1 INTRODUCTION

MARVIN's approach to learning a concept is two-fold. First of all, it requires a trainer to give examples of the concept to be learned. Secondly, the system asks questions like a good student in order to figure out exactly what the desired concept is.

At the beginning of the learning process the trainer presents the system with a positive example of the desired concept. MARVIN considers all the possible classes that the example or parts of it can be put into. Using these classes, it constructs all possible generalizations of the example given. One of these generalizations has to be the target concept.

Next, MARVIN starts generalizing the example in a systematic manner by using the set of classes obtained earlier. After a generalization is hypothesized, it is tested by constructing an example of the hypothesized generalization and presenting it to the trainer. If the trainer says that the example is in the target concept, the system continues by attempting to generalize the newly accepted generalization further. If the trainer says that the object is not in the target concept the current hypothesis is incorrect and the system specializes it because the hypothesized concept might have been too general. If no specialization of the trial (hypothesized) concept can be made, a completely new generalization is tried.

Once there are no more generalizations left to be made, the system must have converged on to the correct concept.

Needless to say, the above discussion is quite vague. In order to clarify some of the main ideas, a few definitions will be required. But to begin with, here is an example of the operation of MARVIN.

2 EXAMPLE OF MARVIN

Here is an example of MARVIN in action. The concept that we are trying to teach is the concept of a garden. We define a garden as a list of vegetables. As background information, we teach MARVIN the concept of vegatables, fruits and food. Food in this domain is just vegetables and fruits since the author is a vegetarian.

```
C-Prolog version 1.5
| ?- [marvin],marvin.
marvin consulted 36412 bytes 8.08333 sec.

MARVIN v3.10

<marvin> learn.
----------------------------------------
<learn concept> veggie.
Arity: 1.
----------------------------------------
Example of 'veggie'  (A): carrot.
A is a 'veggie' if:
        A is carrot
----------------------------------------
The generalization =
A is a 'veggie' if:
        A is carrot
Saving a disjunct of the concept ('veggie')
----------------------------------------
Example of 'veggie'  (A): onion.
A is a 'veggie' if:
        A is onion
----------------------------------------
The generalization =
A is a 'veggie' if:
        A is onion
Saving a disjunct of the concept ('veggie')
----------------------------------------
Example of 'veggie'  (A): pea.
A is a 'veggie' if:
        A is pea
----------------------------------------
The generalization =
A is a 'veggie' if:
```

```
        A is pea
Saving a disjunct of the concept ('veggie')
---------------------------------------
Example of 'veggie'  (A): no.
A is a 'veggie' if:
        A is carrot
A is a 'veggie' if:
        A is onion
A is a 'veggie' if:
        A is pea
---------------------------------------
<learn concept> fruit.
Arity: 1.
---------------------------------------
Example of 'fruit'  (A): apple.
A is a 'fruit' if:
        A is apple
---------------------------------------
The generalization =
A is a 'fruit' if:
        A is apple
Saving a disjunct of the concept ('fruit')
---------------------------------------
Example of 'fruit'  (A): orange.
A is a 'fruit' if:
        A is orange
---------------------------------------
The generalization =
A is a 'fruit' if:
        A is orange
Saving a disjunct of the concept ('fruit')
---------------------------------------
Example of 'fruit'  (A): banana.
A is a 'fruit' if:
        A is banana
---------------------------------------
The generalization =
A is a 'fruit' if:
        A is banana
Saving a disjunct of the concept ('fruit')
---------------------------------------
Example of 'fruit'  (A): no.
A is a 'fruit' if:
        A is apple
```

```
A is a 'fruit' if:
        A is orange
A is a 'fruit' if:
        A is banana
---------------------------------------
<learn concept> food.
Arity: 1.
---------------------------------------
Example of 'food'  (A): pea.
A is a 'food' if:
        A is pea
---------------------------------------
The trial concept is:
        veggie(A)

Is this:
        A is carrot
an example of the desired concept? yes.
---------------------------------------
The generalization =
A is a 'food' if:
        veggie(A)
Saving a disjunct of the concept ('food')
---------------------------------------
Example of 'food'  (A): banana.
A is a 'food' if:
        A is banana
---------------------------------------
The trial concept is:
        fruit(A)

Is this:
        A is apple
an example of the desired concept? yes.
---------------------------------------
The generalization =
A is a 'food' if:
        fruit(A)
Saving a disjunct of the concept ('food')
---------------------------------------
Example of 'food'  (A): no.
A is a 'food' if:
        veggie(A)
A is a 'food' if:
```

```
        fruit(A)
---------------------------------------
<learn concept> garden.
---------------------------------------
Example of 'garden'  (A): [onion,pea].
A is a 'garden' if:
        A is [onion,pea]
---------------------------------------
The trial concept is:
        veggie(head(A))
        A is [pea]

Is this:
        A is [carrot,pea]
an example of the desired concept? yes.
---------------------------------------
The trial concept is:
        food(head(A))
        A is [pea]

Is this:
        A is [apple,pea]
an example of the desired concept? no.
---------------------------------------
The trial concept is:
        veggie(head(A))
        veggie(head(tail(A)))
        the tail of the tail of A is []

Is this:
        A is [carrot,carrot]
an example of the desired concept? yes.
---------------------------------------
The trial concept is:
        veggie(head(A))
        food(head(tail(A)))
        the tail of the tail of A is []

Is this:
        A is [carrot,apple]
an example of the desired concept? no.
---------------------------------------
The trial concept is:
        veggie(head(A))
```

```
        garden(tail(A))

Is this:
        A is [carrot,carrot,carrot]
an example of the desired concept? yes.
---------------------------------------
The generalization =
A is a 'garden' if:
        veggie(head(A))
        garden(tail(A))
Saving a disjunct of the concept ('garden')
---------------------------------------
Example of 'garden'  (A): no.
A is a 'garden' if:
        veggie(head(A))
        the tail of A is []
A is a 'garden' if:
        veggie(head(A))
        garden(tail(A))
---------------------------------------
<learn concept> no.
<marvin>
```

3 DEFINITIONS

Object - the same as example.

Trial concept - a hypothesized generalization. In particular, it is the hypothesis currently being tested by MARVIN.

Target concept - the concept that the trainer wishes to teach MARVIN.

All elaborations - is the set of all possible concepts that an an example (or trial) and parts of the example can belong to.

Crucial object - an object with special properties which make it an object which can be shown to the trainer; MARVIN will only present this type of object.

Generalization - a concept is said to be more general than another if it describes all objects in the other concept plus some more.

Specialization - a specialization of a concept involves restricting what objects it covers to a subconcept.

4 THE CRITICAL ROUTINES OF MARVIN

Three sets of predicates make up the core of MARVIN.

Elaboration. This set contains the predicates required for elaborating examples and trials.

Generalization. This set contains the predicates required to generalize upon the examples given. This is the main process found in MARVIN.

