Learning Concepts by Performing Experiments

Claude Sammut

Department of Computer Science
University of New South Wales

November, 1981

Abstract

Marvin is a program which is capable of learning concepts from many different environments. It achieves this by using a flexible description language based on first order predicate logic with quantifiers. Once a concept has been learnt, Marvin treats the concept description as a program which can be executed to produce an output. Thus the learning system can also be viewed as an automatic program synthesizer.

The ability to treat a concept as a program permits the learning system to construct objects to show a human trainer. Given an initial example by the trainer, Marvin creates a concept intended to describe the class of objects containing the example. The validity of the description is tested when Marvin constructs an instance of the concept to show the trainer. If he indicates that the example constructed by the program belongs to the concept which is to be learnt, called the 'target', then Marvin attempts to generalize the description of its hypothesized concept. If the example does not belong to the target then the description must be made more specific so that a correct example can be constructed. This process is repeated until the description of the concept cannot be generalized without producing unacceptable examples.

Marvin has an associative memory which enables it to match the descriptions of objects it is shown with concepts that it has stored in memory. Complex concepts are learnt by first learning the descriptions of simple concepts which provide Marvin with the knowledge necessary to learn the more complex ones.

A concept may represent a non-deterministic program, that is, more than one output may result from the same input. Not all the possible outputs of a concept are acceptable as training instances. Thus, Marvin must have an 'instance selector' which is capable is choosing the best objects to show the trainer.

Marvin has been tested on a number of learning tasks. Extensive performance measurements were made during these sessions with the program. The results indicate that Marvin is capable of learning complex concepts quite quickly.

Acknowledgements

This project is the result of a stimulating and enjoyable collaboration with Dr. Brian Cohen. Although the work reported here is my own, the inspiration which started it all was Brian's. His continued support and the many friendly arguments we have had helped my greatly.

Thanks also to Dr. Graham McMahon for his supervision and reading of the thesis. Professor Murray Allen is the head of a computer science department whose staff and students have always been prepared to offer assistance. They have made my time at UNSW very enjoyable.

The history of this work begins with Professor Ranan Banerji who developed many of the ideas which inspired both Brian Cohen and myself. He has taken a personal interest in our work and our welfare. For that, I am very grateful.

David Powers and Yves Kodratoff read parts of the thesis. They pointed out a number of weaknesses in my explanations, so if the text is any clearer, it is due to their help.

Most of the diagrams in the thesis were originally prepared using graphics packages written by David Milway, and Richard Grevis. I must also thank Kevin Hill, Peter Ivanov, Perry Brown and Robert Hegedus for their patience in answering many silly questions when Marvin and this thesis were being written.

Finally, I would like to thank my family for their constant love and support over the years.

Table of Contents

1
Generalisations and Experiments

Marvin is a young child playing with some toy blocks in his nursery. He sees his mother building objects with the blocks. Trying to imitate the adult, Marvin makes an attempt at building something himself, say, a stack. The first try fails because he has done something wrong. But eventually, he will be successful as he learns from his mistakes.

By playing with the blocks, Marvin learns about the attributes of blocks which enable him to build stacks. He learns to form a category of objects called 'stacks', so the next time he wants to build a stack, he will not need to make the same mistakes he did the first time. Categories formed this way are called concepts.

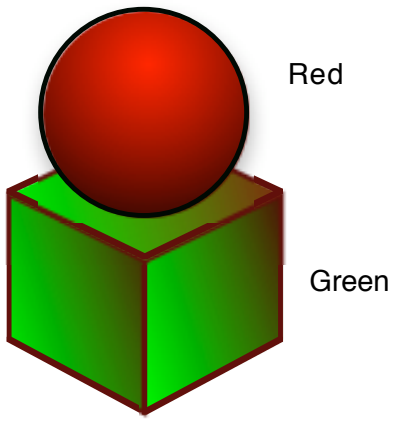
Science has formalized Marvin's trial-and-error method of learning into a process called 'The Scientific Method'. Using this method, a scientist observes a natural phenomenon and forms a theory which is an attempt to explain the observed event. In formulating the theory, the scientist may have drawn on his past experience and knowledge of the world. Once the theory has been formed, it must be tested by performing a carefully designed experiment. Its outcome may confirm the hypothesis or disprove it. If disproved, the scientist must modify his theory and develop a new experiment to test it.

This thesis describes a program which uses this approach to learning concepts. Having seen a particular instance of a concept, the program develops a hypothesis for its description by trying to relate the events it observes to concepts that it has learned before and are stored in its memory. A hypothesis is tested by performing an experiment. That is, the hypothesis is used to construct what the program thinks is a new instance of the concept. If the object really is a correct instance then the program requires some feedback from the environment to tell it so. In the present system, a human trainer supervises the program and answers yes or no to the program's attempt. This is something like playing twenty questions with a computer.

1.1 In the Nursery

As an introduction to the way in which the learning system works, let us return to Marvin and his toy blocks. Since he does not yet have a good understanding of how physical objects interact, when he tries to imitate his mother, who built a stack, he may make some mistakes before he succeeds. For example, he may not understand that the base object must be flat in order to support another block on top. Let's follow Marvin's learning sequence:

Suppose Marvin sees a red ball on top of a green cube.



At his young age, Marvin may not realize it, but to understand the scene, he must have some description in mind of what *on top of* means. Usually, such a description consists of specifying the values of certain attributes, or properties of the objects. A description of this scene may be:

The scene consists of a top object supported by a bottom object.
The shape of the top is a sphere.
The colour of the top is red.
The shape of the bottom is a box.
The colour of the bottom is green.

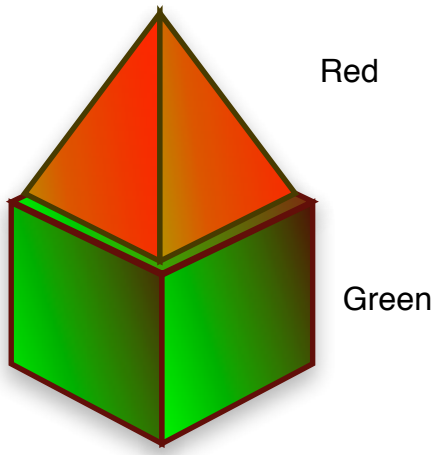
The problem faced by Marvin is: What does this instance of *on top of* have in common with other instances? Once he knows this, he will have some test which will enable him to form a category of scenes which can be labelled *on top of*.

In order to discover something about the objects he is seeing, Marvin may try to associate elements of the scene with concepts he knows already. Let's assume that he knows about the different colours and shapes that objects can have.

The second statement in the description above refers to the shape of the top object. Since Marvin knows about different shapes, he can try his first experiment. 'If I change the shape of the top object, is the new scene still an instance of *on top of*?' Marvin's new hypothesis for the description of the concept is:

The scene consists of a top object supported by a bottom object.
The shape of the top is any shape.
The colour of the top is red.
The shape of the bottom is a box.
The colour of the bottom is green.

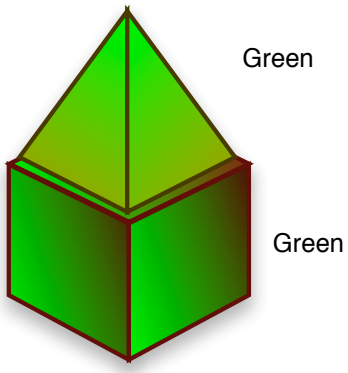
Now Marvin must find out if his generalization is correct. That is, can the top really be any shape at all? He can find out by trying to place, say, a red pyramid, on top of a green block.



In response to this action, Marvin's mother may smile and confirm that he has done the right thing. Flushed with success, Marvin proceeds to generalize more of the description. Now he tries to generalize the colour of the top object:

The scene consists of a top object supported by a bottom object.
The shape of the top is any shape.
The colour of the top is any colour.
The shape of the bottom is a box.
The colour of the bottom is green.

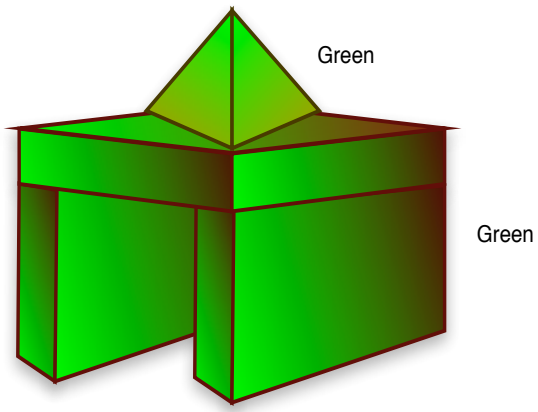
This is tested by constructing the object:



All the possibilities for the top object have been exhausted, so the bottom can be examined next. Marvin notices that the top of the bottom object is flat, so it is reasonable to ask, will any flat object do? A new hypothesis for the concept description is formed,

> The scene consists of a top object supported by a bottom object.
> The shape of the top is any shape.
> The colour of the top is any colour.
> The shape of the bottom is a any flat shape.
> The colour of the bottom is green.

This can be tested by replacing the block on the bottom with a table.



In the same way, the colour of the bottom is generalized by allowing it to be any colour.

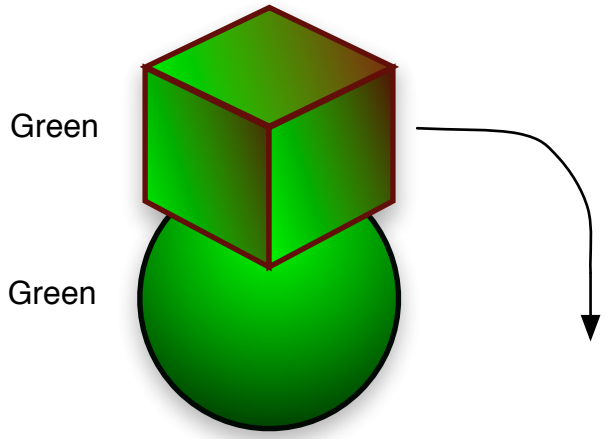
> The scene consists of a top object supported by a bottom object.
> The shape of the top is any shape.
> The colour of the top is any colour.
> The shape of the bottom is any flat shape.
> The colour of the bottom is any colour.

This change is tested by making the table on the bottom green.

Now Marvin has generalized all the values of the attributes of the objects. Is it possible to generalize even more? It is reasonable to ask, 'if the bottom can be any flat object, can it be any shape at all?'

> The scene consists of a top object supported by a bottom object.
> The shape of the top is any shape.
> The colour of the top is any colour.
> The shape of the bottom is any shape.
> The colour of the bottom is any colour.

The object he tries to construct should have a base which does not have a flat top.



Marvin has gone too far in generalizing the shape of the bottom object, since the pyramid falls off the ball. He must backtrack to his previous correct hypothesis. Since there is nothing left to try, the generalizations end here. Marvin's final concept of *on top of* is:

> The scene consists of a top object supported by a bottom object.
> The shape of the top is any shape.
> The colour of the top is any colour.
> The shape of the bottom is any flat shape.
> The colour of the bottom is any colour.

This example is not entirely realistic because Marvin was more systematic than a young child would normally be. When he wanted to make a generalization, Marvin was very conservative, changing only one property at a time. Using a conservative strategy, if a generalization fails, it is clear that the range of values of the property cannot be enlarged. However, if more than one property is changed, then Marvin would not know which property value had been over-generalized.

The 'Conservative Focusing Strategy' was first described by Bruner *et al* (1956) as part of a study of human concept formation. Since then this work has inspired a number of computer models of knowledge acquisition.

1.2 A Computer Program that Performs Experiments

As well as being the name of a clever child, Marvin is the name of a computer program which is capable of learning concepts by performing experiments in much the same way as its namesake.

The program's task is to create a description of a concept based on an example shown by the human trainer. Like the child, the program starts with a very specific description which includes only the training instance. It generalizes this description by using knowledge that it has learned before, just as Marvin used his knowledge of colours and shapes.

The learning process can be characterized as follows: A concept description describes a set of objects. The initial description specifies a set consisting of only one object, the training instance. By generalizing the concept, we describe a new set which includes new objects as well as the objects in the old set. That is, we make the category of objects broader.

Figure 1.1 illustrates this process. The *target* is the concept which the trainer wants the program to learn. Learning means expanding the initial set until it contains all the objects in the target. But as we saw when Marvin tried to place a pyramid on top of a sphere, it is possible to create a description which includes unwanted objects. This kind of generalization is said to be *inconsistent*. In order to discover if a generalization is inconsistent, the program performs an *experiment*. Like Marvin, it constructs on object to show the trainer.

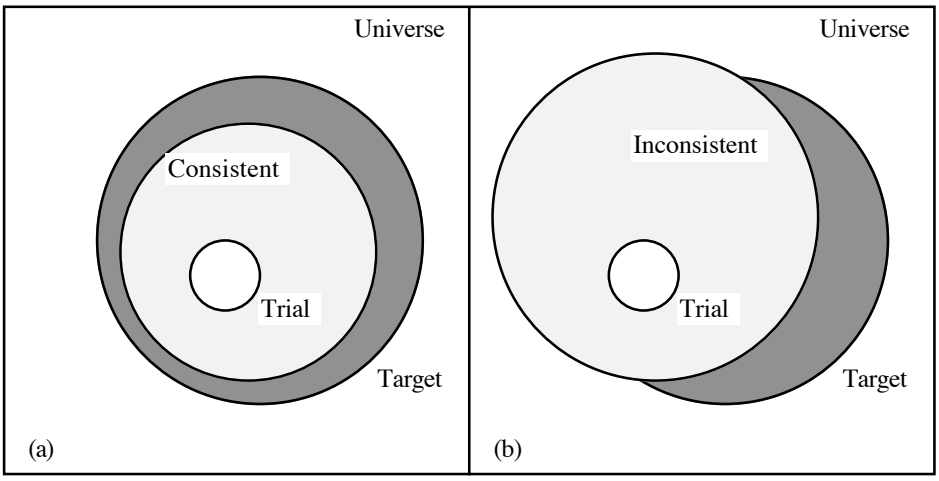


Figure 1.1. Generalizing concepts

Some care must be taken when choosing an object to show. Remember when Marvin wanted to test that the bottom could be any shape? He tried to construct an object whose base was *not* flat. In other words, if an incorrect generalization is made, the object shown must belong to the shaded region of Figure 1.1b, that is, contained in the hypothesis but not in the target. Finding such an object is quite a difficult task and will be dealt with fully in this thesis.

When an inconsistent generalization is made, the program tries to modify its description of the concept so that it contains fewer objects than the hypothesis that failed. If the new description turns out to be consistent then the program can try to generalize it. So we can think of the learning procedure as oscillating around the correct concept description, coming closer and closer until the target is reached.

1.3 Concept Description Languages

Just as Marvin, the child, has some representation of a scene in his mind, Marvin, the program, must also have some means of representing concepts.

The descriptions in English of pyramids and blocks were 'structural' representations of the observed event. Each object was described by specifying the values of properties such as colour and shape or whether one object was supported by another. These values can be considered as the results of measurements performed by the senses. The eye can detect differences in hue, find boundaries and report positional information. In the case of a computer, we must assume that it has cameras and range-finders attached to give it this information.

One of the problems encountered in pattern recognition is trying to decide what sensory information is sufficient to be able to distinguish objects in the universe. If too few measurements are made, perhaps we will not have enough information. On the other hand, if too many measurements are made, there is the possibility of being swamped by too much information.

If a large number of measurements are made, it may be possible to detect patterns in the data. By attaching a name to that pattern, we may simplify the description of a concept. For example, if Marvin had not known that blocks and tables are flat objects, his description of *on top of* may have included a statement such as 'The bottom is a block or the bottom is a table or the bottom is a ...' Instead the same idea can be simply expressed as 'The bottom is flat'.

A recognition system that can use learned concepts in this way is said to be capable of growth (Banerji, 1977), since the descriptive power of the system increases as it learns more concepts.

One of the main design goals of the program, Marvin, was that it should be capable of growth. It uses a language based on first order predicate logic to describe concepts. When a concept such as *flat* has been learned, its description is remembered so that it may be used in the descriptions of other concepts.

1.4 Concept Descriptions and Programming Languages

When a person is asked to write a program for a computer, he often told by his client: 'I want to get this kind of information out of the program, given these data as input.' The first thing that the programmer must do is create some kind of high-level description of what the program is supposed to do. He must understand, form a concept, of the relationship between the input data and the output.

In fact, when we write programs in Prolog (Roussel, 1972), we are writing the relationship between input and output in the form of a statement in first order predicate logic. A Prolog program consists of a set of clauses of the form P :- Q, R, S. This is read as 'P is true if Q and R and S are true.' For example, the program to append two lists producing a third, in Prolog is:

```
append([], X, X).
append([A | B], X, [A | B1]) :- append(B, X, B1).
```

This states that the result of appending any list, X, to the empty list is X itself. The result of appending X to a list whose head is A and whose tail is B is the list whose head is also A and whose tail is B1, obtained by appending B and X.

This program is a group of predicates which describe the relationship between the input lists and the output. When interpreted by a theorem prover, the output can actually be constructed if the input lists are given. So a description language can also be a programming language.

In the same way, Marvin's concept description language can be considered a programming language. When a concept is learned, Marvin is not only able to recognize an object which belongs to that concept. It can also execute the description to construct an instance of the concept. This is the method used to construct training examples to show the trainer during learning.

The program is not limited to learning simple concepts such as *on top of*. It can learn to sort lists, find the maximum number in a list or parse sentences of a simple context-free grammar. Although not designed as an automatic programming system, Marvin is capable of generating programs that compare with those of special purpose systems which use input/output pairs to describe the program. Figure 1.2 shows how Marvin may be used to synthesize programs.

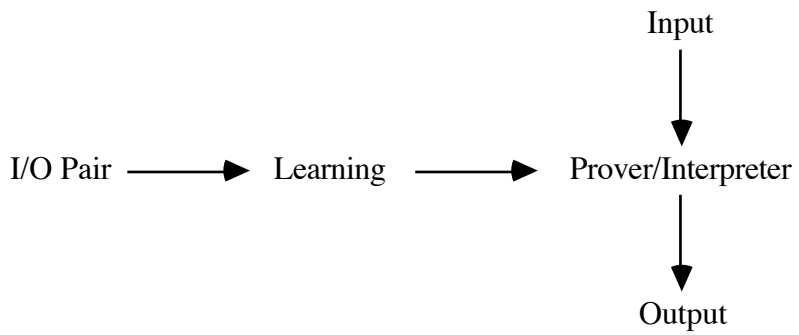


Figure 1.2. Schematic diagram of system

1.5 Objectives of the Research

In this section, we will list the objectives of the research to be described. It was intended to develop a concept learning program which has the following characteristics.

- The program should be capable of learning concepts from many different domains. It would achieve this by using a flexible description language based on first order predicate logic with quantifiers.
- Once a concept has been learned, the learning system should be able to treat the concept description as a program and execute it. Thus the learning program may also be an automatic programming system.
- The ability to treat concepts as programs permits the learning system to construct objects to show to the trainer. The system has greater control over its search for a suitable concept description since it can propose its own training examples, independently of the trainer. This also relieves the trainer of

additional work and provides a more convenient and understandable way of interacting with the program.

- The learning algorithm should be able to use its past experience (i.e. concepts it has learned before) to guide the formation of new concepts. Complex concepts may be learned in stages beginning with easy ones and building up to the more difficult concepts.

Marvin is significant in a number of respects. To my knowledge, no other general purpose learning system is capable of generating its own training instance. However, some special purpose learning programs have been devised, for example, Popplestone's (1967) noughts-and-crosses program. Some natural language acquisition systems also learn by trying to 'say' a meaningful sentence. Special purpose systems have some in-built model, even if elementary, to use in constructing examples. None of these is as flexible as Marvin.

It will be seen that Marvin can learn complex concepts whose descriptions involve existential quantifiers. The descriptions may be disjunctive and recursive. This enables it to learn programming concepts and simple language concepts.

Marvin's learning algorithm can use the concepts already known to guide the search for the target concept description. This was also a goal of Cohen's CONFUCIUS (Cohen, 1978). Marvin is the successor of CONFUCIUS and carries this aspect of the research further.

1.6 Structure of the Thesis

Chapter 2 provides an introduction to current research in concept learning. It discusses a number of the problems encountered and how various researchers have attempted to solve them.

Chapter 3 gives an overview of the entire system. It contains a formal description of Marvin's language and the algorithms it uses.

Chapter 4 contains an extended example of Marvin at work on a difficult learning task.

Chapter 5 describes the implementation of the program in detail.

Chapter 6 describes the results obtained from performance measurements on the program.

Chapter 7 suggests ways of improving Marvin and points out directions for future research.

Chapter 8 is the conclusion.

2
Introduction to Concept Learning

This chapter contains an informal discussion of some current research in concept learning. It is not intended to be a comparative study, since several such works already exist (Banerji, 1977; Banerji and Mitchell, 1980; Dietterich and Michalski, 1977; Smith, Mitchell, Chestek and Buchanan, 1977). There are a number of programs in existence today which are broadly classed as 'concept learning programs'. There are significant differences among them since they were each designed to meet different goals. We will examine the decisions which a designer must make in attempting to achieve those goals.

First let us define, in general, the task which a concept learning program is to perform.

2.1 The Problem

Suppose there is a universe of objects. The objects are as yet unspecified, but they may be physical objects such as tables and chairs or more abstract ones like numbers and lists. When names such as 'table' are used to refer to an object, then that object is classified as belonging to a specific subset of the universe. In object recognition, an observer applies a previously established rule in order to decide to which class an object belongs. That is, the observer has some method for determining what is a table and what is not.

The problem of concept learning is: Given samples of objects of known class membership, the observer must develop the classifying rule.

The practical importance of rule induction has become clear in recent years with the development of knowledge-based expert systems. These are programs which have achieved expert status in a specific domain, such as medical diagnosis or symbolic algebra. Such programs are are difficult to write, particularly as the programmer requires the cooperation of a human expert to develop the rules which guide the problem solver. Very often the human expert cannot describe his own problem solving process. Thus programs which are capable of learning to solve problems have proved very valuable. An example of this is Meta-DENDRAL (Buchanan and Feigenbaum, 1978) which is capable of learning to interpret the results of mass-spectrograms and nuclear magnetic resonance tests.

A more long-term goal of this research is to try to understand the learning process in general. Sometimes this is linked to a study of human learning abilities. However even programs which are not restricted to a specific domain, as DENDRAL is, may be designed without reference to human behaviour.

The classifying rules which describe a concept must must have some representation in the computer. Thus the first decision which the designer must face is, how should a concept be represented?

2.2 Choosing a Representation

The basic properties, which the learning machine's sensors can measure, and their inter-relationships constitute the *language* of the machine. There are two ways of approaching language design. If the learning system is intended to work in a specific domain then the choice of a language to represent the concepts is dictated by the type of object that belongs to the domain. A good example of this is Meta-DENDRAL again.

2.2.1 Special Purpose Languages

Meta-DENDRAL (Buchanan and Feigenbaum, 1978) was designed to form rules that characterize the bonds of molecules which break when the molecules are placed in a mass spectrometer. Molecules are represented by graphs. The nodes of the graph are non-hydrogen atoms. Arcs between the nodes represent the bonds between the atoms. Each rule describes a substructure in which certain bonds are distinguished. If the substructure occurs in a molecule, then the corresponding bonds are predicted to break in the mass spectrometer.

Node	Atom Type	Non-Hydrogen Neighbours	Hydrogen neighbours	Unsaturated Electron
1	C	(2 x _ _)	any number	0
2	N	(1 x)	1	0

Bonds that break: bond between atoms 1 and 2.

Figure 2.1. A Typical Meta-DENDRAL Rule

The rule shown if Figure 2.1 indicates that where there is a carbon atom with any number of hydrogen atoms, attached to a Nitrogen atom with one Hydrogen atom, then the bond between the Carbon atom and the Nitrogen atom will break.

It can be seen that the structure of the language reflects the structure of the objects in the domain. However, if the program is intended to operate in a range of environments, then there cannot be such a direct correspondence. The language must be flexible enough to describe very different kinds of objects. A number of general purpose concept learning systems have used languages based on first order predicate logic.

2.2.2 General Purpose Languages - Predicate logic

A simple predicate is an expression like colour(top,red). The names of constants such as 'top' and 'red' are *parameters*. Colour(top,red) is called an *instantiated form* of the *variable form* colour(X,Y) (Hayes-Roth, 1977). X and Y are variables which may represent any constant. In an expression such as,

[X: colour(X, red)]

when X is treated as a universally quantified variable then this expression defines the set of all red objects. This is the description of the concept 'red object'.
If the language allows conjunctions of predicates, say

[X: colour(X, red) ∧ shape(X, sphere)]

then the set described is the intersection of the set described by the atomic predicates in the expression. Similarly, a disjunction (logical OR) describes the union of the sets defined by the predicates.

As we will see, there are many variations on the pure predicate calculus language. These variations arise from the particular emphasis of the learning system.

2.2.1.1 Parameterized Structural Representations

Hayes-Roth (1977) has developed a language which is equivalent to predicate logic, but has some advantages over it. Training instances and concepts are represented by *P*arameterized *S*tructural *R*epresentations (PSRs). A PSR consists of a set of parameters and a set of relations. For example, to describe the scenes in Figure 2.2 (Hayes-Roth and McDermott, 1978) the PSRs are,

E1: {{TRIANGLE: a, SQUARE: b, CIRCLE: c},
{LARGE: a, SMALL: b, SMALL: c},
{INNER: b, OUTER: a},
{ABOVE: a, ABOVE: b, BELOW: c},
{SAME-SIZE: b, SAME-SIZE: c}}

E2: {{SQUARE: d, TRIANGLE: e, CIRCLE: f},
{SMALL: d, LARGE: e, SMALL: f},
{INNER: f, OUTER: e},
{ABOVE: d, BELOW: e, BELOW: f},
{SAME!SIZE: d, SAME!SIZE: f}}

An expression such as {INNER: b, OUTER: a} is called a *case relation*. It consists of *properties* INNER and OUTER and *parameters* a and b.

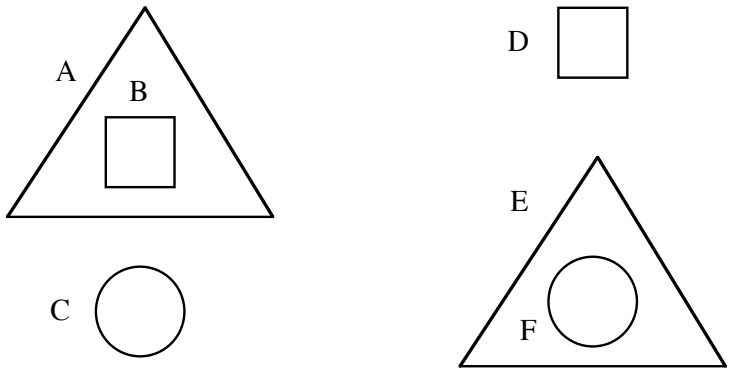
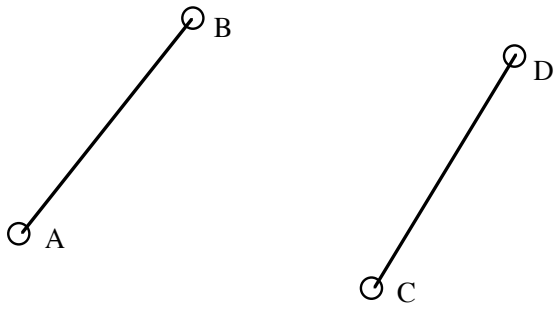


Figure 2.2. SPROUTER example

Why should this method of representation be chosen instead of the conventional predicate logic? Consider this example:



A predicate logic description of this could be:

$$\text{line}(a, b) \wedge \text{line}(b, a) \wedge \text{line}(c, d) \wedge \text{line}(d, c)$$

Note that some duplication is necessary because the parameters above are considered to be *ordered* pairs even though no ordering is wanted. That is, two predicates are required to represent the symmetry of the objects. A PSR representation might be:

E3: {{ENDPOINT: a, ENDPOINT: b} , {ENDPOINT: c, ENDPOINT: d}}

Here the symmetry is obvious. However, this representation is still complete. Here is a second description of two lines:

E4: {{ENDPOINT: w, ENDPOINT: x} , { ENDPOINT: x, ENDPOINT: y}}

The lines share a common endpoint. Implicit in these descriptions is the assumption that the endpoints are the same only if they are labeled by the same parameter. The fact that there are four points (not necessarily distinct) cannot be obtained from the case relations above.

To avoid this problem the PSR's are transformed into *uniform* PSR's. Here, distinct parameters are used in each case relation, and new relations are added to establish the equivalence of variables. Similarly, new relations are added to distinguish different objects. The uniform representation of E4 becomes:

{{endpoint:x1, endpoint:x2},
{endpoint:x3, endpoint:x4},
{DP:x1, DP:x2},
{DP:x1, DP:x3},
{DP:x1, DP:x4},
{SP:x2, SP:x3},
{DP:x3, DP:x4}}

'x' in the first description has been replaced by x2 and x3 which are put into a new relation indicating that they are the *S*ame *P*arameter. The other variables must be distinguished as *D*ifferent *P*arameters. E3 would have a similar uniform description except that x2 and x3 would be different parameters.

As we will see when we discuss various learning strategies, this representation will allow us to discover concepts which could not be found using ordinary predicate logic. However, the language does have its disadvantages. It is not capable of growth, although it may be extended to allow this. PSR's can only represent conjunctive concepts, and the NOT connective of predicate logic has no equivalent here.

2.2.1.2 Relational Productions

Vere's work is concerned with developing formal induction algorithms for expressions in predicate calculus (Vere, 1975). Originally this work was seen as creating a dual for deductive theorem proving (Plotkin, 1970). In Vere's language a *literal* is a list of terms like (ON.X1.X2). An identifier preceded by a period is a variable. Other terms, such as, ON are constants. A *product* is a conjunction of literals:

(COLOUR .X1 RED) (SHAPE .X1 SPHERE)

This language formed the basis for a number of extensions which have increased the descriptive power of the system. One extension was the development of *relational production*.

Relational productions bear some resemblance to STRIPS type productions (Fikes, 1972). For example, the following production describes the change which takes place when a block, a, is moved from on top of another block, b, to a third, c.

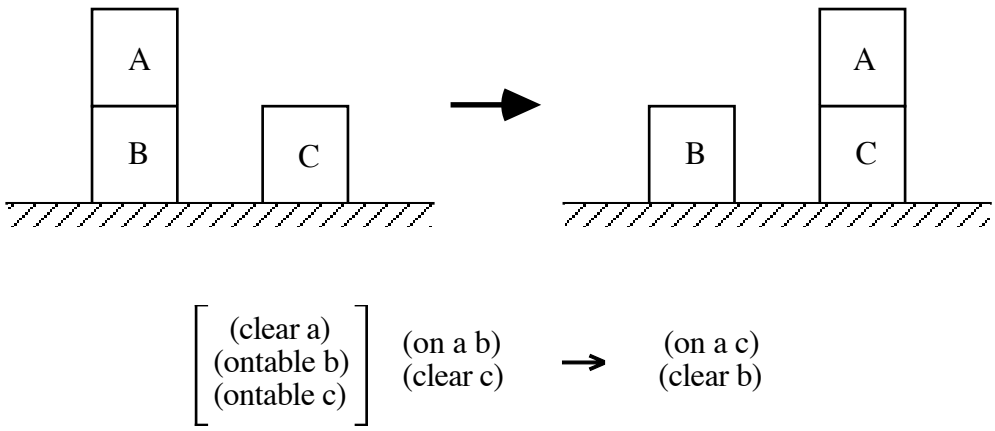
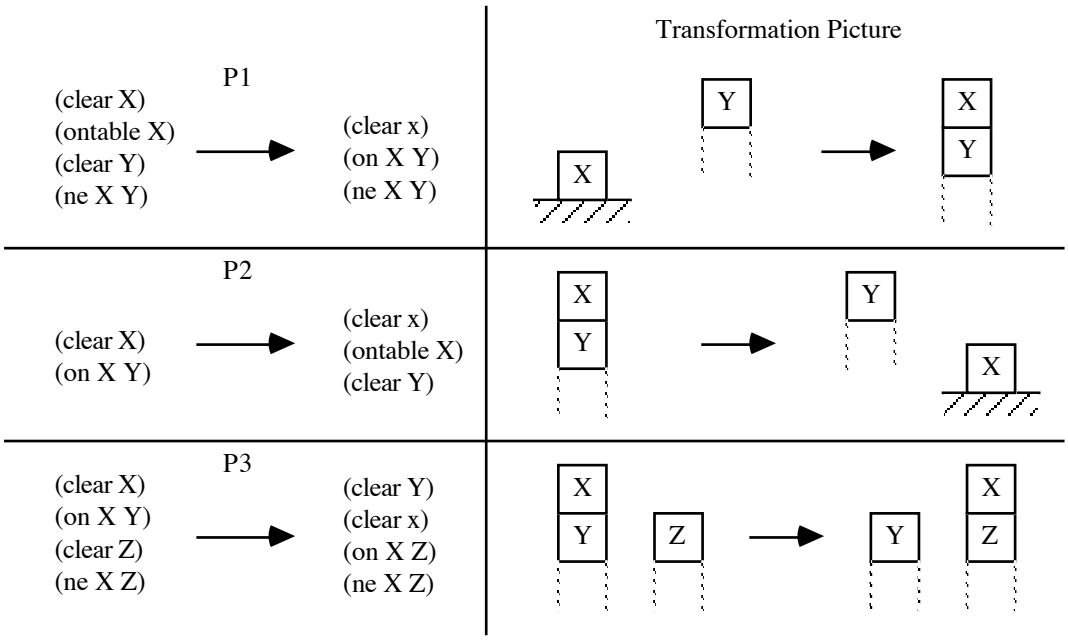


Figure 2.3. Before and after pair

The left-most group of predicates are the *context* or invariant conditions which are not changed by the operation. (on a b) and (clear c) which are true before become false after. (on a c) and (clear b), initially false, become true after the firing of the rule.

