),nl,
        write('Val1 '),write(H1),nl,
        write('Val2 '),write(T1),
        write(F,Name),write(F,'  '),
        write(F,'*'),write(F,' '),write(F,'*'),
        nl(F),
        N1 is N + 1,
        spec_write(F,T,N1).
```

```
:- segment(far1).
:- public commit/2:far.
:- extrn member/2,append/3.
:- extrn split/1.            % in advsplit.ari
:- extrn data_add/2,mark_deleted/2,db_search/4. % in advappmu.ari

:- extrn deflist/1:interp. % used to hold descriptors that default clauses
                           % have been set up for so far.

:- extrn invalid/0:interp. % set when a rule is found to be invalid which means
                           % it cannot be safely be converted into it's prolog
                           % equivalent.
:- extrn ruletype/1:interp. % records the rule type that one is working with.
:- extrn sentence/1:interp. % used to store individual tokens when sentence
                            % is split.


signtype($=<$,$leq$). % the appropriate built in predicate for each type of
                      % arithmetic sign.
signtype($>=$,$geq$).
signtype($=$,$eq$).
signtype($<$,$lst$).
signtype($>$,$grt$).

% comit rules will only be called if all the rules are valid.
commit(F,[]).
commit(F,[H|T]) :-
        construct_rule(H,F),
        nl(F),
        ifthen(retract(dflt(D)) ,(write(F,D),nl(F))),
        commit(F,T).


construct_rule(Rule,X) :-
      ifthen(ruletype(mrl),construct_meta(Rule,X)),
      ifthen(ruletype(crf),ordinary_rule(Rule,X)).

construct_meta(Rule,X) :-
      ifthenelse(string_search($start$,Rule,_),
                 super_rule(Rule,X),
                 meta_rule(Rule,X)
                ).
```

```
% To build a super rule :
% 1. Find the head of the english rule and construct a prolog head for it.
% The head of the super rule is of the form caps(X0,Xn)
% 2. Find the tail of the rule ( the conditions ) and construct a series of
% goals.
% 3. Add the trace predicates and add the default questioning predicate for every
% new value that is to be established for a descriptor word.
% 4. Concatonate the two together with the prlog operators to form a prolog
% rule.
super_rule(Rule,F) :-
        [! Strx = $X0,Xn$,
        string_search($then$,Rule,Pos),
        Apos is Pos - 2,                    % allow for if
        substring(Rule,2,Apos,Head),
        concat(Head,$ es$,Nh),
        split(Nh),
        repeat,
        retract(sentence(C)),
        ifthen((db_search(val,C,Flag,_),Flag == found),
                (
                 concat([C,$($,Strx,$)$],Strh),
                 assert(temp(Strh))
                )
              ),
        C == $es$,
        Tpos1 is Pos + 4,
        string_search($.$,Rule,Tpos2),
        Tlen is Tpos2 - Tpos1,
        substring(Rule,Tpos1,Tlen,Tail),
        construct(Tail,Strt),
        retract(temp(Strh))
        !],
        write(F,Strh),
        write(F,$ :- [!trace(call,$), % add in trace clause.
        writeq(F,Rule),
        write(F,$}!], $),
        write(F,Strt),
        write(F,$,trace(true,$),
        writeq(F,Rule),
        write(F,$). $).

super_rule(Rule,Compiled_rule) :-
        write('Invalid Super rule ......'),nl,
        write(Rule),nl.
```

```
% The tail of a super rule consists of a series of goals which will
% be used to establish values for descriptor names all these variables which
% are used to represent these possible values are used in the call to the next meta
% rule level through the goal docaps(X1,...,Xt,X0,Xn) where X1,...,Xt are the
% established values and X0 and Xn are the parameter list slots before and
% after a rule has been invoked.

construct(Tail,Strt) :-
        [!
        concat(Tail,$ es$,Nt),
        Strx = $X$,
        assert(temp1($$)),
        assert(temp2($docaps($)),
        assert(temp3($X0,Xn)$)),
        split(Nt),
        ctr_set(0,1),
        repeat,
        retract(sentence(C)),
        ifthen((db_search(dsc,C,Flag,_),Flag == found),
                (                               % construct the goals.
                 retract(temp3(S3)),
                 retract(temp1(S1)),
                 ctr_inc(0,Y),
                 int_text(Y,T),
                 concat(Strx,T,Strxn),
                 concat([$[!$,C,$($,Strxn,$)!]$,$,$],Strn), % e.g. [!apptype(X1)!],
                 concat([Strxn,$,$,S3],Nstrn3),
                 concat(S1,Strn,Nstrn1),
                 assert(temp3(Nstrn3)),
                 assert(temp1(Nstrn1)),
                 create_default(C,D),
                 assert(dflt(D)),
                 expunge
                )
        ),
        C == $es$,
        retract(temp1(Str1)),
        retract(temp2(Str2)),
        retract(temp3(Str3)),
        expunge !],                             % will fail if any error
        concat([Str1,Str2,Str3],Strt).
```

```
% create default predicate to be called if the value cannot be established
% the format is descriptor(X) :- default(descriptor,X). The default predicate
% is built into the system ( see advrulgo.ari ).

create_default(Name,D) :-
        retract(deflist(L)),

        ifthenelse(member(Name,L),(assert(deflist(L)),D = $$),
% else
                                (
                                append([Name],L,Newl),
                                assert(deflist(Newl)),
                                concat([Name,$(X) :- default($,Name,$,X).$],D)
                                )
                ),
        expunge.
```

```
% Build the meta rule. The meta rule level should contain a rule which corresponds
% to a docaps call in the super rule level. The head of the meta rule is built
% first. It consists of the goal docaps(val1,val2..valn,X0,Xn) where the values have
% been established at the super rule level. The right hand of the rule contains
% goals which will be solved by calling rules in the ordinary rule levels.

meta_rule(Rule,F) :-
        assert(temp($$)),
        Strx = $X0,Xn$,
        string_search($then$,Rule,Pos),
        Apos is Pos - 2,                    % allow for if
        substring(Rule,2,Apos,Head),
        concat(Head,$ es$,Nh),
        split(Nh),
        repeat,
        retract(sentence(C)),
        ifthen((db_search(val,C,Flag,_),Flag == found),
                (
                 retract(temp(Str)),
                 concat([C,$,$,Str],Strh),
                 assert(temp(Strh))
                )
            ),
        C == $es$,
        retract(temp(Str)),
        concat([$docaps($,Str,Strx,$)$],Temp),!,          % e.g. docaps(dishes,X0,Xn) :-
        concat([Temp,$ :- [!trace(call,$,Temp,$,R)!], $],Nstr),
        string_search($.$,Rule,End),
        Post is Pos + 4,
        Lent is End - Post,
        substring(Rule,Post,Lent,Tail),
        concat(Tail,$ es$,Cond),!,
        rule_make($$,Cond,Prologform),
        concat([Prologform,$,trace(true,R)$,$ .$],X),
        write(F,Temp),
        write(F,$ :- [!trace(call,$),
        writeq(F,Rule),
        write(F,$)!], $),
        write(F,Prologform),
        write(F,$,trace(true,$),
        writeq(F,Rule),
        write(F,$). $).


meta_rule(Rule,X) :-
        write('Invalid Meta rule .....'),nl,
        write(Rule),nl.
```

```
% This predicate builds up the prolog equivalent of the right hand side
% of a rule.

mrbuild(Str1,Str2,Str3) :-
        split(Str2),
      [! data_add($es$,cop) !] ,
        assert(trule($$)),
        repeat,
        retract(sentence(C)),
       [!
        case( [

% If this clause is called and there exists a current parameter list then
%  assert the current clause and start a new clause.
%
        ([!db_search(dsc,C,Flag,_)!],Flag == found)
        -> ( retract(trule(Str)),
             concat([Str,C,$($],Newstr),
             assert(trule(Newstr))
             ),  % end dsc      case

% A conjugate or an operator is appended to the end of the parameter list
% except for the special case of 'is' which is fundemental and is contained
% in the meaning of the clause itself .
%
        ([!db_search(cop,C,Flag,_)!],Flag == found)
        -> (
            ifthen(C == $and$ ,
                        ( retract(trule(Str)),
                          concat(Str,$,$,Nstr),
                          assert(trule(Nstr))
                        )
                   )
            ),      %end cop case

% A values is just appended to the list .
        ((([!db_search(val,C,Flag,_)!],Flag == found));(int_text(I,C)))
        -> (retract(trule(Str)),
            concat([Str,C,$,X0,Xn )$],Newstr),
            assert(trule(Newstr))
           )
           | (assert(invalid),assert(invalid(C)))
        ]) !]      % end case
        , C == $es$,
        retract(trule(Str4)),
        mark_deleted($es$,cop),
        expunge,!,
       ( (invalid,(retract(invalid),!,fail)) ;(concat(Str1,Str4,Str3)) ).
```

```
% The ordinary rule construction is similar to the construction of super and
% meta rules. The head can only have one clause of the form descriptor(value,X0,Xn). The
% body of the rule is made up of clauses which are either calls to other rules
% or else calls to the built in predicates {eq,leq,lst,grt,geq} which can be
% found in advbuild.ari.

ordinary_rule(Rule,F) :-
        get_head(Rule,H),
        get_body(Rule,B),!,
       rule_make($$,H,PrologH),
        concat([PrologH,$ :- [!trace(call,$,PrologH,$,R)!], $],NewPrologH),!,
        rule_make($$,B,PrologClause),
        concat([PrologClause,$,trace(true,R)$,$ .$],X),
        write(F,PrologH),
        write(F,$ :- [!trace(call,$),
        writeq(F,Rule),
        write(F,$)!], $),
        write(F,PrologClause),
        write(F,$,trace(true,$),
        writeq(F,Rule),
        write(F,$). $).


get_head(Rule,H) :-
        string_search($then$,Rule,Pos),
        Apos is Pos - 2,                    % allow for if
        substring(Rule,2,Apos,Head),!,
        concat(Head,$ es$,H).

get_body(Rule,B) :-
        string_search($then$,Rule,Pos),
        Apos is Pos + 4,
        string_search($.$,Rule,End),
        Len is End - Apos,
        substring(Rule,Apos,Len,Body),!,
        concat(Body,$ es$,B).
```

```
% The rule_make predicate is used to construct the prolog equivalent of english
% rules.

rule_make(_,$$,_).
rule_make(Str1,Str2,Str3) :-
        split(Str2),
       [! data_add($es$,cop) !] ,
        assert(conj($$)),
        assert(trule($$)),
        assert(last(0)),
        assert(next(1)),
        repeat,
        retract(sentence(C)),
       [!
        case( [

% If this clause is called and there exists a current parameter list then
%  assert the current clause and start a new clause.
%
        ([!db_search(dsc,C,Flag,_)!],Flag == found)
        -> ( retract(trule(Str)),
             concat([Str,C,$($],Newstr),
             assert(trule(Newstr))
             ),  % end dsc      case

