Boyd, Richard Victor

PLAN GENERATION AND PROLOG

The University of Arizona M.S. 1986

PLAN GENERATION AND PROLOG

by

Richard Victor Boyd

A Thesis Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

MASTER OF SCIENCE
WITH A MAJOR IN ELECTRICAL ENGINEERING

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 8 6

STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgement of source is made. Requests for permission for extended quotation from or reporduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: Richard V. Boyd

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

B. P. ZEIGLER
Professor of Electrical
and Computer Engineering

12/12/86
Date

PREFACE

The purpose of this thesis is to give a an introduction to the concepts of plan generation, a thorough explanation of how these concepts are employed in WARPLAN, an overview of Declarative Languages, and an encouragement to seek a blend of the best features of both LISP and Prolog rather than set the two at odds.

TABLE OF CONTENTS

TABLE OF CONTENTS--Continued

TABLE OF CONTENTS--Continued

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS--Continued

LIST OF ILLUSTRATIONS--Continued

ABSTRACT

WARPLAN is a plan generating program in Prolog developed by D. H. D. Warren of the University of Edinburgh School of Artificial Intelligence. Because Prolog is a highly recursive language employing depth first search with backtracking, and because the versions available of WARPLAN do not employ very descriptive functors and arguments, the flow of logic of this program is less than apparent. An in-depth explanation of how the program works and why is accompanied by flow charts and descriptive variable naming.

An overview of Declarative Languages is presented with an emphasis on the development of new languages seeking the best features of functional (LISP) and logical (Prolog) approaches, rather than setting the two at odds.

CHAPTER 1

INTRODUCTION

A major difference between what is normally considered as programming and what is termed planning is that a program is designed to be executed many times with different sets of data, whereas a plan is typically executed just once. Plans are designed to work in the real world where the unexpected is the norm, so plans must also have the ability to replan.

As the planning process begins, there is no plan, just an initial state, a goal state, and a set of actions for transforming one state into another. It is the plan that must be 'thought up'. The overall task must be broken down into sufficiently small enough elements so that they can be operated on by a predetermined set of rules, subplans then generated, and the pieces recombined into the final plan. If this generated plan is to be used in the real world to direct the actions of a robot in a factory environment, then the plan has to be sufficiently 'good' that the robot does not run 'amok', but it is not required that the plan be optimal. Humans perform quite capably using less than optimal plans, from playing games of chess to tuning up their cars.

The program that is the subject of this thesis was written in Prolog, which employs a first-order predicate calculus syntax written in Horn clauses. The representation of knowledge in this fashion makes it seemingly easy to read at first glance, for instance,

father(john, mary)

is easily discernible to mean,

john is the father of mary

however, one quickly becomes aware that in a highly recursive program which employs a depth first search strategy with backtracking, the fundamental logic of the program flow can become analogous to a ball of tangled twine.

Why ... How?

Why choose this topic and how did it get started? Two avenues merged into one, the first being a PhD dissertation on the topic of planning (Zhu 1985) done by a former student of Dr. Bernard Zeigler, the thesis director for this work. The objective of this dissertation was to combine temporal logic with planning to develop a more efficient approach to the subject, but there was no actual implementation of the concepts. Some type of implementation was to be the objective of this thesis.

The second avenue was a former engineer at Hughes Aircraft in Tucson, Arizona, who, previous to the beginning

of this thesis, provided the author with a copy of a Prolog program which supposedly generated plans. At the time, this seemed like a good starting point from which to implement part of Zhu's dissertation, although later this approach seemed debatable. The code provided can be seen in Appendix B and it presented three major obstacles. First, the documentation was almost non-existent, secondly, it later proved to contain errors, and thirdly, it was written in micro-Prolog. This last problem was a result of the fact that in micro-Prolog all variables are named X, Y, Z, X1, Y1, Z1, X2, Y2, etc. This, in combination with the fact that variable names in one rule do not necessarily correspond to the same variable names in any other rule, since they are only place-holders, made deciphering the code somewhat akin to the work of a cryptographer.