First order predicate logic is not very well suited to describing change of state. However, in robot planning, it is essential to be able to do this easily. Relational productions can extend the descriptive power of ordinary predicate logic so that changes of state can be expressed in a concise way. Production systems are also common in many knowledge based expert systems, consequently, a system capable of learning productions can be used to build an expert's knowledge base.

Note that the production above only describes the way in which a block is taken from one supporting block to another. In order to describe the range of different operations which can take place, a *disjunctive concept* is necessary. This is represented by a set of productions, each of which describes one type of operation.



Sometimes it is necessary to specify exceptions to a rule. For example, an almost universal criterion for establishing that an animal is a bird is that it flies. However, there are some exceptions. Bats are mammals, but they fly. To express this it is necessary to introduce logical negation. For example, (flies.X) ~(bat.X) could describe the concept 'bird'. Vere's THOTH program is capable of learning expressions of the form,

$$P \sim (N_1 \sim (N_2 \sim ...))$$

P is a product which represents the concept. N_1 is a product which describes an exception to *P*, N_2 is an exception to the exception, *etc*. The negative products are called *counterfactuals*. [Of course there are some birds which do not fly; they would go into a separate disjunct of the concept].

Vere's language contains features such as disjunction and negation which Hayes-Roth's language does not have. It also introduces the relational production. However, some of the problems associated with variable bindings in predicate logic, which Hayes-Roth tried to solve, still occur in THOTH. Recently Vere has reported further work associated with variables bindings (Vere, 1981). At present, THOTH is still incapable of adding to its descriptive power by growing.

2.2.1.3 Variable Valued Logic

Variable Valued Logic is the name given by Michalski (1973) to an augmented form of predicate logic. One of the main reasons for developing this class of languages was to make concept descriptions more readable for humans. To achieve this, according to Michalski, the number of disjunctions should be minimized, the number of predicates in a conjunction should also be kept small. Recursion must be avoided if possible.

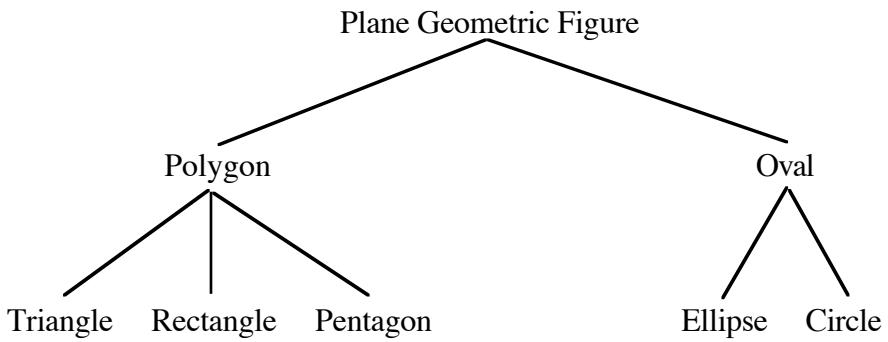
The basic elements of VL languages are called *selectors*. Some examples are:

[colour(box1) = white]
[length(box1) >= 2]
[weight(box1) = 2..5]
[blood-type(P1) = O, A, B]
[on-top(box1, box2)]
[weight(box1) > weight(box2)]
[type(P1).type(P2) = A, B]

One of the most effective ways of simplifying a description is the use of the *internal disjunction*. The expression 2..5 represents a range meaning that the weight of the box may be between 2 and 5 units. The blood-type of person P1 may be any of O or A or B. The last rule above states that both P1 and P2 may have blood-types A or B.

Variable Valued Logic is intended to be a general purpose description language. In order to allow a program using VL to operate in a specific domain, the user supplies the system with domain knowledge. This is done by specifying the type and range of the descriptors that will be used. The types are,

Unordered	There is no structure in this domain (e.g. blood-type)
Linearly Ordered	The domain is a linearly ordered set (e.g. weight)
Tree Ordered	Elements are ordered in a tree where a superior node represents a concept which is more general than its descendents. For example:



Among the other environment specifications, the user may describe the properties of predicate functions such as

$$\forall\, x_1, x_2, x_3 \left([\text{left}(x_1, x_2)][\text{left}(x_2, x_3)] \quad [\text{left}(x_1, x_3)] \right)$$

which states that if x_1 is left of x_2 and x_2 is left of x_3 then x_1 is left of x_3.

The ability to add domain knowledge is one way of tailoring a general purpose language to the requirements of a specific environment. This avoids the necessity of build an entirely new language for each new problem and still provides descriptors that are appropriate for describing concepts succinctly.

2.2.1.4 CODE

Banerji (1969) suggested that it would be possible to create effective descriptions by learning the domain knowledge. This is the approach taken by Cohen (1978) in his program, CONFUCIUS. The description language, called CODE, becomes more powerful as more knowledge is acquired.

Simple expressions are of the form:

colour(X) = red
x ∈ set1
set1 ⊃ set2

For each operator there is also the negation, ~, *etc*, enabling the representation of exceptions. There is also another operator, *contained-in* which is true when an object is contained in a concept that is in CONFUCIUS' memory. Thus,

(X, Y) **contained-in** connected **iff**
neighbour(X) = Y **and**
neighbour(Y) = X

recognizes points X and Y which are connected by a line segment.

(X, Y, Z) **contained-in** triangle **iff**
(X, Y) **contained-in** connected **and**
(Y, Z) **contained-in** connected **and**
(Z, X) **contained-in** connected

recognises the triangle described by the vertices X, Y and Z. Notice that *triangle* used *connected* in its description. A knowledge of triangles requires a knowledge of straight lines, as one would expect. This

demonstrates the way in which CONFUCIUS learns to understand more about its world as it learns more concepts. In many ways this models the behaviour of humans. We develop a greater understanding of our world by a long process of acquiring gradually more sophisticated concepts.

Disjunctive concepts can also be expressed in CODE. The language also allows recursion which is essential for describing abstract concepts of among other things, numbers and lists.

The main goal influencing the design of CODE is the ability of one concept to refer to another; CODE is a growing language. Each description language has its merits and its faults. Which one is chosen depends on the design goals of the learning system. However, the choice of representation also profoundly affects the design of the learning strategy.

2.3 LEARNING STRATEGIES

As we demonstrated in the toy blocks example, the kind of learning we are investigating involves generalizing the description of a particular object to a more general description of a class of objects. In this section we will discuss some different generalization procedures, but first, we must give some informal definitions of a few frequently used terms.

Definitions

1. We will assume the the sensory pre-processors of our learning machine report the results of its measurements as predicates such as colour(X,red). This expression is true when the object represented by X has a property called *colour* and the value of colour is *red*. There is an equivalent representation for such a statement in all of the general purpose languages described in the last section.
2. Basic predicates may be combined with ANDs and ORs in the usual way.
3. If a concept, C, is described by a logical expression, P(X), then we say C*recognizes* the object, Obj if P(Obj) is true.
4. A concept, C_1 is *more* general than another concept C_2 if every object recognized by C_2 is also recognized by C_1.
5. In many learning algorithms, it is necessary to be able to *match* expressions in different concept descriptions. Suppose, for example, that we want to match

$$\text{colour(box1, red)} \wedge \text{size(box1, big)} \qquad (P_1)$$

and

$$\text{colour(X, red)} \wedge \text{size(X, big)} \qquad (P_2)$$

We say that P_1 matches P_2 under the *substitution,* $\sigma = \{box1/X\}$ or $P_1 = P_2\,\sigma$. The expression $P_2\,\sigma$ is obtained by substituting *box1* for every occurrence of *X* in P_2.

6. For conjunctive concepts (concepts with no OR operation) we can give a definition of generalization in terms of the description language. If there exists a substitution σ such that

$$C_1 \supseteq C_2$$

then C_1 is more general than C_2 (Vere, 1975). For example, less(1,Y) which represents the set of all numbers greater than 1, is more general than

$$\text{less(1, X)} \wedge \text{less(X, 5)}$$

If a conjunction is considered as the set of its component literals then, given the substitution {X/Y}, the first expression is a subset of the second. There are fewer constraints on the variables and so it specifies a larger set.

The problem now faced by the designer is to choose a learning strategy that will enable a program to develop a useful generalization efficiently. Learning algorithms are sometimes divided into two classes according to the approach they use.

2.3.1 Data-Driven Methods

If I show you two examples of the same concept and ask 'what is the concept?' your reaction might be to study the examples to see what they had in common. Given two expression E1 and E2, we may consider that a generalization derived from them should contain the features that E1 and E2 hold in common. For example,

$$colour(X, red) \wedge size(X, big) \wedge shape(X, cube)$$

and

$$colour(Y, red) \wedge size(Y, small) \wedge shape(Y, cube)$$

generalizes to

$$colour(Z, red) \wedge shape(Z, cube)$$

So in a sense, we are finding the intersection of sets of predicates. If there are a number of expressions from which we may produce a generalization then we find the intersection of all of them:

$$((E1 * E2) * E3) * E4) \ldots$$

'*' represents the operation of finding the *intersection* or maximal common subexpression of two expressions.

Finding common generalizations isn't as easy as it may first appear. To find common generalizations of two concepts, we have to match predicates. This entails finding consistent parameter bindings between the concepts. In realistic examples, it is usually possible to find more than one substitution. Consider the objects in Figure 2.4 from the example by Dietterich and Michalski (1981).

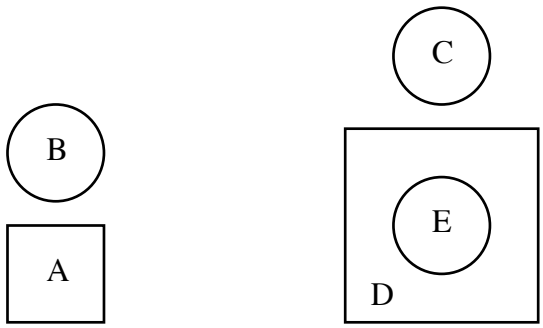


Figure 2.4. Finding the Maximal Match Between Two Examples

These may be described by the following expressions:

E1: $circle(a) \wedge square(b) \wedge small(a) \wedge small(b) \wedge ontop(a, b)$

E2: $circle(e) \wedge square(d) \wedge circle(c)$
$\wedge\ small(e) \wedge large(d) \wedge small(c)$
$\wedge\ ontop(c, d) \wedge inside(e, d)$

If the program begins by trying to find a match for *a* then it may notice that *circle(a)* matches *circle(e)*. Furthermore *small(a)* matches *small(e)*. Thus a substitution { a/e} is possible. However this will not lead to the most obvious generalization, namely that there is a small circle above a square. We may therefore, state our goal as a search for a maximal match of literals and parameter bindings.

The problem of finding greatest common subexpressions is NP-complete. Therefore, enumerative search methods will be very costly unless some heuristics are used to prune the search. The systems developed by Hayes-Roth and McDermott (1978) and by Vere (1975) fall into this category.

2.3.1.1 SPROUTER

Hayes-Roth (1977) has developed an algorithm, called *interference* matching, for extracting the commonalities from examples. The comparison of PSR's is likened to finding the intersection of the sets of case relations. For example, an *abstraction* obtained from the descriptions of E1 and E2 in Figure 2.2 earlier in this chapter, is:

{{ABOVE:1, BELOW:2} ,
{SAME!SIZE:2, SAME!SIZE:1} ,
{SMALL:2}, {SQUARE:1}, {CIRCLE:2}, {TRIANGLE:3}, {LARGE:3}}

There are three objects: a small circle, a small square and a large triangle. The square is above the circle.

Since any subset of the set of common relations is also an abstraction, it is important to distinguish between the set and its proper subsets. An abstraction which is properly contained in no other abstraction is a *maximal abstraction*.

The algorithm to find the maximal abstraction of two PSR's randomly selects a case relation from one PSR and puts it in correspondence with one from the other PSR. Parameters having identical properties are identified as equivalent and the resulting case relation becomes the (primitive) abstraction associated with that set of parameter bindings. Then other pairs of primitive case relations, one from each of the two exemplar PSR's, are out into correspondence. If the new comparison produces bindings which are consistent with previous bindings then the new case relation is added to the abstraction. Otherwise, a new abstraction is formed with the common case relation as primitive. Thus a number of competing abstractions may be produced. Since many unwanted abstractions may be produced heuristics are used to prune the search.

A problem which is encountered using this matching algorithm (and most others) can be illustrated by the following example:

E5: {{SMALL: x}, {SQUARE: x}, {RED: x}}
E6: {{SMALL: y}, {SQUARE: y}, {SQUARE: z}, {RED: z}}

In both E5 and E6 there is a small square and a red square. However, in E5 they are the same object. Thus a method is required that will allow the single instance of SQUARE in E5 to match two instances in E6. Many-to-one binding algorithms are currently under investigation. An essential part of the solution proposed Hayes-Roth and McDermott (1981) is the transformation of a PSR representation into a *uniform* PS representation. By introducing Same Parameter and Different Parameter relations it would be possible to find an abstraction which insisted that the parameters of SMALL and SQUARE are the same, and the parameters of SQUARE and RED are the same, but it doesn't care if the parameters of SQUARE and RED are different.

2.3.1.2 THOTH

In a PSR representation of a concept, the members of a case relation are unordered. Relations must be matched according to corresponding property names such as ABOVE and BELOW. In Vere's representation there are no special property names, instead there are literals which are ordered lists of terms. Although the two languages have much in common, their internal representations are quite different; consequently, the matching algorithms are also different. To illustrate Vere's matching algorithm, we will use one of his own examples (Vere, 1975).

We wish to find a maximal common sub-expression of,

R1 = (B X2) (W X3) (C X2 R) (S X3 X2)
R2 = (W X6) (B X5) (S X6 X5) (D X6 E)

Literals from R1 and R2 which are the same length and contain at least one term in common in the same position are paired.

P1 = (B X2) (B X5)
P2 = (W X3) (W X6)
P3 = (S X3 X2) (S X6 X5)

Terms in the pairs which do not match are replaced by variables producing the generalization,

(W .Y) (B .Z) (S .Y .Z)

In this example, only one set of pairs could be produced, but as we saw in Section 1.4.1, some matching problems may result in several possible ways of pairing literals. In this case the pairs which

give the longest generalization are chosen.

If the concept being learned is conjunctive, then it is sufficient to find the *intersection* of all the products to produce the generalization. Generalizing disjunctive concepts poses a few problems.

It is not possible to simply match descriptions of instances any more. If two instances belong to different disjuncts, then the matching descriptions will produce an empty or incorrect generalization. Therefore Vere (personal communication) adopts the following modification:

> Suppose the instances shown to the program are I1, I2, I3, I4, I5, I6. The procedure begins by trying to generalize I1 and I2 to find a maximal common generalization (mcg). Suppose also that I1 and I2 are successfully generalized into mcg G1. We then try to generalize G1 and I3. Suppose this attempt fails because I3 belongs to a different disjunct to I1 and I2. We jump over I3 and try to generalize G1 and I4. This succeeds giving mcg G2. Now we try to generalize G2 and I5. This fails. Then we try to generalize G2 and I6. This succeeds giving mcg G3. The procedure then begins another pass through the remaining instances, that is, through the list I3, I5. Suppose I3 and I5 can be successfully generalized into G4, then we obtain the disjunctive generalization G3 ∨ G4.

THOTH is also capable of performing other learning tasks. These will be described only briefly here.

In many learning tasks there may be information relevant to the problem but which is not supplied explicitly. This may be in the form of domain knowledge or as Vere calls it *background information*. To illustrate this situation, Vere (1977) uses the example of teaching the program poker hands. To teach *full house* it is sufficient to show the program two hands both of which have three cards with the same number and the remaining two with a different number. No information is required beyond that present in the description of the examples. However, to learn the concept *straight*, the program must know something about the ordering of cards in a suit to recognize that the hands shown as examples contain a numerical sequence. This background information may be represented as:

(next 2 3) (next 3 4) ... (next K A)

When a hand is shown as an instance of *straight* the program must be able associate the description with the background information.

THOTH is capable of learning *counterfactuals*. Taking another of Vere's examples (Vere, 1980), the task of the learning system is to find a description which discriminates between the set of objects on the right and and the set on the left in Figure 2.5. It is not possible to produce such a description without making exceptions. The objects on the right are described as having an object X on top of an object Y. Y is a green cube. X must not be blue except if it is a pyramid.

Recently, Vere has developed a new algorithm for *constrained N-to-1 generalizations*. At the time of writing, complete details of the algorithm were not available.

2.3.2 Model-Driven Methods

Model-driven learning methods usually begin with a single hypothesis which is used as a starting point for a search. Generally these methods can be characterized as follows:

Create initial hypothesis
while hypothesis ≠ target concept **do**
Apply a transform which will produce a new hypothesis which is either
more general or more specific, depending on the search strategy.

Here we have a situation which is, in some ways, similar to game playing or problem solving. There is an initial state and a goal state. By generalizing the description, the program performs an operation which is part of a search for the goal. The difference between playing chess and learning is that the chess program knows when it has reached the goal state, a learning program does not.

The space of the search is the set of all sentences in the language which can be derived from the initial hypothesis by the application of the transforms. Mitchell (1977) calls this the Version Space of the concept formation task. Three things must be decided when developing the search algorithm:

1. *What is the starting point?*
   The program may begin with a hypothesis for the concept which is very specific, being the description of a single object. It may then proceeded to generalize this description until it describes

a sufficiently general class. Alternatively, the program may choose a starting hypothesis which is too general. It then tries to produce new hypotheses which are more specific.

2. *What transformation rules should apply?*
   In each language there must be a way of determining which of two descriptions is more general than the other. This means that the space of concept descriptions is partially ordered (Mitchell, 1977). The transformations rules must use this ordering to produce a new hypothesis.
3. *What search strategy is appropriate?*
   The designer must choose a search strategy such as depth-first or breadth-first etc.

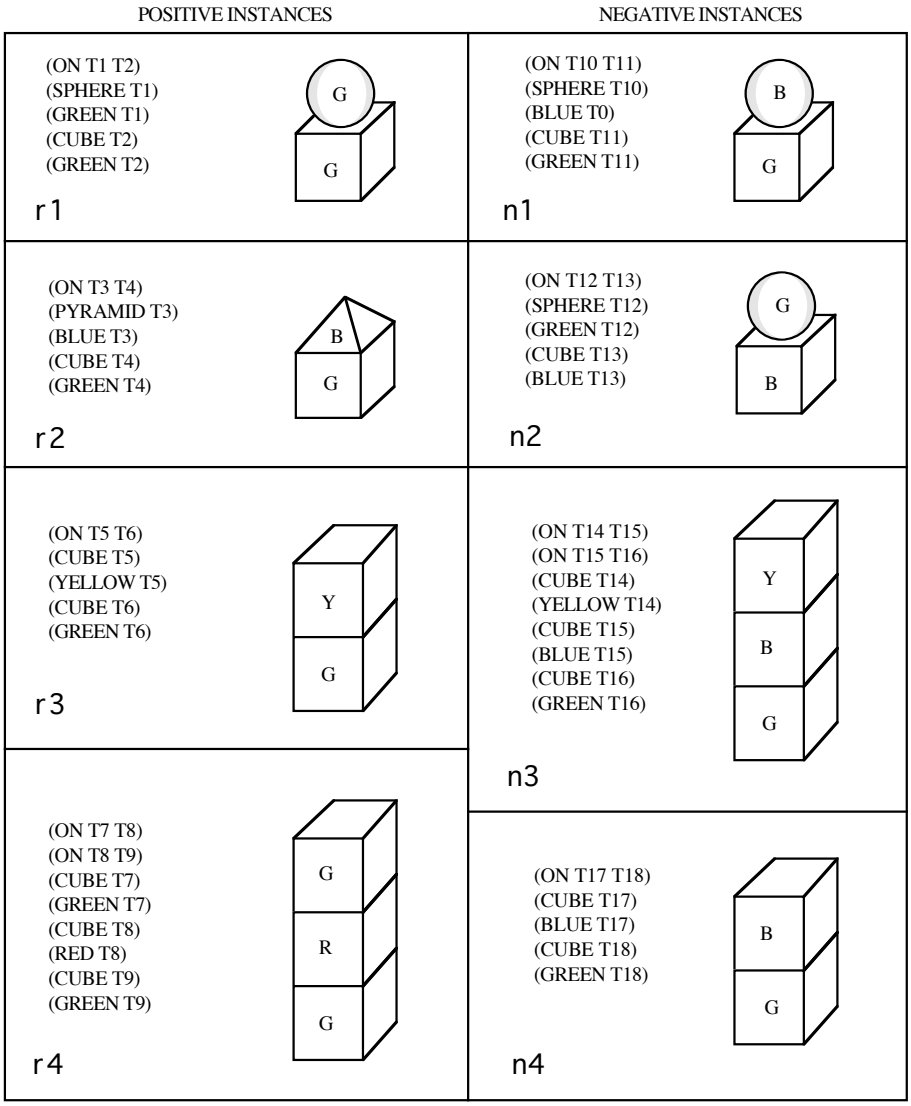


Figure 2.5. Vere's Introductory problem

2.3.2.1 Meta-DENDRAL

When presented with a large number of training instance, how does Meta-DENDRAL generate its rules? The search is *general-to-specific*. That is, it begins with the most general possible hypothesis, then uses a breadth first search to find a more specific rule which is acceptable.

The most general rule is that any bond between any molecule will break.

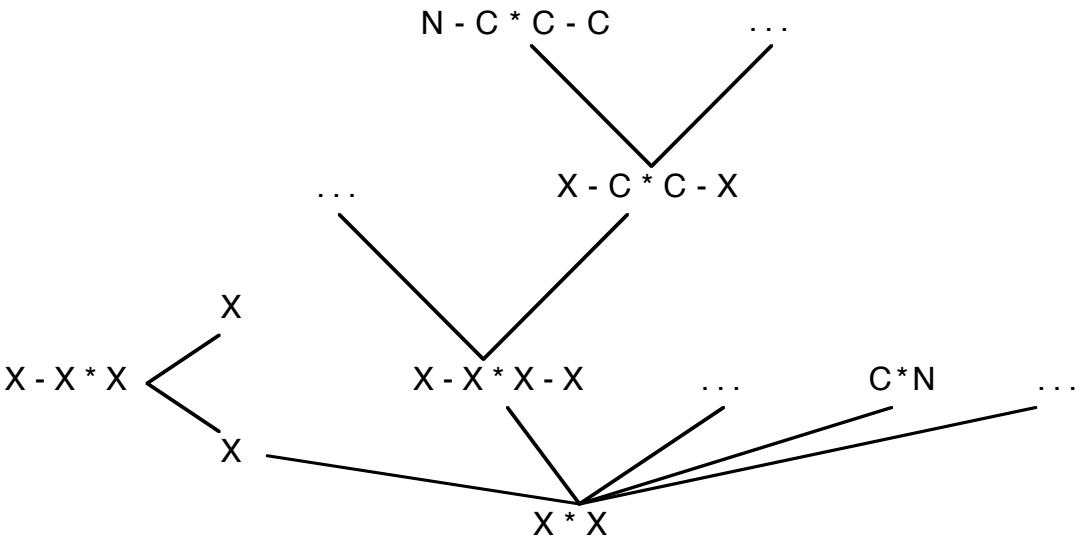


Figure 2.6. Portion of a Meta-DENDRAL Search

This is represented by X*X in Figure 2.6. In one step of the search, each attribute of a node is changed. This results in a number of alternative rules which must be evaluated to determine which are suitable. The criterion used is as follows,

> If a rule matches fewer positive instances than its parent (that is, the rule is more specific) but matches at least one positive instance in at least half of the training molecules (that is, it is not too specific) then this rule is 'better'. The search may continue along this path. If this condition is not met by any of the descendents of a rule, then the parent is output as the most acceptable rule.

Meta-DENDRAL illustrates another problem which must be faced by the designer of a learning program. Some input samples may be incorrectly classified. Such input is called 'noise'. In order to avoid being misled, the program uses a probabilistic evaluation function for guiding its search.

Another point to note is that when the problem domain is well understood, the designer can take advantage of specialized knowledge. For example, the search procedure used in Meta-DENDRAL would be unacceptable in an environment which had a greater branching factor than this problem.

2.3.2.2 VL-Systems

Michalski recognized the importance of domain knowledge in learning. However, his aim was to design a general purpose model-driven learning program. Therefore, the VL_2 systems, developed at the University of Illinois, allows the human user to describe the domain in the description language rather than have the domain knowledge fixed in the structure of the learning program itself.

The problem which the Illinois group deals with is this: Given,

- A set of data rules which specify the input (training samples).
- A set of rules to define the problem environment
- A preference criterion which, for any two symbolic descriptions specifies which is preferable

determine a set of rules to describe the input, which is more general than the data rules.

The purpose of the preference criterion is to give the user some control over the program's search. It is envisaged that the program will be used by human experts interactively to refine knowledge about a particular problem. Therefore an important goal is to produce rules that are easily read by people. The preference criteria allow the user to tell the program what form of rule to look for.

The learning algorithm used in the program, INDUCE-1.1 is described by Dietterich (1978) and Michalski (1980). Briefly, the program begins by augmenting the data rules input by the user, by using the inference rules in the domain knowledge to produce new rules. For example, if an example includes a triangle, the fact that it is also a polygon is also added. When all the inferred rules have been added, the program begins a breadth first search for the most specific generalizations which satisfy the

preference criteria. Throughout the search there are a number of mechanisms to limit the branches which are pursued.

The search proceeds as follows: The algorithm builds a sequence of sets called *partial stars*, denoted by P_i. An element of a partial star is a product (i.e. conjunction of selectors). The initial partial star (P_i) consists of the set of all selectors, e_i, from the description of the training example. These are considered single element products. A new partial star P_{i+1} is formed from an existing partial star P_i such that for each product in P_i, a set of products is placed into P_{i+1} where each new product contains the selectors of the original product plus one new selector of the e_i which is not in the original product. Before a new partial star is formed, P_i is reduced according to a user defined optimality criterion to the 'best' subset before a new partial star is formed.

2.3.2.3 CONFUCIUS

Well before the current group of learning systems were under development, Banerji (1964) had proposed a predicate logic description language which could 'grow'. That is, concepts learned during one training session are stored in memory and may be use by the program in the future to simplify the description of a new concept to be learned. Thus domain knowledge could also be learned.

The original learning algorithm used in CONFUCIUS was derived from the work of Pennypacker (1963). This, in turn, was derived from the Conservative Focusing Algorithm described by Bruner et al (1956). The algorithm developed by Cohen (1978) for CONFUCIUS is:

1. An instance is presented to the program by the trainer.
2. The program generates all the true statements it can to describe the exemplar. This includes statements describing containment in previously learned concepts.
3. CONFUCIUS then proceeds to remove statements from the description. Remember that a subset of a description is a more general description.
4. The new hypothesis obtained by the removal is tested to see if it is more or less general than the target concept. This may be done in either of two ways:
   - by showing the description of the hypothesis to the trainer and asking if it is part of the concept to be learned
   - or if negative examples have been supplied, by seeing if the hypothesis recognizes any of the negative instances.

In implementing this search method there is one major obstacle to overcome. Suppose the statements

$$\text{colour}(X) = \text{red} \wedge \text{colour}(Y) = \text{red} \wedge \text{colour}(X) = \text{colour}(Y)$$

are in the hypothesis. If only one of the statements is removed, then the hypothesis is no more general than it was before,because the removed statement is implied by the remaining two. Thus CONFUCIUS must be able to keep track of implications and remove sets of statements in order to generalize the concept.

The internal representation of a concept, called a GRAFT, was designed to ensure that the recognition of objects is performed quickly. CONFUCIUS contains an associative memory which finds the concepts which are most likely to recognize an object.

2.4 Marvin

In many respects Marvin is the successor of CONFUCIUS. Its general approach to learning is the same as described for the model-driven systems. The main feature which sets it apart from other learning systems is the fact that it can generate its own training examples.

Some of the problems discussed in this chapter have not been considered although they are important. Marvin cannot learn concepts which involve many-to-one variable bindings, nor can it learn exceptions.

3
An Overview of Marvin

In this chapter we will give a formal description of Marvin. The major components of the program are:

1. A language for describing objects and concepts.
2. An interpreter for the language. This must provide the ability to recognize objects belonging to a given concept. It must also be able to construct an instance of a concept, that is, 'perform an experiment'.
3. An associative memory which stores concept descriptions once they have been learned and enables them to be accessed quickly by the learning procedure.
4. A generalization procedure which, given a concept description, can output a more general description.
5. A learning strategy which starts with an initial hypothesis and repeatedly applies the generalization procedure in a search for the target concept description.

The last component, the learning algorithm, links the whole system together. The main steps involved in the algorithm are:

Initialize The example presented by the trainer is described by a form of First Order Predicate Logic. This description forms the first hypothesis for describing the concept to be learned.

Generalize Marvin tries to generalize the hypothesis. If it is not possible to create a generalization, the learning process stops.

Test The generalization is tested by constructing an object belonging to the hypothesised concept. The object is shown to the trainer. If he agrees that the object is recognized by the concept to be learned then *generalise*.

Restrict If the hypothesised concept recognizes objects not belonging to the concept to be learned then a new, more specific hypothesis is created. *Test* the more specific hypothesis.

To understand how Marvin works it is necessary to know the language it uses to represent concepts.

3.1 The Description Language

As we have already seen, Marvin operates in a universe of objects. In order to differentiate between them, the universe is partitioned according to the values of an object's attributes or properties. For example, The red ball on top of a green block which we saw in Chapter 1 can be represented as a list of property/value pairs:

E1 = <top: S1; bottom: B1>

Top and *bottom* are the names of the properties of the object E1. They may be thought of as the field names of a Pascal record (Wirth, 1972). S1 and B1 are the values of those properties. In this case, they are the names of other objects,

S1 = <shape: SPHERE; colour: RED>
B1 = <shape: BOX; colour: GREEN>

RED, GREEN, BOX, SPHERE also name other objects. One such object is,

RED = <value: red>

This time, *red* is simply a word, and stands for no other object.

A concept description is a boolean expression which describes a class of objects. Ultimately, all concept descriptions specify the properties (and their range of values) associated with each object in the class.