% A conjugate or an operator is appended to the end of the parameter list
% except for the special case of is which is fundemental and is contained
% in the meaning of the clause itself .
%
        ([!db_search(cop,C,Flag,_)!],Flag == found)
        -> (
             ifthen( (member(C,[$=$,$<$,$>$,$<=$,$>=$,$<>$]) ),
                        (retract(trule(Str)),
                         signtype(C,Type),
                         switch_op(Str,Type,Newstr),
                         assert(trule(Newstr))
                    )
                     ) ,
            ifthen(C == $and$ ,
                        ( retract(trule(Str)),
                          concat(Str,$,$,Nstr),
                          retract(conj(Cstr)),
                          concat(Cstr,Nstr,Ncstr),
                          assert(conj(Ncstr)),
                          assert(trule($$))
                        )
                   ),
            ifthen(C == $or$ ,
                        ( retract(trule(Str)),
                          concat(Str,$;$,Nstr),
                          assert(trule(Nstr))
                        )
                   )
```

```
            ),      %end cop case

% A values is just appended to the list .
        ([Idb_search(val,C,Flag,_)I],Flag == found)
        -> (retract(trule(Str)),
            retract(last(L)),
            retract(next(N)),
            sentence(X),
            ifthenelse(X == $es$,
                        (                           % then
                         int_text(L,Sl),
                         concat($X$,Sl,Sl1),
                         concat([Str,C,$,$,Sl1,$,$,$Xn)$],Newstr),
                         L1 is N,
                         N1 is N + 1,
                         assert(last(L1)),
                         assert(next(N1))
                        ),
                        (                           % else
                         int_text(L,Sl),
                         int_text(N,Sn),
                         concat($X$,Sl,Sl1),
                         concat($X$,Sn,Sn1),
                         concat([Str,C,$,$,Sl1,$,$,Sn1,$)$],Newstr),
                         L1 is N,
                         N1 is N + 1,
                         assert(last(L1)),
                         assert(next(N1))
                        )
                    ),
            assert(trule(Newstr))
           ) ,
```

```
% if it's not any of those things then check if it's a numeric value
          int_text(I,C)
        -> (retract(trule(Str)),
            retract(last(L)),
            retract(next(N)),
            sentence(X),
            ifthenelse(X == $es$,
                       (                          % then
                        int_text(L,Sl),
                        concat($X$,Sl,Sl1),
                        concat([Str,C,$,$,Sl1,$,$,$Xn)$],Newstr),
                        L1 is N,
                        N1 is N + 1,
                        assert(last(L1)),
                        assert(next(N1))
                       ),
                       (                          % else
                        int_text(L,Sl),
                        int_text(N,Sn),
                        concat($X$,Sl,Sl1),
                        concat($X$,Sn,Sn1),
                        concat([Str,C,$,$,Sl1,$,$,Sn1,$)$],Newstr),
                        L1 is N,
                        N1 is N + 1,
                        assert(last(L1)),
                        assert(next(N1))
                       )
                    ),
            assert(trule(Newstr))
           )

           | (assert(invalid),assert(invalid(C)))
        ]) !]      % end case
        , C == $es$,
        retract(last(_)),
        retract(next(_)),
        retract(trule(Tstr)),
        retract(conj(Cstr)),
        concat(Cstr,Tstr,Str4),
        mark_deleted($es$,cop),
        expunge,!,
       ( (invalid,(retract(invalid),!,fail)) ;(concat(Str1,Str4,Str3)) ).


% When an arithmetic operator is used in a rule then it is replaced by one of the
% built in predicates for assignment { eq,etc. }

switch_op(Str,T,Ns) :-
        string_search($($,Str,Pos),
        substring(Str,0,Pos,S),
        concat([T,$($,S,$,$],Ns).
```

```
:- segment(far1).
:- visible index_explain/0.
:- visible exp/2.
:- extrn index/2:interp.   % The parameter list names
:- extrn abandon/0:interp. % abandon is set if opening a file is abandoned


% The index_explain predicate allows you to set up customised explanation in
% the form of canned text for certain values being assigned to index list items.

index_explain :-
        dialog_run(explain_index,exp).


exp(command(_,nxt),explain_index) :-
          [! send_control_msg(ef_set_text(Index,Index),2,explain_index)!],
             atom_string(A,Index),
             index(A,_), % run error message and beep
             send_control_msg(ef_set_text(Index,$$),2,explain_index),
             send_control_msg(update,2,explain_index),
             send_control_msg(ef_set_text(Val1,$$),5,explain_index),
             send_control_msg(update,5,explain_index),
             send_control_msg(ef_set_text(Val2,$$),8,explain_index),
             send_control_msg(update,8,explain_index),
             send_control_msg(ef_set_text(Explain,$$),11,explain_index),
             send_control_msg(update,11,explain_index),
             atom_string(Ai,Index),
             string_val(Val1,Av1),
             string_val(Val2,Av2),
             ((integer(Av1),Val2 == $$,Ao2 = infin,Ao1 = Av1);
              (integer(Av2),Val1 == $$,Ao1 = infinminus,Ao2 = Av2);
              (Ao1 = Av1,Ao2 = Av2)
             ),
             assertz(explain(Ai,Ao1,Ao2,Explain)).


exp(command(_,nxt),explain_index) :-
        recordz(msg,$Invalid Index name please reenter or Ok...$,_),
        dialog_run(message),
        eraseall(msg).
```

```
exp(command(_,ok),explain_index) :-
        [!
        recordz(fkey,$Enter name for explanation file$,_),
        dialog_run(filename,sfl),
        eraseall(fkey),
        ifthenelse((not(abandon)),
                        (
                         retract(filename(Name)),
                         ifthen( (not(Name == $$)),
                                 (
                                 concat(Name,$.exp$,Fname),
                                 file_list(Fname,explain/4)
                                 )
                               )
                        ),
                        retract(abandon)
                ),
        abolish(explain/4),
        expunge
        !],fail.


exp(command(_,cancel),explain_index) :-
            [!
            assert(explain(_,_,_,_)),
            send_control_msg(ef_set_text(Index,$$),2,explain_index),
            send_control_msg(update,2,explain_index),
            send_control_msg(ef_set_text(Val1,$$),5,explain_index),
            send_control_msg(update,5,explain_index),
            send_control_msg(ef_set_text(Val2,$$),8,explain_index),
            send_control_msg(update,8,explain_index),
            send_control_msg(ef_set_text(Explain,$$),11,explain_index),
            send_control_msg(update,11,explain_index),
            abolish(explain/4),expunge
            !],fail.

exp(Msg,explain_index) :-
        def_dialog_fn(Msg,explain_index).