At the time this code was given to the author, the engineer, who later left Hughes, said he got it from a former professor at the University of Kentucky who, in turn, got it from somewhere in Europe, possibly Portugal. The trail was getting cold.

WARPLAN

The code was eventually deciphered and the results are presented in chapter 3 of this thesis.

During the course of this recoding, a bibliographic search was made of relevant subject matter. Among the

references in Zhu's dissertation was something called WARPLAN by D. H. D. Warren from the University of Edinburgh School of Artificial Intelligence. The author did not initially make the correct connection of the name WARPLAN with WARren PLAN, instead assuming it to be a battle field PLANning strategy for WARfare! Only after the recoding was complete was it discovered that this D. H. D. Warren was THE D. H. D. Warren, one of the principals in the implementation of the Prolog interpreter and compiler on the DEC-10 (Warren and Pereira 1977, p. 109), among many other accomplishments.

As can be seen in Appendix A, a library search during the Spring 1986 semester was unable to locate a copy of WARPLAN from either a US or Canadian library. At the end of May 1986 the library estimated that it would take three to six months to obtain a copy from the University of Edinburgh. As the intention was to finish this thesis during the summer of 1986, this time span seemed to long to be of value and, furthermore, the author still did not realize that this was the basis for what was originally obtained via Hughes.

During Fall semester, a new professor at the University of Arizona, Dr. Jerzy Rozenblit, produced a copy of WARPLAN as found in a reference manual for a version of Prolog running on IBM PC's (Prolog-86 1984). This code is reproduced in Appendix C and appears to be the original as developed by Warren. It is a considerable improvement over

the version the author originally obtained, but still it is difficult to understand the flow of logic from it.

Declarative Languages

More significant than the information on plan generation in Prolog presented here, is the work currently being done by many researchers to develop new languages which combine the best features of both the functional and logical approaches. Chapter 4 contains a survey of some of this research and suggests references for further reading.

CHAPTER 2

PLAN FORMATION

Problem solving is basically a search through a state space from an initial state to a goal state by means of a sequence of allowable operations. To solve a problem, one must be able to define the problem precisely, specify the problem space and operators for moving through that space, list the initial and goal states, determine if the problem is decomposable, clarify if solution steps can be ignored or undone if they violate certain conditions, choose an appropriate knowledge representation scheme, and select a problem solving technique (Rich 1983, p. 106).

When an unmanned, robotic vehicle one day lands on Mars and begins to explore that planet's surface, the vehicle must not only be able to plan its route but also be able to replan as it receives feedback concerning unexpected conditions. The benefits of planning can be summarized as reducing search, resolving goal conflicts, and providing a basis for error recovery (Cohen and Feigenbarum 1982, p. 516).

Two of the major problems in planning are the problem of limiting search and that of interacting

subproblems, and these are related because additional search can result from too early commitment to a set of interacting subgoals.

Decomposition

For simple problems, traversing a complete state description might be possible, but for more complex, real world problems such an approach is not feasible. The concept of problem decomposition is so important because it becomes necessary in complex situations to be able to work on smaller pieces of the main problem, solve them individually, and then recombine the subsolutions into the complete solution. Ideally, this decomposition would result in independent subproblems which would be easier to solve than the original. What normally happens, though, is that these subproblems are dependent and the ability to handle the resulting interactions is a key ingredient of the planning process.

A grocery list would normally be an example of an unordered, independent list of goals, although it would not be wise on a hot summer day to put the ice cream in the shopping basket first! Usually, though, a plan has an implicit ordering of its goals as would be the case in landing an airplane or scuba diving. Building a house without first laying the foundation will be not passed by

the county inspector. Thus the final plan becomes a linear ordering of problem-solving operators, but the goals achieved by these operators often have a hierarchial structure which must be taken into consideration in the planning process (Cohen and Feigenbaum 1982, p. 515).