To describe the concept *on top of* Marvin constructs the concept:

```
[X0:
    [∃ X1, X2, X3, X4, X5, X6:
        X0.top = X1  ∧
        X1.shape = X2  ∧
        X1.colour = X3  ∧
        X0.bottom = X4   ∧
        X4.shape = X5  ∧
        X4.colour = X6  ∧
        any-shape(X2)  ∧
        any-colour(X3)  ∧
        flat(X5)  ∧
        any-colour(X6)
    ]
]
```

This can be interpreted as specifying the set of all X0 such that the top of X0 is X1 and the bottom of X0 is X4. The shape of X1 is X2, which may be any shape, the colour of X1 is X3, it may be any colour, and so on. *Any-shape*, *any-colour* and *flat* are the names of other concepts stored in memory.
To find out if E1 is an instance of *on top of*, Marvin would execute the statement on-top-of(E1). A useful analogy for a concept description is the boolean function of ordinary programming languages. When a function of this kind is called, the actual parameters of the call are substituted for the formal parameters in the body of the function. The boolean expression which constitutes the body is then executed and returns a result of true or false. If the result is true then the objects passed to the concept as parameters are instances of the concept.

Concept descriptions differ from boolean functions when a variable is passed as an actual parameter and the variable is not bound to any value. A boolean function would normally fail in this case. However, when a concept is 'executed', it *creates* a value for the unbound variable which satisfies the concept description.

The interpretation of the language is described fully in the next section. In the remainder of this section we will give a complete specification of the syntax of the language.

- An *object* is an element of the universe. It is represented by a list of property/value pairs. A property is a word which has no interpretation, but may be considered as a label.
- A *value* is a number or a word or another object or object name.
- An *event* is a sequence of values (v_1, v_2, ...). For example, (S1, B1).
- A *term* is a value, or a variable, or a selector.
- *Variables* are written as X0, X1, ...
- A *selector*

  <variable> . <property name>

  Thus, Xi.prop is interpreted as the value associated with property, prop, in the object represented by Xi. The value of *S1.colour* is RED.
- A *statement* is a predicate of the form

  $$C(t_1, .., t_n)$$

  where C is the name of a concept which *recognizes* the event (t_1, .., t_n). The t_i are terms. The term *recognize* is used here as defined in Section 2.3. For convenience the predicate *equal* will be written as

  $$t_1 = t_2$$

  but its internal representation is the same as 'equal(t_1, t_2)'. *Equal* is the only concept which is built into the language. Thus it is possible to say

  S1.colour = red

  However other predicates (concepts), such as flat(B1), must be learned by Marvin.
- A *conjunction* is an expression of the form

$$[\exists\ X0, X1, ... : S1 \wedge S2 \wedge ...\ ]$$

where X0, X1, ... are existentially quantified variables and S1, S2, ... are statements which contain references to X0, X1, ... For example,

$$[\exists\ X1, X2: X1.shape = X2 \wedge X1.value = sphere\ \wedge\ ...\ ]$$

- A *concept*, C has the form

$$C = [X0, X1, ... : D1 \vee D2 \vee ...\ ]$$

where X0, X1, ... are universally quantified variables and D1, D2, ... are conjunctions which contain references to X0, X1, ... The definition of *flat* given in Chapter 1 may be expressed as,

$$[X0: X0.value = box \vee X0.value = table]$$

3.2 Interpreting the Language

In the *on top of* learning task, we saw that to test a generalization, a new object was shown to the trainer. To show an appropriate object, Marvin must treat the generalized concept description as the specification of the object to be created. The object is constructed as a side-effect of an attempt to 'prove' the concept description. This is very similar to the method used by Prolog to interpret its programs.

As an example, consider the concept which describes a list, X2, obtained by appending the list, X1, onto another list, X0 (cf. Prolog definition Section 1.4). This may be described as:

```
append =
[X0, X1, X2:
        X0.value = nil ∧ X2 = X1 ∧ list(X1)
 ∨
        [∃ X3:
            X0.hd = X3
          ∧ X2.hd = X3
          ∧ number(X3)
          ∧ append(X0.tl, X1, X2.tl)
         ]
]
```

If X0 is nil then X2 is the same as list X1, otherwise the head of X0 is the same as the head of X2 and the tail of X2 is the result of appending X1 to the tail of X0.
The concept 'list' describes a list of numbers and may be defined as,

```
list =
[X0:
        X0.val = none
 ∨
        number(X0.hd) ∧ list(X0.tl)
]
```

Marvin has no in-built knowledge of numbers. For this example, let's assume that it has learned the concept 'number' already. We will see how this is done in Chapter 4.
Suppose x is the list [1, 2] and y is [3]. If we assert that

$$[\exists\ z: append(x, y, z)]$$

then Marvin will try to prove that this expression is true. That is, it must prove that the list, z, exists. It is a proof by construction since if such a z can be found the assertion is obviously true.

Assuming that the definition of append is known to Marvin, the 'proof' procedure is as follows:

1. Marvin retrieves the definition of append from its memory and calls it as if it were a boolean function. The quantified variable, z, is represented by a 'dummy value' which is passed as the actual parameter. As the concept is entered, X0 is bound to [1, 2], X1 is bound to [3] and X2 is bound to z's dummy value.

2. Now an attempt is made to prove each conjunction in append until one is found to be true. Since X0.value ≠ nil the first conjunction fails, an attempt is made to prove the second.

3. Encountering the expression ∃ X3: .., the program creates a dummy value for X3. The statement X0.hd = X3 is interpreted as replacing the dummy value of X3 by a real value, i.e. the value of the head of X0. Remember that X2 is bound to z's dummy value. When Marvin sees an expression like X2.hd, it assumes that the dummy value must represent an object which has a head. The value of the head is bound to the value of X3. Thus part of a new list has been created such that the head of the list is the same as the head of X0.

   X0 = <head: X3; tail: ...>
   X2 = <head: X3; tail: * >

4. Next, an attempt is made to prove append(X0.tl, X1, X2.tl). Since X2 represents a list 'under construction', X2.tl is the dummy value, *, representing the tail of the list. This value will be passed as the actual parameter to the next call of append. So the remainder of the list will be constructed by this call.

5. 'append' will be called recursively until

   X0.value = nil ∧ X2 = X1 ∧ list(X1)

   is reached. At this point, X2 is bound to the dummy value, *, representing the tail of the list constructed by the calling concept. This can now be replaced by the value of X1. By now the entire list will have been constructed and the proof terminates.

When we introduced the boolean function as an analog of concept descriptions, we said that the actual parameters (a_1, .., a_n) were substituted for the formal parameters (p_1, .., p_n) in the body of the function. In practice, the function would not be physically changed. Instead, the correspondence between actual and formal parameters is recorded as a set of pairs {a_1/p_1, .., a_n/p_n}. This will be referred to as the *binding environment* of the concept. Similar associations will appear throughout Marvin. Another example of such bindings occurs when a property belonging to an object is bound to a specific value. The property/value list can be regarded as a special set of bindings associated with an object.

Let us now describe how the value of a term is found. The value can only be specified with respect to a particular binding environment. A function 'value(Term, Env)' will be defined. It returns the value associated with term, Term, in binding environment, Env.

```
value(X, Env) = X if constant(X).
value(X, Env) = Y if
        variable(X) and
        member(X/V, Env) and
        Y = value(V, Env).
```

If X is a variable then the value of X is the value of the term bound to X in the current environment.

```
value(X, Env) = Y if
        Obj = value(X, Env) and
        member(P:Y, Obj).
```

The value of the selector X is the value associated with the property, P in the object, Obj, represented by X. If Obj is being constructed and the property, P does not yet exist in Obj, then a new property/value pair, P:* is added, where * is a dummy value.

Now we can specify the semantics of the statements of Marvin's language. A statement is said to be true if the function 'prove' is successful. A proof can only be described with respect to a binding environment.

prove(X = X, Env) is always true.

prove(X = Y, Env) **if**
number(X) **and**
number(Y) **and**
X and Y are numerically equivalent.
prove(X = Y, Env) **if**
word(X) **and**
word(Y) **and**
X and Y are alphabetic equivalents.
prove(X = Y, Env) **if**
object(X) **and**
object(Y) **and**
for each pair $p{:}v_1$ **in** X **there is** a pair $p{:}v_2$ **in** Y
such that prove($v_1 = v_2$, Env).
prove(X = V, Env) **if**
Y = value(X, Env) **and**
prove(Y = V, Env) is true.
prove(X = Y, Env) **if**
variable(X) **and**
X = value(Y, Env).

If X is an unbound quantified variable represented by a dummy value, then the dummy is replaced and X is bound to value(Y, Env).

prove(X = Y, Env) **if**
variable(Y) **and**
prove(Y = X, Env).
prove(P ∨ Q, Env) **if**
prove(P, Env) **or** prove(Q, Env).
prove(P ∧ Q, Env) **if**
prove(P, Env) **and**
prove(Q, Env).

P and Q must be proved to be true simultaneously. If the proof of P creates bindings which prevent Q from being proved then Marvin must try to find an alternative proof for P. This may be done if P represents a disjunctive concept and there is more than one disjunct which may be true.

prove(P(<args>), Env) **if**
P is the name of a concept [<formal>: <expr>] **and**
<actual> are bound to <formal> to create a new environment NewEnv **and**
prove(<expr>, NewEnv) is true.

When a statement like P(x, y, z) is to be proved, the actual parameters x, y and z must be bound to the formal parameters of P. This creates a new binding environment, NewEnv, for the proof of the expression which describes P.

prove([∃ <exvars>: P(<exvars>)], Env) **if**
<exvars> are represented by dummy values **and** prove(P(<exvars>), Env) is true.

Dummy values are created to represent each quantified variable. The dummy values will be replaced by real values during the execution of P.

The specification given above is similar to that given in (Banerji, 1978). The most important difference is that in this system, expressions may contain references to other concepts. And since concepts may be disjunctive, we must consider the possibility of backtracking as discussed above.

Cohen (1978) also proposed an object construction procedure which could be used in a concept learning system like CONFUCIUS.

3.3 Pattern Matching and Memory

Marvin's memory consists of two parts:

1. The set of all concepts learned so far.
2. An index to the learned concepts.

The index consists of a set of pairs <Stmnt, List> where List is the list of all the concepts which contain statements which *match* Stmnt. Given another statement, S, the index enables us to find all the concepts which contain statements which match S by looking up the corresponding Stmnt, which also matches S.

While performing a memory lookup and during other parts of the learning process, Marvin must compare or *match* statements. The matching procedure is a simple unification algorithm very much as one would find in a theorem prover (Robinson, 1965). Two expressions X and Y may be matched using the following algorithm:

```
unify(X, Y)
        if X is a variable then
            if X is bound to value v then return unify(v, Y)
            else bind v to Y and return TRUE
        if Y is a variable then return unify(Y, X)
        if X and Y are both objects or atoms then
            return TRUE if X is identical to Y
        if X and Y are both numbers then
            return TRUE if the numerical values are the same
        if (X = Obj1.Prop) and (Y = Obj2.Prop) then
            return unify(Obj1.Prop, Obj2.Prop)
        if X and Y are both statements then
            if the predicate names are the same
            and corresponding arguments of X and Y unify
            then return TRUE
        else return FALSE
```

As in the object construction procedure, unification also binds values to variables, although the purpose is now slightly different. If an attempt is made to match an unbound variable with a value or variable, then it becomes bound. In this way, the unification algorithm builds up *substitutions*. Following the practice of resolution theorem proving literature, we obtain a substitution instance of an expression by substituting terms for variables in that expression. A substitution is a set of ordered pairs $\{t_1/v_1, t_2/v_2, ...\}$. To denote the substitution instance of an expression, E, using a substitution, σ, we write $E\sigma$. Thus $C(X0.colour, X1) = C(X1, X2)\ \sigma$, where $\sigma = \{ X0.colour/X1, X1/X2\}$.

Unify enables us to build an index to memory which has the property that for every statement S' in every concept C, there exists a pair <S, L> such that $S' = S\ \sigma$ for some σ and $C \in L$.

The learning algorithm will ensure that concepts are learned in disjunctive normal form. That is, if a concept consists of a number of conjunctions, then one conjunction is learned at a time. And one conjunction is stored at a time. Part of the storing routine ensures that the assertion above is always true.

3.4 Generalisations

In Chapter 2 we defined the meaning of 'generalisation' for a simple predicate logic language. Let us now do the same thing for Marvin's language, but in more detail.

Definition 3.1: The *cover* of a concept, C, (written cover(C)) is the set of all events which are recognized by C.

Definition 3.2: If there are two concepts:

$$C_1 = [X: P(X)] \text{ and } C_2 = [X: Q(X)]$$

C_1 defines the set of all events (X) such that P(X) is true and C_2 defines all (X) such that Q(X) is true. The cover of concept which is the conjunction of P and Q:

$$[X: P(X) \wedge Q(X)]$$

is cover(C_1) $\cap$ cover(C_1).

Definition 3.3: Similarly the cover of a concept which is the disjunction of P and Q:

$$[X: P(X) \vee Q(X)]$$

is cover(C_1) $\cup$ cover(C_2).

Definition 3.4: A concept C_1 is *more general* than a concept C_2 if

$$\text{cover}(C_1) \supseteq \text{cover}(C_2)$$

That is, C_1 recognizes all the events which C_2 recognizes and possibly others as well.

An important part of the learning algorithm that we are going to develop is a method for transforming an expression which represents a concept into an new expression describing a more general concept. Therefore we must define generalizations in terms of the constructs of the language. Several authors have previously proposed definitions of generalization for first order predicate logic languages (Reynolds, 1970; Plotkin, 1970; Vere, 1975). It will be useful to recall Vere's definition: A conjunction of literals D_1 is a *generalization* of conjunction D_2 if

$$\exists\, \sigma: D_2\, \sigma \supseteq D_1$$

This follows from definition 3.2 which stated that the cover of a conjunction is the intersection of the covers of the individual literals.

Definition 3.5: C_1 is a *proper generalization* of C_2 if

1. $\exists\, \sigma: D_2\, \sigma \supset D_1$
2. $\exists\, \sigma: D_2\, \sigma \supseteq D_1$ but σ is not an alphabetic variant substitution. Expressions are alphabetic variants if they only differ in the names of variables.

Definition 3.6: An expression, C_1, in disjunctive normal form (a disjunction of conjunctions), is a *generalization* of another disjunctive expression, C_2 if for each conjunction in C_2 there is a conjunction in C_1 which is more general. That is:

$$[\forall\, D_2 \in C_2: [\exists\, D_1 \in C_1: D_1 \geq D_2]]$$

The symbol $\geq$ is used to indicate the D_1 is a generalization of D_2.

Now let's see how these definitions apply to Marvin. In Chapter 1 when Marvin, the child, tried to generalize his hypothesised concepts, he replaced a statement (or group of statements) by a more general statement. To illustrate this, consider the description of E1, the original training example:

```
[X0:
    [∃ X1, X2, X3, X4, X5, X6:
          X0.top = X1
        ∧ X1.shape = X2
        ∧ X2.value = sphere
        ∧ X1.colour = X3
        ∧ X3.value = red
        ∧ X0.bottom = X4
        ∧ X4.shape = X5(1)
        ∧ X5.value = box          (2)
        ∧ X4.colour = X6          (3)
        ∧ X6.value = green        (4)
    ]
]
```

This might be Marvin's first attempt at producing a description of the concept. It will be called trial T_0.

When, say, the shape of an object is generalized, a specific shape, such as X2.value = sphere is *replaced* by any-shape(X2). Most learning algorithms, such as CONFUCIUS, would *remove* any reference to shape rather than replace it. In Marvin's case, removal is not appropriate because the generalization will be tested by constructing an object to show the trainer. Suppose the object is to be displayed on a colour graphics terminal, then it must have a shape; hence the policy of replacement rather than removal.

Let us now try to describe this replacement process. Let a concept, P, be stored in memory:

```
P =
[X0:
    [∃ X1, X2:
            X0.colour = X1
          ∧ X1.value = red
          ∧ X0.shape = X2
          ∧ X2.value = sphere
    ]
∨
    [∃ X1, X2:
            X0.colour = X1          (P3)
          ∧ X1.value = green        (P4)
          ∧ X0.shape = X2(P1)
          ∧ X2.value = box          (P2)
    ]
]
```

P(X) is true if X is a red sphere or a green box. Under the substitution $\sigma = \{X0/X4, X1/X5, X2/X6\}$ the numbered statements in the description of E1 match *all* the statements in the second disjunct of P. Since all the statements in a disjunct of P can be matched, P(X4) must be true. Thus, we obtain a new concept T_1 by replacing the matched statements in T_0 by the statement P(X4).

```
[X0:
    [∃ X1, X2, X3, X4:
            X0.top = X1
          ∧ X1.shape = X2
          ∧ X2.value = sphere
          ∧ X1.colour = X3
          ∧ X3.value = red
          ∧ X0.bottom = X4
          ∧ P(X4)
    ]
]
```

Now, I claim that T_1 is more general than T_0. T_0 may be split into two sets $T = M \cup M'$, where M is the set of statements replaced by the new statement, S, and M' is the set of remaining statements in T_0. Since S refers to a disjunctive concept and M matches one of the disjuncts of the concept, M must represent a subset of the objects described by S. Thus, cover({ S}) cover(M). Therefore,

$$\text{cover}(\{S\} \cup M') \supseteq \text{cover}(M \cup M')$$

and therefore $T_1 \geq T_0$. The replaced statements in M are said to *directly imply* the new statement, S. To the previous definitions of generalization we can add the following:

Definition 3.7: If C_1 and C_2 are two concepts and there is a subset of statements, M, in C_2 such that M implies a new statement, S, then

$$C_1 = C_2 - M \cup \{S\}$$

is a generalization of C_2. If, S, refers to a disjunctive concept then $C_1 > C_2$. Also, if $C_1 \geq C_2$ and $C_2 \geq C_3$ then $C_1 \geq C_3$.

This defines a replacement transform, R, such that

$$R(T_0, M, S) = T_1 \textbf{ and } T_1 \geq T_0$$

Of course, at any time there may be a number of replacements possible, as is the case with *on top of*. Choosing which ones and the order in which they are done is the task of the learning strategy.

3.5 The Learning Strategy

The learning algorithm is a combination of the generalization and object construction procedures. Our aim is to begin with an initial hypothesis for the concept, called trial T_0, and using the generalization procedure, create a succession of new trials T_i. Eventually there will be a T_n such that any attempt to produce a further generalization, T_{n+1} will result in an *inconsistent generalization*. A generalization is said to be inconsistent if the resulting trial recognizes events that the concept to be learned does not. T_n is called the *target*.

A new trial can be created in either of two ways: If the current trial, T_i is consistent, then T_{i+1} is created by applying the generalization procedure which replaces some statements by a single, more general one. However, if T_i is inconsistent then we do not want to generalize the trial. Instead we must create a T_{i+1} which is more specific and does not recognize those events which the target does not. A more specific trial may be created by *adding* new statements or returning removed statements to the description. Remember from definition 3.2 that by increasing the number of statements, the cover of the trial becomes smaller.

If a statement, S, refers to a concept which has only a single conjunction and S is used to replace its implicants, then the replacement will result in a new trial which is equivalent to the old one. Since S describes exactly those events which are described by its implicants, the new statement will not increase the cover of the trial. Before a new trial is created, Marvin checks that it will be a proper generalization, if it is not then this trial is ignored.

The learning process begins when a training instance is shown to Marvin. First, the description of the event is transformed into a concept description, T_0. This will be the first hypothesis used by the learning algorithm.

The algorithm for this transformation is quite straightforward:

```
for each object in event
    create a variable, X, to represent object
    for each pair p:v in object
        if v is an object then
            Y:= new variable representing v
        else Y:= v
        create statement X = Y
        if v represents an object then
            create statements to describe it as well
```

An example of the way in which this transformation works was given in the previous section when the description of E1 was converted from property lists to a logical description.

During this initial description process, a list of the variables is kept along with the objects which they represent. So at the end we have a substitution which records the correspondence between the objects shown and the variables used.

The statements generated by this procedure are called *primary statements*. The trial at this point is not a generalization because it describes an event identical to the instance which the trainer has shown. All generalizations will be obtained by matching statements in the trial with concepts stored in memory. The procedure may be applied repeatedly. Statements found by generalization of the primary statements may be added to the trial. These new statements, together with the primaries may be used to match against other concepts to produce new generalizations which are in turn added to the trial.

As we saw in Section 3.4, if there is a subset, M, of the trial which matches a disjunct of a concept, P which is in Marvin's memory, then we may replace the statements in M by one of the form P(Xn, .., Xm). If P is a conjunctive concept, that is, there is only one conjunction, then the new trial is exactly equivalent to the old one. However, if P is disjunctive then then trial is a proper generalization because the cover of the trial has been enlarged.

The learning algorithm begins by scanning down the list of statements in the trial.

```
generalize(Trial)
  for each statement, t in Trial,
    TryConceptsWith(t)
```

The variable, Trial, is a global variable known to all the procedures in the program. **TryConceptsWith** uses the statement, t, to look for a concept, P, which might be a replacement to generalize Trial. If it finds a P which contains a match for t, it checks P to find out if all its statements have a match. That is, P recognizes some part of the training event.

Next, the program determines if replacing the matched statement, M, by a reference to P will produce a proper generalization. If so, the new statement is added at the *end* of the new trial.

```
TryConceptsWith(t)
    for each concept P in memory such that
        ∃ a disjunct D in P and D contains a match for t:
        for each subset M, Trial ⊃ M such that M = Dσ
            make new statement S = P(Xn, .., Xm)
            if adding S will make a proper generalization then
                create new Trial = Trial - M ∪ {S}
                if not qualified(Trial) then
                        remove S from Trial and restore M,
                        i.e. return to old Trial
            else ignore P
        if no M can be found then return FALSE
```

σ is the substitution created when the disjunct, D is matched with M. The parameters Xn, .., Xm in the new statement are obtained from σ.

It is possible that the replacement has produced a trial which recognizes events which the target does not. This is an inconsistent generalization. Marvin performs an experiment to test the consistency of the generalization. If the new trial is consistent, the program continues to search for more replacements in order to generalize the new trial. Note that a statement which has already been removed may still be allowed to match parts of a concept in memory. It is only necessary that at least one of the replaced statements still be in the trial.

If the new trial is inconsistent then it must be made sufficiently specific that it is contained in the target. A concept is made more specific by *adding* statements. When the statements in M are replaced by a more general statement, S, we lose some information contained in M. If the generalization was consistent, then the information lost was not important. However, if the generalization is inconsistent then too much information has been lost. This suggests that by re-examining the implicants in M we can determine which statements should be added to make a more specific trial, that is, to return the relevant information to the description of the concept.

```
qualified(Trial)
    if experiment with trial failed then
        for each i in M, (M is inherited from TryConceptsWith)
            put i back into trial
            if TryConceptsWith(i) then return TRUE
            else remove(i)
    else return TRUE
```

The procedure **qualified** searches for the statements which will make the trial sufficiently specific. **Qualified** takes an implicant, i, of the new statement, S and returns it to the trial. It immediately uses, i, to look for new references to concepts to add. This is done by calling **TryConceptsWith** *recursively*. When further statements are added they too must be qualified. This continues until the experiment succeeds or there are no more concepts to try.

The *experiment* which Marvin performs involves showing the trainer an instance of the new trial. Suppose we have the situation shown in Figure 3.1, where a consistent trial T_0 is generalized to an inconsistent trial T_{i+1}.

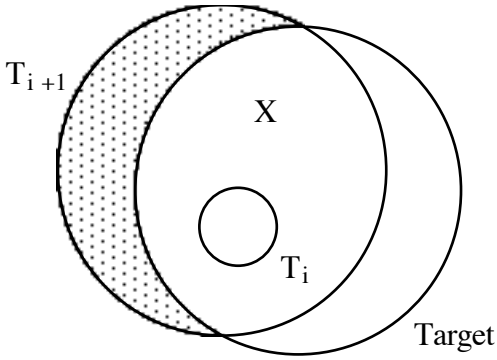


Figure 3.1. Inconsistent Trial

Marvin must produce an object which will tell it if the generalization is consistent or not. Suppose the object, X, in Figure 3.1 is shown. X satisfies T_{i+1} but it also satisfies the target. If X is shown to the trainer, he will answer 'Yes, this is an example of the concept', even though T_{i+1} is inconsistent.

The object construction routine must produce an example from the shaded region in order to be useful. That is the object must be recognized by T_{i+1} but not by the target. How can this be done if the description of the target is not known?

Although the target description is not known, we do, at least, know a set of statements which contain the target. Suppose A is the set of all statements which can be inferred from the primaries, T_0. The target will eventually be obtained by replacing some statements by others which they imply. So the target, T_n must be a subset of A.

$$A \supset T_{i+1} \textbf{ and } A \supset T_n$$

T_{i+1} differs from T_i in that some set of statements, M, has been replaced by a statement S where M S (M implies S).

$$T_{i+1} = T_i - M \cup \{S\}$$

Let $T'_{i+1} = A - T_{i+1}$. T_{i+1} cannot contain all of the statements in T_n otherwise $T_n \geq T_{i+1}$. Therefore T'_{i+1} contains some statements of T_n.

T'_{i+1} may contain some statements which are implied from within T_{i+1}. Let these implied statements form the set Q. Let $N = T'_{i+1} - Q$. N must contain statements in T_n otherwise every statement in T_n is implied by T_{i+1}. That is $T_n \geq T_{i+1}$ which is a contradiction.

Thus, if an event, E, is generated such that T_n is true but no statement in N is true (denoted by $T_n \sim N$) then E cannot belong to T_n because some statement in T_n has been made false. Thus if the set A is generated it is possible to guarantee under any circumstances that the example shown will be a useful one.

Marvin uses a method of creating examples which avoids generating every statement in A. However, this is done at some cost, as we will see. When a statement is removed from the trial it must be falsified by the object construction procedure, *experiment*, unless that statement is implied by statements still in the trial. If the latter is true, it would be impossible to falsify. For example, if any-shape(X) has been removed, this cannot be falsified if flat(X) is still in the trial.

Suppose that a consistent trial, T_i, is generalized to an inconsistent one T_{i+1} by adding a statement which refers to the concept, P. Let us further suppose that P consists of three disjuncts D_1, D_2, D_3. D_1 is the disjunct which matches a subset, M, of T_i. The problem is, will falsifying M produce a useful training example?

If M must be false then the object construction procedure will choose an object which satisfies either D_2 or D_3 in P. Since T_{i+1} is inconsistent, one of these disjuncts must make T_{i+1} describe an event which will be outside T_n. Let us assume that D_2 is that disjunct and D_3 produces an event in T_n. If the object construction procedure chooses D_3 then the training example will not be acceptable.

To avoid this problem, we insist that Marvin's memory must be partially ordered according to the following criteria:

1. If C_1 and C_2 are two concepts in memory and $C_1 \geq C_2$ then C_1 must contain a reference to C_2.
2. If there are two concepts C_1 and C_2 such that:

$$\text{cover}(C_1) \cap \text{cover}(C_2) \neq \emptyset$$

and there is a third concept C_3 such that

$$\text{cover}(C_3) = \text{cover}(C_1) \cap \text{cover}(C_2)$$

then C_1 and C_2 must refer to C_3. In other words, if two concepts describe a common set of instances then, this intersection should be separated out as a third concept referred to by both.

Concepts are stored in a hierarchy with the more general concepts at the top. If $C_1 \geq C_2$ then since C_1 must contain a reference to C_2, C_1 will not be used in a generalization until a statement referring to C_2 has been introduced into the trial. [This is so because all of the statements in a conjunction must be matched before a replacement is made and one of those statements must look like, $C_2(x, y, z, ...)$]. Thus the learning algorithm uses a specific-to-general search strategy.

In the example above, if $T_n \geq D_3$ then the second ordering requirement of the memory would be violated. The trainer must teach D_3 as a separate concept first since it is common to P and the target. If this is done then Marvin will be able to generate correct training examples by ensuring that only the statements removed so far are false. There is no need to generate all the statements in A provided that the memory is 'well structured'.

The method used here places the responsibility of maintaining a well ordered memory on the trainer. This is necessary only if we do not wish to generate the set A. In Chapter 7 a method is proposed that will allow Marvin to maintain the ordering automatically.

The procedure **experiment** creates training examples:

```
experiment(Trial)
    Simplify Trial
    F:= statements to be made false
    repeat
        Generate a new training instance by proving Trial
    until all the statements in F are false
    return last instance generated so that it can be shown to the trainer
```

Since the proof procedure used to generate objects is fairly primitive, it is easy to create concepts for which the proof will not terminate. To avoid this some pre-processing has to be done. The Trial is simplified by removing redundant statements. For example if one statement, S_1, implies another, S_2, then there is no need to prove S_2 since S_2 must be true anyway.

The program must show an example which has not been seen before. To do this, it chooses statements from among those that have been removed and makes sure that any object it produces does *not* satisfy the statements in F.

4
A Complete Example

Many of the ideas introduced in Chapter 3 may become clearer by observing Marvin performing a complex learning task.

We have already seen that Marvin has no knowledge of numbers. A character string such as '12' is no more than an identifier, it receives no special interpretation. Therefore, if we want to teach concepts which require a knowledge of numbers, then first we have to teach Marvin about numbers. This process will be described in this chapter. First Marvin will need to know how to represent numbers, later it will learn to compare them using the 'less than' relation and finally, Marvin will learn how to find the maximum number in a list.

A binary number is a string of digits such as, 100110. Leading zeros will not be allowed, so the number zero itself will not be allowed. The string '100110' may be represented as the left recursive binary tree shown in Figure 4.1.

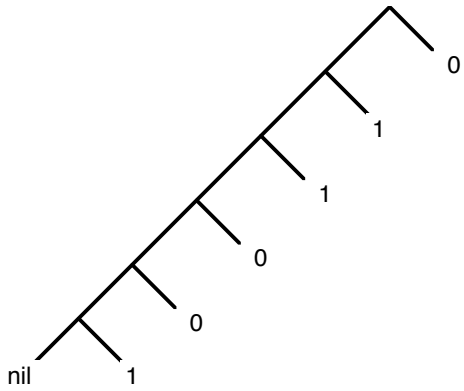


Figure 4.1. Representation of 100110

In fact, numbers as presented to Marvin are objects of the form <left: X; right: Y> where X is another number and Y is a digit. The number 'one' is

<left: nil; tail: <val: 1>>

where the left hand side of the tree is nil and the right hand side has a value of 1.

The remainder of this chapter consists of an annotated printout produced by Marvin. Marvin is instructed to 'learn numbers'. 'Numbers' is the name of a file which contains the definitions of objects which the trainer will show as training examples. For example, d0, is the name of the digit 0. 'Two' is the name of the object representing 2. 'e1' .. 'e4' are the names of lists which will be used to teach Marvin about 'Maximum'. The contents of 'numbers' is:

```
d0 = <val: 0>
d1 = <val: 1>
one = <left: nil; right: d1>
two = <left: one; right: d0>
three = <left: one; right: d1>
four = <left: two; right: d0>
five = <left: two; right: d1>
six = <left: three; right: d0>
seven = <left: three; right: d1>
e1 = <head: one; tail: none>
e2 = <head: two; tail: e1>
e3 = <head: two; tail: none>
e4 = <head: one; tail: e3>
```

What is the name of the concept? *digit*

Marvin prompts the trainer, asking the name of the concept to be learned. Its first task is to learn what are valid binary digits. At this stage the memory is completely empty; no concepts are known.

```
Show me an example of digit: (d0)
This disjunct is:
        X0.val = 0
```

When the digit d0 is shown, Marvin remembers it without questioning the trainer at all. Since there is nothing in memory that it can refer to, no generalizations can take place, so it learned by rote.

```
Show me an example of digit: (d1)
This disjunct is:
        X0.val = 1
```

Similarly, the description of d1 does not match anything in memory so it is also remembered without question.

```
Show me an example of digit: no
```

Since 0 and 1 are the only binary digits, the trainer refuses to show any further examples. At this point Marvin displays the concept it has learned.

```
Description of digit is:
[X0:
        X0.val = 0
v
        X0.val = 1
]

Learned in  0.03 secs
```

Note that one disjunct has been created for each example shown. The description states that a digit has the value 0 or the value 1.

```
Do you want to teach me another concept? yes
What is the name of the concept? number
Show me an example of number: (one)
```

Again Marvin prompts the trainer asking to learn more. This time the concept 'binary number' is to be learned. The first example is the number 'one': <left: nil; right: d1>

```
Testing:
        X0.left = nil
        X0.right = X1
==>     X1.val = 1
        digit(X1)
```

Marvin first generates all the primary statements, that is, the statements which exactly describe the example. It then attempts to match the statements of the the first trial concept with the statements of concepts that are in memory. In this case 'X1.val = 1' matches the second disjunct of *digit*. The program replaces X1.val = 1 by the new statement 'digit(X1),' thus producing a generalization of the first trial. The '==>' preceding a statement indicates that it has been removed from the trial.

```
Is
X0 = <left: nil; right: <val: 0>>
```

recognized by the concept? *no*

In order to test its hypothesis, Marvin uses the trial concept to generate its own example to show the trainer. Thus it shows,

<left: nil; right: <val: 0>>

In order to create this object, Marvin ensured that the removed statement was false. That is, no object whose right part was 1 would be allowed. Since, for this example, we do not wish Marvin to know about zero, the trainer answers that this is not a valid example of number. The program then attempts to find some way of making the trial more specific. In this case, the only way this can be done is by abandoning its generalization. Thus 'digit(X1)' is removed and 'X1.val = 1' is restored.

```
This disjunct is:
        X0.left = nil
        X0.right = X1
        X1.val = 1
```

Since no more generalizations can be made, Marvin enters this description as the first disjunct of number.

