Computer Science and Systems Analysis

Computer Science and Systems Analysis Technical Reports

Miami University *Year* 1993

An Expert System Shell Performing the Generic Task of Hierarchical Classification

Jen Wazel
Miami University, commons-admin@lib.muohio.edu

MIAMI UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE & SYSTEMS ANALYSIS

TECHNICAL REPORT: MU-SEAS-CSA-1993-000

An Expert System Shell Performing the Generic Task of Hierarchical Classification

Jens Wazel

School of Engineering & Applied Science | Oxford, Ohio 45056 | 513-529-5928

An expert system shell performing the generic task of hierarchical classification

FINAL REPORT

Presented in Partial Fulfillment of the Requirements
for the Degree of
Master of Systems Analysis
in the
Graduate School of Miami University

BY

Jens Wazel

Miami University

1993

Reading Committee:

Prof. Weldon Lodwick, Advisor
Prof. Alton F. Sanders
Prof. Mufit Ozden

Table of contents

0. Introduction 4

1. The generic task concept 5

2. Hierarchical classification 8

3. The HICLASS system 9
    - 3.1. Introduction 9
        - 3.1.1. Terminology 9
        - 3.1.2. The concept of "table" 10
        - 3.1.3. Knowledge representation 11
            - 3.1.3.1. Sets as a basic approach 11
            - 3.1.3.2. Preenumerated solutions 12
            - 3.1.3.3. "Don't care" values 13
        - 3.1.4. The hierarchy 14
        - 3.1.5. The control strategy 15
    - 3.2. Comparison of the HICLASS knowledge representation to other representations 16
    - 3.3. Fuzzy borders: From frame-like objects to rules 17
        - 3.3.1. The "classical" hierarchical approach 17
        - 3.3.2. Opening the "classical" structure 18
        - 3.3.3. The combination of the two concepts 19
        - 3.3.4. Introduction of a factbase 20
        - 3.3.5. A Rule-based system 22
    - 3.4. Local control strategies 23
        - 3.4.1. MATCH 24
        - 3.4.2. Left-to-Right 25
        - 3.4.3. Heuristic driven 26
    - 3.5. Class descriptions with different weights 27
        - 3.5.1. MATCH 27
        - 3.5.2. Left-to-right 28
        - 3.5.3. Heuristic driven 29
    - 3.6. Different classes with the same content but different weights 30
    - 3.7. ASKFIRST 30
    - 3.8. One class, one table and multiple children 31
    - 3.9. An answer UNKNOWN 33
        - 3.9.1. MATCH 34
        - 3.9.2. Left-to-right 35
        - 3.9.3. Heuristic driven 36
        - 3.9.4. Global effects 37
    - 3.10. Dealing with uncertainty 41
    - 3.11. Concluding other values 48
    - 3.12. Explaining the reasoning process 50
    - 3.13. Incorporating metaknowledge 52
    - 3.14. Learning 53
    - 3.15. Global attributes 54
    - 3.16. Checking the consistency of the system 54
    - 3.17. HICLASS and the rest of the world 54
    - 3.18. Several paths - which one to follow? 55

3.19. Additional features 56
    3.19.1. Entering initial data 56
    3.19.2. Saving the system state in case of an interruption 56
    3.19.3. Using information from terminated paths 56
    3.19.4. Numerical input 57
3.20. HICLASS - an expert system shell 60

4. The implementation of the HICLASS system 61
  4.1. HIEDIT 61
    4.1.1. FILES screen 62
    4.1.2. DEFINITIONS screen 63
    4.1.3. EXAMPLES screen 66
    4.1.4. SPECIAL screen 67
  4.2. HICLASS 69
    4.2.1. The example 70
      4.2.1.1. Hierarchy structure of the example 70
      4.2.1.2. Content of the tables 71
      4.2.1.3. The FILES screen 73
      4.2.1.4. Questioning the user 74
      4.2.1.5. History 75
      4.2.1.6. Results 76
      4.2.1.7. Example sessions 77
    4.2.2. Possible improvements 79
  4.3. Implementational details 80
    4.3.1. Main data structures 80
    4.3.2. The file structure for a table 82
    4.3.3. Efficiency 83

5. Evaluation of the HICLASS system 84
  5.1. HICLASS as a tool for a generic task 84
  5.2. HICLASS as a tool for hierarchical classification 86
  5.3. HICLASS in comparison 88
    5.3.1. Description of CSRL 88
    5.3.2. HICLASS vs. CSRL 91
    5.3.3. Description of 1st-CLASS 93
      5.3.3.1. 1st-CLASS specifications 93
      5.3.3.2. Using 1st-CLASS 94
    5.3.4. HICLASS vs. 1st-CLASS 99

6. Further research 100
  6.1. HIHYPO - hierarchical hypothesis matching 100
    6.1.1. Local control strategy and knowledge representation 102
    6.1.2. Selected special problems 104
      6.1.2.1. Class descriptions with different weights 104
      6.1.2.2. An answer UNKNOWN 105
  6.2. A complex problem-solver 106
  6.3. Inductive learning 108
    6.3.1. Version space 109
    6.3.2. Quinlan's ID3 110
    6.3.3. AQ11 111
    6.3.4. Genetic algorithms 113
  6.4. An inductive learning algorithm for HIHYPO 114

7. Conclusions 119

References 120

Further Reading 121

Appendix A List of files on the program disk

Appendix B The Software Engineering aspect of the project

Appendix C Modules

0. Introduction

Any expert system shell that performs the generic task of hierarchical classification must deal explicitly with the issues of knowledge representations, control strategies, inductive learning, and ways of handling uncertainty, ambiguity, and contradictions. This research is mainly concerned about the creation of the expert system shell HICLASS. Aspects crucial to this task are challenged from both a theoretical and an implementational point of view.

The principles of generic tasks and hierarchical classification are described. Important concepts of HICLASS are introduced, followed by a detailed description of the knowledge representation and local control strategies developed for the system, including a discussion of special problems and respective solutions. It is described how HICLASS handles uncertainty. Important issues like concluding values, explanation, learning, incorporating metaknowledge, and the global control strategy of HICLASS are discussed. Then, the actual implementation of the table editor HIEDIT as well as HICLASS is described in detail. It is shown that HICLASS is a genuine tool for the generic task of hierarchical classification. The system is compared to two well-known tools for hierarchical classification. Using the ideas raised for HICLASS, the development of a hierarchical hypothesis matcher, HIHYPO, is proposed. Essential features of HIHYPO are introduced. A theoretic overview about algorithms for inductive learning is followed by the description of an inductive learning algorithm developed for HIHYPO. Appendix B provides an overview about software engineering methods, and a discussion about methods actually used to create the HICLASS package. In Appendix C, the definitions of all modules developed for the package are shown.

1. The generic task concept

The following two chapters are mainly a synthesis of [5, pp.215-239], as it relates to this research. An ongoing discussion in AI research is concerned about the classification of expert system tasks. Hayes-Roth, for instance, tried to reflect "the different kinds of task that can be addressed by expert systems technology" [10, p.235]. Two of the categories identified by Hayes-Roth are *diagnosis* and *design*. "Diagnosis systems infer system faults from symptom data. ... Design systems develop configurations of objects that satisfy certain constraints" [10, p.235]. The Hayes-Roth approach has received some criticism, "largely because it appears to mix up different dimensions, and because the categories employed are not mutually exclusive" [10, p.235]. Clancey, on the other hand, proposed an analysis in terms of *generic operations* on a system to answer the question what kinds of operation a program can perform with respect to a real-world system. "Clancey distinguished between *synthetic* operations that *construct* a system and *analytic* operations that *interpret* a system" [10, p.236]. These general concepts can further be specialized, in the case of *construct* into *specify*, *design*, and *assemble*. Expert system shells like *Heracles* (Clancey) and *COAST* (Bennett) have been built that "consider high-level problems and propose architectures that support specific behavorial strategies for them" [5, p.235]. Heracles incorporates *heuristic classification*, a strategy for diagnosis, while COAST is concerned about configuration systems. Despite these efforts to distinguish between different types of knowledge-based reasoning most expert system methodologies developed so far "apply the same strategy ... to both design and diagnosis, as well as to any other task" [5, p.215].

Chandrasekaran proposes the concept of *generic tasks*. Generic tasks are "building blocks out of which more complex problem-solvers or architectures for them can be fabricated" [5, p.235]. Each building block stands for a different type of reasoning "such that each of the types is both generic and widely useful as components of complex reasoning tasks." For each identified task, "languages are developed that encode both the problem-solving strategy and knowledge that is appropriate for solving problems of that type." The intention of the generic task approach is to give the knowledge engineer "access to tools that work at the level of the problem, not the level of the implementation language" [5, pp.215-216].

Each generic task "is characterized by:

1. The kinds of information required as input and the information produced as a result of performing the task.
2. A way to represent and organize the knowledge needed to perform the generic task.
3. The process (algorithm, control, problem solving) that the task uses." [5, pp.215-216]

Some important features of generic tasks as given in [5, pp.234-235] are:

- *multiformity*
  Each task "provides a different way to organize and use knowledge. ... Different problems can use different generic tasks and different combinations of generic tasks."

- *modularity*
  "A knowledge-based system can be designed by functionally decomposing its intended problem-solving task" (e.g. diagnosis) "into several cooperating generic tasks. ... Each generic task provides a way to decompose a particular function into its conceptual parts, ... and allows domain knowledge of other forms to be inserted."

- *knowledge acquisition*
  "Each generic task is associated with its own knowledge acquisition strategy."

- *explanation*
  "... the control strategy of each generic task is specific enough for generating explanations of why the problem solver chose to evaluate or not to evaluate a piece of knowledge."

- *exploiting the interaction between knowledge and inference*
  "... each generic task specifically integrates a particular way of representing knowledge with a particular way of using that knowledge."

Like the ones described below, each generic task is "constrained to perform a limited type of problem solving". A generic task "requires the availability of appropriate domain knowledge" [5, p.235]. The task "needs to be coherent and simple in the sense that it ought to be characterizable by a simple type of knowledge and a family of inference types" [5, p.217].

Types of generic tasks as identified in [5, p.216] are:

hierarchical classification
"... is finding the categories in a classification hierarchy that apply to the situation being analyzed."

plan selection and refinement
"... is designing an object using hierarchical planning."

knowledge-directed information passing
"... is determining the attribute of some datum based on the attributes of conceptually related data."

> *hypothesis matching*
> "... is matching hypotheses to a situation using a hierarchical representation of evidence abstractions. The general idea is that we have a set of data which potentially pertain to a concept. We want to know how well the concept matches the data."

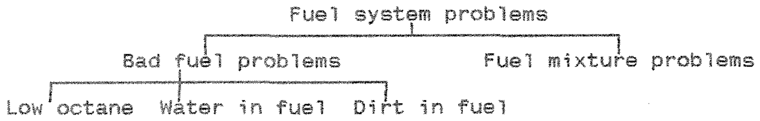
> *hypothesis assembly*
> "... is constructing composite hypotheses in order to account for some set of the data."

A "number of well-known expert systems can be thought of as decomposable into one or more of these generic tasks. For example, R1 performs a simplified type of plan selection and refinement, while MYCIN performs classification and data abstraction" as well as "plan selection". "PEIRCE is the tool for the hypothesis assembly task; INTERNIST and DENDRAL also perform this task in large measure" [5, p.217]. In all of these cases, the system performs the tasks, but not necessarily with a method most natural for the particular task. As mentioned, different problems can use different generic tasks and different combinations of generic tasks. For example, diagnosis uses classification and hypothesis assembly. It is a compound task, since different distinct types of knowledge and inferences are used.

2. Hierarchical classification

Hierarchical classification performs one problem-solving task in human reasoning, classification, under the condition that there is a "classification hierarchy that organizes the classificatory hypotheses" [5, p.218]. "Hierarchical classification requires as input a data description of the problem to be solved. After processing, the task yields all the categories of the malfunction hierarchy that apply to the given data. ... The classifier requires a preenumerated list of the categories that it will be using. Furthermore, these categories must be organized into a hierarchy in which the children (...) of a node represent subhypotheses of the parent. ... As the hierarchy is traversed from the top down, the categories (...) become more specific" [5, p.218].

Example:



"Each node in the hierarchy is responsible for calculating the 'degree of fit', or confidence value, of the hypotheses that the node represents. ... Each node can be thought of as an expert in determining whether the hypothesis is true. For this reason, each node is termed a specialist in its small domain. To create each specialist, knowledge must be provided to make the degree-of-confidence decision. The general idea is that each specialist specifies a list of features that are important in determining whether the hypothesis it represents is true and a list of patterns that map combinations of features to confidence values" [5, p.219].

In order to efficiently traverse the hierarchy, a type of hypothesis refinement is used: establish-refine. That is, "a specialist that establishes its hypothesis (...) refines itself by activating its more detailed subspecialists, while a specialist that rules out or reject its hypothesis (...) does not send any messages to its subspecialists, thus avoiding that entire part of the hierarchy. ... The establish-refine process continues until no more refinements can take place. This can occur either by having reached the tip level hypothesis of the hierarchy or by having ruled out mid-hierarchy hypotheses" [5, p.219].

3. The HICLASS system

3.1. Introduction

The expert system shell that is the topic of this research (to be referred to as *HICLASS*) will essentially follow the ideas raised in chapters 1 and 2. In chapters 5.1. and 5.2. a critique of HICLASS, with respect to the issues raised in the first two chapters, is given.

3.1.1. Terminology

In HICLASS

- a category will be referred to as a *class*
- a *table* is a node in a hierarchy (or a specialist)
- a table consists of one or more classes
- a class is described by one or more *attributes* (or features)
- all classes within one table share the same set of attributes
- an attribute is defined on an underlying finite set of acceptable *values* for that attribute
- a class is described by a list of *instances* (or patterns) that map combinations of values to *weights* (or confidence values)

3.1.2. The concept of "table"

One important feature of HICLASS is the existence of the concept *table*. In a table, several classes are combined. This is an advantage if several classes with the same parent share attributes and can therefore be compared with each other under the assumption that they are distinguishable by the attributes. The attempt is to come up with one relevant class per set to partition the search space in the most radical way. This among other things prevents the system from requiring additional information if the evidence for one class assures that all other classes can be ruled out. If two or more classes have a certainty of being true greater than a predefined threshold, then there will be several solutions for the particular table.

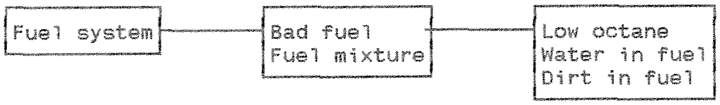


Figure 3.1.2.1. Example of a hierarchy of tables

Given a situation in which classes with the same parent do not share attributes, sets of classes within a table become impractical, since all the attributes of all classes have to be considered. In this case, each class is described by its own table, and all of these tables have to be considered in order to establish or rule out hypotheses.

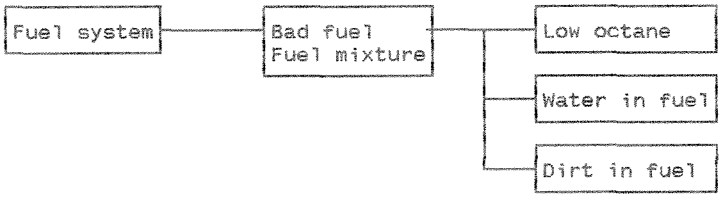


Figure 3.1.2.2. Example of a hierarchy of tables (single classes)

3.1.3. Knowledge representation

3.1.3.1. Sets as a basic approach

Within a node of the hierarchy, given there are several classes combined in a table, the task is to differentiate instances of one class from another. "... instances, each described in terms of a fixed number of attributes. Each attribute in turn has a small number of discrete possible values, and so an instance is specified by the values it takes for each attribute." [12, p.196]. An instance can be considered a set of values, a set of instances describes a class and a table is a set of classes. Therefore, the basic knowledge representation is based on sets and basic set operations serve as operators on sets.

In earlier stages of this research the special philosophy of XBOOLE [2,14] has been used to define these sets. XBOOLE is based on an extension of BOOLEAN algebra while introducing a third state variable '-', standing for '0' OR '1', in other words for 'Don't care'. The introduction of 'Don't care' is crucial to an improvement of the performance of a classification system. It was proved though that it is not useful for this purpose to use the toolbox XBOOLE/XB_PORT [7]. The amount of problems introduced would be much higher than the number of advantages gained.

3.1.3.2. Preenumerated solutions

An important feature of a classification system is that solutions can be enumerated in advance, e.g. "in the diagnosis phase of MYCIN, the program selects from a fixed set of offending organisms" [10, p.242]. In HICLASS, tables are defined within a hierarchy. A number of attributes is defined for each table. These attributes have well defined values for instances of a particular class description. They serve to rule out classes in the case of a class set and to determine the certainty value of one or more succeeding classes. A class description consists of one or more instances that provide values for all the attributes, including one special attribute, the *result*, representing a hypothesis. Prior certainties can be bound to the values and to the instance itself (a weight in the latter case). Thus, if an instance can be matched, a result with an associated certainty is produced. The results of a table represent the interfaces to nodes on a lower level in the hierarchy.

Example:

type	size	location	creature	weight
cetacea	25 ft.	at sea	whale	1.0
cetacea	20 ft.	at sea	whale	0.9
cetacea	6 ft.	near coast	porpoise	1.0
cetacea	6 ft.	at sea	dolphin	1.0
fish	1 ft.	n.pacific	salmon	1.0
fish	6 ft.	at sea	shark	1.0

Figure 3.1.3.2.1. Example of a table definition

In the above example there are four attributes defined, each of them described by several values.

```
type     = {cetacea , fish}
size     = {25 ft. , 20 ft. , 6 ft. , 1 ft.}
location = {at sea , near coast , n.pacific}
creature = {whale , porpoise , dolphin , salmon , shark}
```

The attribute creature has the special property of describing a result. There are five different class descriptions. Except the class whale (two instances), all classes are described with one instance. There are weight values bound to each instance. A weight of '1.0' means 'It is for sure that this description is true'. Weights are defined as numbers [0.1 .. 1.0].

3.1.3.3. "Don't care" values

A "Don't care" value will be denoted by '*' and substitutes values for a given attribute. Within the limited universe of one attribute definition it stands for all possible values this attribute has, and therefore it actually disables this attribute from being part of a decision using an instance including "Don't care" for this attribute. There are several reasons why it can be useful to have such a special value:

1. If the knowledge used to describe a class is incomplete, i.e. if no decision can be made which of the defined values for the attribute is the appropriate one.

2. If in a table one attribute is not applicable (or not defined) for a certain class.

3. If the attempt is to generalize the description.

Examples:

1.

```
cetacea    25 ft.    *    whale    0.8
```

"If <cetacea>and<25 ft.>and<at sea,near coast,n.pacific> then <whale> with 80% confidence".

There was no information available about the location of <whale>, thus this attribute was disabled and the weight adjusted according to the incomplete information.

2.

type	size	color_of_feathers	creature	weight
cetacea	25 ft.	*	whale	1.0
bird	1 ft.	white	albatross	1.0

"If <cetacea>and<25 ft.>and<at sea,near coast,n.pacific> then <whale> for sure".

Here, '*' means "Not applicable". Since this is not distinguishable to the "Don't care" case, problems could arise concerning not only explanation features but also the logic of questions generated by the system. It should therefore be avoided to build tables in this fashion.

3.

```
cetacea    25 ft.    *    whale    1.0
```

"If <cetacea>and<25 ft.>and<at sea,near coast,n.pacific> then <whale> for sure".

Now, the class description was generalized in order to make the inference process easier and faster. It doesn't matter, what the location of <whale> is, since there is confidence that <whale> can be found at all possible locations.

3.1.4. The hierarchy

Given an input vector, describing a special instance, the task is to classify this instance, this means to find the class (the result) it belongs to according to the description of this class. This process is common to many domains. Examples can for instance be found in zoology and botany applications; field guides provide a form of 'manual classification'. A set reduction takes place, the original set of all classes will be reduced when the classification process goes on.

"The classes involved usually have a hierarchical organization, in which subclasses possess the discriminating features of their superclasses, and classes which are 'siblings' in the hierarchy are mutually exclusive with respect to the presence or absence of some set of features." [10, p.238].

Example: (based on [6])

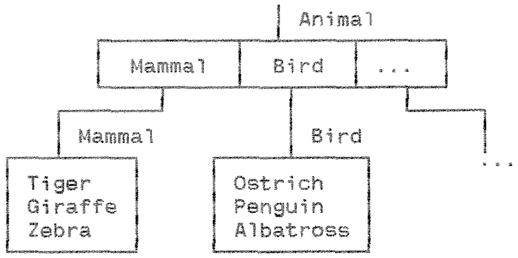


Figure 3.1.4.1. Example of a hierarchy

In the example the first level in the hierarchy is more abstract than the second one. At the first level we only have one *root* table (a set of classes) describing basic principles of animals (principle 'mammal', 'bird'). Attributes are used to distinguish among classes (like 'can-fly'). If the problem at the first level is solved, which means the result of this table is found (one or more classes), the search space can be reduced not only by the non-succeeding classes of this set but also by entire branches of the tree, starting with the children of the classes that are rejected. At the next level, considering that we found to deal with a 'mammal', we observe the appropriate set, again consisting of classes but different attributes to distinguish among these new classes (like 'has-long-neck').

3.1.5. The control strategy

Performance-oriented expert systems (HICLASS can be considered as belonging to this category) "start with a representation of knowledge about a task or domain and attempt to build a program that displays competent behavior in that domain" [3, pp.25-26]. There must be knowledge about HOW an expert would solve the problem based on the knowledge available, HOW the problem solving process is guided, how to detect and measure errors and to deal with contradictions.

Since we have to deal with a hierarchy, 'top-down-refinement' can be used. "The method of 'top-down-refinement' uses 'levels' of abstraction. Higher levels are more abstract than lower levels. When the expert system has solved the problem at one level it moves down to the next, more detailed, level. The order and content of levels is predefined, whereas the order in which sub-problems on a particular level are tackled is dependent on the task in hand." [9, p.449].

The basic principle of reasoning in a classification system is reasoning by elimination. "Reasoning by elimination is an approach in which non-solutions (or solutions with low plausibility) are pruned from the search space as early as possible. To do this, the expert must partition the search space in such a way that early pruning can be achieved." [9, p.448]. This definition is adequate for a system based on sets and set reduction.

A control strategy designed to serve reasoning by elimination is Chandrasekarans "establish-refine" (as described in chapter 2). HICLASS uses the very idea of this strategy. In HICLASS, one or more results with a certainty value bound to them are produced after a table is "solved" - hypotheses are established. The process continues while invoking the subspecialists (or classes) a particular result is pointing to (the hypothesis refines itself). If the subspecialists are combined in a set, only one pointer is necessary, otherwise more than one. In class sets, wrong hypotheses are either automatically ruled out in the set reduction process or a certainty value of zero is assigned to them. In both cases, the subspecialist of these classes will not be established. The process stops when all paths followed terminate because all current tables are leafs in the classification tree, and when all current hypotheses are either ruled out or hold a certainty value smaller than a predefined threshold.

3.2. Comparison of the HICLASS knowledge representation to other representations

The hierarchy of tables as well as their predefined content using attributes can be seen as a frame-like structure. Besides some similarities though, there are basic differences between the two forms of knowledge representation.

"Each frame contains information about one particular object, concept, or event and typically has slots which contain values. ... The prototype frame for a class will contain the list of slots applicable to the class and can also contain default values or valid ranges of values for these slots. An instance frame for that class will then contain the detailed information for that particular instance." [11, p.285].

One similarity between frames and a hierarchical table structure is that a table includes attributes (or slots) and that there are values defined for these slots. The major difference is that *multiple* concepts can be stored in a table and that there is nothing like a prototype frame. This leads to a more compact description and incorporates, differently than in a basic frame structure, the reasoning principle, in this case a set reduction philosophy used to distinguish between classes.

Dealing with frames there are 'ISA' and 'AKO' relationships, where 'ISA' defines an instance of a class defined by the prototype and 'AKO' ('a kind of') the superclass-subclass relationship between frames. In our case there is an 'AKO' relationship ('mammal' is a kind of 'animal'), referring to a *subset* (a class) of the corresponding class set on a higher level. The table *itself* can be seen as a prototype frame, filled with different class descriptions. The 'AKO' relationships have all the typical qualities, like inheritance of values from general classes to more specific classes.

However, there are also some similarities to rule-based systems. A construction like 'if (bird)and(cannot fly)and(has long neck)and(...) then ostrich' is a rule. One of the basic differences between a *real* rule-based system and the current approach is that the interaction between rules follows other principles (no channel of interaction via a database). But, as we will see in the next chapter, a combination of ideas from different basic approaches can make distinctions like this very fuzzy.

3.3. Fuzzy borders: From frame-like objects to rules

As described above, there are similarities between the HICLASS knowledge representation and a representation based on rules. Even if the system will not be implemented in a rule-like fashion, it can be interesting how the proposed representation could be altered to handle this task. Additionally, there is a chance to open up the classical hierarchical approach in order to enrich the reasoning process.

In the following examples, the attribute name *result* denotes that an exit-condition is bound to the table, whereas *class* has to be seen as describing a subsolution.

3.3.1. The "classical" hierarchical approach

Since the approach below follows the description in 3.1.4., no further comments are given. The hierarchy in figure 3.3.1.1. consists of the three tables A, B and C. Attributes (a1,a2,...,c4) are defined for each table. A special attribute denotes the *result* of a particular table (class_A, result_B, result_C).

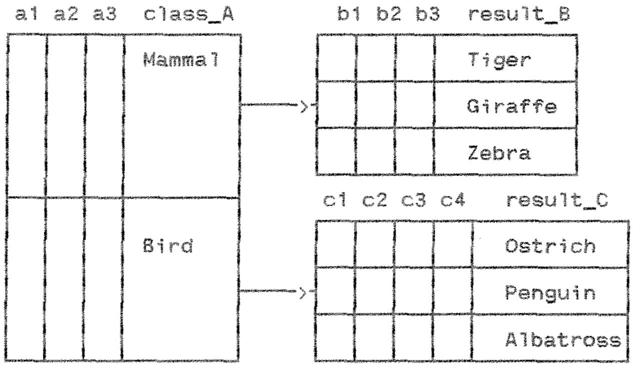
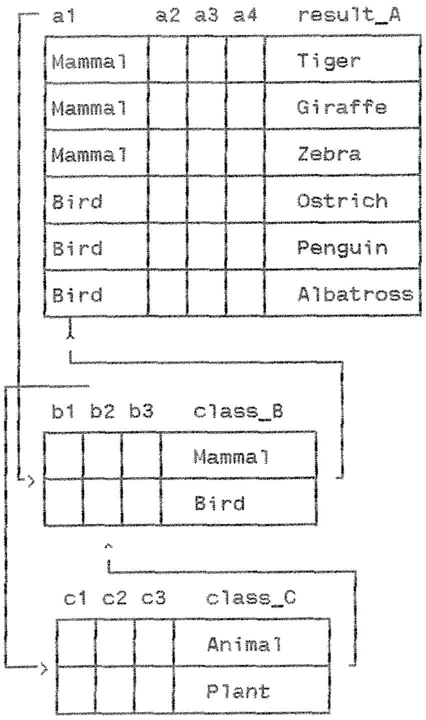


Figure 3.3.1.1. Tables in a hierarchy

3.3.2. Opening the "classical" structure

So far it was assumed that the attributes (a1,a2,...,c4) are only dependent on an external input (e.g. a question to the user) rather than that their values are provided by another table. Given an application in which the number of results (combined with an exit-condition) is relatively small and the distinction between them fairly easy except some attributes which itself are more complex to be derived, the following structure would make more sense, since it can turn out that it is not necessary to derive these complex values (other attributes might be reasonable for a distinction), thus we don't have to go all the way down in a tree structure.

NOTE: The zoological content of the example is not very appropriate to illustrate the concept, it nevertheless was chosen to be consistent.



a1	a2	a3	a4	result_A
Mammal				Tiger
Mammal				Giraffe
Mammal				Zebra
Bird				Ostrich
Bird				Penguin
Bird				Albatross

b1	b2	b3	class_B
			Mammal
			Bird

c1	c2	c3	class_C
			Animal
			Plant

Figure 3.3.2.1. Complex interactions between the tables

In figure 3.3.2.1., the description for the attributes a1 and b2 includes a pointer to other tables. Table B can be used to provide a value for attribute a1 of table A, and table C would provide a value for attribute b2. If the knowledge of the value of attribute a2 is sufficient to come to a result for table A we don't have to 'fire' the tables B and C. Otherwise, the classification 'Mammal/Bird' derived from table B could serve as an input to detect the value of a1. For the first moment this structure seems to drop the feature of inheritance between values from general classes to more specific classes. But this information is not lost, since table B holds all the necessary information which can be derived from it. Actually, the structure doesn't violate any important feature of a classification structure; it only represents it in a different way.

3.3.3. The combination of the two concepts

Depending on the problem on hand, the structure must be flexible enough to handle very different hierarchy descriptions. A combination of the original with the modified structure is possible and allows much more flexibility. The tables can be chained in any imaginable way resulting in a structure serving most classification and efficiency needs. This is the approach implemented in HICLASS.

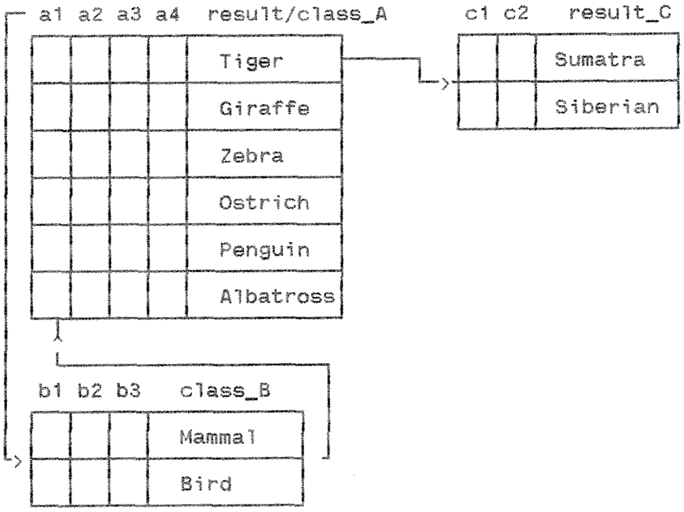


Figure 3.3.3.1. Complex interactions in a hierarchy

3.3.4. Introduction of a factbase

Up to now a hierarchy was more or less very strictly defined. In applications with more complex structures and eventually competing solutions, it might be more appropriate to not follow the branches of a predefined tree. The strictly defined chaining could be substituted by an independent channel of interaction - a database or factbase. One could say that this is the beginning of giving up any kind of hierarchy. This is not true. Even in pure rule-based systems (that are obviously the destination of the 'evolution' process described here) one can find a sort of hierarchy in most of the cases; more hidden, sometimes not *truly* hierarchic. Given a backward-chaining approach and a rule activating another, the two rules often have a hierarchical relationship between each other. "The indirect, limited interaction is also, however, the most significant factor that makes the behavior of a Production System more difficult to analyze". [3, p.32]. For the beginning we still maintain the concept of tables, which are sets of classes that are distinguishable from each other using attributes defined for the table. The difference is the way the tables interact. Now, they are not explicitly calling each other but having access to a central factbase which will be updated depending on the process of leading the search.

a1	a2	a3	a4	result/class_A
Mammal				Tiger
Mammal				Giraffe
Mammal				Zebra
Bird				Ostrich
Bird				Penguin
Bird				Albatross

b1	b2	b3	class_B
			Mammal
			Bird

c1	c2	c3	result_C
Tiger			Sumatra
Tiger			Siberian

d2	d3	class_D
		Tiger
		Lion

Figure 3.3.4.1. Sets interacting via a factbase

Assuming that 'Sumatra' is the desired result of a particular run of the small expert system incorporating a hierarchy as shown in figure 3.3.4.1., the fact 'Tiger' is needed in order to fire table C. Both table A and D can provide this fact as a result. If one of these tables can successfully be solved, the result of the table is added to the factbase, from where the control instance could get the necessary information to fire C. Table D though might provide the fact 'Tiger' much faster and easier than table A would do. Thus, it might be better to follow the short path rather than the longer one. However, the decision of which table to 'fire' remains a problem for a good control strategy.

3.3.5. A Rule-based system

In a pure rule-based system, the one and only interaction between the *rules* is realized by the factbase. "...we have a completely ordered set of rules, with no interaction channel other than the database. ... all interaction must occur by the effect of modifications written in the database; ... these modifications are accessible to every one of the rules. ... a system, that is strongly modular, since no rule is ever called directly. ... Production systems emphasize the statement of independent chunks of knowledge from a domain and make control a secondary issue." [3, pp.32,35].

With respect to classification issues, the *only* structures remaining are tables holding the description of one specific class, that are rules. A rule-based approach allows a greater flexibility in mapping attributes to different classes, that are now self-supporting and not explicitly embedded in a context with other classes from which they have to be distinguished, which in turn also raises the chance of incorporating contradictions and incomplete descriptions.

It is still possible to keep principles of inheritance. For example, the rules for (tiger) and (giraffe) indirectly refer to the (mammal) concept. If one class has an unique attribute, it will only appear in this special rule, whereas before it had to be incorporated for all other members of the same set as well (a lot of "Don't cares" are the result).

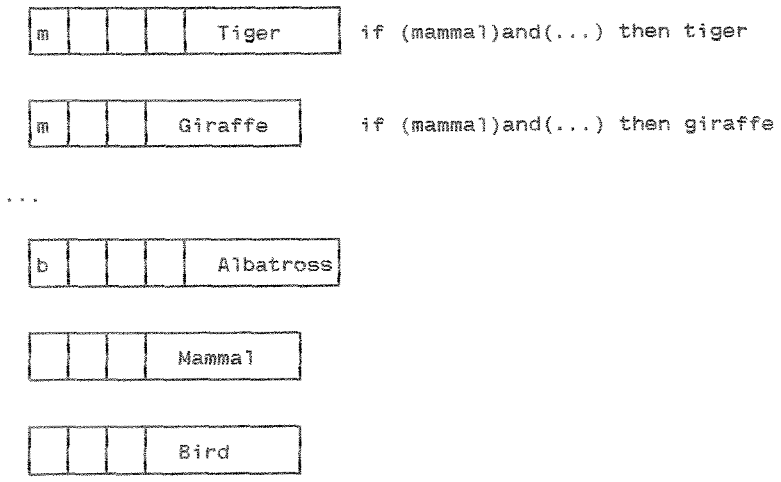


Figure 3.3.5.1. Independent rules interacting via a factbase

3.4. Local control strategies

The main goal of a classification system is to find a class an unknown instance belongs to (a *result*). This overall goal simplifies some things. The termination condition of a search within a table will be true if there is only one result left, even if there are more rows in the table referring to that result, given that all the instances left carry the same weight. If the latter is not the case, the termination condition is fulfilled if there are only instances with the same weight left. Thus, the goal is to come up with an *unique* result, unique in both the value for the result and the weight attached to it. The basic reasoning method used in a classification system is reasoning by elimination. Dealing with predefined hierarchy structures we have to lead the process of getting a result *within* one table.

A forward chaining approach is also known as a *data-driven* approach. In a rule-based system the rule application module "cycles through the rules looking for one whose condition part is satisfied by the database. When it finds such a rule, it invokes the action part. In many cases, the action results in changes to the database which enable other rules. The rule application module continues cycling either (i) the problem is solved (the goal is achieved) or (ii) a state is reached where no more rules can be invoked. ... In using this method, one begins by entering data about the current problem into the database." [9, pp.421-422, 424]

The statement given above is concerned about rule-based systems. Some things change in HICLASS. In general, there will be no initial facts given, a factbase in the sense of a rule-based system doesn't exist anyway. Knowledge is provided with class descriptions (that can also be interpreted as *rules*), grouped together within tables. The first step is up to the system, which will start with asking a question to a user, attempting to acquire information about a particular result while giving a multiple choice of all values defined for a particular attribute.

To decide which question to ask next is the problem of the control strategy we are concerned about in this chapter. Since there is a predefined hierarchy structure, there will be one table to be invoked first. If the first question really comes from this table or from a table called from it, depends on the particular situation. The control strategy only works in a limited environment, within one table. Contrasting this with a data-driven approach in rule-based systems we can say that in our case it is clear which "rule" to fire according to the hierarchy. We are concerned about which part of the "rule" to examine next. The process is not driven by the input data but by the remaining data in the knowledgebase. Hence, we will use the term forward chaining, thus contrasting to a backward chaining, goal-driven approach.

Using ideas of a forward chaining approach, there are some different ways of deciding which question to ask/investigate next. This depends on the knowledge incorporated in the table.

3.4.2. Left-to-Right

Again, the questions are asked left to right, starting with the first one. This time though, every answered question leads to a reduction of the contents of the table (using intersection again). Hence it can happen that not all questions need to be asked anymore. Attributes with the same value or with <Don't care> in all rows do not need to be considered anymore; the first applicable question identified is selected.

Example:

```
What is the class of the creature?   fish

fish       1 ft.    n.pacific   salmon     1.0
fish       6 ft.    at sea      shark      1.0


What is the length of the creature?   1 ft.

fish       1 ft.    n.pacific   salmon     1.0

Result: The creature is a salmon.
```

3.4.3. Heuristic driven

The approaches described above are straight forward, the control strategy does not involve any knowledge (in the case of MATCH) or only a little knowledge (in the case of Left-to-Right) about the contents of the table. A heuristic approach would not simply go in one predefined direction but choose every question according to the knowledge available.