Crucial object construction. This set contains the predicates required in finding the crucial objects to present to the trainer.

5 ELABORATION

One of the most important parts of MARVIN's algorithm is the process of elaboration. Elaboration is just deciding what possible classifications of an object or parts of an object can be made given the concepts already in memory.

For example, suppose that we have the concepts of digit and bit already in memory. The elaborations of the example `eq(A,1)` are `digit(A)`, `bit(A)` and `eq(A,1)` because all these are possible classifications.

We use the elaborations of a trial or an example in the constructing of hypotheses (trial concepts). Any possible generalization of a trial is a subset of the set of all elaborations of the trial. However, in order to construct a generalization, we need to know what replacement operations were used to obtain the set of elaborations.

For example, suppose that the bit class is a subset of the digit class. (It must be defined as a subclass in MARVIN; see [2]). The elaborations of `eq(A,1)` are `eq(A,1)`, `bit(A)`, `digit(A)` and the replacement operations used to obtain this list are `bit(A) :- eq(A,1)` and `digit(A) :- bit(A)`. To construct a hypothesis we apply a replacement operation on our currently accepted generalization. In this case, our accepted generalization would be `eq(A,1)` if 1 was presented by the trainer. Applying the first replacement operation (`bit(A) :- eq(A,1)`) we obtain the hypothesized generalization bit(A). Note that the other replacement operation cannot be applied yet. It can only be applied if the `bit(A)` generalization becomes accepted.

Another and more important use of the set of elaborations is in constructing crucial objects. See the section on crucial objects for more information.

The predicate `all_elaborations` is the only predicate used by the generalization predicates. All other predicates in this section are internal.

all_elaborations(Trial Concept,
Replacement Operations,
All Elaborations)

```
1       all_elaborations(Tr,ReOps,AE) :-
2               assert_trial(Tr),
3               bagof([A,B], elaboration(A,B), L),
4               usable_elaborations(L,Tr,ReOps,AE),
5               retract_trial(Tr),
6               !.

7       all_elaborations(Tr,[],Tr) :-
8               retract_trial(Tr),
9               !.
```

`all_elaborations` is given the trial to be elaborated as its first argument. The set of all elaborations possible is returned as the third argument. Meanwhile, the set of replacement operations used to create the elaborations is composed and is returned as the second argument.

The trial concept is asserted in order that normal Prolog matching can be used rather than by doing matching in lists or something similar.

Obtaining the bag of the elaborations is not sufficient because the elaboration predicate does not always report all items in the trial. It will report them all only if they can be used in some generalization. Also, in the bag there will be repetitions. Further, the bag is a list of replacement operation / elaboration pairs. To make them usable, they must be divided into two lists. The `usable_elaborations` predicate accomplishes this.

Note the use of the cut. We want only want one possible elaboration set, not all permutations of it.

Notice that the first clause may fail if there are not any replacements possible. In this case the second predicate will simply return the trial concept itself as all elaborations.

usable_elaborations(Bag of Elaborations,
Trial Concept,
Replacement Operations,
All Elaborations)

```
10      usable_elaborations(L,Tr,ReOps,AE) :-
11              bagof(X,firstmember(X,L),ReOps1),
12              bagof(Y,secondmember(Y,L),AE1),
13              flattened(ReOps1,ReOps2),
14              flattened(AE1,AE2),
15              norepeats(ReOps2,ReOps3),
16              norepeats(AE2,AE3),
17              union(Tr,AE3,AE),
18              delete_list( [[]], ReOps3, ReOps).

19      firstmember(X,L) :-
20              member([X,_],L).

21      secondmember(Y,L):-
22              member([_,Y],L).
```

Usable replacement operations are those that are in a list on their own (not along with their respective elaborations) and those that appear in the list only once. Note that [] replacement operations are useless.

Elaboration is useful if they too are in a list on their own, have no repeats and also include the items in the trial concept being elaborated.

---

elaboration(Replacement Operations, All Elaborations)

```
23      elaboration(ReOps,AE):-
24              concept(_,_,Hd),
25              prolog_trace(Hd,ReOps,AE).
```

This predicate says what is true given the current trial in the database. It reports if a classification is possible. If it is possible, the elaborations obtained and replacement operations used in the classification are reported.

prolog_trace(Goal, Replacement Operations, Satisfied Goals)

```
26      prolog_trace((A,B), ReOps, Trace) :-
27              prolog_trace(A,R1,T1),
28              prolog_trace(B,R2,T2),
29              concat(R1,R2,ReOps),
30              concat(T1,T2,Trace).

31      prolog_trace(A, [], [A]) :-
32              clause(A,true).

33      prolog_trace(A, [(A:-Body)|ReOps], [A|Trace]) :-
34              clause(A,Body),
35              ( not symmetric(A)  ;
36               ( symmetric(A) ->
37                      A =.. [_| Args],
38                      OGoal =.. [ordered|Args],
39                      OGoal )),
40              prolog_trace(Body,ReOps,Trace).
```

This predicate is a Prolog interpreter which reports back which goals it satisfied during its attempt to satisfy the major goal. Also reported are the clauses used during the process.

---

ordered(FirstArg, SecondArg)

```
41      ordered(A,B) :- var(A),var(B),!.
42      ordered(head(A),head(B)) :- !,ordered(A,B).
43      ordered(tail(A),tail(B)) :- !,ordered(A,B).
44      ordered(A,head(B)) :- !,ordered(A,B).
45      ordered(A,tail(B)) :- !,ordered(A,B).
46      ordered(head(A),B) :- !,ordered(A,B).
47      ordered(tail(A),B) :- !,ordered(A,B).
48      ordered(A,B) :- !,name(A,[A1]),name(B,[B1]),A1 < B1.
```

`ordered` is used to cut down on processing for symmetric predicates. It accomplishes this by only allowing those variables that are in order. For example, `same(A,B)` will be allowed but `same(B,A)` will not.

In order to make MARVIN realize that a concept is symmetric, put the predicate `symmetric` in the saved concept file. For example, in the file which contains the `same` concept put the line

```
symmetric( same(A,B) ).
```

6 GENERALIZATION

This set of predicates contains the core of MARVIN. MARVIN is just a system which attempts to generalize examples. This is exactly what these generalization predicates do.

generalization_of_example(Example, Generalization)

```
49        generalization_of_example(Ex,Gen) :-
50                all_elaborations(Ex,ReOps,AE),
51                generalization(Ex,ReOps,ReOps,AE,Gen).
```

Given an example of the target concept, this predicate gives the least specific generalization possible which is still consistent. This means that MARVIN will try to generalize the example as much as possible.

At all times during the generalization process there will be a currently accepted generalization. To start off with, we let the currently accepted generalization be the example itself since that is the most specific generalization possible.

Instead of having to recalculate all_elaborations of the currently accepted generalization several times, it is computed once and instantiated to the AE variable.