Show me an example of number: *(two)*

To learn the second disjunct the trainer shows Marvin the number 'two'.

```
Testing:
        X0.left = X1
==>     X1.left = nil
==>     X1.right = X2
==>     X2.val = 1
        X0.right = X3
        X3.val = 0
        number(X1)
```

The program can match three of the primary statements with the first disjunct of number, so they are removed and replaced by 'number(X1).' This illustrates the importance of the ordering of examples shown by the trainer. It is essential that 'one' be shown first, otherwise the recursive nature of *number* would not be apparent.

Is
X0 = <left: <left: <left: nil; right: <val: 1>>; right: <val: 0>>; right: <val: 0>>

recognized by the concept? *yes*

The new trial is used to show the trainer an example of Marvin's current idea of what a number should look like. Since the left part of X0 cannot be the number 'one' (this is precluded by insisting that the removed statements are false) *number* is called recursively to create the number 4. Note that Marvin has never seen a 4 and yet it can still create one.

```
Testing:
        X0.left = X1
==>     X1.left = nil
==>     X1.right = X2
==>     X2.val = 1
        X0.right = X3
==>     X3.val = 0
        number(X1)
        digit(X3)
```

Since 4 is an acceptable number Marvin can continue to generalize the trial. It matches 'X3.val = 0' with the first disjunct of digit and makes the generalization digit(X3).

```
Is
X0 = <left: <left: <left: nil; right: <val: 1>>; right: <val: 0>>;  right: <val: 1>>

recognized by the concept? yes
```

The new trial also produces a valid number, 5.

```
This disjunct is:
        X0.left = X1
        X0.right = X3
        number(X1)
        digit(X3)
```

Since Marvin cannot make any more generalizations, it concludes that it has learned the second disjunct.

```
Show me an example of number: no
```

The trainer indicates that there are no more disjuncts to learn.

```
Description of number is:
[X0:
        [∃ X1:
                X0.left = nil
                X0.right = X1
                X1.val = 1
        ]
v
        [∃ X1, X3:
                X0.left = X1
                X0.right = X3
                number(X1)
                digit(X3)
        ]
]

Learned in  0.25 secs
```

The final description of 'binary number' is: X0 is a number if its left part is nil and its right part is 1 or the left is a number and the right is a digit.

```
Do you want to teach me another concept? yes
What is the name of the concept? lessd
Show me an example of lessd: (d0, d1)
```

Before we can teach Marvin how to recognize that one number is less than another, we must first teach it the ordering among digits. The purpose of learning *lessd* is simply to tell Marvin that 0 comes before 1.

```
Testing:
==>     X0.val = 0
        X1.val = 1
        digit(X0)
Is
X0 = <val: 1>
```

```
X1 = <val: 1>

recognized by the concept? no
Testing:
        X0.val = 0
==>     X1.val = 1
==>     digit(X0)
        digit(X1)
Is
X0 = <val: 0>
X1 = <val: 0>

recognized by the concept? no
```

Since Marvin has seen digits before, it tries to generalize *lessd*. However, the trainer answers *no* to both generalizations.

```
This disjunct is:
        X0.val = 0
        X1.val = 1

Show me an example of lessd: no
Description of lessd is:
[X0, X1:
        X0.val = 0
        X1.val = 1
]

Learned in  0.08 secs
```

Marvin learns that if there are two objects and the first one is the digit 0 and the second is the digit 1 then this is a *lessd* event.

```
Do you want to teach me another concept? yes
What is the name of the concept? less
Show me an example of less: (two, three)
```

Now Marvin is ready to learn *less*. The first example it will be shown is the pair (2, 3).

```
Testing:
        X0.left = X2
==>     X2.left = nil
==>     X2.right = X3
        X3.val = 1
        X0.right = X4
        X4.val = 0
        X1.left = X2
        X1.right = X3
        number(X2)
```

The first generalization illustrates an important feature. Although three statements including 'X3.val = 1' matched the first disjunct of number, only two were removed. The digit 1 occurs elsewhere in the description of (2, 3). It is referred to by the statement X1.right = X3. Therefore, a description of X3 must remain in the trial. However, this has not prevented the description of X2, the left part of X0, from being generalized.

```
Is
```

```
X0 = <left: <left: <left: nil; right: <val: 1>>; right: <val: 0>>;  right: <val: 0>>
X1 = <left: <left: <left: nil; right: <val: 1>>; right: <val: 0>>; right: <val: 1>>

recognized by the concept? yes
```

The program shows the trainer the pair (4, 5) which is a valid example of less. So Marvin knows that the generalization was consistent.

```
Testing:
        X0.left = X2
==>     X2.left = nil
==>     X2.right = X3
==>     X3.val = 1
        X0.right = X4
        X4.val = 0
        X1.left = X2
        X1.right = X3
        number(X2)
        digit(X3)
Is
X0 = <left: <left: <left: nil; right: <val: 1>>; right: <val: 1>>; right: <val: 0>>
X1 = <left: <left: <left: nil; right: <val: 1>>; right: <val: 1>>;  right: <val: 0>>

recognized by the concept? no
```

However, the next generalization that it makes is not. When X3 is generalized to any digit, the program construct the pair (6, 6). This is obviously not an instance of *less* so Marvin must make the trial more specific.

```
Testing:
        X0.left = X2
==>     X2.left = nil
==>     X2.right = X3
==>     X3.val = 1
        X0.right = X4
==>     X4.val = 0
        X1.left = X2
        X1.right = X3
        number(X2)
        digit(X3)
        lessd(X4, X3)
```

Marvin created the statement 'digit(X3)' when it was examining X3.val = 1. That is, a call TryConceptsWith(X3.val = 1) was executed. There are two concepts with matches for this statement: *digit* and *lessd*. If there were no preferred order Marvin might have tried lessd first. However, since lessd is conjunctive, replacing its implicants by a statement lessd(X4, X3) would not have created a trial that was a proper generalization of the old one. Thus lessd was ignored.

Now we see that lessd does have some use. Since the introduction of digit(X3) made the trial inconsistent, Marvin looks at the implicant of digit(X3) in order to make a trial which is more specific. Lessd may now be used for that purpose.

```
Is
X0 = <left: <left: <left: nil; right: <val: 1>>; right: <val: 0>>;  right: <val: 0>>
X1 = <left: <left: <left: nil; right: <val: 1>>; right: <val: 0>>; right: <val: 1>>

recognized by the concept? yes
```

In fact the trial is now consistent because Marvin was able to show (4, 5) which is an instance of less.

Note that normally Marvin would not ask about (4, 5) again because it can remember the instances it has generated before. This part of its memory has been switched off for this demonstration.

```
Testing:
        X0.left = X2
==>     X2.left = nil
==>     X2.right = X3
==>     X3.val = 1
        X0.right = X4
==>     X4.val = 0
==>     X1.left = X2
==>     X1.right = X3
        number(X2)
==>     digit(X3)
        lessd(X4, X3)
        number(X1)
Is
X0 = <left: <left: <left: nil; right: <val: 1>>; right: <val: 0>>; right: <val: 0>>
X1 = <left: <left: nil; right: <val: 1>>; right: <val: 0>>

recognized by the concept? no
```

A further generalization is made. This creates the instance (4, 2). Up till now it has been possible to ensure that all removed statements are false when an instance is constructed. However, here we have a case where 'digit(X3)' has been removed but lessd(X4, X3) insists that X3 is, in fact, a digit. A removed statement can only be falsified if it is not implied by statements in the trial. (4, 2) indicates that the trial is inconsistent again, since the second element of the pair cannot be just any number.

```
This disjunct is:
        X0.left = X2
        X0.right = X4
        X1.left = X2
        X1.right = X3
        number(X2)
        lessd(X4, X3)
```

The inconsistent trial cannot be made more specific without going back to the previous trial, and no more generalizations can be made, so the process ends for the first disjunct of *less*. The description states that if two numbers have the same left part, but the right part of the first number is *lessd* than the right part of the second then *less* is true.

Show me an example of less: *(five, six)*

To teach Marvin the second disjunct, the trainer shows the pair (5, 6).

```
Testing:
        X0.left = X2
        X2.left = X3
==>     X3.left = nil
==>     X3.right = X4
        X4.val = 1
        X2.right = X5
        X5.val = 0
        X0.right = X4
        X1.left = X6
        X6.left = X3
        X6.right = X4
        X1.right = X5
        number(X3)
```

```
Is
X0 = <left:
        <left: <left: <left: nil; right: <val: 1>>;  right: <val: 0>>;  right: <val: 0>>;
       right: <val: 1>>
X1 = <left:
        <left: <left: <left: nil; right: <val: 1>>; right: <val: 0>>;  right: <val: 1>>;
       right: <val: 0>>

recognized by the concept? yes
```

The first generalization creates a concept which construct the pair (9, 10), thus it is consistent.

```
Testing:
        X0.left = X2
        X2.left = X3
==>     X3.left = nil
==>     X3.right = X4
==>     X4.val = 1
        X2.right = X5
        X5.val = 0
        X0.right = X4
        X1.left = X6
        X6.left = X3
        X6.right = X4
        X1.right = X5
        number(X3)
        digit(X4)
Is
X0 = <left:
        <left: <left: <left: nil; right: <val: 1>>;  right: <val: 1>>;  right: <val: 0>>;
       right: <val: 0>>
X1 = <left:
        <left: <left: <left: nil; right: <val: 1>>;  right: <val: 1>>;  right: <val: 0>>;
       right: <val: 0>>

recognized by the concept? no
```

The next trial constructs (12, 12) which is not consistent and must be made more specific just as was done in the first disjunct.

```
Testing:
        X0.left = X2
        X2.left = X3
==>     X3.left = nil
==>     X3.right = X4
==>     X4.val = 1
        X2.right = X5
==>     X5.val = 0
        X0.right = X4
        X1.left = X6
        X6.left = X3
        X6.right = X4
        X1.right = X5
        number(X3)
        digit(X4)
        lessd(X5, X4)

Is
```

```
X0 = <left:
        <left: <left: <left: nil; right: <val: 1>>;  right: <val: 0>>;  right: <val: 0>>;
       right: <val: 1>>
X1 = <left:
        <left: <left: <left: nil; right: <val: 1>>;  right: <val: 0>>;  right: <val: 1>>;
       right: <val: 0>>

recognized by the concept? yes
```

The new trial constructs (9, 10) again.

```
Testing:
        X0.left = X2
==>     X2.left = X3
==>     X3.left = nil
==>     X3.right = X4
==>     X4.val = 1
==>     X2.right = X5
==>     X5.val = 0
        X0.right = X4
        X1.left = X6
==>     X6.left = X3
==>     X6.right = X4
        X1.right = X5
==>     number(X3)
        digit(X4)
        lessd(X5, X4)
        less(X2, X6)

Is
X0 = <left:
        <left: <left: <left: nil; right: <val: 1>>;  right: <val: 0>>;  right: <val: 1>>;
       right: <val: 1>>
X1 = <left:
        <left: <left: <left: nil; right: <val: 1>>;  right: <val: 1>>; right: <val: 0>>;
       right: <val: 0>>

recognized by the concept? yes
```

The next trial is generated by a replacement which adds a recursive call to less. This is consistent since the instance shown is (11, 12). Note that 'number(X3)' was eliminated by this replacement. Statements which are inferred from the primaries can also take part in statement matching.

```
Testing:
        X0.left = X2
==>     X2.left = X3
==>     X3.left = nil
==>     X3.right = X4
==>     X4.val = 1
==>     X2.right = X5
==>     X5.val = 0
        X0.right = X4
        X1.left = X6
==>     X6.left = X3
==>     X6.right = X4
        X1.right = X5
==>     number(X3)
        digit(X4)
```

```
==>     lessd(X5, X4)
        less(X2, X6)
        digit(X5)
```

The replacement which added 'less(X2, X6)' to the trial could not remove 'lessd(X5, X4)' because X5 is referred to elsewhere in the concept so some description of X5 is required. However, it may now be possible to relax the restriction on X5 so Marvin tries replacing 'lessd(X5, X4)' in favour of 'digit(X5).'

```
Is
X0 = <left: <left: <left: nil; right: <val: 1>>; right: <val: 0>>;  right: <val: 0>>
X1 = <left: <left: <left: nil; right: <val: 1>>; right: <val: 1>>; right: <val: 1>>

recognized by the concept? yes
```

Since (4, 7) was constructed this relaxation was a good generalization.

```
MORE SPECIFIC WITH number(X6)
MORE SPECIFIC WITH number(X2)
Testing:
        X0.left = X2
==>     X2.left = X3
==>     X3.left = nil
==>     X3.right = X4
==>     X4.val = 1
==>     X2.right = X5
==>     X5.val = 0
        X0.right = X4
==>     X1.left = X6
==>     X6.left = X3
==>     X6.right = X4
==>     X1.right = X5
==>     number(X3)
        digit(X4)
==>     lessd(X5, X4)
        less(X2, X6)
==>     digit(X5)
==>     number(X6)
==>     number(X2)
        number(X1)
```

As Marvin continues to generalize, it deduces that X6 and X2 are numbers. However, neither statement can be used to generalize the trial. This is because all their implicants either have been removed already or they cannot be removed at all. Since no statement can be removed, the addition of statements could not produce a proper generalization and may even make the trial more specific. Thus 'number(X6)' and 'number(X2)' are flagged as out of the concept immediately.

```
Is
X0 = <left: <left: <left: nil; right: <val: 1>>; right: <val: 0>>; right: <val: 0>>
X1 = <left: nil; right: <val: 1>>

recognized by the concept? no
```

'Number(X1)' does make a more general trial. This time it is inconsistent, constructing (4, 1).

```
This disjunct is:
        X0.left = X2
        X0.right = X4
        X1.left = X6
```

```
        X1.right = X5
        digit(X4)
        less(X2, X6)
        digit(X5)
```

With 'number(X1)' in the trial, no consistent generalizations can be made, so 'number(X1)' must be removed. In fact, no more generalizations are possible even after removing the statement. The second disjunct is complete: If the left part of the first number is less than the left part of the second, and both the right parts are digits, then the pair is an instance of *less*.

```
Show me an example of less: (one, two)
```

The two disjuncts of *less* learned so far cover all the possibilities except (1, 2). This is the example which the trainer shows Marvin to complete the description of *less*

```
Testing:
==>     X0.left = nil
==>     X0.right = X2
==>     X2.val = 1
        X1.left = X0
        X1.right = X3
        X3.val = 0
        number(X0)
Is
X0 = <left: <left: nil; right: <val: 1>>; right: <val: 0>>
X1 = <left: <left: <left: nil; right: <val: 1>>; right: <val: 0>>;  right: <val: 0>>

recognized by the concept? yes
```

The first trial constructs (2, 4) which is valid.

```
Testing:
==>     X0.left = nil
==>     X0.right = X2
==>     X2.val = 1
        X1.left = X0
        X1.right = X3
==>     X3.val = 0
        number(X0)
        digit(X3)
Is
X0 = <left: <left: nil; right: <val: 1>>; right: <val: 0>>
X1 = <left: <left: <left: nil; right: <val: 1>>; right: <val: 0>>;  right: <val: 1>>

recognized by the concept? yes
```

the second trial constructs (2, 5) which is also valid.

```
Testing:
==>     X0.left = nil
==>     X0.right = X2
==>     X2.val = 1
==>     X1.left = X0
==>     X1.right = X3
==>     X3.val = 0
        number(X0)
==>     digit(X3)
        number(X1)
```

```
Is
X0 = <left: <left: nil; right: <val: 1>>; right: <val: 0>>
X1 = <left: nil; right: <val: 1>>

recognized by the concept? no
```

The last generalization is inconsistent since it constructs (2, 1).

```
This disjunct is:
        X1.left = X0
        X1.right = X3
        number(X0)
        digit(X3)
```

The final description of thus disjunct states that if X1 = 2*X0 or X1 = 2*X0+1 then *less* is true. This is more general than strictly necessary, but it does the job.

```
Show me an example of less: no

Description of less is:
[X0, X1:
        [∃ X2, X3, X4:
                X0.left = X2
                X0.right = X4
                X1.left = X2
                X1.right = X3
                number(X2)
                lessd(X4, X3)
        ]
v
        [∃ X2, X4, X5, X6:
                X0.left = X2
                X0.right = X4
                X1.left = X6
                X1.right = X5
                digit(X4)
                less(X2, X6)
                digit(X5)
        ]
v
        [∃ X3:
                X1.left = X0
                X1.right = X3
                number(X0)
                digit(X3)
        ]
]

Learned in  4.80 secs

Do you want to teach me another concept? yes
What is the name of the concept? max
```

The trainer now tells Marvin that it will learn *max*. *max(X, Y)* is true if X is a list of numbers and Y is the largest number in X

```
Show me an example of max: (e1, one)
```

The first example shown by the trainer teaches Marvin about single element lists. The example is ([1], 1). For convenience lists will be enclosed in square brackets (in the text only). The lists are presented to Marvin as objects with properties 'head' and 'tail' corresponding to 'car' and 'cdr' in LISP.

```
Testing:
        X0.head = X1
        X0.tail = none
==>     X1.left = nil
==>     X1.right = X2
==>     X2.val = 1
        number(X1)
Is
X0 = <head: <left: <left: nil; right: <val: 1>>; right: <val: 0>>;  tail: none>
X1 = <left: <left: nil; right: <val: 1>>; right: <val: 0>>

recognized by the concept? yes
This disjunct is:
        X0.head = X1
        X0.tail = none
        number(X1)
```

Marvin shows ([2], 2) and thus has learned that if X1 is the only number in X2 then *max* is true.

```
Show me an example of max: (e2, two)
```

The next example shown by the trainer is a pair ([2, 1], 2) in which the head of the list is the maximum.

```
Testing:
        X0.head = X1
        X0.tail = X2
        X2.head = X3
==>     X3.left = nil
==>     X3.right = X4
==>     X4.val = 1
        X2.tail = none
        X1.left = X3
        X1.right = X5
        X5.val = 0
        number(X3)
Is
X0 = <head:
        <left: <left: <left: nil; right: <val: 1>>; right: <val: 0>>;  right: <val: 0>>;
        tail:
        <head: <left: <left: nil; right: <val: 1>>; right: <val: 0>>;
          tail: none>>
X1 = <left: <left: <left: nil; right: <val: 1>>;  right: <val: 0>>; right: <val: 0>>

recognized by the concept? yes
```

Marvin proceeds as before, making a generalization and showing an instance ([4, 2], 4) which is valid.

```
Testing:
        X0.head = X1
        X0.tail = X2
==>     X2.head = X3
==>     X3.left = nil
==>     X3.right = X4
```

```
==>     X4.val = 1
==>     X2.tail = none
        X1.left = X3
        X1.right = X5
        X5.val = 0
==>     number(X3)
        max(X2, X3)

Is
X0 = <head:
        <left: <left: <left: nil; right: <val: 1>>;  right: <val: 0>>;  right: <val: 0>>;
        tail:
        <head: <left: <left: nil; right: <val: 1>>; right: <val: 0>>;
          tail: <head: <left: nil; right: <val: 1>>; tail: none>>>
X1 = <left: <left: <left: nil; right: <val: 1>>; right: <val: 0>>; right: <val: 0>>

recognized by the concept? yes
```

The next instance shown is ([4, 2, 1], 4). At this point a recursive reference to *max* has been introduced.

```
Testing:
        X0.head = X1
        X0.tail = X2
==>     X2.head = X3
==>     X3.left = nil
==>     X3.right = X4
==>     X4.val = 1
==>     X2.tail = none
        X1.left = X3
        X1.right = X5
==>     X5.val = 0
==>     number(X3)
        max(X2, X3)
        digit(X5)
Is
X0 = <head:
        <left: <left: <left: nil; right: <val: 1>>; right: <val: 0>>;  right: <val: 1>>;
        tail:
        <head:  <left: <left: nil; right: <val: 1>>;  right: <val: 0>>;
          tail: <head: <left: nil; right: <val: 1>>; tail: none>>>
X1 = <left: <left: <left: nil; right: <val: 1>>; right: <val: 0>>; right: <val: 1>>

recognized by the concept? yes
```

([5, 2, 1], 5) is the next instance. Marvin is still creating consistent generalizations.

```
Testing:
        X0.head = X1
        X0.tail = X2
==>     X2.head = X3
==>     X3.left = nil
==>     X3.right = X4
==>     X4.val = 1
==>     X2.tail = none
==>     X1.left = X3
==>     X1.right = X5
==>     X5.val = 0
==>     number(X3)
```

```
            max(X2, X3)
==>         digit(X5)
            number(X1)
Is
X0 = <head: <left: nil; right: <val: 1>>;
            tail:
            <head: <left: <left: nil; right: <val: 1>>; right: <val: 0>>;
               tail: <head: <left: nil; right: <val: 1>>; tail: none>>>
X1 = <left: nil; right: <val: 1>>

recognized by the concept? no
```

It finally goes too far with ([1, 2, 1], 1). Now the concept must be made more specific.

```
MORE SPECIFIC WITH less(X3, X1)
Testing:
            X0.head = X1
            X0.tail = X2
==>         X2.head = X3
==>         X3.left = nil
==>         X3.right = X4
==>         X4.val = 1
==>         X2.tail = none
==>         X1.left = X3
==>         X1.right = X5
==>         X5.val = 0
==>         number(X3)
            max(X2, X3)
==>         digit(X5)
            number(X1)
            less(X3, X1)
Is
X0 = <head:
            <left: <left: nil; right: <val: 1>>; right: <val: 1>>;
            tail:
            <head: <left: <left: nil; right: <val: 1>>;  right: <val: 0>>;
               tail: <head: <left: nil; right: <val: 1>>; tail: none>>>
X1 = <left: <left: nil; right: <val: 1>>; right: <val: 1>>

recognized by the concept? yes
```

When Marvin learned *less*, some replacements were not attempted because they would not result in proper generalizations. The introduction of 'less(X3, X1)' does not remove any of its implicants either, however since we are now trying to restrict the trial, that doesn't matter.

```
This disjunct is:
            X0.head = X1
            X0.tail = X2
            max(X2, X3)
            less(X3, X1)
```

With the addition of 'less(X3, X1)', Marvin has learned that if the head of the list is greater than the maximum of the tail then the maximum of the whole list is the head.

Until now, all the existentially quantified variables could be eliminated by changing pairs of statements such as

$$X0.right = X4 \wedge X4.val = 1$$

into

$$X3.right.val = 1$$

However, this disjunct of *max* uses the first genuinely quantified variable, X3.

```
Show me an example of max: (e4, two)
```

To teach the next disjunct of *max* the trainer shows Marvin ([1, 2], 2). If the head of the list is less than the maximum of the tail then the maximum of the whole list is the maximum of the tail.

```
Testing:
          X0.head = X2
==>       X2.left = nil
==>       X2.right = X3
==>       X3.val = 1
          X0.tail = X4
          X4.head = X1
          X4.tail = none
          X1.left = X2
          X1.right = X5
          X5.val = 0
          number(X2)
Is
X0 = <head:
          <left: <left: nil; right: <val: 1>>;  right: <val: 0>>;
          tail:
          <head:
             <left: <left: <left: nil; right: <val: 1>>;  right: <val: 0>>;  right: <val: 0>>;
           tail: none>>
X1 = <left: <left: <left: nil; right: <val: 1>>; right: <val: 0>>; right: <val: 0>>

recognized by the concept? yes
```

([2, 4], 4) is the first instance constructed.

```
Testing:
          X0.head = X2
==>       X2.left = nil
==>       X2.right = X3
==>       X3.val = 1
          X0.tail = X4
          X4.head = X1
          X4.tail = none
          X1.left = X2
          X1.right = X5
==>       X5.val = 0
          number(X2)
          digit(X5)
Is
X0 = <head:
          <left: <left: nil; right: <val: 1>>; right: <val: 0>>;
          tail:
          <head:
                    <left: <left: <left: nil; right: <val: 1>>; right: <val: 0>>;  right: <val: 1>>;
            tail: none>>
X1 = <left: <left: <left: nil; right: <val: 1>>; right: <val: 0>>;  right: <val: 1>>

recognized by the concept? yes
```

([2, 5], 5) is the next instance.

```
Testing:
          X0.head = X2
==>       X2.left = nil
==>       X2.right = X3
==>       X3.val = 1
          X0.tail = X4
          X4.head = X1
          X4.tail = none
==>       X1.left = X2
==>       X1.right = X5
==>       X5.val = 0
          number(X2)
==>       digit(X5)
          number(X1)
Is
X0 = <head:
          <left: <left: nil; right: <val: 1>>; right: <val: 0>>;
          tail:
          <head: <left: nil; right: <val: 1>>;
            tail: none>>
X1 = <left: nil; right: <val: 1>>

recognized by the concept? no
```

When Marvin shows ([2, 1], 1), the program has created an inconsistent generalization by introducing 'number(X1)'.

```
Testing:
          X0.head = X2
==>       X2.left = nil
==>       X2.right = X3
==>       X3.val = 1
          X0.tail = X4
          X4.head = X1
          X4.tail = none
==>       X1.left = X2
==>       X1.right = X5
==>       X5.val = 0
==>       number(X2)
==>       digit(X5)
          number(X1)
          less(X2, X1)
Is
X0 = <head:
          <left: <left: nil; right: <val: 1>>;  right: <val: 0>>;
          tail:
          <head:
                    <left: <left: nil; right: <val: 1>>;  right: <val: 1>>;
            tail: none>>
X1 = <left: <left: nil; right: <val: 1>>; right: <val: 1>>

recognized by the concept? yes
```

It qualifies this generalization by adding less(X2, X1). The instance ([2, 3], 3) indicates that the trial is consistent once again.

```
Testing:
```

```
            X0.head = X2
==>         X2.left = nil
==>         X2.right = X3
==>         X3.val = 1
            X0.tail = X4
==>         X4.head = X1
==>         X4.tail = none
==>         X1.left = X2
==>         X1.right = X5
==>         X5.val = 0
==>         number(X2)
==>         digit(X5)
==>         number(X1)
            less(X2, X1)
            max(X4, X1)
Is
X0 = <head:
            <left: <left: nil; right: <val: 1>>; right: <val: 0>>;
      tail:
            <head:
                        <left: <left: nil; right: <val: 1>>; right: <val: 1>>;
              tail:
                        <head: <left: nil; right: <val: 1>>;
                          tail: none>>>
X1 = <left: <left: nil; right: <val: 1>>; right: <val: 1>>

recognized by the concept? yes
```

The trial is generalized further with max(X4, X1). The instance constructed is ([2, 3, 1], 3) which is valid.

```
This disjunct is:
            X0.head = X2
            X0.tail = X4
            less(X2, X1)
            max(X4, X1)

Show me an example of max: no
Description of max is:
[X0, X1:
            X0.head = X1
            X0.tail = none
            number(X1)
v
            [∃ X2, X3:
                        X0.head = X1
                        X0.tail = X2
                        max(X2, X3)
                        less(X3, X1)
            ]
v
            [∃ X2, X4:
                        X0.head = X2
                        X0.tail = X4
                        less(X2, X1)
                        max(X4, X1)
            ]
]
```

```
Learned in  5.83 secs
Do you want to teach me another concept? no
End of run
```

In fact the target has been reached.

A fourth disjunct is necessary to say that if the head is equal to the maximum of the tail then the maximum is the head (or the maximum of the tail). This is not shown here since the learning sequence is much the same as the other disjuncts of *max*.

5
A Tour Through Marvin

Marvin is the result of several stages of evolution. Originally, the program was written in Prolog, but the latest version, which is described in this chapter, is implemented in Pascal. Currently Marvin runs on a VAX 11/780 - UNIX system. The entire program, including data areas runs in about 80K bytes and the source consists of approximately 2200 lines of Pascal code.

At its highest level Marvin looks like this:

```
Marvin:
        look at world;
        repeat
                learn new concept
                remember concept
                ask trainer  'Do you want to teach me another concept?'
        until answer = no
```

Marvin's world is a file containing the descriptions of all the objects it can see during one training session. The world may also contain the definitions of concepts which have been learned in previous sessions. Marvin begins by reading the world file and then it repeatedly asks the trainer to teach it new concepts based on the objects it can see. Once a concept is learned, it is stored in an associative memory.

5.1 Learning Disjunctive Concepts

Marvin learns disjunctive concepts by learning one conjunction at a time. It begins by asking trainer to show an example of the concept to be learned. This example is used to learn one conjunction of the entire concept. As we have seen, a sequence of trial concepts is generated and tested until the target is reached. When it has finished, Marvin assumes that one conjunction has been learned and asks the trainer to show it a new example so that the program can learn another conjunction. This can be summarized as follows:

```
learn:
        ask  'What is the name of the concept? '
        read conceptname
        Look up name in program's dictionary
        repeat
                ask  'Show me an example of the concept'
                read example
                if example = no  then
                        the complete concept has been learned
                else LearnConjunction(conceptname, example)
        until complete concept is learned
        print description of concept
```

Marvin first asks the name of the concept. It looks up this name in a dictionary to see if the name is already known. If it is this means that the trainer has already taught Marvin part of the concept. The conjunction about to be learned will be appended to the existing description. If the name is not known then a new entry is made in the dictionary.

The dictionary is implemented as a hash table. Each entry associated with a concept name is a pair consisting of the formal parameters of the concept and its definition. An entry may be represented by a Pascal record:

```
concept    =    record
                    formal: list of variable;
                    definition: list of conjunction
                end
```

The definition is a list of conjunctions. Each new conjunction is appended to the end of the list.

The learning algorithm begins with a call to *LearnConjunction* which, as the name suggests, learns one conjunction of the concept description.

```
LearnConjunction(ConceptName, Example)
        description := primary(Example)
        Trial:= create(description)
        generalize(Trial)
        simplify(Trial)
        remember(Trial)
```

Trial and *description* are global variables, known to the whole program. The actions performed by *LearnConjunction* are:

- The primary statements are constructed from the training instance. This is done by procedure *primary* which also generates the list, *Args*, of universally quantified variables which will become the formal parameters of the concept if this is the first conjunction to be learned. The existentially quantified variables, *exvars*, are also created.
- The list of statements which form the description of the training instance is then used to *create* a conjunction which will become the initial trial.
- The trial is then *generalize*d. This procedure is the heart of the learning algorithm.
- Once the target concept has been learned, it is *simplified*, that is, redundant statements and variables are removed.
- Finally, the conjunction is *remember*ed by updating the associative memory. This involves adding the new statements in the conjunction to the *index*.

5.2 Creating Descriptions of Concepts

The first action which Marvin must perform when trying to learn a concept is to convert the description of an event into a representation that it can manipulate. That representation is first order predicate logic. Each object in the training instance is assigned a unique variable name which the system will use to refer to it. This is accomplished by the procedure *primary* which scans through the objects in a sample event and describes each in turn.

```
primary(Example)
        for each object  in Example,
                create a variable, X to represent object
                if X is identical to another object, Y  then
                        Create an identity statement X = Y
                        Add the statement to the isin list of X and Y
                else describe(X)
        return description
```

An object may appear more than once in an event. If this happens then an identity relation between the variables representing the object is created. For example, if we append a list, *L*, onto the empty list, nil, then the result is identical to *L*. A training example to teach this case of *append* may be *(nil, L, L)*. The statement *X2 = X1* will be generated to indicate that the second and third arguments are identical.