One idea for a heuristic could be to use the one developed in [14]. There, values have been evaluated according to the amount to which the knowledge about them would reduce the amount of values remaining. But we have to deal with a different situation now. The answer to a particular question is not dual anymore, but can be chosen in a multiple choice fashion, thus involving all possible answers. We cannot predict the answer of the user. The goal to be achieved is to find a unique result. Hence, the question chosen should serve best to distinguish between classes.

A useful heuristic would be to prefer a question with the highest amount of possible answer values, since the more values are defined, the higher is the chance of an distinction among classes. 'Possible answer values' means that only values are counted which are still valid in a reduced set.

Example:

```
type: 2 values
size: 3 values
location: 3 values

What is the length of the creature?   1 ft.

fish       1 ft.    n.pacific   salmon     1.0

Result: The creature is a salmon.
```

This heuristic seems to work fine. An important point to make is that a heuristic driven approach is only applicable if the order of questions is not predefined!

3.5. Class descriptions with different weights

As mentioned above, the termination condition of the local strategies is fulfilled if there is only one class left and if all the instances of this class carry the same weight. The goal is to come up with an unique result, unique in both the value for the result and the weight attached to it. So far, the problem of different weights for a succeeding class was ignored and will be discussed below.

The following table will be used during the discussion:

type	size	location	creature	weight
cetacea	25 ft.	at sea	whale	1.0
cetacea	25 ft.	near coast	whale	0.9
fish	1 ft.	near coast	salmon	1.0

3.5.1. MATCH

Applying a MATCH strategy to the table given above, the following sequence of action is possible:

Example:

```
What is the class of the creature?     cetacea
What is the length of the creature?    25 ft.
Where does the creature live?          near coast

cetacea   25 ft.   near coast   whale      0.9

Result: It is 90% for sure that the creature is a whale.
```

There is no problem, all questions are asked anyway and a unique result is produced.

3.5.2. Left-to-right

If a Left-to-right strategy is applied to the table, the following could happen:

Example:

What is the class of the creature? cetacea

cetacea	25 ft.	at sea	whale	1.0
cetacea	25 ft.	near coast	whale	0.9

After the first question is answered with <cetacea>, a set reduction takes place, removing all class descriptions from the set that have values for <type> different to <cetacea>. As the result, there is only one class left, but this time the result is not unique yet, since there are different weights attached to the instances of this class. We have to ask one more question in order to get a unique result for the table.

Where does the creature live? at sea

cetacea	25 ft.	at sea	whale	1.0

Result: It is for sure that the creature is a whale.

A different way of looking at this problem could be not to ask the last question but stating that the result is <whale> with 90% confidence (0.9 is the minimum of all the weights for <whale>). Doing this, we would not use all the knowledge available, and therefore weaken the power of advice possible. In order to allow this "shortcut", a specific order of action has to be maintained. First, the goal is to find unique results, then, given there are multiple instances of one or more results left carrying different weights, unique results with a specific weight are needed.

3.5.3. Heuristic driven

Finally, a heuristic driven control strategy could be applied in an attempt to solve the table:

Example:

```
type: 2 values
size: 2 values
location: 2 values

What is the class of the creature?   cetacea

cetacea   25 ft.   at sea      whale      1.0
cetacea   25 ft.   near coast  whale      0.9


size: 2 values (but only one value for this attribute left in the table)
location: 2 values

Where does the creature live?     near coast

cetacea   25 ft.   near coast  whale      0.9

Result: It is 90% for sure that the creature is a whale.
```

The order and amount of questions turned out to be the same as in 3.5.2., which must not necessarily be the case and is due to the simple example.

3.6. Different classes with the same content but different weights

It might happen that different class descriptions have the same content but a different weight attached to it. It should be possible to handle the following example.

Example:

cetacea	25 ft.	at sea	whale	0.9
cetacea	25 ft.	at sea	monster	0.1

There is no way to distinguish between the two classes, hence we will have to deal with a set of results rather than with one unique result.

3.7. ASKFIRST

Using an approach in which questions for values of specific attributes can be answered by another table, we sometimes should nevertheless consider to let the user answer first. If by chance he knows the answer, we don't have to involve the other table or, even worse a series of tables. In case the human's answer is UNKNOWN we then can "fire" the table to solve the problem. *When* this combined option should be used, depends on the specific situation. This principle is also used in the MYCIN system: "...each parameter be labeled as an ASKFIRST attribute (...) or as a parameter that should first be determined by using rules rather than by asking the user." [3, p.64].

3.8. One class, one table and multiple children

In 3.1.2. it was stated that in a situation in which classes with the same parent do not share attributes, sets of classes within a table become impractical, since all the attributes of all classes have to be considered. In this case, each class is described by its own table, and all of these tables have to be considered in order to establish or rule out hypotheses. Does this have any effect on the local control strategies used?

The termination condition for local control strategies so far was to produce one unique class with a unique weight. In the best case, only a subset of all possible questions has to be asked in order to terminate the search (except MATCH, all strategies are aimed at minimizing the amount of questions necessary).

Example:

A:

type	size	size_of_babies	creature	weight
cetacea	25 ft.	3 ft.	whale	1.0
cetacea	25 ft.	6 ft.	whale	0.9

B:

type	size	location	creature	weight
fish	1 ft.	near coast	salmon	1.0

There are two tables, both are called from the same parent, and both have to be considered in order to continue with the classification. What can be done? There are two instances with different weights in table A. A heuristic driven strategy can be used to simplify the table content.

What is the size of the babies? 3 ft.

A:	cetacea	25 ft.	3 ft.	whale	1.0
B:	fish	1 ft.	near coast	salmon	1.0

Both tables fulfill the termination condition - unique class descriptions with unique weights. At least we know that <whale> has <3 ft.> long babies (the question was asked only for this table!). But so far, we also know for sure that the creature is a salmon, without even generating a question (a very smart system...,or not?). There is no way to use knowledge acquired within the context of one table for another, since completely different attributes are defined; even if they can share the same name, e.g. <type>, they are NOT the same! The only way to *really* be sure that no wrong information will be produced is to ask all questions available within one table, i.e. we have to use MATCH as long as there are multiple children left.

Additionally, we have to give another answer option to the user to avoid confusion, since questions and answers might be generated that do not relate at all to the instance which has to be classified. This option is <Not applicable> and results, when chosen, in an immediate termination of the search and in assigning a certainty of zero to the particular table.

a) assuming the user "sees" a <whale>

Table A:

What is the class of the creature?	cetacea
What is the length of the creature?	25 ft.
What is the size of the babies?	3 ft.

A:	cetacea	25 ft.	3 ft.	whale	1.0

Table B:

What is the class of the creature?	Not applicable

The choices <fish> and <Not applicable> were given to the user.
The user chose <Not applicable> because he/she "sees" a <whale>,
which is NOT a <fish>.

Result: The creature is a whale.

b) assuming the user "sees" a <salmon>

Table A:

What is the class of the creature?	Not applicable

The choices <cetacea> and <Not applicable> were given to the user.
The user chose <Not applicable> because he/she "sees" a <salmon>,
which is NOT a <cetacea>.

Result: The creature is a salmon.

It was not necessary to ask any other question, since B is the only table left and has a unique result with a unique weight (the world view is limited to the information stored in the system!).

A practical problem appears given a parent calling multiple children. There is only one result defined per instance, thus only one pointer. In order to call multiple tables at the same time, a dummy table has to be inserted.

3.9. An answer UNKNOWN

Imagine trying to identify this strange animal you saw the other day. The computer asks you "With what are the supraorbital processes fused to the braincase?" and gives the choices "with part of posterior projection" or "with posterior extension". WHAT??! "I don't know!"

To handle situations like this there is a need for a default answer, namely UNKNOWN. And maybe other questions are easier to answer and it is nevertheless possible to proceed in the classification. How to deal with this answer? Which problems arise given this choice?

The easiest way to handle the situation is to simply ignore the UNKNOWN question/answer as not helpful to the classification process. This of course will have more or less serious effects, depending on the control strategies chosen and on the content of the particular set. Let us now investigate the effects, considering different control strategies. Without further discussion it should be stated that, if a question marked as ASKFIRST is answered with UNKNOWN, we will try to find an answer using a table designed to provide this answer (if there is such a table).

Example:

type	size	location	creature	weight
cetacea	25 ft.	at sea	whale	1.0
cetacea	6 ft.	near coast	porpoise	1.0
cetacea	6 ft.	at sea	dolphin	1.0
fish	1 ft.	n.pacific	salmon	1.0
fish	6 ft.	at sea	shark	1.0

3.9.1. MATCH

With this approach, every question is asked from left to right without any further action involved. An input vector is built and compared to the table.

Example:

```
vector1 = { cetacea, UNKNOWN, near coast }

cetacea   6 ft.    near coast  porpoise   1.0

vector2= { cetacea, UNKNOWN, at sea }

cetacea   25 ft.   at sea      whale      1.0
cetacea   6 ft.    at sea      dolphin    1.0
```

We can get sets of solutions rather than one unique result after exhausting all the possible questions. Since the input vector is incomplete it might not be possible anymore to fully classify instances. The answer UNKNOWN has no effect on the way the control strategy works, since all questions are asked anyway.

3.9.2. Left-to-right

Questions are asked from the left to the right, and every answer leads to a reduction of the table. Hence it can happen that not all questions have to be asked anymore. The answer UNKNOWN is ignored, no action will take place, thus no reduction. Depending on the particular situation this will effect the number of questions to be asked.

Example:

```
What is the class of the creature?   cetacea

cetacea   25 ft.    at sea      whale      1.0
cetacea   6 ft.     near coast  porpoise   1.0
cetacea   6 ft.     at sea      dolphin    1.0


What is the length of the creature?   UNKNOWN
no change in the table


Where does the creature live?   at sea

cetacea   25 ft.    at sea      whale      1.0
cetacea   6 ft.     at sea      dolphin    1.0
```

The result is the same as with MATCH, all questions had to be asked. It can easily be seen that if the answer to the second question would have been <25 ft.>, the third question would have been useless, since the result would have been clear: <whale>. Hence, the answer UNKNOWN effected the efficiency of the control strategy.

3.9.3. Heuristic driven

The heuristic driven approach does not simply proceed in one predefined direction, but chooses every question according to the knowledge available. The heuristic to be used here is to prefer a question with the highest amount of possible answer values.

Example:

```
type: 2 values
size: 3 values
location: 3 values

What is the length of the creature?   UNKNOWN
no change in table


Where does the creature live?   at sea

cetacea    25 ft.    at sea      whale      1.0
cetacea    6 ft.     at sea      dolphin    1.0
fish       6 ft.     at sea      shark      1.0


type: 2 values

What is the class of the creature?   cetacea

cetacea    25 ft.    at sea      whale      1.0
cetacea    6 ft.     at sea      dolphin    1.0
```

Again, we have the same result as before; and the same effects on the efficiency of the control strategy. For the heuristic driven as well for the Left-to-Right approach the effects will not always be as strong as it turned out here, where no savings at all were left after only one answer UNKNOWN. But the effects will be there and there is no way to avoid them.

3.9.4. Global effects

We found that there are two reasons for having a set of results rather than one single unique result: a) if we deal with multiple classes having the same description but different weights attached to them, and b) if multiple results were produced after an answer UNKNOWN.

Since the tables are embedded in a hierarchy structure, we have to take the results from one table in order to find our way through the tree. Different to the examples already given we are not done with stating that there are multiple results and maybe displaying the special values. Depending on the particular situation we basically have to deal with two situations.

Example:

a) A:

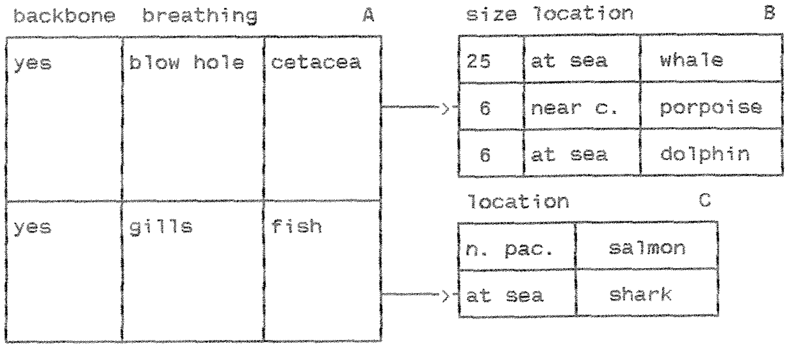
backbone	breathing	type	weight
yes	blow hole	cetacea	1.0
yes	gills	fish	1.0

B:

size	location	creature	weight
25 ft.	at sea	whale	1.0
6 ft.	near coast	porpoise	1.0
6 ft.	at sea	dolphin	1.0

C:

location	creature	weight
n.pacific	salmon	1.0
at sea	shark	1.0



backbone breathing A

yes	blow hole	cetacea
yes	gills	fish

size location B

25	at sea	whale
6	near c.	porpoise
6	at sea	dolphin

location C

n. pac.	salmon
at sea	shark

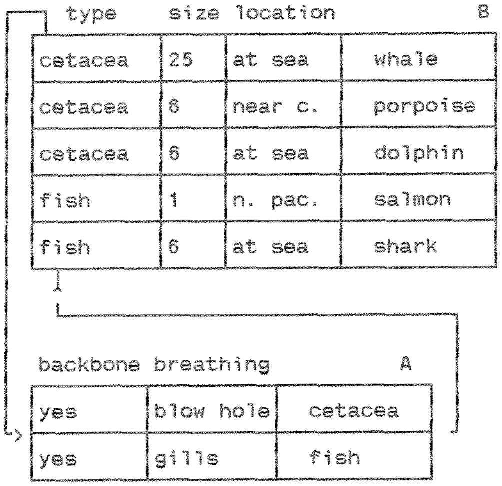
b) A:

backbone	breathing	type	weight
yes	blow hole	cetacea	1.0
yes	gills	fish	1.0

B:

type	size	location	creature	weight
cetacea	25 ft.	at sea	whale	1.0
cetacea	6 ft.	near coast	porpoise	1.0
cetacea	6 ft.	at sea	dolphin	1.0
fish	1 ft.	n.pacific	salmon	1.0
fish	6 ft.	at sea	shark	1.0

(The chaining condition for attribute <type> in A is chosen to be ASKFIRST=false)



B

type	size	location	
cetacea	25	at sea	whale
cetacea	6	near c.	porpoise
cetacea	6	at sea	dolphin
fish	1	n. pac.	salmon
fish	6	at sea	shark

A

backbone	breathing	
yes	blow hole	cetacea
yes	gills	fish

Using a left-to-right control strategy, for both a) and b) the first question will be

```
How does the creature breath?    UNKNOWN
```

resulting in

```
A:   yes        blow hole    cetacea     1.0
     yes        gills        fish        1.0
```

Both classes in this set are true given this answer. They both have the same certainty of being true.

For a case like a) we have to follow several paths in parallel, as long as we either reach a dead end or come to a solution. This means that we have to carry the problem with us right to the end, maybe introducing more paths due to additional UNKNOWN answers.

With b) both results must be used in order to reduce table B, which in this particular case (all possible values are still valid) doesn't result in any reduction, which means the same as an answer UNKNOWN for the attribute <class> of table B. We should now ask the user if he/she can specify the class of the creature; we assume here that this answer would be UNKNOWN as well. In other cases though, it might turn out, that at least some reduction can take place, since not all results of the called table might be valid anymore.

In order to show different aspects connected with the raised problems, we will have a look at different classification goals. A left-right control strategy will be used.

I) assuming the user "sees" a <whale>

Ia) What is the size of the cetacea? 25 ft.
(Question from table B.)

Result B: The creature is a whale for sure.

Where does the fish live? Not applicable
(Question from table C.)

The choices <n.pacific>, <at sea> and <not applicable> were given to the user.
And, since the question referred to a fish, the user chose <Not applicable>.

Result: The creature is a whale for sure.

Again, there is a need for an answer <Not applicable>. The condition to be fulfilled for generating this answer is the fact that the investigation is done parallel for different tables within one level of the hierarchy. It is very important to be very explicit in the formulation of the question string to avoid confusion as much as possible. The question should be related to the overall concept of the table. A question like

Where does the creature live?

for table C would certainly be correct, but is not giving any hint, that it is aimed to inquire the location of a <fish>, while the user might "see" a <cetacea>.

Ib) What is the size of the creature? 25 ft.

Result: The creature is a whale for sure.

It turns out that a unique result could be achieved regardless of the previous uncertainty. No further action is necessary.

II) assuming the user "sees" a <shark>

IIa)

```
What is the size of the cetacea?   Not applicable
(Question from table B.)
The choices <25 ft.>, <6 ft.> and <1 ft.> and <not applicable> were given to the user.

What is the location of the fish?   at sea
(Question from table C.)

Result: The creature is a shark.
```

Basically the same comments have to be made as in Ia). Even though, coincidentally, the attribute <location> is defined for both tables does not mean that an answer to one of the appropriate questions is also the answer to the other one. Again it has to be stated: the tables B and C have nothing to do with each other, except the fact that they both are "fired" by table A. They are separate units with their own attribute names, values and questions.

IIb)

```
What is the size of the creature?   6 ft.

B:    cetacea    6 ft.    near coast  porpoise   1.0
      cetacea    6 ft.    at sea      dolphin    1.0
      fish       6 ft.    at sea      shark      1.0

Where does the creature live?  at sea

B:    cetacea    6 ft.    at sea      dolphin    1.0
      fish       6 ft.    at sea      shark      1.0
```

Now, the result is not unique anymore. Both possible results are valid and can be given to the user. No further activity is possible.

3.10. Dealing with uncertainty

Given multiple results so far, we only had to deal with one instance per class and with an equal likelihood of the different results. This will change if there are different weights for class descriptions. The subject of uncertainty is not trivial, a lot of controversy is going on in this area of research. The attempt will be to find a reasonable, consistent approach for the task on hand while providing a practical scheme to work with. Despite the theoretical problems with a certainty theory applied in the MYCIN system, we will use basic ideas of this approach.

As described in [9, p.403], a "certainty measure C(S) is associated with every 'factual' statement S such that:

a) C(S) = 1.0 if S is known to be true
b) C(S) = -1.0 if S is known to be false
c) C(S) = 0.0 if nothing is known about S
d) intermediate values indicate a measure of certainty or uncertainty in S ".

For our problem, we limit the range of measures to the certainty, which means to a range from 0.1 to 1.

a) C(S) = 0.1 if very little is known about S
b) C(S) = 1.0 if S is known to be true
c) intermediate values indicate a measure of certainty in S

The only predefined certainty measures in the system are weights attached to descriptions of classes (or in a different view measures of belief for rules) and thresholds for the termination of paths. Only positive class descriptions with a certainty value equal to or bigger than 0.1 are allowed. For purposes of inductive learning though, a weight of 0.0 can be attached to an example to denote a negative example.

Example:

type	size	location	creature	weight
cetacea	25 ft.	at sea	whale	1.0
cetacea	6 ft.	near coast	porpoise	1.0
cetacea	6 ft.	at sea	dolphin	1.0
fish	1 ft.	n.pacific	salmon	1.0
fish	6 ft.	at sea	shark	1.0

The input vector { cetacea, UNKNOWN, at sea } leads to:

cetacea	25 ft.	at sea	whale	1.0
cetacea	6 ft.	at sea	dolphin	1.0

Both results have the same likelihood of being true. The two remaining class descriptions are independent from each other, each vector can also be seen as one rule leading to a particular result. Therefore we can work with them separately. One result cannot be favored over the other according to the (incomplete) information given to the system. Both results are possible.

Within one vector (or rule) we have to deal with complex conditions, the overall certainty value of these conditions can be computed using results from the theory of fuzzy sets:

```
certainty of [A AND B] = minimum of {C(A), C(B)}
```

If there is missing information (expressed by an answer UNKNOWN), we simply ignore this condition in the process of computing a certainty measure as also shown in MYCIN [3, p.254]:

```
CF[h,s1 & s?] = CF[h,s1]
```

For our example this means:

```
min ( C(cetacea), C(25 ft), C(at sea))
= min (    1.0     ,          ,     1.0  )
= 1.0

min ( C(cetacea), C(6 ft),  C(at sea))
= min (    1.0     ,          ,     1.0  )
= 1.0
```

There can be weights different than <1.0> associated with class descriptions, that are measures of the reliability of those descriptions.

Example:

```
cetacea   25 ft.   at sea   whale   (0.9)
```

to be read as:
If it is a cetacea and 25 ft. long and is living at sea, then
there is strong evidence (0.9) that the creature is a whale.

Assuming that all values of the example can be determined with a certainty of 1.0, we will calculate the resulting certainty value of the class description by multiplying the predefined weight of the description (0.9) with the combined certainty of the condition part (1.0).

```
C(whale) = 1.0 * 0.9 = 0.9
```

Let us now assume a class description, in which *whale* is described with two different vectors (to be read as: 'a whale is a cetacea, is 20 or 25 ft. long and lives at sea').

Example:

type	size	location	creature	weight
cetacea	20 ft.	at sea	whale	0.9
cetacea	25 ft.	at sea	whale	1.0
cetacea	6 ft.	near coast	porpoise	1.0
cetacea	6 ft.	at sea	dolphin	1.0
fish	1 ft.	n.pacific	salmon	1.0
fish	6 ft.	at sea	shark	1.0

Now, the input vector { cetacea, UNKNOWN, at sea } leads to:

cetacea	20 ft.	at sea	whale	0.9
cetacea	25 ft.	at sea	whale	1.0
cetacea	6 ft.	at sea	dolphin	1.0

We have two vectors left for *whale*. Since we are interested in the certainty of the whole class, we have to combine the certainties of both occurrences. Again it is not possible to state that <whale> is more certain than <dolphin>, even if the first one is expressed with two vectors. The two classes are independent of each other. According to the information, both are well defined. The fact THAT there are multiple solutions though points out that there is a problem, that the solution is not unique. To combine two occurrences (or two rules) of the same class, we will use the formula given in [9, p.403]:

$$C(X/A) = C(X) + [CF * [1.0 - C(X)]] .$$

This formula is explained in the following way: "If the certainty value C(X) of a statement is positive, then the most that a rule with positive CF can increase the certainty of X is 1.0 - C(X). This amount is multiplied by CF and added to C(X)."

C(whale) = 0.0 {initial setting}

C(whale/1) = 0.0 + [0.9 * [1.0 - 0.0] = 0.9
C(whale/2) = 0.9 + [1.0 * [1.0 - 0.9] = 1.0

So far we assumed, that the condition part has a certainty value of 1.0, in other words, all conditions are certainly true or unknown, and that the whole table is "fired" for certain. In a hierarchy with numerous interactions between tables this is not true anymore. Tables can be invoked with a certainty less 1.0 and values of conditions determined by other tables can also hold a certainty less than 1.0. It is obvious that these certainties have to be propagated to the table under consideration. Hence, we also have a chance to distinguish between different possible solutions, which we got following different paths of the decision tree. We need to know the total certainty of such a path, such that it expresses the quality of a specific solution.

Given a situation, in which one table calls another to determine the value of one of its attributes and the result of this table is not unique, whereas other values of the first table can be determined with certainty or are unknown, we can get the following:

a1	a2	a3	Result
0.9		1.00	0.9
0.6		1.00	1.0

a1: value = result of another table
a2: value = UNKNOWN
a3: value = certainly true
Result: certainty factor

First, we have to find the minimum of the certainties of the conditions, ignoring the unknown value. Since the condition part holds a certainty less than 1.0 now, the certainty factor of the conclusion must be modified accordingly.

a1	a2	a3	Result	
0.9		1.0	0.9	min(0.9,1.0) * 0.9 = 0.81 = 0.8
0.6		1.0	1.0	min(0.6,1.0) * 1.0 = 0.6

And again, it can happen that a particular class is described by several vectors.

a1	a2	a3	Result		
0.9		1.0	0.9	R1	min(0.9,1.0) * 0.9 = 0.81 = 0.8
0.9		1.0	1.0	R1	min(0.9,1.0) * 1.0 = 0.9
0.6		1.0	1.0	R2	min(0.6,1.0) * 1.0 = 0.6

C(R1) = 0.0
C(R1/1) = 0.0 + [0.8 * [1.0 - 0.0] = 0.8
C(R1/2) = 0.8 + [0.9 * [1.0 - 0.8] = 0.98 = 0.9

NOTE: A 1-digit arithmetic with chopping is used.

We can derive the certainties within one table now. But we also have to consider the certainty with which this table had been invoked, a prior certainty for the particular table.

	a1	a2	a3	Result		
0.5 →	0.9		1.0	0.9	R1	C(R1) = 0.9
	0.9		1.0	1.0		
	0.6		1.0	1.0	R2	C(R2) = 0.6

Again, we multiply the appropriate certainties, resulting in

$C(R1) = 0.9 * 0.5 = 0.45 = 0.4$
$C(R2) = 0.6 * 0.5 = 0.3$

We have to consider all certainties in the path for a particular result to measure its overall quality in comparison to other results.

	a1	a2	a3	Result		
0.5 →	0.9		1.0	0.9	R1	C(R1) = 0.4
	0.9		1.0	1.0		
	0.6		1.0	1.0	R2	C(R2) = 0.3

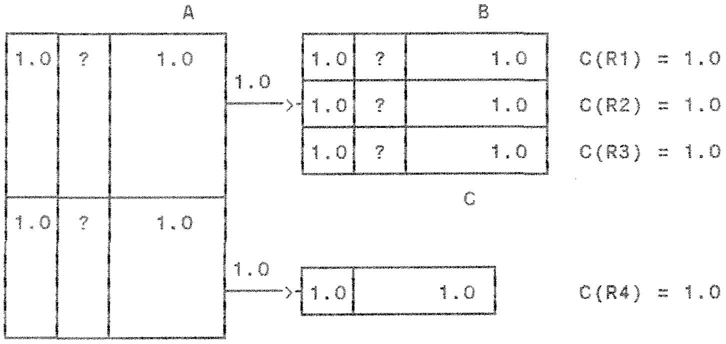
	b1	b2	Result		
0.5 →	1.0	1.0	1.0	R3	C(R3) = 0.5

For the example, result R3 would be more likely the solution than the results R1 or R2.

It was assumed that several descriptions for one class are independent of each other. This is of importance, since "the rules of certainty theory ... are strictly only applicable if the pieces of evidence are statistically independent." [9, p.405] Is this really true here? To answer this question we can again compare the description of a class with a rule consisting of several conditions and a conclusion. There might be several descriptions for the same class, or several rules with the same conclusion. Class descriptions are combined in sets here, which can be seen as a context. Several attributes with a number of values are defined within the table. The class descriptions depend on the same attributes, but not on each other, even if it might happen that a number of equal values are found in the conditional part of two vectors. The "rules" are all diagnosed at the same time, thereby reducing the set of applicable ones by rules with not fulfilled conditions.

So far, an answer UNKNOWN had no impact on the certainty of a particular conclusion. We only examined conditions with a certainty greater than zero, combined with predefined weights for the conclusions and simply ignored unknown parts of the conditions. Having a situation with all certainty factors equal 1.0, we will never have certainties different than zero (no information at all for a particular table) or 1.0 (at least one answer for every table along the path).

Example:

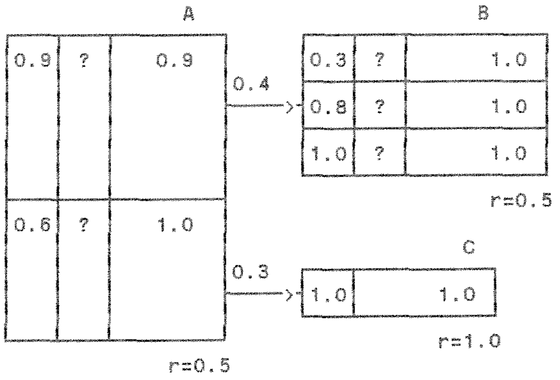


That all the certainty values happen to be 1.0 is totally correct; given the amount of information they are all possible. But it also seems to be appropriate to say that Result R4 is more likely to be the final result, since it includes more positive evidence than the other results, looking at the number of questions answered with UNKNOWN along the path. One way to incorporate this knowledge is to count the number of questions answered with UNKNOWN in a table, subtract this number from the total number of questions asked and compute a ratio, resulting in the percentage of questions NOT answered with UNKNOWN.

$$r = \frac{\Sigma \text{ questions} - \Sigma \text{ UNKNOWN}}{\Sigma \text{ questions}}$$

This ratio is then multiplied by the certainty factor of the results. The information about the amount of unanswered questions has also to be incorporated, if there are certainty values different than 1.0.

Example:



C(R1) = 0.3*0.5*0.4*1.0 = 0.06 = 0
C(R2) = 0.8*0.5*0.4*1.0 = 0.16 = 0.1
C(R3) = 1.0*0.5*0.4*1.0 = 0.2
C(R4) = 1.0*0.3*0.5*1.0 = 0.15 = 0.1

A path will be terminated if its combined certainty value drops under a predefined threshold. These thresholds are predefined for each table. Assuming that table A has a threshold of <0.4>, then table C will never be invoked, since the certainty value for the result "firing" the table is smaller than the threshold. Assuming that table A has a threshold of <0.3> and tables B and C both have a threshold of <0.2>, the only valid result will be R3.

3.11. Concluding other values

Depending on the amount of reduction of the number of questions asked, there will be additional knowledge in our knowledgebase that is bound to the particular result(s), and that can be given to the user if requested. The attempt is to exhaust all the information available. Values for attributes not covered by any question generated and/or values for attributes the user has no knowledge about (answer UNKNOWN) can be derived from the descriptions stored in the knowledgebase.

Except working with a MATCH control strategy the goal is to reduce the number of questions necessary to classify a particular instance. In the best case we only have to ask one question, which could for example result in the following:

Example:

```
type        size      location     creature    weight
cetacea     25 ft.    at sea       whale       1.0
cetacea     6 ft.     near coast   porpoise    1.0
cetacea     6 ft.     at sea       dolphin     1.0
fish        1 ft.     n.pacific    salmon      1.0
fish        6 ft.     at sea       shark       1.0

What is the length of the creature?  25 ft.

cetacea     25 ft.    at sea       whale       1.0

The creature is a whale for sure.
```

Is there a way to conclude the information about other attributes of the class and give this data to the user if requested? Of course: <type> is <cetacea> and <location> is <at sea>. In case that there are <Don't care> values embedded in the description, all of the possible values for the appropriate attribute are valid and have to be given to the user.

Example:

```
type        size      location     creature    weight
cetacea     25 ft.    *            whale       1.0
cetacea     6 ft.     near coast   porpoise    1.0
cetacea     6 ft.     at sea       dolphin     1.0
fish        1 ft.     n.pacific    salmon      1.0
fish        6 ft.     at sea       shark       1.0

What is the length of the creature?  25 ft.

cetacea     25 ft.    *            whale       1.0

The creature is a whale and has the following characteristics:
       type = cetacea
       size = 25 ft.
       location = at sea OR near coast OR northern pacific
```

Given a situation, in which several instances of the same class are left, we have to exhaust all information left.

Example:

type	size	location	creature	weight
cetacea	25 ft.	at sea	whale	1.0
cetacea	20 ft.	at sea	whale	0.9
cetacea	6 ft.	near coast	porpoise	1.0
cetacea	6 ft.	at sea	dolphin	1.0
fish	1 ft.	n.pacific	salmon	1.0
fish	6 ft.	at sea	shark	1.0

What is the length of the creature? UNKNOWN
Where does the creature live? at sea
What is the type of the creature? cetacea

cetacea	25 ft.	at sea	whale	1.0
cetacea	20 ft.	at sea	whale	0.9
cetacea	6 ft.	at sea	dolphin	1.0

C(whale) = 0.5
C(dolphin) = 0.6

The creature can be a dolphin (60% sure).

A dolphin has the following characteristics:

type = cetacea
size = 6 ft.
location = at sea

AND

The creature can be a whale (50% sure).

A whale has the following characteristics:

a) (1.0) type = cetacea
size = 25 ft.
location = at sea

b) (0.9) type = cetacea
size = 20 ft.
location = at sea

To conclude values of attributes belonging to parent tables in the hierarchy, the same steps have to be performed. We must keep track of the path we went down to a solution, for instance with a pointer back to (a) calling table(s).

3.12. Explaining the reasoning process

The ability to explain the reasoning used to ask a particular question is an important feature of an expert system. As can be seen easily this option is less powerful with control strategies like MATCH and Left-to-Right. Since the selection of questions asked is not very sophisticated in these cases the only explanation could be given regarding to the overall goal within the context of the particular table, eventually complemented by some already concluded results on higher hierarchy levels.

Examples:

a)

type	size	location	creature	weight
cetacea	25 ft.	at sea	whale	1.0
cetacea	6 ft.	near coast	porpoise	1.0
cetacea	6 ft.	at sea	dolphin	1.0
fish	1 ft.	n.pacific	salmon	1.0
fish	6 ft.	at sea	shark	1.0

Question: What is the type of the creature?
Explanation: This question serves to conclude the creature.

b)

A:

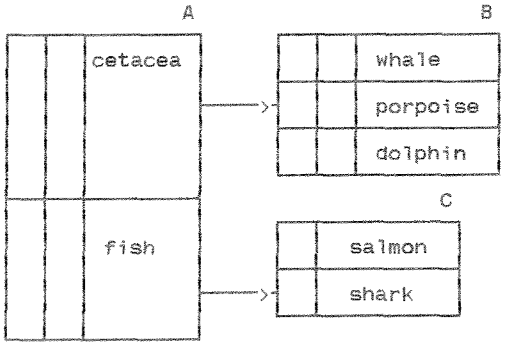
backbone	breathing	type	weight
yes	blow hole	cetacea	1.0
yes	gills	fish	1.0

B:

size	location	creature	weight
25 ft.	at sea	whale	1.0
6 ft.	near coast	porpoise	1.0
6 ft.	at sea	dolphin	1.0

C:

location	creature	weight
n.pacific	salmon	1.0
at sea	shark	1.0



Question: How does the creature breath?
Explanation: This question is asked in order to define the type of the creature.
This is necessary to conclude the creature.

Question: What is the length of the fish?
Explanation: It was concluded that the creature is a fish.
This question serves to conclude the creature.

Using a heuristic driven forward chaining, some more detailed explanations are possible, based on the philosophy of the heuristic itself.

Example: using Example a)

```
Question: What is the length of the creature?
Explanation: This question serves to conclude the creature.
             The length of the creature serves best to accomplish this goal,
             since the creatures differ very much in their length.
```

Given a more complex hierarchy and additional features like answers UNKNOWN, several tables with the same parent etc., the explanations will be more sophisticated.

3.13. Incorporating metaknowledge

As described before there are different ways of controlling the reasoning process, depending on the contents of the knowledge description and on the task at hand. The decision which one to use can only be made according to this information, thus it is not possible to 'hardcode' them.

One particular table in the system can have an order dependency, whereas others don't have this constraint. Hence, for the first one we have to use either MATCH or Left-to-right, the other ones can more efficiently be explored using heuristic methods.

"Consequently, there are good reasons for making control knowledge explicit. ... metarules, which are invoked as part of the conflict resolution strategy, can capture and implement strategic knowledge about a domain." [9, p.435]. "Meta-rules are distinguished from ordinary rules in that their role is to *direct* the reasoning required to solve a problem, rather than to actually *perform* that reasoning." [10, p.147].

In our case, the following meta-rules could for example be stated:

```
IF there is a favored strategy bound to the table THEN use this
                                                  ELSE use Heuristic

IF there is an order within the table THEN use Left-to-right or Match
                                      ELSE use Heuristic

If the user answers with UNKNOWN very often
THEN ignore the ASKFIRST flags and branch to other tables without asking.

If there are multiple instances of one class with different weights
THEN first try to reduce the set to get one result class and
THEN try to reduce the set to get one result class with a unique weight.

If there are other classes with the same parent left
THEN use MATCH
```

The following example illustrates how metarules are used within HICLASS to decide which control strategy has to be applied for a particular table:

```
strat:=favored_strategy;  {use user's favored strategy}
if strat=0 then strat:=3; {if no strategy is favored use heuristic}
if (predefined)and(strat=3) then strat:=2; {cannot use heuristic if predefined order}
if sib then strat:=1; {there are active siblings in the hierarchy}

case strat of
        1: match(table,qb);
        2: left_right(table,qb);
        3: heuristic(table,qb);
end;
```

3.14. Learning

In our classification system it might happen that the incorporated knowledge is incomplete, that there are special cases which have not been considered before. For example, the user could "see" a value for an attribute that is not incorporated in the multiple choice presented by the system. Rather than simply stating that the system cannot classify this instance, new knowledge could be acquired from the user and the system could try to proceed with this new information, at the same time memorizing the specific constellation for maintenance purposes.

Example:

```
(interval size = 0% = unique values)

type        size     location    creature   weight
cetacea     25 ft.   at sea      whale      1.0
cetacea     6 ft.    near coast  porpoise   1.0
fish        1 ft.    n.pacific   salmon     1.0


What is the class of the creature?   cetacea

cetacea     25 ft.   at sea      whale      1.0
cetacea     6 ft.    near coast  porpoise   1.0


What is the length of the creature?  23 ft.

NOTE: The user enters a length that is inconsistent with the given knowledge.
      The system tries now to come up with a result anyway.


Where does the creature live?   at sea

Result: It is likely that the creature is a whale.
        A porpoise normally has a length of 25 ft.
        Please check the length you entered (23 ft.).
        If you are sure about the length, then enter YES.
```

If the user confirms the new knowledge, it will be stored and can be checked in a maintenance run performed by a human expert. If he/she can confirm this new instance as belonging to the class detected by the system, the description for this class has to be updated. Otherwise, if the new information is inconsistent with the class, there might be a complete new class to add.