Also, the Replacement Operations computed while finding All Elaborations of the original example will be computed once, here in this predicate. These will be used in two ways: the first set will be used to generalize trials and the second set will be used to help specialize inconsistent generalizations.

generalization(Accepted Trial,
Replacement Operations for generalizing,
Replacement Operations for specializing,
All Elaborations, Generalization)

```
52      generalization(Tr,[],_,_,Tr).

53      generalization(Tr,[R|Rtail],ReOps,AE,Gen) :-
54              replacement(Tr,R,Tr1),
55              qualifies(gen,AE,Tr1,R,ReOps,Gen1),
56              all_elaborations(Gen1, ReOps1, GenAE),
57              intersect(ReOps, ReOps1, ReOps2),
58              generalization(Gen1,ReOps2,ReOps2,GenAE,Gen).

59      generalization(Tr,[R|Rtail],ReOps,AE,Gen) :-
60              not (Rtail = []),
61              not (replacement(Tr,R,_)),
62              append(R,Rtail,ReOps1),
63              generalization(Tr,ReOps1,ReOps,AE,Gen).

64      generalization(Tr,[R|Rtail],ReOps,AE,Gen) :-
65              delete(R,ReOps,ReOps1),
66              R = (A :- _),
67              all_elaborations([A],_,ElabsofA),
68              delete_list(ElabsofA,AE,NewAE),
69              generalization(Tr,Rtail,ReOps1,NewAE,Gen).
```

The currently accepted generalization is kept if it cannot be generalized any more.

If can be generalized using a certain replacement operation, we make the new generalization our currently accepted generalization if it qualifies. Note that a new set of replacement operations is created to be used in subsequent generalizations.

If this particular replacement operation cannot be used to generalize the currently accepted generalization, then we try to generalize using the replacement operations we haven't tried yet and save the current replacement operation for later.

If it doesn't qualify then we try to generalize using the replacement operations we haven't tried yet. Note that since the generalization didn't work out, we know that it cannot be used in later specializations.

replacement(Trial, Replacement Operation, New Trial)

```
70      replacement(Tr,(Head:-Body),NewTr) :-
71              structure_to_list(Body,BodyList),
72              subtract_list(BodyList,Tr,Tr1),
73              not member(Head,Tr1),
74              append(Head,Tr1,NewTr).
```

This takes the current trial and applies a replacement operation to it to create a new trial concept

---

qualifies(Gen or Spec,
All Elaborations,
Trial,
Last Replacement Operation,
All Replacement Operations,
Generalization)

```
75      qualifies(gen,AE,Tr,R,ReOps,Gen) :-
76              all_elaborations(Tr,_,AE1),
77              delete_list(AE1,AE,Diff),
78              !,
79              ((crucial_object(Tr,Diff,Obj) ->
80                  ask_trainer(Tr,Obj),
81                  Gen = Tr);
82              (specialization(Tr,ReOps,R,NewR,Spec),
83                 qualifies(spec,AE1,Spec,NewR,ReOps,Gen))).

84      qualifies(spec,AE,Tr,R,ReOps,Gen) :-
85              all_elaborations(Tr,_,AE1),
86              delete_list(AE1,AE,Diff),          %list_difference
87              !, not(Diff = []),                 % redundant specialization
88              ((crucial_object(Tr,Diff,Obj) ->
89                 ask_trainer(Tr,Obj),
90                 Gen = Tr);
91              (specialization(Tr,ReOps,R,NewR,Spec),
92                 qualifies(spec,AE,Spec,NewR,ReOps,Gen))).
```

To qualify a generalization, we must create a crucial object. If a crucial object can be found, we present it to the trainer and ask the trainer whether the object is in the target. If one cannot be found, or the trainer says the object is not in the target, we specialize the current trial.

Note there are two qualify clauses. The first qualifies generalizations while the second qualifies generalizations which have been specialized.

---

ask_trainer(Trial, Object)

```
 93        ask_trainer(Tr,Obj) :-
 94                draw_line,
 95                show(['The trial concept is: ',nl]),
 96                show_trial(Tr),
 97                show([nl,'Is this: ',nl]),
 98                show_trial(Obj),
 99                show(['an example of the desired concept? ']),
100                yes_no,
101                !.
```

Here we ask the trainer (oracle) if the object is in the target concept. Note that in a real system, the generalization (trial concept) would not be presented.

---

specialization(Trial,
All Replacement Operations,
Latest Replacement Operation,
Replacement Operation Used,
New Specialized Trial)

```
102        specialization(Tr,ReOps,R,NewR,NewTr) :-
103                non_disjoint_clause(NewR,ReOps,R,Int),
104                concat(Tr,Int,Tr1),
105                replacement(Tr1,NewR,NewTr).
```

To specialize a generalization we need the replacement operation which was used to create the generalization. We can create a generalization which is more specialized by combining two possible generalizations which have something in common. To do this, look through the list of replacement operations to find a clause which has a predicate in its body in common with a predicate in the body of the old replacement operation.

We fail if no such clause can be found. Otherwise, we create the new specialized trial.

---

non_disjoint_clause(Clause,
Replacement Operations,
Latest Replacement Operation,
Intersection of Clauses)

```
106        non_disjoint_clause((S:-M),ReOps,(S1:-M1),Int):-
107                member((S:-M),ReOps),
108                not ((S:-M) = (S1:-M1)),
109                structure_intersect(M,M1,Int),
110                not( Int = [] ).
```

7 CRUCIAL OBJECT CONSTRUCTION

Perhaps the most interesting part about MARVIN is the fact that it is a learning by experimentation system. To do experiments requires not only hypothesis formation, but also a method of testing these hypotheses. To do this, MARVIN constructs objects to present to his trainer.

We have four pieces of information that we use to construct a crucial object.

1. The Hypothesis (Trial Concept)
2. All Elaborations of the Trial Concept
3. The Currently Accepted Generalization
4. All Elaborations of the Currently Accepted Generalization

First of all, we find an object which is covered by the trial concept. Suppose that the currently accepted generalization was just the object presented by the trainer:

```
1,1        Prolog: eq(A,1), eq(B,1)
```

Also suppose that our trial concept is:

```
digit(A),digit(B)
```

Clearly we want to present an object like 0,4 or 1,3 etc, not an object that has nothing to do with digits (like mary,bob).

To test out our hypothesis we shouldn't present any objects covered by the currently accepted generalization. In this case, we shouldn't present 1,1 again. We know that all objects in the currently accepted generalization are positive examples of the target concept. Presenting one of these won't tell us anything new.

Now suppose that we have the concept `same(A,B)` in memory. In this case we should not present an object to the trainer that has both digits equal. If we do and the trainer verifies the object, we won't know if the object is in the target because both parts of the object are digits or because they have the additional property of being the same.

So, we should not produce an object that is in danger of being in the target concept for a reason other than the one currently being tested.

In satisfying these goals we will make an object called a crucial object, one that will adequately help us refute or verify our hypothesis.