Rule 4 in the Analysis and Implementation chapter handles subgoal interactions by regressing back through the current plan until a point is found at which the new action and its preconditions do not cause conflicts with what has already been accomplished up to that point. A somewhat similar approach seems to have been taken in RSTRIPS (Nilsson 1980, pp. 321-33) wherein a goal regression mechanism is used for circumventing goal interaction problems. When such a problem does occur, RSTRIPS also rearranges the plan by regressing preconditions back through the current plan to a position in which the problem interaction is removed.

Backtracking

In simple problems, if a particlular solution path leads to a deadend, a new one can be explored by backtracking to the last choice point and then proceeding in a different direction. Yet many situations in the real world need the correct solution the first time and can not allow backtracking to try a new approach as, for example, in a moon landing. A computer simulation, however, can provide

the correct solution path via backtracking and only then would the correct one be carried out.

This is what planning is all about, deciding on a course of action before acting. A plan is, then, a representation of a course of action (Cohen and Feigenbaum 1982, p. 515).

Frame Problem

The frame problem, which has nothing to do with the frame as a representation formalism, and which is a problem common to all representation formalisms, concerns the inability to determine which things change and which do not as one moves from one state to another. For other than very simple systems, it becomes infeasible to continually recopy each new state produced by application of an operator. Because almost all statements in a given state continue to hold true in the next state, instead of explicitly transforming each state into the next, one usually seeks to describe only the changes, and assume the rest remains constant. This assumption was made in STRIPS (STanford Research Institute Problem Solver) (Fikes and Nilsson 1971, p. 189).

The seeming inadequacies of logic to handle the frame problem also led to the development of specialized techniques within STRIPS to deal with it (Kowalski 1979, p. 133). All the clauses for world models were stored in a

common memory structure and associated with each clause was a 'visibility flag'. All clauses initially were visible and those that did not change from one state to another remained visible. The system would then only consider clauses that were so marked. To specify a given world model, STRIPS would mark as visible the clauses in the 'addition list' and mark invisible those in the 'deletion list' (Fikes and Nilsson 1971, p. 200). STRIPS was later simplified (Nilsson 1980, p. 316).

Expressing the Frame Axiom

Cordell Green made one of the first attempts to solve robot problems by converting all assertions to clause form so that a resolution theorem proving system could solve them. To keep track of which facts were true in which state, Green included a state variable among the arguments of each predicate. This approach could not adequately handle the frame problem. The plan generation program that is the subject of this thesis offers another approach to this problem by using terms to name statements and by using the frame axiom top-down rather than bottom-up.

An action executed on one state will result in a new state and the functional expression,

```
determine_next_state(Action,
                     Present_state,
                     New_state)
```

is used to denote the mapping of one state onto another (Nilsson 1980, p. 309). The major effect of an action can be formulated as the following implication,

```
        clear(Block, Present_state) &
        clear(To, Present_state) &
        on(Block, From, Present_state) &
        notequal(Block, To)
  ===>  determine_next_state(move(Block, From, To),
                             Present_state,
                             Next_state) &
        clear(Block, Next_state) &
        clear(From, Next_state) &
        on(Block, To, Next_state).
```

This states that if Block and To are clear and if Block is on From in the Present_state and Block does not equal To, then Block and From will be clear and Block will be on To in the Next_state resulting from the action move(Block, From, To).

A problem arises because the above formula does not specify the state of relations <u>not</u> affected by the action, and these must be specified in Green's formulation for each assertion. For example, to express that blocks not moved stay in the same position would require,

```
        on(Block, Location, Present_state) &
        notequal(Block, Object)
```

```
===>  determine_next_state(move(Object, From, To),
                            Present_state,
                            Next_state) &
      on(Block, Location, Next_state).
```

Assertions would also be needed to state that blocks remain clear and so on.

For each action, Green's formulation would require a separate frame assertion for each predicate. For anything but a trivial system, it is easy to see why it was thought logic was inadequate to handle the frame problem.