3.15. Global attributes

If there are attributes in different tables that are literally the same and have the same domain of possible answer values, which will most likely be the case if there are multiple tables with the same parent in parallel (as described in 3.8.), then it should not be necessary for the user to answer the same question several times. Attributes like this will be marked "global" and a question will only be generated for the first time the attribute is encountered. The attribute name together with the user's answer will be stored. Then, given an attribute marked as global is invoked again, the answer will be taken from this internal list rather than asking the user. Of course, the value should be included in the set of possible answer values of the current attribute, the common domain condition as mentioned above has to hold, this is one of the consistency constraints of the system (the topic will further be discussed in the next section).

3.16. Checking the consistency of the system

There are two major consistency problems within the hierarchy of tables. First, all values of a global attribute have to be defined within the same domain for all occurrences of this global attribute. Second, the domain of result values of a table that is invoked from another table to provide a value for an attribute of the calling table has to be the same as the domain for the attribute of the calling table. If this constraint does not hold, contradictions would be introduced. In order to check the whole system, a consistency test routine could be introduced, checking all global attributes and the interfaces of chained tables.

3.17. HICLASS and the rest of the world

So far, HICLASS has been thought of as an independent program solving the task of hierarchical classification. It might be possible though to embed the program in a bigger system to solve one part of a task. We're talking the use of HICLASS as a building block for more complex problem-solvers or architectures. In chapter 5 it will be shown that HICLASS indeed fulfills conditions to implement a generic task. Hence, extended by appropriate interfaces to other building blocks, HICLASS could serve as part of a more complex system.

Additionally, the interaction with the world could be performed in different ways than described so far (questions are generated and a user answers while choosing a multiple choice answer or typing a value). The "questions" could be calls to other programs or real world processes to determine values which are sent back to HICLASS to proceed in the classification. Thus, no human user has to be involved anymore; HICLASS would serve as the control unit of an automated process.

3.18. Several paths - which one to follow?

It can happen that the system has to maintain several proper paths at the same time, each of them carrying a certainty value with it. The question is in which order we proceed in the reasoning process. There are basically two ways of dealing with this the problem.

We could apply a depth-first search, which means that we would follow the leftmost path until a leaf table is solved, or until the path terminates because the certainty of a table becomes smaller than the threshold bound to it. If there are other paths left, then again the leftmost of those will be followed first.

Or, we could apply a best-first search. If there are several paths, we would further explore the path with the highest certainty value until a leaf table is solved or the certainty value of the path becomes smaller than the *maximum* certainty value of the other paths under consideration. If a leaf table is solved, a solution can be given to the user and a question can be generated to inquire if the user likes the reasoning process to continue (and to try to come up with another result, which will have a *smaller* certainty than the first result). If the user agrees, the same process starts over again with the remaining paths. In the case that the certainty value of a path becomes smaller than the value of another path currently defined, paths will be switched. In other words, the current path will be disabled and we continue with the most promising path as before. This algorithm assures that we are not wasting time while exploring paths with a very little certainty of leading to the result. Of course, we cannot predict future events, and the least promising path might succeed in the end, a problem that is common to most of the search techniques developed for AI applications and that is tackled in one way or another by the more sophisticated ones.

3.19. Additional features

3.19.1. Entering initial data

It was stated before that in general there will be no initial data given. But one also might think of an application in which some data can be provided by the user before the system starts its reasoning process. MYCIN uses a "tabular representation", that can initially be filled with some values from a patient's record. This concept seems useless for a classification system. As described in [3, p.62] there is "the attendant risk of asking for information that would not actually be used in some cases." For our special application, "some cases" would be "almost all". Since we are moving within a hierarchy, only the questions in the first table invoked are relevant for sure. On the next level in the hierarchy, only a subset of all possible questions is valid anymore, since we "close" whole branches of the decision tree.

3.19.2. Saving the system state in case of an interruption

A useful option to add would be that in case of an interruption of the session due to a number of reasons, one is the need of the user to get more information before being able to proceed, it should not be necessary to enter the already given information again. This could be achieved by storing the state of the system in a way that information about all questions and answers so far are saved and then used as an automatic input when running the system again, such that the state of the system can easily be restored.

3.19.3. Using information from terminated paths

As stated above, a path will be terminated if its combined certainty value drops under a predefined threshold. If all paths terminate and none of the paths reached a tip level, thus a final result, the logical answer of the system would be to state "Due to insufficient information I have no advice", period. But maybe at least some confidence about subgoals along the paths was accumulated. It could for instance be sure that the creature is a <fish>. The system should be able to at least give this information to the user.

3.19.4. Numerical input

So far, only symbols in the shape of strings are allowed as an input to a specific question, given in a multiple choice to the user. It might also be useful to have a numerical input option. For the examples above, *size* could be inquired as a number rather than to give a predefined choice. The problem is how to interpret a numerical input. Is it useful to work with intervals? How can this be incorporated in a rule?

Example:

The following instances are used to set up the table:

26 ft.	whale
6 ft.	dolphin
2 ft.	salmon

Up to now, only the three predefined sizes were allowed to be chosen by the user. A <whale> was <26 ft.> long, a <dolphin> had to be <6 ft.> and a <salmon> <2 ft.> in order to be recognized by the system. Working with the table (that is embedded in a hierarchy), we should be able to use the dimensions given to derive intervals in order to allow a more flexible input. In order to calculate the left border of an interval, we subtract the next smaller value from the value under consideration, then multiply the difference by 0.5, and subtract the result from the current value. To calculate a right border, we take the next higher value, subtract the current value, multiply the result by 0.5, and add the result to the current value. If a value is the smallest in the set, no left border needs to be calculated; the same is true with the highest value and the right border.

```
<whale>:   26 - 6  = 20
           20 * .5 = 10
           26 - 10 = 16

<dolphin>:  6 - 2  = 4
            4 * .5 = 2
            6 - 2  = 4

            26 - 6  = 20
            20 * .5 = 10
            6 + 10  = 16

<salmon>:   6 - 2  = 4
            4 * .5 = 2
            2 + 2  = 4
```

Resulting in:

```
if size ≥ 16 then whale
if 4 ≤ size < 16 then dolphin
if size < 4 then salmon
```

If the user is prompted to identify the size of the creature to be classified, the following classification is possible now:

```
What is the length of the creature?   23 ft.
Result: The creature is a whale.
```

The idea described is of course very general and might not serve all situations. But, values could be marked as unique, therefore not extendable to an interval, or the interval size could be limited. So far, a full half range between values was used (50%). In HICLASS, an interval size has to be defined for each table, ranging from

0% = unique values   to
50% = full half range

The left border of an interval is now calculated by subtracting the next smaller value from the value under consideration. Then, the result is weakened by the predefined interval size (a value of 25 defines that the left interval border will only be 25% of the full distance between the two values away from the current value). Again, the result is subtracted from the current value to get the left border of the interval. Right borders are calculated accordingly.

```
interval size = 25%

<whale>:     26 - 6     = 20
             20 * .25   = 5
             26 - 2     = 21

<dolphin>:    6 - 2   = 4
              4 * .25 = 1
              6 - 1   = 5

              26 - 6   = 20
              20 * .25 = 5
              6 + 5    = 11

<salmon>:     6 - 2   = 4
              4 * .25 = 1
              2 + 1   = 3


if size ≥ 21 then whale
if 5 ≤ size < 11 then dolphin
if size < 3 then salmon
```

If there are several descriptions for one class including numerical values and carrying the same weight, the system so far is not able to combine those descriptions.

Example:

26 ft.	whale	1.0
24 ft.	whale	1.0
6 ft.	dolphin	1.0
2 ft.	salmon	1.0

The algorithm does not care about relationships between values found in descriptions for the same class. Given a full half range, it would come up with:

```
if size ≥ 25 then whale
if 16 ≤ size < 25 then whale
if 4  ≤ size < 16 then dolphin
if size < 4 then salmon
```

This result is logically completely correct and the system would have no problem classifying a <whale>, that is <20 ft.> long. For compactness and explanation reasons though it would be better to combine the two descriptions for <whale> to

```
if size ≥ 16 then whale
```

3.20. HICLASS - an expert system shell

The term "user" so far was referring to a person using a ready made expert system to solve a problem. But who is actually creating the system? One could refer to this person as a "knowledge engineer". His/her task is to acquire the knowledge, organize it and encode it together with all the necessary control structures. As already mentioned, HICLASS is a tool for hierarchical classification. It provides a knowledge representation and control structures suited for this purpose. The wheel does not have to be reinvented every time a knowledge engineer attempts to build a new expert system. HICLASS has to be thought of as an "empty" system, capable of solving the task of hierarchical classification if fed with tables containing all the necessary information. Therefore, it is an *expert system shell*.

"Shells are intended to allow non-programmers to take advantage of the efforts of programmers who have solved a problem similar to their own." [10, p.339].

Thus, the knowledge engineer does not even have to know a programming language in order to build an expert system. The only thing needed is a tool for creating and maintaining tables, together with other control parameters. This tool is the editor *HIEDIT*, which will be described in chapter 4.1.

Every attempt to provide a lot of flexibility goes hand in hand with a problem to provide special features which might be needed for a particular application. Hence, for a *very* special classification task, HICLASS might be useless, since the knowledge engineer's chances to change the behavior of the system are somewhat limited. One example is the handling of uncertainty. "...most if not all of these shells are either inconsistent with probability theory or have properties that are simply hard to analyze. Although a pragmatic justification can often be given for a particular treatment of uncertainty in the context of a particular application (for example, Shortliffe's rationale for using certainty factors in MYCIN), it is a much more dangerous enterprise to adopt such a treatment simply because it comes with the shell one is using." [10, p.342].

Jackson [10, p.342] also mentioned some advantages of expert system shells like the fact that they are widely available for smaller machines, that because they are mostly written in "non-AI" languages they can aid portability and interfacing to other software and finally, that they are inexpensive compared to especially designed systems. It really depends on the application if the use of an expert system shell can be recommended or not.

4. The implementation of the HICLASS system

The HICLASS system is divided into two major parts: HIEDIT, the table editor program, and HICLASS, the application program performing hierarchical classification based on tables chained together in a hierarchy. Approximately 16.000 lines of TURBO PASCAL 6.0 code were written. A description of the software engineering techniques used can be found in Appendix B, section 7. Appendix C includes a description of almost all modules designed for the project.

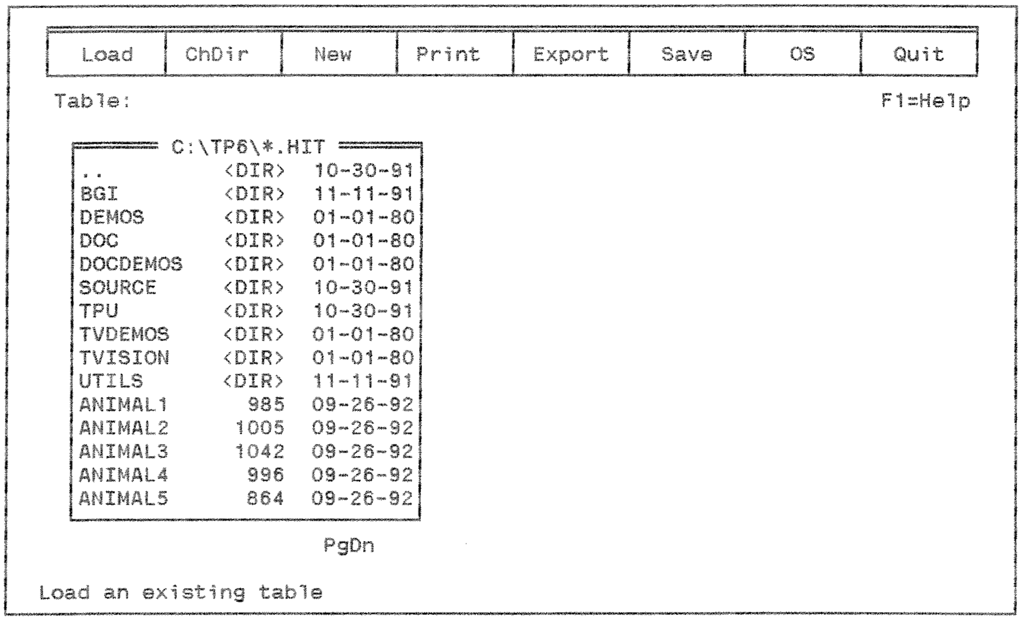
4.1. HIEDIT

As discussed in 3.20., a tool is needed to create and maintain the tables used in HICLASS as well as in HIHYPO. This tool is HIEDIT, a special table editor program. HIEDIT supports the whole process from defining attributes for a table, defining a domain of values for each attribute, stating examples for descriptions of classes and adding a number of control parameters attached to each table. Additionally, an inductive learning feature to create a distinction-oriented set of descriptions for HIHYPO is embedded. The program is completely pull down menu driven, values and examples can easily be entered and changed in a spreadsheet. There are four screens defined, the user can move between the screens with F9/F10.

```
FILES - DEFINITIONS - EXAMPLES - SPECIAL

<------------ F9 <--> F10 ------------->
```

A context-sensitive help is provided to explain features of the program. Additionally, error and other messages help to guide the process of defining a table.

4.1.1. FILES screen



```
Table:                                                          F1=Help

     ═══════ C:\TP6\*.HIT ═══════
     ..          <DIR>   10-30-91
     BGI         <DIR>   11-11-91
     DEMOS       <DIR>   01-01-80
     DOC         <DIR>   01-01-80
     DOCDEMOS    <DIR>   01-01-80
     SOURCE      <DIR>   10-30-91
     TPU         <DIR>   10-30-91
     TVDEMOS     <DIR>   01-01-80
     TVISION     <DIR>   01-01-80
     UTILS       <DIR>   11-11-91
     ANIMAL1       985   09-26-92
     ANIMAL2      1005   09-26-92
     ANIMAL3      1042   09-26-92
     ANIMAL4       996   09-26-92
     ANIMAL5       864   09-26-92

                          PgDn

   Load an existing table
```

Figure 4.1.1.1. FILES screen in HIEDIT

Explanation of features:

Load = Load a table from disk. Only directories and table files (*.HIT) can be selected from a pull down menu.

Chdir = Change the current directory. Can be used to store tables in a different directory.

New = Start a new table.

Print = Print the content of a table (not implemented yet)

Export = Export the content of a table as a text file or a file with a format compatible with programs like LOTUS 1-2-3. (not implemented yet)

Save = Save a table to disk.

OS = Access to DOS without quitting HIEDIT.

Quit = Quit HIEDIT.

4.1.2. DEFINITIONS screen

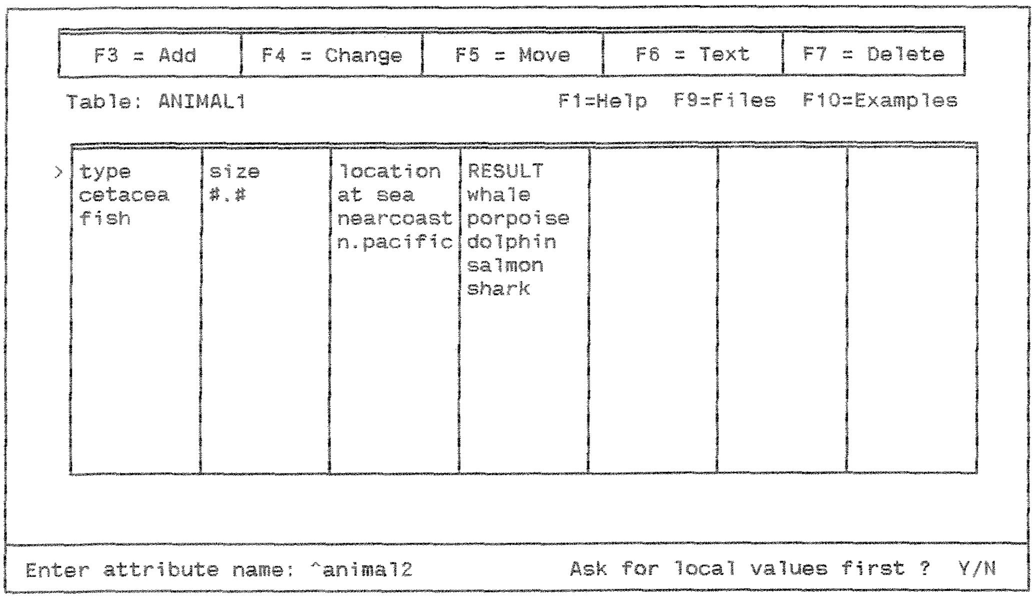


Table: ANIMAL1

type	size	location	RESULT			
cetacea	#.#	at sea	whale			
fish		nearcoast	porpoise			
		n.pacific	dolphin			
			salmon			
			shark			

Enter attribute name: ^animal2   Ask for local values first ? Y/N

Figure 4.1.2.1. DEFINITIONS screen in HIEDIT

Explanation of features:
(depending on the position in the spreadsheet)

Add = Add up to 12 attributes.
If the new attribute name starts with a "^", then the system will inquire if the introduced call to another table should have an ASKFIRST option or not.
If the name starts with "!", then a global attribute will be defined.
The attribute RESULT is predefined.

Add up to 26 values per attribute in a spreadsheet.
"#.#" denotes that the value is numeric (default is numeric).

Change = Change the name of an attribute.
Change the name of a value.

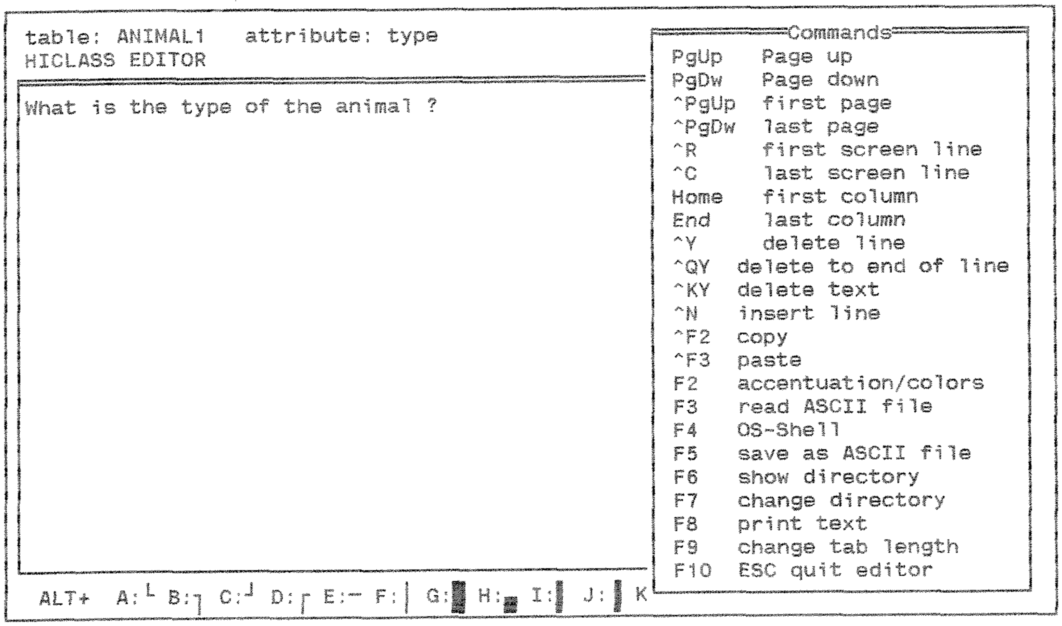
Move = Move an attribute to another location.
Move a value to another location.

Text = Invoke a full screen editor for editing up to 20 lines of text to be attached to an attribute or to values of RESULT.

Edit one line of text for a value.

Delete = Delete an attribute.
Delete a value.

Some of the operations will have an effect on possibly already defined examples. The effects will be propagated to all example definitions.



```
table: ANIMAL1   attribute: type
HICLASS EDITOR

What is the type of the animal ?

Commands
PgUp    Page up
PgDw    Page down
^PgUp   first page
^PgDw   last page
^R      first screen line
^C      last screen line
Home    first column
End     last column
^Y      delete line
^QY   delete to end of line
^KY   delete text
^N    insert line
^F2   copy
^F3   paste
F2    accentuation/colors
F3    read ASCII file
F4    OS-Shell
F5    save as ASCII file
F6    show directory
F7    change directory
F8    print text
F9    change tab length
F10   ESC quit editor

ALT+  A:└ B:┐ C:┘ D:┌ E:─ F:│ G:█ H:▄ I:▌ J:▐ K
```

Figure 4.1.2.2. Editor within ATTRIBUTES screen

The full screen editor has a lot of features common to ASCII editors. It supports a word-wrap function, includes an easy editing of graphic elements and is able to display color.

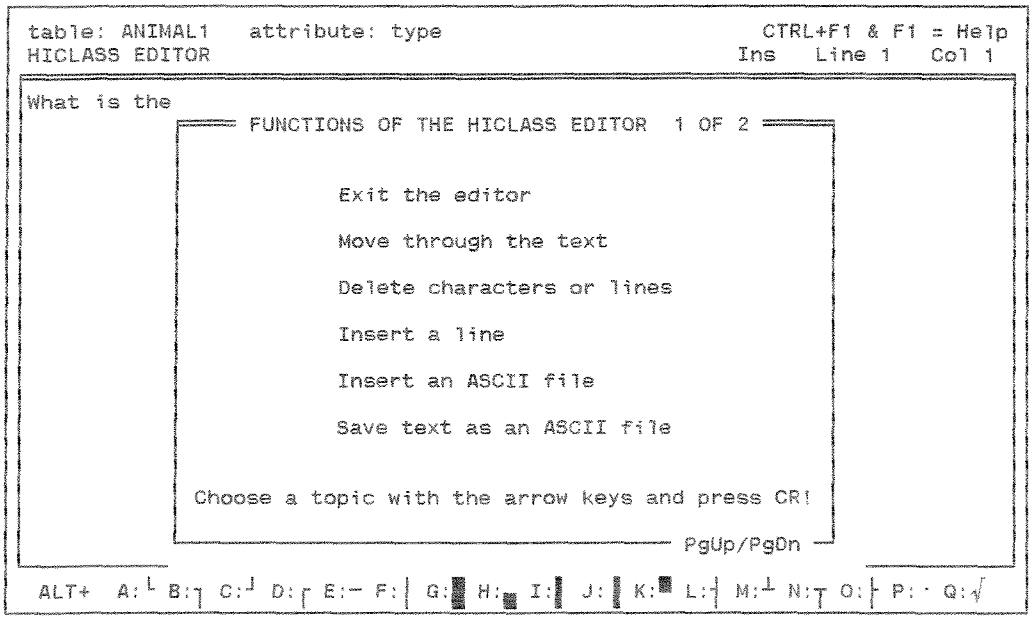


table: ANIMAL1   attribute: type   CTRL+F1 & F1 = Help

HICLASS EDITOR   Ins   Line 1   Col 1

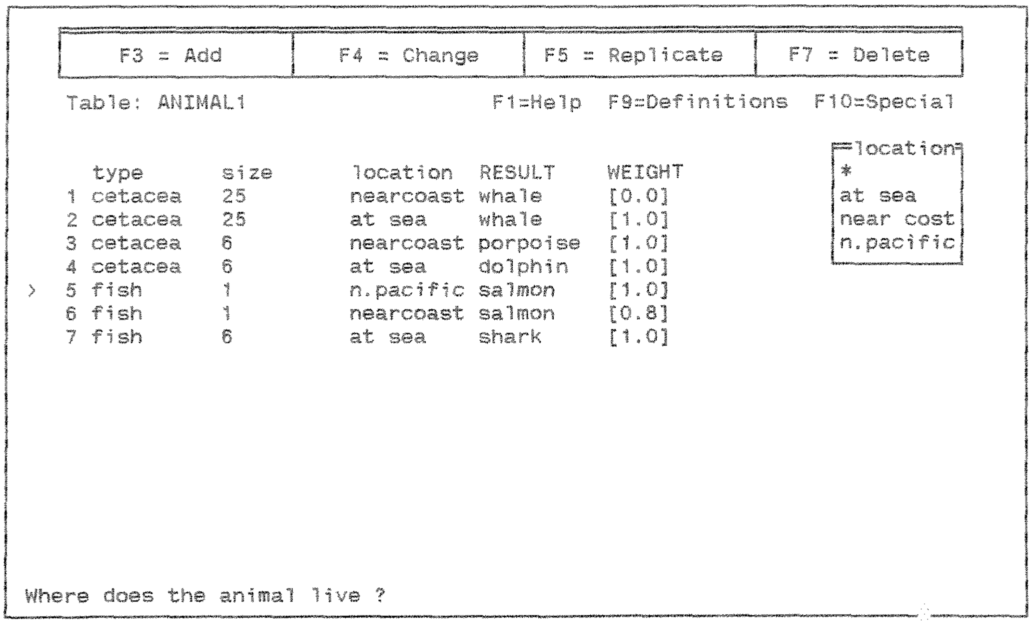
What is the

FUNCTIONS OF THE HICLASS EDITOR 1 OF 2

Choose a topic with the arrow keys and press CR!

Figure 4.1.2.3. Context-sensitive help within the editor

4.1.3. EXAMPLES screen



```
Table: ANIMAL1

    type      size      location  RESULT    WEIGHT
  1 cetacea   25        nearcoast whale     [0.0]
  2 cetacea   25        at sea    whale     [1.0]
  3 cetacea   6         nearcoast porpoise  [1.0]
  4 cetacea   6         at sea    dolphin   [1.0]
> 5 fish      1         n.pacific salmon    [1.0]
  6 fish      1         nearcoast salmon    [0.8]
  7 fish      6         at sea    shark     [1.0]

Where does the animal live ?
```

Figure 4.1.3.1. EXAMPLES screen in HIEDIT

Explanation of features:

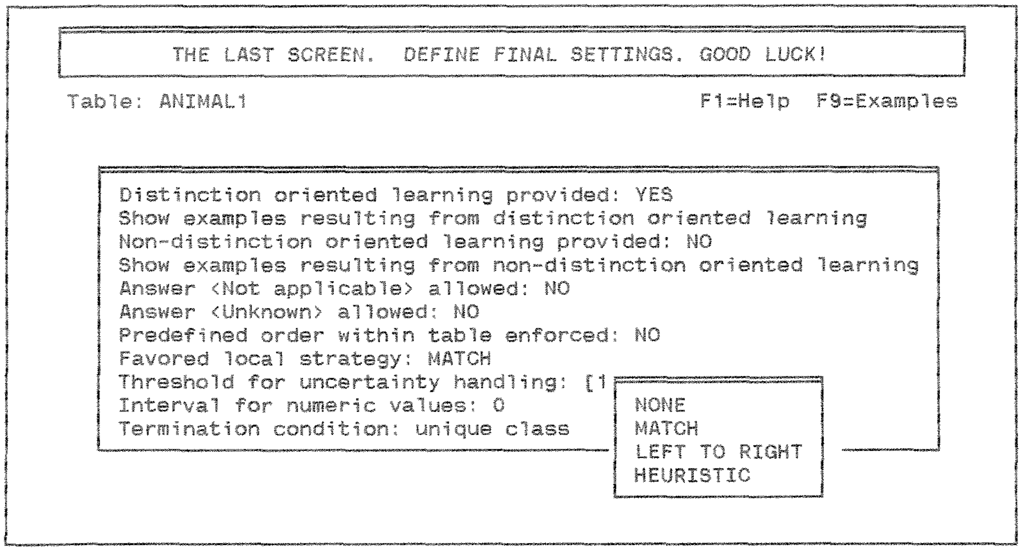
Add = Add up to 255 examples in a spreadsheet.
The values for attributes can either be chosen from a pull down menu or a numeric value can be defined.
"*" denotes <Don't care>.
Each example has a weight attached to it:
[0.1..1.0] = degree of confidence
[0.0] = negative example

Change = Change a value for an example.

Replicate = Create a new example identical to the current one.

Delete = Delete an example.

4.1.4. SPECIAL screen



```
        THE LAST SCREEN.  DEFINE FINAL SETTINGS. GOOD LUCK!
Table: ANIMAL1                                      F1=Help  F9=Examples

   Distinction oriented learning provided: YES
   Show examples resulting from distinction oriented learning
   Non-distinction oriented learning provided: NO
   Show examples resulting from non-distinction oriented learning
   Answer <Not applicable> allowed: NO
   Answer <Unknown> allowed: NO
   Predefined order within table enforced: NO
   Favored local strategy: MATCH
   Threshold for uncertainty handling: [1
   Interval for numeric values: 0
   Termination condition: unique class
```

Figure 4.1.4.1. SPECIAL screen in HIEDIT

Explanation of features:

Distinction oriented learning provided:
- YES/NO (default=NO)

Show examples resulting from distinction oriented learning
- create and show distinction oriented set of examples

Non-distinction oriented learning provided:
- YES/NO (default=NO)

Show examples resulting from non-distinction oriented learning
- create and show non-redundant set of examples
  (not implemented yet)

Answer <Not applicable> allowed
- YES/NO (default=NO)

Answer <Unknown> allowed
- YES/NO (default=NO)

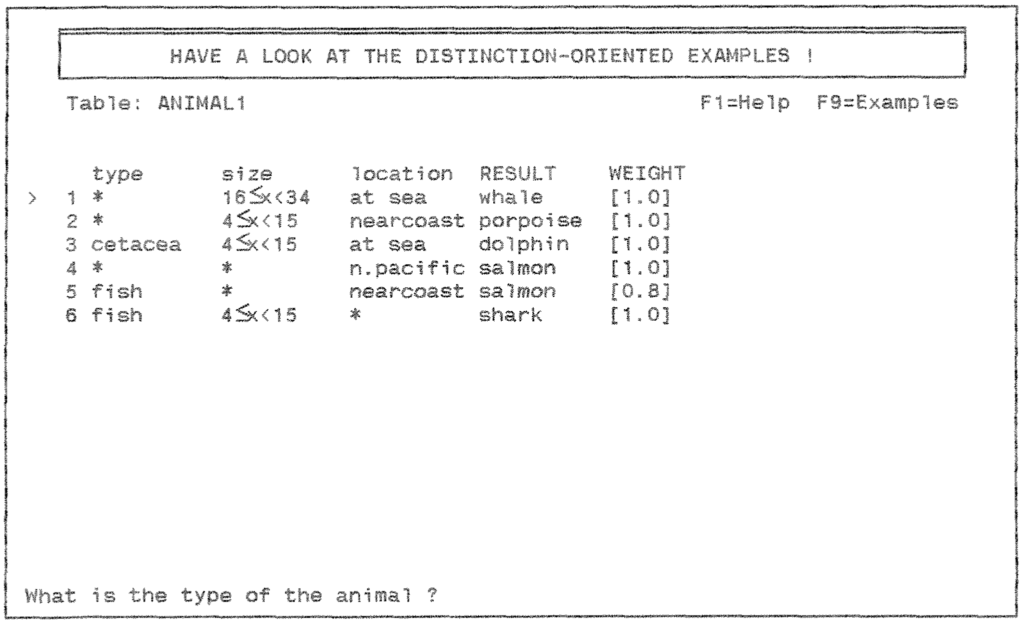
Predefined order within table enforced
- YES/NO (default=NO)

Favored local strategy:
- NONE / MATCH / LEFT TO RIGHT / HEURISTIC
  (default=NONE - no *favored* strategy)

Threshold for uncertainty handling:
·[0.1..1.0] (default=1.0)

Interval for numeric values:
·[0..50] (default=0 = unique values)

Termination condition:
·<unique class> OR
<unique class AND unique weight>
(default=<unique class AND unique weight>)



```
          HAVE A LOOK AT THE DISTINCTION-ORIENTED EXAMPLES !

   Table: ANIMAL1                                   F1=Help  F9=Examples


      type      size      location   RESULT    WEIGHT
 >  1 *         16≤x<34   at sea     whale     [1.0]
    2 *         4≤x<15    nearcoast  porpoise  [1.0]
    3 cetacea   4≤x<15    at sea     dolphin   [1.0]
    4 *         *         n.pacific  salmon    [1.0]
    5 fish      *         nearcoast  salmon    [0.8]
    6 fish      4≤x<15    *          shark     [1.0]









 What is the type of the animal ?
```

Figure 4.1.4.2. Set of distinction-oriented examples

4.2. HICLASS

The program HICLASS is based on the theoretic discussion made earlier. Due to the high implementational effort, not all of the features described have already been implemented yet. Nevertheless, the program is solving the task of hierarchical classification using tables created with HIEDIT successfully, providing all of the major features covered.

The features not implemented yet are:

- using information from terminated paths
- explaining the reasoning process
- dealing with new information
- checking the consistency of the system
- interface to other programs

Even if these features are not implemented yet, they were thought of; and the data structures as well as the program structure are designed to allow additions.

The global control strategy implemented uses a depth-first search. Every time a table is called either from a result or from an attribute of another table, the file containing this table is loaded and some initialization steps are performed (e.g. applying the metarules to decide upon the control strategy to be used). Then, the local control strategy starts working. If values for an attribute have to be provided by another table, then this table is called using a nested call to the same procedure used for the original table (the main procedure *meta* is called recursively every time a new table is invoked). Depending on the control strategy chosen, the table content is reduced during the question/answer dialogue or (in the case of MATCH) at the end of the dialogue. Checks for unique results (or unique results AND unique weights) are performed to check the termination condition. If a table is solved successfully, certainties for the results are calculated and these results are given to the user if necessary. In the case of multiple table calls as the result of a table, these calls are placed in a control list and the resulting paths are invoked from left to right as siblings. If a table is called to provide values for an attribute, then the particular subtree is solved first, following the same steps as in the "main" tree. So far, it is only possible, that ONE table provides values for an attribute of a calling table; it doesn't matter if this is the table invoked first or another table of the subtree. After a table is solved completely, it is disposed. This and the fact that tables are only invoked if necessary allows to build very large systems without running out of memory. Tables can be disposed since all values for global attributes are stored in a separate list, and the whole dialogue including questions, answers, results and certainties is documented in a history list. Of course, values as well as results of a table can be numeric; numeric values are processed within the boundaries of the predefined interval range.

The basic description given above roughly outlines "the way it works". It will be supplemented by additional information given in the next sections.

4.2.1. The example

In order to illustrate the performance of HICLASS, a special example was created showing as many of the features as possible. The zoological content of the example is mainly based on [4].

4.2.1.1. Hierarchy structure of the example

The following figure shows the hierarchy structure of the example. Connections marked with '<' and '>' denote that values for an attribute are provided by another table. The table NOTCETAC is a dummy table and is inserted in order to call several tables from just one result (there is only one attribute defined in this table, the result; the table acts as a routing device). Tables can be used several times in the hierarchy (an example is table SIZE). Another special feature is shown with table ANIMAL1. This table either provides values for ANIMAL or calls NOFISH to provide these values (the decision is made according to the certainty of the results of ANIMAL1; if the minimal certainty of the "real" results providing values is smaller than the minimal certainty of all results which are calls, then the calls are made; otherwise the results are given back to the calling table).

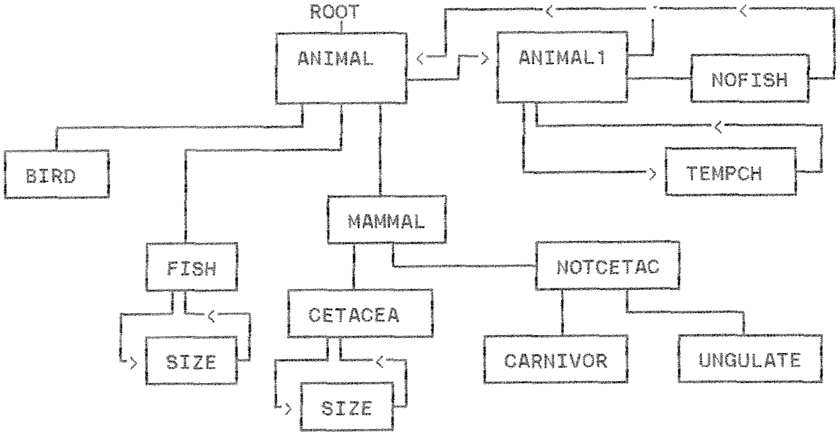


Figure 4.2.1.1.1. Hierarchy structure of the example

4.2.1.2. Content of the tables

Calls to other tables are made using their filename. A '^' denotes that ASKFIRST=false, whereas a '~' stands for ASKFIRST=true, and the user has to be asked before a table is invoked. An '!' at the beginning of an attribute name means that this attribute is defined globally.

There are control parameters bound to each table, among others:

U=UNKNOWN allowed
N=NOT APPLICABLE allowed
T=Threshold
F=Favored strategy
I=Numeric Interval

The table EXAMPLE1 provides some introductory comments for the example.