---

crucial_object(Trial Concept,
Elaboration Difference,
Crucial Object)

```
111     crucial_object(Tr,Diff,Obj) :-
112             covered_object(Tr,Obj),
113             not infer(Diff,Obj),
114             !.

115     crucial_object(Tr,Diff,Obj) :-
116             recursive_concept(Tr,C1),
117             recursively_expanded(Tr,C1,Tr1),
118             covered_object(Tr1,Obj),
119             delete_list(Tr1,Diff,NewDiff),
120             !, % only try once, else infinate loop may occur
121             not infer(NewDiff,Obj),
122             !.


123     recursive_concept(P,C2):-
124             current_concept(C,C1),
125             functorcopy(C1,C2),
126             member(C2,P).

127     recursively_expanded(P,C,P1) :-
128             delete(C,P,P2),
129             current_concept(CName,_),
130             make_gen_clause(CName,P,(C:-C1)),
131             structure_to_list(C1,L1),
132             concat(L1,P2,P1).
```

The elaboration difference is all elaborations of the currently accepted generalization except for those that are also members of all elaborations the trial concept. The objects covered by items in this difference are those that are cov-

ered by the currently accepted generalization and those that can be in the target concept for a different reason other than the reasons in the trial concept.

For example, suppose the currently accepted generalization was the object 1,1 as above. All elaborations of it are

```
eq(A,1),eq(B,1),digit(A),digit(B),same(A,B)
```

Also suppose that the trial concept is digit(A),digit(B). All elaborations of it are

```
digit(A),digit(B)
```

The elaboration difference in this case is

```
eq(A,1),eq(B,1),same(A,B)
```

This says we should pick an object such that the two parts are not the same and neither equal 1.

So in general we need an object covered by the trial concept which does not satisfy anything in this elaboration difference.

Note that we might have to make a recursive application of the current trial to obtain a crucial object.

Note also the cut at the end of the predicate. We will only produce one crucial object.

---

covered_object(Trial Concept, Object)

This predicate says the object instantiated to the second argument is covered (satisfies) the trial concept.

The middle argument called Sofar is the part of the object that has been created so far during processing. It is required for processing `same(X,Y)` correctly.

Notice that there will no be repeated items in the object.

```
covered_object(Tr,Obj) :-
        covered_object(Tr,[],Obj1),
        norepeats(Obj1,Obj).

covered_object([],_,[]).
```

The builtin concept **same** poses problems because when it is used, its arguments had better be instantiated. To handle this problem we deal with **same** after the rest of the trial concept is satisfied.

```
covered_object([same(A,B)|Q],SoFar,Obj):-
        covered_object(Q,SoFar,Obj1),
        concat(SoFar,Obj1,Tot),
        member(eq(A,C),Tot),
        member(eq(B,D),Tot),
        C = D, % inst after member
        concat([eq(A,C),eq(B,D)],Obj1,Obj).
```

An item in the trial concept is atomic if there is a clause in the database for it without a body

```
covered_object([P],_,[P]) :-
        clause(P,true).
```

An item in the trial concept is not atomic if there is a clause in the database which has a body corresponding to it.

```
covered_object([P],SoFar,Obj)  :-
        not (P = same(_,_)),
        clause(P,Body),
        Body \== true,
        structure_to_list(Body,BL),
        ((not recursive(P,BL),
           covered_object(BL,SoFar,Obj));
         (recursive(P,BL),
           covered_object(BL,SoFar,Obj),
          !)). % only one per recursive application!
```

An item in the trial concept is atomic if there are no clauses in the database for it.

```
156        covered_object([P],_,[P]) :-
157                not clause(P,_).
```

If the first item in the trial concept has variables in it, we process it after we have processed the reset of the trial concept

```
158        covered_object([P|Q],SoFar,Obj) :-
159                [P|Q] \== [P],
160                not(P = same(_,_)),
161                ((novars(P),
162                  covered_object([P],[],Obj1),
163                  concat(SoFar,Obj1,SoFar1),
164                  covered_object(Q,SoFar1,Obj2));
165                 (not novars(P),
166                  covered_object(Q,[],Obj1),
167                  concat(SoFar,Obj1,SoFar1),
168                  covered_object([P],SoFar1,Obj2))),
169                concat(Obj1,Obj2,Obj).
```

---

infer(Conditions List, Object)

```
170        infer([],[]) :- !,fail.
171        infer([D|Dtail],Obj) :- infer1(D,Obj) ; infer(Dtail,Obj).

172        infer1(D,Obj) :- assert_trial(Obj),
173                 D,
174                 retract_trial(Obj).

175        infer1(D,Obj) :- retract_trial(Obj),!,fail.
```

This predicate says that at least one condition in the conditions list can be satisfied by, or inferred from, the object.

In this case the conditions list will be the elaboration difference.

References

[1] Kolokouris, A. T. *Machine Learning.* In Byte, November 1986, pp 225-231.

[2] Krawchuk, B.J. *How to Ask the Right Questions.* Dept of Comp. Sci. , University of Calgary, 1987.

[3] Sammut, C. *Learning Concepts by Performing Experiments* PhD. diss., Department of Computer Science, University of New South Wales, 1981.

[4] Sammut, C., and Banerji, R. B. *Hierarchical Memories: An Aid to Concept Learning.* In Proceedings of the International Machine Learning Workshop, R.S. Michalski (Ed.), Dept of Comp. Sci. , University of Illinois, Urbana, Ill, 1983.

[5] Sammut, C., and Banerji, R. B. *Learning Concepts by Asking Questions.* In Machine Learning: An Artificial Intelligence Approach, Vol. II, R.S. Michalski, J.G. Carbonell, and T.M. Mitchell (Eds.), Morgan Kaufmann, Los Altos, Calif., 1986.

A LIBRARY ROUTINES

A.1 Common Predicates

```
176     member(A, [A|L]).
177     member(A, [_|L]) :- member(A, L).

178     concat([], L, L).
179     concat([X|L1], L2, [X|L3]) :- concat(L1, L2, L3).

180     append(A, [], [A]).
181     append(A, [X|T], [X|T1]) :- append(A,T,T1).

182     reverse([],[]).
183     reverse([A|T],T1) :-
184             reverse(T,T2),
185             append(A,T2,T1).

186     delete(A, [A|L], L).
187     delete(A, [B|L],[B|L1]) :- delete(A,L,L1).

188     flattened([],[]).
189     flattened([X|Y],[X|Z]) :-
190             not list(X),
191             flattened(Y,Z).
192     flattened([X|Y],L) :-
193             list(X),
194             flattened(X,X1),
195             flattened(Y,Z),
196             concat(X1,Z,L).

197     union([],L,L).
198     union([A|A1],L,[A|L1]) :-
199             not member(A,L),
200             union(A1,L,L1).
201     union([A|A1],L,L1) :-
202             union(A1,L,L1).
```

```
intersect([],L1,[]).
intersect([A|L],L1,[A|L2]):-
        delete(A,L1,L3),
        intersect(L,L3,L2).
intersect([A|L],L1,L2) :-
        not member(A,L1),
        intersect(L,L1,L2).

norepeats(A,X) :-
        norepeats(A,A,X),!.
norepeats([],L1,L1).
norepeats([A|L],L1,Lf) :-
        delete(A,L1,L2),
        (( member(A,L2),
           norepeats([A|L],L2,Lf)) ;
          (not member(A,L2),
           norepeats(L,L1,Lf))).

delete_list([],A,A).
delete_list([A|L],L1,L2) :-
        delete(A,L1,L3),
        delete_list([A|L],L3,L2).    % delete all
delete_list([A|L],L1,L2) :-
        not member(A,L1),
        delete_list(L,L1,L2).

subtract_list([],A,A) :- !.
subtract_list([A|L],L1,L2) :-
        member(A,L1),
        delete_list([A],L1,L3), % delete all occurrences
        subtract_list(L,L3,L2),!.

list([]).
list([_|_]).
```