As previously mentioned, the program in this thesis offers another approach, an approach first suggested by Robert Kowalski in which he employs a binary representation of state space problems and simplifies the statement of the frame assertions first of all by transforming what would ordinarily be predicates in Green's formulation into terms (Nilsson 1980, p. 311). Figure 2.1 shows a comparison of initial conditions as seen by Green and by Kowalski, where 'possible' implies a possible state that can be reached. Note that the literal,

```
on(a, floor, state0)
```

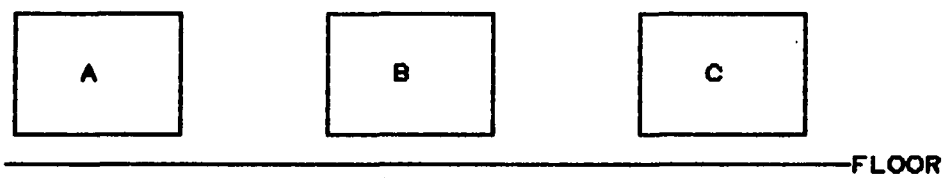
is replaced by the literal,

```
exists_in(on(a, floor), state0)
```

in which the term,

```
on(a, floor)
```

is used to denote the concept of A being on the Floor. This



GREEN

```
on(a, floor, state0).
on(b, floor, state0).
on(c, floor, state0).
clear(a, state0).
clear(b, state0).
clear(c, state0).
```

KOWALSKI

```
possible(state0).
exists_in(on(a, floor), state0).
exists_in(on(b, floor), state0).
exists_in(on(c, floor), state0).
exists_in(clear(a), state0).
exists_in(clear(b), state0).
exists_in(clear(c), state0).
```

Fig. 2.1. Two Logic Approaches

representation of concepts as individuals by Kowalski is a way of gaining some of the benefits of a higher order logic in a first order formulation and can be regarded as a formalization of part of the meta-language. In should be noted that in an object language one uses sentences, whereas in a meta-language one talks about sentences.

The effects of actions is reflected in the 'add-list' by using a separate literal for each relation made true by the action. In the case of the action,

```
move(Block, From, To)
```

we have,

```
will_add(move(Block, From, To), clear(From)).
will_add(move(Block, From, To), on(Block, To)).
```

The major advantage of Kowalski's formulation is that only one frame assertion is needed for each action (Nilsson 1980, p. 314). For the above example, this single frame assertion is,

```
determine_next_state(move(Block, From, To)
                          Present_state,
                          Next_state) &
holds_true_in(Statement, Present_state) &
notequal(Statement, on(Block, From)) &
notequal(Statement, clear(To))
  ===>  holds_true_in(Statement, Next_state).
```

This expression states that all terms different than,

```
on(Block, From)
```

and,

clear(To)

continue to hold true in the next state produced by performing the action,

move(Block, From, To).

This one statement would cover all cases such as blocks not moved stay in the same position, blocks remain clear, etc. that need separate assertions in Green's formulation. This constitues the first part of Kowalski's solution to the frame problem in which he avoids employing a separate frame axiom for every relation by using a single frame axiom.

In this formulation, both states and statements are regarded as individuals and are represented by means of terms (Kowalski 1979, p. 134). A flowchart of a typical action is seen in Figure 2.2.

Top-Down versus Bottom-Up

The logic of the blocks world problem remains separate from its use. Clauses can be used top-down, bottom-up, or a combination of both (Kowalski 1979, p. 137). Bottom-up reasoning is to reason forward form the initial state, whereas top-down reasons backward from the goal.