EXAMPLE1 (U=no, N=no, T=1.0, F=none, I=0)

	goon	more	RESULT	WEIGHT
1	goon	goon	^animal	[1.0]

ANIMAL (U=yes, N=no, T=0.1, F=none, I=0)

	^animal1	RESULT	WEIGHT
1	mammal	^mammal	[1.0]
2	fish	^fish	[1.0]
3	bird	^bird	[1.0]

ANIMAL1 (U=yes, N=no, T=0.1, F=none, I=0)

	~tempch	breathing	RESULT	WEIGHT
1	variable	gills	fish	[1.0]
2	constant	lungs	^nofish	[1.0]

TEMPCH (U=yes, N=no, T=0.1, F=none, I=0)

	tempch	RESULT	WEIGHT
1	yes	variable	[1.0]
2	no	constant	[1.0]

NOFISH (U=yes, N=no, T=0.1, F=left-to-right, I=0)

	bodytemp	reproduct	RESULT	WEIGHT
1	107	eggs	bird	[1.0]
2	98	uterus	mammal	[1.0]

MAMMAL (U=yes, N=yes, T=0.5, F=none, I=0)

	skin	RESULT	WEIGHT
1	naked	^cetacea	[1.0]
2	hair	^notcetac	[0.8]

CETACEA (U=yes, N=no, T=0.1, F=none, I=50)

	~size	location	RESULT	WEIGHT
1	6	nearcoast	porpoise	[1.0]
2	20	*	whale	[1.0]

SIZE (U=yes, N=no, T=0.1, F=none, I=0)

	!size	RESULT	WEIGHT
1	small	1	[1.0]
2	medium	6	[1.0]
3	big	21	[1.0]
4	very big	26	[1.0]

NOTCETAC (U=no, N=no, T=0.1, F=none, I=0)

	RESULT	WEIGHT
1	^carnivor	[1.0]
2	^ungulate	[1.0]

CARNIVOR (U=yes, N=yes, T=0.1, F=none, I=0)

	!hoofs	!darkspot	!blstrip	RESULT	WEIGHT
1	no	yes	no	cheetah	[1.0]
2	no	no	yes	tiger	[1.0]

UNGULATE (U=yes, N=yes, T=0.1, F=none, I=0)

	!hoofs	!longneck	!longlegs	!darkspot	!blstrip	RESULT	WEIGHT
1	yes	yes	yes	yes	no	giraffe	[1.0]
2	yes	no	no	no	yes	zebra	[1.0]

BIRD (U=yes, N=yes, T=0.1, F=none, I=0)

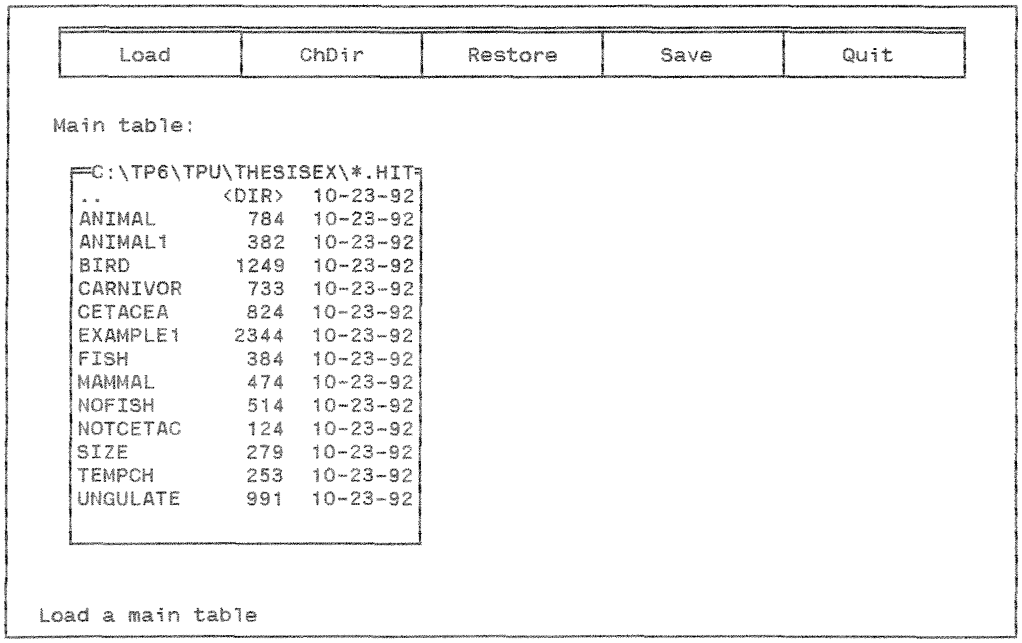
	canfly	!longneck	!longlegs	!color	canswim	RESULT	WEIGHT
1	no	yes	yes	b&w	no	ostrich	[1.0]
2	no	no	no	b&w	yes	penguin	[1.0]
3	yes	no	no	white	no	albatross	[1.0]

FISH (U=yes, N=yes, T=0.1, F=none, I=50)

	^size	RESULT	WEIGHT
1	1	salmon	[1.0]
2	10	shark	[1.0]

4.2.1.3. The FILES screen

The first screen of HICLASS is concerned about directories and files. A table can be chosen with LOAD, this table represents the root table of the hierarchy. The current directory can be changed with CHDIR. SAVE saves a session report, and RESTORE loads a session report file. After RESTORE, the system takes the information in the file as a "background" input and proceeds as if the user would have been questioned, taking all the former answers stored in the session report as answers for the current session. This option can be used after a session was interrupted to restore the former system state. QUIT terminates HICLASS.



Main table:

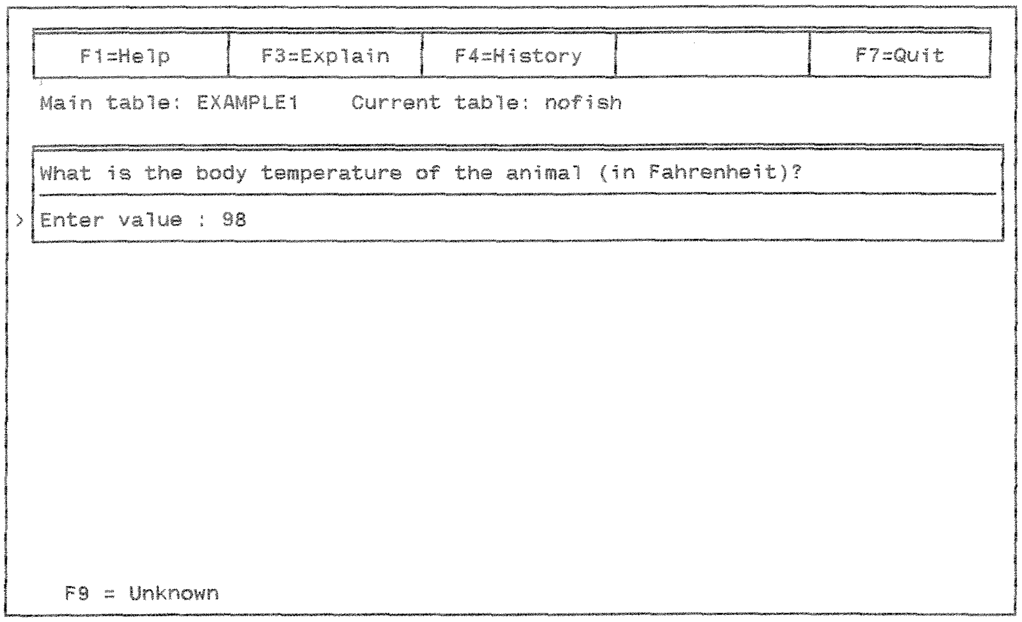
```
C:\TP6\TPU\THESISEX\*.HIT
..          <DIR>   10-23-92
ANIMAL        784   10-23-92
ANIMAL1       382   10-23-92
BIRD         1249   10-23-92
CARNIVOR      733   10-23-92
CETACEA       824   10-23-92
EXAMPLE1     2344   10-23-92
FISH          384   10-23-92
MAMMAL        474   10-23-92
NOFISH        514   10-23-92
NOTCETAC      124   10-23-92
SIZE          279   10-23-92
TEMPCH        253   10-23-92
UNGULATE      991   10-23-92
```

Load a main table

Figure 4.2.1.3.1. The FILES screen in HICLASS

4.2.1.4. Questioning the user

The system generates questions using the text screens and the text for values defined in HIEDIT. The user can browse through the question text (up to 20 lines) and then either choose an answer in a multiple choice fashion, or as shown in figure 4.2.1.4.1. enter a numeric value. F1 provides a context sensitive help, and F3 explains the reasoning process (not implemented yet). F7 allows the user to interrupt the current session and he/she can save the state of the system in the FILES screen. If the answer UNKNOWN is allowed, then F9 provides this answer, the same is true for F10 and NOT APPLICABLE (these choices are only given in appropriate situations).



Main table: EXAMPLE1   Current table: nofish

What is the body temperature of the animal (in Fahrenheit)?

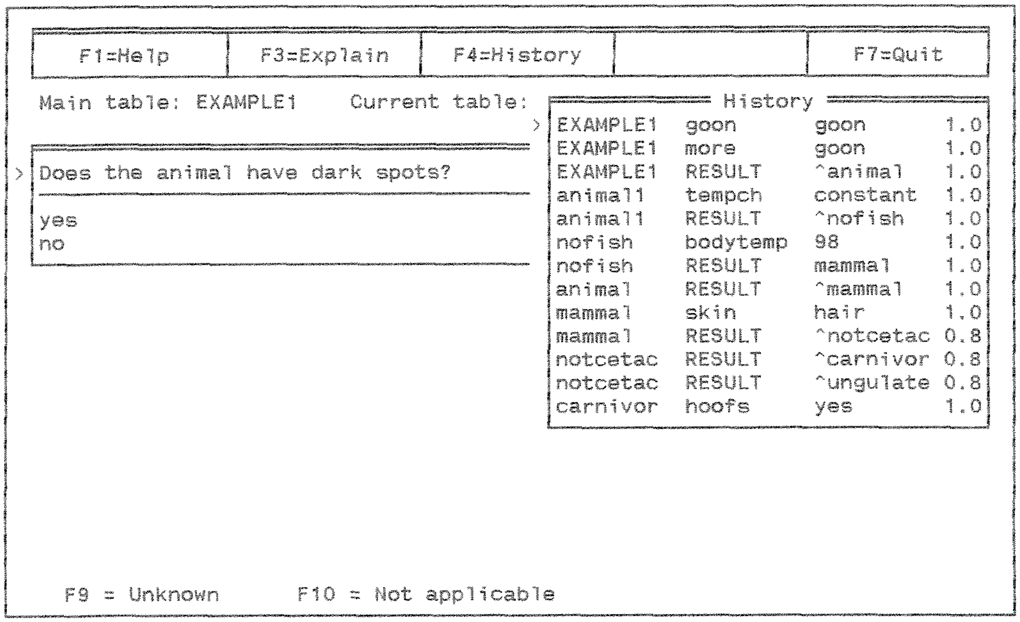
> Enter value : 98

F9 = Unknown

Figure 4.2.1.4.1. Questions and answers in HICLASS

4.2.1.5. History

Choosing F4, a list of all questions, answers and results as well as their certainty for the current session is displayed.



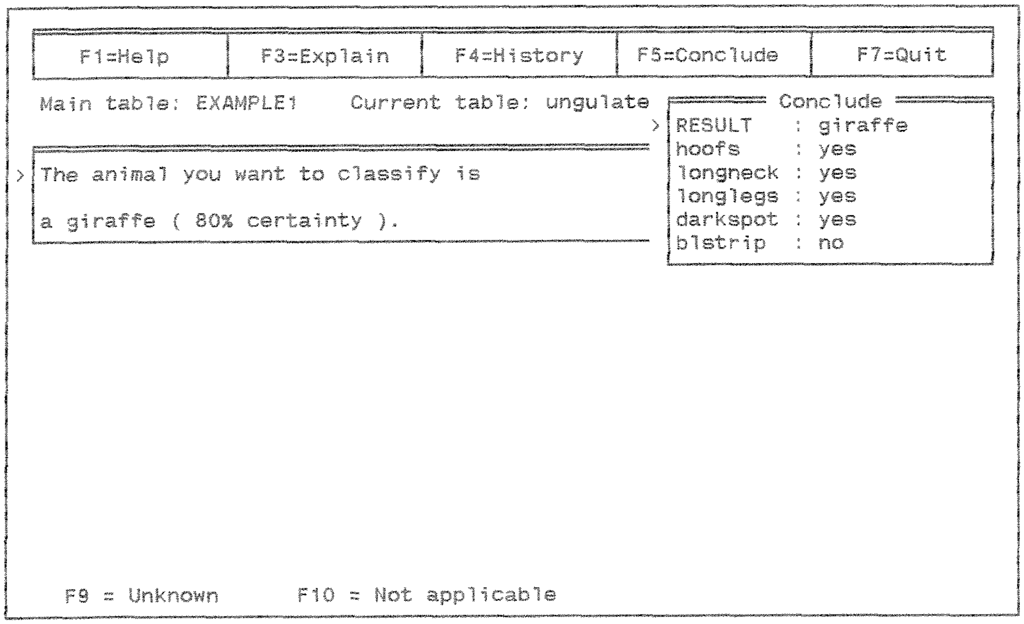
```
  F1=Help        F3=Explain       F4=History                       F7=Quit

 Main table: EXAMPLE1    Current table:  ======== History ========
                                       > EXAMPLE1  goon      goon      1.0
                                         EXAMPLE1  more      goon      1.0
> Does the animal have dark spots?       EXAMPLE1  RESULT    ^animal   1.0
                                         animal1   tempch    constant  1.0
  yes                                    animal1   RESULT    ^nofish   1.0
  no                                     nofish    bodytemp  98        1.0
                                         nofish    RESULT    mammal    1.0
                                         animal    RESULT    ^mammal   1.0
                                         mammal    skin      hair      1.0
                                         mammal    RESULT    ^notcetac 0.8
                                         notcetac  RESULT    ^carnivor 0.8
                                         notcetac  RESULT    ^ungulate 0.8
                                         carnivor  hoofs     yes       1.0

    F9 = Unknown       F10 = Not applicable
```

Figure 4.2.1.5.1. History in HICLASS

4.2.1.6. Results

Each table invoked will have zero, one or more results. The text screens of results are only shown, if the text for the attribute RESULT is not empty, otherwise the system just moves on in the reasoning process without stating results. This is useful if the user should or should not be informed about subresults along the path. It is also possible to display a result text if the particular result is a call to another table. If it is indicated that a result text should be produced and there is no proper result for the particular table, then the message "Sorry, no advice possible" is generated. If there are several valid results for a table, then all the texts are given, divided by an "OR". In order to incorporate the certainty information for results, two special strings can be defined in the HIEDIT editor. These strings will be replaced by the actual certainty value. "$$$" will show the certainty on a per cent scale (certainty 0.8 will be displayed as '80'), and "$.$" produces a notation similar to the internal representation (certainty 0.8 is displayed as '0.8').

With F5, the CONCLUDE option can be activated. The system shows all values for all attributes for a particular result. If a value is '*', then the whole domain of values is provided. Again, if there are multiple results, values for every single result are provided.



Main table: EXAMPLE1   Current table: ungulate

The animal you want to classify is

a giraffe (80% certainty).

```
RESULT   : giraffe
hoofs    : yes
longneck : yes
longlegs : yes
darkspot : yes
blstrip  : no
```

Figure 4.2.1.6.1. Stating results in HICLASS

4.2.1.7. Example sessions

In order to illustrate the performance of HICLASS, the protocols of a number of sample runs are provided below. The protocols are copies of session report files created by HICLASS.

Example 1:

The user "sees" a whale.

```
EXAMPLE1  goon      goon      10
EXAMPLE1  more      goon      10
EXAMPLE1  RESULT    ^animal   10
animal1   ~tempch   UNKNOWN
tempch    tempch    UNKNOWN
tempch    RESULT    NO RESULT 0
animal1   breathing lungs     10
animal1   RESULT    ^nofish   5
nofish    bodytemp  UNKNOWN
nofish    reproduct uterus    10
nofish    RESULT    mammal    2
animal    RESULT    ^mammal   2
mammal    skin      naked     10
mammal    RESULT    NO RESULT 0
```

NOTES:
The user has no knowledge about the body temperature of the animal. Nevertheless it can be concluded that the animal is a mammal. Since the threshold of table mammal is 0.5 and the certainty of the path is 0.2 by now, no advice can be given.

Example 2:

The user "sees" a zebra.

```
EXAMPLE1  goon      goon      10
EXAMPLE1  more      goon      10
EXAMPLE1  RESULT    ^animal   10
animal1   ~tempch   constant  10
animal1   RESULT    ^nofish   10
nofish    bodytemp  98        10
nofish    RESULT    mammal    10
animal    RESULT    ^mammal   10
mammal    skin      hair      10
mammal    RESULT    ^notcetac 8
notcetac  RESULT    ^carnivor 8
notcetac  RESULT    ^ungulate 8
carnivor  !hoofs    yes       10
carnivor  !darkspot no        10
carnivor  !blstrip  yes       10
carnivor  RESULT    NO RESULT 0
ungulate  !hoofs    yes       10
ungulate  !longneck no        10
ungulate  !longlegs no        10
ungulate  !darkspot no        10
ungulate  !blstrip  yes       10
ungulate  RESULT    zebra     8
```

4.2.2. Possible improvements

After working on the programs and while reviewing the results it became clear that a number of improvements could be made in addition to the features not implemented yet at all. A number of these improvements are mentioned below.

- For several reasons, it could be more efficient to implement the global control strategy in a best-first manner. This could for instance be achieved while treating each table as an independent object and maintaining a global control list storing crucial information about the objects currently present in the system. If a table is called, information about this table including its prior certainty, the name of the calling table, the fact if it is called by a result or by an attribute could be stored in the global list. The global control strategy would decide about the table to work on next. Different to the current implementation of HICLASS, tables would only be called by the global strategy, not by other tables. If a table is waiting for the answer of another table, and the second table happens to be solved, then the first table can request this information from the second one.

- The reasoning capabilities could be extended in order to first make some basic checks about the likelihood of a table (as described for CSRL; see sections 5.3.1. and 5.3.2.).

- The CONCLUDE option could be extended to cover intervals of numeric values and not only single values.

- The HISTORY option could be designed more user-friendly while not using internal attribute and value names, but a more sophisticated output.

- So far, the accentuations and colors provided with the editor in HIEDIT are not accessible within HICLASS. This could be changed.

- If a table is calling another, then up to now only the results of ONE table can be given back to the calling table. It should be possible that all appropriate results produced in a subtree can be provided.

- The type of numeric values should be changed from integer in the range of [0..253] to floating point.

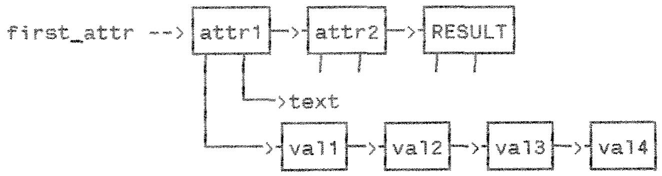
4.3. Implementational details

4.3.1. Main data structures

Due to the dynamic nature of the problem, almost all data structures are designed in a dynamic way. Data fields are created when needed, and disposed after use. This is true for menus and spreadsheets as well as for table definitions. In HICLASS, a table is read from disk, it is processed, and then disposed after it is completely solved. This allows to build very large systems without running out of memory. Every attempt to create new data fields is combined with a memory check. It is checked if after the memory allocation the remaining memory space is sufficient to allow a proper program performance, e.g. accessing the menus in HIEDIT. The main data structure in HICLASS is a table. A table, a dynamic data structure itself, is defined in the following way:

```
table ---> table

main_table   = record               {HICLASS table format}
                name:string[9];               {table name}
                max_attr:integer;             {number of attributes}
                max_ex  :integer;             {number of examples}
                first_attr:attr_pointer;      {start of attributes}
                first_ex:ex_pointer;          {start of examples}
                unknown_allowed:boolean;      {unknown allowed}
                dont_applic_allowed:boolean;  {don't applicable allowed}
                predefined:boolean;           {predefined order?}
                favored_strategy:byte;        {favored local strategy}
                threshold:byte;               {threshold for uncertainty}
                interval:byte;                {interval numeric values}
                shortcut:boolean;             {shortcut allowed?}
                strategy_used:byte;           {strategy used}
                prior_certainty:byte;         {prior cert. for table}
                reader:pointer;               {reader for table}
                no_ques:byte;                 {number of questions}
                no_unknown:byte;              {number of answers UNKNOWN}
                num:num_pointer;              {list of numeric values}
                numeric:boolean;              {results numeric}
               end;
```

There is a pointer providing access to the contents of the table which has a *name* attached to it. The number of attributes defined (*max_attr*), and the number of class descriptions stored in the table (*max_ex*) are provided. Pointers to the beginning of linked lists for the attribute definitions (*first_attr*) as well as the class descriptions (*first_ex*) are included. A number of control fields are bound to each table definition, providing information important for the performance of the local and global control strategies. A pointer (*reader*) allows to access the dialogue window produced for the user interface. The number of questions asked and the number of questions answered with UNKNOWN are recorded and stored within the table data structure. Each table includes all information necessary to continue its processing even if the particular table is not the one currently focused on (there is only one *active* table at a time, but maybe several tables are waiting to be completed; see section 4.2. for an explanation of the control strategy). The attribute/value definitions are separated from the class descriptions. They are stored in linked lists.



```
main_attr    = record                  {attribute}
                  name:string[9];              {attribute name}
                  text:text_pointer;           {text for attribute}
                  askfirst:byte;               {0=no 1=askfirst 2=no askfirst}
                  max_val:byte;                {number of values}
                  values:val_pointer;          {value definitions}
                  min_cert:byte;               {current minimal certainty of values}
                  next:attr_pointer            {next attribute}
               end;

main_val     = record                  {value}
                  name:string[9];              {name of value}
                  text:string[74];             {text for value}
                  textres:text_pointer;        {text for values of RESULT}
                  cert:byte;                   {current certainty of value}
                  next:val_pointer;            {next value}
              end;

main_text   = record                           {text format}
                  anzherv:byte;                                {number of accentuations}
                  text:array[1..eh] of string[eb];             {text itself}
                  herv:array[1..max_hervor,1..5] of byte;      {accentuations}
             end;
```

The class descriptions of a table are stored in a separate linked list. The reference to the attribute and value definitions is realized using numeric values referring to the relative position of the appropriate definition in the attribute/value linked lists. If during the reduction of a table class descriptions and/or values are deleted, the reference is updated accordingly. Numeric values are stored as such. The last field in the class description holds the weight of the description.

```
main_ex      = record                          {example}
                  v:array[1..abs_max_attr+1] of byte;
                  next:ex_pointer;
              end;
```

A complete description of the data structures used within the system can be found in Appendix C.

4.3.2. The file structure for a table

The following logical file structure is used for storing a table defined in HIEDIT on a storage device. Not all information is used by HICLASS, the distinction-oriented knowledge representation is included for the use by HIHYPO only, and the non-distinction-oriented representation (most general description to be derived) is not implemented yet.

- file tag
- distinction-oriented learning examples provided (y/n)
- non-distinction-oriented learning examples provided (y/n)
- answer UNKNOWN allowed (y/n)
- answer NOT APPLICABLE allowed (y/n)
- predefined order enforced (y/n)
- favored strategy (0=none, 1=MATCH, 2=LEFT-TO-RIGHT, 3=HEURISTIC)
- threshold for uncertainty handling [0.1..1.0]
- interval range for numeric values [0..50]
- unique results only (y/n)
- number of attributes
- for all attributes
    - attribute name
    - askfirst (0=not valid, 1=askfirst=true, 2=askfirst=false)
    - number of values
    - for all values
        - value name
        - value text
- number of examples
- for all examples
    - example content
- number of distinction-oriented examples
- for all examples
    - example content
- number of non-distinction-oriented examples
- for all examples
    - example content

4.3.3. Efficiency

A lot of thought was given to an efficient storage management. Most data structures are implemented dynamically to allow a very flexible performance of the system. In the case of HICLASS, only a few global variables and information crucial to further process invoked but yet unsolved tables are kept in main memory. Algorithms used to process tables are designed to be time efficient.

As described in Appendix B, the system is strongly modularized. Maintenance can be focused on the very module performing a specific task. The modules have well-defined interfaces between each other. Thus, a module can be changed without affecting other modules. If, for example, mouse support is desired, only the low-level utility modules concerned with the user interface have to be changed. A change in global constants and data structures affects the whole system. In most cases no changes in any of the modules are necessary.

The program shows small processing delays working with small to medium size tables. With large size tables, processing time increases. Thus, time crucial parts of the program could further be optimized. It is proposed to strongly modularize an expert system to be built. Small tables are not only faster to process, they are also easier to change and to comprehend.

5. Evaluation of the HICLASS system

5.1. HICLASS as a tool for a generic task

It was stated that HICLASS attempts to serve as a tool for the generic task hierarchical classification as given by Chandrasekaran. Let us first have a look at the features of a generic task to prove if this attempt was successful, since a successful tool for a particular task would have to fulfill these requirements.

Multiformity:

HICLASS is based on a special way to organize and use knowledge. Sets are linked together in a hierarchy, preserving inheritance. An establish-refine strategy is used to traverse the hierarchy tree. Basically, reasoning by elimination takes place and specific control strategies serve to guide the performance in the most promising manner using operations designed to deal with the data structure. HICLASS is best suited for performing a hierarchical classification task.

Modularity:

HICLASS can be used as an independent tool, but it also can be incorporated in a complex knowledge-based system as a subtask cooperating with other generic tasks. The particular function, hierarchical classification in this case, is decomposed into its conceptual parts. These parts are tables, including one ore more class descriptions. Domain knowledge of other form is inserted, e.g., evidence-accumulation knowledge.

Knowledge acquisition:

A knowledge acquisition strategy has to be used to build a HICLASS application. The system allows a very flexible organization of the hierarchy to be built. The knowledge engineer has to determine useful categories and ways of linking the categories together in the hierarchy.

Explanation:

HICLASS provides an explanation feature in order to explain current steps of the control strategy to the user in a local or global manner. This is possible, since the control strategy is very specific.

Exploiting the interaction between knowledge and inference:

As already mentioned, a particular way of representing knowledge (values of attributes in class sets) is integrated with a particular way of using that knowledge (set reduction according to the match of input data with the values of the class descriptions). Additionally, the global control strategy is especially designed to deal with the structure of the knowledge embedded in the system.

The discussion shows that HICLASS addresses all of the important features of a generic task, and can therefore be useful as a tool serving to fulfill this task.

5.2. HICLASS as a tool for hierarchical classification

To show that HICLASS is a genuine hierarchical classification tool it must be demonstrated that HICLASS incorporates the problem-solving strategy and knowledge appropriate for the specific task as defined by Chandrasekaran [5].

> "Hierarchical classification requires as input a data description of the problem to be solved. After processing, the task yields all the categories of the hierarchy that apply to the given data." [5, p.218]

The input required by HICLASS is a data description of the problem. The data is entered by answering questions the system generates. Answering questions can mean that the user types in an answer, but it can also mean that an external program or real world process sends the data. The basic attempt of most of the local control strategies is that only a minimum number of questions have to be asked. The system will come up with one or more results, supplemented by certainty values defining the likelihood of the particular result. It is also possible to list all the subresults along the relevant path(s).

> "The classifier requires a preenumerated list of the categories that it will be using. Furthermore, these categories must be organized into a hierarchy in which the children (...) of a node represent subhypotheses of the parent. ... As the hierarchy is traversed from the top down, the categories (...) become more specific." [5, p.218]

Each HICLASS system has a preenumerated list of the categories it uses. These categories are referred to as *classes* in HICLASS. The classes are organized into a hierarchy, in which the children represent subhypotheses of the parent. The classes become more specific going down in the hierarchy tree. A special feature of HICLASS is that classes can be combined in a table.

> "Each node in the hierarchy is responsible for calculating the "degree of fit", or confidence value, of the hypotheses that the node represents. ... Each node can be thought of as an expert in determining whether the hypothesis is true. For this reason, each node is termed a specialist in its small domain." [5, p.218]

The "degree of fit" is expressed in the shape of certainty values in HICLASS. A certainty value is assigned to each hypothesis that is not ruled out, depending on prior certainties and the number of answers UNKNOWN. In fact, each class can be thought of as a specialist in a limited domain. In the case of classes combined in a table, the system tries to determine which of the classes is true with which certainty. If there is only one class per table, then the system tries to come up with the certainty of this class.

"To create each specialist, knowledge must be provided to make the degree-of-confidence decision. The general idea is that each specialist specifies a list of features that are important in determining whether the hypothesis it represents is true and a list of patterns that map combinations of features to confidence values." [5, p.219]

In HICLASS, a number of *attributes* is defined for each table. These attributes, or features, have well defined values for a particular class description. They serve to rule out classes in the case of a class set and to determine the certainty value of one or more succeeding classes. A class description consists of one or more instances that provide values for all the attributes, including one special attribute, the *result*, representing a hypothesis. Prior weights are bound to the instances. Thus, if an instance can be matched, a result with a special certainty is produced.

In order to efficiently traverse the hierarchy, a type of hypothesis refinement is used: establish-refine. "A specialist that establishes its hypothesis (...) refines itself by activating its more detailed subspecialists, while a specialist that rules out or reject its hypothesis (...) does not send any messages to its subspecialists, thus avoiding that entire part of the hierarchy. ... The establish-refine process continues until no more refinements can take place. This can occur either by having reached the tip level hypothesis of the hierarchy or by having ruled out mid-hierarchy hypotheses." [5, p.219]

The control strategy described above is the global strategy used to guide the classification process. In HICLASS, one or more results with a certainty value bound to them are produced after the table is "solved" - hypotheses are established. The process continues while invoking the subspecialists a particular result is pointing to (the hypothesis refines itself). If the subspecialists are combined in a table, only one pointer is necessary, otherwise more than one. In class sets, wrong hypotheses are either automatically ruled out in the set reduction process or a certainty value of zero is assigned to them. In both cases, the subspecialist of these classes will not be established. The process stops when all paths followed terminate because all current tables are leafs in the classification tree, and when all current hypotheses are either ruled out or hold a certainty value of zero.

It could be shown that HICLASS addresses all the issues raised by Chandrasekarans definition of the generic task hierarchical classification. HICLASS incorporates the problem-solving strategy and knowledge appropriate for this task.

5.3. HICLASS in comparison

5.3.1. Description of CSRL

CSRL (Conceptual Structure Representation Language) is introduced as a language for writing hierarchical-classification expert systems. Chandrasekaran [5, pp.215-239] describes the basic idea of CSRL at a level of detail which allows to make a general comparison, some details though can only be assumed or are not known.

In CSRL, each specialist for a particular hypothesis is implemented individually. The parents (referred to as *superspecialists*) and subspecialists of a specialist are declared within the definition (DECLARE). A skeletal outline of a specialist definition for a bad-fuel node is the following:

```
(SPECIALIST BadFuel
 (DECLARE (SUPERSPECIALIST FuelSystem)
          (SUBSPECIALIST LowOctane WaterInFuel DirtInFuel))
 (KGS...)
 (MESSAGES...)
```

The KGS section (knowledge group section) consists of knowledge groups that "contain knowledge that matches the features of a specialist against the case data. Each knowledge group is used to determine a confidence value for some subset of features used by the specialist. ... A knowledge group is implemented as a cluster of production rules that maps the values of a list of expressions (...) to some conclusion on a discrete, symbolic scale" [5, p.220]. One knowledge group of BadFuel called "relevant" has the following content:

```
(RELEVANT TABLE
 (MATCH
  (ASKYNU? "Is the car slow to respond")
  (ASKYNU? "Does the car start hard")
  (AND(ASKYNU? "Do you hear knocking or pinging sounds")
     (ASKYNU? "Does the problem occur while accelerating")
 WITH (IF T?? THEN -3
       ELSEIF ?T? THEN -3
       ELSEIF ??T THEN 3
       ELSE 1)))
```

The expressions in MATCH query the user. ASKYNU? is a LISP function asking the user for YES, NO or UNKNOWN and translates the answer into T (true), F (false) or U (unknown). Any LISP function can be used instead. The results of the MATCH are then compared to a condition list. A '?' in a pattern means 'doesn't matter'. If the first question is answered with YES, then the first pattern 'T??' is true and -3 becomes the value of the knowledge group (the values are assigned on a discrete scale from -3 to 3, where -3 means "ruled out" and 3 stands for "confirmed"). Otherwise, the other patterns are evaluated. If none of the rules match, the value for the knowledge group will be 1 (default value). The following knowledge is encoded with the group:

> "If the car is slow to respond or the car starts hard, then BadFuel is not relevant in this case. Otherwise, if there are knocking or pinging sounds and if the problem occurs while accelerating, then BadFuel is highly relevant. In all other cases, BadFuel is only middle relevant" [5, p.221].

A specialist can contain several knowledge groups, which are separately checked in a specific order. Special knowledge groups can be designed to combine values of several groups into a single confidence value, thus abstracting the results of a number of knowledge groups.

```
(SUMMARY TABLE
 (MATCH RELEVANT gas
  WITH (IF 3 (GE 0) THEN 3
        ELSEIF 1 (GE 0) THEN 2
        ELSEIF ? (LT 0) THEN -3)))
```

The MATCH expressions stand for the two knowledge groups "relevant" and "gas". For example, if the value of the relevant knowledge group is 3 and the value of the gas knowledge group is greater or equal to 0 (GE(0)), then the value of the summary knowledge group (and so the confidence value of BadFuel) is 3.

The overall control strategy is realized with inserting a MESSAGE section into the definition of a specialist. This section "contains a list of message procedures that specify how the specialist will respond to different messages from its superspecialist" [5, p.222]. There are two predefined messages: ESTABLISH and REFINE.

"The ESTABLISH message procedure of a specialist determines the confidence value (...) of the specialist's hypothesis" [5, p.222].

```
(ESTABLISH (IF (GE relevant 0)
            THEN (SETCONFIDENCE self summary)
            ELSE (SETCONFIDENCE self relevant)))
```

The terms "relevant" and "summary" refer to knowledge groups defined within the specialist, "self" stands for the name of the specialist itself. The example procedure first tests the value of the relevant knowledge group (if it is not evaluated yet, then this is done now). If the value is greater or equal to 0, then the confidence value of BadFuel is set to the value of the summary knowledge group; otherwise it is set to the value of the relevant knowledge group. The strategy behind this is that if BadFuel is not a relevant hypothesis to hold (indicated by a value less than 0), then the confidence of the specialist is set to the degree of relevance. Otherwise, more complicated reasoning is performed to determine the confidence value (the summary knowledge group combines the values of other knowledge groups).

"The REFINE message procedure determines which subspecialist should be invoked and which messages they are sent" [5, p.223].

```
(REFINE (FOR specialist IN subspecialists
         DO (CALL specialist with ESTABLISH)
            (IF (+? specialist)
            THEN (CALL specialist WITH REFINE))))
```

The procedure calls each subspecialist with an ESTABLISH message. If the subspecialist establishes itself, then it is sent a REFINE message (+? tests whether the confidence value is +2 or +3). Other than having a 'Big Brother'-control structure organizing the establishment of hypotheses, the nodes itself are active and invoke children if necessary.

There are several aspects of hierarchy within this philosophy. First, the categories are organized in a hierarchical manner. Second, the knowledge within one knowledge group is organized in a way that if a row of the group is matched, then none of the subsequent rows is evaluated.

5.3.2. HICLASS vs. CSRL

First of all, CSRL is a LISP-based *language*. HICLASS, on the other hand, is an expert system shell, even if there also is the possibility to build a language around the basic concepts. As a language, CSRL can be applied very flexible; especially the feature of user-defined LISP functions is a powerful option. Part of the attempt of HICLASS is to free the user from *programming* the system in the sense of the word. Rather than writing functions, a user in HICLASS would enter his/her knowledge into predefined tables, supplemented by prior certainty and threshold values, as well as important information like the order dependency of attributes and hierarchy structure information. The system itself would decide about control strategies to apply and solve the problem according to a predefined plan of action. Less flexibility is the price to pay.

The basic distinction between CSRL and HICLASS though, resulting in a number of differences, can be found at another level of abstraction. The whole philosophy is different in a way. In HICLASS, a *specialist* is a class, described by one or more instances, most likely combined in a table together with other class descriptions. Each table has a parent, the information about this link is not given within this child table to allow a flexible usage, but can be derived from the hierarchy structure within the particular system. Further on, each class, which in fact is a hypothesis, has a result which is either a true statement about the world (at the leaf level) or a pointer to other class tables, which can be referred to as children. So far, there is no difference to the definition of a specialist in CSRL.

The difference is that in the case of CSRL the certainty of a hypothesis is not derived from only one class description and that hypotheses are never combined in one single data structure like a table in HICLASS. In order to prove a hypothesis in CSRL, several knowledge groups (KGS) can be considered, thus allowing a very flexible and extendable proof. Each KGS provides a confidence value which can be summarized in a user-controlled way to provide a value for the whole specialist. The overall control structure in CSRL is realized with the help of MESSAGES, which are sent from a parent to its children, determining how the confidence value of the subhypothesis should be determined and which threshold should be used to refine the subhypotheses itself. The latter contrasts to HICLASS in a way, that there the overall control strategy is implicit given in the system (if a class has a certainty equal or greater than a threshold, then we move on to the next level in the tree). Considering only the messages ESTABLISH and REFINE, there is no difference in the performance. But user-defined messages can be passed as well in CSRL, which makes the control explicit and more flexible.