A.2 Less Common Predicates

assert_once asserts something into the data base only if it is not there already

```
233         assert_once(G) :- call(G), !.
234         assert_once(G) :- assertz(G).
```

structure_to_list the first argument is the equivalent to the second argument except for the fact that the second argument is a list and the first is in structure form.

```
235         structure_to_list(X, [X]) :-
236                 var(X).
237         structure_to_list((X,Y), [X|Z]) :-
238                 structure_to_list(Y, Z).
239         structure_to_list(X, [X]) :-
240                 not X=(_,_).
```

functorcopy the second argument is a duplication of the first argument but with different variables. This is useful if we need to temporalily instantiate something when we cannot backtrack.

```
241         functorcopy(In, Out) :-
242                 functorcopy(In, Out, _).

243         functorcopy(Atom, Atom, _)  :-
244                 atomic(Atom), !.
245         functorcopy(Var, Var1, Ids) :-
246                 var(Var), !,
247                 check_binding((Var,Var1), Ids).
248         functorcopy([], [], _)      :- !.
249         functorcopy([Arg|Rest], [Arg1|Rest1], Ids) :-
250                 !,
251                 functorcopy(Arg, Arg1, Ids),
252                 functorcopy(Rest, Rest1, Ids).
253         functorcopy(Functor, Functor1, Ids) :-
254                 Functor  =.. [Name|Args],
255                 functorcopy(Args, Args1, Ids),
256                 Functor1 =.. [Name|Args1].
```

check_binding used by functorcopy

```
257     check_binding(Thing, [End|_])       :-
258             var(End), !, Thing = End.
259     check_binding((A,A1), [(B,B1)|Rest]) :-
260             A == B, !, A1=B1.
261     check_binding(Thing, [_|Rest])      :-
262             check_binding(Thing, Rest).
```

novars succeeds if there are no uninstantiated variables in the object given

```
263     novars([A]) :-
264             var(A),
265             !,fail.
266     novars([A|T]) :-
267             not(var(A)),
268             atomic(A),
269             novars(T).
270     novars([A|T]) :-
271             not(var(A)),
272             not( atomic(A)),
273             A =.. [G|Args],
274             novars(Args),
275             novars(T).

276     novars(A) :-
277             atom(A).
278     novars(A) :-
279             not list(A),
280             A =..[G|Args],
281             novars(Args).
```

functornames the first argument is a list of things such as [cat(felix),dog(goofy),bird(tweety)]. The second variable is instantiated to a list of just the principal functors of the items in the first list. In this case: [cat,dog,bird].

```
282     functornames([],[]).
283     functornames([A|T],[A1|T1]) :-
284             A =.. [A1|_],
285             functornames(T,T1).
```

recursive the arguments here represent the head and the body of a Prolog Clause. A Prolog Clause is recursive if something in the body has the same principal functor as the principal functor of the head.

```
286         recursive(Head,BodyList) :-
287                 Head =.. [H|_],
288                 functornames(BodyList,Flist),
289                 member(H,Flist).
```

structure_intersect Instead of taking the intersection of two lists we intersect two structures, typically the body part of Prolog Clauses.

```
290         structure_intersect(S1,S2,Int) :-
291                 structure_to_list(S1,S1list),
292                 structure_to_list(S2,S2list),
293                 intersect(S1list,S2list,Int).
```

B MARVIN USER INTERFACE

B.1 MAIN PREDICATES

MAIN LOGIC

```
294     marvin:-init,
295             repeat,
296             prompt,
297             read1(X),
298             menu(X),
299             fail.

300     version('3.10').

301     init :- version(X),show([nl,'MARVIN v',write(X),nl,nl]).

302     prompt  :- show(['<marvin> ']).

303     good_bye :-
304             show(['Do you really want to quit? ']),
305             read1(X),
306             yes(X),
307             show([nl,'MARVIN SESSION COMPLETE',nl]),
308             halt.
309     good_bye.
```

THE MENU

```
310     menu_item(help)  :- give_help.
311     menu_item(active) :- active_concepts.
312     menu_item(bye) :- good_bye.
313     menu_item(deactivate) :- kill_concepts.
314     menu_item(activate) :- load_concepts.
315     menu_item(classify)   :- classify_objects.
316     menu_item(learn) :- learn_concepts.
317     menu_item(store) :- store_concepts.
318     menu_item(show) :- show_concepts.
```

```
319       menu(A) :- menu_item(A),!.
320       menu(A) :- alias(A,B), menu_item(B),!.
321       menu(A) :- show(['I don''t understand!',nl,
322         'Type help for help',nl]).
```

MENU ALIASES

```
323       alias(a,active).
324       alias(c,classify).
325       alias(halt,bye).
326       alias(load,activate).
327       alias(new,learn).
328       alias(u,activate).
329       alias(use,activate).
330       alias(q,bye).
331       alias(end,bye).
332       alias(b,bye).
333       alias(h,help).
334       alias(d,deactivate).
335       alias(kill,deactivate).
336       alias(l,learn).
337       alias(s,store).
338       alias(sh,show).
339       alias(display,show).
```

HELP

```
340       give_help :-
341               write('MARVIN MENU v1.0'), nl,nl,
342               write('KEYWORD                  '),
343               write('  MEANING'),nl,
344               write('-------                  '),
345               write('  ----------------------'),
346               write('--------------------------'),nl,
347               write('a,active                 '),
348               write('- display a list of the active concepts'),nl,
349               write('c,classify               '),
350               write('- classify concepts'),nl,
351               write('d,deactivate,kill        '),
352               write('- deactivate an active concept'),nl,
```

```
353             write('h,help                    '),
354             write('- display this message '),
355             write('(bet you didn''t know that)'),nl,
356             write('l,learn,new               '),
357             write('- learn a new concept '),
358             write('(or add a new disjunct to an old one)'),nl,
359             write('s,save,store              '),
360             write('- store a concept '),nl,
361             write('sh,show,display           '),
362             write('- show a concept''s internal representation'),nl,
363             write('u,use,activate,load       '),
364             write('- make a stored concept active'),nl,
365             write('b,bye,q,halt,end          '),
366             write('- end the MARVIN session'),nl.
```