Each variable has an *isin* list which is a list of all the statements that the variable occurs in. As we will see later, this information is used by the learning algorithm. An object is described by making assertions about the values of its properties. Each *property:value* pair in the object to be described is taken in turn and a primary statement is constructed from that pair.

```
describe(ObjectName)
        for each property:value pair  in object,
                Create a statement ObjectName.property = V
                 if value is an object  then
                        if object has been described before  then
                                V = variable name given previously
                        else
                                V = new variable to represent the object
                        describe(V)
                Add the statement to the isin lists of its variables
```

Note that *describe* is recursive. If the value of an object is itself an object then the value must also be described. However, Marvin must take care not to describe the same object twice. For example, there may be two brothers:

Fred = <age: 12; father: Jack>
Bill = <age: 14: father: Jack>

and

Jack = <age: 38; wife: Jill>

To create primary statements for this example Marvin would first describe Fred. During this description, Jack would also be described.

X0.age = 12
X0.father = X2
X2.age = 38
X2.wife = X3
. . .

Bill's turn is next, but Jack has already been described so the new statements created are simply:

X1.age = 14
X1.father = X2

A list must be kept which contains the names of all the objects that have been described so far. In this way we do not created two descriptions of the same object, and in the case of two objects which refer to each other, the program does not get caught going around a circular list.

If the concept being learned is *brothers* then *X0* and *X1* become the formal parameters of the concept (i.e. the universally quantified variables) and *X2, X3* etc. become the existentially quantified variables (exvars).

The *description* of an event is a list of the statements in the current trial (as well as those that will be removed during the generalization process). A statement may be represented by the following data structure:

```
statement   =   record
                        state: integer;
                        implicants: list of statement;
                        predicate-name: word;
                        args: list of value
                end
```

Rather than physically removing statements from the description, the learning algorithm changes the *state* of the statement. The state field of a statement, S, keeps a count of the number of statements implied by S which are in the trial. If the state is 0 then S is in the trial, since it has not been replaced by any statement which it implies. If the state is a positive number then it has been replaced by at least one other statement.

Sometimes, a replacement will fail. A new statement may be introduced to replace its implicants.

However, this statement may result in a trials which can never be consistent as long as it is part of the description. When this occurs the new statement is removed. This is indicated by a state of -1. In Chapter 4 the statements which had non-zero states in the description were marked by an arrow ==>.

The implicants field is a list of the statements which imply S. Primary statements have no implicants. However, statements which are introduced by a replacement operation have, as their implicants, the statements which they replace. *Args* is the list of actual parameters which may be variables, selectors or constants. Values have type tags associated with them so that the program can determine the type of each argument. Selectors, distinguished by a *SEL* tag, have two attributes: *obj* and *prop* which indicate the the property *prop* is being selected in the object *obj*.

As an example of a statement structure, consider *append(X0.tl, X1, X2.tl)*. A graphical representation of this is shown in Figure 5.1. Primary statements have the same internal structure. For example, *X0.colour = red* is represented by the structure in Figure 5.2. Once the primary statements have been constructed, a new conjunction is created. A conjunction is represented by the following structure:

```
conjunction  =   record
                     alternatives:  list of conjunction
                     exvars:  list of variables;
                     description:  list of statement;
                 end
```

Exvars is a list of the existentially quantified variables which appear in the conjunction. *Description* is the list of statements in the conjunction and *alternatives* is the list of remaining conjunctions for the concept.

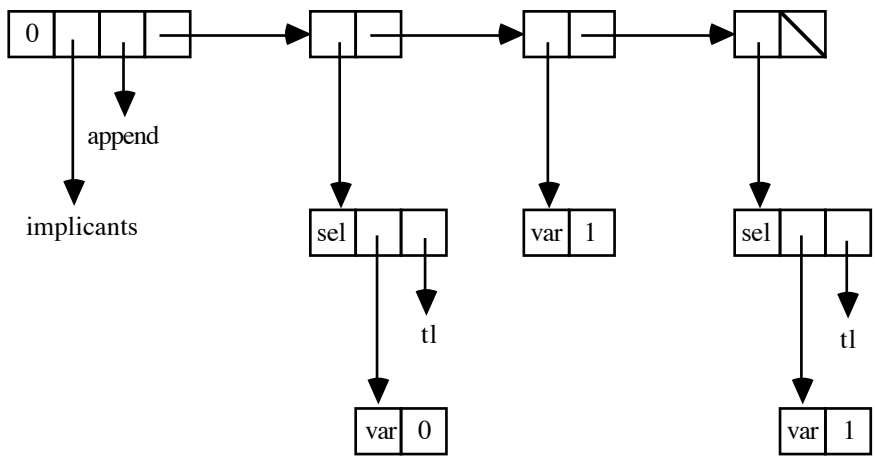


Figure 5.1. append(X0.tl, X1, X2.tl)

The procedure *create*, called by *LearnConjunction*, allocates a new record and sets *exvars* and *description* to the values returned by *primary*. Remember the initial description is the list of primary statements. The value of *alternatives* is nil until a conjunction following the present one is learned. The new conjunction is placed as the last alternative in the concept structure described earlier.

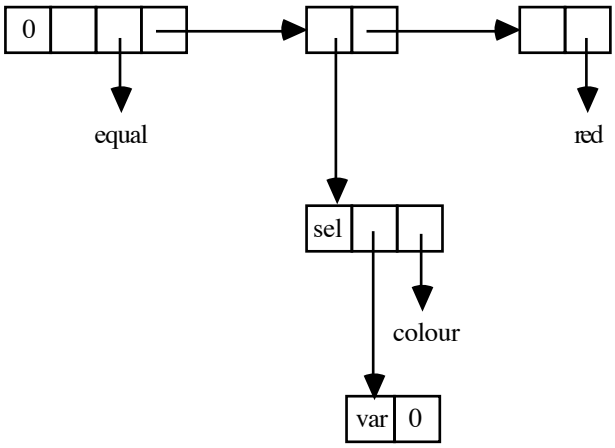


Figure 5.2. X0.colour = red

To complete this discussion of the representation of concepts, Figure 5.3 shows the entire data structure for the trial concept of *number* in Chapter 4.

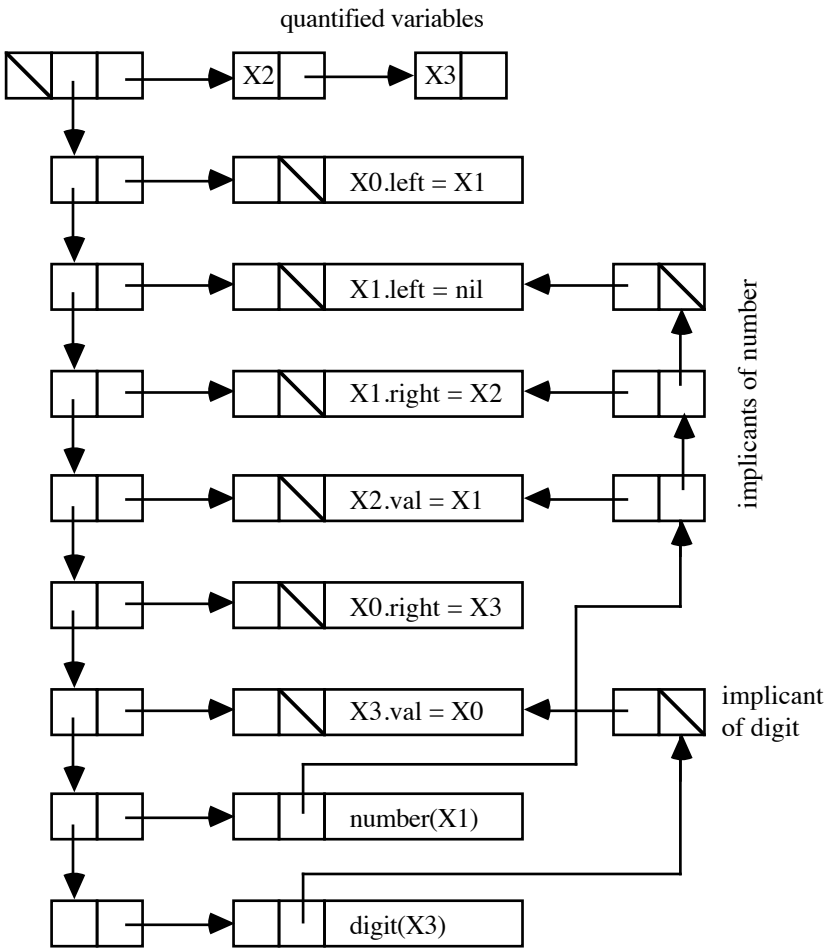


Figure 5.3. The last trial of number

5.3 Memory

The function of the memory is to enable Marvin to recognize patterns in the examples it has been shown. This is done by recalling concepts which describe a part of the world that it can see. Marvin's internal representation of the scene is the set of primary statements, T_0. In Section 3.4 we saw that if a concept, C, contains a disjunct which matches a subset of a trial T_i then C is true. Thus the main rô le of the associative memory will be to assist the generalization procedure to look for subsets of the trial description which match disjuncts of concepts in memory.

Marvin's memory consists of a list of associations:

```
association  =  record
                    stmnt: statement;
                    UsedIn:  list of concept
               end
```

Each association records the fact that a statement which matches *stmnt* is used in all of the concepts in the list, *Usedin*.

After each conjunction in a concept is learned, a procedure, *index,* is called to update the memory. Each statement in the new conjunction is taken in turn and placed in the index. It operates as follows:

```
index(statement, concept)
        if memory is empty  then
                memory := NewAssociation(statement, concept)
        else
                ConceptList:= lookup(statement)
                 if ConceptList empty  then
                        add NewAssociation(statement, concept) to memory
                 else if concept  not in ConceptList  then
                        add concept to ConceptList
```

The function *NewAssociation* simply creates a new association record as defined above. If Marvin's memory is empty then a new association is added immediately. Otherwise Marvin looks up the statement in the index to the memory. The procedure *lookup* returns the list of concepts which contain statements which match the parameter given.

```
lookup(stmnt1)
        for each pair <stmnt2, UsedIn>  in memory,
                if match(stmnt1, stmnt2)  then
                        return UsedIn
```

The program just scans through the list which represents memory looking for a match. For the scale of problems which have been used to test Marvin, a linear search has proved adequate. However, if a very large data base is required, faster lookup techniques could be used. A method for improving *lookup* is discussed in Chapter 7.

The function *match* performs the pattern matching between statements. It uses the unification algorithm outlined in Section 3.3.

```
match(stmnt1, stmnt2)
        if stmnt1.predicate-name = stmnt2.predicate-name
        and length(stmnt1.Args) = N
        and length(stmnt2.Args) = N
        then for i  in 1..N,
                if not unify(stmnt1.arg[i], stmnt2.arg[i]) then
                        return FALSE
```

Two statements match if they have the same predicate name and all their arguments unify. As was seen earlier, variables are named with respect to a particular binding environment, so a complete call to

unify must include the environments of the statements being matched. For example,

```
unify(stmnt1.arg[i], env1, stmnt2.arg[i], env2)
```

When searching for a match between statements in the trial and statements in the index, env1 will represent the environment created for the primary statements by *primary*. The variables in *env2* will be instantiated by the pattern matching procedure.

Let us now give a complete definition of *unify:*

```
unify(term1, env1, term2, env2)
        if term1 is a variable  then
                if term1 is bound to value, v in env1  then
                        return unify(v, env1, term2, env2)
                else
                bind(term1, term2, env1);
                record the substitution for term1
                return TRUE
        else if term2 is a variable  then
                return unify(term2, env2, term1, env1)
        else if term1 is same type as term2  then
        case type of
                ATOM:  return (term1 = term2);
                NUMBER:  return (value of term1 = value of term2);
                SELECTOR:
                        if term1.property = term2.property  then
                                return unify(term1.obj, env1, term2.obj, env2)
                        else return FALSE
        else return FALSE
```

If either term is a variable then the values must be looked up in the set of substitutions or bound if they are not already bound. Since atoms are stored only once, references to atoms will match only if the point to the same atom. Two references to numbers match only if the numerical values referred to are the same. Selectors match if the property name is the same and the variables in the selector match.

When a substitution is made it is recorded in a special stack called the *trail*. This is done for the benefit of the generalization procedure which, at certain times, will need to backtrack and undo some of the substitutions created by the pattern matcher.

So far in this section we have used the terms *substitution* and *binding environment* without describing how they are implemented. The method used is well known to compiler writers. A binding environment is represented by a group of slots, called a *frame,* on a stack. Each variable is associated with one slot in the frame. Variable Xn is bound to the value stored in position, *frame + n,* in the stack, where *frame* points to the base of the binding environment of that variable. For example, the substitution *{ X0/RED, X1/BLUE, X2/GREEN}* is represented as

frame + 2	Green
frame + 1	Blue
frame --->	Red

The advantage of a stack implementation is that it makes backtracking very easy. When it becomes necessary to change the binding environment back to a previous state, we need only change the frame pointer.

5.4 The Generalization Algorithm

The goal of the generalization algorithm is to look for replacements which create a new trial which is more general than the current one. The program scans down the list of statements which form the description. If a statement has not been replaced previously (i.e. state ≤ 0), the algorithm tries to introduce new statements which refer to a concept whose description contains a match for the statement.

```
generalize(description)
        for each statement in description,
                if statement.state ≤ 0 then
                        TryConceptsWith(statement)
```

The statement which is being used to look for concepts is called the *focus* (Bruner, 1956).

TryConceptsWith will look up the focus in memory to find the list of concepts which contain statements similar to the focus. In the following procedure, this list is called *UsedIn*.

```
TryConceptsWith(focus)
        declare global StmntsOut
        recognized:= FALSE
        UsedIn:= lookup(focus);
        for each concept in UsedIn,
                if CheckConcept(concept) then
                        recognized:= TRUE
        if recognized then return TRUE
        else return FALSE
```

CheckConcept is called for each member of *UsedIn* to discover if any of these concepts recognize any part of the description. Those concepts which do will be used to generalize the trial. If no such concept is found then *TryConceptsWith* returns false.

During the life of *TryConceptsWith,* some statements in the description will be replaced by new statements. As a statement is removed, a reference to it is placed at the head of the list *StmntsOut*. *TryConceptsWith* will be called recursively so it should be remembered when another call to it is encountered, that each invocation of *TryConceptsWith* creates a new StmntsOut. The new StmntsOut will remain in existence only as long as the invocation of *TryConceptsWith* which created it remains in existence.

```
CheckConcept(concept)
        for each disjunct in concept,
                if Contains(disjunct) then return TRUE
        return FALSE
```

CheckConcept tries each disjunct of the concept to see if the trial *Contains* a match for that disjunct. When a match is found, the procedure returns true. If no match is found then it returns false.

Contains implements a search for a subset, *M*, of the description which matches a conjunction. *Contains* is implemented as a recursive procedure in order to perform a depth first search for all possible matches between a disjunct of a known concept and the description of the training event. The argument *conjunction* is a list of statements in a concept which is already stored in memory. With each recursive call of *Contains,* the program moves down this list, trying to match its head with a statement in the description. If the head of *conjunction* matches a statement, *S*, then *S* may be an implicant, so it is removed. That is, the state indicator of *S* is incremented by 1.

Contains is then called recursively to find a match for the remainder of the conjunction. If *contains* fails, then the program could not find a complete match for the conjunction. That is, *S* is not an implicant so it is restored to the trial (the state indicator is decremented). The variable *PartOut* records the statements that have been temporarily removed. When *contains* has found one complete match, *StmntsOut* is partially restored so that other matches can be found. Note that *StmntsOut* is used as a stack. Since *PartOut* points into this stack, the assignment *StmntsOut:= PartOut* has the effect of partially cutting back StmntsOut.

When a match fails, the substitutions created during the pattern matching operations must be *forgotten*. *OldSubst* is used to record the position on the stack to which the program must backtrack in

order to forget the variable bindings. Notice that by backtracking, the algorithm tries to find *all* possible matches for the disjunct of a concept.

```
Contains(conjunction)
        if conjunction = nil then
                if Replacement(StmntsOut, focus) succeeds  then
                        return TRUE
                else return FALSE
        else
                PartOut:= StmntsOut
                OldSubst:= Substitutions
                 for each statement  in description do
                        if match(statement, conjunction.head)  then
                                remove(statement)
                                if Contains(conjunction.tail)  then
                                        A complete match has been made
                                        StmntsOut:= PartOut
                                else restore(StmntsOut, PartOut)
                        ForgetSubst(OldSubst)
                return TRUE if a least one complete match was found
```

When the procedure reaches the end of the conjunction, a complete match has been found. At this point Marvin will try to finish the replacement process by creating a new statement.

```
Replacement(StmntsOut, focus)
        if focus  not in StmntsOut  then
                return FALSE
        S:= CreateStatement(StmntsOut)
        if S = nil then
                return FALSE
        if not MoreGeneral and not restricting  then
                remove(S)
                return FALSE
        if Consistent(S)  then
                TryUnremoved(StmntsOut)
                return TRUE
        else return FALSE
```

Replacement must perform a number of tests before a new trial can be created.

- The focus statement which was passed as a parameter to *TryConceptsWith* must be in the statements removed.
- If the program is trying to generalize the trial, then it must insist that at least some statements have been removed, otherwise it would not be a proper generalization. However, if the program is attempting to make the trial more specific (restricting it) then it doesn't matter if no statement can be removed.
- Once a new trial has been formed, it must be tested to see if it is consistent. To do this an instance of the trial is shown to the trainer. If the trial is not consistent, it must be made more specific.

```
CreateStatement(StmnstOut)
        Find the arguments of the new statement.
        if there is a  many-to-one binding then return nil
        if TriedBefore(ConceptName, ArgList) then return nil
        FindRemovable(StmntsOut)
        S:= NewStatement(concept, Args)
        append S to description
        return S
```

To create the new statement, Marvin must find the actual parameters of the call from the substitutions produced by the pattern matcher. It is possible that the match contains a many-to-one variable binding. At present Marvin is incapable of dealing with this situation. The arguments for the new statement can be found quite easily by looking up the values bound to each formal parameter of the concept.

```
FindArgs:
        ArgList:= nil
        for each argument  in formal parameters of concept,
                find value, v bound to argument in substitution
                if v  in Arglist  then
                        many-to-one:= true
                        return FALSE
                else place v in ArgList
        return ArgList
```

TriedBefore finds out if the same concept has been used to recognize the same event before. If this is true then the statement being created will be a duplicate of one already in the description. A new statement should not be created if there is a statement already in the description which has the same predicate name and an identical argument list. So we define *TriedBefore* as,

```
TriedBefore(ConceptName, Arglist)
        if there exists a statement, S
        such that S.predicate-name = ConceptName
        and S.args = Arglist
        then return TRUE
```

The program must also check that the statements in StmntsOut may be removed without violating the condition that the trial must be able to specify a complete object.

Suppose a training instance shown by the trainer is the event *(fred, bill, jack)* where *fred, bill* and *jack* are the names of objects. When the primary statements are generated these names are replaced by the variables *(X0, X1, X2)*. These will become the *formal parameters* of the new concept, once it has been learned. Obviously, there must always be some statement in the trial which describes each formal parameter.

```
FindRemovable(StmntOut)
        for each statement, S  in StmntsOut,
                if CannotRemove(S)  then restore(S)
```

Most of the primary statements will have the form *Xn.property = value*. If there is a statement such as *X0.head = X3* where *X3* is an existentially quantified variable then this statement will be called a parent of *X3*. (The statement has introduced *X3* to the world.) *X3* must be described somewhere; thus if the program attempts to remove a statement containing *X3* and a parent of *X3* is still in the trial then that statement may not be removed unless there are other references to *X3* elsewhere in the trial.

```
CannotRemove(statement)
        if statement is a primary  then
                return NoOtherRef(second argument of statement)
        else for each argument  in statement,
                if argument is a formal parameter of target
                and NoOtherRef(argument)
                then return TRUE
                else if ParentIn(statement) and NoOtherRef(argument) then
                        return TRUE
        return FALSE
```

Let us describe the procedures for finding the parent of a variable and for finding if there are other references to it.

```
ParentIn(value v)
        L:= list of statements which contain variable, v
        for each statement, S  in L,
                if s = Xn.prop = v  and S  in Trial  then return TRUE
```

The *isin* list of v is being used to find out which statements contain a reference to *v*. These statements are then examined to see which one is the parent of *v*.

To find out if a variable is referred to in other statements, Marvin scans through the *isin* list of *v* and checks if *v* is a member of the argument list one of the statements which is in the trial.

```
NoOtherRef(v)
        if v  in Args  then return FALSE
        L:= list of statements containing v
        for each statement, S  in L  and S  in Trial,
                if member(v, arguments of S)  then return FALSE
        return TRUE
```

Args is the list of arguments of the statement which is about to be added to the trial. If *v* is in *Args* then it doesn't matter if there are no other references elsewhere.

Finally, the new statement can be constructed and added to the description. The arguments found by *FindArgs* become the parameters to of the statements. *StmntsOut* contains the list of implicants. Once a record for the statement has been allocated, it must be placed in the occurrence list of each of the variables contained in the statement.

```
NewStatement(concept, Args)
        allocate record for new statement, S
        predicate-name of S:= concept;
        S.Arguments:= Args;
        S.Implicants:= copy of StmntsOut;
        S.state:= 0
        for each argument of S,
                place S in occurrence list of argument
        return S
```

The trial created by the replacement which has just taken place must be checked to ensure that it is a proper generalization of the previous trial. This can be guaranteed if at least one of the implicants of the new statement has been removed for the first time. A statement may already be outside the trial because it is the implicant of another statement.

```
MoreGeneral:
        for each statement  in StmntsOut,
                if statement has just been removed  then return TRUE
```

If a statement is in *StmntsOut* then its state indicator has just been incremented by 1. If the state is equal to one then the statement has not been removed previously. Therefore the replacement being attempted now generalizes the trial because a statement has been removed.

The program now has a trial which it can test to see if it is consistent or not. This is one of the functions of *Consistent*. If the trial is consistent then further generalizations will be attempted when *Consistent* returns. If it is not consistent, *Consistent* will try to construct a more specific trial by using the implicants of the new statement to add more information to the trial

```
Consistent(NewStatement)
        if experiment with trial fails  then
                for each statement  in StmntsOut,
                        if statement is not in trial  then
                                restore(statement)
                                if TryConceptsWith(statement)  then
                                        return TRUE
                                else remove(statement)
                remove(NewStatement)
                return FALSE
        else return TRUE
```

This algorithm chooses each implicant of the new statement in turn and tries to introduce new concepts with these implicants. Since this is recursive, *TryConceptsWith* will create a new trial and test it. If this one is also inconsistent, the program will check the consistency of that trial as well. If all these attempts fail then the new statement must be abandoned since it was impossible to create a consistent trial which includes *NewStatement*. When *NewStatement* is removed, the state indicator is set to -1. This allows the statement to be used as the focus for *TryConceptsWith,* but excludes it from the trial.

When Marvin was learning *less* we saw that not all the statements in *StmntsOut* could be removed. *X3.val = 1* could not be removed when *number(X2)* was introduced because *X3* was referred to by more than one object. However, after the replacement was completed, the statement could be removed. *TryUnremoved* looks at the implicants of a new statement to see if there are any primary statements which can be removed after the replacement has been completed.

```
TryUnremoved(StmntsOut)
        for each statement  in StmntsOut,
                if statement is a primary  and it is in trial  then
                        relax(statement)
```

This code bears some resemblance to parts of *Consistent* except that Marvin is now relaxing some constraints on the concept rather than introducing new ones.

```
relax(statement)
        if CanNotRemove(statement)  then return
        else remove(statement)
        for each implicant of statement,
                if implicant is not in trial  then
                        restore(implicant)
                        if TryConceptsWith(implicant)  then
                                return
                        else remove(implicant)
        restore(statement)
```

The final program to consider in the learning algorithm is the part of the program which performs experiments.

```
PerformExperimentWith(trial)
        experiment(trial)
        ask  'Is object recognized by concept?'
        if answer is  'yes'  then return TRUE
        else return FALSE
```

Experiment invokes the proof procedures which will construct an instance of the concept. There are two phases in generating an instance: constructing an event, and ensuring that the event is one that will enable Marvin to learn something new.

5.5 Executing Concepts as Programs

In order to produce a training example, Marvin treats a concept description as a program. The output is an event which is recognized by the concept. In Section 3.2 we discussed the semantics of the description language. Now let us look in detail at how objects are constructed.

An object is constructed by the actions of the primary statements, which can sometimes be thought of as assignment statements. When an argument of the '=' predicate is an unbound variable (or property) a value is assigned to it. The other constructs of the language, the concept calls, AND and OR connectives, control the execution of the primaries.

In the first stage of our tour through Marvin's object construction program, we will look at how objects and values are represented during execution.

5.5.1 Binding Environments and Stacks

Earlier we saw that a stack was used to store substitutions for variables during pattern matching. The same sort of mechanism can be used to represent substitutions during the execution of a program.

Remember that the definition of *append* is:

```
[X0, X1, X2:
        X0.value = nil ∧  X2 = X1 ∧  list(X1)
∨
        [∃ X3:
                X0.hd = X3
              ∧ X2.hd = X3
              ∧ number(X3)
              ∧ append(X0.tl, X1, X2.tl)                    (C1)
        ]
]
```

During the execution of *append,* the variables *X0, .., X3* will have some values associated with them. This association is implemented by placing a reference to the value of variable *Xn* in the stack position, *frame + n,* where *frame* is the base of the binding environment of this call to *append*. Each time a concept is called, space for its binding environment must be allocated on the stack. This will be described shortly.

Considering *append(X0, X1, X2)* as a procedure call in a conventional language, *X0, X1* and *X2* are values which are passed to *append* to be bound to the formal parameters of the procedure. Of course, our purpose is to have the interpreter supply values for *X0, X1* and *X2*. This is indicated by asking Marvin to prove

[∃ X0, X1, X2: append(X0, X1, X2)]

When the interpreter encounters ∃ *Xn,* a dummy value, called a *QVAR,* is created and a reference to it is placed in the stack as the value of *Xn*. *QVAR*s are intermediate storage locations which, initially, are empty. At some point in the execution of the concept, a *QVAR* will be assigned a value by a primary statement.

*Implementation Note: QVAR*s may be allocated off a stack so that the space they occupy may be reclaimed after use.

5.5.2 Executing Primary Statements

Suppose the interpreter is executing the statement:

X0.colour = red

X0 is an object which is going to be constructed with a property *colour* whose value is *red*. The value of *X0* is represented by a *QVAR, Q,* which is unbound initially. In order to execute the statement, a new object must be created, and a property: value pair must be put into the object with the values *colour: red* inserted. Part of the structure resulting after completing this statement is shown in Figure 5.4.

The interpreter calls the function *equiv* to execute an equivalence statement. It tries to find the values of the left and right hand arguments. If it cannot find a value then a new *QVAR* is created. For example, since *X* is represented by an unbound *QVAR*, when Marvin looks for the value of *X0.colour,* a new object is constructed with the property *colour*. The value of *colour* is not known yet, so a new *QVAR* is created to represent it.

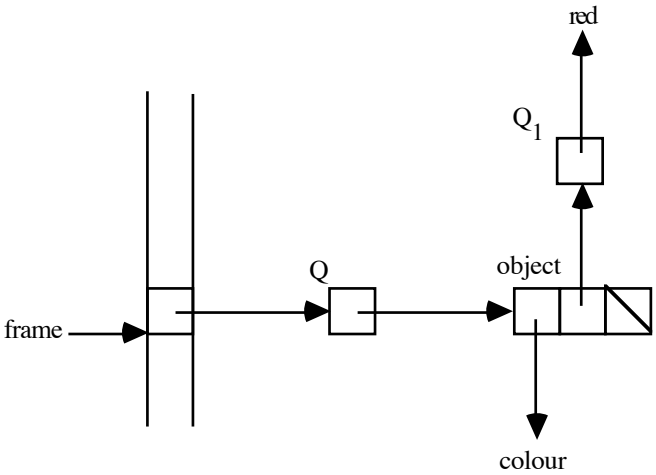


Figure 5.4. Executing X0.colour = red

```
equiv(value1, value2)
        x:= value-of(value1)
        y:= value-of(value2)
        if x or y are nil then return nil
        else if x is an unbound QVAR then
                x:= y and put x on trail
                return TRUE
         else if y is an unbound QVAR then
                y:= x and put y on trail
                return TRUE
         else both x and y are bound
                return equal(x, y)
```

In our example, *equiv* is called thus: *equiv(X0.colour, red)*. The first task of the procedure is to discover what the values of its arguments are. *X0.colour* may already be bound, if it is not, a *QVAR* must be created to represent it.

```
value-of(x)
        if x = nil then return nil
        else case type(x) of
                ATOM    : if x is the name of an object then
                                return object
                          else return x
                NUMBER, OBJECT:         return x
                VARIABLE:               return value-of(stack[frame + n])
                SELECTOR:               return value-of(get(value-of(obj, prop)))
                QVAR:                   return val-of-qvar(x)
```

If *x* is a variable then the value bound to *x* must be found. This can be done by locating the stack slot associated with the variable. The stack is an array of values, thus the value of the variable, *Xn,* is *stack[frame + n]* where, *frame,* is the index of the base of the binding environment for the current concept call.

If *x* is a selector of the form *Xn.prop,* then the value of *Xn* must first be found. A procedure, *get,* is then called to get the value associated with the property, *prop,* in the value of *Xn*.

During execution it is possible to build up a chain of *QVAR*s where one *QVAR* points to another. *Val-of-qvar* returns the value at the end of the chain.

Get is the function which creates objects. The arguments of *get* are *obj,* the name of an object (that is, a variable) and *prop* the name of a property. If *obj* is not bound then a new object is created and a pair created with *prop* in the property field. The value field of the pair is not yet known, so a new *QVAR* is created and put in.

```
get(obj, prop)
        if obj = nil then return nil
        else if obj is a word or number then return nil
        else if unbound(obj) then
                obj:= new object
                add obj to trail
                make a new property:value pair
                        property := prop and value := new QVAR
                return QVAR
        else if ∃ pair <property:value> in object then
                return value
        else if obj not complete then
                make a new property:value pair
                        property := prop and value := new QVAR
                return QVAR
        else return nil
```

A new pair is also created if the object exists, but is still 'under construction'. An object must be completely specified within one concept.

To complete the description of *equiv* we now specify the meaning of *equal*

```
equal(value1, value2)
        x:= value-of(value1)
        y:= value-of(value2)
        if x or y are nil then return FALSE
        case type(x) of
                QVAR, ATOM:     return TRUE if x and y are identical
                NUMBER:         return TRUE if numerical values are same
                OBJECT:         return TRUE if for each pair p1:v1 in x,
                                ∃ pair p2:v2 in y:
                                        p1 = p2 and equal(v1, v2)
```

5.5.3 The Control of Execution and Backtracking

The execution of primary statements instantiates objects. The remainder of the interpreter is involved in controlling the order in which the primaries are executed.

The interpreter executes the statements in a conjunction sequentially. If the statement is a primary statement, it is evaluated. If the statement is a reference to another concept, then then execution environment is modified and execution begins on a new conjunction.

To call a concept, the interpreter performs the following actions:

```
call(concept, actual-parameters)
        bind formal-parameters to actual parameters
        D:= first conjunction in concept
        save alternatives
        make QVARs for quantified variables in D
        return D
```

We have already seen how the formal paramaters are bound. When a concept consists of a number of conjunctions, Marvin cannot know which conjunction must be proved so that the entire trial will be true. Thus, the interpreter will try to prove the first conjunction it finds and saves the rest in case the proof is not successful and another alternative must be tried.

In order to be able to backtrack, quite a lot of information must be stored. When a new concept is to be proved, the environment of the calling concept, the parent, must be saved. The following items are saved as a record, called a *control node,* on Marvin's *control stack:*

frame: When backtracking occurs the system must return to the original binding environment. Thus the stack frame pointer of the calling concept must be saved.

TrailPoint: As well as returning to the previous binding environment, *QVAR*s which have been assigned values since the present environment was saved must be cleared. When a *QVAR* is assigned a value, it is placed on a stack called the *trail*.

alternatives: This points to the next conjunction to be attempted when backtracking returns to this point.

parent: When a conjunction terminates, control must return to the calling concept - the parent. Thus a pointer to the control node of the parent is saved.

continuation: Apart from knowing which conjunction called the present one, the interpreter must know which statement to continue executing from.