The task of a specialist in CSRL can be considered as defining a group of KGS which all serve to prove one specific hypothesis. The KGS itself are independent and can be used by several specialists. Each KGS provides a confidence value. The confidence values of several KGS can be combined in a flexible way. Once an overall confidence value for a specialist is determined, it is compared with a threshold. If the check is successful, then activating messages are sent to the children, otherwise the path is closed. No accumulation of evidence takes place from one level of the tree to another, only within one specialist - another difference to HICLASS.

A very interesting and useful fact is that CSRL allows to first call a KGS to make some basic checks, and then depending on the result, to either turn down the hypothesis or to perform more detailed checks in order to prove the hypothesis on a finer scale. This is also the reason for a default confidence value within a KGS. It allows to introduce a decision if some more reasoning should be done, even if none of the patterns in the particular group matches the data. This is not necessary in HICLASS, since we only deal with one group of patterns, and if these cannot be matched, then the hypothesis can be turned down immediately. The combination of confidence values of different KGS happens in a totally user controlled manner in the shape of predefined calculation rules, that seem difficult to derive.

For an example that only has *one* KGS in order to prove the hypothesis of a specialist, HICLASS comes to the same results CSRL would do. Differences are that in HICLASS evidence would be accumulated for a particular path and that several specialists could be combined in one table. The latter one could be important if several specialists share attributes and are distinguishable from each other. Questions for special attributes can be answered by invoking other tables; if this is also possible in CSRL cannot be derived from Chandrasekarans description. It is also not clear if the nature of questions (only YES/NO/UNKNOWN) can be changed and if the order dependent left-right strategy in asking these questions has to be maintained, which both seem to limit the flexibility otherwise very strong within CSRL. Another similarity is that in HICLASS as well as in CSRL several class descriptions can be combined (the three questions in the example KGS *bad fuel* can be implemented as three instances in a HICLASS table).

As described above, one of the major differences between the approaches, if not THE major difference, is that a class description in HICLASS is only realized within the boundaries of one table, whereas in CSRL different KGS can be combined to establish one overall hypothesis.

5.3.3. Description of 1st-CLASS

5.3.3.1. 1st-CLASS specifications

NOTE: The description of the expert system shell 1st-CLASS provided below is directly taken from an explanation file delivered with the 1st-CLASS package; only information relevant for a comparison with HICLASS will be given.

Copyright :	(C) Copyright 1985, 1986. Programs in Motion Inc., Wayland MA
Program type:	Expert System Generator.
Methods used:	Inductive classification, Database search, and/or Direct rule construction & editing.
Data entry method:	Examples in a spreadsheet format or direct rule construction.
Data types:	Logical (choices) and numeric (floating point).
Example editor:	Multiple choice entry plus editing functions.
Size of one module:	Up to 32 factors, 32 results, and 255 examples.
Chained modules:	No limit except on-line disk capacity.
Expert advisor:	Auto-generated or user-created advisor screens.
Advisor editor:	Full screen editor, supports color/attributes.
Rule generation:	four algorithms can be used: - optimized decision tree construction; - ordered, allows you to choose processing order; - matching, for pattern matching applications; - direct building/editing of rules.
Speed of operation:	Since the rules are compiled, there are no delays during use.
Rule editor:	On screen, graphical rule editor.
Weights:	Can be assigned to each example; several statistical indexes can be calculated from them and displayed.
Answerback:	Summarizes how answer was reached and allows the user to change an answer and run again.
Report generation:	Can build a report on disk automatically.
Data interchange:	Exchanges data with other programs.
File access:	Can process data from disk files.
Programming language:	Not required to use 1st-CLASS; can be used for special needs if desired.
External programs:	Can be written in any language, and can pass data to and from 1st-CLASS.
Logic engine:	1st-CLASS can be called from other programs and can return an answer to them.

5.3.3.2. Using 1st-CLASS

The following examples were edited and performed in 1st-CLASS (the second example is a sample knowledgebase provided with the 1st-CLASS package), parts of screens of the program are shown below.

Example 1:

Examples:

```
    type       size       location    RESULT      Weight
1:  cetacea    25ft       atsea       whale       [1.00]
2:  cetacea    6ft        nearcoast   porpoise    [1.00]
3:  cetacea    6ft        atsea       dolphin     [1.00]
4:  fish       1ft        npacific    salmon      [1.00]
5:  fish       6ft        atsea       shark       [1.00]
```

Rule optimized:

```
size??
├25ft:─────────────────────────────────────────────whale
├6ft:type??
│    ├cetacea:location??
│    │          ├nearcoast:────────────────────────porpoise
│    │          ├npacific:─────────────────────────no-data
│    │          └atsea:────────────────────────────dolphin
│    └fish:────────────────────────────────────────shark
└1ft:──────────────────────────────────────────────salmon
---- end of rule ----
```

Statistics for <whale>:

```
Active  examples:    5   Result's  examples:    1   Examples: 1
Result frequency: 0.20   Result probability: 0.20   Relative probability: 1.00
Total weight:     5.00   Result weight:      1.00   Average weight:       1.00
```

Rule left to right:

```
type??
├cetacea:size??
│         ├25ft:───────────────────────────────────whale
│         ├6ft:location??
│         │     ├nearcoast:────────────────────────porpoise
│         │     ├npacific:─────────────────────────no-data
│         │     └atsea:────────────────────────────dolphin
│         └1ft:────────────────────────────────────no-data
└fish:size??
      ├25ft:───────────────────────────────────────no-data
      ├6ft:────────────────────────────────────────shark
      └1ft:────────────────────────────────────────salmon
---- end of rule ----
```

Example 2:

Examples:

```
     sauce       pref-color  main-comp   color       Weight
 1:  *           red         *           red         [1.50]
 2:  tomato      *           *           red         [1.00]
 3:  *           white       *           white       [1.50]
 4:  cream       *           *           white       [1.00]
 5:  *           *           meat        red         [1.00]
 6:  *           *           veal        white       [1.00]
 7:  cream       *           turkey      white       [1.00]
 8:  cream       *           poultry     white       [1.00]
 9:  tomato      *           turkey      red         [1.00]
10:  tomato      *           poultry     red         [1.00]
11:  *           *           fish        white       [1.00]
12:  *           *           fish        white       [1.00]
13:  cream       *           other       white       [1.00]
14:  tomato      *           other       red         [1.00]
```

Parts of the rule:

```
sauce??
├cream:pref-color??
│      ├red:main-comp??
│      │    ├meat:────────────────────────────────red    (!)
│      │    │    & - - - - - - - - - - - - - - - -white
│      │    ├veal:────────────────────────────────white
│      │    │    & - - - - - - - - - - - - - - - -red
│      │    ├turkey:──────────────────────────────white
│      │    │      & - - - - - - - - - - - - - - -red
│      │    ├fish:────────────────────────────────white
│      │    │    & - - - - - - - - - - - - - - - -red
│      │    ├poultry:─────────────────────────────white
│      │    │       & - - - - - - - - - - - - - - red
│      │    └other:───────────────────────────────white
│      │          & - - - - - - - - - - - - - - - red
│      └white:main-comp??
│              ├meat:─────────────────────────────white
│              │    & - - - - - - - - - - - - - - -red
│              │
...
```

Statistics for first path of <red> (!):

```
Active  examples:   14   Result's  examples:    2   Examples: 1,5
Result frequency: 0.43   Result probability: 0.17   Relative probability: 0.71
Total weight:    15.00   Result weight:      2.50   Average weight:       1.25
```

Dialogue:

```
sauce        = cream
pref-color   = red
main-comp    = meat
```

Result:

```
You've selected red wine, with a confidence of 71.43%.
 -- or --
You've selected white wine, with a confidence of 28.57%.
```

After experimenting with the program, the following details of behavior seem to be important for a comparison with HICLASS. Example 2 will be used in the discussion.

```
The following examples contributed to the final result:

 1:   *           red         *           red         [1.50]
 4:   cream       *           *           white       [1.00]
 5:   *           *           meat        red         [1.00]


Statistics for one of the succeeding paths (red):

Active  examples:    14   Result's  examples:    2   Examples: 1,5
Result frequency: 0.43    Result probability: 0.17   Relative probability: 0.71
Total weight:     15.00   Result weight:      2.50   Average weight:       1.25
```

a) The weight for a result is determined by simply adding up all the weights of examples for this result.

b) The statistics calculated for a particular path are statistics in the sense of the word.

```
For the succeeding path of <red>:

Result frequency     = # of examples for result /  # of all examples
                     = 6 / 14 = 42.8

Result probability   = result weight / total weight
                     = 2.5 / 15 = 0.166

Relative probability = result weight / total result weight of succeeding result
                     = 2.5 / 3.5 = 0.71
```

c) The goal of the performance is to come up with one or several results with a ceratin probability. If there is only one succeeding result, then this is given with 100% certainty, even if there are examples describing this result with a weight different to 1.0.

d) If a question is answered by another knowledgebase, this knowledgebase is invoked first, there is nothing like an ASKFIRST option.

e) If a question is answered by another knowledgebase, and this knowledgebase has several results with different probabilities, then only the result with the highest probability is given back to the calling knowledgebase (or in the case of equal probabilities the first result identified). There is always only ONE result per knowledgebase. The probability of this result is NOT used in the reasoning process of the calling knowledgebase.

f) If a knowledgebase has several results with a certain probability, and these results call other knowledgebases, then only the knowledgebase is invoked that is called by the result with the highest probability. There is always only ONE knowledgebase called. Probabilities are NOT propagated down to the next knowledgebase(s).

Summarizing the comments so far it can be stated that the global control strategy as well as the uncertainty handling is very straight forward and simplifies results of the reasoning process in order to maintain this straight forward philosophy. That this simplification can lead to problems will be shown in the following example, solved with 1st-CLASS.

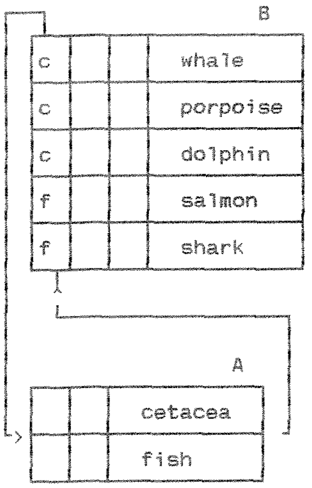
Example:

A:

backbone	breathing	type	weight
yes	blow hole	cetacea	1.0
yes	gills	fish	1.0

B:

type	size	location	creature	weight
cetacea	25 ft.	at sea	whale	1.0
cetacea	6 ft.	near coast	porpoise	1.0
cetacea	6 ft.	at sea	dolphin	1.0
fish	1 ft.	n.pacific	salmon	1.0
fish	6 ft.	at sea	shark	1.0



The following sequence of action was recorded by the report file created by 1st-CLASS. The first column shows the active knowledgebase, the second column denotes the attribute (called *factor* in 1st-CLASS) and the third column the value.

a) the user "sees" a shark

```
a      breathing   UNKNOWN
a      backbone    yes
B      a           fish
B      size        6ft

B      RESULT      shark
```

Knowledgebase A is solved, the user does not know the value for <breathing>, thus both results are valid with a relative probability of 0.5, but only the first result <fish> is taken. In this case, this does not result in any problem, the proper result can be found.

b) the user "sees" a whale

```
a      breathing   UNKNOWN
a      backbone    yes
B      a           fish
B      size        25ft

B      RESULT      UNKNOWN
```

Now, the system is not able to give an advice due to the fact that the result <cetacea> was rejected in knowledgebase A, even if it had the same probability of being true as <fish>.

5.3.4. HICLASS vs. 1st-CLASS

A lot of the features of HICLASS look similar to the ones of 1st-CLASS, which is due to the fact that the ideas used to develop HICLASS were influenced by the 1st-CLASS system, which incorporates a number of very useful approaches to solve the problem of building an expert system shell for hierarchical classification. To a certain extent, the same principles of building a hierarchy, holding several class descriptions in one table, maintaining a set of preenumerated solutions, allowing "Don't care" and "UNKNOWN", designing local strategies, and others can be found in both systems. Besides similarities though, a number of important differences have to be mentioned. One could look at HICLASS as a successor of 1st-CLASS, using useful approaches but trying to resolve serious limitations of this program.

Examples of useful 1st-CLASS features which will not be implemented in HIEDIT or HICLASS due to the high implementational effort are the graphic rule editor and the very flexible interface with the "world".

There are a number of problems addressed by HICLASS that are not touched by 1st-CLASS at all or solved in a questionable way. The fact that rules in 1st-CLASS are build beforehand helps to speed up an advising session since only a small number of calculations have to be performed, but this obviously does limit the flexibility of the system and it does not allow to guide the process of finding a result with regard to the current situation. Since no set reduction of the table takes place during an advising session, choices given to the user can be not valid anymore and the chance of wrong conclusions is introduced.

1st-CLASS does not exhaust all reasoning possibilities which can be derived from the data provided for the system (one example was given at the end of 5.3.3.2.). The simplification of the uncertainty handling and of a global control strategy leads to a loss of accuracy. HICLASS provides a more sophisticated uncertainty reasoning and a propagation of uncertainty. In HICLASS it is possible to follow several paths at the same time and to maintain thresholds for the termination of these paths. The certainties for several instances per class can be combined and an explanation feature is embedded in the system. In 1st-CLASS it can happen that the system follows a "blind" path (there is only ONE path at a time) and it is not possible to call multiple children.

As mentioned before: HICLASS incorporates a lot of the features of 1st-CLASS but attempts to overcome problems limiting an accurate performance of solving the task of hierarchical classification.

6. Further research

6.1. HIHYPO - hierarchical hypothesis matching

One of the generic tasks identified by Chandrasekaran [5] is hypothesis matching, which he defines as "matching hypotheses to a situation using a hierarchical representation of evidence abstraction. The general idea is that we have a set of data which potentially pertain to a concept. We want to know how well the concept matches the data. For example, the concept may be a disease and data may be patient data relevant to the disease, and we wish to know what the likelihood of the disease is. Hypothesis matching is a very common subtask in a number of reasoning tasks." [5, pp.216]

This chapter will be concerned about sketching a hierarchical hypothesis matching system (to be referred to as HIHYPO), using a number of ideas from HICLASS, complemented by goal-oriented control strategies and a special distinction-oriented knowledge representation. HIHYPO has no implementation yet.

The knowledge in HIHYPO will be organized in a hierarchy of tables. Again, the assumption is made that several concepts may share attributes and can therefore be combined in a table. Everything which was said about the HICLASS representation is valid: there are preenumerated solutions, "Don't care" values, weights, questions answered by other tables etc.

Example:

A:

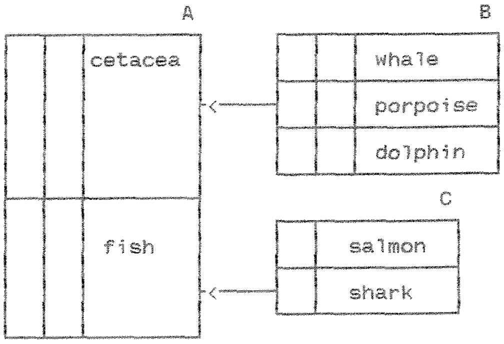
backbone	breathing	type	weight
yes	blow hole	cetacea	1.0
yes	gills	fish	1.0

B:

size	location	creature	weight
25 ft.	at sea	whale	1.0
6 ft.	near coast	porpoise	1.0
6 ft.	at sea	dolphin	1.0

C:

location	creature	weight
n.pacific	salmon	1.0
at sea	shark	1.0



Assuming, the hypothesis to be matched is <whale> to for instance answer a user's question "Can the creature I saw be a whale?", the hierarchy has to be traversed from bottom to top, whereas in HICLASS the opposite direction was followed. The assumption <whale> is taken as a goal and the attempt of HIHYPO is to confirm this goal. A backward chaining in contrast to the global forward chaining in HICLASS has to be performed, since in order to confirm <whale> the whole path from the very special description up to the most general one has to be confirmed. This is because *the whole path* describes the concept <whale>, not just the last table.

With a backward chaining (or goal-driven) approach "the system focuses its attention by only considering rules that are relevant to the problem on hand. In this approach, the user begins by specifying a goal by stating an expression E whose truth value is to be determined. ... The main advantage of the goal-driven approach is that it does not seek data and does not apply rules which are unrelated to the problem in hand." [9, pp.428, 430] If a goal can be confirmed within one table in the relevant path, the parent of this table has to be considered, thus serving as a global subgoal, and so on, until the top of the hierarchy is reached. For rejecting a goal though, it is sufficient to reject one of the subgoals along the path.

6.1.1. Local control strategy and knowledge representation

Within the context of one table, the attempt of the system is to confirm or reject a result of the table. The local control strategy is strictly goal-driven and it is mainly focused on the differences between concepts (again referred to as classes) combined in a table. Thus, values for attributes that are unique or most unique for the special goal compared to other classes in the table serve this strategy best. A special distinction-oriented knowledge representation, which is non-redundant and shows only the differences between classes would be perfect for this.

Example:

type	size	location	creature	weight
cetacea	25 ft.	at sea	whale	1.0
cetacea	6 ft.	near coast	porpoise	1.0
cetacea	6 ft.	at sea	dolphin	1.0
fish	1 ft.	n.pacific	salmon	1.0
fish	6 ft.	at sea	shark	1.0

Figure 6.1.1.1. Original class descriptions

type	size	location	creature	weight
*	25 ft.	*	whale	1.0
*	*	near coast	porpoise	1.0
cetacea	6 ft.	at sea	dolphin	1.0
*	1 ft.	*	salmon	1.0
fish	6 ft.	*	shark	1.0

Figure 6.1.1.2. Distinction-oriented class descriptions

The non-redundant, distinction oriented class descriptions in figure 6.1.1.2. are generated from the descriptions in figure 6.1.1.1. using a special kind of mechanical induction (the algorithm is described in section 6.4.). The new descriptions only include the information crucial to distinguish between the different classes within the table. The information, for example, that the size of the creature is <25 ft.> is sufficient to confirm that the creature is a <whale> within the (limited) worldview of the particular table. The '*' has to be read as 'Don't care' only by the control strategy, it can for instance not be derived that it doesn't matter if <whale> is a <fish> or a <cetacea>. Values for attributes describing a particular class are only defined if they are unique for this class, and not only values but also sets of values.

Example:

```
*           1 ft.     *           salmon     1.0
The knowledge about the length is sufficient to confirm or reject the goal <salmon>.
```

```
cetacea     6 ft.     at sea      dolphin    1.0
All three attributes have to be known to confirm or reject the goal <dolphin>.
```

A local goal-driven control strategy takes single values or sets of values as local subgoals and generates questions in order to inquire if these values match the user's data or not. If only one of these subgoals cannot be confirmed, the whole goal in a local as well as in a global sense has to be rejected.

Example:

a) to confirm a goal with one subgoal

```
goal: salmon
subgoal: 1 ft.

What is the length of the creature?   1 ft.
The only subgoal is confirmed, thus the goal is confirmed.
Result: The creature is a salmon.
```

b) to confirm a goal with multiple subgoals

```
goal: shark
subgoals: fish, 6 ft.

What is the class of the creature?   fish
What is the length of the creature?   6 ft.
All two subgoals could be confirmed, thus the goal is confirmed.
Result: The creature is a shark.
```

c) to reject a goal

```
goal: salmon
subgoal: 1 ft.

What is the length of the creature?  25 ft.
The only subgoal is rejected, thus the goal is rejected.
Result: The creature is not a salmon.
```

With a class description consisting of several instances per class, we choose the instance with the least amount of unique values. To confirm the goal class we have to confirm all of these values. To reject the goal though it is necessary to reject at least one unique value per instance.

6.1.2. Selected special problems

Almost all of the problems raised for HICLASS have to be considered in HIHYPO as well. Important questions to be answered are for instance:

- What happens if instances describing one class carry different weights?
- How to deal with uncertainty in general?
- How can the reasoning process be explained?
- Are there metarules to guide the process?
- What should be done if there is a predefined order within the table?

Although it is not in the scope of this work to fully cover all aspects of the HIHYPO system, let us nevertheless have a look at some crucial problems.

6.1.2.1. Class descriptions with different weights

Different to a local forward chaining control strategy, the attempt in HIHYPO is to confirm or reject a special local goal. This goal can be described by several instances, which in turn can have different weights attached to it. In order to fully exhaust all the information provided, we are not done with simply confirming the goal and moving up in the hierarchy, we also have to take care of the certainty with which the goal is confirmed.

Example:

*	*	at sea	whale	1.0
cetacea	*	near coast	whale	0.9
fish	*	*	salmon	1.0

```
goal: whale
subgoal: cetacea
What is the class of the creature?   cetacea

goal: whale (1.0)
subgoal: at sea
Where does the creature live?   at sea
Result: The creature is a whale for sure.
```

Why was <cetacea> chosen as the local subgoal for the first question, and not <at sea>? This was done according to the fact that a specific order is maintained: *first,* the local goal is confirmed using the distinction to other classes in the table, *then*, given there are multiple instances describing this goal carrying different weights, the appropriate weight is needed. Hence, one of the instances is chosen as the new local goal.

6.1.2.2. An answer UNKNOWN

Allowing the user to answer with UNKNOWN, additional uncertainty will be introduced. How to proceed in this case? The example used below is the distinction-oriented version of the example given in 6.1.1.

Example:

```
A:   *         blow hole   cetacea    1.0
     *         gills       fish       1.0

B:   25 ft.   *           whale       1.0
     *        near coast  porpoise    1.0
     6 ft.    at sea      dolphin     1.0


global goal      : whale
local goal B     : whale
local subgoal B  : 25 ft.

B: What is the size of the cetacea?  UNKNOWN

global goal      : whale
local goal A     : cetacea
local subgoal A  : blow hole

A: How does the creature breath?      through a blow hole

Result: It is possible that the creature is a whale (1.0).
        Due to incomplete information though, this can only be
        confirmed with 0.5.
```

A similar strategy is used as for a forward chaining in HICLASS. The answer UNKNOWN is ignored as not helpful to confirm the subgoal (in fact we act as if a confirmation took place). The answers UNKNOWN are counted for a particular table and this value is used to determine a certainty for the conclusion.

6.2. A complex problem-solver

The task of the hypothesis matching system could be extended such that in case of the rejection of a goal, the system could try to come up with a concept matching the user's data - a hierarchical classification task. To allow a combination of the two concepts, it is useful to start the hypothesis matching at the root table (this could be an advantage anyway, because more general questions might better be suited to early reject a goal). Having a goal to confirm, the system would trace the path up to the root table, maintaining a list of local goals, one for each table touched. If one local goal cannot be confirmed, the system could switch over to a classification mode to find a valid solution if there is one. This behavior would allow to not inquire information twice. Another idea would be to NOT start at the root, but jumping there in case of the rejection of the current goal, while keeping already observed tables including the user's answers in memory in case we have to invoke them again.

Example:

A:	*	blow hole	cetacea	1.0
	*	gills	fish	1.0

B:	25 ft.	*	whale	1.0
	*	near coast	porpoise	1.0
	6 ft.	at sea	dolphin	1.0

C:	n.pacific	salmon	1.0
	at sea	shark	1.0

a) starting at the root

```
global goal: whale
local goals: whale, cetacea
local subgoal A: blow hole

A: How does the creature breath?   through gills
C: Where does the fish live?   at sea

Result: The creature is NOT a whale but a shark.
```

The rejection of the local subgoals led to a rejection of all the local goals as well as the global goal. Table A was solved with the result <fish>. We went on forward and invoked table C, coming to a result matching the user's data.

b) without starting at the root

```
global goal     : whale
local goal B    : whale
local subgoal B : 25 ft.

B: What is the length of the cetacea?        6 ft.
A: How does the creature breath?             through a blow hole
B: Where does the cetacea live?              at sea

Result: The creature is NOT a whale but a dolphin.
```

The global goal was rejected. We jumped to the root node and solved table A. The result pointed to table B, which was partly solved before, thus we do not have to ask all of the questions again.

The sequence of action described above is an example of using two different generic tasks, namely hypothesis matching and hierarchical classification for building a more complex problem solver. Both implementations incorporate their own problem-solving strategy and knowledge representation appropriate for solving the specific problem. They partly share the same data description (organization of the hierarchy, attributes, values, etc.), complemented by specific table contents (distinction- and non-distinction-oriented tables).

6.3. Inductive learning

An important issue for almost all applications in the field of AI is the process of coding information, of incorporating knowledge into a predefined data structure. "...machine learning is not merely a short-cut method of building expert systems; learning is the key to intelligent behavior, and that is a lesson the AI community will have to learn if artificial intelligence is ever to deserve this title." [8, p.197].

There are several ways of learning: simple memorization, learning from instruction, learning by analogy, learning from examples, learning by discovery [1, p.87]. With respect to the hierarchical systems HICLASS and HIHYPO, a simple memorization would mean to encode a given non-redundant description of a class (for the field-guide-example the encoding of the provided key hierarchy). This requires that there is explicit knowledge about it. However, it will be more often the case that only examples are available describing instances of the class (e.g.: description of different rabbits). There are applications like CENTAUR that use these examples directly, for instance in the shape of prototypes. "The goal of the system is to confirm that one or more of the prototypes in the prototype network match the data in an actual case." [3, p.426].

Another way is to generalize the information given with a set of instances to obtain a good, in the best case non-redundant description of the class. We're talking the implementation of mechanical induction. "A simple inductive learning task is to induce a generalized description of a single concept or class of objects. A training set of individual instances of the concept is provided, each with a description. ... The goal of the learner is to establish a maximally specific generalization of the concept." [1, pp.88-89]

"...induction by machine is easy; useful induction is hard. The problem is not that machines cannot generalize. On the contrary, there are too many ways of generalizing" [8, p.211]. Some ways of generalizing include dropping conditions, internal disjunction, relaxing a constraint and making a constant into a 'don't care' variable. Model-driven (top-down) algorithms are guided by prior assumptions about the form of hypotheses, whereas data-driven (bottom-up) approaches are guided by patterns in the training data.

In the next sections, several approaches to solve the problem of inductive learning for expert systems will be covered in an overview manner.

6.3.1. Version space

Learning can be viewed as a search "through the space of all possible descriptions for those which are valuable for the task in hand" [13]. Since the number of valid descriptions can be astronomical, a heuristic method has to be found to guide the search. [8, p.198].

Mitchell developed an algorithm for searching the space of possible concepts (the *version space*): candidate-elimination, a cross between the *Sieve of Eratosthenes* and the *Binary Chop* [13].

The basic idea is to first list all possible descriptions and then to cross off those that do not apply to the training data. A partial ordering exists among the descriptions, from general to specific. The system maintains two boundary sets: S, the set of the most specific possible descriptions compatible with the training data so far, and G, the set of the most general possible descriptions. The two sets are gradually made to converge as more and more training instances are examined. The convergence is achieved as follows

1. When a positive instance is encountered, any description in G that does not cover it is eliminated, and all elements of S are generalized as little as possible so that they cover it.
2. When a negative instance is encountered, any description in S that covers it is deleted, and all elements in G are specialized as little as possible so that they no longer cover it.

Theoretically, this procedure is optimal, but "it starts to get into trouble with quite modest amounts of noise in the training data" [8, p.202].

6.3.2. Quinlan's ID3

In a discussion of Quinlan's ID3 algorithm Michie [12, p.222] states that the basic purpose of this algorithm is to "grow a small example set from an exhaustive set of situations stored on a database." The ID3 algorithm is iterative. A decision tree is formed using a subset of the training set. Then, the other members of the training set are classified using this tree. In the case of an misclassification the decision tree is rebuilt using additional instances. The iterative way of building the tree is said to be more efficient than to build it in a single step using all the data.

A major disadvantage of the original algorithm is stated in [12, p.224], referring to the results of the application of ID3 to a chess database: "The induced decision-tree rule looked good by the criteria of synthesis cost, compactness, and execution efficiency; but it made no sense to the chess expert." This is, especially for explanation features embedded in expert systems, a serious critique.

Forsyth [8, p.203] does list a number of ID3's shortcomings: "1. The rules are not probabilistic; 2. several identical examples have no more effect than one; 3. it cannot deal with contradictory examples; 4. the results are therefore over-sensitive to small alterations to the training database."

And also Jackson [10, p.450] criticizes the algorithm: "... you cannot consider additional training data without reconsidering the classification of previous instances. Also, ID3 is not guaranteed to find the simplest decision tree that characterizes the training instances, because the information-theoretic evaluation function for choosing attributes is only a heuristic. Nevertheless, ... its decision trees are relatively simple and perform well in classifying unseen objects."

Besides these problems though, ID3 has been incorporated into a number of commercial packages, like *ExTran* and *1st-CLASS*.

6.3.3. AQ11

The program AQ11, designed by Michalski, Larson and Chilausky "is the one which found better rules for soybean disease diagnosis than a human expert" [8, p.207]. The rules in AQ11 are generated in a language called VL1, where a description is a set of terms called 'selectors'.

Sample VL1 rule:

```
D3: [leaves = normal] [stem = abnormal]
    [stem cankers = below soil line]
    [canker lesion color = brown]
OR  [leaf malformation = absent] [stem = abnormal]
    [stem cankers = below soil line]
    [canker lesion color = brown]
```

The rule consists of two descriptions linked by an OR, whereas a description is a conjunction of terms. Each selector compares one variable with a constant (or range of constants). "AQ11 works in an incremental fashion, each step adding another conjunctive term (i.e. a new selector) starting off from a null description. The idea is to introduce new items of evidence one at a time, or a few at a time, and extend the growing rule to deal with them. The AQ11 method can be outlined in the following pseudo-code.

```
P = {set of Positive instances of the concept}
N = {set of Negative instances of the concept}
A = {answer set, initially empty}
G = {set of most general rules, initially null}

repeat until P is empty
 [choose an element p from P;
 apply 1-sided Candidate Elimination with p versus N
    using a conjunctive rule language;
 select a description g from G;
 append g to A;
 remove from P all elements covered by g;]
save and/or display A.
```

The main step in this top-level algorithm is best understood as a one-sided variant of the candidate-elimination algorithm; there is a G set of maximally general descriptions (..) but no S set. The place of the S set is taken by a single example,p. The method specializes G as little as possible to exclude all N (negative examples)" [8, pp.208-209].

"In the soybean work, the system made a complete pass through the data for each disease type, treating cases of that disease as positive examples and all other cases as negative examples. ... It is also possible to treat previously generated rules as negative examples... AQ11 rules start off very general and become more and more specific. It adds new terms to exclude negative examples, while still covering as many as positive cases as possible" [8, p.209].

For an example used several times in the discussion of HICLASS the algorithm is assumed to work like shown below (there are some parts of the algorithm which are not clearly described by Forsyth, but it is beyond the scope of this work to clarify these problems).

Example:

learning <whale>:

```
P = {cetacea    25 ft.    at sea}

N = {cetacea    6 ft.     near coast,
     cetacea    6 ft.     at sea     ,
     fish       1 ft.     n.pacific  ,
     fish       6 ft.     at sea     }

G = {cetacea}     (new conjunctive term)

p = {cetacea   25 ft.   at sea}
```

candidate elimination:

```
G covers p -> no change in G

G is specialized to not longer cover examples in N:
G = {cetacea, 25 ft.)
```

result:

```
A = {cetacea, 25 ft.}
```

6.3.4. Genetic algorithms

Genetic algorithms are inspired by evolution. They are "very general and robust in the face of noise" and they "are inherently parallel". "Evolutionary algorithms are very simplified versions of what goes on in nature, but they share the inherent parallelism of the natural process" [8, p.218].

For Forsyth [8, pp.216-219] the essence of a genetic algorithm is that "the expected number of 'offspring' of a rule is proportional to the success of that rule in the task being learned." The author looks at genetic algorithms as an advanced form of the 'Monte Carlo' method. "Using the basic Monte Carlo approach, a computer simply generates potential solutions, evaluates them and retains the one with the highest score. The longer the system runs, the greater the probability that it will find a solution within a preset distance from the optimum" [8, p.216]

Forsyth states that genetic algorithms "take the Monte Carlo idea one stage further by maintaining a population of potential solutions and biasing the search for new candidate solutions towards regions of the search space that have proved successful in past trials" [8, p.216].

6.4. An inductive learning algorithm for HIHYPO

The algorithm developed for creating a distinction-oriented representation for the HIHYPO system uses a number of ideas mentioned in the last sections and applies them (and other ideas) to the special problem.

"The choice of representation for encoding a system's knowledge is at least as important as the details of the learning algorithm it uses. ... It is also very convenient if the representation for the input data is the same as that for the descriptions (or rules)..." [8, p.198]. The choice about the knowledge representation for HICLASS as well as for HIHYPO has already been made: sets as described in 3.1.3. The attempt of building a distinction-oriented class description will be based on the same representation; the "system is said to employ the 'single representation trick'" [8, p.198].

An important fact is that the worldview of an HIHYPO system is somewhat limited (preenumerated solutions, attributes and values). Therefore the algorithm starts generating all possible descriptions and crossing off those that do not apply to the training data (the resemblance to the Sieve of Eratosthenes). The number of possible descriptions is likely to produce a combinatorial explosion; within the limits of a practical HIHYPO implementation though (something like maximal 13 attributes and maximal 26 values per attribute) it is still possible to deal with the problem.

Example:

Training data:

type	size	location	creature	weight	
cetacea	25 ft.	at sea	whale	1.0	(t1)
cetacea	25 ft.	near coast	whale	0.9	(t2)
fish	1 ft.	near coast	salmon	1.0	(t3)
fish	1 ft.	at sea	salmon	0.0	(t4)

Desired result:

type	size	location	creature	weight
*	*	at sea	whale	1.0
cetacea	*	near coast	whale	0.9
fish	*	near coast	salmon	1.0

Attribute sets:

type = {cetacea , fish}
size = {25 ft. , 1 ft.}
location = {at sea , near coast}

class 1 = <whale> (c1)
class 2 = <salmon> (c2)

A recursive algorithm generates descriptions, checks them against the training data and stores the descriptions that apply without contradictions supplemented by attributes showing the amount of reduction which could be achieved using this description, the class this description belongs to and the weight it carries. Contradictions to be checked are the following:

a) a description is true for several classes and the maximal combined weight of instances is greater than 1.0
b) a description is true for instances of one class carrying different weights
c) a description is true for a negative example for a class

For the example the generated descriptions and the result of the contradiction check would be the following:

```
d1  = {cetacea ,  *     ,    *      } applies to t1,t2; different weights; reject
d2  = {cetacea , 25 ft. ,    *      } applies to t1,t2; different weights; reject
d3  = {cetacea , 25 ft. , at sea    } applies to t1; take
d4  = {cetacea , 25 ft. , near coast} applies to t2; take
d5  = {cetacea ,  1 ft. ,    *      } does not apply
d6  = {cetacea ,  1 ft. , at sea    } does not apply
d7  = {cetacea ,  1 ft. , near coast} does not apply
d8  = {cetacea ,  *     , at sea    } applies to t1; take
d9  = {cetacea ,  *     , near coast} applies to t2; take
d10 = {fish    ,  *     ,    *      } applies to t3,t4; t4 is neg. example; reject
d11 = {fish    , 25 ft. ,    *      } does not apply
d12 = {fish    , 25 ft. , at sea    } does not apply
d13 = {fish    , 25 ft. , near coast} does not apply
d14 = {fish    ,  1 ft. ,    *      } applies to t3,t4; t4 is neg. example; reject
d15 = {fish    ,  1 ft. , at sea    } applies to t4; t4 is neg. example; reject
d16 = {fish    ,  1 ft. , near coast} applies to t3; take
d17 = {fish    ,  *     , at sea    } applies to t4; t4 is neg. example; reject
d18 = {fish    ,  *     , near coast} applies to t3; take
d19 = {   *    , 25 ft. ,    *      } applies to t1,t2; different weights; reject
d20 = {   *    , 25 ft. , at sea    } applies to t1; take
d21 = {   *    , 25 ft. , near coast} applies to t2; take
d22 = {   *    ,  1 ft. ,    *      } applies to t3,t4; t4 is neg. example; reject
d23 = {   *    ,  1 ft. , at sea    } applies to t4; t4 is neg. example; reject
d24 = {   *    ,  1 ft. , near coast} applies to t3; take
d25 = {   *    ,   *    , at sea    } applies to t1,t4; combined weight=1.0; take
d26 = {   *    ,   *    , near coast} applies to t2,t3; reject
```

Descriptions still valid:

```
the second set has to be read as:
{result, weight, number of examples substituted)

d1  = {cetacea , 25 ft. , at sea    } {c1, 1.0, 1}
d2  = {cetacea , 25 ft. , near coast} (c1, 0.9, 1}
d3  = {cetacea ,  *     , at sea    } {c1, 1.0, 1}
d4  = {cetacea ,  *     , near coast} {c1, 0.9, 1}
d5  = {fish    ,  1 ft. , near coast} {c2, 1.0, 1}
d6  = {fish    ,  *     , near coast} {c2, 1.0, 1}
d7  = {   *    , 25 ft. , at sea    } {c1, 1.0, 1}
d8  = {   *    , 25 ft. , near coast} {c1, 0.9, 1}
d9  = {   *    ,  1 ft. , near coast} {c2, 1.0, 1}
d10 = {   *    ,   *    , at sea    } {c1, 1.0, 1}
```

For each class, descriptions have to be found that are maximal general. The criterion for "maximal general" is the minimum number of values different than '*'. The algorithm performs the following loop:

```
for each class do begin
  repeat
    look for maximal general description
    check if description is already covered by result set
    if not then append this description to the result set
    delete this description in the description set
  until there are no more descriptions for the class
end
```

For the example of c2 (<salmon>) the sequence of action is the following:

Description set (D):

```
d1  = {fish    ,  1 ft. , near coast} {c2, 1.0, 1}
d2  = {fish    ,  *     , near coast} {c2, 1.0, 1}
d3  = {  *     ,  1 ft. , near coast} {c2, 1.0, 1}
```

Result set (R):

```
r1  = {  *     ,    *    , at sea    , whale , 1.0, 1}
r2  = {cetacea ,    *    , near coast, whale , 0.9, 1}
```

Look for maximal general description

```
d2  = {fish    ,  *     , near coast} {c2, 1.0, 1}
```

Description already covered by result set? NO

Append to result set

```
r1  = {  *     ,    *    , at sea    , whale , 1.0, 1}
r2  = {cetacea ,    *    , near coast, whale , 0.9, 1}
r3  = {fish    ,    *    , near coast, salmon, 1.0, 1}
```

Delete in description set

```
d1  = {fish    ,  1 ft. , near coast} {c2, 1.0, 1}
d2  = {  *     ,  1 ft. , near coast} {c2, 1.0, 1}
```

Look for maximal general description

```
d2  = {  *     ,  1 ft. , near coast} {c2, 1.0, 1}
```

Description already covered by result set? YES

Delete in description set

```
d1  = {fish    ,  1 ft. , near coast} {c2, 1.0, 1}
```

Look for maximal general description

```
d1  = {fish    ,  1 ft. , near coast} {c2, 1.0, 1}
```

Description already covered by result set? YES

Delete in description set

Result:

```
r1  = {   *     ,    *    , at sea     , whale , 1.0, 1}
r2  = {cetacea  ,    *    , near coast, whale , 0.9, 1}
r3  = {fish     ,    *    , near coast, salmon, 1.0, 1}
```

The criterion for the decision "Description already covered by result?" is the following:

```
if (result_value<>'*')and(description_value<>'*')and(result_value<>description_value)
then included:=false
```

Using this criterion we achieve a minimal set of results.