% if it's an string convert it to an integer else convert it to an atom

string_val(In,Out) :-
        int_text(Out,In).
string_val(In,Out) :-
        atom_string(Out,In).
```

Appendix B

The List of

The CAPS Database

Parameter List

Appendix B : The CAPS parameter list.

1	MECHANICAL PROPERTIES		
02	Density	g/ccm	Dens
08	Tensile Strength	MPa	T.Str
09	Elongation @ break	%	Elong
10	Tensile Modulus	MPa	T.Mod
11	Creep Mod 1 h	MPa	C1h/10
12	Creep Mod 1000 h	MPa	C1000
13	Flexural Strength	MPa	F.Str
14	Flexural Modulus	MPa	F.Mod
15	Charpy Unnotched @ 23xC	KJ/m}	CU23
16	Charpy Unnotched @ -40xC	KJ/m}	CU-40
17	Charpy Notched @ 23xC	KJ/m}	CN23
18	Charpy Notched @ -40xC	KJ/m}	CN-40
19	Izod Notched @ 23xC	J/m	IN23
20	Izod Notched @ -40xC	J/m	IN-40
06	Shore Hardness D	-	Shore
07	Ball Indentation @ 30s	MPa	Ball
1	THERMAL PROPERTIES		
22	HDT @ 1.80 MPa	xC	HDT1.8
21	HDT @ 0.45 MPa	xC	HDT.45
23	Vicat B Temperature (50 N)	xC	Vicat
24	Continuous Service Temperature	xC	CSTemp
25	Linear Thermal Expansion Coeff.	1/K	TECoef
1	ELECTRICAL PROPERTIES		
32	Dissipation Factor @ 1MHz	-	Dissip
31	Dielectric Strength	KV/mm	D.Str
30	Tracking Resistance KC	Volts	T.Res
28	Volume Resistivity	Ohm.cm	V.Res
29	Surface Resistivity	Ohm	S.Res
1	MISCELLANEOUS		
01	Price / dm cubed	DM	Price
05	MFI / 10 minutes	g	MFI
04	Mould Shrinkage	%	M.Shr
26	UL 94 Rating	HB-V0	UL94
27	Oxygen Index	%	O2Ind
03	Water Absorption 23/50%	%	W.Abs
5	CHEMICAL RESISTANCE		
01	Boiling Water		Water
02	Dilute Organic Acids		DOA
03	Concentrated Organic Acids		COA
04	Dilute Mineral Acids		DMA
05	Concentrated Mineral Acids		CMA
06	Concentrated Oxidized Mineral Acids		COMA
07	Bases		Base
08	Alcohols		Alc
09	Aldehydes		Ald
10	Esters		Ester

Appendix B : The CAPS parameter list.

11 Ketones	Ketone
12 Aliphatic Amines	AlAm
13 Aromatic Amines	ArAm
14 Glycols	Glycol
15 Aliphatic Hydrocarbons	AlHC
16 Aromatic Hydrocarbons	ArHC
17 Chlor Hydrocarbons	ClHC
18 Detergents	Deterg
19 Petrol	Petrol
20 Motor Oil	Oil

2 PROCESSING METHODS

01 Injection Moulding	Inj
02 Extrusion	Ext
03 Blow Moulding	Blow
04 Compression Moulding	Comp
49 Film Extrusion	Film

2 ADDITIVES & FILLERS

41 Impact Modifier	Impact
12 UV Stabiliser	UV
43 Heat Stabiliser	Heat
44 Flame Retardant	Flame
45 Lubricant	Lub
46 Plasticiser	Plast
17 Glass Fibre	GF
18 Coupled Glass Fibre	CGF
19 Glass Sphere	GSph
20 Mineral	Min
21 Aramid Fibre	Aramid
22 Carbon Fibre	CarFib
23 Carbon Black	CarBlk
24 Calcium Carbonate	CaCO3
25 Talcum	Talcum
26 Mica	Mica
27 <=20% Filler	20%
28 25% Filler	25%
29 30% Filler	30%
30 35% Filler	35%
31 40% Filler	40%
32 > 50% Filler	> 50%

2 PROPERTY FEATURES

05 Easy Flow	Easy
39 Transparent	Transp
40 Translucent	Transl
16 High Gloss	Gloss
08 Scratch Resistant	Scratc
07 Low Wear	Low
14 Electroplatable	Electr
42 Nucleated	Nuc
11 Antistatic	Antist
06 E.S.C.R.	ESCR

Appendix B : The CAPS parameter list.

10 Microwave Rst	Micro
09 Conductive	Conduc
13 Food Approved	Food
2 MISCELLANEOUS FEATURES	
33 Homopolymer	Homo
34 Copolymer	CoPol
35 Elastomer	Elast
36 Blend	Blend
37 Unfilled	UnFill
38 Filled	Filled
15 Coloured	Colour
47 Development	Devel
3 FAMILIES	
01 POLYETHYLENE	PE
02 ETHYLENE VINYL ACETATE	EVA
03 POLYPROPYLENE	PP
04 POLYMETHYPENTENE	PMP
05 POLYVINYLCHLORIDE	PVC
06 POLYSTYRENE	PS
07 ACRYLONITRILE BUTADIENE STYRENE	ABS
08 ACRYLESTER STYRENE ACRYLONITRILE	ASA
09 STYRENE ACRYLONITRILE	SAN
10 STYRENE MALEIC ANHYDRIDE	SMA
11 POLYMETHYLMETHACRYLATE	PMMA
12 CELLULOSICS	Cell
13 POLYARYLATES	PAryl
14 POLYIMIDES	PI
15 POLYOXYMETHYLENE	POM
16 POLYCARBONATE	PC
17 POLYETHYLENE TEREPHTHALATE	PET
18 POLYBUTYLENE TEREPHTHALATE	PBT
19 LIQUID CRYSTAL POLYMERS	LCP
20 POLYETHER ETHER KETONE	PEEK
21 POLYSULPHONE	PSU
22 POLYETHERSULPHONE	PES
23 POLYPHENYLENEOXIDE	PPO
24 POLYAMIDE 4.6	PA4.6
25 POLYAMIDE 6	PA6
26 POLYAMIDE 6.6	PA6.6
27 POLYAMIDE 6.10	PA6.10
28 POLYAMIDE 6.12	PA6.12
29 POLYAMIDE 11	PA11
30 POLYAMIDE 12	PA12
31 SPECIAL-POLYAMIDE	PA
32 POLYPHENYLENESULPHIDE	PPS
33 THERMOPLASTIC POLYURETHANE	T-PUR
34 FLUOROPOLYMERS	Fluoro
35 PC + ABS BLEND	PC/ABS
36 PC + PBT BLEND	PC/PBT
37 PPO + PA BLEND	PPO/PA
38 PPE + PA BLEND	PPE/PA
39 PP + EPDM BLEND	PP/EPD
40 PP + NBR BLEND	PP/NBR

Appendix B : The CAPS parameter list.

41 STYRENE BUTADIENE STYRENE	SBS
42 STYRENE ETHYLENE B.S.	SEBS
43 POLYESTER ELASTOMER	TPE
44 POLY ETHER BLOCK AMIDE	PEBA

4 PRODUCERS

01 AKZO	AKZO
02 AMOCO	AMOCO
63 APPRYL	APPRYL
03 A.SCHULMAN	SCHUL
04 ASEA Compound	ASEA
05 ATOCHEM	ATO
06 BASF	BASF
07 BAYER	BAYER
08 BERGMANN	BERGM
09 BIP	BIP
10 BORG-WARNER	BORG
11 BP	BP
12 CABOT	CABOT
61 CHI MEI	CHIMEI
16 CIBA-GEIGY	CIBA
17 CONTINENTAL	CONT
60 COURTAULDS	COURT
18 DAVATHANE	DAV
19 DEGUSSA	DEG
20 DOW	DOW
21 DR ILLING	ILLING
22 DSM	DSM
23 DUTRAL SPA	DUTRAL
24 DU PONT	DUPONT
26 EASTMAN KODAK	KODAK
27 EMS	EMS
28 ENICHEM	ENI
29 EXXON	EXXON
30 FERRO	FERRO
31 GEN. ELECTRIC	GE
32 GOODRICH	GOOD
33 HIMONT	HIMONT
34 HOECHST	HOCHST
14 HST.CELANESE	H.CEL
35 HULS	HULS
25 HULS TROISDORF	HULS T
62 HUNTSMAN	HUNTS
36 ICI	ICI
37 JACKDAW	JACK
38 KRAIBURG	KRAIB
39 LNP	LNP
40 MICROPOL	MICRO
41 MITSUBISHI	MISHI
42 MITSUI	MISUI
43 MONTEPIDE SPA	MONT
44 MONSANTO	MONSAN
45 NESTE	NESTE
13 NORSOLOR (CDF)	NORSOL

Appendix B : The CAPS parameter list.

59	PERRITE	PERR
46	PERSTORP ADD.	PERST
14	PETRO DANUBIA	DANUB
47	PHILLIPS PET.	PHILL
48	POLYPENCO	POLY
49	RHONE POULENC	RHONE
50	ROHM	ROHM
51	ROHM & HAAS	R&H
52	SABIC	SABIC
53	SHELL	SHELL
54	SNIA	SNIA
55	SOLVAY	SOLVAY
56	STATOIL	STAT
57	TUBIZE	TUBIZE
58	WILSON FIBER.	WILSON

Appendix C

An Evaluation of

Arity Prolog

Appendix C. An Evaluation of Arity Prolog.

1.1. Introduction.

This Appendix assesses the Arity Prolog Programming Language and Programming environment with a view to giving an insight into the Programming Language behind the Advisor shell, while at the same time evaluating the Arity Prolog Product.

Arity Prolog provides both an excellent program developmental environment for Prolog as well as providing compiling facilities to create stand alone applications. It uses standard predicates as defined in Clocksin & Mellish [CM87] as well incorporating some enhancement features. The interpreter program and applications developed in Arity consist of two parts. Firstly there is the **compiled code**. This is static code which cannot be changed during the execution of the program. The compiled code is in the form of an executable file. This contains all the compiled predicates. The second component of the system is the **program database** which is used to store data in the form of Prolog clauses and terms which can be manipulated by the compiled predicates or by other predicates present in the program database (using database management predicates as outlined in Chapter 4. of [API]).

In discussing Arity Prolog [1] I will first look at the features which make the Interpreter environment such an attractive environment to work in. I will then go on to discuss some of the enhancement features which the Arity Product possesses, mentioning some of the useful built in predicates which are not standard Prolog, but which make the language more accessible to those who have programmed in more conventional languages

[1]Arity Prolog is a trade mark of the Arity Corporation, 30 Domino Drive Concord, Massachusetts.

such as C or Pascal. I will then examine the process of moving from a working program in the interpreter environment towards creating a stand alone application. I will finish the appendix by discussing some of the problems which I encountered while using Arity Prolog Version 5.0.

1.2. The Arity Prolog Interpreter.

The Arity Prolog interpreter is used to develop and check Prolog Programs. It is a menu driven system which provides a suite of useful functions to perform this task. It also utilises dialog boxes to provide a clear and easy to use user interface which can be called using a combination of ALT + Highlighted characters, together with the arrow keys and return key. When you enter the system you are presented with the Main Window which has a menu from which all functions may be invoked. The **Main Window** can be thought of as the Users' interface to the program database. All programs are consulted into this portion of the interpreter and questions (goals to be solved) to the system are initiated in this window.

The Arity Interpreter possesses a **virtual Program database** so that if RAM (main memory) becomes full then the system starts to swap pages between RAM and the disk. This feature while it does affect the performance of the system is essential for loading and checking large programs.

Most of the functions present on the **popdown menus** in the interpreter (those with three dots after them) give rise to what are termed **dialog boxes**. Dialog boxes also form the main part of the user interface in the Advisor shell and dialog box management from a programmers viewpoint is discussed in Section 1.5. of this appendix. As an example of how a dialog box function works we can take the Consult command from the File popdown menu. This command loads a Prolog file directly from disk into the program database and checks it's syntax while it is doing so. When you choose this command a dialog box (See Fig 1.2a) appears presenting all the files in the current directory with extension '.ari'. To choose the file to be consulted you can.....

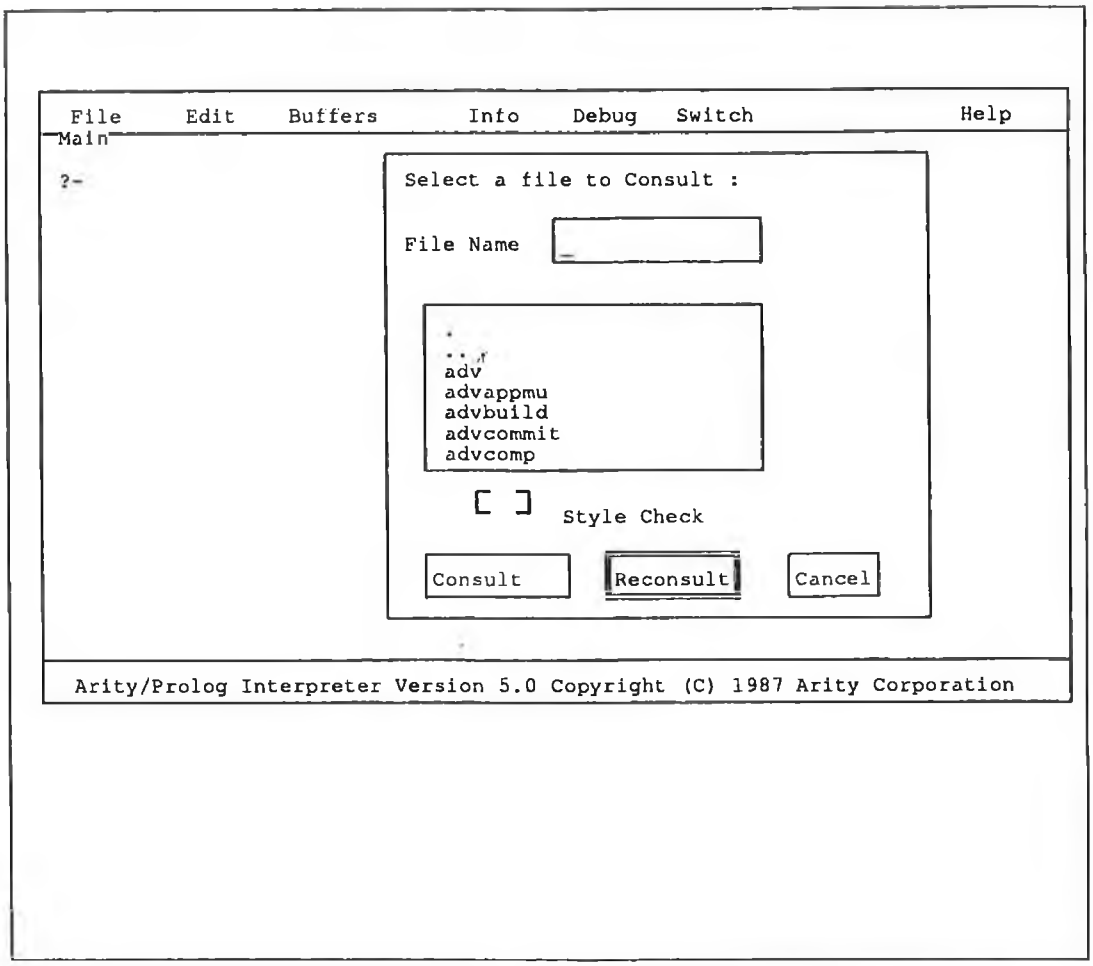

Fig 1.2a. The consult dialog box.

1. Either type in the filename directly as you are currently in an edit field or...

2. Press the Tab key to enter the choice list box and use the space bar to choose one or more files to be reconsulted into the program database.

3. Pressing Enter will reconsult [2] and check the files into the Program database as the Consult push button is in focus.

The Arity interpreter also possesses an excellent **editor** which allows you to edit

[2] Consult means that clauses are added to the program database regardless of whether they reside there already or not. Reconsult replaces clauses with matching heads with the newly added clauses thus avoiding duplicates.

up to nine files at the same time (this is in keeping with the fact that modular development is encouraged in Arity Prolog) and also allows you to switch easily between each of the edit buffers using the **Buffers** popdown menu or associated function keys. Sensible use of the **function keys** provides the user of the interpreter with a quick alternative to using the popdown menus for the more important and heavily used functions in the system. The edit buffers are presented in the form of edit box controls and as such a full range of editing functions may be used in each buffer, (including cut, copy, paste, find, replace etc.) some of which have dialog boxes associated with them to set certain options. Code can be easily transferred from buffer to buffer thus allowing parts of one developed and tested program to be incorporated in another new program. In addition Prolog code in the edit buffers can be consulted or reconsulted into the program database and checked for syntactic errors. The user can toggle between the editor and the Main Window with ease (Using the F8 function key) and test the code he/she is currently working, by typing goals to be solved while in the main window.

For the purpose of logically testing programs the interpreter has a sophisticated **debugger** which allows you to set **spy points** and control the level of tracing you wish to perform. Spy points are set using the spy option on the debugger menu. When this option is chosen a dialog box appears showing a list of all the user defined predicates currently loaded in the program database. You use the space bar to choose the predicates on which you wish to spy. The spy points allow you to selectively **trace** predicates which you suspect to be faulty. Once a spy point is set and the debugger is turned on (by choosing the trace option on debug popdown menu), pressing the letter 'l' (leap to next spy point) will cause the program to execute normally until the predicate which is to be spied on is invoked. When the debugger is in operation a popup window displays the goal currently being executed and also the state of that goal (i.e. wether it is being called, it has failed, it is being redone, or it has succeeded and is being exited). You can specify the level of tracing you desire before enabling the debugger using any combination of the CALL,

REDO, FAIL, EXIT [3] options which appears in the debug menu. You can also affect the action of the debugger while it is executing using one letter commands (such as 'l' for leap mentioned above) which can be displayed together with a terse explanation of their meaning by pressing 'h' while in the debugger Window.

The debugger can also be called from within a predicate using the **trace predicate** (the notrace predicate turns it off). This feature is useful for tracing the execution of large programs. When the debugger is turned on using the debug menu option 'trace' it immediately uses up local stack space until program termination. When a large program is executing it may need a large amount of local stack space which the debugger is using. This will cause the program to halt and report an 'out of local stack space' error. Using inline calls to the debugger only uses the stack space starting from the point at which the call to trace is made and continues until the **notrace predicate** is encountered, at which time the stack space is restored for program use once more.

Arity also provides a useful **on-line help** facility which can be invoked by pressing F1 or ALT + H. It provides help on topics such as predicate syntax, errors editor commands, and the debugger commands and can be called from anywere in the interpreter. The Help box is presented in the form of a Edit Box control. This fact allows you to copy the predicate format from the Help Box and paste it in one of the edit buffers to ensure that you have the right format and syntax if you so wish. You can also build up your own help files if you wish, which could record some of the less well documented features of the language or perhaps to explain how to use some evaluable predicates which you may have added to the system. These too can be viewed using the help dialog box and help edit box.

---

[3] This idea of four possible states for a goal in the context of debugging has been used elsewhere notably in [CM87].

Arity allows you to add your own defined predicates to those already provided in the interpreter [4]. Once added they will be as fast as and treated the same as any standard built in predicate. Thus you can expand the capabilities of the interpreter and customise it to your own particular specification. These added predicates may be written in Prolog or indeed in some other language (C, Pascal or Assembler).

[4] Those predicates which are built into the Arity interpreter and are also contained in the Arity library file are called Evaluable Predicates.

1.3. Language Enhancement features.

Arity Prolog provides a number of useful predicates which facilitate programmers who are more familiar with programming in more conventional languages. These were found to be most useful as they allow programmers to ease themselves into the Prolog style of programming which is very different to conventional programming languages.

1. repeat fail loops.

The repeat predicate has been added to Arity Prolog as an evaluable predicate in order to facilitate some form of looping structure. The repeat predicate is defined as...