Suppose there is a conjunction P ∧ Q ∧ R. Before *P* is executed, the current environment is saved. The frame on the variable stack is saved. The current trail pointer is saved so that if *P* assigned values to *QVAR*s but eventually failed, those assignments can be undone. If *P* succeeds, the interpreter must know that it should continue execution at *Q*. This is the purpose of the 'continuation' pointer. If *Q* fails it may be because *P* bound a variable to a value unacceptable to *Q*. In this case another disjunct of *P* should be tried in an attempt to produce variable bindings which are acceptable to both *P* and *Q*. Thus before entering P for the first time, one conjunction is selected for execution and the remaining alternatives are placed on the stack. When *Q* fails, the interpreter will look for *P*'s alternatives on the stack.

Figure 5.5 shows the state of all the stacks during the execution of *append*. The explanations to follow may become clearer by referring to this diagram. It shows a snapshot of the system when the first disjunct of *number* is being executed. It is assumed that *number* was called recursively by the second disjunct of *number* which, in turn, was called from the second disjunct of *append*. *Append* was called from the original request to construct an example to show the trainer.

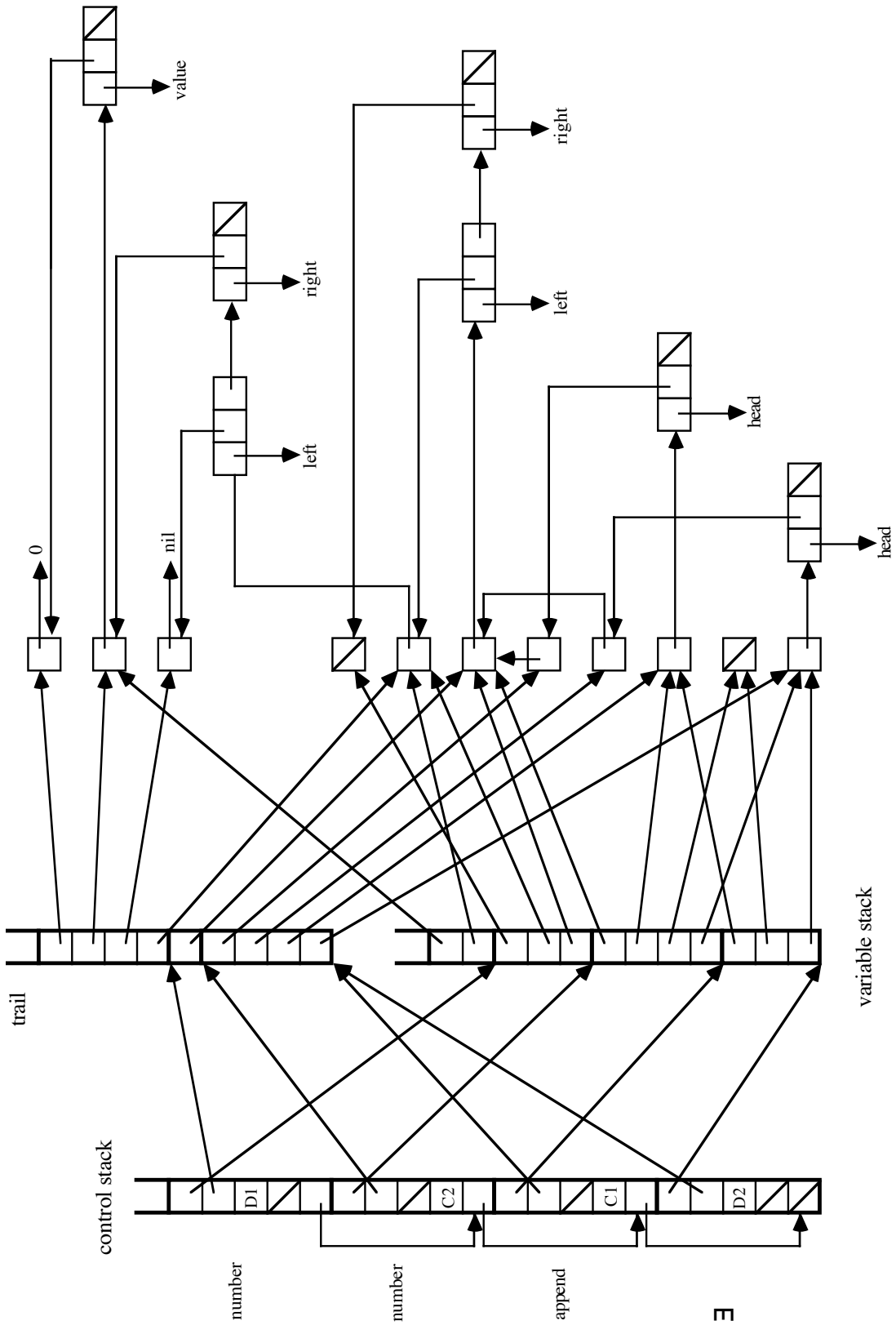


Figure 5.5. Stack Organization During Execution of append

C1 and *C2* are continuations recorded on the control stack. They refer to the corresponding statements marked in the descriptions of the concepts. *D1* is the second disjunct of number which is an alternative left on the stack in case the first disjunct fails. *List* is defined as:

```
[X0:
        X0.val = none
∨
        number(X0.hd) ∧ list(X0.tl)
]
```

The definition of *number* is:

```
[X0:
        [∃ X1:
            X0.left = nil
          ∧ X0.right = X1
          ∧ X1.value = 1
        ]
∨
        [∃ X1, X2:
            X0.left = X1                                    (D1)
          ∧ X0.right = X2
          ∧ number(X1)
          ∧ digit(X2)
        ]                                                   (C2)
]
```

and *digit* is

```
[X0: X0.value = 0 ∨  X0.value = 1]
```

Remember that one field in a conjunction record contains a list of quantified variables. Before execution of the conjunction can begin, each variable must be assigned a new *QVAR*.

Execute contains the main execution loop for a conjunction. This procedure moves down the list of statements, *D,* in the conjunction. As it encounters a primary statement, it is evaluated by *equiv*. If a concept reference is encountered, *call* is invoked. This saves the current value of *D* on the control stack and changes *D* to the first conjunction of the called concept. *Succeed* and *backtrack* also change the value of *D*. *Succeed* sets *D* to the continuation of the parent and *backtrack* sets it to a new alternative to try after the proof has failed for previous alternatives.

```
execute(D)
        repeat
            successful:= true
            while (D ≠ nil) and successful do
                S:= head of D
                if S is a primary then
                        if equiv(S.arg[1], S.arg[2]) then
                                D:= tail of D
                        else
                                D:= backtrack
                                successful:= FALSE
                else D:= call(S)
            if successful then D:= succeed
        until CSP = 0
```

All changes in the environment are recorded on the control stack. The current environment is indicated by the *C*ontrol *S*tack *P*ointer (CSP). When the while loop terminates, a conjunction may have been executed or failed. In both cases, the *CSP* will be modified. Before the program can continue execution

it must check that the entire program has not been completed - either successfully or unsuccessfully. Termination occurs when the control stack is empty. Since all the work that the interpreter must do is scheduled by the contents of the stack, when *CSP = 0,* all the work is done.

Let us now consider what actions must be performed when a conjunction has been successfully executed.

```
succeed:
        while CSP ≠ 0 do
            node:= ControlNode[CSP]
            mark QVARs in this node's environment as complete
            CurrentFrame:= node.frame
            if node.continuation = nil then CSP:= node.parent
            else return node.continuation
```

When a concept has been successfully executed, the objects that it constructed are marked as complete. The binding environment is changed to the binding environment of the parent. If the parent's continuation is nil, that is, there are no more conditions to satisfy in the parent, then the procedure skips to the control node of the parent's parent. Otherwise, it returns the continuation of the parent. This becomes the new *D* in *execute*.

The arguments of a completed concept are marked as complete so that another concept cannot add new pairs to the object.

When failure has occurred, Marvin must try another way of finding the solution by backtracking. It scans down the control stack, looking for a concept which still has some alternatives left.

```
backtrack
        while CSP ≠ 0 do
            node:= ControlNode[CSP]
            if node.alternatives = nil then
                CSP:= CSP - 1
                CurrentFrame:= node.frame
            else
                clear_trail(node.TrailPoint)
                CurrentFrame:= node.frame
                D:= first alternative of node
                make QVARs for D
                node.alternatives:= rest of node.alternatives
                return D
```

If a concept with alternatives is found, a new execution environment must be set up. All the *QVAR*s which where assigned values in the conjunctions that failed must be cleared. The *QVAR*s were stored on the trail. The binding environment pointer, CurrentFrame, must be reset to the new bindings. The next conjunction to execute is obtained from the alternatives and its quantified variables are initialized.

5.6 Performing Experiments

Being able to construct an event from a concept description doesn't mean that Marvin can perform a valid experiment. As was seen in Chapter 3, Marvin must ensure that if the trial is inconsistent, the constructed event must not be recognized by the target. To do this, the statements which have been removed from the trial by a replacement (i.e. state ≥ 0) and which are not implied by statements still in the trial, must be false.

Before searching for an event to be used as an experiment, Marvin performs some pre-processing. First it simplifies the trial and then it determines which of the statements removed may be made false (or denied). The overall design of the experiment is as follows:

```
experiment(trial)
        T:= simplified(trial)
        Out:= statements to be denied.
        repeat
            execute(T)
            if denied(Out)  then
                output event that has been constructed
            else T:= backtrack
```

The simplified trial is executed to construct an training event. *Denied* is then called to make sure that the statements in *Out* are all false. If one is not then the system must backtrack to construct an new event. Note that when the system backtracks, it changes the value of T so that it points to the group of statements with which it will resume execution again.

A backtracking point is created when a call is made to a concept which has alternative conjunctions. The procedure *backtrack* returns to the point most recently placed on the control stack. However, it may be that choosing the most recent alternative will not change the property which caused the failure. At present a very simple (but not very efficient) method is used to solve this problem. Suppose a statement, *S*, in *Out* is true, that is, *execute* failed to produce an event which the learning algorithm can use. Another attempt is made to constructed a useful example. If *S* is true following the second attempt then Marvin did not backtrack far enough since the alternative chosen was not one which changed the property that made *S* true. Therefore Marvin must backtrack further and try again.

The procedure *denied* is quite simple:

```
denied(Out)
        for each statement  in Out,
            evaluate statement
            if statement is TRUE  then
                return FALSE
        return TRUE
```

A statement is evaluated by executing it as a simple boolean expression.

A trial is *simplified* by removing statements which are implied by other statements of the trial. For example, there may be two statements: *less(X0, X1)* and *number(X0)*. The first statement implies the second. If *less(X0, X1)* is true then *number(X0)* must be true. Therefore there is no need to prove *number(X0)*. Eliminating redundant statements also makes the job of the proof procedure easier.

```
simplified(Trial)
    for each statement, S  in Trial,
        if ∄ statement, S'  in Trial: S'.args ⊂ S.args  then
            place S in list T
    return T
```

Statement S is in the simplified trial, *T,* if there is no statement *S*' whose arguments contain the arguments of *S* as a subset.

A removed statement, *S*, cannot be made false if:

- *S* is an implicant of a statement, *S*', in the trial and *S*' refers to a conjunctive concept. *S*' specifies only one set of values for its arguments, there can be no alternatives.
- The statement, *S,* is implied by statements in the trial. If the implicants are true then *S* must be true also. *S* is implied by statements in the trial if:
  - the implicants of *S* are a subset of the trial.
  - there is a statement, *S*' whose arguments contain the arguments of *S*.
- *S* may be a member of every conjunction in a concept which is called from the trial. Therefore there is no conjunction which can be true while *S* is false. (Case 1 is a special case of this).

The procedure *ToBeDenied* scans the removed statements looking for occurrences of cases 1 and 2.

```
ToBeDenied(StmntsOut)
Out:= copy of StmntsOut
for each statement removed,
        if statement implies conjunctive concept
        or statement is implied by statements in Trial
        or ∃ a statement which gives a more specific description of event
        then delete statement from Out
        return Out
```

Since the last condition occurs infrequently, and is time consuming to detect, an *ad hoc* approach has been used to deal with the problem. It is ignored! However, if the program is incapable of producing a result because a statement could not be falsified - the offending statement is assumed to fit the third category and is removed from Out.

A complete description of experiment is now:

```
experiment(Trial)
        T:= simplified(Trial)
        Out:= ToBeDenied(StmntsOut)
        repeat
            set up environment for executing T
            repeat
                execute(T)
                if successful  then
                        if denied(Out)  then
                                output constructed event
                        else T:= backtrack
            until successful  or cannot backtrack any more
            if not successful  then
                delete statement which caused failure from Out
        until successful
```

The inner **repeat** loop represents the first version of *experiment*. If this loop fails to produce a result because of one statement in *Out* then the statement is removed from *Out,* and the process is repeated.

Note that the control and variable stacks and the trail remain in the state they were in at the completion of *execute,* by backtracking and resuming *execute,* the program can continue to find the next alternative solution.

5.7 Remembering Concepts

When a conjunction has been learned, it must be stored in Marvin's memory. Before doing so, the description of the concept must be cleaned up.

All the removed statements are disposed of, and a procedure similar to *simplified* is called to remove redundant statements.

```
simplify(description)
        for each statement  in description
            if state of statement ≠  0
            or statement implied by another statement  in Trial
            then remove statement permanently
```

Once the final form of the conjunction has been established, the conjunction may be stored in the memory by calling *index* to update the statement index.

```
remember(description)
        for each statement  in description,
            index(statement)
```

This completes the learning process for one conjunction. Control returns to *learn* and the entire process is repeated until the trainer has no more concepts to teach Marvin.

6
Performance Evaluation

This chapter presents the results of tests performed during a number of learning sessions with Marvin. The tasks have been chosen from several different domains, including geometric concepts, grammatical inference, and automatic programming.

The measurements made were:

- The total time taken to learn the concept.
- The proportion of the total time spent in generating training examples to show the trainer.
- The number of hypotheses formed while learning a concept.
- The number of hypotheses which were incorrect.

In addition, the program was profiled while learning the 'number' concepts of Chapter 4. Profiling yields the number times each procedure was called. This allows us to determine where the program spent most of its time and where its performance could be improved.

During this discussion we will try to answer the following questions: What concepts can Marvin learn and when will it fail? How efficient are the various algorithms? How efficient is the implementation of those algorithms? How does it compare with other concept learning programs?

6.1 Learning Geometric Concepts

6.1.1 Blocks World

At the beginning of this work, we speculated about the way in which a child might learn spatial relationships between physical objects under the guidance of an adult. Marvin can learn concepts such as 'on-top-of' in much the same way as we expect the child to perform the same task.

Before being able to learn the circumstances in which one object may be placed on top of another, Marvin must first learn about some properties of physical objects, such as colour and shape. Marvin learns by rote (i.e. without generalization) that the values red, green and blue are colours:

is-colour =
[X0: X0.value = red ∨ X0.value = blue ∨ X0.value = green]

and the values 'box' and 'table' are 'flat'.

flat = [X0: X0.value = box ∨ X0.value = table]

A value is a shape if,

is-shape =
[X0: X0.value = sphere ∨ X0.value = pyramid ∨ flat(X0)]

In Chapter 1 we assumed that an adult showed the child a red sphere on a green box. From this instance of 'on-top-of', he learnt a general description of the concept by experimenting with the configuration of the objects.

Marvin, the program, was also shown a red sphere on a green box: the object E1,

E1 = <top: S1; bottom: B1>
S1 = <shape: SPHERE; colour: RED>
B1 = <shape: BOX; colour: GREEN>

This instance was generalized to the description:

on-top-of =

```
[X0:
        [∃ X1, X2, X3, X4, X5, X6:
                X0.top = X1
                X1.shape = X2
                X1.colour = X3
                X0.bottom = X4
                X4.shape = X5
                X4.colour = X6
                is-shape(X2)
                is-colour(X3)
                flat(X5)
                is-colour(X6)
        ]
]
```

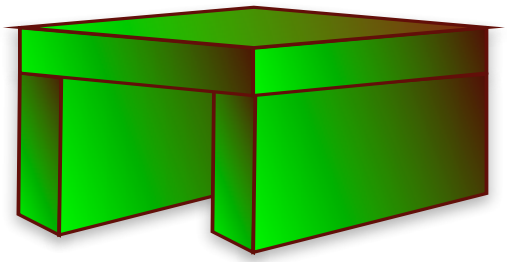
That is, the top can be any shape and any colour, the bottom can also be any colour, but its shape must be flat.

The five questions that were asked by the child in Chapter 1 were, in fact, those asked by Marvin when it performed this task.

6.1.2 Winston's Arch

No work dealing with concept learning would be complete without some reference to Winston's famous ARCH (Winston, 1970). The reader may recall that Winston's program was capable of learning the description of an arch from examples supplied by the trainer in the form of line drawings. The training instances illustrated both arches and non-arches. The description of an arch may be paraphrased in English as

> There are three objects A, B and C. A and B are blocks which are standing up. A is left of B and they do not touch. A supports C and B supports C. C may be any shape, but it is lying on top of A and C.



Marvin can also learn to describe arches. One difference between this program and Winston's is that only one example is shown to Marvin by the trainer. After that Marvin shows the trainer its own examples. However, the trainer cannot show a line drawing, he must present the training arch as an object description consisting of property/value pairs. It could be assumed that a front-end program performed the low-level recognition of line drawings and output its results as object descriptions which Marvin can understand.

There are several different ways of representing an arch in the language. We will chose a fairly simple method:

```
top = <shape: BRICK; orientation: LYING; supported-by: L1>
L1 = <left: left-side; touches: FALSE; right: right-side>
left-side = <shape: BRICK; orientation: STANDING; supported-by: FLOOR>
right-side = <shape: BRICK; orientation: STANDING; supported-by: FLOOR>
```

This instance of an arch is a brick which is lying on top of a group of objects, L1. L1 consists of a left and a right side which do not touch. Both the left and right sides are standing bricks which are supported by the floor.

In our representation we will define a physical object as follows:

```
phys-obj =
[X0:
        [∃ X1, X2, X3:
                X0.shape = X1
                X0.orientation = X2
                X0.supported-by = X3
                shape(X1)
                orientation(X2)
                support(X3)
        ]
∨
        [∃ X1, X2, X3:
                X0.left = X1
                X0.touches = X2
                X0.right = X3
                phys-obj(X1)
                boolean(X2)
                phys-obj(X3)
        ]
]
```

This states that a physical object is a block which has shape and orientation and it must have a support. An object may also be a group of blocks listed from left to right. It is necessary to specify if the components of a group are touching or not. This is done by giving the property 'touches' the value TRUE or FALSE. A support may be the floor or another object.

```
support = [X0: X0.is = FLOOR / phys-obj(X0)]
boolean = [X0: X0.val = TRUE / X0.val = FALSE]
shape = [X0: X0.val = BRICK / X0.val = WEDGE]
orientation = [X0: X0.val = LYING / X0.val = STANDING]
```

The description of *arch* learnt by Marvin was:

```
arch =
[X0:
        [∃ X1, X2, X3, X4, X8, X9:
                X0.shape = X1
                X0.orientation = X2
                X2.val = LYING
                X0.supported = X3
                X3.left = X4
                X3.touches = X8
                X8.val = FALSE
                X3.right = X9
                shape(X1)
                phys-obj(X4)
                phys-obj(X9)
        ]
]
```

One criticism of Winston's approach to concept learning is that the trainer must carefully choose the examples he shows the program. In particular, Winston points out that the negative instances should be 'near-misses'. These are objects which are not recognized by the target concept because only a small number of properties do not have the required values. The 'small number' is usually one. Thus the trainer must know quite a lot about the program's learning process in order to prepare the examples.

Marvin also relies on near-misses to test its hypotheses. However, since these examples are generated by the program itself, the trainer need know nothing about the algorithm used to create the

concept descriptions.

An alternative representation for learning ARCH is to use Marvin's language as a meta-language for describing a picture description language. The input data specifying the training example might be a list of objects such as:

```
<pred: on-top; arg1: X0; arg2: X1>
<pred: left-of; arg1: X1; arg2: X2>
<pred: don't-touch; arg1: X1; arg2: X2>
<pred: orientation; arg1: X0; arg2: LYING>
etc.
```

This is equivalent to a set of predicates such as,

```
on-top(X0, X1)
left-of(X1, X2)
don't-touch(X1, X2)
orientation(X0, LYING)
etc.
```

A concept which describes an object like ARCH would actually specify part of the grammar of the description language.

6.1.3 East Bound Trains

Michalski (1980) describes an example of two sets of trains, east-bound and west-bound. The task of the INDUCE-1.1 program described in that paper and by (Dietterich, 1978) is to distinguish between the two sets. See Figure 6.1. The task we will set Marvin is this: given the example of one east-bound train, learn to distinguish all east-bound trains.

Each car is specified by the following properties:

Car shape:	The shape of a car may be an open rectangle, an open trapezoid, U-shaped, an ellipse, a closed rectangle, jagged-topped or it may have a sloping top. The ENGINE is a special car.
length:	The car may be long or short.
Number of Wheels:	A car may have either two or three wheels.
Load:	A car contains loads of various shapes including: circles, rectangles, triangles, hexagons.
Number of parts:	There may be one, two or three parts in the load.

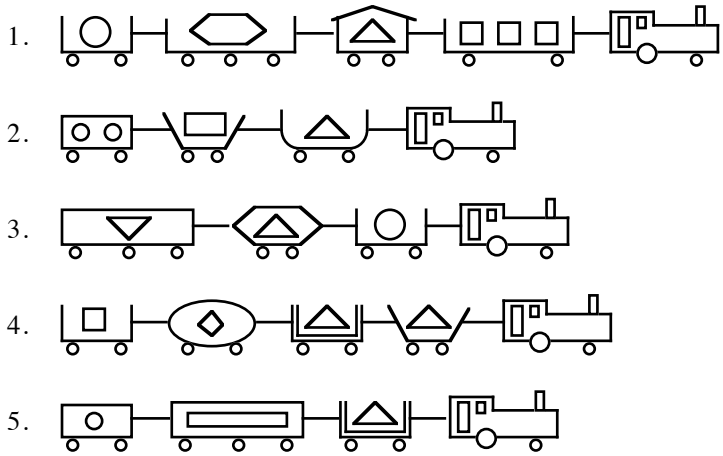
A complete train may be described as a left-recursive list:

<infront: TRAIN; behind: CAR>

Thus the first east bound train is:

```
car1 =
<
        car-shape: open-rect;
        length: long;
        nr-wheels: two;
        load: rectangle;
        nrpts-load: three
>
```

1. Eastbound Trains



2. Westbound Trains

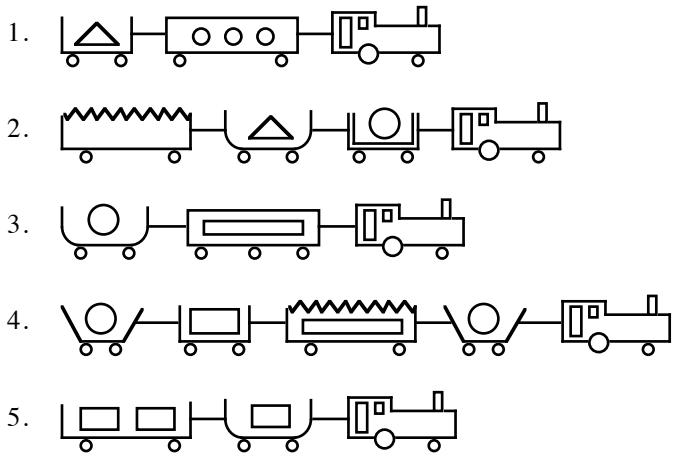


Figure 6.1. East and West Bound Trains

car2 =
<
car-shape: sloping;
length: short;
nr-wheels: two;
load: triangle;
nrpts-load: one
>

and so on.

engine = <car-shape: ENGINE>
T3 = <infront: engine; behind: car1>
T2 = <infront: T3; behind: car2>
T1 = <infront: T2; behind: car3>
train1 = <infront: T1; behind: car4>

The INDUCE-1.1 program is also supplied with rules which describe the problem environment. This domain dependent information may be used by the program while learning the concept. Such rules include the fact that hexagons, triangles and rectangles are polygons. Cars whose shapes are open rectangles, open trapezoids or U-shaped are classed as having 'open tops'. Other cars have closed tops.

Marvin can also be supplied with domain knowledge in the form of concept definitions entered by the trainer. However, an important feature of Marvin is that it can *learn* the domain knowledge.

Other domain knowledge includes the following concepts:

```
train =
[X0:
        X0.car-shape = ENGINE
∨
        [∃ X1, X2:
                X0.infront = X1
                X0.behind = X2
                train(X1)
                car(X2)
        ]
]
```

A train is an engine or a car with a train in front. A car is in a train if it is the hindmost car or it is in the train infront of the hindmost car.

```
in-train =
[X0, X1:
        [∃ X2:
                X1.infront = X2
                X1.behind = X0
                train(X2)
                car(X0)
        ]
∨
        [∃ X2, X3:
                X1.infront = X2
                X1.behind = X3
                car(X3)
                in-train(X0, X2)
        ]
]
```

When shown the first east-bound train as an example, Marvin replied by showing the trainer 20 different trains until it determined that an east-bound train could be distinguished by the description:

```
east-bound =
[X0:
        [∃ X6, X9, X10, X11, X12, X13:
                X9.car-shape = X10
                X9.length = X11
                X11.val = SHORT
                X9.nr-wheels = X6
                X9.load = X12
                X9.nrpts-load = X13
                number(X6)
                closed-top(X10)
                number(X13)
                is-load(X12)
                in-train(X9, X0)
        ]
]
```

This states that there is a car X9 in train X0. The length of X9 is SHORT and it has a closed top. The type of load, the number of wheels and the number of parts in the load may be any value.

This concept was learnt in approximately 14 seconds CPU time on the VAX-11/780. Of that time, 58% was spent in generating the 20 examples to show the trainer. Marvin created more intermediate hypotheses in learning this concept than in any other given to it. However, of the twenty hypotheses, only four were inconsistent. The reason that so many trials were produced is that a large amount of data is present in the examples.

6.2 Learning Grammar

6.2.1 Winograd's Grammar

Marvin is capable of learning simple grammars. For example, Winograd (1972) uses a simple context free grammar to demonstrate the use of systemic grammars in his program SHRDLU. This example is quite interesting since it demonstrates Marvin's ability to partially learn a concept, leave it temporarily, learn a new concept and then return to the original concept to complete its description. This is necessary since several of the concepts describing the grammar refer to each other.

The grammar to be learnt is:

S	NP VP
NP	Pnoun
NP	DET NOUN
Pnoun	John
Pnoun	Mary
DET	a
DET	the
NOUN	apple
NOUN	giraffe
VP	IVERB
VP	TVERB NP
IVERB	sleeps
IVERB	dreams
TVERB	dreams
TVERB	eats

Some typical sentences which may be generated are:

John dreams.
John eats the apple.
A giraffe eats the apple.
The giraffe sleeps.

A sentence is made up of a noun phrase, followed by a verb phrase. The noun phrase may consist of a proper noun or a determiner followed by an ordinary noun. A verb phrase may contain a single intransitive verb or a transitive verb followed by another noun phrase.

Note that the meanings of the words are completely ignored so that 'The apple eats a giraffe' is considered a valid sentence! To teach Marvin this grammar we will represent a sequence of words as a list, terminated by STOP. Part of the data supplied is:

np1 = <head: JOHN; tail: STOP>
np2 = <head: THE; tail: np3>
np3 = <head: APPLE; tail: STOP>

vp1 = <head: SLEEPS; tail: STOP>
vp2 = <head: EATS; tail: np2>

sent1 = <head: A; tail: L1>
L1 = <head: GIRAFFE; tail: vp2>

Marvin must know the parts of speech of each of the words above, so concepts classifying them must

be learnt or supplied as part of the 'dictionary'. Rather than give the full concept definitions, we list below the dictionary.

Nouns	GIRAFFE, APPLE
Proper Nouns	JOHN, MARY
Determiners	THE, A
Intransitive Verbs	DREAMS, SLEEPS
Transitive Verbs	DREAMS, EATS

The description of the grammar as learnt by Marvin is:

```
list =
[X0:
        X0.val = STOP
v
        [∃ X1: vp(X0, X1)]
]

np =
[X0, X1:
        [∃ X2:
                X0.head = X2
                X0.tail = X1
                pnoun(X2)
                list(X1)
        ]
v
        [∃ X2, X3, X4:
                X0.head = X2
                X0.tail = X3
                X3.head = X4
                X3.tail = X1
                det(X2)
                noun(X4)
                list(X1)
        ]
]

vp =
[X0, X1:
        [∃ X2:
                X0.head = X2
                X0.tail = X1
                X1.val = STOP
                iverb(X2)
        ]
v
        [∃ X2, X3:
                X0.head = X2
                X0.tail = X3
                X1.val = STOP
                tverb(X2)
                np(X3, X1)
        ]
]


sent =
```

```
[X0, X1:
        [∃ X5:
                np(X0, X5)
                vp(X5, X1)
        ]
]
```

The concepts *np* (noun phrase) *vp* (verb phrase) and *sent* (sentence) have two arguments. These concepts attempt to parse a list of words. The first argument represents the beginning of the list which will be recognized and the second argument is the remainder of the list which is left over when the words recognized are removed.

Note that *vp* refers to *np* and *np* refers to *list*. *list* in turn refers to *vp*. These circular references prevent Marvin from learning the definition of *vp* and *list* independently. It is necessary to learn the first disjunct of *list,* then the first disjunct of *vp*. Since *vp* is now known, the second disjunct of *list* may be learnt. Both disjuncts of *np* can be learnt together. Now the definition of *vp* can be completed since *np* is known. Having completed the circle, Marvin can finally learn *sent*.

Although the grammar used here is very simple, it is possible to teach Marvin rules that ensure that, for example, only animate objects may perform actions.

6.2.2 Active-Passive Transformations

The problem of learning the rules for transforming a sentence in the active form to one in the passive has been solved previously by Hayes-Roth and McDermott (1978) and Vere (1978). The problem is: Given the description of pairs of sentences in a transformational grammar, find a set of rules which determine the relationship
between the two sentences.

One example given by Hayes-Roth is

The little man sang a lovely song
∨
A lovely song was sung by the little man.

A graphical representation of this pair is shown in Figure 6.2.

The equivalent representation for Marvin is:

```
noun11 = <nst: man; number: singular>
noun2 = <nst: song; number: singular>
np11 = <det: the; adj: little; noun: noun11>
np22 = <det: a; adj: lovely; noun: noun2>
verb11 = <number: singular; vst: sing; tense: past-part>
aux11 = <auxst: have; tense: present; number: singular>
vp1 = <aux: aux11; verb: verb11; np: np22>
s1 = <np: np11; vp: vp1>

s2 = <np: np22; vp: vp2>
vp2 = <aux: aux11; pb: pb1; verb: verb11; pp: pp1>
pb1 = <pbst: BE; number: singular; tense: past-part>
pp1 = <prep: by; np: np11>
```

In order to learn the transformation rules, Marvin must first understand that a noun construct consists of a noun instance and the number of the noun. A verb has associated with it a number and a tense, and so on. As in the previous learning task, the definitions of noun phrases and verb phrases must also be known.