Changing the criterion to

```
if (result_value<>'*')and(result_value<>description_value)
then included:=false
```

the description

```
d2  = {   *     ,  1 ft. , near coast} {c2, 1.0, 1}
```

would have been valid, thus the result set would have been

```
r1  = {   *     ,    *    , at sea     , whale , 1.0, 1}
r2  = {cetacea  ,    *    , near coast, whale , 0.9, 1}
r3  = {fish     ,    *    , near coast, salmon, 1.0, 1}
r4  = {   *     ,  1 ft.  , near coast, salmon, 1.0, 1}
```

which is a proper, but not minimal result set.

The check if a description is already included in another description could have been made while checking the descriptions against the examples; we simply would have performed another check against the already defined descriptions. That this was not done is due to the attempt to maintain a simple logical flow of the algorithm.

The algorithm works properly for all training data sets checked so far. The combinatorial explosion problem is limited by limiting the number of possible combinations. It is not a perfect algorithm for applications with a higher amount of attributes or values; within the context of HIHYPO though it is a useful approach. Noise in the training data sets causes trouble, thus the algorithm works perfectly only with an idealized, noise-free training set.

The example given in [10, p.447] could properly be solved by the algorithm:

Outlook	Temperature	Humidity	Windy	Class
sunny	hot	high	false	N
sunny	hot	high	true	N
overcast	hot	high	false	P
rain	mild	high	false	P
rain	cool	normal	false	P
rain	cool	normal	true	N
overcast	cool	normal	true	P
sunny	mild	high	false	N
sunny	cool	normal	false	P
rain	mild	normal	false	P
sunny	mild	normal	true	P
overcast	mild	high	true	P
overcast	hot	normal	false	P
rain	mild	high	true	N

A generalization of the class P (N denoting negative examples for P) is given with:

P = [outlook=overcast] or
[(outlook=sunny)and(humidity=normal)] or
[(outlook=rain)and(windy=false)]

7. Conclusions

An expert system shell solving the generic task of hierarchical classification has been created. Crucial aspects have been challenged from both a theoretical and an implementational point of view. Issues of knowledge representations, control strategies, inductive learning, ways of handling uncertainty, ambiguity, and contradictions, and more have been covered. Additionally, the development of a hierarchical hypothesis matcher has been proposed. A special algorithm for inductive learning has been developed and implemented.

The main goals of the research could successfully be achieved. The next logical steps would be to

- perform a complexity study
- implement the proposed but yet not integrated features of HICLASS
- improve the HICLASS system as described in 4.2.2.
- expand the research on learning, explaining the reasoning process, and concluding facts
- implement HIHYPO, a hierarchical hypothesis matcher
- design a complex problem solver combining HICLASS and HIHYPO

Ultimately, the system could be perfected to market it.

References

[1] Black. Intelligent Knowledge Based Systems - an introduction. Berkshire/England: Van Nostrand Reinhold (UK), 1987.

[2] Bochmann/Steinbach. Logikentwurf mit XBOOLE. (Logic Design with XBOOLE). Berlin: Verlag Technik, 1991.

[3] Buchanan/Shortliffe, eds. Rule-Based Expert Systems (The MYCIN Experiments of the Stanford Heuristic Programming Project). Reading MA: Addison Wesley, 1984.

[4] Burton, ed. The world encyclopedia of animals. New York: World Publishing, 1972.

[5] Chandrasekaran. Building blocks for knowledge-based systems based on generic tasks: The classification and routine design examples. In: Liebowitz/De Salvo, eds. 1989. Structuring Expert Systems. Englewood Cliffs NJ: Yourdon Press, 1989.

[6] Doukidis/Whitley. Developing Expert Systems. Bromley, Kent, U.K.: Chartwell-Bratt, 1988.

[7] Dresig/Kuemmling/Steinbach/Wazel. Programmieren mit XBOOLE. (Programming with XBOOLE). Chemnitz/Germany: TU Chemnitz Publication Series, 1992.

[8] Forsyth. Inductive learning for Expert Systems. In: Forsyth, ed. Expert Systems - principles and case studies. London: Chapman and Hall Computing. Lund/Sweden: Chartwell-Bratt Studenylitteratur, 1989.

[9] Frost. Introduction to Knowledge Base Systems. New York: MacMillan Publishing Company, 1986.

[10] Jackson. Introduction to Expert Systems. Reading MA: Addison-Wesley, 1990.

[11] Liebowitz/De Salvo, eds. Structuring Expert Systems. Englewood Cliffs NJ: Yourdon Press, 1989.

[12] Michie, ed. Introductory readings in Expert Systems. New York: Gordon and Breach, 1984.

[13] Mitchell. Generalization as a search. In: Artificial Intelligence, 18 (1982), pp.203-26.

[14] Wazel. An AI system that plays 'Guess what it is'. Project paper in SAN 586 at Miami University, Oxford OH, 1991.

Further Reading

[15] Frenzel. Crash course in Artificial Intelligence and Expert Systems. Indianapolis, IN: Howard W. Sams & Co, 1987.

[16] Gupta/Kandel/Bandler/Kiszka. Approximate reasoning in Expert Systems. Amsterdam: North-Holland, 1985.

[17] Hayes-Roth/Waterman/Lenat, eds. Building Expert Systems. Reading MA: Addison-Wesley, 1983.

[18] Hendler, ed. Expert Systems: the user interface. Norwood NJ: Ablex Publishing Corporation, 1988.

[19] Kruse/Schwecke/Heinsohn. Uncertainty and vagueness in knowledge based systems. Berlin: Springer-Verlag, 1991.

[20] Levine/Drang/Edelson. AI and Expert Systems - a comprehensive guide. New York: McGraw-Hill, 1990.

[21] Martin/Oxman. Building Expert Systems. Englewood Cliffs NJ: Prentice Hall, 1988.

[22] McDaniel. An introduction to decision logic tables. New York: PBI, 1978.

[23] Neguita. Expert Systems and fuzzy systems. Menlo Park CA: The Benjamin/Cummings Publishing Company, 1985.

[24] Parsaye/Chignell. Expert Systems for Experts. New York: John Wiley & Sons, 1988.

[25] Pedersen. Expert Systems Programming. New York: Wiley, 1989.

[26] Reichgelt. Knowledge representation: An AI perspective. Norwood NJ: Ablex Publishing Corporation, 1991.

[27] Rich/Knight. Artificial Intelligence. New York: McGraw-Hill, 1991.

[28] Steinbach/Dresig. XBOOLE/XB_PORT - Informationen fuer den Programmierer (XBOOLE/XB_PORT - Programmer's Guide). Unpublished. TU Chemnitz, Germany, 1989.

[21] Wazel. Der Einsatz von FIRST CLASS in der computergestuetzten Lehre und Ausbildung am Beispiel des Fachs Betriebswirtschaft - Jahresarbeit. (The application of FIRST CLASS to computer based training for the example of business management courses - term paper). TU Chemnitz, Germany, 1989.

Appendix A

List of files on the program disk

README.TXT	: brief description of the system
HIEDIT.EXE	: program HIEDIT
HICLASS.EXE	: program HICLASS
HICLASS.HLP	: help file for system
HICLASS.CON	: control file for help file
EXAMPLE1.HIT	: root table for first example (covered in section 4.2.1.)
BIRD.HIT	
CARNIVOR.HIT	
CETACEAN.HIT	
FISH.HIT	
MAMMAL.HIT	
NOFISH.HIT	
NOTCETAC.HIT	
SIZE.HIT	
TEMPCH.HIT	
UNGULATE.HIT	
EXAMPLE2.HIT	: root table for second example (covered at the end of section 6.4.)

Appendix B

The Software Engineering aspect of the project

Report for Independent Study

Table of contents

0. Introductory comments 3

1. The traditional software life cycle 4

2. Characteristics of good design 6
    2.1. Modularity 6
    2.2. Levels of abstraction, Information hiding 6
    2.3. Coupling 7
    2.4. Cohesion 8
    2.5. Control issues 8

3. System design 10
    3.1. Process-oriented design techniques 10
        3.1.1. Modular Programming 10
        3.1.2. Functional Decomposition 10
        3.1.3. Data Flow Design Methods 11
        3.1.4. Data Structure Design Methods 12
            3.1.4.1. Jackson's Design Methodology 13
            3.1.4.2. Warnier's Methodology 13
        3.1.5. HIPO 14
    3.2. Data-oriented design techniques 14

4. Program design 15
    4.1. Top-down design 15
    4.2. Nucleus extension 16
    4.3. Bottom-up design 16

5. Implementation and Testing 17

6. Object-oriented development 19
    6.1. Introduction 19
    6.2. Basic concepts 20
        6.2.1. Objects 20
        6.2.2. Classes 20
        6.2.3. Inheritance 20
        6.2.4. Polymorphism 21
        6.2.5. Message passing 21
    6.3. Object-oriented software life cycle 22
    6.4. Some advantages of object-orientation 22

7. The HICLASS project 24

References 26

0. Introductory comments

"Systematic software development practices are applicable to virtually any class of computer-based systems which will have a lifetime considerably longer than its development time and which requires more than a single person to carry out the design and development" [13, p.27]. The following discussion will be concerned about systematic software development methodologies. The traditional software life cycle with its several stages will be introduced. The object-oriented software development methodology will be covered. Due to the fact that the HICLASS system has been developed by 'a single person', the features of the life cycles concerned about management and communication during the development of the system will not be addressed in detail. The discussion below will mainly be focused on the methods and techniques which had to be considered and/or which have actually been applied while developing HICLASS. It should be understood that this is not a complete overview about all software engineering methodologies and concepts, but a selection of issues important within the scope of the HICLASS project. Finally, a brief discussion will show which methodologies have actually been used to create the HICLASS system.

1. The traditional software life cycle

Back in the 1960s, "every piece of software for every information system was a 'custom design', with no consistent pattern to follow and little experience from previous efforts" [13, p.25]. There was no systematic approach to system design and development. This was one of the reasons for research on Software Engineering with the idea to apply an "engineeringlike form of discipline" to building software systems. A number of concepts were developed including top-down design, modularity and structured programming. One of the most important steps though was the development of a software life cycle, with which it became possible to merge techniques for software production with adequate management techniques. Several stages of software development are defined within the framework of the life cycle including requirements analysis and definition, design and maintenance. Aspects of management and communication play an important role throughout the whole process serving to "tie the stages together and provide the organizational environment in which the technical procedures can be made effective" [13, p.26].

"Ideally, we would like to derive our programs from a statement of requirements in the same sense that theorems are derived from axioms in a published proof", but "we will never find a process that allows us to design software in a perfectly rational way." [8, p.251]. There are a number of reasons for that. Users might not be able to exactly specify their needs, many details become clear during the implementation, projects are subject to change due to external reasons, errors will occur, parts of the software might be shared with other projects and therefore not be the ideal software for the current project, and so on. Parnas/Clements [8] suggest that nevertheless an ideal process should be assumed, and that documentation should be produced that makes it appear as if the software was designed in an ideal manner. They talk about "faking a rational design process".

The traditional description of the software life cycle is based on the "waterfall" model. It "attempts to discretize the identifiable activities within the software development process as a linear series of actions, each of which must be completed before the next is commenced" [5, p.143]. There are several levels of detail with which the model is described. At the most general level there are three phases defined [5]:

- analysis
- design
- construction/implementation

During the analysis phase, the needs of the user are analyzed and a feasibility study is done. The design phase includes various concepts of design (system and program design). In the last phase of the model, programs are written and tested, the system is delivered and maintained.

Several authors use different approaches to subdivide the three phases [5,9]; these approaches only differ in the level of detail. The following description is used in [5].

- user requirements analysis
- user requirements specification
- software requirements specification
- logical design (system design)
- physical design (program design)
- implementation/coding
- program testing: units
- program testing: systems
- program use
- software maintenance

The first two stages try to answer a WHAT-question; the attempt is to identify the problem. Starting with the software requirements specification, the question HOW is beginning to be answered, moving the process towards a solution. "The design stage is perhaps the most loosely defined since it is a phase of progressive decomposition toward more and more detail and is essentially a creative, not a mechanistic, process" [5, p.144].

There are several problems with a traditional approach using the classical life cycle. These problems include that there is "no iteration, no emphasis on reuse and no unifying model to integrate the phases" [6, p.40]. Alternative models like the spiral and fountain model (as described in section 6.3) have been developed to overcome these problems. And, as described above, the ideal process can be "faked", while for instance repeating some steps, and performing an iteration back to a previous stage.

2. Characteristics of good design

There is no method developed for software design which can claim to be completely systematic. Method-independent guidelines have emerged to supplement the design methods in providing guidance during the process of design, and to allow judgments on the quality of the designed software [2]. The characteristics of a good software design discussed below center around the idea of modularity. The discussion is based on [2,9,12,13,16].

2.1. Modularity

Modularization can be defined as dividing a program into parts on some systematic basis. A module is "a functional entity with a well-defined set of inputs and outputs. ... A module is well-defined if all inputs to the module are essential to the function of the module and all outputs are produced by some action of the module" [9, p.140]. "The term module is used to refer to a set of one or more contiguous program statements having a name by which other parts of the system can invoke it and preferably having its own distinct set of variable names" [12, p.244]. When each activity of the system is performed by exactly one module, the system is said to be modular.

There are different points of view on how big a module should be. Some authors suggest that a module should occupy no more than one page of text, others prefer even smaller modules (seven lines or less). The smaller the module, the higher is the number of modules, and so is the number of levels, which may result in more confusion. On the other hand, small modules are much easier to comprehend, since we only should look at one module at a time. One could also argue that small modules increase the overhead of subroutine linkage. "The question here is whether it is more important for a program to be easy to understand or whether it is more important for it to run quickly" [2, p.28]. One idea is to develop the software using small modules, and to rewrite particular procedures which are invoked frequently, although it is not very likely that the subroutine linkage has a big impact at all. Rather, it has been shown that about 50% of the execution time of a program is spent on executing about 10% of the code. Hence, it is important to optimize these parts of the code.

Modules with the same number of program lines can have a different complexity. One among many attempts to measure complexity is McCabe's cyclomatic complexity. McCabe asserts that complexity depends on the decision structure of the program. If the cyclomatic complexity of a particular module, derived by counting the number of predicates and adding one, is greater than ten, then it is too complex. Criticisms of this method include that the measure ignores, for example, references to data. [2, pp.30-31]

2.2. Levels of abstraction, Information hiding

Usually, the modules at one level refine those in the level above; the top level is the most abstract one. The modules are arranged in levels of abstraction. High-level modules give the opportunity to view the problem as a whole, while hiding the details of the functional components.

A similar idea is that modules could hide the internal details and processing from one another. *Information hiding*, or encapsulation, suggests that for each data structure the structure itself, the statements that access the structure and the statements that modify it should be part of a single module [2, p.32]. The encapsulated data cannot be accessed directly, only via one of the procedures associated with the data. This principle supports an easy changeability, independent development and better comprehensibility. The concept of information hiding is one of the underlying principles of object-oriented design.

2.3. Coupling

Independence between modules is desirable because it is easier to understand a module if its function is not tied to others, and it is easier to modify an independent module. Additionally, the spread of damage may be limited, if an error occurs in one module. Two criterias have been developed to measure the degree of module independence: coupling and cohesion. The goal is to create modules in a way that there is a minimum of interaction between modules (low coupling) and a high degree of interaction within a module (high cohesion).

Coupling measures how much modules depend on each other. Highly coupled modules are very dependent on each other, loosely coupled modules have some independence, whereas uncoupled modules have no interconnection at all. Coupling depends on several things [9, pp.143-145]:

- The references made from one module to another.
- The amount of data passed from one module to another.
- The amount of control one module has over another.
- The degree of complexity in the interface between modules.

Coupling represents a range of dependence, some types of coupling are less desirable than others. *Content coupling* is the least desirable. It occurs when one module actually modifies another. This might occur when one module modifies an internal data item in another module (or even worse, when the code of the other module is altered), or when a module branches into the middle of another module (also referred to as "entering at the side door").

Common coupling appears when data items are put in a global or common data area to which two or more modules have access to. A number of problems arise doing so. Adding new data to the shared data may cause a name clash with an existing local data item within one of the involved modules. In order to understand an individual module it is necessary to understand all of the shared data. Additionally, it can be difficult to determine which module is responsible for having set a variable to a particular value.

Control coupling between two modules appears when one module passes flags (also called switches) to control the activity of another module. It is impossible for the controlled module to function without direction from the controlling one. In order to minimize control coupling, it should be tried to split a single multi-purpose module into several, each carrying out a single action.

A better way than passing a set of control flags and data items is to use a data structure to pass information from one module to another. The data structure allows an argument list to be used. *Stamp coupling* occurs when the data structure itself is passed, whereas *data coupling* is performed if only data are passed.

The most desirable coupling is achieved without any transfer of control between modules. One module passes a serial stream of data to another. The outputting module has no access to the data, once it has released them. This option though is not widely available in most programming systems.

2.4. Cohesion

The nature of the interactions *within* a module is described by cohesion. "The more cohesive a module, the more related are the internal parts of the module to each other and to the functionality of the module" [9, p.147]. The goal is to make a module as cohesive as possible. Again, there are several types of cohesion, ranging from less to most desirable.

Coincidental cohesion describes the fact that the parts of a module are completely unrelated to one another. With *logical cohesion*, several logically but not functionally related functions are placed in the same module. *Temporal cohesion* occurs, when a module performs a set of functions that are only related in time, such as initialization operations. A module is *procedurally cohesive*, if functions are grouped together in a module just to insure a certain order of performance. If functions acting on common data are grouped together in a module, this module is said to be *communicationally cohesive*. *Sequential cohesion* occurs, when the output from one part of the module is the input to the next part. The ideal type of cohesion is *functional cohesion*, in which every single operation in the module contributes towards the performance of a single well-defined task.

2.5. Control issues

Another aspect in measuring the quality of a piece of software is focused on the control of several modules by a single module. "Fan-in is the number of modules controlling a particular module, and fan-out is the number of modules controlled by a module" [9, p.150]. Modules with a low fan-out have to be preferred, because a high fan-out can indicate that a module is performing more than one function. It is often useful to create a set of utility modules which can be called from many other modules. These utility modules have a high fan-in. In general, the attempt is to create modules with a high fan-in and a low fan-out.

Another aspect is that modules should not effect other modules over which they have no control. "The *scope of control* of a module is that module plus all modules that are ultimately subordinate to that module. ... The *scope of effect* of a decision is the set of all modules that contain some code whose execution is based upon the outcome of the decision" [12, p.250]. No module should be in the scope of effect if it is not in the scope of control.

Summarizing the above discussion, the characteristics of a good design are the following [9]:

- low coupling of modules
- highly cohesive modules
- minimal number of modules with high fan-out
- scope of effect of a module limited to its scope of control

3. System design

This stage of the classical software life cycle is also be referred to as logical [5] or architectural [15] design. "A design is a determination of the modules and intermodular interfaces that satisfy a specified set of requirements" [9, p.140]. Various design alternatives are analyzed, and different solutions are evaluated according to the existing constraints, such as machine resources, development time or costs, and operational costs. In the system design, "the emphasis is on determining the structure of the system, decomposing the system into modules, and precisely specifying the interfaces between modules" [13, p.30-31]. Data items and structures are described in a relatively abstract way.

There are different ways of classifying techniques developed for the system design stage. Pfleeger [9] divides the approaches into *decomposition* and *composition*, while Yau/Tsai [15] emphasize the distinction between *process-oriented* and *data-oriented* approaches. The latter definition will be used further on. Some of the methodologies described below are not limited to an use for the system design stage, the attempt is to use one consistent approach throughout several stages of the life cycle.

3.1. Process-oriented design techniques

"The process-oriented design technique emphasizes the process of decomposition and structure in creating a software architecture" [15, p.714].

Important process-oriented design techniques are:

- modular programming
- functional decomposition
- data flow design methods
- data structure design methods
- HIPO

In the next sections, these techniques will be discussed in more detail.

3.1.1. Modular Programming

A complex system is divided into several parts, and each of the modules only performs a single function. The module size is small to allow an efficient testing. Following coding and test of single modules, they are integrated. Then, the whole system is tested. Advantages of this approach are that it is easier to write, test and maintain the programs. Most of the other methods desribed below use modular programming.

3.1.2. Functional Decomposition

Functional decomposition [2,5,15] focuses on the functions that a program has to carry out. The system is viewed in terms of what it is intended to do. The technique is a *top-down* method; it starts with the overall task of the program. But it is also called *stepwise refinement*. At any stage of decomposition "the solution is expressed in terms of operations that are assumed to be available and provided by an abstract (or virtual) machine" [2, p.46]. "...each module is characterized by a designer's decision. Only certain information of this module is needed by other modules, and communications between modules are through well-defined interfaces." [15, p.714]. The method can also be viewed at as a variety of *structured programming*.

There are two basic approaches to functional decomposition - *breadth first* and *depth first*. Using a breadth first approach, the design is refined level-by-level, growing a tree structure. With depth first, the focus is directed on only one branch of the tree at a given time, developing the branches one after another. Design tools used for functional decomposition include data flow diagrams, data dictionaries and structure charts.

The functions of the system play a more important role than the data. The data structures are derived during the decomposition as they are needed, and when it becomes clear what needs to be done with them. Thus, the data are tailored to the operations. System developed this way are very likely to be unable to take new data structures or new functions into account.

Functional decomposition is very flexible and generally applicable. It is most useful though if the procedural steps of the desired system are clearly evident. The method "guides our thinking but allows us plenty of scope for creativity". It requires "significant creativity and judgement to be employed" [2, p.50].

Disadvantages of the approach include that it is somewhat unpredictable and that it is hard to know, if the best possible design was created; in fact, it is complicated to choose between different designs. The method is not so well-defined as others. This might be on of the reasons that it has not been marketed yet.

3.1.3. Data Flow Design Methods

With a data flow design method, information flow is the driving force for the design process. Various mapping functions are used to transform information flow into software structure. The method suggests that software should be build from parallel programs, even if it is widely used for designing sequential programs.

Structured design [2,12,13,15] was originally developed by Constantine, and advanced by Yourdon and Myers. It has its origins in the era of modular programming, and it suggests a "definite procedure by which the structure of a large program or software system could be expressed in terms of consistent modules" [2, p.81]. It tries to overcome the shortcomings of functional decomposition since it provides criteria to compare alternative designs, and to determine their relative quality. The method does not automatically lead to a unique, ideal solution. Alternative designs are possible.

In structured design, the data flow of a problem is mapped into its software structure using some design analysis technique. Some of the characteristics for a good design as discussed in section 2 of this paper, have their origins in the research for this methodology. One of the key issues is modularity, which is mainly measured in terms of cohesion and coupling. The "goal of structured design is to create system structures in which the modules have high cohesion and low coupling" [13, p.32]. The method is not very helpful in the detailed design and implementation stages.

In a data flow design, the flow of data and the transformation that will act upon these flows is examined. A vital step is to draw the data flow diagram with bubbles representing a "transformation that converts an input flow into an output flow". There is "no definite, systematic way" [2, p.74] of doing this. Working in a non-parallel environment, the data flow diagram has to be transformed into a structure for a sequential program, since the bubbles can be seen as programs that input a serial stream of data from one bubble and output a serial stream to another. The end product of the method is the structure chart for the software showing the modules and the interaction between these modules.

Besides the fact that data flow design attempts to create a design with the best possible modularity, the method is based on the idea that most programs have a similar overall structure as shown in figure 3.1.3.1. "In general, a piece of software will require that several transformations are carried out on its input data streams and that, after the main processing, several transformations are carried out on its output data streams" [2, p.78].

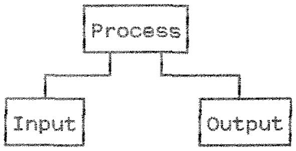


Figure 3.1.3.1. Overall structure of most programs

Structured design is closely related to the *Structured Analysis Design Technique*, which is based on *Structured Analysis*. Structured Analysis "is a graphical language used for explicitly expressing hierarchical and functional relationships among any objects and activities" [15, p.714]. The Structured Analysis Design Technique includes management planning and configuration control procedures, and is most effective in the early and late stages of the software life cycle.

3.1.4. Data Structure Design Methods

The methodologies for data structure design emphasize the structure of a problem. The two different approaches developed by Jackson and Warnier can be used in the system as well as in the program design phase.

3.1.4.1. Jackson's Design Methodology

The basic idea behind the Jackson method [2,5,13,15] is "that the structure of a program should match the structure of the file or files that the program is going to act on" [2, p.52]. "The basic structure of a software system is determined by the structure of the data it processes, and software is viewed as a mechanism by which input data are transformed into output data" [15, p.715]. In [1] it is argued that essentially functional decomposition takes place, but now the data structures are used to assist in this process. The Jackson method is not concerned about detailed operations, only with the structure of the program.

The method starts with analyzing the structure of the data which needs to be processed. A data structure diagram is produced, organized in a hierarchical fashion. The structures available correspond to the constructs of structured programming - sequence, selection, repetition. The data structure diagram is then transformed into a program structure diagram. Next, a list of all elementary operations that the program will have to carry out is produced, and these operations are placed in their appropriate positions in the program structure diagram. The final step is to transform this diagram into pseudocode called schematic logic.

Problems can arise, if several files have to be processed. Data structure diagrams for each input and output file are drawn, and a program structure is obtained which incorporates all the structural aspects of all the different diagrams. It can happen though, that two or more data structures cannot be mapped into a single program structure. Jackson describes this situation as a *structure clash*. Several approaches have been developed to resolve such a clash, one is to "create an intermediate structure that can successfully interface with both the input and the output structure" [13, p.32].

Working with the Jackson method, a workable design is created, which is not necessarily the best design. The method is recognized as the most systematic design method currently available [2], and has therefore heavily been used. It is said to be non-inspirational, rational, teachable, consistent, simple and easy to use, and it produces designs that can be implemented in any programming language [2].

3.1.4.2. Warnier's Methodology

The methodology developed by Warnier is very similar to Jacksons approach. Warnier though provides more detailed procedures to software design. Four kinds of design representation are used: data organization diagram, logical sequence diagram, instruction list, and pseudocode [15].

3.1.5. HIPO

HIPO (Hierarchical-Input-Output) [11,13,15] was developed by IBM primarily as a documentation aid. It consists of two basic components: "a hierarchy chart, which shows how each function is divided into subfunctions; and input-process-output charts which express each function in the hierarchy in terms of its input and output" [11]. The design process is an iterative top-down activity, modular decomposition is achieved; the hierarchy chart and the input-process-output charts are developed concurrently. HIPO has the "ability to represent the relationship between input/output data and software process" and the ability "to decompose a system in a hierarchical way without involving logic details" [15, p.715]. The technique can be used in system design as well as in program design, testing and maintenance. It is easy to learn and use and has widely been applied.

3.2. Data-oriented design techniques

"A data-oriented design technique emphasizes the data design components of a software system and the techniques for deriving the data design" [15, p.715]. Process-oriented design techniques are focused on the functional aspect of the problem. This is also true for the data structure design methods, which use the data structures to assist in the process. Data-oriented design techniques, on the other hand, favor the data and derive functionality to transform data. Yau/Tsai [15] discuss the object-oriented design technique as belonging to this category of methodologies. A different point of view is that the object-oriented method is more a blend of process- and data-oriented design techniques; it attempts to achieve a balance between both. The latter approach will be followed here, thus object-oriented design is understood as a third category. This can also be justified by the fact that a completely different life cycle is used for the object-oriented case. One example of a data-oriented design is the conceptual database design methodology. This technique is related to the formal specification method, which describes how "programs can be built systematically from a formal specification of the data they deal with" [15, p.715]. Techniques of automated programming and the proof for correctness of a program can be developed based on formal specification. The conceptual database design methodology can "guide a designer in the process of translating data and requirements specifications into a database conceptual schema" [15, p.715]. The aim is to establish a unified conceptual model, such that the software design process is proceeding while building a data model.

4. Program design

This stage of the software life cycle is referred to as physical [5], detailed [13,15] or program [2,9,16] design. "During detailed design, the emphasis is on the selection and evaluation of algorithms to carry out the logical steps specified for the individual modules" [13, p.31]. "Program design defines modules and intermodular interfaces so that each module of the system corresponds to a new set of modules containing program specifications" [9, p.185]. The specifications describe the input, output, and processing to be performed; they are technical and detailed, referencing specific data formats and describing the steps of the algorithms. In program design it is fine-tuned *what* should be done before it is considered *how* to do it. Modularity is a key factor in good program design; characteristics of a good design have been described in section 2. Well-known design tools are flowcharts, pseudocode and Nassi-Shneiderman charts [9,15]. There are basically three approaches to program design [16], which will be discussed below in more detail.

4.1. Top-down design

The first step in top-down design [9,16] is a statement of the function of the program. The subfunctions are then identified in a recursive fashion, such that each of the subfunctions may be further subdivided until a level is reached where the parts are easily comprehended. Only data and control information and structures necessary for a particular module are defined, while details of the design at lower levels remain hidden. Top-down design (also referred to as functional decomposition) includes the strategies of stepwise refinement and transactional analysis.

With stepwise refinement, the decomposition stages are treated "as programs in successively lower level and more procedural programming languages" [16, p.72]. A pseudocode notation with structured programming control structures can be used. This pseudocode, also known as a program design language (PDL) may be viewed as a very-high-level programming language. There are formal PDLs, which impose a programming language-like syntax upon the user, and informal PDLs. The PDL is used to describe the interfaces between the given module and other modules, to give a brief statement of the function performed by the module, and to describe the logic used in realizing that function. There are several advantages of using a PDL compared to flowcharts, among others the fact that they are machine-processable. At each stage of the refinement process, details of the data manipulation are suppressed, and will be addressed later. Decisions regarding control structures though cannot be postponed. One advantage of stepwise refinement is that attention is focused on developing a correct program, not just on understanding the problem situation. A disadvantage is that later stages may uncover the need for structural changes, which might result in changes in earlier designs.

Transactional analysis is based on the analysis of data flow through the program. The processes that transmit and transform a single input element are analyzed. The steps involved in identifying data streams and processes are similar to those followed in the data flow design methods for system design.

4.2. Nucleus extension

Rather than starting with the function of a program as a whole, the starting point in nucleus extension [16] is a collection of contributory functions. Two main strategies have been developed: Parnas' module specification and Jackson's hierarchical modular design.

The module specification strategy was developed as a means of providing building blocks for defining families of system programs. The strategy starts with identifying areas of design decision where there are competing solutions. These areas are isolated in separate modules. Some of the modules might be reused in other programs or systems, whereas others are specific to the program specifications. The decisions are further broken down into parts identifying lower-level decision areas. After algorithms, data structures, and access modes have been selected, the flow of control is designed, and the modules are recombined.

With hierarchical modular design, program structure is based on the structure of the input and output data. The strategy works best with highly structured data, and it is based on hierarchical diagrams. Its philosophy is basically the same as described in 3.1.4.1. The input and output charts combine elements of logical structure with elements of physical structure, with major focus on a logical description of the data.

4.3. Bottom-up design

"Bottom-up design starts by identifying what might be called the utility functions needed by a program" [16, p.89]. The utility modules are very low-level and they are generally useful, and might therefore be reused in other programs or systems. Once the low-level modules have been designed, they are used in the definition of higher-level functions. These modules in turn contribute to a higher level, and so on until the entire program design has been built. Since a utility function can be shared by several higher level functions, coupling among modules usually increases [9].

5. Implementation and Testing

In the traditional software life cycle, the phase of design is followed by implementation/coding, by a test of units, and then by a test of the whole system. All modules are designed and coded; the low-level components are tested first. Then, modules are combined into subsystems, which are tested again. And so on, until the complete system is built and tested. There are a couple of problems connected with this bottom-up approach [2]. A lot of time has to be spent on the construction of test data and test harnesses (programs to invoke a component under test), which are often simply thrown away. If there are errors concerning the integration of subsystems, the whole process of designing, coding and unit testing has to be repeated. Major flaws in the design of the whole system are not discovered until the very end, and there is no working system until the very last stage.

An alternative approach is *top-down development* [2,13]. It can be seen as a blend of the different stages mentioned above. The process proceeds from high-level components down. The high-level components are coded before lower levels are designed. *Program stubs* are designed to stand in for invoked but yet unwritten lower-level components. As necessary, test data are constructed. The system is assembled and tested. "Implementation proceeds by selecting lower-level components (formerly stubs) for design and coding and incorporation into the system" [2, p.198]. Some variations of the method seem to be necessary. "In practice some low-level components need to be designed, coded and tested at an early stage" [2, p.198]. On the other hand, it might be useful in some cases to first complete the design of the entire program before starting top-down coding and testing. And, some components are easier to test in isolation. There are a number of advantages of top-down development as described in [2]. Major flaws are detected at an early stage of the process. The reliability of software components (especially the high-level components) increases, since they are tested again and again. It is easier to locate a fault, since faults can be found in the single new component just added or in the interface with higher-level modules. It has been said that top-down development is well-suited to projects that are undertaken by a team of programmers. Studies show that coding takes up approximately 20% of developing time, while 50% of the development effort takes place after the code is written. Top-down development seems to be a way to change these proportions.

Another approach is to construct a *prototype*. Prototyping is a technique for requirements analysis. A prototype is a working version of a piece of software, constructed to identify the major characteristics of the system to be built. "The purpose is to aid the analysis and design stages of a project by enabling users to see very early what the system will do" [2, p.201]. The question arises if the prototype should simply be thrown away after serving this purpose, or if it should be tried to transform it into the final system, for instance while looking at the construction of the system as an optimization of the prototype (this approach is generally dangerous).

Yau/Tsai [15] list a number of useful guidelines for a good programming style:

- modularize the system
- strive for program readability
- avoid programming tricks
- restrict use of global data
- use data abstraction concepts
- minimize the number of paths through programs
- give preference to static data structures

Although a number of these guidelines might not be applicable under certain conditions (the last one, for instance, is not useful if the problem to be solved is very dynamic), these points are important for the support of not only testing and verification, but also for program maintenance. A decision has to be made concerning the program language in which the system should be implemented. Constraints for this choice include the availability of compilers, the compatibility with other subsystems, and the availability of modules written for other systems which can be reused to reduce development time.

According to [13, p.38], testing "is a series of controlled experiments that seek to provide empirical evidence that a program behaves properly (and provides the desired results for broad classes of anticipated inputs)." Verification, on the other hand, can be "a formal, mathematical proof that the program is in conformity with its specifications" [13, p.38]. The cost of this kind of verification is quite high. There are other manual or automated verification and validation techniques, including walkthroughs and inspections. A walkthrough is an organized but informal meeting at which a program is examined; the programmer presents his/her code and the documentation to a review team. Program inspection, on the other hand, is a formal review in which the review team checks the program against a prepared list of concerns.

There are several methods to perform a test [2]. With one method, a *selection* of input data values is devised, and the actual outcome is compared with the expected one. A better method would be to use all possible input values, and to check the outcome. Obviously this approach is very impracticable, because of the usually large number of possible values. The first two methods consider the program as a black box. It is better though to use knowledge about the internal structure of the program, considering the program as a *white box*. One suggestion is to use test data that causes every path to be executed in all possible combinations. Again, this process is too lengthy. A more practicable approach is "to devise test data that causes execution of every program path (though not all combinations of paths), at least once in the testing" [2, p.195]. One might also view of testing as making sure to test the actions that a program takes in special cases.