ACTIVE CONCEPTS

```
367     active_concepts :- setof(A,active(A),X),
368         X \== [],
369         showlist(X),!.
370     active_concepts :- show(['No active concepts',nl]).

371     active(A) :- concept(A,_,_).
```

RETRIEVAL OF PREVIOUSLY LEARNED CONCEPTS

```
372     load_concepts   :-
373             show(['<activate> ']),
374             read1(X),
375             not no(X),
376             consult(X),
377             load_concepts.
378     load_concepts.
```

DEACTIVATION OF CONCEPTS

```
379     kill_concepts :-
380             show(['<deactivate> ']),
```

```
381             read1(X),
382             not no(X),
383             ((concept(X,_,Hd),
384             killconcept(Hd),
385             show([write(X),' deactivated',nl]),!);
386             (show([write(X),' is not active.',nl]))),
387             kill_concepts.
388     kill_concepts.

389     killconcept(Hd) :- clause(Hd,Body),retract((Hd:-Body)),fail.
390     killconcept(Hd) :- retract(concept(_,_,Hd)).
```

SAVING OF CONCEPTS FOR OTHER MARVIN SESSIONS

```
391     store_concepts :- show(['<store concept> ']),
392             read1(X),
393             not no(X),
394             ((concept(X,A,Hd),
395               show(['What File? ']),
396               read1(File),
397               not no(File),
398               tell(File),
399               write(concept(X,A,Hd)),
400               write('.'),nl,
401               write_things(Hd),
402               close(File),
403               tell(user),
404               show(['Wrote ',write(File),nl]),!);
405             (show([write(X),' is not active',nl]))),
406             store_concepts.
407     store_concepts.

408     write_things(Y) :-
409             clause(Y,Z),
410             write((Y:-Z)),
411             write('.'),nl,
412             fail.
413     write_things(_).
```

CLASSIFY OBJECTS

```
414     classify_objects:-
415             show(['<classifier> ']),
416             read1(Ex),
417             not no(Ex),
418             structure_to_list(Ex,ExL),
419             make_trial(0,ExL,Tr1),
420             expand_lists(Tr1,Tr),
421             show_trial(Tr),
422             show(['Possible Classifications are: ',nl]),
423             all_classifications(Tr),!,
424             classify_objects.

425     classify_objects.

426     all_classifications(Tr):-
427             assert_trial(Tr),
428             concept(_,_,Head),
429             Head,
430             indent,write(Head),nl,
431             fail.

432     all_classifications(Tr):-
433             retract_trial(Tr).
```

DISPLAY CONCEPTS

```
434     show_concepts  :-
435             show(['<show concept> ']),
436             read1(X),
437             not no(X),
438             ((concept(X,_,Hd),
439               showconcept(X,Hd),!);
440              (show([write(X),' is not active.',nl]))),
441             show_concepts.
442     show_concepts.

443     showconcept(X,Hd):-
444             show(['Something is a ',quote(X),' if ',nl]),
445             clause(Hd,Tail),
446             name_args((Hd:-Tail),(Hd:-Tail)),
```

```
447             structure_to_list(Tail,Tr),
448             showtheor,
449             show_trial(Tr),
450             fail.
451     showconcept(_,_):- (sh1, retract(sh1)); true.

452     showtheor :- sh1,     show(['or if',nl]).
453     showtheor :- not sh1, assert_once(sh1).
```

NEW CONCEPTS

```
454     learn_concepts :-
455             (( retract(current_concept(_,_)),! ); true ),
456             draw_line,
457             validconceptname(X),
458             new_concept(X),
459             learn_concepts.

460     learn_concepts.

461     validconceptname(X) :-
462             show(['<learn concept> ']),
463          read1(X1),
464             not no(X1),
465             ((concept(X1,_,_),
466              X = X1,!);
467              (current_predicate(X1,_),
468               show([write(X1),' is an internal predicate!',nl]),
469               validconceptname(X),!);
470              (not current_predicate(X1,_),
471               X = X1)).
472

473     new_concept(Concept) :-
474             concept(Concept,_,_),
475             get_examples(Concept), !.
476     new_concept(Concept) :-
477             not concept(Concept,_,_),
478             assertz(concept(Concept,Arity,Head)),
479             get_examples(Concept), !.
480     new_concept(Concept) :-
481      show_concept(Concept).
```

GET EXAMPLES FROM THE USER

```
482        get_examples(Concept) :-
483                arityof(Concept,Ar),
484               draw_line,
485                show(['Example of ', quote(Concept),'  ']),
486                show_args(Ar),
487                read1(Example), !,
488                not no(Example),
489                example(Concept, Example),
490                !, get_examples(Concept).

491        example(Concept, Example) :-
492                structure_to_list(Example, ExampleList),
493                make_trial0(Concept,ExampleList,T0),
494                (   old_example(Concept, T0)
495                  ; new_example(Concept, T0) ).
```

NEW EXAMPLES

```
496        new_example(Concept,Ex) :-
497                show_trial(Concept,Ex),
498                generalization_of_example(Ex,Gen),
499                draw_line,
500                write('The generalization = '),nl,
501                show_trial(Concept,Gen),
502                store_generalization(Concept,Gen).
```

INPUT PREPROCESSING

```
503        make_trial0(Concept,EL,Tr) :-
504                make_trial(0,EL,T0),
505                expand_lists(T0,Tr),
506                arityof(Concept,Ar),
507                make_vars(Ar,Vars),
508                G1 =.. [Concept|Vars],
509                retract( concept(Concept,Arity,_) ),
510                assertz( concept(Concept,Arity,G1) ),
511                assert_once(current_concept(Concept,G1)),!.
```

```
512     make_trial(_,[],[]) :-!.
513     make_trial(Acnt, [[A|B]|Atail],
514                      [eq(head(V),A),eq(tail(V),B)|Tail]) :-

515             (atom(A) ; number(A)),!,
516             N2 is Acnt + 65,
517             name(V,[N2]),
518             Acnt1 is Acnt + 1,
519             make_trial(Acnt1,Atail,Tail).

520     make_trial(Acnt,[A|Atail],[eq(V,A)|Tail]) :-
521             (atom(A) ; number(A)),!,
522             N2 is Acnt + 65,
523             name(V,[N2]),
524             Acnt1 is Acnt + 1,
525             make_trial(Acnt1,Atail,Tail).
526     make_trial(Acnt,[A|Atail],[A|Tail]) :-
527             make_trial(Acnt,Atail,Tail).

528     expand_lists([],[]).
529     expand_lists([eq(tail(A),[Lhead|Ltail])|T],
530                               [eq(head(tail(A)),Lhead)|T1]) :-

531             expand_lists([eq(tail(tail(A)),Ltail)|T],T1).
532     expand_lists([A|T],[A|T1]) :- expand_lists(T,T1).

533     make_vars(0,[]) :- !.
534     make_vars(N,[A|Atail]) :- N1 is N-1, make_vars(N1,Atail).
```