```
repeat.
repeat :- repeat.
```

and as such will always succeed. The following predicate demonstrates the use of repeat.....

```
test1 :- repeat,read(X),write(X),nl,X=stop.
```

The way the test1 predicate works is as follows. Firstly it finds the 'repeat.' fact it then read X and writes X and a newline (carriage return). The predicate then checks to see if the value which has been assigned to X unifies with the atom 'stop'. If it does then the predicate succeeds and returns 'yes'. However if X is not stop then Prolog backtracks to try and resatisfy the repeat statement which causes the second repeat clause to be tried and as this is recursive the search starts from the start of the database again and it again finds the fact 'repeat.' and begins to chain forward again, thus giving a looping effect. This type of loop is known as a repeat and fail loop.

2. ifthen and ifthenelse predicates.

The ifthen and ifthenelse predicates are provided to handle conditions in Prolog programs. The general format is[5] ....

ifthen(+G,+A) If G (the condition) succeeds then term A (the action) is executed. If G fails A is not executed but the ifthen predicate still succeeds. You can compound conditions and actions using ';' or ',' and you can use negation 'not', but you must use brackets wisely to group the goals.

ifthenelse(+G1,+A1,+A2) If G succeeds then term A1 is executed otherwise term A2 is executed. If A2 fails the predicate still succeeds.

The example shown in Fig 1.3a. demonstrates the use of these two predicates mentioned above and also introduces some other important language features.

3. Read and write.

The **read** predicate takes input from standard input (i.e. the keyboard) and unifies it with the supplied variable. The supplied value must end in a full stop and a carriage return. There are many derivatives of the basic read predicate which allow you to read strings, lines and read from files.

The **write** predicate is used to write the contents of variables and constants to the screen. You can write more than one item in a write statement if you separate them by colons ':' . The write predicate also has many derivatives.

---

[5] The plus ('+') symbol is used to denote that the programmer supplies this value. A minus symbol('-') symbol is used to denote a value returned by the system.

Although the read and write predicates are standard, the derivatives of these two predicates are not standard but are essential to developing useful programs (especially those for manipulating files).

```
test2 :-
      [!
      repeat,read(Num1),integer(Num1),  % read number until it's an integer.
      repeat,read(Num2),integer(Num2),
      cls,
      write('First Number' : Num1),
      write('Second Number' : Num2),
      ifthen(Num1 == Num2,(write('The Two Numbers are equal'), nl,
          write(finished)
                              )
            )
      !],
      Num1 \= Num2,          % test to see if numbers are not equal.
      ifthenelse(Num1 > Num2,
                  (write('The First number is greater than or equal to the Second'),
               nl,write('finished')
                  ),
                  (write('The Second number is greater or equal to the First'),
                   nl,write('finished')
                     )
                 ).
test2.
```

Fig 1.3a. The Use of built in Non-standard Arity Prolog predicates.

4. The Snips ([! and !]).

The Snips are a adaption of the cut mechanism used in Prolog to control backtracking. The cut is signified as a '!'. It always succeeds while the inference mechanism is going forward. However if the cut is met while Prolog is backtracking then the whole predicate fails even if there are clauses following the current clause, for example given

```
a :- b,c,!,d.
a :- e,f.
```

If a is called and b or c fail then the second clause of the predicate (a :- e,f.) will be called. However if b and c succeed (the ! always succeeds) and d fails then Prolog attempts to redo the previous goals. But when it meets the !, the whole predicate fails, it does not try the alternative 'a' and returns with the answer "no.".

The Snips work differently to the cut mechanism. The goals that they contain are skipped over during backtracking and the snips do not cause the predicate to fail when they are encountered. They are useful for pruning the depth first search tree constructed by Prolog. Using the following example

```
a :- [! b,c !],d.
a :- e,f.
```

If the goal d fails then Arity Prolog will not attempt to resatisfy the goals b and c but will immediately try the second 'a' clause. This can save time and lead to more efficient searches through the program database.

5. Counters.

Sometimes it is useful to maintain numeric values over a number of predicates or to keep numeric values even though backtracking occurs. For this reason Arity Prolog has 31 special counters (numbered from 0 to 30) built in which can be manipulated by the following predicates:

ctr_set(Ctrno,N)	Sets a counter Ctrno to the number N you desire.
ctr_dec(Ctrno,N)	Decrements a counter and returns the counter's previous value.
ctr_inc(Ctrno,N)	Increment the counter and returns the counter's previous value.
ctr_is(Ctrno,N)	Returns the current value of a counter.

Counters are especially useful in repeat and fail loops were ordinary variables would be uninstantiated during backtracking and avoiding this uninstantiation action would be cumbersome.

```
do_X_times(0).
do_X_times(X) :-
   ctr_set(0,1),
   repeat,
   ctr_inc(0,Y),
   write(Y),
```

Fig 1.3b. Example of the use of Arity Prolog counters.

6. The Case statement.

The case statement which is found in most programming languages allows the selection of one of a number of choices with an associated action. This could be replaced in standard Prolog by including the condition as the first goal in a series of clauses which would have the same head. It has the format...