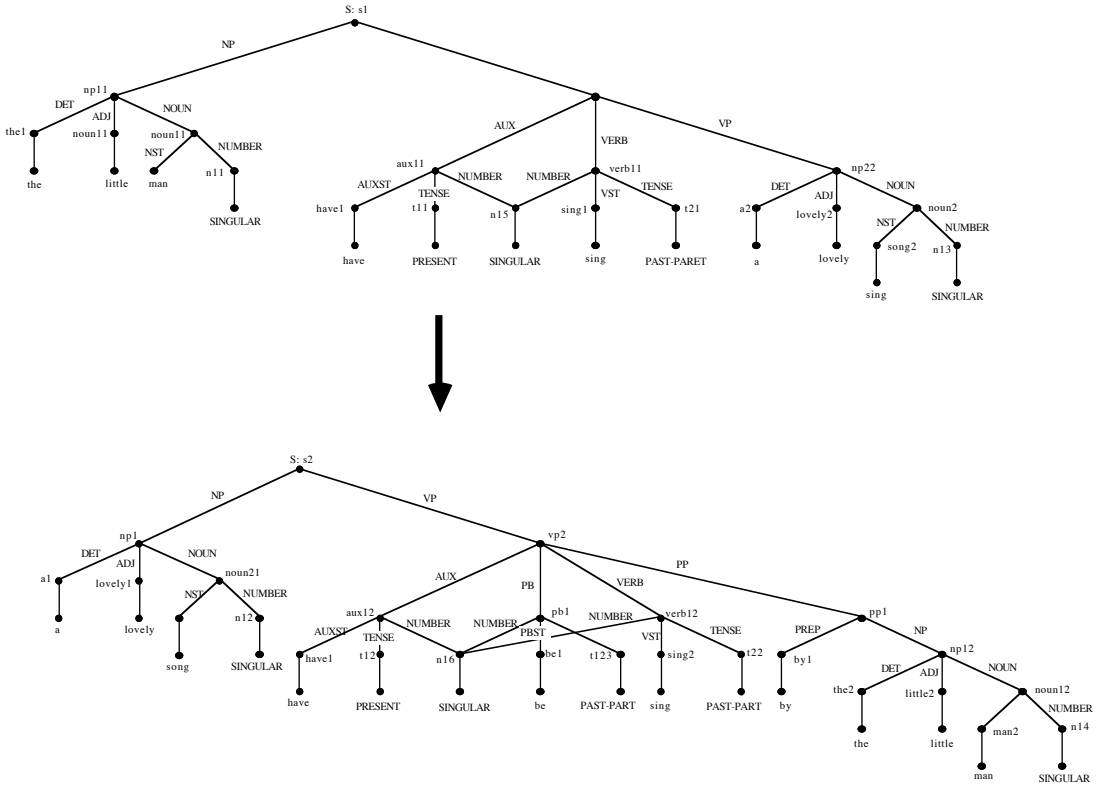


Figure 6.2. Active and Passive forms of a sentence

After generating 18 pairs of sentences to show the trainer, Marvin produced the following rule:

act-pas =
[X0, X1:
[∃ X2, X7, X8, X9, X12, X13, X14,
X15, X16, X17, X18, X21, X22, X23:
X0.np = X2
X0.vp = X8
X8.aux = X9
X8.verb = X12
X12.number = X7
X12.vst = X13
X12.tense = X14
X8.np = X15
X15.det = X16
X15.adj = X17
X15.noun = X18
X1.np = X15
X1.vp = X20
X20.aux = X9
X20.pb = X21
X21.pbst = BE
X21.number = X7
X21.tense = X14
X20.verb = X12
X20.pp = X22
X23.isprep = BY
X22.np = X2
is-number(X7)

```
                aux(X9)
                is-verb(X13)
                is-tense(X14)
                is-det(X16)
                is-adj(X17)
                noun(X18)
                np(X2)
            ]
    ]
```

After a careful examination it can be seen that this is equivalent to the rule learnt by Hayes-Roth as shown in Figure 6.3.

Marvin required 40 seconds to learn this concept. 80% of this time was spent in generating the training examples. For this task, SPROUTER required 30 minutes on a DEC KA-10 processor and THOTH took 30 seconds on an IBM 370/158.

This learning task demonstrates that Marvin is capable of learning a variety of complex production rules that might be used in an expert programming system.

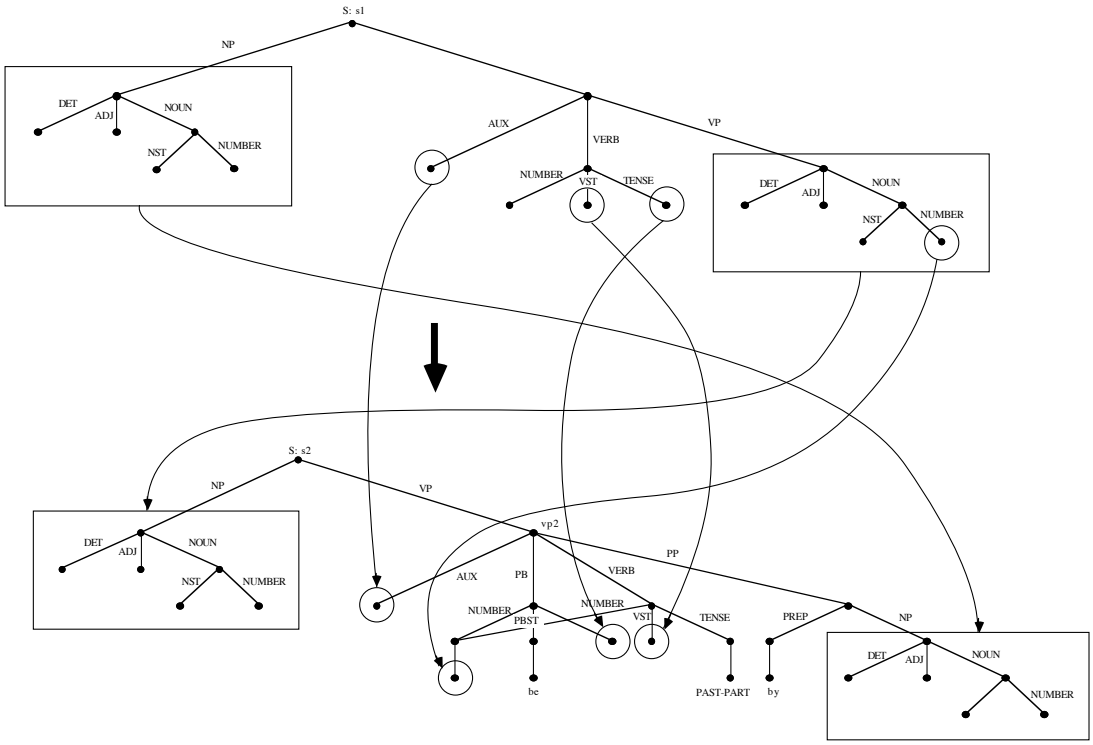


Figure 6.3. Active-Passive Transformation Rule

6.3 Automatic Programming

In Chapter 4 we saw that Marvin can learn concepts which may be executed as programs. These included a program to find the maximum number in a list. This section describes a set of list processing program learnt by Marvin. Among them are: append, list reversal, deleting negative numbers from a list, and a simple insertion sort.

Obviously, Marvin must first learn what a list is. Throughout this section, only lists of numbers will be considered. Given the examples [] and [1], that is the empty list and the single element list containing the number 1, Marvin learns that a list is,

```
list =
[X0:
        X0.val = nil
∨
        [∃ X1, X2:
                X0.head = X1
                X0.tail = X2
                number(X1)
                list(X2)
        ]
]
```

The concept *apppend* requires three arguments, X0, X1, X2. X2 is the result of appending X1 to X0. In the first case, if X0 is nil then X2 is the same as X1. In the second case, the first element of X2 is the same as the first element of X0 and the tail of X2 is obtained by appending X1 to the tail of X0.

```
append =
[X0, X1, X2:
        [∃ X3:
                X0.val = nil
                X2 = X1
                list(X1)
        ]
∨
        [∃ X3, X4:
                X0.head = X3
                X0.tail = X4
                X2.head = X3
                number(X3)
                append(X4, X1, X2.tail)
        ]
]
```

This is the first example in which an identity such as X2 = X1 has appeared. This was created because the same sample object was used for both X1 and X2. Thus the training event ([], L1, L1) where L1 = [1] was presented in order for Marvin to learn the first disjunct. For the second disjunct ([1], [1], [1, 1]) would be adequate.

In order to learn reverse, it is necessary to learn *append1* which appends a single element rather than a complete list as is done by append. Once this has been learnt the following definition of reverse may be learnt:

```
reverse =
[X0, X1:
        X0.val = nil
        X1.val = nil
∨
        [∃ X2, X3, X4:
                X1.head = X2
                X1.tail = X4
                append1(X3, X2, X0)
                reverse(X4, X3)
        ]
]
```

This was learnt in 2.45 seconds after asking 7 questions.

A problem which Biermann posed for the International Workshop on Program Construction (Biermann, 1980) was to produce a program which would delete the negative numbers from a list. For example, if the first argument is, X0 = [-6, 3, -7, -2, 1] then the second argument, X1 = [3, 1]. A number is represented as an object of the form: N = <sign: S; mag: M>. S may be '+' or '-' and M is an unsigned cardinal number which is the magnitude of N. The concept learnt was:

```
delete =
[X0, X1:
        X0.val = nil
        X1.val = nil
∨
        [∃ X2, X3, X4:
                X0.head = X2
                X2.sign = '-'
                X2.mag = X3
                X0.tail = X4
                cardinal(X3)
                delete(X4, X1)
        ]
∨
        [∃ X2, X3, X4, X5:
                X0.head = X2
                X2.sign = '+'
                X2.mag = X3
                X0.tail = X4
                X1.head = X2
                X1.tail = X5
                cardinal(X3)
                delete(X4, X5)
        ]
]
```

This states that if X0 is empty, then X1 is also empty. If the head of X0 is negative number, then X1 is obtained by deleting the negative number from the tail of X0. If the head of X0 is positive then the head of X1 is the same number and the tail if X1 is obtained by deleting the negative numbers from the tail of X0.

The final example we will give in this chapter is a simple insertion sort. First the concept of insertion must be learnt. This is a three place predicate. X0 is a number to be inserted in to the list X1 such that the resulting list, X2 is correctly ordered.

```
insert =
[X0, X1, X2:
        X1.val = nil
        X2.head = X0
        X2.tail = X1
        number(X0)
∨
        [∃ X4, X6:
                X1.head = X4
                X1.tail = X6
                X2.head = X0
                X2.tail = X1
                list(X6)
                less(X0, X4)
        ]
∨
        [∃ X3, X6, X7:
                X1.head = X3
                X1.tail = X6
                X2.head = X3
                X2.tail = X7
                insert(X0, X6, X7)
                less(X3, X0)
        ]
]
```

If X1 is empty then X2 is the single element list containing X0. If X0 is less than the first element of X1 then the first element of X2 is X0 and the tail is X1, i.e. X0 is placed at the front of the list. If X0 is greater than then first element of X1 then, the first element of X2 is the first of X1 and X0 is inserted into the tail of X1 to produce the tail of X2.

An insertion sort works by taking each element from its first argument, X0, and *insert*ing it into the second argument, X1. When X0 is empty the entire sorted list will be in X1. X2, the third argument which returns the result will have the final value of X1 assigned to it.

```
sort =
[X0, X1, X2:
        X0.val = nil
        X2 = X1
        list(X1)
 v
        [∃ X3, X4:
                X0.head = X3
                X0.tail = X4
                X4.val = nil
                insert(X3, X1, X2)
        ]
v
        [∃ X3, X5, X6:
                X0.head = X3
                X0.tail = X5
                insert(X3, X1, X6)
                sort(X5, X6, X2)
        ]
]
```

When choosing examples to show Marvin, the trainer should look for the simplest events possible. By minimizing the amount of data that must be processed, he makes Marvin's task much easier.

The simplest example that satisfies the last disjunct of *sort* is ([1], [], [1]). The primary statement generated by this event are:

```
X0.head = X3
X3.left = none
X3.right = X4
X4.val = 1
X0.tail = X1
X1.val = nil
X1.head = X3
X2.tail = X1
```

There are not enough variables present in this set of statements to construct the concept. [When the event is being recognized, the same object is bound to more than one variable].

There is a way of avoiding this problem in *sort*. The more general case will be considered in Chapter 7. If the event ([1, 1], [], [1, 1]) is shown as the trainer's example, the primary statements generated will contain enough variables. However, another problem arises. *Insert* and *sort* must both recognize components of the event so that the statements insert(X3, X1, X6) and sort(X5, X6, X2) can be created. Thus insert(1, [1], [1]) must be true and sort([1], [1], [1, 1]) must be true.

This is the reason that a redundant disjunct must be learnt. The second disjunct is finally unnecessary because the case it covers is also covered by the more general third disjunct. But the second one must be learnt in order to learn the third.

6.4 Concepts that Marvin cannot learn

The difficulty of learning *sort* leads us to discuss a limitation of Marvin. That is, existentially quantified variables are only created when they represent some part of the training example. There are concepts where this is insufficient.

A simple example is the ordering of decimal digits. Suppose the following concept is known:

```
lessd =
[X0, X1:
        X0.val = 0 ∧ X1.val = 1
∨       X0.val = 0 ∧ X1.val = 2
                .......
∨       X0.val = 8 ∧ X1.val = 9
]
```

This describes which digits are adjacent to each other in numerical order. However, it does not give a complete ordering. Marvin must learn the additional disjunct:

$$[\exists\ X2\text{: lessd}(X0, X2) \wedge \text{lessd}(X2, X1)]$$

That is, X0 is less than X1 if there is an X2 such that X0 is less then X2 and X2 is less than X1. The trainer only shows the arguments X0 and X1. For example, (0, 2) may be shown resulting in the primary statements:

$$X0.val = 0 \wedge X1.val = 1$$

There is insufficient information in the example to instruct Marvin to create the additional variable which would allow the two *lessd*'s to be inferred.

Possible solutions to this problem will be discussed in Chapter 7.

6.5 Summary of Results

Table 6.1 contains results obtained from the measurement of Marvin's performance while learning the examples described above.

The total time required to learn a concept was measured. The object generation procedure accounts for a significant amount of the total time so the percentage of the time devoted to generating objects was obtained. We can get some idea of how easily Marvin learnt a concept by counting the number of questions it had to ask the trainer before it discovered the target concept. Another indication of the difficulty of the concept is the number of times the trainer answered 'no' to a question. This is the number of times Marvin generated an inconsistent trial.

In columns 3 and 4, separate figures are given for each disjunct in the concept. The proportion of time spent generating objects for *vp* is given as two figures because this concept was learnt in two steps.

Absolute times are not necessarily a good indication of a program's performance because they will vary greatly depending on the hardware and the programming language used to implement the system. However, the important thing to note from the times given is that Marvin provides rapid real-time response to the trainer. This is essential if the learning system is to be used to acquire knowledge for an 'expert' programming system. In this case the program must provide a comfortable working environment for the human expert who may not be familiar with computers. The times also demonstrate that a system such as Marvin is capable of learning quite complex concepts in a reasonable time. Thus it seems likely that the techniques used in Marvin may be useful in developing practical systems such as knowledge based expert programs.

The next point to note from the figures in Table 1 is that the object generation process accounts for a very large proportion of the learning time. There is considerable variation in the percentage of time because the complexity of the objects varies for the different tasks. However, the performance of the program can obviously be improved if the theorem prover in Marvin were to be speeded up.

One measure of the efficiency of a concept learning system was suggested by Dietterich and Michalski (1977). That is to find the proportion of generalizations produced which are consistent with

the target. That is: how many times did the trainer answer 'yes' compared with the number of times he said 'no'. On the average 60% of the generalizations made by Marvin are consistent. This compares very well with other learning programs. Partly, this is due to the fact that the generalizations made by Marvin are fairly conservative. That is, small parts of a trial concept are changed at any time. If a mistake is made, Marvin usually does not have to look very far before a consistent trial is found.

From these results we see that the basic ideas behind the learning algorithm are sound. The next question is how efficiently have these ideas been implemented.

Table 1: Summary of Marvin's performance on test cases.

Task	Total Time	%Object.Gen.	Questions	Inconsistent
Lessd	0.08	27.3	2	2
Less	4.80	17.9	4:6:3	2:2:1
Max	5.83	35.5	1:5:5	0:1:1
List	0.18	27.3	0:2	0:0
Append	1.82	37.6	3:8	1:3
Reverse	2.45	44.9	2:5	2:3
Delete	1.45	48.7	0:3:3	0:1:1
Insert	7.27	35.5	6:3:4	0:1:1
Sort*	47.35	76.1	10	6
EastBound	14.18	58.0	20	4
Arch	6.33	46.3	11	5
On-top-of	0.63	60.5	5	1
vp	1.3	16+27.1	1:5	0:1
np	0.57	41.1	2:3	0:0
word-list	0.24	45.6	0:3	0:1
sentence	6.30	24.3	9	1
Active-Passive	40.52	80.3	18	3

* Measurements for *sort* are given only for the final disjunct.

6.6 Efficiency of Implementation

A good method for discovering where a program's weaknesses lie is to count the number of times each procedure is called. This allows us to find out if time is being wasted in some parts of the program. The Berkeley Pascal compiler running under UNIX/32V on VAX computers provides this information when a program is profiled. This was done while Marvin learnt the number concepts presented in Chapter 4. Some of the results are presented in this section.

Five concepts were learnt ranging from the very simple definition of *digit* to the rather more complex concept, *maximum*. In all 11 training examples were shown to Marvin, one for each conjunction. Table 6.2 shows the number of times the major procedures in Marvin were called. The bar graph in Figure 6.4 shows more clearly which procedures dominate the Marvin's time. Obviously the statement matching procedures are the most used procedures.

The learning algorithm uses a relatively inefficient method for generating a new statement. The program scans through a trial description in linear order, using each statement as a focus for its search for new concepts. Since a concept usually contains more than one statement, it is possible that the concept will be tried several times. In fact, almost 50% of the attempts to generate a new statement resulted in a statement that had already been tried before. [This can be seen from the fact that *TriedBefore* was called 60 times but *prove,* which is called after *TriedBefore* returns true, was called only 29 times].

Another reason for the excessive use of the pattern matcher is the fact that a depth first search is used to try to match statements in a stored concept with statements in the trial.

The next highest frequency in the histogram is due to the object generation procedure *prove*. Remember that in order to generate an object which is useful to the learning algorithm, the program creates an object, then tests to see if any of the removed statements are true. If one is true, then a new object must be created. In fact 60% of the objects generated are rejected in this way.

Table 2: Requency of Procedure Calls

Count	Procedure
1	marvin
29	prove
1391	addtrail
121	clearq
284	SaveEnv
2463	unbound
3903	valofq
1269	get
8361	valueof
568	equal
1018	equiv
298	mkbind
499	mkqvars
284	call
126	backtrack
238	succeed
14	eval
273	falsified
96	denied
96	execute
29	ToBeDenied
363	simplified
11	primary
44	describe
67	MakeStatement
2129	bind
9940	isbound
2129	RecordSubst
21368	ForgetSubst
28403	unify
21356	match
13	newassoc
43	index
87	lookup
1485	restore1
1423	replace
1413	restore
11	generalize
29	contained
87	TryConceptsWith
116	CheckConcept
60	TriedBefore
129	findargs
31	NewStatement
31	OccursIn
73	CanNotRemove
49	NoOtherRef
47	ParentIn
55	NoSpec
60	CreateStatement
31	FindRemovable
31	restricts
31	NotRelaxed
1694	Contains
29	qualified
1	relax
23	TryUnRemoved
1	restricts2
31	MoreGeneral
109	simplify
11	create
11	remember
11	cleanup
11	LearnConj
5	LearnedCons
5	learn
1	init

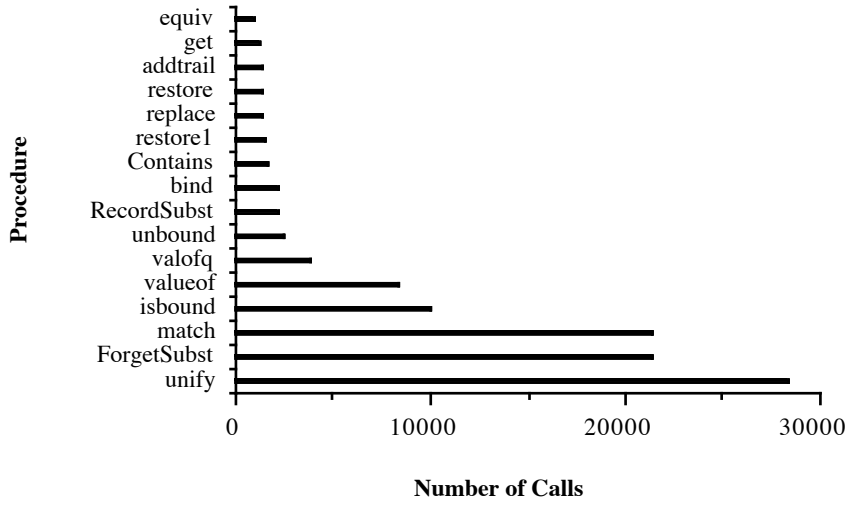


Figure 6.4. Frequency of Procedure Calls

6.7 Comparisons with other systems

Marvin is the only general purpose concept learning program that is capable of generating its own training examples. Some domain specific programs do have this ability (Popplestone, 1970; Lenat, 1977). Cohen (1978) and Mitchell (1978) have both proposed methods for a learning program to generate its own training instances. The main advantage of being able to do this is that the program can control its focus of attention without relying on the trainer. This is also of importance in Lenat's AM system which discovers theories in mathematics by proposing its own 'agenda' of interesting things to be explored.

Since Marvin is a descendent of CONFUCIUS, the two programs share certain characteristics. Among them is the emphasis placed on 'growing' description languages. Since concepts, once they are learnt, are stored in memory and may be used in future concept descriptions, the descriptive power of the language grows with time. Indeed this ability is necessary if recursive concepts are to be learnt. Recursion is also a feature which distinguish Marvin and CONFUCIUS from systems such as INDUCE (Larson, 1977) and Thoth (Vere, 1978). Michalski (1980) claims that it should be avoided because recursive descriptions are difficult for humans to read. However, to learn abstract concepts concerning, for example, lists or numbers, recursion is necessary.

Dietterich and Michalski (1977) have suggested a number of characteristics of learning systems which serve as points of comparison between the programs. Let us list these characteristics as they apply to Marvin.

Intended Application:

general.

Marvin is not restricted to any particular domain. It can learn concepts in a wide variety of environments.

Language:

Marvin's description language is first order logic with quantifiers. Connectives in the language include conjunction and disjunction.

Generalization Rules:

A number of different types of generalizations rules are proposed by Dietterich and Michalski.

Dropping Condition:

This rule involves removing statements from conjunctions. In Marvin's case the dropping rule must be modified slightly to a 'replacement rule'. This is the main generalization method used.

Turning constants into variables:

When Marvin sees a description of an object as a list of property/ value pairs, it constructs a logical description in which the object names are replaced by variable names. The values of these variables can then be generalized by the replacement procedure.

Climbing Generalization:

This is the sort of rule which enables the program to deduce that if an object is a triangle, then it also belongs to the class of polygons. Being a polygon, it is a plane geometric figure, etc.This is achieved in Marvin by learning the various classifications above as concepts.A triangle would then be recognized by all of those concepts.

Efficiency:

This is the most difficult comparison.As we have already mentioned, the measure used by Dietterich and Michalski compares the number of generalizations made with the number that were actually used in the final concept. On the average, in the programs they analyzed, about 30% of the generalizations made were useful. The nearest comparison that can be made with Marvin is the ratio of the number of training examples generated to the number of examples which were found to be in the target concept. The average over the various learning tasks described indicates that about 60% of the trial concepts produced are consistent.

Extensibility:

- Applications: Marvin has not been used to develop any concepts for expert programs, although we expect that the techniques used will be applicable to developing knowledge based programs.
- Marvin can learn disjunctive as well as conjunctive concepts.
- No special mechanism has been included to deal with *noisy data*. However, bad data are placed in separate disjuncts.
- Domain Knowledge: This can be learnt as preliminary concepts, which may then be used to describe other concepts.
- Constructive Induction: According to Dietterich and Michalski, most programs produce descriptions which involve the same descriptors which were present in the initial data. Such programs perform non-constructive induction. A method performs constructive induction if it includes mechanisms which can generate new descriptors not present in the input data. To a certain extent Marvin is able to do this since knowledge stored in its memory is used to augment the description of a training instance. However, there is no meta-language which would be able to perform true feature extraction. This problem will be discussed further in Chapter 7.

6.8 Conclusion

Marvin can learn concepts which can be described in first order predicate logic with quantifiers. It cannot learn concepts with the logical negation. It cannot learn concepts which require existentially quantified variables which cannot be derived directly from the training instances.

The present implementation has shown that the learning algorithm works very well considering the experimental nature of the program. There are a number of deficiencies which could be overcome in a new implementation.

7
Future Directions

In Chapter 1 we defined a set of objectives for the research described in this thesis. From the results given in Chapter 6 we can see that Marvin has satisfied these objectives, but of course, there is always more work to be done. This chapter defines some new goals for further work in concept learning. Some of the suggestions will be aimed at improving the present implementation of Marvin. Others indicate ways of extending the program so that it will be capable of learning more complex concepts in more complex environments.

7.1 The Language

Expressions in Marvin's description language specify a class of objects in the universe. An object is distinguished by the values of its properties.

In the current language an object is input by listing its property value pairs. This is not always a convenient form of description. For example, to describe a relationship such as 'father', we might use the following:

Jack = <age: 38; son: Bill>
Bill = <age: 12; father: Jack>

An alternative method for presenting the object descriptions is to enter a series of predicates which correspond to what are now the primary statements. In this example the single predicate *father(Bill, Jack)* would eliminate the need to specify the values of two properties in two different objects. Note, however, that we still consider the objects as being characterized by the values of certain properties, so the theoretical basis for this language is consistent with Banerji (1969) and Cohen (1978). These changes bring the language nearer to the notations used by Vere and Michalski.

Let us take the description of ARCH as a further example:

supports(side1, top)
∧ supports(side2, top)
∧ left-of(side1, side2)
∧ ~ touch(side1, side2)
∧ shape(top, BRICK)
∧ ...

The identifiers *top, side1* and *side2* are the names of objects and *BRICK* is an atom. It is up to the learning program to substitute variables for those names so that the values may be generalized. An advantage of this notation over the present one is that it is more convenient for expressing relationships between objects. This can be seen by comparing the description of ARCH given here and the description in Chapter 6.

If we assume that Marvin has a pre-processor attached to it to perform basic pattern recognition of a scene, then the input to Marvin will be the results of the measurements performed by the pre-processor. The primary statements, such as *supports(side1, top)*, may be considered as describing those results. Sets are not strictly necessary in a description language, since they can be simulated by list objects. However, they are useful because they allow events consisting of a number of parts to described very succinctly. It is possible to represent sets implicitly. This is illustrated by the example,

son(Mary, Fred) ∧ son(Mary, Bill)

This expression describes a set { Fred, Bill} which is the value of Mary's property *son*.

In this version of the language, there is no longer an explicit representation of objects (and sets), so how can the learning program show the trainer an example? Suppose the program is trying to build a number:

```
number =
[X0:
        [∃ X1:
                  left(X0, null)
                ∧ right(X0, X1)
                ∧ value(X1, 1)
        ]
∨
        [∃ X1, X2:
                  left(X0, X1)
                ∧ right(X0, X2)
                ∧ number(X1)
                ∧ number(X2)
        ]
]
```

left, right and *value* are all primary statements. When ∃ X1 is encountered, the program may create a new symbol, say *OBJ1* to represent the value of X1. To execute a primary statement such as *value(X1, 1),* the variables present are replaced by their values, giving *value(OBJ1, 1)*. The result is then placed on a stack. All the expressions on the stack represent the description of the object being generated. The description on the stack must always be consistent; thus there must not be two expressions such as

value(X1, 0) ∧ value(X1, 1)

present simultaneoulsy. On the other hand it is acceptable to construct two statements such as,

son(Mary, Fred) ∧ son(Mary, Bill)

The difference is that the value of the property *value* is expected to be a single value, whereas the value of the property *son* is a set. The type of the value of a property must be supplied by the trainer as domain knowledge so that the interpreter knows how to maintain the consistency of the stack. This method of generating objects can be compared with the implementation of the WARPLAN problem solver (Warren, 1974).

Note that the equivalence relation '=' has been eliminated from the language. This is no longer necessary as a built in relation. It is possible to determine the similarity of objects by learning an 'equal' concept for each type of object.

7.2 Generating Statements

In the present implementation, the pattern matcher (i.e. the statement generation procedure) and the search strategy (the learning algorithm) are combined. As could be seen from the profiled program in the previous chapter, a great deal of redundant pattern matching is performed. In fact, 50% of the statements generated had been generated at least once before.

Marvin's performance can be improved if the pattern matcher and search strategy are separated into co-routines and a discrimination net is used to speed up the statement indexing. The structure of the discrimination net and how it is used will be described in this section.

Currently, when a complete conjunction has been learnt, its statements are entered into an index which is represented by a linear list of associations. The associations are between statements and the concepts in which they appear. A fairly obvious way of improving the search time is to replace that list with a more sophisticated mechanism. Instead of maintaining a single list, we will keep a list for every constant known to Marvin. The list for constant, *X,* will contain the associations for all statements which contain *X*. When we want to look up a statement, we take each constant in the statement and find the intersection of all the lists associated with them. This results in a small set of statements (usually only one) which will make finding a match much easier.

For example, let's create an index for the statements:

colour(X0, red) (S1)
colour(X1, green) (S2)
size(X0, big) (S3)
size(X1, big) (S4)

The result is the set of associations below.

colour:	S1, S2
red:	S1
green:	S2
size:	S3, S4
big:	S3, S4

If we want to look up a statement *colour(X, green)* we find the entries for the constants *colour* and *green* which appear in the statement. We then find the intersection of { S1, S2} and { S2} which are the lists associated with the *colour* and *green*. This results in the set of statements which could match *colour(X, green)*. The set is { S2} . We have just discussed indexing for the statements of concepts in memory. This index is permanent since data can only be added to it. Marvin does not, at present have any form of indexing on the statements in the trial concept. Since the trial is searched regularly such an index would be very useful. This index would only be temporary. Once a conjunction has been learnt, its index may be removed.

Remember that the goal of the statement generating procedure is to find conjunctions stored in memory which are subsets of the trial. The data structure we propose to use will contain a list of references to the candidate conjunctions. Associated with each conjunction will be the statements contained in it. Associated with each of the statements in the conjunction will be a list of the statements in the trial which match it, along with the bindings resulting from the match.

Consider the following example:

```
digit =
[X0:
        value(X0, 0)                    (D1)
∨
        value(X1, 1)                    (D2)
]

number =
[X0:
        [∃ X1:                          (D3)
                left(X0, null)
              ∧ right(X0, X1)
              ∧ value(X1, 1)
        ]
∨
        [∃ X1, X2:                      (D4)
                left(X0, X1)
              ∧ right(X0, X2)
              ∧ number(X1)
              ∧ digit(X2)
        ]
]
```

The labels *D1 .. D4* refer to the disjuncts of each concept. If the primary statements in a trial include the following:

```
left(X0, X2)
left(X2, null)
right(X2, X3)
value(X3, 1)
right(X0, X4)
value(X4, 0)
```

then the index built up will be:

D4	left(X0, X1)	left(X0, X2)	{ X0/X0, X1/X2}
		left(X2, null)	{ X0/X2, X1/null}
	right(X0, X2)	right(X2, X3)	{ X0/X2, X2/X3}
		right(X0, X4)	{ X0/X0, X2/X4}
D3	left(X0, null)	left(X2, null)	{ X0/X2}
	right(X0, X1)	right(X2, X3)	{ X0/X2, X1/X3}
		right(X0, X4)	{ X0/X0, X1/X4}
	value(X1, 1)	value(X3, 1)	{ X1/X3}
D2	value(X0, 1)	value(X3, 1)	{ X0/X3}
D1	value(X0, 0)	value(X4, 0)	{ X0/X4}

The index tells us that there are four statements in the trial which have matches in D4, the second disjunct of *number*. This gives us reason to think that some part of the trial may be recognized as a number. However, only two statements in *number* match the four statements in the trial. The second column indicates which statements in D4 were matched. Column three shows the corresponding statements in the trial. Both *left(X0, X2)* and *left(X2, null)* match the same statement. The substitutions resulting from each match are shown in the last column.

Since only two of the four statements in *D4* can be matched, this disjunct cannot be true. On the other hand all the statements in *D3* have been matched - one of them twice. If we can find a consistent substitution among the matched statements, then *D3* is true. By 'consistent' we mean that a variable may appear only once on the left hand side of a substitution and only once on the right. Our problem is which of the two *right* predicates do we want to match? The statement, *value(X3, 0)* creates a substitution, {X1/X3} . This conflicts with the substitution {X1/X4} present for *right(X0, X4)*. The substitutions for *left(X0, null), right(X2, X3)* and *value(X3, 0)* can be combined without conflicting. Therefore, these statements are the implicants of *D3* with the substitution {X0/X2, X1/X3} .