STORE CONCEPT (a single disjunct) IN MEMORY

```
535     store_generalization(Concept,Gen):-
536             show(['Saving a disjunct of the concept (',quote(Concept),')',nl]),
537             make_gen_clause(Concept,Gen,CL),
538             assertz( CL ).

539     make_gen_clause(Concept,Gen,(C1:-C2)):-
540             arityof(Concept,Ar),
541             varnames(Ar,Varnames),
542             make_vars(Ar,Vars),
543             getothercapvars(Gen,Varnames,Exnames,Num),
```

```
544             make_vars(Num,ExVars),
545             concat(Varnames,Exnames,Allnames),
546             concat(Vars,ExVars,AllVars),
547             replacevars(Allnames,AllVars,Gen,NewGen),
548             C1 =.. [Concept|Vars],
549             structure_to_list(C2,NewGen).

550     getothercapvars([],_,[],0).
551     getothercapvars([Thing|Gen],VN,CN,Num) :-
552             getothercapvars(Gen,VN,CN1,Num1),
553             getvarsfrom(Thing,VN,CN1,CN2,Num2),
554             concat(CN2,CN1,CN),
555             Num is Num1 + Num2.

556     getvarsfrom([],VN,CN1,[],0).

557     getvarsfrom([A|T],VN,CN1,[A|CN2],Num2) :-
558             atom(A),
559             capitalized(A),
560             not member(A,VN),
561             not member(A,CN1),
562             getvarsfrom(T,VN,CN1,CN2,Num3),
563             Num2 is 1 + Num3.

564     getvarsfrom([A|T],VN,CN1,CN2,Num2) :-
565             atom(A),
566             capitalized(A),
567             ( member(A,VN);
568              member(A,CN1) ),
569             getvarsfrom(T,VN,CN1,CN2,Num2).

570     getvarsfrom([A|T],VN,CN1,CN2,Num2) :-
571             (atom(A); number(A)),
572             not capitalized(A),
573             getvarsfrom(T,VN,CN1,CN2,Num2).

574     getvarsfrom([A|T],VN,CN1,CN3,Num3) :-
575             not atom(A),
576             not number(A),
577             getvarsfrom(A,VN,CN1,CN2,Num2),
578             getvarsfrom(T,VN,CN2,CN3,Num3).

579     getvarsfrom(Thing,VN,CN1,CN2,Num2) :-
580             not atom(Thing),
```

```
581             not list(Thing),
582             Thing =.. [T|Args],
583             getvarsfrom(Args,VN,CN1,CN2,Num2).


584     capital(Ascii) :- Ascii < 91, Ascii > 64.
585     capitalized(Name) :- name(Name,[A|_]), capital(A).


586     replacevars([],[],Gen,Gen).
587     replacevars([Name|Ntail],[V|Vtail],Gen,NewGen):-
588             replacevar(Name,V,Gen,TmpGen),
589             replacevars(Ntail,Vtail,TmpGen,NewGen).

590     replacevar(Name,V,[],[]).
591     replacevar(Name,V,[G|Gen],[New|NewGen]) :-
592             G =.. [F|Args],
593             replvar(Name,V,Args,NewArgs),
594             New =.. [F|NewArgs],
595             replacevar(Name,V,Gen,NewGen).
596
597     replvar(Name,V,[],[]).
598     replvar(Name,V,[A|Args],[NA|NewArgs]):-
599             (A == head(Name); A == tail(Name)),
600                 A =.. [A1,Name],
601                 NA =.. [A1,V],
602                 replvar(Name,V,Args,NewArgs).
603     replvar(Name,V,[A|Args],[NA|NewArgs]):-
604             ((A == Name,    NA=V);
605              (not(A==Name), NA=A)),
606             replvar(Name,V,Args,NewArgs).
607
```

OTHER PREDICATES

```
608     name_args((C:-E), (C1:-E1)) :-
609             name_args((C:-E), _, _, (C1:-E1)).
610     name_args((C:-E), Concept, Args, (C1:-E1)) :-
611             functorcopy((C:-E), (C1:-E1)),
612             functor(C,_,NumArgs),
613             varnames(NumArgs,Args),
```

```
614             C1=..[Concept|Args].

615     varnames(N,Args) :- varnames(N,0,Args).
616     varnames(N,N,[]) :- !.
617     varnames(N,N1,[A|Atail]) :-
618             N2 is N1 + 65,
619             name(A,[N2]),
620             N3 is N1 + 1,
621             varnames(N,N3,Atail).

622     arityof(Concept,Ar) :-
623             concept(Concept,Ar,_),
624             not var(Ar), !.
625     arityof(Concept,Ar) :-
626             show(['Arity: ']),
627             read1(Ar),
628             not no(Ar),
629             integer(Ar),
630             retract( concept(Concept,Ar,X) ),
631             assertz( concept(Concept,Ar,X) ).
```

OLD EXAMPLE CHECKING

```
632     old_example(Concept, T0) :-
633             assert_trial(T0),
634             example_of(Concept,_),
635             show(['The example is already an instance of ',
636                             quote(Concept), nl]),
637             retract_trial(T0).

638     old_example(Concept, T0) :-
639             retract_trial(T0),!,fail.

640     example_of(Concept,Goal) :-
641             concept(Concept,Ar,Head),
642             name_args((Head:-true),(Goal:-true)),
643             Goal.

644     assert_trial(Ex) :- member(M,Ex), asserta(M), fail.
645     assert_trial(Ex).
```

```
646         retract_trial(Ex) :- member(M,Ex), retract(M), fail.
647         retract_trial(Ex).
```