```
case([ Condition1 -> Action1,
       Condition2 -> Action2,
          .          .
          .          .
       ConditionN -> ActionN,|
       Default
    ]).
```

1.4. Manipulating the Program database (PD).

The predicates for manipulating the PD are divided into predicates that deal with clauses and those which manipulate database items or terms. Clauses are used to record Prolog facts and rules. Terms are used to record information in any form in the program database. Together the two give the power and flexibility of the Prolog programming language while providing the means to maintain and manipulate a database.

Manipulating Clauses.

To add a clause to the database you use the **assert** predicate. The assert predicate has two similar forms **assertz** and **asserta**. These predicates allow you to alter the Prolog program as it is being executed in the program database, by adding new rules or facts. The asserta predicate adds the clause to the beginning of a chain of clauses with the same functor name if they exist otherwise it places it in an arbitrary position in the PD. The assert and assertz predicates both add clauses to the end of a list of clauses with the same functor and number of arguments. The opposite of this predicate is the **retract** predicate. The retract predicate deletes one clause at a time form the PD starting at the last clause in a list of clauses if the argument values are not supplied. If an argument is provided then it will be matched before it can be deleted from the database. In addition to retract Arity Prolog also provides a predicate called **abolish** which can be used to retract all the clauses with a specified name and arity (number of arguments).

Manipulating terms.

In addition to predicates for manipulating clauses Prolog also provides predicates for manipulating what are called **terms.** These predicates work at a lower level than those mentioned above. They allow the PD to be treated as an ordinary database rather than program database. Terms are stored as doubly linked lists in the program database under a certain key. Terms can be any valid Prolog data type or structure (strings, atoms, integers, lists, etc.). Each term in the chain has a unique database reference number (DRN). The key name appears to be stored somewere in the system (hidden to the user) together with a reference number to the list of terms associated with this key. The predicate **key** is used to return this DRN. This DRN points to both the first and last reference number (pointer) in the list of terms thus allowing the list to be processed in a forward or backward direction. Certain predicates are provided to allow you to chain through the list. The **nref** predicate is used to get the DRN of the next item in the list. The **pref** predicate is used to get the previous DRN and chain backwards through the list. The diagrammatical representation of this can be seen in Fig 1.4a. The **instance** predicate is used to return the actual data that the reference number points to. Clauses too can be considered as terms were the key is the functor name and arity of the clauses.

Terms are added to the PD using record, recorda, and recordz (the operations of which are logically the same as the ones for manipulating clauses). These three predicates have a common format which is

record(+Key,+Term,-Ref).

The **record_after** predicate is used to insert a term after a given reference number in a list of terms. In order to use it you must find the reference number of the preceeding term using one of the chaining predicates mentioned above. The format for the record_after

predicate is

record_after(+PrevRef,+Term,-Ref).

There is also a precdicate **replace** predicate which allows you to change the contents of a node pointed to by a DRN.

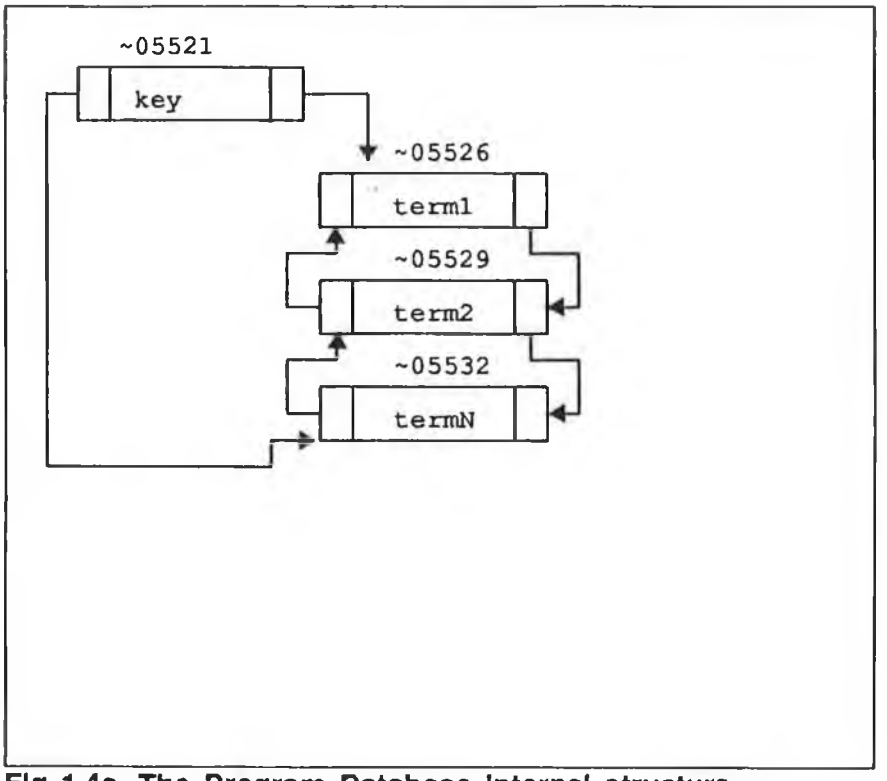

Fig 1.4a. The Program Database Internal structure.

To delete items from the database erase predicates have been provided which allow the deletion of individual items using **erase(+Ref)** or deleteion of all items stored under a given key using **eraseall(+Key)**.

These database manipulation features are non-standard features which were not

found in other popular versions of Prolog[6]. In addition to the predicates mentioned above the Arity Product allows you to organise and structure the database as the need arises. This is done using the concept of worlds and also using predicates which allow you to index the database.

Arity Prolog allows you to logically divide a program database into what are termed **worlds**. A world can be classified as a code world or a data world. The current **code world** is the world in which the interpreter searches for clauses. The predicates such as assert (and all it's derivatives), abolish, call and retract operate in the current code world. The current **data world** is the world in which data manipulation occurs. The built in predicates such as record (and all it's derivatives) and the erase predicate work in this world. One world can be both the data world and the code world and this is the case in the interpreter were the default world is called **main**. Arity provides predicates to create, destroy and move between these worlds. Worlds are rarely used and indeed were not used in the course of developing Advisor. The example that is quoted in Chapter 4 Section 5 of [AP1] on the use of worlds is quite weak and indeed they even say themselves "In practice, you never have to use worlds or the world management predicates..."

Arity Prolog also allows you to index the data in the database using both **hasing** and **btree** techniques. But these techniques are only useful for large numbers of similar items that can be classified in a similar manner, such as student records stored as database facts see Fig 1.4b. Arity Prolog provides special predicates for creating and manipulating both Btrees and hash tables. These features were not used in the development of the Advisor shell as there was no need for them.

---

[6] IBM Prolog, Turbo Prolog, Prolog86, Smalltalk\v5 Prolog did not have these features.

```
student(ralph,01,93,270).
student(susan,24,75,250).
student(adam,12,100,288).
student(linda,05,53,188).
student(dean,33,0,160).
    .     .     .
    .     .     .
    .     .     .
```

recordb(+Tree_name,+Sort_key,+Term) would be used to record the data in the b-tree. e.g.

```
recordb(student,ralph,student(ralph,01,93,270))).
```

The b-trees have default split sizes which can be overwritten.

Fig 1.4b. The student B-tree.

Finally Arity Prolog allows you to interface with other programming languages such as C, Pascal and Assembler. But the documentation on these features is erroneous which makes them unreliable and as such this feature was not used in the Advisor system. A discussion on how to actually get this interface working can be found in Section 1.8.

All the enhancement features mentioned above make Arity Prolog more powerful and easy to use compared to more standard PC versions such as Prolog86, while at the same time not straying too far from the standard as laid down in [CM87], as in the case of Turbo Prolog.

1.5. The dialog box management.

The user interface in Advisor is presented in the form of dialog boxes. The programming carried out which allows dialog boxes to function correctly is discussed here. In order to program the dialog boxes effectively it was necessary to study the dialog box management process in detail.

Programming dialog boxes consist of two sperate phases. Firstly you create the dialog box definitions file which defines how the dialog box appears on the screen. This file must be consulted into the program database prior to invoking the dialog box. It cannot be compiled. Secondly there are the predicates both built into Arity Prolog and those defined by the programmer which perform actions in response to certain actions (usually in the form of key presses) being preformed by the person using the dialog box. These can be consulted into the program database or alternatively compiled.

1.5.1 The definitions file.

A **dialog box editor** is provided with the Arity toolkit which allows you to create dialog boxes on the screen using the arrow keys (to position the chosen controls and to adjust the size of the dialog box and placed controls) and popdown menu functions (to choose the controls to be placed in the dialog box and call built in commands). This dialog box editor produces the code for the predicates necessary to produce these dialog boxes on the screen when consulted into the interpreter program database or a program database of a compiled application and run using the **dialog_run** predicate.

However it necessary to edit this code as it has a few minor errors as pointed out in Section 1.8. Examples of this code can be seen in the program 'advdefns.ari' in Appendix A together with some menu definitions and a small example of the dialog box

definitions code can be seen in Fig 1.5a. This code defines the size, screen attributes and position of the dialog box on the screen and, the positions and type of the controls within the dialog box. It can also provide information which should appear in the controls when they are consulted into the program database and run. In addition to this code the programmer must write his own predicates to customise the actions which are performed by the dialog box as seen in Fig 1.5b below.

1.5.2. The dialog box manager.

The programmer defined clauses always have and arity of two. The first argument is the message that is to be recognised by this clause and the second argument is the name of the dialog box with which the dialog box manager is currently working. The last clause in this list of clauses contains the **default dialog function.** It is the purpose of the default dialog function (def_dialog_fn) to handle messages which the programmer defined functions have not been programmed to understand [7]. The default dialog box function and the programmer defined functions together form the **dialog box manager** which can be seen Fig 1.5c. together with the message passing architecture. The dialog box manager processes messages from the user and the dialog box control predicates and affects the result of these messages on the controls in the dialog box on the screen or in some other way (as in the case of the second clause in Fig 1.5b).

[7] The init_dialog and draw_ctrls messages are examples of two of these types of messages. They are sent by the dialog_run predicate when it is used to invoke a dialog box.

```
begin_dialog(filename,filename,(4 , 10),(19 , 60),(120 , 113),116,  popup).
ctrl(text,1,$~File name$,(3 , 3),113,11).
ctrl(efield,1,_,(4 , 3),(113 , 113),29,$$).
ctrl(edit_region,0,_,(1 , 1),(2 , 59),(177 , 113),r,fkey).
ctrl(push,1,$~Ok$,(9 , 4),(113 , 113),default(ok)).
ctrl(push,1,$~Cancel$,(9 , 22),(113 , 113),cancel).
end_dialog(filename).
```

Fig 1.5a. Dialog box code.

```
sfl(command(_,ok),filename) :-
    [! send_control_msg(ef_set_text(Oldtext,Oldtext),2,filename),
    send_control_msg(update,2,filename),
    assert(filename(Oldtext)) !],
    fail.

sfl(command(_,cancel),filename) :-
    [! assert(abandon) !],fail.