Dijkstra stated that "testing can only show the presence of bugs, never their absence". Consequently, it might be more appropriate to look at a test that reveals no bugs as an *unsuccessful* test!

6. Object-oriented development

6.1. Introduction

In section 1, some of the problems with a traditional software life cycle have been addressed. One most recent approach to overcome these (and other) problems is the use of an object-oriented paradigm. It is important to note that object-orientation is more than just another software development method or another programming style. The way of how systems are viewed is fundamentally different to other approaches. One way of explaining the distinction between the traditional and the object-oriented view is given in [7]. Traditionally, a project-based approach is used; the subject of discourse is the project, starting with a certain specification, and ending with a delivery of a program. With object-orientation, the subject of discourse is reusable components rather than individual projects.

Another way for a discrimination is to investigate what the basic focus of the methods is [1,2,5,6]. There are traditional methods focusing on the functional aspect of the system with minimal consideration given to data in earlier development stages. Other methods favor the data and derive functionality to transform data. "The object-oriented mindset allows a developer to see systems in terms of active components made up of data fused together with associated functionality" [1, p.3]. Process driven and data driven approaches place their emphasis on either processes or data. The object-oriented approach applies a world view based on active, interacting *entities*, called *objects*, which encapsulate both data and procedures. These objects are grouped into *classes*. An *inheritance* relation is added to the traditional dependencies between data elements.

The goals of object-orientated development are not new, and so are many of the concepts used within this framework. The intent is to "simplify the generation of large, complex software systems, and to encourage the production of software that is modular, easily understood, reusable, and adaptable to change" [2, p.122]. The evolution of the object-oriented paradigm started with a purely procedural approach, and was enhanced by an object-based approach. Both of the "older" approaches basically utilize functional decomposition to develop the architecture of a system. The object-oriented approach though gives emphasis to data by utilizing the relationships between objects.

6.2. Basic concepts

There are a number of concepts crucial to an understanding of the object-oriented approach to software development. Most of the concepts are *not* new in a sense that they have exclusively been developed for this paradigm. "It is the blending of inheritance with the other ... concepts in specific ways that characterizes object-oriented programming" [6, p.42]. The discussion will mainly be based on [1,2,6]. The concepts covered are *objects*, *classes*, *inheritance*, *polymorphism*, and *message passing*. Other concepts like *composition* and *generic typing* will not be discussed.

6.2.1. Objects

An object is a "thing" with an identity, with a state and a certain behavior. The behavior is defined by the services, or operations, it can perform. Some methods have to be defined to carry out these operations. Objects have a boundary. They offer their services to other objects, *clients* in this case. A client requests the services of another object by sending it a *message*. Each object can be thought of as a small virtual processor whose behavior is defined by how it responds to receiving a message. The objects are independent, active agents. Meaning and behavior are internal to the objects.

6.2.2. Classes

A class defines a set of possible objects. Its definition describes the form and behavior of all objects of that class. There is an "is-a" relationship between an objects and its class, an object is an *instance* of its class. Therefore, a class defines the structure and function of a potentially infinite set of individual objects. Ideally, a class is an implementation of an *abstract data type*. Implementation details and all data of a class are private to this class, enforcing the principle of information-hiding; the *boundary* of the abstract data type is established. Two kinds of methods can be found in the public interface of such a class. There are functions that return meaningful abstractions about the state of an instance, and there are transformation procedures used to move an instance from one valid state to another. Other objects rely only on the interface of a class, independent of its implementation.

6.2.3. Inheritance

"Inheritance is a relation between classes that allows for the definition and implementation of one class to be based on that of other existing classes" [6, p.43]. Once the base class is understood, there is only the need to understand how a derived class differs from the more general base class, since derived classes are described only in terms of these differences. Inheritance supports reuse across systems and it directly facilitates extensibility within a given system. It minimizes the amount of new code needed when adding additional features. Given a derived class Y and a base class X, Y has a derived and an incremental part. The derived part is inherited from X, whereas the incremental part is the new code, especially written for Y. Class Y now has all the features of X. Y is an X, but is it more than an X [6].

6.2.4. Polymorphism

The term polymorphism in general means the ability to take more than one form. A polymorphic reference in the context of object-oriented languages is one that can refer to instances of more than one class. The idea of polymorphism is coupled with the nature of inheritance. If "Y inherits from X, Y is an X, and therefore anywhere that an instance of X is expected, an instance of Y is allowed" [6, p.45]. There are several forms of polymorphism, the one used above is referred to as *inclusion polymorphism*. Other forms are *parametric polymorphism* (procedures work uniformly for a range of types), *overloading* (a single operator or function name may be applied to multiple types), and *coercion* (values of different types are used in the same expression) [1].

6.2.5. Message passing

As stated earlier, an object requests the services of another object by sending it a *message*. The service corresponds to an internal method of the called object. Message passing is different to simple function calls, which are resolved at link time. A message is a request for action, not a function call. It might happen that the code associated with a call is not known until the moment of the call at runtime, and it may be the case that one of several different responses are possible. "The process of determining which of the possible responses is appropriate then finally invoking the appropriate function is called *dynamic binding*" [1, p.10]. Dynamic binding is associated with polymorphism and inheritance. A procedure call associated with a polymorphic reference may depend on the dynamic type of that reference, and dynamic binding is only required in the presence of inheritance.

6.3. Object-oriented software life cycle

Several authors identify the three traditional activities of analysis, design, and implementation within their description of an object-oriented software life cycle [5,6]. The main difference to traditional approaches is that the distinct boundaries between the phases are eliminated. This is based on the fact that the items of interest in each phase are the same. "Beginning in the requirements phase, objects are identified. By developing specifications of the entities found in the problem domain a clear and well-organized statement of the problem is actually built into the application. These objects form a high-level layer of definitions that are written in the terminology of the problem domain. During the refinement of the definitions and the implementation of the application entries, other entities, or classes, are identified. ... In one phase the analyst identifies problem domain objects while in the next phase, the designer specifies additional objects necessary for a specific computer-based solution. The design process is repeated for these implementation-level objects" [6, p.48].

The object-oriented development process is iterative. Henderson-Sellers/Edwards [5] therefore replace the waterfall model by the *fountain model*. The fountain model represents both iteration and overlap. The starting point is the requirements analysis and specification, following stages include system design, program design, coding, unit testing, system testing and program use. The life cycle "grows upward to a pinnacle of software use, falling only in terms of necessary maintenance. This effectively reverts the stage of the cycle to a lower level" [5, p. 151]. Or as stated in [6, p.41]: "Development reaches a high level only to fall back to a previous level to begin the climb once again".

There are two separate components in object-oriented design, class design and application design. Each identified entity leads to a class description. Once these descriptions have been developed, the application can be designed while connecting instances of the classes. The pattern of interaction between these instances provides the structure of the application. The development of an object-oriented application is a blend of class description and application configuration. "... since an object-oriented program will be developed essentially as an interacting system of classes (...), the stages of the life cycle model can be applied more accurately to the development cycle of each individual class rather than the system as a whole" [5, p.152]. A special life cycle for a tightly related group of classes, or cluster, has been developed. The *cluster model* [5,7] has three phases: 1) specification, 2) design and implementation, and 3) validation and generalization. The cluster model is significant as a branch of the systems specification in the software life cycle.

Besides the fact that special object-oriented software life cycles are developed, other authors argue that the object-oriented paradigm can be used with traditional life cycles, serving as a consistent underlying theme, and preserving a higher conceptual integrity throughout the development process.

6.4. Some advantages of object-orientation

There are a number of advantages one can gain while applying object-oriented thinking and methods [1,6]. The special paradigm provides natural support for decomposing a system into modules, classes in this case. Information hiding is supported through the separation of the class interface and the class implementation. Weak coupling and strong cohesion are other important results of object-oriented design. Easily extendable designs are produced, and reusability is strongly supported. The approach helps to control complexity, and it helps to preserve conceptual integrity in all aspects of software development.

7. The HICLASS project

The HICLASS system was developed by a one person team. Hence, management and communication problems could not arise. As a result of the theoretic development of the system's functions, some data structures as well as algorithms have been developed beforehand. Basically, the traditional software life cycle has been applied with modifications such as adding iteration and repeating some steps. In system design, a depth-first functional decomposition has been applied, with the modification of already having some data structures defined in advance. A top-down development as described in section 5 was used for the next stages of the life cycle. Testing was done using a white box approach, taking special care of special cases. Due to the dynamic nature of the problem, almost all data structures were developed in a dynamic way, which resulted among other things in numerous checkpoints to insure a safe execution of the program. Lots of thought was given to a practicable and easy-to-use user interface, achieved through the use of pull-down menus and spreadsheets. Global data has been defined, restricted though to variables needed by many modules of the system. The programming was done in TURBO PASCAL 6.0 (approximately 16.000 lines of source code). The decision to choose PASCAL was influenced by the fact that a number of utility toolboxes were available, and that the programmer was most experienced in this language.

The HICLASS system was divided into two major parts: HIEDIT, the table editor program, and HICLASS, the application program performing hierarchical classification based on tables chained together in a hierarchy. For a brief discussion of the software design process performed the focus will be on HIEDIT.

HIEDIT was developed in a modular fashion. Four different screens have been identified, each of those performing special actions. The screens have been designed one after another, following a depth-first functional decomposition. Low-level modules like a library of basic utility functions and pull-down menu functions have been identified. These utility modules were designed, coded and tested at an early stage, using especially designed test harnesses and test data. Then, a top-down development strategy has been applied. The high-level components were coded before lower levels were designed. Program stubs were used to stand for invoked but yet unwritten lower-level components. Test data did not have to be constructed, since the flow of data in the system is very linear, and the output of one screen is the input for the next. Hence, if one screen was coded, the data produced within this screen could be used as test data for the development of the next screen.

In figure 7.1., the root of the tree stands for the overall task of the system. The next level shows the two program which had to be devoloped. At the third level, the four screens of HIEDIT are shown. In the implementation, the functions of each of those screens are grouped in a separate PASCAL unit. The next level shows the decomposition of the FILES screen. Each of the functions identified corresponds to a procedure/function implemented for this screen. The tree structure of figure 7.1. is not complete, it outlines the basic design of the system.

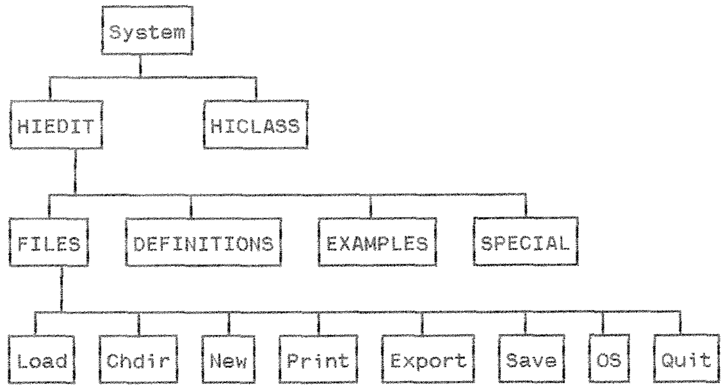


Figure 7.1. Parts of the functional decomposition of the HICLASS project

The system is modular, it is built from well-defined modules. Appendix C includes almost all modules designed for the project. Most modules are smaller than one page of text. The modules are arranged in levels of abstraction. Information hiding is realized to a high degree within the boundaries of units as well as single procedures and functions. Content coupling does not appear, whereas common coupling does because some variables used by many modules are defined globally. Control coupling appears for some of the utility modules. Some modules use stamp coupling, whereas the majority of modules is data coupled. Coincidental cohesion does not appear. There is some logical, temporal, procedural, and communicational cohesion. Most of the modules though are either sequentially or functionally cohesive. The majority of modules has medium fan-in and fan-out. Utility modules though have a very high fan-in, and a low fan-out. Only some of the higher-level control modules have a high fan-out. The scope of effect of the modules is limited to the scope of control.

As the software life cycle needed to be chosen, some thought was given to a possible implementation using the object-oriented paradigm. And in fact, this would have been a fruitful idea only considering the amount of code written for the user interface with all its menus, message boxes, etc., and the implementation of a best-first search in HICLASS. But practical constraints prevented the use of the object-oriented methodology. Several toolboxes were already in place, and the programmer had little practical experience with an object-oriented language such as C++, or the object-oriented features of TURBO PASCAL 6.0.

References

[1] Ames. A comprehensive description and critical analysis of object-oriented software development. Final report for the degree of Master of Systems Analysis. Miami University, 1991.

[2] Bell/Morrey/Pugh. Software Engineering - a programming approach. Englewood Cliffs NJ: Prentice/Hall, 1987.

[3] Caine/Gordon 1975. PDL - a tool for software design. In: Freeman/Wasserman, eds. Tutorial on Software Design Techniques. New York: The Institute of Electrical and Electronics Engineers, 1980.

[4] Fournier. Practical guide to structured system development and maintenance. Englewood Cliffs NJ: Yourdon Press, 1991.

[5] Henderson-Sellers/Edwards. The object-oriented systems life cycle. In: Communications of the ACM, Vol.33, No. 9, September 1990, pp.142-159.

[6] Korson/McGregor. Understanding object-oriented unifying paradigm. In: Communications of the ACM, Vol. 33, No.9, September 1990, pp.40-60.

[7] Mandrioli/Meyer. Advances in object oriented software engineering. New York: Prentice/Hall, 1992.

[8] Parnas/Clements. A rational design process: how and why to fake it. In: IEEE Transactions on Software Engineering, Vol. SE-12, No.2, February 1986, pp.251-257.

[9] Pfleeger. Software Engineering. New York: Mac Millan Publishing Company, 1987.

[10] Ross/Goodenough/Irvine 1975. Software Engineering: Process, Principles, and Goals. In: Freeman/Wasserman, eds. Tutorial on Software Design Techniques. New York: The Institute of Electrical and Electronics Engineers, 1980.

[11] Stay 1976. HIPO and Integrated Program Design. In: Freeman/Wasserman, eds. Tutorial on Software Design Techniques. New York: The Institute of Electrical and Electronics Engineers, 1980.

[12] Stevens/Myers/Constantine 1974. Structured Design. In: Freeman/Wasserman, eds. Tutorial on Software Design Techniques. New York: The Institute of Electrical and Electronics Engineers, 1980.

[13] Wasserman 1980. Information system methodology. In: Freeman/Wasserman, eds. Tutorial on Software Design Techniques. New York: The Institute of Electrical and Electronics Engineers, 1980.

[14] Wirth 1971. Program development by stepwise refinement. In: Freeman/Wasserman, eds. Tutorial on Software Design Techniques. New York: The Institute of Electrical and Electronics Engineers, 1980.

[15] Yau/Tsai. A survey of software design techniques. In: IEEE Transactions on Software Engineering, Vol. SE-12, No.6, June 1986, pp.713-721.

[16] Ziegler. Programming system methodologies. Englewood Cliffs NJ: Prentice/Hall, 1983.

Appendix C

Modules

NOTE:

The HICLASS system was decomposed into two main programs (HIEDIT and HICLASS). Both programs were further decomposed into PASCAL units, each consisting of a number of modules. Additional utility units provide low-level functions used by the higher-level modules. This appendix lists almost all modules designed for the system. For the units, both the interface and the implementation parts are shown in the shape of constant, type, variable and module definitions. Nested module definitions are indicated as such. If a module definition is declared in the interface definition of a unit, it is repeated only if there are nested modules within the particular module. For the utility units, only the interface definitions are included.

Table of contents

unit hiall 3

program hiedit 7

unit hiedfile 8

unit hieddef 10

unit hiedex 12

unit hiedspec 14

program hiclass 16

unit hiclfile 17

unit hiclload 18

unit hiclask 19

unit hiclutil 21

unit hiclread 24

unit himenu 25

unit hispread 27

unit hieditor 29

unit LSutil 30

unit LShelpk 32

```
unit hiall;

{ ************************************************************************** }
{ *                                                                        * }
{ * This unit is part of HIEDIT,HICLASS (c) 1992                           * }
{ * Program author : Jens Wazel                                            * }
{ * Programming environment : Turbo Pascal 6.0                             * }
{ *                                                                        * }
{ ************************************************************************** }
{ * Services : Declarations for both systems                               * }
{ *            Messages and error messages                                 * }
{ *            Initialize help system                                      * }
{ ************************************************************************** }
{ * Actions  : Get calling path                                            * }
{ ************************************************************************** }


interface

uses crt,lshelpk,lsutil;

const

      CMrahm    = Black  + LightGray*16;  {pull down menu colors}
      CMhead    = Black  + LightGray*16;
      CMtext    = Black  + LightGray*16;
      CMhigh    = Black  + Green*16;
      CMhot1    = Yellow + LightGray*16;
      CMhot2    = Yellow + Green*16;


      DIrahm    = Black  + LightGray*16;  {dialog boxes colors}
      DIhead    = Black  + LightGray*16;
      DItext    = Black  + LightGray*16;
      DIhigh    = Black  + Green*16;

      EXhead    = Black  + LightGray*16;  {HIEDIT example menu colors}
      EXheadc   = Yellow + LightGray*16;
      EXnum     = Black  + LightGray*16;
      EXtext    = Lightgray + Blue*16;
      Exhigh    = Black  + Magenta*16;
      Exmove    = Black  + LightGray*16;

      Valrahm    = Black  + LightGray*16;  {HIEDIT def menu colors}
      Valhead    = Black  + LightGray*16;
      Valtext    = Black  + LightGray*16;
      Valhigh    = Black  + Green*16;
      Valnum     = Black  + Lightgray*16;
      Valheadc   = Black  + Cyan*16;

      Qshad    = Magenta;                  {Colors for question & result}
      Qrahm    = Black  + LightGray*16;
      Qhead    = Black  + LightGray*16;
      Qtextn   = Black  + LightGray*16;
      Qtextu   = Black  + Green*16;
      Qtexts   = Yellow + LightGray*16;
      Qmove    = Black  + Magenta*16;
      Qnum     = Black  + Magenta*16;


      Surahm     = Yellow + Black*16;      {Colors for History & Conclude}
      Suhead     = Yellow + Black*16;
      Sutext     = Yellow + Black*16;
      Sunum      = Yellow + Black*16;
      Sumove     = Yellow + Black*16;

      errcol    = Yellow + Red*16;        {error color}
      hintcol   = Yellow + Green*16;      {hint color}
```

```
      framecol  = Black  + LightGray*16;  {big frame color}
      sframecol = White  + Blue*16;       {single frame color}
      cframecol = Red    + LightGray*16;  {control messages top}

      backcol   = Blue;                   {background of normal text}
      textcol   = Yellow;                 {normal text}
      f1backcol = LightGray;              {background of F1 message}
      f1forecol = Black;                  {foreground of F1 message}
      commcol   = Yellow + Black*16;      {short screen explanation}

      kb_suff  = '.HIT';                  {file suffix for table files}
      rpt_suff = '.HIR';                  {file suffix for report files}
      file_tag = 'HICLASS (c)1992';       {file tag for table files}

      {Editor settings}
      eb=74;                       {maximal number of text columns}
      eh=20;                       {maximal number of text rows}
      ep=20;                       {text rows per page}
      max_hervor=20;               {maximal number of accentuated strings}

      {HIEDIT settings}
      abs_max_attr = 13;           {maximal number of attributes per table}
      abs_max_val  = 26;           {maximal number of values per attribute}
      abs_max_ex   = 255;          {maximal number of examples}

type  attr_pointer        = ^main_attr;
      val_pointer         = ^main_val;
      text_pointer        = ^main_text;
      ex_pointer          = ^main_ex;
      exlearn_pointer     = ^main_exlearn;
      table_pointer       = ^main_table;
      history_pointer     = ^main_history;
      global_attr_pointer = ^main_global;
      control_pointer     = ^main_control;
      restore_pointer     = ^main_restore;
      num_pointer         = ^main_num;

      main_attr    = record                   {attribute}
                           name:string[9];
                           text:text_pointer;
                           askfirst:byte;     {0=no 1=askfirst 2=no askfirst}
                           max_val:byte;
                           values:val_pointer;
                           min_cert:byte;
                           next:attr_pointer
                     end;

      main_val     = record                    {value}
                           name:string[9];
                           text:string[74];
                           textres:text_pointer;  {only for values of RESULT}
                           cert:byte;
                           next:val_pointer;
                     end;

      main_ex      = record                     {example}
                           v:array[1..abs_max_attr+1] of byte;
                           next:ex_pointer;
                     end;

      main_exlearn = record                      {learned example}
                           {if numeric then left interval limit or unique}
                           v:array[1..abs_max_attr+2] of byte;

                           {if numeric right interval limit or 0; else 0}
                           v1:array[1..abs_max_attr-1] of byte;
                           next:exlearn_pointer;
                     end;
```

```
main_text   = record                        {text format}
                  anzherv:byte;
                  text:array[1..eh] of string[eb];
                  herv:array[1..max_hervor,1..5] of byte;
              end;

numeric_table = record     {supporting table for numeric value handling}
                  max:byte;
                  v:array[1..abs_max_val] of byte;
                end;

main_num      = array[1..abs_max_attr-1] of numeric_table;

main_table   = record                        {HICLASS table format}
                  name:string[9];               {table name}
                  max_attr:integer;             {number of attributes}
                  max_ex  :integer;             {number of examples}
                  first_attr:attr_pointer;      {start of attributes}
                  first_ex:ex_pointer;          {start of examples}
                  unknown_allowed:boolean;      {unknown allowed}
                  dont_applic_allowed:boolean;  {don't applicable allowed}
                  predefined:boolean;           {predefined order?}
                  favored_strategy:byte;        {favored local strategy}
                  treshhold:byte;               {treshhold for uncertainty}
                  interval:byte;                {interval numeric values}
                  shortcut:boolean;             {shortcut allowed?}
                  strategy_used:byte;           {strategy used}
                  prior_certainty:byte;         {prior cert. for table}
                  reader:pointer;               {reader for table}
                  no_ques:byte;                 {number of questions}
                  no_unknown:byte;              {number of answers UNKNOWN}
                  num:num_pointer;              {list of numeric values}
                  numeric:boolean;              {results numeric}
               end;

main_history   = record        {session report}
                   table_name:string[9];
                   table_question:string[9];
                   answer:string[9];
                   certainty:byte;
                   next:history_pointer;
                 end;

main_global    = record        {list of global attributes and values}
                   attr:string[9];
                   value:string[9];
                   numval:byte;
                   cer:byte;
                   next:global_attr_pointer
                 end;

main_restore   = record        {list of answers for restore}
                   table_name:string[9];
                   table_question:string[9];
                   answer:string[9];
                   next:restore_pointer;
                 end;


main_control   = record        {global control instance}
                   table_name:string[9];    {name of table}
                   siblings:boolean;        {siblings in hierarchy active}
                   call_attr:boolean;       {called from attribute}
                   call_res :boolean;       {called from result}
                   call_table:string[9];    {name of calling table}
                   solved:boolean;          {table solved completely}
                   cert:byte;               {prior certainty for table}
                   next:control_pointer;
                 end;
```

```
var
{Definitions for HICLASS,HIEDIT}

      help_avail:boolean;                {help system available}
      cpath:string;                      {calling path}
      apath:string;                      {current path of current drive}
      kb_file:text;                      {file holding a table}
      kb_filename:string;                {name of table}
      saved:boolean;                     {file succesfully saved}
      overwrite:boolean;                 {overwrite existing file?}
      save_request:boolean;              {save current table or report}
      load_kb:boolean;                   {load new table or report}
      quit_total:boolean;                {quit program}

{Definitions for HIEDIT}

      deletee:boolean;                   {delete items}
      first_attr:attr_pointer;           {start of attribute definitions}
      first_ex:ex_pointer;               {start of examples}
      first_ex_dist:exlearn_pointer;     {start of examples from
                                          distinction oriented learning}
      first_ex_norm:exlearn_pointer;     {start of examples from
                                          non-distinction oriented learning}
      max_attr:integer;                  {number of attributes defined}
      max_ex  :integer;                  {number of examples defined}
      max_ex_dist:integer;               {number of rows from
                                          distinction oriented learning}
      max_ex_norm:integer;               {number of rows from
                                           non-distinction oriented learning}
      learn_dist:boolean;                {distinction orientied learning?}
      learn_norm:boolean;                {non-distinction oriented learning?}
      unknown_allowed:boolean;           {answer unknown allowed}
      dont_applic_allowed:boolean;       {answer not applicable allowed}
      predefined:boolean;                {predefined order}
      favored_strategy:byte;             {favored local strategy}
                                         {0 = NONE
                                          1 = MATCH
                                          2 = LEFT TO RIGHT
                                          3 = HEURISTIC}
      treshhold:byte;                    {threshold for uncertainty}
      interval:byte;                     {interval size for numeric values}
                                         {[0..50]%  0=unique values}
      shortcut:boolean;                  {shortcut in control strategy}


{Definitions for HICLASS}

     first_history:history_pointer;         {start of session report}
     max_history:integer;                   {number of history entries}
     interrupt_session:boolean;             {interrupt current session}
     first_global_attr:global_attr_pointer; {list of global attributes}
     first_control:control_pointer;         {control list}
     advice_at_all:boolean;                 {was advice possible}
     rpt_file:text;                         {report file}
     rpt_filename:string;                   {name of report file}
     restore:boolean;                       {answers come from report file}
     first_restore:restore_pointer;         {position in restore}

{ *************************************************************************** }

function message(fall:byte):char;
  {-Reads answers to messages}
  {-Returns <CR> or:  'N','n' (fall=0);  'Y','y' (fall=1)}

procedure err(error_nr:byte);
  {-Displays error message and reads user answer if applicable}

procedure init_help;
  {-Initialize help system}
```

```
{$M 16000,150000,200000}
program hiedit;

{ ************************************************************************ }
{ *                                                                      * }
{ * This program is the main program for HIEDIT   (c) 1992               * }
{ * Program author : Jens Wazel                                          * }
{ * Programming environment : Turbo Pascal 6.0                           * }
{ *                                                                      * }
{ ************************************************************************ }

uses dos,crt,lsutil,hiall,himenu,hiintr,hiedfile,hieddef,hiedex,hiedspec;

var screen_no:byte;

procedure init;
{initializes global system variables}


(*--------------------------------main-------------------------------------*)
begin
   init;

   screen_no:=1;
   repeat
    case screen_no of
      1:screen_no:=file_screen;
      2:screen_no:=def_screen;
      3:screen_no:=ex_screen;
      4:begin spec_screen;screen_no:=3; end;
    end
   until screen_no=0;

  RestoreConfig;
  clrscr
end.
```

```
unit hiedfile;

{ ************************************************************************** }
{ *                                                                        * }
{ * This unit is part of HIEDIT (c) 1992                                   * }
{ * Program author : Jens Wazel                                            * }
{ * Programming environment : Turbo Pascal 6.0                             * }
{ *                                                                        * }
{ ************************************************************************** }
{ * Services : Load a table                                                * }
{ *            Change the current directory                                * }
{ *            Start a new table                                           * }
{ *            Save a table                                                * }
{ *            Leave temporarily for DOS                                   * }
{ *            Quit the program                                            * }
{ ************************************************************************** }
{ * Actions  :                                                             * }
{ ************************************************************************** }

interface

uses crt,dos,lsutil,hiall,himenu,hispread,hieditor,hieddef,hiedex,hiedspec;

var file_menu_active:boolean;

function file_screen:byte;
 {-returns 0 if end of program
           1 if definition attribute screen is next}

{ ************************************************************************** }

implementation

var ok,exit_left,exit_right:boolean;
    file_screen_pointer:pointer;


function get_filename(mode:byte):byte;
{-Ask user for filename, used in menu_new and menu_save}
{-Returns 0 if filename
          1 if out of memory
          2 if no filename}

function ws(stri:string):boolean;
{-write a string to the file}

function save_attr:boolean;
{-save attributes and values}

function save_ex:boolean;
{-save examples}

function save_exlearn(which:byte):boolean;
{-save learned examples}

procedure save_table;
{-save the current table}

procedure init;
{-initialize file screen}

procedure init_menu;
{-initilize menu on file screen}

procedure Menu_F1;
{-provide context-sensitive help}
```

```
procedure menu_save;
{-initiate saving the current table}

procedure dispose_all;
{-dispose all active menus and spreadsheets in system}
{-re-initialize global variables}

procedure menu_dir;
{-initiate loading a table from disk}

procedure menu_chdir;
{-change the current path}

procedure menu_new;
{-start a new table}

procedure menu_print;
{-print the contents of a table}

procedure menu_export;
{-export the contents of a table}

procedure menu_os;
{-temporary exit to DOS}

procedure menu_quit;
{-initiate quitting the program}
```

```
unit hieddef;

{ ************************************************************************ }
{                                                                        * }
{ * This unit is part of HIEDIT (c) 1992                                 * }
{ * Program author : Jens Wazel                                          * }
{ * Programming environment : Turbo Pascal 6.0                           * }
{ *                                                                      * }
{ ************************************************************************ }
{ * Services : Edit attributes and values                                * }
{ *            (Add, Change, Move, Text, Delete)                         * }
{ ************************************************************************ }
{ * Actions  :                                                           * }
{ ************************************************************************ }

interface

uses crt,dos,lsutil,hiall,himenu,hispread,hieditor,hiedex;

var  def_menu_active :boolean;
     def_screen_pointer:pointer;

function def_screen:byte;
 {-returns  1 if back to file screen
            3 if definition value screen is next}

procedure dispose_table;
 {-dispose all attributes and values}

{ ************************************************************************ }

implementation

var ok,exit_left,exit_right,action:boolean;
    mem_preserve:LongInt;


procedure calc_mem;
{-calculate memory to be preserved for screen}

procedure define_head;
{-define headlines of columns for spreadsheet menu}

procedure init_menu;
{-initialize spreadsheet}

function rb(mode:byte;var by:integer;var stri:string):boolean;
{-load a string and convert if necessary to value}
{-returns string (mode=0) or value (mode=1)}

function load_attr:boolean;
{-load all attributes and values}

function load_ex:boolean;
{-load examples}

function load_exlearn(which:byte):boolean;
{-load learned examples}

procedure load_table;
{-load a table}

function get_attr_name(var st:string;var akf:byte):byte;
{-get attribute name}
{-returns 0 if no attribute was defined  1 otherwise}

procedure init_attr;
{-initialize attribute}
```

```
procedure add_attr;
{-add an attribute}

procedure change_attr;
{-change the name of an attribute}

procedure move_attr;
{-move an attribute to another location}

procedure del_attr;
{-delete an attribute}

procedure text_attr;
{-text for attribute}

procedure dispose_table;
{-dispose all attributes and values}

procedure init;
{-initialize definition screen}

function get_val_name(which_attr:byte;var st:string):byte;
{-get the name for a value}
{-returns 0 if no new value was defined  1 otherwise}

procedure move_val;
{-move a value to another location}

procedure add_val;
{-add a value for an attribute}
   function check_val:boolean;
   {-check if new value is ok}

procedure change_val;
{-change the name for a value}

procedure del_val;
{-delete a value}

procedure text_val;
{-text for value}
```

```
unit hiedex;

{ ********************************************************************************** }
{ *                                                                                * }
{ * This unit is part of HIEDIT (c) 1992                                           * }
{ * Program author : Jens Wazel                                                    * }
{ * Programming environment : Turbo Pascal 6.0                                     * }
{ *                                                                                * }
{ ********************************************************************************** }
{ * Services : Edit examples                                                       * }
{ *            (Add, Change, Replicate, Delete)                                    * }
{ ********************************************************************************** }
{ * Actions  :                                                                     * }
{ ********************************************************************************** }

interface

uses crt,dos,lsutil,hiall,himenu,hispread;

var ex_menu_active :boolean;
    ex_screen_pointer:pointer;


function ex_screen:byte;
{-returns   2 if back to definition screen
            4 if special screen is next}

function attr_used(which_attr:byte):boolean;
{-Test if attribute is used in examples}
{-returns true if used}

function val_used(which_attr,which_val:byte):boolean;
{-Test if value is used in examples}
{-returns true if used}

function num_val_used(which_attr:byte):boolean;
{-Test if numerical value is used in examples}
{-returns true if used}

procedure del_attr_used(which_attr:byte);
{-Delete an attribute in all examples}

procedure del_val_used(which_attr,which_val:byte);
{-Delete a value in all examples}

procedure reset_val_used(which_attr:byte);
{-Reset all values for an attribute in all examples}

procedure add_attr_ex(which_attr:byte);
{-Add an attribute in all examples}

procedure add_val_ex(which_attr,which_val:byte);
{-Add a value in all examples}

procedure move_attr_ex(source,dest:byte);
{-Move an attribute in all examples}

procedure move_val_ex(which_attr,source,dest:byte);
{-Move a value in all examples}

procedure dispose_ex;
{-Dispose all examples}

procedure update_ex;
{-update example menu}
```

```
{ ************************************************************************** }

implementation

var ok,exit_left,exit_right,action:boolean;
    mem_preserve: LongInt;

procedure calc_mem;
{-calculate memory to be preserved for screen}

procedure define_head;
{-define headlines of columns for spreadsheet menu}

procedure init_menu(which:byte;number_of_attributes:byte);
{-initialize spreadsheet for screen}

procedure init;
{-init example screen}

function get_value(name:string;ex_va:byte):byte;
{-get a value (numeric input)}

function select_value(which_attr:integer;var ex_value:byte):boolean;
{-get a value for an attribute for an example}

function get_weight(which_attr:byte;var ex_value:byte):boolean;
{-get weight for an example}

procedure init_ex;
{-initialize examples}

procedure add_ex;
{-add an example}

procedure change_ex(which:byte);
{-change a value for an attribute of an example}

procedure replicate_ex;
{-replicate an example}

procedure del_ex(which:byte);
{-delete an example}

procedure check_examples;
{-check examples for consistency}
```