B.2 INPUT PREDICATES

```
read1(X) :- myread(Y), read1(X, Y),!.
read1(_, X) :- abort(X), abort.
read1(X, spy(S)) :- spy(S), !, read1(X).
read1(X, nodebug) :- nodebug, !, read1(X).
read1(X, X).


myread(X) :- myread1(X1), name(X2,X1), atom_to_structure(X2,X),!.
myread1([N|X]) :- get(N), N \== 46, myread1(X).
myread1([])    :- get0(_).  % CR


atom_to_structure(Atom,Struct) :-
 name(Atom,Alist),
 cl(StructList,Alist,[]),
 structure_to_list(Struct,StructList).


cl([T|SL])  --> term(T), comma, cl(SL).
cl([T]) --> term(T).

term(T)  --> listterm(T).
term(T)  --> termhead(H),args(A),{name(H1,H),T=..[H1|A]}.

listterm(X) --> opensquarebracket, closesquarebracket ,
                    { X =.. [[]] }.
listterm(H)  --> opensquarebracket, termlist(H), closesquarebracket.

termlist([H|T]) --> term(H), comma, termlist(T).
termlist([H])  --> term(H).

termhead([H|T]) --> alphanumeric(H), termhead(T).
termhead([H|T]) --> pathnamechar(H), termhead(T).
termhead([],A,A).

args(A) --> openroundbracket, arglist(A), closeroundbracket.
args([],A,A).

arglist([H|T]) --> term(H), comma, arglist(T).
arglist([A]) --> term(A).
```

```
676     opensquarebracket  --> [91].
677     closesquarebracket  --> [93].
678     openroundbracket --> [40].
679     closeroundbracket --> [41].
680     comma --> [44].

681     pathnamechar(47) --> [47]. % '/'
682     pathnamechar(126) --> [126]. % '~'

683     alphanumeric(A) -->  alphabetical(A).
684     alphanumeric(A) -->  numeric(A).
685     alphabetical(A) -->  capital(A).
686     alphabetical(A) -->  locase(A).


687     numeric(A,[A|T],T) :- A < 58, A > 47.
688     capital(A,[A|T],T) :- A < 91, A > 64.
689     locase(A,[A|T],T)  :-  A > 96, A < 124.
```

OTHER INPUT-RELATED PREDICATES

```
690     yes_no :- read(X), yes_no(X).

691     yes_no(A) :- abort(A), abort.
692     yes_no(Y) :- yes(Y), !.
693     yes_no(N) :- no(N), !, fail.
694     yes_no(X) :- write('please answer yes or no (or abort)'),
695                  nl, yes_no.

696     yes(yes).
697     yes(y).

698     no(no).
699     no(n).

700     abort(abort).
701     abort(ab).
702     abort(quit).
```

B.3 OUTPUT PREDICATES

```
703     show([])     :- !.
704     show(do(A))  :- !, call(A).
705     show(write(A))  :- !, write(A).
706     show(quote(A)) :- !, write(''''), write(A), write('''').
707     show(list(A)) :- !, show_list(A).
708     show(nl)     :- !, nl.
709     show([A|B])  :- !, show(A), show(B).
710     show(A)      :- write(A).

711     show_list([]) :- !.
712     show_list([A]) :- !, show(A).
713     show_list([A|L]) :- !, show(A), write(','), show_list(L).
714     show_list(A) :- show(A).


715     show_args(N) :- show_args(N,0).
716     show_args(N,N) :- write('): '),!.
717     show_args(N,0) :- write('('),alpha(0),show_args(N,1).
718     show_args(N,N1):- write(','),alpha(N1),N2 is N1+1,show_args(N,N2).

719     alpha(N) :- N1 is N + 65,name(X,[N1]),write(X).


720     sorted_trial([],[]).
721     sorted_trial([A|T],S):-
722         sorted_trial(T,ST),
723         insertit(A,ST,S).
724     insertit(A,[B|S],[B|S1]) :-
725         ic(B,A),!,
726         insertit(A,S,S1).
727     insertit(A,S,[A|S]).

728     ic(A,B) :- varof(A,_,Asc), varof(B,_,Asc1),
729             ((Asc < Asc1);
730              (Asc = Asc1,
731               ia(A,B))).

732     ia(A,B) :-
733             A =.. [_|[A2|_]],
```

```
734             B =.. [_|[B2|_]],
735             io(A2,B2).

736     io(A,A).
737     io(head(A),tail(B)) :- io(A,B).
738     io(head(A),head(B)) :- io(A,B).
739     io(A,B) :- atom(A), not atom(B).
```

varof(structure,nesting,varname,asciiofvar)

```
740     varof(S,S,Ascii) :- atom(S), name(S,[Ascii]).
741     varof(head(S),S1,Ascii) :- varof(S,S1,Ascii).
742     varof(tail(S),S1,Ascii) :- varof(S,S1,Ascii).
743     varof(S,S1,Ascii) :- S=.. [_|[H|_]], varof(H,S1,Ascii).

744     show_trial(Concept,Tr) :-
745             sorted_trial(Tr,SortedTr),
746             concept(Concept,Ar,G1),
747             show_clause_list(G1,SortedTr),!.

748     show_trial(Tr) :-
749             sorted_trial(Tr,Tr2),
750             structure_to_list(Tr1,Tr2),
751             show_clause(8,Tr1),!.

752     show_object([]).
753     show_object([O|Otail]) :- show_clause(8,O),show_object(Otail).

754     show_clause_list(Head, List) :-
755             structure_to_list(Body, List),
756             show_clause((Head:-Body)).

757     show_clause((C:-E)) :-
758             name_args((C:-E), Concept, Args, (C1:-E1)),
759             structure_to_list(Args2,Args),
760             show([do(write(Args2)), ' is a ',
761                         quote(Concept), ' if:', nl]),
762             show_clause(8, E1),!.

763     show_clause(_,[]).
764     show_clause(Indent, (A,B)) :- A = (eq(head(_),_)),
765            listable((A,B)),tab(Indent),
766            show_list(start,(A,B),C),show_clause(Indent,C).
```

```
show_clause(Indent, (A,B)) :-
        show_clause(Indent, A), show_clause(Indent, B).
show_clause(Indent, A) :-
        not A=(_,_), tab(Indent), show_clause(A).

show_clause(eq(X,B)) :-
        show_thing(X),
        show([' is ', do(write(B)), nl]),!.
show_clause(isa(X,B)) :-
        show_thing(X),
        show([' is an instance of ', quote(B), nl]),!.
show_clause(same(X, Y)) :-
        show_thing(X),
        show([' is the same as ']), show_thing(Y), nl,!.
show_clause(A) :- write(A),nl.


listable(A) :- structure_to_list(A,L), listable1(L).
listable1([]).
listable1([eq(tail(_),_)|_]).
listable1([eq(head(_),_)|B]) :- listable1(B).

show_list(start,(eq(head(A),B),C),D) :- varof(A,V,_), show_thing(V),
   show([' is [',do(write(B))]),
            show_list(middle,C,D),!.
show_list(middle,(eq(head(A),B),C),D) :- show([',',do(write(B))]),
   show_list(middle,C,D),!.
show_list(middle,(eq(tail(A),[]),C),C) :- write(']'),nl,!.
show_list(middle,eq(tail(A),[]),[]) :- write(']'),nl,!.

show_thing(head(X)) :- show(['the head of ']), show_thing(X).
show_thing(tail(X)) :- show(['the tail of ']), show_thing(X).
show_thing(X) :- atomic(X), write(X).

show_concept(Concept) :-
        concept(Concept,Ar,C),
        not var(C),
        clause(C, E),
        show_clause((C:-E)),
        fail.
show_concept(_).

draw_line :-
  show(['----------------------------------------',nl]).
```

```
804     showlist([]).
805     showlist([A|T]) :- indent,write(A),nl, showlist(T).

806     indent   :- tab(8).

807     mytab(0) :- !.
808     mytab(N) :- write(' '), N1 is N - 1, mytab(N1).
```