sfl(Msg,filename) :-
    def_dialog_fn(Msg,filename).
```

Fig 1.5b. The user defined predicates for filename.

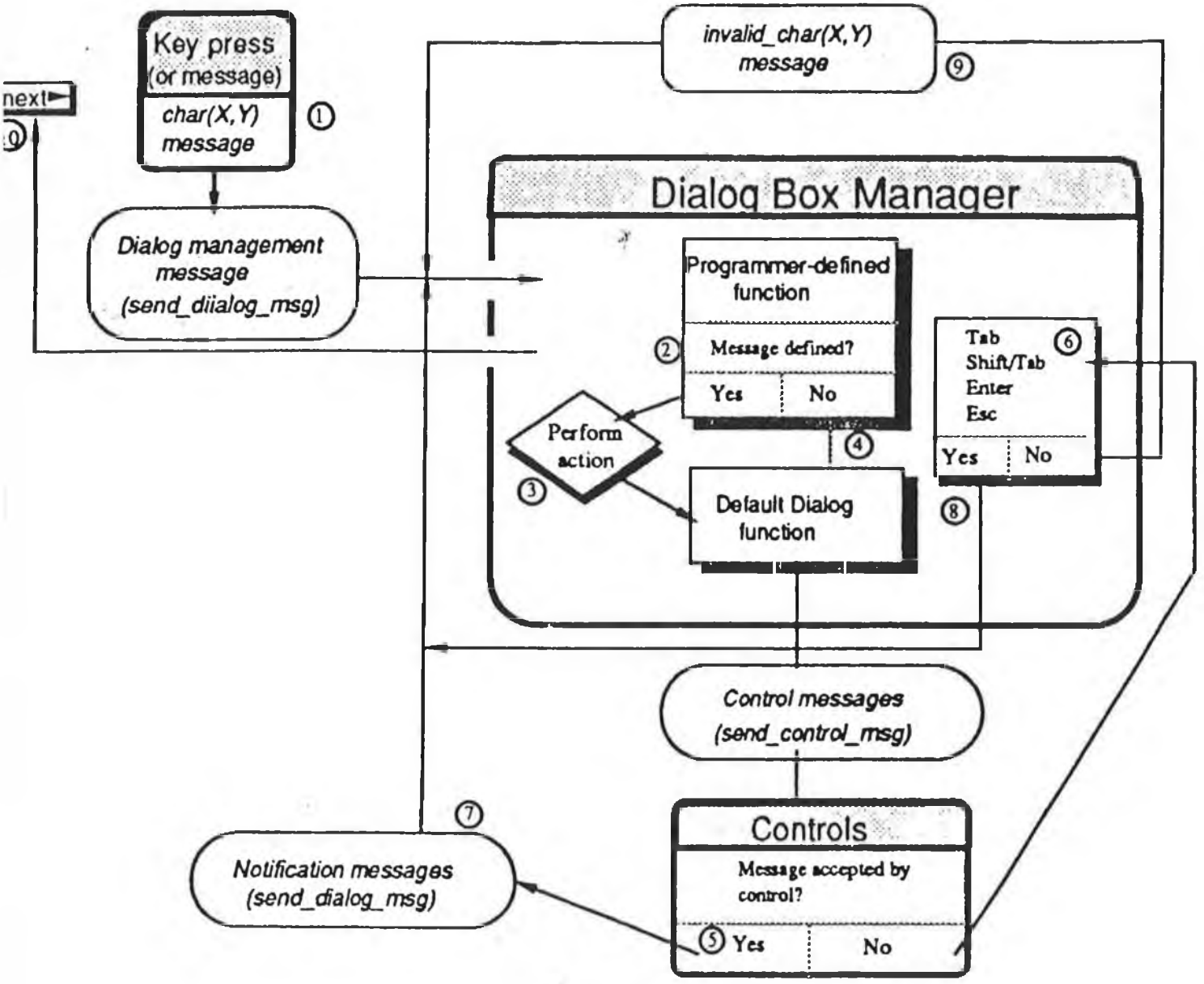

Fig. 1.5c. The Dialog Box Manager.

1.5.3. Activating dialog boxes.

Dialog boxes are called using the **dialog_run** evaluable predicate. The dialog_run predicate has two forms **dialog_run(Dbox_name)** and **dialog_run(Dbox_name,User_predicate)**. The first form is useful for viewing what the dialog box looks like. The second form is the form used for actually processing information gathered by the dialog box.

The **dialog_run(Dbox_name)** form runs the dialog box on it's own using the control definitions (see fig 1.5a.) which must be previously loaded into the program database. It uses the evaluable predicate **def_dialog_fn** to process all the messages to be passed to the dialog box controls. The def_dialog_fn in turn calls various other **control predicates** (a full list of these predicates is included in 'advctrld.ari' in Appendix C) which are defined for each type of control. These control predicates actually affect the controls that are displayed in the dialog box on the screen. The def_dialog_fn finds out which control is were by using the definitions contained between the **begin_dialog** clause and the **end_dialog** (see Fig 1.5a.) clause in the program database and constantly keeps track of which control is currently in focus. All this is hidden to the user but can be seen through use of the debugger provided with the Arity interpreter. The calling process can be seen clearly in Fig 1.5d.

The second form of the dialog_run predicate specifies the **user defined predicate** (see Fig 1.5b) which will be use in conjunction with the def_dialog_fn. The first parameter of a user defined clause states the message which that particular clause will act on receiving. The second parameter states which dialog box the clause is working with. The order of the user defined clauses insures that they will be tried before the def_dialog_fn.

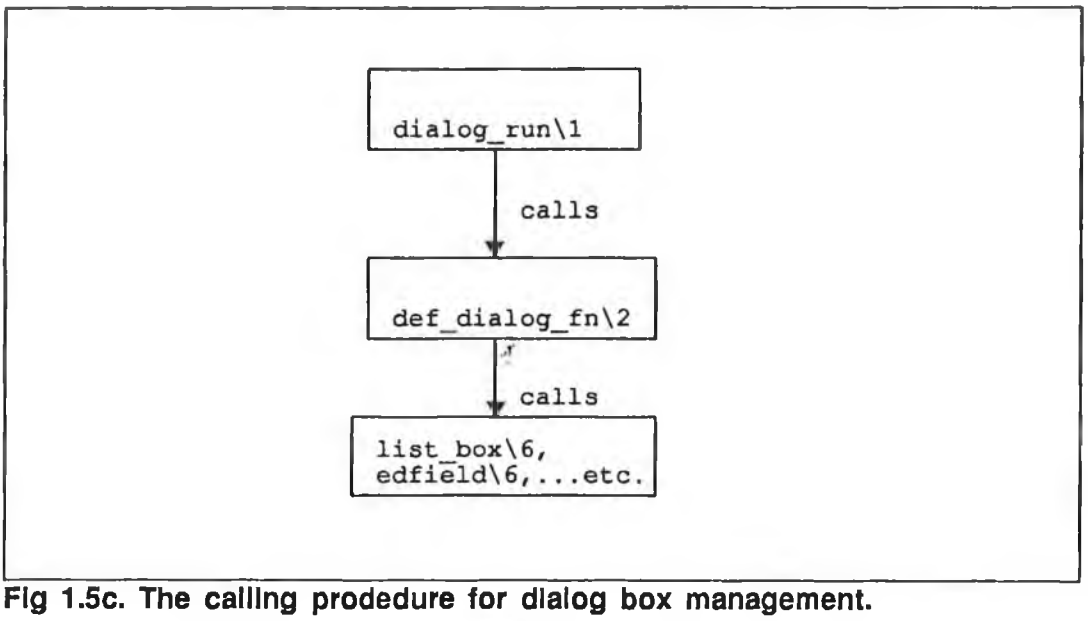

Fig 1.5c. The calling prodedure for dialog box management.

This is why the def_dialog_fn goal is included in the catch all clause [8] which must appear as the last clause in the list of clauses which make up the user defined predicate. The **send_control_msg** predicates which are used by the user defined predicate to manipulate controls on the screen also call the control predicates mentioned above (e.g. list_box/6,etc.). The calling sequence for this can be seen clearly in Fig 1.5d below.

The user defined predicate as seen in fig 1.5b. would be called thus

dialog_run(filename,sfl).

[8] Catch all is a term used in Prolog to refer to a clause which will always succeed, stopping the overall predicate from failing.

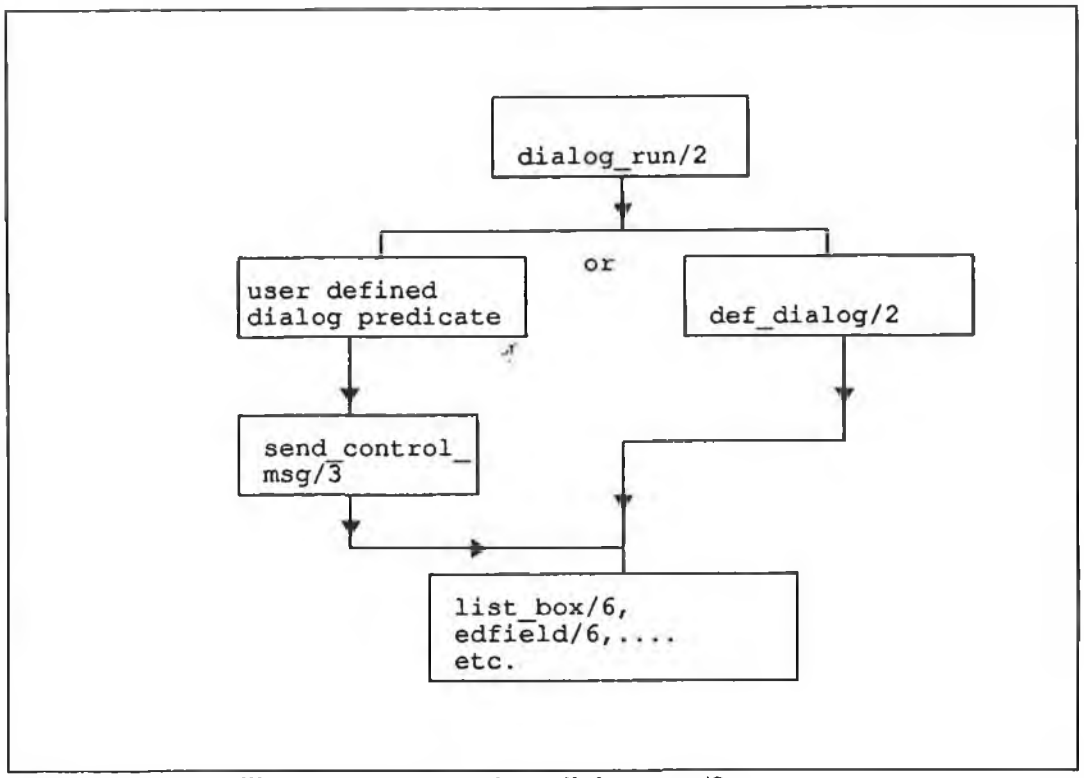

Fig 1.5d. The calling sequence using dialog_run/2.

1.5.4. Message passing and processing information.

A dialog box processes information typed in by the user through the use of message passing. Every time a user presses a key while in the dialog box, the system passes a message. It is through the use of messages that the programmer programs what actions should be performed after a particular control has been selected or altered. The controls only provide a means of gathering information they do not process it. It is the task of the user defined dialog box predicate to read information contained in controls and to display information in controls. This is done using the message passing predicates mentioned below. There are four classes of messages:

1. Dialog box management messages. These messages are sent to the dialog box manager

through the **send_dialog_msg** predicate but this feature is rarely used by the programmer as most of the messages associated with it are usually handled by the default dialog function mentioned above.

2. Control messages are sent to the controls through the use of the **send_control_msg** predicate and there are messages for manipulating all the nine types of controls in various and useful ways. These messages can be used to gather information from the control or send information to be displayed by the control. When sending these messages from a programmer defined predicate, you must specify the control number as well as the message to be passed. Controls are numbered from 1 to N in the order in which they are found in the definitions file see Fig 1.5b. above. The general format for a send_control_msg is:

send_control_msg(Msg,Control_number,Dialog_box_name).

3. Notification messages are sent by some controls to the dialog box manager to indicate that some change has occurred to themselves. These messages are usually handled by the default dialog function.

4. Invalid character messages are sent by controls if they cannot understand a message sent to them and it is not one of the predefined key presses for changing the control in focus. These too are usually handled by the default dialog function.

There is a whole library of messages which may be passed to controls associated with each of these four categories of messages. A full list of these can be found in [AP1] Chapter 12.

1.6. Menu Management.

Menus are treated in a similar fashion to dialog boxes in that both operate on a message passing architecture. The menu manager also uses a definitions file to store information which is used to define how the menu will appear on the screen.

An example of a menu definitions file can be seen in Fig 1.6a and all the definition used to define menus in the Advisor system can be seen in 'advdefns.ari' which is contained in Appendix A.