unit hiedspec;

```
{ ************************************************************************** }
{ *                                                                        * }
{ * This unit is part of HIEDIT (c) 1992                                   * }
{ * Program author : Jens Wazel                                            * }
{ * Programming environment : Turbo Pascal 6.0                             * }
{ *                                                                        * }
{ ************************************************************************** }
{ * Services : Change a number of settings for the table                   * }
{ *            Provide learning features                                   * }
{ ************************************************************************** }
{ * Actions  :                                                             * }
{ ************************************************************************** }

interface

uses crt,dos,lsutil,hiall,himenu,hispread;

var spec_menu_active :boolean;
    spec_menu_pos:integer;
    spec_screen_pointer:pointer;

procedure spec_screen;

procedure dispose_ex_dist;
 {-Dispose all distinction oriented examples}

{ ************************************************************************** }

implementation

type
 group_pointer = ^main_group;

 main_group = record
                 v:array[1..abs_max_attr+1] of byte; {group contents}
                 e:integer;                          {# of examples for group}
                 next:group_pointer;
                end;

var num: main_num;
    mem_preserve: LongInt;


procedure learn_distinction-oriented

   procedure check_for_numeric;
   {-detect numeric values in examples and store unique ones}

   function make_inits:boolean;
   {-initialize learning process}

   function store_group:boolean;
   {-store a new group}

   function check_ex_and_build_group:boolean;
   {-check generated group with examples and build group if ok}

   function make_group(offset:byte):boolean;
   {-generate groups in a recursive fashion and check with examples}

   procedure build_new_examples;
   {-build a non-redundant set of distinction oriented examples}

      procedure look_for_max;
      {-check which of the remaining groups substitues the most examples}
```

```
      function check_in:boolean;
      {-check if succeding group is already included in new example set}

      function append_ex:boolean;
      {-append a new example to the distinction oriented example set}

   procedure dispose_groups;
   {-dispose all remaining groups in the group set}

   procedure build_interval;
   {-build intervals of numeric values}

procedure show_ex_dist;
{show distionction-oriented examples}

   procedure define_head;
   {-define attribute names for spreadsheet}

   procedure init_spread;
   {-initialize spreadsheet}

procedure init;
{-initialize special screen}

procedure get_treshhold;
{-get threshold for uncertainty}

procedure get_interval;
{-get interval range for numeric values}

procedure get_strategy;
{-get local forward strategy to be used}

procedure init_spec_menu;
{-initialize menu on special screen}

procedure learn_dist_main;
{-initiate distinction-oriented learning}
```

```
{$M 40000,0,655360}
program hiclass;

{ ************************************************************************** }
{ *                                                                        * }
{ * This program is the main program for HICLASS   (c) 1992                * }
{ * Program author : Jens Wazel                                            * }
{ * Programming environment : Turbo Pascal 6.0                             * }
{ *                                                                        * }
{ ************************************************************************** }

uses crt,dos,lsutil,hiall,hiclfile,hiclask,hiclload,hiclutil;

var qc:q_a;

begin
 kb_filename:='';
 rpt_filename:='';
 saved:=true;

 while file_screen do begin
    first_global_attr:=nil;
    first_control:=nil;
    advice_at_all:=false;
    first_history:=nil;
    max_history:=0;
    meta('ROOT',kb_filename,10,false,false,qc);
    saved:=false;
    if (not (qc.ans in [0,255]))and(not advice_at_all) then err(25);
 end;
 RestoreConfig;
 clrscr
end.
```

```
unit hiclfile;

{ *************************************************************************** }
{ *                                                                         * }
{ * This unit is part of HICLASS (c) 1992                                   * }
{ * Program author : Jens Wazel                                             * }
{ * Programming environment : Turbo Pascal 6.0                              * }
{ *                                                                         * }
{ *************************************************************************** }
{ * Services : Load a main table                                            * }
{ *            Change the current directory                                 * }
{ *            Restore an interrupted session                               * }
{ *            Save a report file                                           * }
{ *            Quit the program                                             * }
{ *************************************************************************** }
{ * Actions  :                                                              * }
{ *************************************************************************** }

interface

uses crt,dos,lsutil,hiall,himenu,hieditor,hiclload;


function file_screen:boolean;
 {-returns false if quit program, 1 otherwise}

{ *************************************************************************** }

implementation

var ok,exit,quit:boolean;
    file_screen_pointer:pointer;


procedure init_menu;
{-initialize menu}

procedure menu_save;
{-initiate saving a report file}

procedure menu_dir;
{-initiate loading a root table}

procedure menu_restore;
{-initiate restoring a former session}

procedure menu_chdir;
{-change the current path}

procedure menu_quit;
{-initiate quitting the program}
```

```
unit hiclload;

{ ************************************************************************** }
{                                                                          * }
{ * This unit is part of HICLASS (c) 1992                                  * }
{ * Program author : Jens Wazel                                            * }
{ * Programming environment : Turbo Pascal 6.0                             * }
{ *                                                                        * }
{ ************************************************************************** }
{ * Services : Load a table                                                * }
{ *            Save a session report file                                  * }
{ *            Load a session report file                                  * }
{ ************************************************************************** }
{ * Actions  :                                                             * }
{ ************************************************************************** }

interface

uses crt,dos,lsutil,hiall,hieditor;

function load_table(kb_name:string;var table:table_pointer):boolean;
   {-Load a table and store in table record}
   {-Return true if succesful}

procedure save_report;
   {-Save a report file}

function load_report:boolean;
   {-Load a report file and store in history}
   {-Return true if succesful}

implementation

{ ************************************************************************** }

var mem_preserve:LongInt;

function load_table;

   function rb(mode:byte;var by:integer;var stri:string):boolean;
   {-load a string and convert if necessary to value}
   {-returns string (mode=0) or value (mode=1)}

   function load_attr:boolean;
   {-load all attributes and values}

   function load_ex:boolean;
   {-load examples}

procedure save_report;

   procedure prepare;
   {-prepare a string to be stored in report file}

function load_report;

   function rb(var stri:string):boolean;
   {-load a string}

   function load_ent:boolean;
   {-load entries of restore file}
```

```
unit hiclask;

{ ************************************************************************ }
{                                                                        * }
{ * This unit is part of HICLASS (c) 1992                                * }
{ * Program author : Jens Wazel                                          * }
{ * Programming environment : Turbo Pascal 6.0                           * }
{ *                                                                      * }
{ ************************************************************************ }
{ * Services : Perform local control strategies                          * }
{ ************************************************************************ }
{ * Actions  :                                                           * }
{ ************************************************************************ }

interface

uses crt,dos,lsutil,hiall,hiclread,hiclload,hiclutil;

procedure meta(call_name,tname:string;cert:byte;call:boolean;sib:boolean;
               var qb:q_a);
   {-load new table with tname}
   {-apply metarules and user's choice to decide about control strategy}
   {-provide results of table}
   {-with prior certainty cert}
   {-if call then called from result else called from attribute}
   {-name of calling table: call_name}
   {-if there are active siblings in the hierarchy then sib=true}
   {ans = 0     if user wants to quit
          1..26 number of answers
          98    Unknown (no answer)
          99    Not applicable
          255   if memory problem
            numeric = true if answers are numeric
            val = strings for answer values
            num = numeric answers
            cer = certainty for answer}


implementation

{ ************************************************************************ }

type string9 = string[9];


procedure ask_user(table:table_pointer;question:byte;var qa:q_a);
{-ask a question from table and provide answer values}
{-or invoke other table to answer}
{-or take answer from global list}

   procedure load_texts;
   {-load question and answer texts}

   procedure user_choice;
   {-give selection to user and process answer}

   procedure update_global;
   {-add answer to global list}

   procedure add_history;
   {-add to history}

   procedure process_called_table;
   {-process input from table called}

   procedure check_restore;
   {-take answer from restore list}
```

```
procedure left_right(table:table_pointer;var qa:q_a);
{-performs local strategy LEFT-TO-RIGHT}
{ans = 0     if user wants to quit
       1..26 number of answers
       98    Unknown (no answer)
       99    Not applicable
       255   if memory problem
         numeric = true if answers are numeric
         val = strings for answer values
         num = numeric answers
         cer = certainty for answer}

procedure match(table:table_pointer;var qa:q_a);
{-performs local strategy MATCH}
{ans = 0     if user wants to quit
       1..26 number of answers
       98    Unknown (no answer)
       99    Not applicable
       255   if memory problem
         numeric = true if answers are numeric
         val = strings for answer values
         num = numeric answers
         cer = certainty for answer}

procedure heuristic(table:table_pointer;var qa:q_a);
{-performs local strategy HEURISTIC}
{ans = 0     if user wants to quit
       1..26 number of answers
       98    Unknown (no answer)
       99    Not applicable
       255   if memory problem
         numeric = true if answers are numeric
         val = strings for answer values
         num = numeric answers
         cer = certainty for answer}

   function find_best:byte;
   {-find best question}
   {-returns number of question}
   {-0 if no more question can be selected}


procedure meta;

   function add_control:boolean;
   {-adds a new control field}

   function update_control:boolean;
   {-update control fields}

      procedure init_control(which:byte);
      {-inits a new called table}

   procedure call_tables;
   {-invokes new tables from left to right}
```

unit hiclutil;

```
{ ************************************************************************** }
{                                                                          * }
{ * This unit is part of HICLASS (c) 1992                                  * }
{ * Program author : Jens Wazel                                            * }
{ * Programming environment : Turbo Pascal 6.0                             * }
{ *                                                                        * }
{ ************************************************************************** }
{ * Services : Support local control strategies                            * }
{ ************************************************************************** }
{ * Actions  :                                                             * }
{ ************************************************************************** }

interface

uses crt,dos,lsutil,hiall,hiclread,hiclload;

type q_a = record
                 ans:byte;
                 numeric:boolean;
                 val:array[1..abs_max_val] of string[9];
                 num:array[1..abs_max_val] of byte;
                 cer:array[1..abs_max_val] of byte;
           end;

           {holds answers to a question after ask_user or ask_table}
           {ans = 0      if user wants to quit
                  1..26 number of answers
                  98    Unknown (no answer)
                  99    Not applicable
                  255   if memory problem
            numeric = true if answers are numeric
            val = strings for answer values
            num = numeric answers
            cer = certainty for answer}

     {multiple answers for interface control strategy - reduce table}
     mans1=array[1..abs_max_attr*26] of string[9];  {answer value}
     mans2=array[1..abs_max_attr*26] of byte;       {numeric answer}
     mans3=array[1..abs_max_attr*26] of byte;       {question}


procedure dispose_garbage(table:table_pointer);
   {-dispose all of the table content}

procedure show_results(table:table_pointer;var qa:q_a);
   {ans = 0      if user wants to quit
          1..26 number of results
          98    Unknown (no result)
          99    Not applicable
          255   if memory problem
            numeric = true if answers are numeric
            val = strings for result values
            num = numeric results
            cer = certainty for results}

procedure check_for_numeric(table:table_pointer);
   {-detects numeric values for attribute}
   {-and stores them in ascending order for later use}

function update_history(table:table_pointer;
   ph:attr_pointer;ans:byte;numer:boolean;val:string;nu,cer:byte):boolean;
   {-adds new information to the history}

procedure show_history;
   {-shows the complete history of the current session}
```

```
procedure check_global(table:table_pointer;p:attr_pointer;var cg:q_a);
   {-search for answer to a question in global list}
   {-returns  g_a  with ans=254 if not found}

function add_global(attr,value:string;numval,cert:byte):boolean;
   {-adds the value for a global attribute to the global list}
   {-returns false if memory problem, true otherwise}

function check_unique_values(table:table_pointer;question:byte):boolean;
   {-check if there is only a unique value left for current question}

function check_unique_result(table:table_pointer):boolean;
   {-check if there is a unique result left}

function check_table_solved(table:table_pointer):boolean;
   {--check if only one result with one weight (if shortcut) left}

procedure delete_nonvalid_values(table:table_pointer);
   {-Deletes values not valid anymore}
   {-Update examples accordingly}

function reduce_table(table:table_pointer;max,question:byte;numer:boolean;
value:mans1;numval:mans2;ques:mans3):boolean;
   {--reduce the table according to the result of a question}
   {--return true if table is solved, false otherwise}

procedure init_screen(table:table_pointer);
   {-Initialize the screen}

procedure dispose_lists;
   {-Dispose all control lists}


implementation

{ ************************************************************************** }

procedure conclude(table:table_pointer);
{-concludes other values of the results}

   procedure find_ex;
   {-find corresponding example of original table}

   procedure add_val;
   {-add values of attributes}

   procedure add_result;
   {-add result}

procedure show_results;

   procedure update_num;
   {-Update numeric fields for result}

   procedure make_certainty(table:table_pointer);
   {-Make certainty calculation for results}

      function min:byte;
      {-Find minimum certainty for example values}

   procedure reduce_result(table:table_pointer);
   {-reduce result set for certainty=0}

   procedure update_result_history;
   {-update history for results}

   function check_text:boolean;
   {-check if result texts should be given}
```

```
   procedure check_and_replace_dollar(var st:string;hhj,pos:byte);
   {-adds certainty values to the next if necessary}

   procedure add_result_text1;
   {-add result texts to reader not numeric}


   procedure add_result_text2;
   {-add result texts to reader numeric}

   procedure build_result;
   {-build result set}

procedure delete_nonvalid_values;

   function check_ex(question,value:byte):boolean;
   {-check if value is still valid in examples}
   {-returns true if value is still in use}


function reduce_table;

   procedure reduce_table_numeric;
   {-build value ranges for numeric values in examples}
   {-delete examples with a range not appropriate for answer}
```

unit hiclread;

```
{ *************************************************************************** }
{ *                                                                         * }
{ * This unit is part of HICLASS (c) 1992                                   * }
{ * Program author : Jens Wazel                                             * }
{ * Programming environment : Turbo Pascal 6.0                              * }
{ *                                                                         * }
{ *************************************************************************** }
{ * Services : Install new reader                                           * }
{ *            Display reader                                               * }
{ *            Remove reader                                                * }
{ *************************************************************************** }
{ * Actions  :                                                              * }
{ *************************************************************************** }

interface

uses crt,Lsutil;

type  exitread = (rno,resc,rcr,rF1,rF2,rF3,rF4,rF5,rF6,rF7,rF8,rF9,rF10);
      {ways to exit menu selection}

var readexit:exitread;       {way of exit reader}
    readpos:integer;         {selected position in reader}
    readval:byte;            {numeric value returned}

function install_read
   (var name:pointer;          {Identifier returned}
    xpos:byte;                 {Upper left corner}
    ypos:byte;
    row:byte;                  {maximal number of rows on screen}
    width:byte;                {width of rows}
    headline:string            {Title for reader}
    ):boolean;                 {Returns true if successful}
    {-Allocate and initialize, but do not display, a new reader}

procedure install_read_col
   (name:pointer;
    back:byte;                 {Color of shadow}
    frame:byte;                {Color of Frame}
    ntext:byte;                {Color of normal text rows}
    text:byte;                 {Color of unselected row}
    highlight:byte;            {Color of selected row}
    pg:byte;                   {Color of PgUp...}
    numbcol:byte;              {Color of numbering}
    head:byte);                {Color of headline}
    {-Define the colors for a menu}


procedure add_row_read(name:pointer;rtext:string;rselect,rrow:integer);
    {-Add a row to reader identified by name at row rrow}
    {-if rselect=1 then row is selectable, else rselect=0}
    {-if rselect=2 then numeric value is needed}
    {-if rrow=0 then append row}

procedure delete_read_from_screen(name:pointer);
    {-Dispose heap space for window}

procedure delete_read_from_memory(name:pointer);
    {-Dispose all of reader heap space}

procedure reset_reader(name:pointer);
{-Deactivate reader, erase rows, leave reader on screen}
```

```
function showread
    (name:pointer;
    shadowed:boolean;         {true if a shadow is wished}
    delaftershow:boolean      {true if reader should be erased after selection}
    ):boolean;                {Return true if succesful}
    {~Display reader, let user browse it if provided, return readpos and way of exit}
```

unit himenu;

```
{ *********************************************************************** }
{ *                                                                     * }
{ * This unit is part of HICLASS, HIEDIT (c) 1992                       * }
{ * Program author : Jens Wazel                                         * }
{ * Programming environment : Turbo Pascal 6.0                          * }
{ *                                                                     * }
{ *********************************************************************** }
{ * Services : Install new pull down menus                              * }
{ *            Display pull down menus                                  * }
{ *            Change content of pull down menus                        * }
{ *            Remove pull down menus                                   * }
{ *            Screen saver routines                                    * }
{ *********************************************************************** }
{ * Actions  : Disable screen saver                                     * }
{ *********************************************************************** }

interface

uses crt,Lsutil,hiintr;

type  exitmenu = (no,esc,cr,F1,F2,F3,F4,F5,F6,F7,F8,F9,F10,left,right);
      {ways to exit menu selection}

var menuexit:exitmenu;           {way of exit menu}
    menupos:byte;                {selected position in menu}

function install_menu
   (var name:pointer;          {Identifier returned}
    xpos:byte;                 {Upper left corner}
    ypos:byte;
    row:byte;                  {number of items per column}
    col:byte;                  {number of columns}
    width:byte;                {width of items}
    dyna:boolean;              {dynamic reduction of row according
                                to # of items allowed}
    headline:string;           {Title for Menu}
    hxpos:byte;                {Start position for help to item}
    hypos:byte;
    hlength:byte               {maximal length of background for help to item
                                hlength=0 -> no helpitems at all}
    ):boolean;                 {Returns true if successful}
    {-Allocate and initialize, but do not display, a new pulldown menu}

procedure install_menu_col
   (name:pointer;
    back:byte;                 {Color of shadow}
    frame:byte;                {Color of Frame}
    text:byte;                 {Color of unselected item}
    highlight:byte;            {Color of selected item}
    pg:byte;                   {Color of PgUp...}
    head:byte;                 {Color of headline}
    hcolor:byte;               {Color of help to item}
    hotcolor1:byte;            {Color of hotkey unselected}
    hotcolor2:byte);           {Color of hotkey selected}
    {-Define the colors for a menu}

procedure add_item (name:pointer;nitem,helpitem:string;position:byte);
    {-Add an item and help to menu identified by name at position}
    {-Hotkeys will be identified by a heading "°" (ALT+248)}
    {-if position=0 then add at end}

procedure move_item(name:pointer;source,dest:byte);
    {-Move an item from position source to dest within the menu}

function delete_item(name:pointer;position:byte):byte;
    {-Delete an item from position pos, return # of items left}
```

```
procedure change_item(name:pointer;nitem,helpitem:string;position:byte);
    {-Change the content of an item}

procedure reset_menu(name:pointer;reset_pos:boolean);
    {-Deactivate menu, erase items, leave menu on screen}
    {-if reset_pos then set position=1 else leave old position alone}

procedure reset_headline(name:pointer;headline:string);
    {-Enter new headline for menu, used in conjunction with reset_menu}
    {-Order: reset_menu -> reset_headline -> showmenu}

procedure delete_menu_from_screen(name:pointer);
    {-Dispose heap space for window}

procedure delete_menu_from_memory(name:pointer);
    {-Dispose all of menu heap space}

function showmenu
    (name:pointer;
    shadowed:boolean;        {true if a shadow is wished}
    delaftershow:boolean     {true if menu should be erased after selection}
    ):boolean;               {Return true if succesful}
    {-Display menu system, let user browse it, return menupos and way of exit}

procedure reset_screen_saver;
    {-Start screen saver waiting time}

procedure set_screen_saver_time(time:integer);
    {-Set time screen saver waits (in seconds)}

procedure set_screen_saver(onoff:byte);
    {-Enable or disable screen saver}

procedure screen_saver;
    {-Counting loop for screen saver and performance}
```

```
unit hispread;

{ ************************************************************************** }
{ *                                                                        * }
{ * This unit is part of HIEDIT (c) 1992                                   * }
{ * Program author : Jens Wazel                                            * }
{ * Programming environment : Turbo Pascal 6.0                             * }
{ *                                                                        * }
{ ************************************************************************** }
{ * Services : Install new spreadsheet menu                                * }
{ *            Display spreadsheet menu                                    * }
{ *            Change content of spreadsheet menu                          * }
{ *            Remove spreadsheet menu                                     * }
{ ************************************************************************** }
{ * Actions  :                                                             * }
{ ************************************************************************** }

interface

uses crt,Lsutil;

type  exitspread = (sno,sesc,scr,sF1,sF2,sF3,sF4,sF5,sF6,sF7,sF8,sF9,sF10);
      {ways to exit menu selection}

var spreadexit:exitspread;       {way of exit menu}
    spreadcpos,
    spreadrpos:byte;               {selected position in menu}

function install_spread
   (var name:pointer;          {Identifier returned}
    xpos:byte;                 {Upper left corner}
    ypos:byte;
    row:byte;                  {maximal number of rows on screen}
    col:byte;                  {maximal number of columns on screen}
    width:byte;                {width of items}
    headline:string;           {Title for Menu}
    hxpos:byte;                {Start position for help to column}
    hypos:byte;
    hlength:byte               {maximal length of background for help to column
                                hlength=0 -> no helpitems at all}
    ):boolean;                 {Returns true if successful}
    {-Allocate and initialize, but do not display, a new pulldown menu}

procedure install_spread_col
   (name:pointer;
    back:byte;                 {Color of shadow}
    frame:byte;                {Color of Frame}
    text:byte;                 {Color of unselected item}
    highlight:byte;            {Color of selected item}
    pg:byte;                   {Color of PgUp...}
    head:byte;                 {Color of headline}
    hcolor:byte;               {Color of help to item}
    headcol:byte;              {Color of column headings unselected}
    headhigh:byte;             {Color of column headings selected}
    numbcol:byte;              {Color for numbered option}
    clrcol:byte);              {Color for clrscr (not full)}
    {-Define the colors for a menu}

procedure add_column_headline_spread(name:pointer;headl,helpitem:string;position:byte);
   {-Add a headline and a help for a column}

procedure change_column_headline(name:pointer;headl,helpitem:string;position:byte);
   {-Change the content of a column headline}

procedure add_row_spread(name:pointer;rrow:byte);
    {-Add a row to menu identified by name at row rrow}
    {-if rrow=0 then append row}
```

```
procedure delete_row_spread(name:pointer;drow:byte);
    {-Delete row drow in menu identified by name}

procedure add_item_spread (name:pointer;nitem:string;ncol,nrow:byte);
    {-Add an item to menu identified by name at column ncol and row nrow}
    {-if ncol=0 then append item in row}

procedure change_item_spread(name:pointer;nitem:string;chcol,chrow:byte);
    {-Change the content of an item}

function delete_item_spread(name:pointer;dcol,drow:byte):byte;
    {-Delete an item from position dcol,drow; return # of items left in row}

procedure move_item_spread(name:pointer;scol,srow,dcol,drow:byte);
    {-Move an item from position scol,srow to dcol,drow within the menu}

procedure delete_spread_from_screen(name:pointer);
    {-Dispose heap space for window}

procedure delete_spread_from_memory(name:pointer);
    {-Dispose all of menu heap space}

procedure reset_spread(name:pointer;reset_pos:boolean);
    {-Deactivate menu, erase items, leave menu on screen}
    {-if reset_pos then set position to first row,column}
    {-else leave old position alone}

procedure inc_row_spread(name:pointer);
    {-inc current row if possible}
    {-provided an addition took place}

procedure inc_col_spread(name:pointer);
    {-inc current col if possible}

procedure dec_col_spread(name:pointer);
    {-dec current col if possible}

procedure reset_col_spread(name:pointer);
    {-set current col to 1}

procedure reset_row_spread(name:pointer);
    {-set current row to 1}

procedure dec_row_spread(name:pointer);
    {-dec current row if possible}
    {-provided a deletion took place}

procedure reset_headline_spread(name:pointer;headline:string);
    {-Enter new headline for menu, used in conjunction with reset_spread}
    {-Order: reset_spread -> reset_headline_spread -> showspread}

procedure clear_the_screen(name:pointer);
    {-in case of full=false, a clrscr is needed when examples are deleted}

function showspread
    (name:pointer;
    full:boolean;            {true if frame/lines etc. should be shown}
    justshow:boolean;        {true if menu should be shown w/o selection}
    numbered:boolean;        {true if rows should be numbered}
    shadowed:boolean;        {true if a shadow is wished}
    delaftershow:boolean     {true if menu should be erased after selection}
    ):boolean;               {Return true if succesful}
    {-Display menu system, let user browse it, return menupos and way of exit}
```

```
unit hieditor;

{ ******************************************************************************** }
{ *                                                                              * }
{ * This unit is part of HICLASS,HIEDIT (c) 1992                                 * }
{ * Program author : Jens Wazel                                                  * }
{ * Programming environment : Turbo Pascal 6.0                                   * }
{ *                                                                              * }
{ ******************************************************************************** }
{ * Services : Edit a text                                                       * }
{ *             Display directory menu                                           * }
{ *             File handling for text                                           * }
{ ******************************************************************************** }
{ * Actions   :                                                                  * }
{ ******************************************************************************** }

interface

uses dos,crt,printer,lsutil,lshelpk,himenu,hiall;

function edit(table_str,attribute_str,value_str:string;var htext:main_text):boolean;
 {--Edit htext and display table_str and attribute_str and value_str at the top}

function topdir_menu(modus,x1,y1,rows,colu,backcol,framecol,textcol,highcol:byte;
                      suff:string; var dir_string,dir_of_file:string):byte;
   {--Display directory with rows rows and colu columns with colors *col
      at x1,y1, let user browse through and select filename dir_string
      with suffix suff in path dir_of_file, modus: 1=wide 2=normal
      returns
           0: if succesful
           1: if there is not enough memory
           2: if there is a disk error}

procedure init_text(var te:main_text);
    {--initialize text}

function save_text(te:main_text;var fi:text):boolean;
    {--append te to fi}

function load_text(var te:main_text;var fi:text):boolean;
    {--load te from fi}
```

```
unit LSutil;

{ ************************************************************************ }
{ *                                                                      * }
{ * Program author : Jens Wazel                                          * }
{ * Programming environment : Turbo Pascal 6.0                           * }
{ *                                                                      * }
{ ************************************************************************ }
{ * Services : All kinds of useful routines                              * }
{ ************************************************************************ }
{ * Actions  : Detect present graphic card mode (grafik/mono)            * }
{ *            Detect memory location of text screen                     * }
{ *            Save configuration elements                               * }
{ ************************************************************************ }

interface

uses crt,dos;

const MaxDirSize=128;

type BufPtr = ^BufferArray;
     BufferArray = array[0..MaxInt] of byte; {for SaveWindow}

     display_card  = (mono,grafik);
     color_mode    = (farbe,sw);
     dirptr        = ^DirRec;
     dirrec        = record
                       Attr: Byte;
                       Time: Longint;
                       Size: Longint;
                       Name: string[12];
                     end;
     dirlist      = array[0..MaxDirSize-1] of DirPtr;

var  present_card:display_card;             {Present display card}
     start_of_buffer:longint;               {Start adress of display page}
     video_mode:color_mode;                 {Present color mode}
     cop1,cop2:string;
     ins_mode:boolean;                      {true if INSERT is active}
     DispMode:byte;                         {Present video mode}

procedure RestoreConfig;
 {-Restore display mode, window, TextAttr, Cursor which are automatically saved}

function SaveWindow(x1, y1, x2, y2 : Byte; Allocate : Boolean;
                    var Pscrstore : Pointer) : Boolean;
 {-Save the specified window in and allocate buffer space if requested}

procedure RestoreWindow(x1, y1, x2, y2 : Byte;
                        Deallocate : Boolean; var Pscrstore : Pointer);
 {-Restore specified window and deallocate buffer space if requested}

procedure print(wort:string;e,r:byte;whzahl:integer;attribut:byte);
 {-Print a string at e,r whzahl times with color attribut}

procedure rahmen(x1,x2,y1,y2,line,farbe,fhead:byte;headline:string);
 {-Draw a frame around the window specified by x1,y1,x2,y2 with
   line type in color farbe and prints a headline in color fhead}

function rco:string;
procedure cooo;

procedure read_cu_mode;
 {-Read the present color_mode and save it in cu_lines}

procedure cu(control:byte);
 {-Turn cursor on (control=1) or off (control=0)}
```

```
procedure SaveWindowCursor(var p:pointer);
 {-Save present window, cursor coordinates, cursor status and textattribute}

procedure RestoreWindowCursor(var p:pointer);
 {-Restore window, cursor coordinates, cursor status annd textattribute}

function file_name_string(filenam:string):boolean;
 {-returns true if filenam could be a filename}

function  printer_ok:boolean;
 {-returns true if printer is ready for working}

procedure sound_message;
 {-Make a special sound}

procedure reads(var s:string;l,textcol:byte);
 {-Read a string in textcol with maximal l characters from the keyboard}

procedure qw;
 {-Wait for a keystroke and do not affect any keyboard input in the background}

function os_shell:byte;
 {-Call os_shell and return
                              0: if call was succesful
                              1: if COMMAND.COM is not present
                              2: if there is not enough memory}

function topchdir(x1,y1,framecol,textcol,textcolh:byte):boolean;
 {-Display a message and change into wished directory if possible
   print frame at x1,y1 with framecol, normal text with textcol and
   intensive text with textcolh, returns true if succesful}

function topdir(x1,y1,framecol,textcol:byte):byte;
 {-Display directory with framecol and textcol at x1,y1, returns
                              0: if succesful
                              1: if there is not enough memory
                              2: if there is a disk error}

function textaroundcursor(max_len,xg,yg:byte):string;
 {-Returns the string around position xg,yg with maximal length max_len}

function get_calling_path(hilfstr:string):string;
 {-Returns the calling path of the program}
```

unit LShelpk;

```
{ *************************************************************************** }
{ *                                                                         * }
{ * Program author : Jens Wazel                                             * }
{ * Programming environment : Turbo Pascal 6.0                              * }
{ *                                                                         * }
{ *************************************************************************** }
{ * Services : Provide a context sensitive help system                      * }
{ *            (help files created with HELPEK  (c) 1991 by Jens Wazel)     * }
{ *************************************************************************** }
{ * Actions  : Detect present graphic card mode (grafik/mono)               * }
{ *************************************************************************** }

interface

uses dos,crt,LSmenu,LSutil;

const absmaxhelp=120;             {maximal number of helps in a system}
      max_file=5;                 {maximal number of help_files in a system}
      xa=16;                      {Coordinates}
      ya=6;                       {of help window}
      xb=65;
      yb=22;
      eb=xb-xa+1;                 {maximal number of text columns}
      eh=yb-ya+1;                 {maximal number of text rows}
      max_hervor=20;              {maximal number of accentuated strings}
      max_quer=8;                 {maximal number of cross connections}
      es=8;                       {maximal number of pages per help}

type name_string = string[10];
     help_colors = array[1..12] of byte;
     control_main = record        {Control table for help system}
                      name:name_string;
                      seek:integer;
                    end;
     display_card  = (mono,grafik);
     color_mode    = (farbe,sw);

var colo:help_colors;                     {Colors for display help}
    quernum:byte;                         {Choosed cross connection
                                           (mode queron=false) }
    help_co:array[1..max_file,1..absmaxhelp] of control_main;
                                          {Control table of system}
    endcol:byte;                          {color of ende_request}
    level:byte;                           {level of difficulty}
    sdelay:word;                          {delay in show_mode}
    present_card:display_card;            {Present display card}
    video_mode:color_mode;                {Present color mode}


function detect_and_init
   (hfile:string;           {Filename of helpfile}
   status:string;           {Status line}
   stx,sty:byte             {Position of status line}
   ):boolean;               {Returns true if successful}
   {-Look for helpfile and initialize help system}
```

```
function hilfe
   (kennung:name_string;   {Name of help}
    savebefore:boolean;    {true if save last screen}
    clrbefore:boolean;     {true if clrscr before display help}
    shadowed:boolean;      {true for drawing a shadow}
    drawhelpscreen:boolean;{true for clrscr and drawing a frame}
    show_delay:boolean;    {true if show_mode with delays is active}
    showsignal:boolean;    {true if PgUp/PgDn information should be given}
    queron:boolean;        {true if cross connection work is allowed}
    startpage:byte;        {figure of first shown page, last page=9}
    Pgexit:boolean;         {true if exit at last page is allowed with PgDn
                               and exit at first page is allowed with PgUp}
    total_ende_request_at_esc:boolean {true if this request should be done}
    ):byte;   {-Display help and returns
                               0 if Successful
                               2 if user escaped with PgUp
                               3 if user escaped with PgDn
                               4 if Disk error
                               5 if Help is not available
                               6 if Heap Overflow
                               7..14 if user escaped with ESC
                                     (page number:= - 6) }

procedure set_help_colors;
{-Sets all colors of the help-screen (basic settings)}
```

The HIEDIT/HICLASS package provides the means to build an Expert System for any problem which can be solved using hierarchical classification.

Copyright

* (c) Copyright 1992 Jens Wazel
* 331 Oxford College Hall Oxford, OH 45056 (513) 529-6522
* E-mail: JWAZEL@MIAMIU

Program type

* HIEDIT : Expert System Editor.
* HICLASS : Expert System Shell.
  Hierarchical Classification for any problem type.

Written in

* TURBO PASCAL 6.0 (16.000 lines).

Programming language

* Not required to use HIEDIT and HICLASS.

User interface

* Pull-down menus, spreadsheets.
* Full screen editor.
* Context-sensitive help system (over 100 help screens).

Knowledge representation

* Tables in a hierarchy.
* Several descriptions for one concept.
* Several concepts combined in one table.
* Concepts described by attributes with values.
* Certainty values (weights) for each concept.

Size of one table

* Up to 12 attributes, 26 values per attribute.
* Up to 255 concept descriptions.

Size of hierarchy

* Almost unlimited.
* Tables only invoked when needed.
* Minimal amount of information kept in main memory.

Data types

* Logical and numeric (interval-based).

Global attributes

* Attributes can be defined globally.

Don't care

* Concept descriptions can include "Don't care" values.

Questions

* Are asked to acquire data using customized text screens.

Answers

* Can be entered by the user (multiple choice, numeric).
* Can be provided by a subtree of tables.

"UNKNOWN", "NOT APPLICABLE"

* Two special answer options.

Global control strategy

* Depth-first, several paths.
* Tables can be used several times in the hierarchy.

Local control strategies

* Goal: minimum amount of questions asked.
* Chosen by user or automatically.
* MATCH: ask all questions and compare to table content (database search).
  LEFT-TO-RIGHT: ask questions left to right, reduce table content.
  HEURISTIC: heuristic decides which questions to ask, reduce table content.

Uncertainty handling

- Based on certainty and fuzzy set theory.
- Derived from weights of concepts and amount of answers UNKNOWN.
- Certainty values combined for paths.
- Thresholds for path termination.

Session report

- Session report file built automatically on disk.
- Interrupted session can be resumed.

History

- All questions, answers, and conclusions of current session.

Conclude

- All values for results.

Inductive learning

- Inductive learning algorithm for creating a distinction-oriented knowledge representation used by hierarchical hypothesis matcher HIHYPO (c) 1992 Jens Wazel.

Example

- '*.HIT' on your program disk; root table: 'EXAMPLE1'.
- Have a look at the table definitions with HIEDIT.
- Load 'EXAMPLE1' into HICLASS and classify animals.

More information

- Access the help system.
- Write, call, or send an E-mail to Jens Wazel.

Jens' thesis

Revision as of 1/26/93

1. Rethinking the evaluation of answers UNKNOWN

In section 3.10. it was explained why it is necessary to incorporate the information about answers UNKNOWN into the certainty of the result(s) of a table. The approach developed there was to count the number of answers UNKNOWN for a particular table, and to weaken each result by the ratio

$$r = \frac{\Sigma \text{ questions} - \Sigma \text{ UNKNOWN}}{\Sigma \text{ questions}}$$

while multiplying this ratio with the certainty of each result.

This approach of incorporating the amount of answers UNKNOWN is justified, if the table has more than one result. If, on the other hand, the result set of the table only consists of one result, then the certainty of this result should NOT be weakened. This is due to the limited world view applied for each table. Let us consider an example:

type	size	location	creature	weight
cetacea	25 ft.	at sea	whale	1.0
cetacea	6 ft.	near coast	porpoise	1.0
cetacea	6 ft.	at sea	dolphin	1.0
fish	1 ft.	n.pacific	salmon	1.0
fish	6 ft.	at sea	shark	1.0

If the answers of the user are

<UNKNOWN>, <UNKNOWN>, <at sea>

then the result set would be the following

type	size	location	creature	weight
cetacea	25 ft.	at sea	whale	1.0
cetacea	6 ft.	at sea	dolphin	1.0
fish	6 ft.	at sea	shark	1.0

It should be clear that these three results cannot be given with 100% certainty. The fact that 2 out of three questions could not be answered weakens the quality of the results. Therefore, it is totally correct to derive

$$r = (3 - 2) / 3 = 0.3$$

and to give the results

whale	0.3
dolphin	0.3
shark	0.3

If, on the other hand, the user answers with

<UNKNOWN>, <25 ft.>, <at sea>

the result set would only consist of one element, a <whale>. Despite the unknown *type*, a unique result could be found. We can be sure that the animal to be classified is in fact a <whale>, since it is unique with respect to its size. In this case, the certainty of the result should NOT be weakened.

Given the above reasoning, the following changes have been made to the HICLASS system:

> The amount of answers UNKNOWN weakens the certainty of the results if, and only if, there are several results.
>
> If there is only one result of the table, the amount of answers UNKNOWN is NOT incorporated in the calculation of the certainty for that result.

2. Providing a command line option for HICLASS

While working on an implementation of HICLASS (a topic advisor for the German department) it became clear that in some cases it is not necessary, or even desired, to allow a user to have access to the full power of HICLASS. This is true for the first FILES screen as well as for options like CONCLUDE and HISTORY. Therefore, the following changes have been made to HICLASS:

The program CAN now be called with the following parameters:

```
HICLASS [tname] [-[e[i[c]]]] [+headline]
```

Explanation of parameters:

1. **tname**
    - tname = name of root table to be called
    - FILES screen is not shown
    - a report file cannot be stored
    - after an interrupt or an advice the program halts
    - the name of the help file is changed from HICLASS.HLP to a customized help file tname.hlp
2. -eic
    - following the '-', the following options can be disabled:
        - e: disable EXPLAIN
        - i: disable HISTORY
        - c: disable CONCLUDE
3. **headline**
    - the predefined headline showing root and actual tables is replaced by *headline*, which follws a '+'

Calling HICLASS without any parameter, nothing changes. The system is customized by calling the program with one, two or all three of the parameters.

The HIEDIT/HICLASS package provides the means to build an Expert System for any problem which can be solved using hierarchical classification.

Copyright

* 331 Oxford College Hall Oxford, OH 45056 (513) 529-6522
* E-mail: JWAZEL@MIAMIU

Program type

* HIEDIT : Expert System Editor.
* HICLASS : Expert System Shell.
            Hierarchical Classification for any problem type.

Written in

* TURBO PASCAL 6.0 (16.000 lines).

Programming language

* Not required to use HIEDIT and HICLASS.

User interface

* Pull-down menus, spreadsheets.
* Full screen editor.
* Context-sensitive help system (over 100 help screens).

Knowledge representation

* Tables in a hierarchy.
* Several descriptions for one concept.
* Several concepts combined in one table.
* Concepts described by attributes with values.
* Certainty values (weights) for each concept.

Size of one table

* Up to 12 attributes, 26 values per attribute.
* Up to 255 concept descriptions.

Size of hierarchy

- Almost unlimited.
- Tables only invoked when needed.
- Minimal amount of information kept in main memory.

Data types

- Logical and numeric (interval-based).

Global attributes

- Attributes can be defined globally.

Don't care

- Concept descriptions can include "Don't care" values.

Questions

- Are asked to acquire data using customized text screens.

Answers

- Can be entered by the user (multiple choice, numeric).
- Can be provided by a subtree of tables.

"UNKNOWN", "NOT APPLICABLE"

- Two special answer options.

Global control strategy

- Depth-first, several paths.
- Tables can be used several times in the hierarchy.

Local control strategies

- Goal: minimum amount of questions asked.
- Chosen by user or automatically.
- MATCH: ask all questions and compare to table content (database search).
LEFT-TO-RIGHT: ask questions left to right, reduce table content.
HEURISTIC: heuristic decides which questions to ask, reduce table content.

Uncertainty handling

- Based on certainty and fuzzy set theory.
- Derived from weights of concepts and amount of answers UNKNOWN.
- Certainty values combined for paths.
- Thresholds for path termination.

Session report

- Session report file built automatically on disk.
- Interrupted session can be resumed.

History

- All questions, answers, and conclusions of current session.

Conclude

- All values for results.

Inductive learning

- Inductive learning algorithm for creating a distinction-oriented knowledge representation used by hierarchical hypothesis matcher HIHYPO (c) 1992 Jens Wazel.

Example

- '*.HIT' on your program disk; root table: 'EXAMPLE1'.
- Have a look at the table definitions with HIEDIT.
- Load 'EXAMPLE1' into HICLASS and classify animals.

More information

- Access the help system.
- Write, call, or send an E-mail to Jens Wazel.