Since a complete disjunct of *number* is satisfied, *number(X2)* is a new statement that can be tested by replacing its implicants. *digit(X3)* and *digit(X4)* are also new statements. There is only one possible substitution in the case of *number(X2);* however, it can often happen that more are possible. The program must therefore try all combinations.

Since new statements have been created, these may also be added to the index. They will appear as new entries for D4.

D4	...	...	...
	number(X1)	number(X2)	{X1/X2}
	digit(X2)	digit(X3)	{X2/X3}
		digit(X4)	{X2/X4}

With these additions, all the statements in D4 have been matched. The only consistent substitution is { X0/X0, X1/X2, X2/X4} . Thus *number(X0)* can be generated and also added to the index and the process may continue. This method of generating statements has already been implemented, although it has not been integrated into Marvin. It is significantly faster than the old method of generating statements.

7.3 Generating Objects to Show the Trainer

At present, when Marvin shows a example to the trainer, it generates a complete object and then checks it to ensure that none of the removed statements are true. If one is true, the interpreter backtracks to find a new object that satisfies the trial. Backtracking only returns one level on the control stack, so it may happen that the property which caused a removed statement to be true remains unchanged. Therefore, the new object will fail again. When two failures in a row are due to the same removed statement, the interpreter backtracks deeper into the stack until the correct alternative is found. This is very inefficient.

Before trying to generate objects, Marvin creates two lists. One is the list of statements which must be true, and the other is the list of statements which must be false. To improve the program's performance, these two lists could be merged.

Suppose the description of an object, X includes the statements:

colour(X, Y) ∧ value(Y, red)

The colour may be generalized by removing the statement *value(Y, red)* and replacing it with *any-colour(X)*. In the present system, the entire example would be constructed before checking that *any-colour* did not make X red. Instead Marvin could create a description:

colour(X, Y) ∧ any-colour(Y) ∧ ~value(Y, red)

Immediately after *any-colour* assigns Y a value, we execute ~ *value(Y, red)* to make sure that Y is not red.

The new strategy for constructing objects requires that statements involving a variable, *X,* which must be false, will be placed immediately after the positive statement which assigns *X* a new value. The program must still backtrack, but it will not be the blind backtracking currently being done.

7.4 Learning Logical Negation

Programs such as Vere's Thoth (vere, 1980) learn counterfactuals (or exceptions to the rule) from negative examples. When Marvin tests an inconsistent generalization it generates negative examples. Thus, it may be possible to use these to learn predicates which must not be true.

When an inconsistent generalization is made, Marvin tries to make the trial more specific. Suppose the trial,

colour(X, Y) ∧ any-colour(Y)

is inconsistent. The example shown to the trainer may have Y as blue, which the trainer says is incorrect. One way of making the trial more specific is by taking the statement *value(Y, blue)* which is in *any-colour* and negating it:

colour(X, Y) ∧ any-colour(Y) ∧ ~value(Y, blue)

Usually, the addition of positive information will result in a better restriction of the trial. However, if no positive information is available, Marvin could try adding the negation of the disjunct of the concept referred to by the statement which created an unacceptable example.

This method has some problems which must be studied further. For example, to test the restricted trial, Marvin may show the trainer a green object which is acceptable. This does not necessarily indicate that the new trial is consistent. *Black* may also be a colour which is not allowed, but Marvin hasn't tested that yet, so it cannot assume that blue is the only exception. This problem is similar to the one we discussed in Section 3.5 when we wanted to generate an instance of an inconsistent trial which did not belong to the target.

7.5 Problems with Quantified Variables

Suppose we want to teach Marvin the ordering of the decimal digits 0..9. The first step is to learn that 0 comes before 1, 1 before 2 etc.

```
lessd =
[X0, X1:
        value(X0, 0) ∧ value(X1, 1)
∨       value(X0, 1) ∧ value(X1, 2)
∨       ...............................
∨       value(X0, 8) ∧ value(X1, 9)
]
```

The final disjunct that must be learnt is,

[∃ X2: lessd(X0, X2) ∧ lessd(X2, X1)]

To teach this disjunct the trainer might show Marvin the example (1, 3) which would result in the primary statements,

$$\text{value}(X0, 1) \wedge \text{value}(X1, 3)$$

Unfortunately, this pair does not match any of the conjunctions in lessd, so how is it possible to learn that there exists a digit in between the two given?

The problem is that the input does not provide enough information to make the connection between the two digits. One solution is to require the trainer to provide additional objects as 'hints' to guide the program. If the digit 2 is supplied as an extra piece of information then Marvin, as implemented already could learn the concept. This is similar to Vere's approach with background information (Vere, 1977).

In Marvin's case, this solution is not very desirable because it places too much responsibility on the trainer. A second alternative is to modify the statement generation procedure. A new statement can be introduced only if *all* the statements in one disjunct are matched. However, if we allow *partial* matching then more statements can be generated. For example, the primary *value(X0, 1)* will match statements in the first and second disjuncts of *lessd*. In both cases there is no object which will the satisfy the other statements in each conjunction. However, when a partial match occurs, Marvin may postulate the existence of new objects which satisfy the conjunctions. For example,

$$[\exists\ X2, X3: \text{lessd}(X2, X0) \wedge \text{lessd}(X0, X3)]$$

When the new objects, X2 and X3, are created by a partial match, they must be able to participate in other matches. For some values of X3, lessd(X3, X1) will be true. It must be possible to discover this since it will result in the target concept. One way of allowing X2 and X3 to be used in further pattern matching is to generate instances of them by executing *less(X2, X1)* and *less(X1, X3)* as was done by the learning system. The descriptions of the instances of X2 ad X3 may then be generalized in the same way that the descriptions of input objects are generalized.

There is one very difficult problem with the partial matching approach. When learning a complex concept, many unwanted statements will be generated. To demonstrate this, consider the concept *quicksort*.

```
sort =
[X0, X1:
        value(X0, nil) ∧  value(X1, nil)
∨
        [∃ X2, X3, X4, X5, X6, X7, X8:
                head(X0, X2)
              ∧ tail(X0, X3)
              ∧ head(X4, X0)
              ∧ tail(X4, X5)
              ∧ partition(X2, X3, X6, X7)
              ∧ sort(X6, X8)
              ∧ sort(X7, X5)
              ∧ append(X8, X4, X1)
        ]
]

partition =
[X0, X1, X2, X3:
          value(X1, nil)
        ∧ value(X2, nil)
        ∧ value(X3, nil)
        ∧ number(X0)
∨
        [∃ X4, X5, X6:
                head(X1, X4)
              ∧ tail(X1, X5)
              ∧ head(X2, X4)
              ∧ tail(X2, X6)
```

```
                ∧ less(X4, X0)
                ∧ partition(X0, X5, X6, X3)
        ]
∨
        [∃ X4, X5, X6:
                  head(X1, X4)
                ∧ tail(X1, X5)
                ∧ head(X3, X4)
                ∧ tail(X3, X6)
                ∧ less(X0, X4)
                ∧ partition(X0, X5, X2, X6)
        ]
]
```

The sorted version of a list, X0 is X1. *Sort* works by taking the tail of X0, that is X3, and partitioning it into two lists, X6 and X7 such that X6 contains all the elements of X3 which are less than X2, which is the head of X0. X7 contains all the elements greater than X2. X6 and X7 are then sorted giving X8 and X5 respectively. Finally the completely sorted list, X1, is obtained by appending X8 and X4. This joins the two smaller, sorted lists with X2 in the middle. The definition of *partition* is given without further explanation.

Because there are several intermediate steps in *sort,* quite a few variables must be used to transmit information from one predicate to another. No example given by the trainer can provide the the necessary information to generate these statements unless partial matching is used.

Suppose Marvin is trying to create the two *sort* predicates. We will assume that the only conjunction of *sort* which is in memory at present is the first one which expects both arguments to be nil. At least four empty lists must be present to generate the recursive calls to *sort*. However, since the trainer only showed two lists to Marvin there can only be two empty lists in the input. These can form the basis for some partial matches. That is, new lists whose values are nil would be created in order to satisfy the first disjunct of *sort*. However, these new lists may also be used in more partial matches producing other *sort* predicates and also new *partition* predicates which involve three null lists. All the lists created could participate in still more matches, and so on.

The learning process is a search for the most appropriate set of predicates to describe a concept. While we insist on all-or-nothing pattern matching the search space remains bounded. However, when partial matching is introduced, the search space is potentially infinite. If partial matching is going to be used then some means of directing the search must be found.

When students in Computer Science are taught the quicksort algorithm, they already know what a sorted list is. They probably also know a simple sorting algorithm such as an insertion sort. Since the goal of the quicksort is clear, it should be easier for them to understand the reason for the various steps involved. Perhaps we should not expect the machine to learn complex and efficient descriptions of concepts on the first attempt. If a naive definition is learnt first, this may provide Marvin a way of restricting its search. With the additional information provided by some prior knowledge of the concept it may be possible to evaluate which matches are more likely to be useful in building the target concept.

7.5 Learning Universal Quantifiers

Suppose we show Marvin the training example, (5, { 4, 1, 3, 2}). Part of a trial which may be generated is:

```
value(X1, 5)
member(X2, X1)
less(X1, X0)
member(X3, X1)
less(X3, X0)
............
```

For each element of the set X1 there is a matching set of statements,

$$\text{member}(X, X1) \wedge \text{less}(X, X0)$$

Thus it is possible to generalize the trial by replacing all of those statements by

$$[\forall X: \text{member}(X, X1) \wedge \text{less}(X, X0)]$$

To do this, the pattern matcher used in statement generation may be asked to look for matches within the trial as well as within concepts in memory. If a number of matches can be made, all with consistent bindings, then the *forall* statement may be attempted.

Discovering which statement implies the other within the *forall* statement may present some problems to the object generator. If the two predicates in the example above are swapped, then when executed left to right, Marvin may start producing an infinite set of numbers less than *X0* and testing them for membership in *X1*. In fact it should select the elements of *X1* and then perform *less(X, X0)*. Both predicates specify a range of values for *X,* but *member(X, X1)* describes a subset of *less(X, X0)*. Thus *member(X, X1)* implies *less(X, X0)*.

If a set A is a subset of another set, *B,* then *B* must contain objects not in *A*. To determine which statement should imply the other, Marvin can use one predicate to try to generate an object not in the other, just as it does already when it creates training example to show the trainer.

7.6 Feature Extraction

One weakness of Marvin is that it still must trust the trainer to teach it concepts in an order that will ensure that the memory is well structured. Let us see if there is a way of making Marvin more autonomous.

We have discussed how partial matching can be used between statements in the trial and memory to learn *sort*. Partial matching can also be performed between statements in the trial and themselves to learn *forall* statements. Now let consider matching concepts in memory with other concepts in memory.

Although Marvin has control over its own training examples, it has no control over the order in which concepts are learnt. The present algorithm is sensitive to this order, so Marvin must rely on the trainer to choose the order correctly; otherwise the memory organization would become unstructured. An algorithm can be designed which is insensitive to the fact that the memory is not well structured, however, a better solution might be to provide Marvin with a mechanism for reviewing its memory and restructuring it if necessary.

A partial matching procedure would allow Marvin to compare concepts it has learnt. If two concepts contain a common subset of statements, then this subset can be made into a new concept. The statements in the first two concepts can be replaced by a single statement referring to the third concept.

For example, when Marvin was taught *on-top-of,* we assumed that *flat* would have to be learnt before *any-shape*. This time let's do it in reverse order. Marvin first learns that *any-shape* is

$$\begin{aligned} & \text{value}(X, \text{red}) \\ \vee\ & \text{value}(X, \text{table}) \\ \vee\ & \text{value}(X, \text{sphere}) \\ \vee\ & \text{value}(X, \text{pyramid}) \end{aligned}$$

and later it learns that *flat* is,

$$\text{value}(X, \text{flat}) \vee \text{value}(X, \text{table})$$

Before *flat* is stored in the memory, Marvin performs some pattern recognition. It discovers that part of *any-shape* matches *flat,* so matching statements are replaced by a reference to *flat*.

$$\text{flat}(X) \vee \text{value}(X, \text{sphere}) \vee \text{value}(X, \text{pyramid})$$

If the new concept did not completely match another concept, but had some statements in common, then those common statements could be extracted to form a third concept. This process ensures that Marvin's memory is always well structured.

7.7 Learning to Learn

A criticism that may be levelled at Marvin is that its generalizations are too conservative. If it is learning a concept which involves quite complex objects, many properties of the object, such as colour, may be irrelevant. Yet Marvin must generalize colour before moving on to higher level generalizations. This is attributable to the specific-to-general nature of Marvin's search strategy. That is, the initial hypothesis describes only a limited set of objects, and the cover of the concept is very gradually expanded. In contrast, Meta-DENDRAL uses a general-to-specific search (Mitchell, 1978) which starts with the most general concept that can be generated and then proceeds to make this description more specific.

When a human looks at an object he usually focuses on the important details first because he has learnt that some properties, say its colour or texture, are not likely to be distinguishing features. The search strategy of the learning algorithm may be made extended so that it can learn, over a period of time, which properties should be tested and which ones it can generalize without testing.

If Marvin has often found that the specific colour or texture of an object could be generalized to any colour or texture then the next time it sees an object which has those properties, it immediately introduces the concepts 'any-colour' and 'any-texture' without testing them. This could be done using a relatively simple mechanism. Each concept may have associated with it a 'score' for the number of times it could has been introduced into trial concept without being restricted. That is, if a replacement during the learning process introduces a statement which results in a consistent generalization, then the concept referred to by that statement is given a higher score. If the generalization was inconsistent, then the score is decreased.

Suppose the trainer shows Marvin an object which has colour and shape. Because it has already learnt in *on-top-of* that the colour could be generalized to any colour it may assume that the same can be done immediately for the new concept. Since the shape of objects had to be restricted in *on-top-of,* it is reasonable to assume that the shape will have to be tested in the new concept as well.

If Marvin's assumptions are correct then the scores for *colour* and *shape* can be adjusted to reinforce the idea that shape is a more important distinguishing feature than colour. However, if the assumption did not work, then the score for 'colour' would have to be decreased, and the colour of the object must be tested.

Note that this strategy involves risks. If the concept to be learnt conforms to Marvin's assumptions about the world then the concept will be learnt more quickly than if it had used the present, conservative algorithm. However, a consequence of the new method is that more than one property will be changed when a new example is shown to the trainer. If the example is a negative instance, more work will have to be done to make the trial more specific because we don't know which property was responsible for the inconsistent generalization.

The best case for the new algorithm gives a performance which is substantially better than the conservative version. However, the worst case may result in a worse performance. Bruner, Goodnow and Austin (1956) describe a *Focus Gambling* algorithm used by some of the human subjects in their tests. This method corresponds closely to the suggestions made here.

7.8 Summary

No research effort is ever complete, since there are always many more problems that need to be solved. Among them are:

- The description language used by Marvin is limited in a number of respects. Sometimes it is difficult to neatly express relational descriptions. There is no built in set concept which would reduce the complexity of some descriptions considerably. At present, sets must be learnt. Constructs such as logical negation and universally quantified variables do not exist.
- The pattern matching and statement generation procedures can be made more efficient.
- The object generation procedures can also be improved by dealing with negative information in a more intelligent way.
- If the *not* connective is to be added to the language, there ought to be a procedure for learning

concepts with *counterfactuals* (Vere, 1980).

- At present, patterns in a trial description are matched in an 'all-or-nothing' manner with concepts stored in memory. A partial matching procedure similar to those developed by Hayes-Roth and McDermott(1977) and Vere (1975) will enable Marvin to attempt more complex concepts.
- The partial matching algorithm should also enable Marvin to detect 'forall' relationships.
- One limitation that must be imposed on the trainer is that he must present concepts to Marvin in a specific order, simple concepts first, followed by larger concepts which contain the simple ones. This is necessary because Marvin has relatively little control over the structure of its memory. A further application of the partial matching algorithm is to give Marvin the ability to compare concepts its has stored in memory and extract common features. This would allow the program to ensure that memory is always well structured.
- The learning algorithm currently in use is very conservative. If a complex object is shown, every property is the subject of a generalization. A *Focus Gambling* algorithm may be used which selects the most promising properties for generalization, thus reducing the time required to learn a concept.

Some of the proposals listed above are relatively straightforward improvements to the implementation. Others are, in themselves, complete research topics for the future.

8
Conclusion

When research in Artificial Intelligence first began in the 1950's, emphasis was placed on creating programs with general intelligence. That is, they should not be limited to working in a particular domain. However, after a decade of work in the field, opinions changed. Researchers recognized that to perform tasks with an acceptable level of competence, a great deal of knowledge about the environment was required. As a result, a number of very succesful 'expert' problem solvers have been constructed.

The most significant problem encountered by designers of such programs has come to be called 'knowledge engineering'. In order to develop an expert program, the designers must create a large knowledge base, usually requiring the help of human experts. This process is time consuming and often involves *ad hoc* programming methods.

These difficulties have led us back to considering a more general approach to Artificial Intelligence where the generality is moved a level higher than it was before. The special purpose problem solvers remain; however, the knowledge needed to drive them should be acquired by a general purpose learning system.

A number of very useful algorithms have been developed for concept learning. Some of these were discussed in Chapter 2. The project described in this work was intended to add to this 'bag of tools' for the knowledge engineer.

8.1 Summary

When a learning program expects the examples it is shown to be carefully selected by the trainer, it assumes that the trainer already knows the definition of the concept to be learnt and that he knows something about how the program works. Marvin is capable of generating its own training instances, so a lot of the hard work involved in learning is shifted away from the trainer, to the program (Mitchell and Utgoff, 1980; Lenat, 1977; Sussman, 1975).

Marvin uses a 'generate and test' model of learning. Given an initial example, the program creates a concept intended to describe the class of objects containing this example. It tests its hypothesis by performing an experiment. That is, it creates its own instance of the concept that has been developed. If the example shown to the trainer is an instance of the target concept, Marvin may continue to generalize its hypothesis. Otherwise it must modify the hypothesis so that a correct instance can be created.

Concepts are described in terms of a description language based on first order predicate logic with quantifiers. An important ability which Marvin has is that, like Cohen's CONFUCIUS (Cohen, 1978), the description language allows the program to describe complex concepts in terms of simpler ones that have been learnt before.

The description of an event shown to Marvin is converted to an expression in first order logic. The program then performs a pattern matching operation to find associations between the input event and the knowledge it has stored in its memory. The purpose of this operation is to find the concepts that are already known to Marvin which recognize parts of the event. A concept is true if it is implied by a subset of the trial description.

A trial is generalized by replacing the implicants of a concept stored in memory by a statement refering to that concept. New trials will continue to be generalized until one creates an instance which is not recognized by the target concept. When this occurs, an attempt is made to make a trial which is more specific than the one which failed. This is done by adding statements to the trial description without removing any other statements. In this way a sequence of trials is produced which 'oscillates' around the target concept, getting closer for each new trial until the target is finally reached.

In order to be able to create instances to show the trainer, Marvin treats a concept description as a program in a logic programming language such as Prolog. During the execution of such a program, any unbound variable becomes bound to a value which will result in the entire concept being true. The language is non-deterministic since there may be more than one possible set of bindings.

Not all the possible outputs of a concept are acceptable as training instances. If a trial concept is inconsistent, that is, it recognizes events not recognized by the target, then the object construction routine must generate one of those events not in the target. Thus, Marvin must have an 'instance selector' which is capable of choosing the best objects to show the trainer. In Chapter 3 we saw that, as long as memory remains well structured, if an event does not satisfy any statement which has been removed from the trial (and the removed statement is not implied by any in the trial) then the event is an acceptable training instance.

In Chapters 4 and 6 we saw that Marvin can be taught a wide variety of complex concepts. The trainer does not require any detailed knowledge about how the program works and Marvin's response is usually quite fast. Thus it seems likely that a system such as this one will prove useful in creating knowledge bases for intelligent problem solvers.

8.2 Discussion

An interesting aspect of Marvin's design is that it brings induction and theorem proving together in one program. Not only are there two components in Marvin for performing these functions, but also the procedures have a great deal in common in their implementations. They both rely very heavily on a unification algorithm for pattern matching.

Figure 8.1 contains a schematic representation of a learning system based on Marvin.

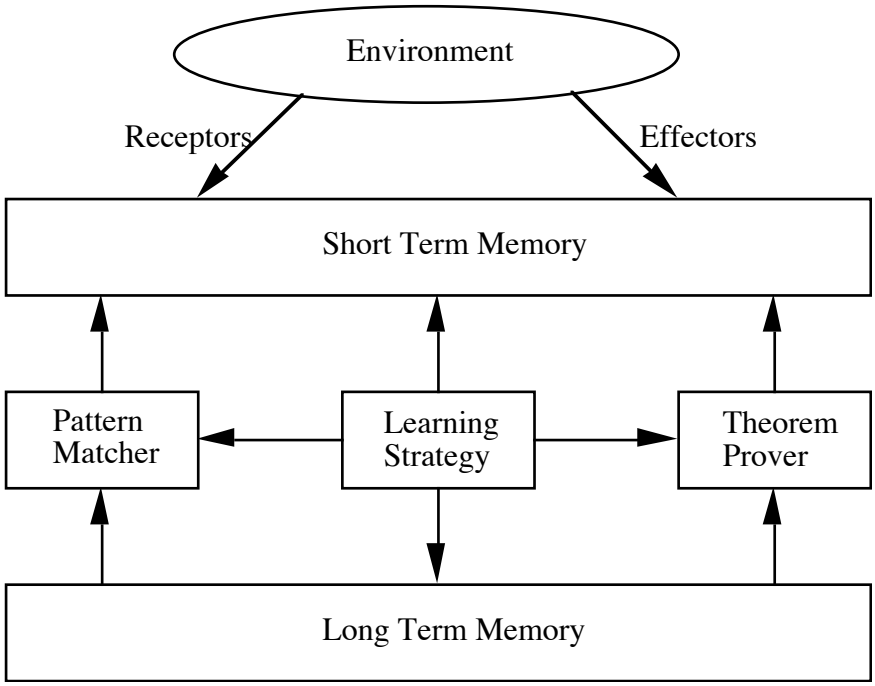


Figure 8.1. A general purpose concept learning system.

The system's long term memory consists of the collection of concepts that it has learnt to date. When some new object is seen, its representation is entered into the short term memory where the pattern recognition device is able to access it and compare this description with the contents of the long term memory.

As long as the system continues to observe the world, it tries to maintain a consistent model which explains the relationships among everything it sees. The model is represented by the concepts in long term memory. When something new and unexplained is encountered, the world model must be modified to take into account the new phenomenon. Updating long term memory is the responsibilty of

the learning mechanism. It uses what it already knows about the world to create a theory which describes the new event. It must also propose an experiment for testing the theory. To do this the learning strategy invokes a theorem prover which will attempt to establish the validity of the theory. The outcome of the experiment must be observed to discover if the theory was correct. If it was not, then a new theory must be advanced and tested.

The advantage of incorporating a theorem prover (or some kind of problem solver) into a learning system is that it can learn by *doing*. The program is not merely a passive system analyzing data as it is input. A system like the one proposed in this work can *actively* search for a better model by performing some actions in the world it it is investigating.

There is one particularly important part of this design which requires further attention. The system should be capable of some 'introspection'. It should be able to examine its memory to try to discover new concepts from the knowledge it already has (Lenat, 1977). The system should also be able to evaluate its learning strategy so that it can be adjusted according to the circumstances (Mitchell and Utgoff, 1980). Undoubtedly these problems will keep us busy for some time to come.

References

Banerji, R. B. (1964). A Language for the Description of Concepts, *General Systems,* **9**.

Banerji, R.B. (1969). *Theory of Problem Solving - An Approach to Artificial Intelligence,* American Elsevier, New York.

Banerji, R.B. (1976). A data structure which can learn simple programs from examples of input-output, in *Pattern Recognition and Artificial Intelligence*, ed. Chen, Academic Press.

Banerji, R. B. (1977). *Learning with Structural Description Languages,* for NSF grant MCS-76-0-200, Temple University.

Banerji, R.B. (1978). Using a Descriptive Language as a Programming Language, in *Fourth International Joint Conference on Pattern Recognition*, pp. 346-350.

Banerji, R.B. (1980). *Artificial Intelligence: A Theoretical Approach,* North Holland, New York.

Banerji, R. B. and Mitchell, T. M. (1980). Description Languages and Learning Algorithms: A Paradigm for Comparison, *Policy Analysis and Information Systems,* **4** (2).

Bruner, J. S., Goodnow, J. J. and Austin, G. A. (1956). *A Study of Thinking,* Wiley, New York.

Buchanan, B. G. and Feigenbaum, E. A. (1978). DENDRAL and META-DENDRAL: Their Applications Dimension, *Artificial Intelligence,* **11**, pp. 5-24.

Buchanan, B. G. and Mitchell, T. M. (1977). *Model-Directed Learning of Production Rules,* STAN-CS-77-597, Department of Computer Science, Stanford University.

Cohen, B. L. (1977). A Powerful and Efficient Pattern Recognition System, *Artificial Intelligence,* **9**, pp. 223-256.

Cohen, B. L. (1978). *A Theory of Structural Concept Formation and Pattern Recognition,* Ph.D. Thesis, Dept. of Computer Science, University of N.S.W.

Cohen, B. L. and Sammut, C. A. (1978). CONFUCIUS: A Structural Concept Learning System, *Australian Computer Journal,* **10** (4), pp. 138-144.

Cohen, B. L. and Sammut, C. A. (1978). CONFUCIUS - A Structural Approach to Object Recognition and Concept Learning, in *Australian Universities Computer Science Seminar*, pp. 249-259.

Cohen, B. L. and Sammut, C. A. (1978). Pattern Recognition and Learning with Structural Description Languages, in *Fourth International Joint Conference on Pattern Recognition*, pp. 1443-1446.

Cohen, B. L. and Sammut, C. A. (1979). Object Recognition and Concept Learning with CONFUCIUS, *Pattern Recognition Journal*.

Cohen, B.L. and Sammut, C.A. (1980). Learning Concepts in Complex Environments, *Australian Computer Science Communications,* **2** (1), pp. 13-23.

Dietterich, T. (1978). *INDUCE 1.1 - The program description and a user's guide,* (internal), Department of Computer Science, University of Illinois, Urbana.

Dietterich, T. G. and Michalski, R. S. (1977). Learning and Generalization of Characteristic Descriptions: Evaluation Criteria and Comparative Review of Selected Methods, in *Fifth International Joint Conference on Artificial Intelligence,* pp. 223-231.

Eastwood, E. (1968). Control Theory and the Engineer, *Proceedings of the Institution of Electrical Engineers,* **115** (1), pp. 203-211.

Fikes, R. E., Hart, P. E. and Nilsson, N. J. (1972). Learning and Executing Generalized Robot Plans, *Artificial Intelligence,* **3**.

Hayes-Roth, F. (1973). A Structural Approach to Pattern Learning and the Acquisition, in *First International Joint Conference on Pattern Recognition*, pp. 343-355.

Hayes-Roth, F. and McDermott, J. (1977). Knowledge Acquisition from Structural Descriptions, in *Fifth International Joint Conference on Artificial Intelligence,* pp. 356-362.

Hayes-Roth, F. and McDermott, J. (1978). An Interference Matching Technique for Inducing Abstractions, *Communications of the ACM,* **21**, pp. 401-411.

Holland, J.H. (1975). *Adaptation in Natural and Artificial Systems,* University of Michigan Press, Ann Arbor.

Horn, A. (1951). On Sentences which are True of Direct Unions of Algebras, *Journal of Symbolic Logic,* **16**, pp. 14-21.

Jensen, K. and Wirth, N. (1974). *PASCAL, User Manual and Report,* Springer Verlag.

Larson, J. (1977). *Inductive Inference in the Variable Valued Logic System VL21: Methodology and Computer Implementation,* Technical Report 869, Department of Computer Science, University of Illinois, Urbana-Champaign.

Lenat, D. B. (1977). Automated Theory Formation in Mathematics, in *Fifth International Joint Conference on Artificial Intelligenc*e, pp. 833-842.

Michalski, R. S. (1973). Discovering Classification Rules Using Variable Valued Logic System VL1, in *Third International Joint Conference on Artificial Intelligence,* pp. 162-172.

Michalski, R. S. (1980). Pattern Recognition as Rule-Guided Inference, *IEEE Transactions on Pattern Analysis and Machine Intelligence,* **2** (4), pp. 349-361.

Mitchell, T. M. (1978). *Version Spaces: An Approach To Concept Learning,* STAN-CS-78-711, Ph.D Thesis, Department of Computer Science, Stanford University.

Mitchell, T. M., Utgoff, P. E. and Banerji, R. B. (1980). Learning Problem Solving Heuristics by Experimentation, in *Proceedings of the Workshop on Machine Learning,* Carnegie-Mellon University.

Pennypacker, J. C. (1963). *An Elementary Information Processor for Object Recognition,* SRC 30-I-63-1, Case Institute of Technology.

Plotkin, G. D. (1970). A Note on Inductive Generalization, in *Machine Intelligence 5*, pp. 153-163, ed. B. Meltzer and D. Michie, Edinburgh University Press.

Plotkin, G. D. (1971). A further note on inductive generalization, in *Machine Intelligence 6,* ed. B. Meltzer and D. Michie, Elsevier.

Popplestone, R. J. (1970). An Experiment in Automatic Induction, in *Machine Intelligence 5*, ed. B. Meltzer and D. Michie, Edinburgh University Press.

Reynolds, J. C. (1970). Transformational Systems and the Algebraic Structure of Atomic Formulas, in *Machine Intelligence 5,* pp. 153-163, ed. Meltzer and Michie.

Robinson, J. A. (1965). A Machine Oriented Logic Based on the Resolution Principle, *Journal of the ACM,* **12** (1), pp. 23-41.

Rothenberg, D. (1972). Predicate Calculus Feature Generation, in *Formal Aspects of Cognitive Behaviour,* pp. 72-126, ed. T. Storer and D. Winter, Springer-Verlag Lecture Notes in Computer Science.

Roussel, P. (1975). *Prolog: Manual de reference et d'utilization,* Internal Report, Groupe d'Intelligence Artificielle, Marseille-Luminy.

Rulifson, J. F., Derksen, J. A. and Waldinger, R. L. (1972). *QA4: A Procedural Calculus for Intuitive Reasoning,* Technical Note 73, SRI Artificial Intelligence Center.

Sammut, C. (1981). Concept Learning by Experiment, in *Seventh International Joint Conference on Artificial Intelligence*, pp. 104-105.

Sammut, C.A. and Cohen, B.L. (1980). A Language for Describing Concepts as Programs, in *Language Design and Programming Methodology,* ed. J. M. Tobias, Springer-Verlag Lecture Notes in Computer Science, Vol 79.

Shapiro, Ehud Y. (1981). *Inductive Inference of Theories From Facts,* 192, Yale University.

Smith, R. G., Mitchell, T. M., Chestek, R. A. and Buchanan, B. G. (1977). A Model for Learning Systems, in *Fifth International Joint Conference on Artificial Intelligence,* pp. 338-343.

Sussman, G.J. (1975). *A Computer Model of Skill Acquisition,* American Elsevier, New York.

Vere, S. (1975). Induction of Concepts in the Predicate Calculus, in *Fourth International Joint Conference on Artificial Intelligence*, pp. 351-356.

Vere, S. A. (1977). Induction of Relational Productions in the Presence of Background Information, in *Fifth International Joint Conference on Aritficial Intelligence*.

Vere, S. A. (1978). Inductive Learning of Relational Productions, in *Pattern-Directed Inference Systems,* pp. 281-295, ed. D. A. Waterman and F. Hayes-Roth, Academic Press.

Vere, S. A. (1980). Multilevel Counterfactuals for Generalizations of Relational Concepts and Productions, *Artificial Intelligence,* **14** (2), pp. 139-164.

Vere, S. A. (1980). *Learning Disjunctive Concepts,* Personal Communication, Jet Propulsion Laboratory.

Vere, S. A. (1981). *Constrained N-to-1 Generalizations,* Internal Report, Jet Propulsion Laboratory, Pasadena.

Warren, D. H. D. (1974). *WARPLAN: A System for Generating Plans,* Department of Computational Logic Memo No. 76, University of Edinburgh.

Winograd, T. (1972). *Understanding Natural Language,* Edinburgh University Press, Edinburgh.

Winston, P. H. (1970). *Learning Structural Descriptions From Examples,* Ph.D Thesis, MIT Artificial Intelligence Laboratory.