```
begin_menu(mainm,79,colors((113,31),(113,31),(123,27),(116,116))).

item($~Application Descript$,app_run).
item($~Rule Editor$,rulerun).
item($~Load$,
        [item($~Meta Rules$,lmr),
         item($~Rule File$,lrf),
         item($~Compiled rules$,lcr),
         item($~Explanation File$,lex),
         item($Change ~Index$,insert_index)]).
item($~Trace$,
        [item($Trace O~N$,ton),
         item($Trace O~FF$,toff)]).
item($~Go$,mgo).
item($~Help$,[item($~System Help$,mhelp),
              item($System ~Updates$,sysupdate)]).
item($~Shell$,shl).
item($~Quit$,mquit).

end_menu(mainm).
```

Fig 1.6a. A Menu definitions file.

The definitions for the items to appear on the menu bar are contained between the **begin_menu** clause and the **end_menu** clause. The begin menu clause also defines the size and screen attributes which the menu bar possesses. The items or options which will appear

on the menu bar are defined using the **item** clause. The first argument of the item clause defines the string which will appear on the screen. The '~' character is used to indicate the letter which is to be highlighted and used as a **fast key** for that particular option. The second parameter in an item clause is the predicate which will be called when that particular option is selected. This predicate can be a user defined predicate or an evaluable predicate. If an item has sub-items (which will appear in popdown mode when that particular item is selected) they are represented as a list of item clauses, which are treated similarly to ordinary item clauses.

Messages are passed to menus (as defined in the definitions file) through the use of the **send_menu_msg** predicate. To activate the menu as defined in Fig 1.6. one would send the message

send_menu_msg(activate(mainm(0,0)),ReturnValue)

were the **ReturnValue** argument is the name of the predicate associated with the selected menu item. This predicate can be called by just stating the ReturValue as a goal to be solved. If the goal fails then control just returns to the menu bar. There are other messages which can be sent to a menu to change the states of items, and to check the states of items but these proved to be unpredictable and in some cases failed to work. A full description of all menu management messages can be found in chapter 13 of [AP1].

1.7. Compiling Arity Prolog Programs.

It was shown in section 1.1. that the Arity Interpreter consists of two separate parts. The first being the executable code and the second being the program database. This is also true of all programs produced using the Arity Prolog Compiler. While the interpreter provides an excellent program development environment it does suffer from the fact that code consulted and executed in the program database is quite slow. Compiled programs however run up to ten times faster than there interpreted equivalent. When you compile a Prolog Program, the database that was available from the interpreter is still available to the compiled application. Every Prolog application has it's own database file which is created by the compiler at the same time that the compiler creates an object file. As the compiler goes through the Prolog source file it puts the executable code in the object module and puts the program's atoms into the database file. The remaining database space can be used by the program at runtime to store user information, as in a database application, or store clauses which form Prolog code (Programs). The process of moving from an interpreted form of a program to a compiled application involves:

1. Identifying the predicates which use predicates in other files or modules.
2. Adding in all the compiler directives which are used by the compiler to check and produce current code.
3. Compiling the program modules using the Arity Prolog Compiler to produce object code and a program database.
4. Linking the appropriate object modules and supplied libraries using the standard Microsoft linker or the Prolog Linker Plink86 to produce an executable program.

Arity Prolog supports modular programming. This is advantageous in that you need only define a predicate once in one file and you can call it from another predicate in an other file. This fact presents no problem in the early stages of development when you are using the interpreter, as you just consult all the files into the program database and all the predicates are globally available to each other. However when compiling files you must identify which predicates use other predicates defined in other files as this has an effect on which compiler directives should appear in the files.

1.7.1. Important Compiler directives.

All compiler directives appear at the top of each file to be compiled and linked to form one executable application. Examples of compiler directives used in the Advisor Program can be seen in the programs in Appendix A.

The **extrn** compiler directive is used to declare that the definition of a predicate, which is used in the current file, is to be found in one of the files to be linked in at a later stage. This same predicate must be declared as being **public** in the file in which it is defined. An example of the two definitions can be seen in Fig 1.7a.

Declared public in the file in which it is defined.

```
:- public reverse_text/3.

reverse_text(Y,X,Text) :-
  name(Text,L),
  length(L,N),
  move(Y,X),
  wa(N,112),
  write(Text).
```

Declared extrn in the file in which it is used but not defined.

```
:- extrn reverse_text/3.

write_name(A,B,Text) :-
    tmove(10,10),
    reverse_text(a,B,Text).
```

Fig 1.7a. Use of the extrn and public directives.

The **visible** directive is used to declare a predicate that is to be called by clauses

which are present in the program database as apposed to being called from other compiled predicates. This is because when a clause in the program database calls another clause it looks for that clause in the program database and not in the compiled section of the program. Declaring the clause to be called as visible tells the compiler to produce code that will allow the calling clause to find the compiled clause in the compiled section of the program. All programmer defined dialog box management predicates must be declared visible so that they can communicate with the control manipulation predicates and the control definition predicates that reside in the program database at run time. Also all menu predicates which are associated with menu items must be declared visible so that they can be called when a menu item is selected (as menu definitions are loaded into the program database at run time also). An example of the use of the visible declaration can be seen in Fig 1.7b.

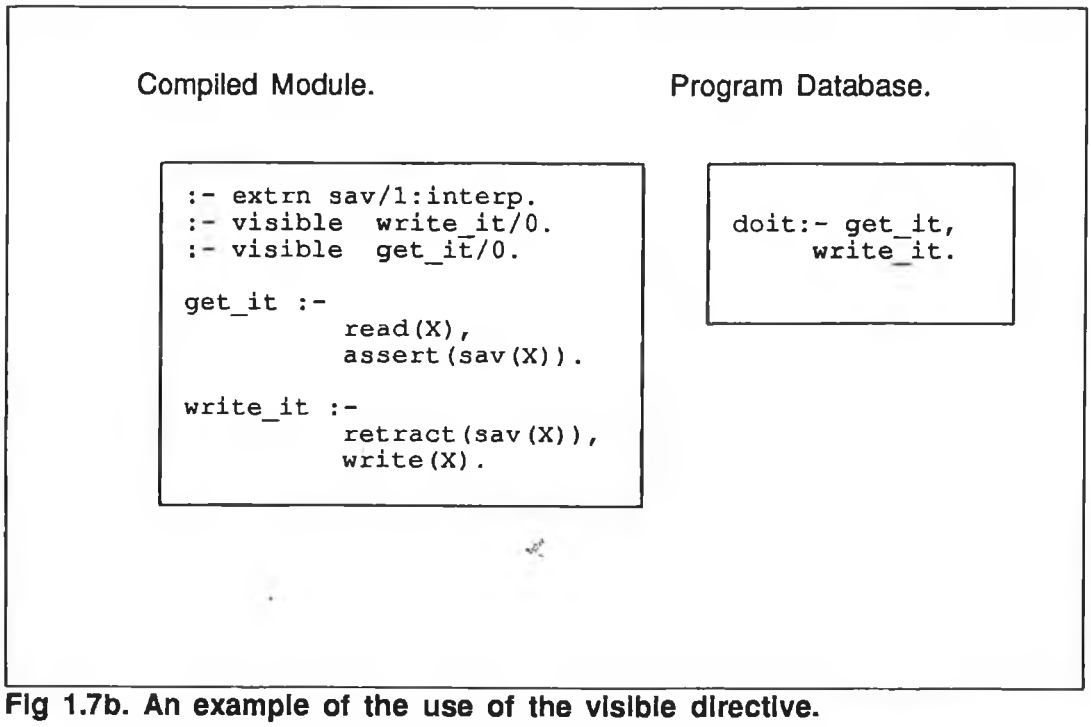

Fig 1.7b. An example of the use of the visible directive.

The **segment** directive is used in cases were a code segment exceeds 64K. It specifies that the contents of the current file when compiled and linked should be placed in the named code segment. When a program is split over more than one code segment it

is necessary to use the clone utility provided with Arity Prolog and also to add certain suffixes to the extrn and public directives described above. The clone utility creates an empty code segment into which you can place the code which has exceeded the 64K limit. To use the clone utility you just type in "clone filename" and the clone program produces a object file called
"filename.obj".

In addition to using the clone utility program and specifying the segment directive you must add the suffix **far** to public and extrn directives of predicates in different code segments so that the linker can produce code that will allow a far call. Predicates in the same code segment do not need this suffix. Examples of this can be seen in Fig 1.7c.

For clauses that are added to the program database during the execution of the program and again called from the same predicate or other predicates the **interp** suffix must be used in addition to the prefix extrn. An Example of this can be seen in the form of the sav clause in Fig 1.7b above. This avoids an external reference error during the linking of the object modules or a compile time error.

The more important compiler directives have been mentioned here as they were used in the process of compiling the Advisor Program. Other compiler directives do exist for tasks such as incorporating other languages with Arity Prolog and making an executable file more compact by only making those evaluable predicates which are used by the program visible (the 'default(invisible) ' directive is used for this purpose together with declaring only the evaluable predicates to be used by the program as visible). All these compiler directives must appear in a strict order which can be seen together with a complete list of the compiler directives available to the programmer in Section 8.3 of [AP2].

Declared public in the file in
which it is defined.

```
:- segment(far).
:- public reverse_text/3:far.

reverse_text(Y,X,Text) :-
  name(Text,L),
  length(L,N),
  move(Y,X),
  wa(N,112),
  write(Text).
```

Declared extrn in the file in
which it is used but not defined.

```
:- extrn reverse_text/3:far.

write_name(A,B,Text) :-
    tmove(10,10),
    reverse_text(a,B,Text).
```

Fig 1.7c. Use of the segment directive and far suffix

In addition to the compiler directives every program that is compiled using the Compiler must have an entry point which is predicate called **main** (similar to the C programming language convention). Execution of the program begins with this predicate. There is also an option to include a **restart** predicate which determines what happens if the user of the program press **Ctrl + Break**. You could use this feature to ensure that the program database is in a safe state or save the current state of the database before exiting the application.

1.8. Some peculiarities with Arity Prolog.

During the course of creating the Advisor shell some peculiarities with Arity Prolog Version 5.0 were discovered which were not covered by the documentation provided with the product. Some of these oddities are here now reported.

It was mentioned earlier that the dialog box editor in the development toolkit could be used to create the dialog boxes on the screen while also writing the control predicates to represent these controls at the same time. It was also mentioned that this code produced by the dialog box editor is faulty in some respects and leads to unpredictable results in other cases. This is due to some erroneous entries in the ctrl predicates which define each control. An example of the incorrect form produced by the dialog box editor and the corrected form as edited by the programmer can be seen in Fig 1.8a.

Firstly the list boxes drawn with the dialog box editor appear without their border being drawn properly. The box label contains the '~' character which is used to denote the highlighted character to be used together with the ALT key to quickly bring that control into focus. It is this character which causes the borders to be drawn incorrectly so a list box label cannot contain a highlighted character [9]. This however can be overcome by having no label with the list box and placing a text control just before the list box definition in the definitions file, which can contain a '~' character. Then if you change the value of the second argument (the tab stop setting) of the text control definition to 0 instead of 1 control will pass to the list box instead of the text control when ALT and the highlighted letter is pressed.

---

[9] This would suggest that the built in ctrl predicate which deals with the list box is not built to deal with this eventuality together with the fact that this character is also used to denote a database reference number.

Faulty Dialog code

```
begin_dialog(test,test,(3 , 4),(22 , 75),(120 , 113),116,stayup).
ctrl(efi eld,1,_4610,(2 , 2),(113 , 113),25,$EditField$).
ctrl(edit_region,1,_4610,(0 , 34),(3 , 54),(113 , _4668),rw,'$text').
ctrl(list_box,1,$~List Box$,(4 , 36),(14 , 69),(113 , 113),radio,(1 , 1),'$list').
ctrl(push,1,$~Push Button$,(15 , 20),(113 , 113),default(null)).
end_dialog(test).
```

Corrected version.

```
begin_dialog(test,test,(3 , 4),(22 , 75),(120 , 113),116,stayup).
ctrl(efield,1,_,(2 , 2),(113 , 113),25,$$).
ctrl(edit_region,1,_,(0 , 34),(3 , 54),(113 , _),rw,edkey).
ctrl(text,0,$~List Box$,(3 , 36),113,11).
ctrl(list_box,1,$$,(4 , 36),(14 , 69),(113 , 113),radio,(1 , 1),lkey).
ctrl(push,1,$~Ok$,(15 , 20),(113 , 113),default(ok)).
end_dialog(test).
```

Fig 1.8a. The dialog Box code corrections.

The second problem with the definitions produced using the dialog box editor results from the fact that the ctrl definition for an edit field takes the underscore character '_' [10] as one of it's arguments. But the dialog box editor places an '_nnnn' symbol (were n is a hex digit between 1 and F) in this argument position. This can lead to unpredictable results as this symbol is used by Arity to signify a memory address which may be already in use and contain a value at the time that the control is brought into focus.

These two problems were encountered during the early stages of Advisors development. During the later stages when creating a stand alone application and compiling the user defined dialog box management predicates some further irregularities were discovered.

[10] The underscore or don't care symbol is used in Prolog to signify that the called predicate should ignore that particular parameter which might be a variable in other circumstances.

Firstly in order to allow the dialog boxes to function correctly in a compiled application it was necessary to make each of the controls used visible (these control predicates are actually hidden to the programmer) so that the **send_control_msg** and **send_dialog_msg** can find them. A file called 'ctrlvis.ari' was provided for this purpose (see Fig 1.8b).

```
%%%
%%%    CTRLVIS.ARI
%%%
%%% Make all dialog box controls and the default dialog function
%%% visible, so that calls to send_control_msg and send_dialog_msg
%%% can find them.
%%%


:- extrn
   list_box/6,
   edit_box/6,
   efield/6,
   push/6,
   radio/6,
   choice/6,
   text/6,
   def_dialog_fn/2.

:- visible
   list_box/6,
   edit_box/6,
   efield/6,
   push/6,
   radio/6,
   choice/6,
   text/6,
   def_dialog_fn/2.

%%%
%%%    END OF CTRLVIS.ARI
%%%
```

Fig 1.8b. The fualty control definitions file

But this was found to be incomplete as it did not contain certain predicates which also needed to be made visible. Also on this point the manual made no mention of the fact that the functions behind the menu options needed to be made visible or indeed that the **send_menu_msg** predicate needed to be declared visible. The correct form of the 'ctrlvis.ari' file can be seen in the file 'advctrld.ari' (also in Fig 1.8c).

```
%%%
%%%    advctrld.ARI
%%%
%%% Make all dialog box controls and the default dialog function
%%% visible, so that calls to send_control_msg and send_dialog_msg
%%% can find them.
%%%


:- extrn
   list_box/6:far,
   edit_box/6:far,
   efield/6:far,
   edit_region/6:far,
   push/6:far,
   radio/6:far,
   choice/6:far,
   text/6:far,
   dialog_run/2:far,
   dialog_run/1:far,
   def_dialog_fn/2:far,
   send_menu_msg/2:far. % added so that menus will work

:- visible
   list_box/6,
   edit_box/6,
   efield/6,
   edit_region/6,
   push/6,
   radio/6,
   choice/6,
   text/6,
   def_dialog_fn/2,
   dialog_run/2,
   dialog_run/1,
   send_menu_msg/2.     % added so that menus will work
```

Fig 1.8c. The Correct control code.

The second and stranger peculiarity with compiled applications and dialog box predicates was to do with compiling user defined dialog box management predicates. While working in the interpreter it did not matter if predicates which were not user defined dialog box management predicates appeared mixed among the user defined dialog box management predicates (see Fig 1.8d). But when these predicates were compiled the dialog boxes to which these user defined dialog box management predicates referred, failed. It was only through much trial and error (and much to the annoyance of the author) that the problem was established. User defined dialog box management predicates must appear consecutively and uninterrupted in the Prolog source file so that the compiler can produce the correct code. The predicates were subsequentially changed so that they appeared consecutively in the source file (see Fig 1.8e for the correct coding of Fig 1.8d).

```
% F3
support(char(0,61),dapplic):-
    ifthenelse(retract(replace(Com)),
     display_mess(Com),display_mess($$)
    ).

display_mess(Str) :-
    send_control_msg(ef_set_text(Oldtext,Str),5,dapplic),
     send_control_msg(update,5,dapplic).

support(Msg,dapplic) :-
    def_dialog_fn(Msg,dapplic).

( adapted from the support predicate in advappmu.ari ).
```

Fig 1.8d. Incorrect User defined dialog box management predicates.

```
% F3
support(char(0,61),dapplic):-
      ifthenelse(retract(replace(Com)),
       display_mess(Com),display_mess($$)
      ).

support(Msg,dapplic) :-
      def_dialog_fn(Msg,dapplic).

display_mess(Str) :-
    send_control_msg(ef_set_text(Oldtext,Str),5,dapplic),
    send_control_msg(update,5,dapplic).
```

Fig 1.8e. The correct form of a user defined dialog box predicate.

Another feature which seemed to be problematic with Arity Prolog was calling C functions from Prolog programs and visa versa, and passing parameters between the two modules. Several people before the current Author tried to get this language interface working but with little success. One of the sample C programs provided (in the Arity Prolog Manual [AP2] Chapter 16) was typed in using QuickC editor. The calling Prolog

Program was typed in using the Arity Interpreter. The compiling and linking procedures[11] laid down were followed strictly but the resulting program was unpredictable and erroneous. The problem appears to be in the values for SUCCESS (defined as 1) and FAILURE (defined as 0) returned to the calling Prolog program (in this particular case Prolog is the dominant language). It appears that the authors of the C interface and the Prolog interface have mixed the return values around[12]. The programs were entered as per the manual (See Fig 1.8f below) and the C function 'setfreq' was called and returned SUCCESS (this implied that the problem was not with the functions for passing data types between Prolog and C). But when the program control returned to the Prolog predicate execution just halted at that line and the second call to the C function 'speaker' was not initiated. By writing a second catch all main predicate and using printf statements in the C function I discovered that even though the correct actions had been taken in the C function, when the value SUCCESS was returned, the Goal (which is the call to 'setfreq') failed thus causing the second main predicate to be called . This lead me to the conclusion that in order to get the interface to work it would be necessary to return the value FAILURE to indicate success, but in addition to this I found it necessary to negate the calling goal in the main predicate. The resulting program executed successfully (See Fig 1.8g).

[11] The Arity Compiler was used to compile the Prolog code. The Microsoft compiler V5.0 was used to compile the C code. The Microsoft Linker V5.0 was used to create the executable code.

[12] C functions return 0 on successful termination.

```
% The incorrect version of the Arity Program to interface with C
:- public main/0.
:- extrn setfreq/1:c('_setfreq').
:- extrn speaker/1:c('_speaker').

main :-
    repeat,
    write('What frequency do you want? '),
    read( X ),
    setfreq( X ),
    speaker( 1 ),
    get0_noecho(_),
    speaker( 0 ),
    fail.
```

```
/* The incorrect C version of the Arity C/Prolog interface */
#include <stdio.h>
#include "apctype.h"

setfreq(hertz)
reftype hertz;

{
        unsigned int divisor,cint;
        int g;
        g = getint_c(hertz,&cint);
        if( g != 0)
                {
                 divisor = 1193180L | cint;
                 outp(0x43, 0xb6);
                 outp(0x42, divisor & 0377);
                 outp(0x42, divisor >> 8);
                 return SUCCESS ;
                }
        else    return FAILURE;
}

speaker(on)
reftype on;
{
        int portval,cone,g;
        g = getint_c(on,&cone);
        if(g != 0)
                {
                 portval = inp(0x61);
                 if(cone)
                        portval |= 03;
                 else
                        portval &=~03;
                 outp(0x61,portval);

                 return SUCCESS;
                }
        else return FAILURE;
}
```

Fig 1.8f The Faulty C/Prolog Interface code.

```
% The correct version of the Arity Program to interface with C
:- public main/0.
:- extrn setfreq/1:c('_setfreq').
:- extrn speaker/1:c('_speaker').
:- visible setfreq/1.
:- visible speaker/1.

% In order to get the interface between C and Prolog working you
must:
% 1. negate the calling predicate
% 2. Return FAILURE instead of SUCCESS from the C function
% If you return SUCCESS then the program completes it's
% execution premiturly

main :-
     write('What frequency do you want? '),
     read( X ),
     not(setfreq( X )),
     not(speaker( 1 )),
     get0(_),
     not(speaker( 0 )).

main :- write('Main failed').

/* The correct C version of the Arity C/Prolog interface */
#include <stdio.h>
#include "apctype.h"

setfreq(hertz)
reftype hertz;

{
          unsigned int divisor,cint;
          int g;
          g = getint_c(hertz,&cint);
          if( g != 0)
                    {
                     divisor = 1193180L | cint;
                     outp(0x43, 0xb6);
                     outp(0x42, divisor & 0377);
                     outp(0x42, divisor >> 8);
                     return(FAILURE) ;
                    }
          else     { printf("GETINT_C FAILED 1 "); return(SUCCESS);}
}

speaker(on)
reftype on;
{
          int portval,cone,g;
          g = getint_c(on,&cone);
          if(g != 0)
                    {
                     portval = inp(0x61);
                     if(cone)
                               portval |= 03;
                     else
                               portval &=~03;
                     outp(0x61,portval);

                     return(FAILURE);
                    }
          else { printf("GETIN_C FAILED");return(SUCCESS);}
}
```

Fig 1.8g. The corrected version of the C/Prolog interface code.