If the frame axiom is used bottom-up, then preserved facts must be copied from state to state. It is at this point that STRIPS abandoned the frame axiom and used special purpose procedures instead (Kowalski 1979, p. 138). An

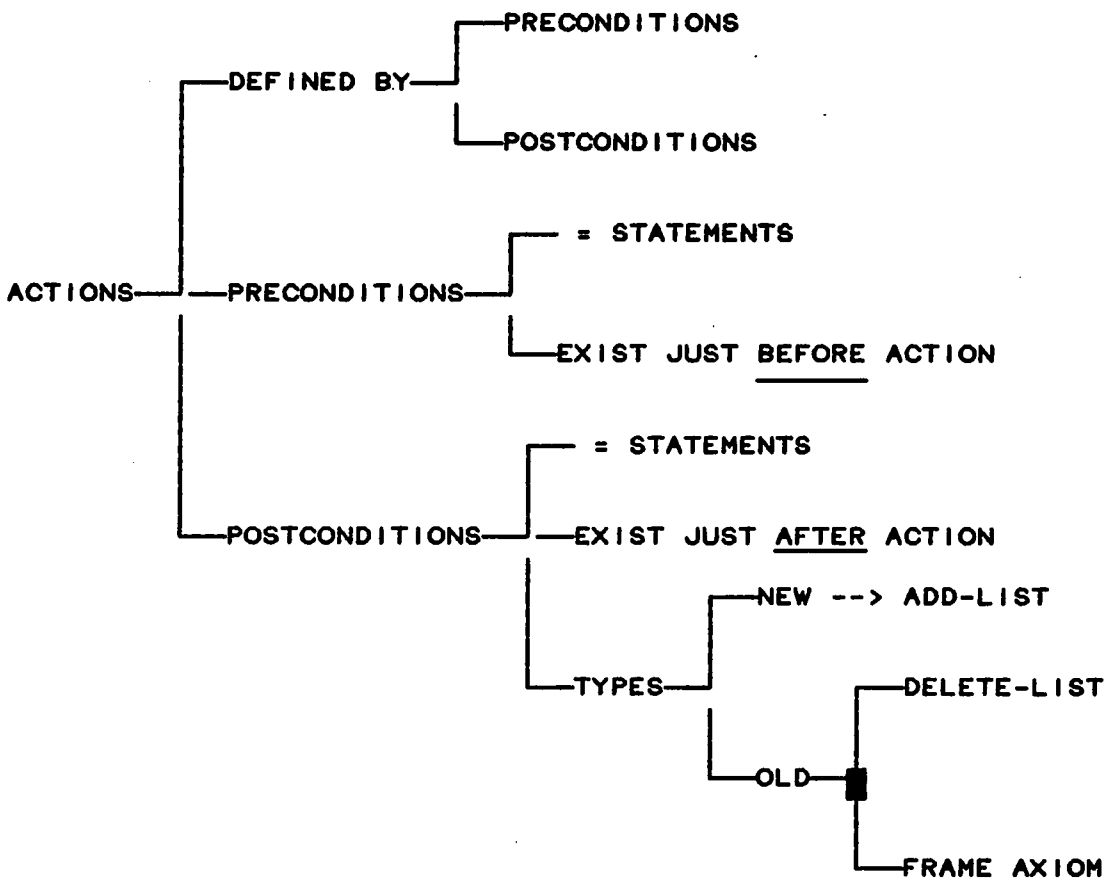


Fig. 2.2. Actions

alternate approach, although, would be to interpret the frame axiom top-down.

1. show that a statement is added by an action.

or 2. if the statement is not deleted by the action, then determine if the statement holds in the previous state.

This is the approach taken in Rule 7 of the Analysis and Implementation chapter where Rule 7A checks if a statement/goal is added by the last-action and, if not, then 7B determines if it can-be-achieved-by the previous state(s). Rule 7C eventually checks if was true in the initial state, as an exiting condition.

Changing the direction of execution of the frame axiom improves the algorithm by improving its control without changing its logic.

Bottom-Up Execution of Preconditions. Figure 2.3 illustrates the search space determined by bottom-up execution of the following preconditions as found in the database.

```
precond_of(on(Block, From) &
                notequal(From, floor) &
                clear(Block),
                move(Block, From, floor)).
```

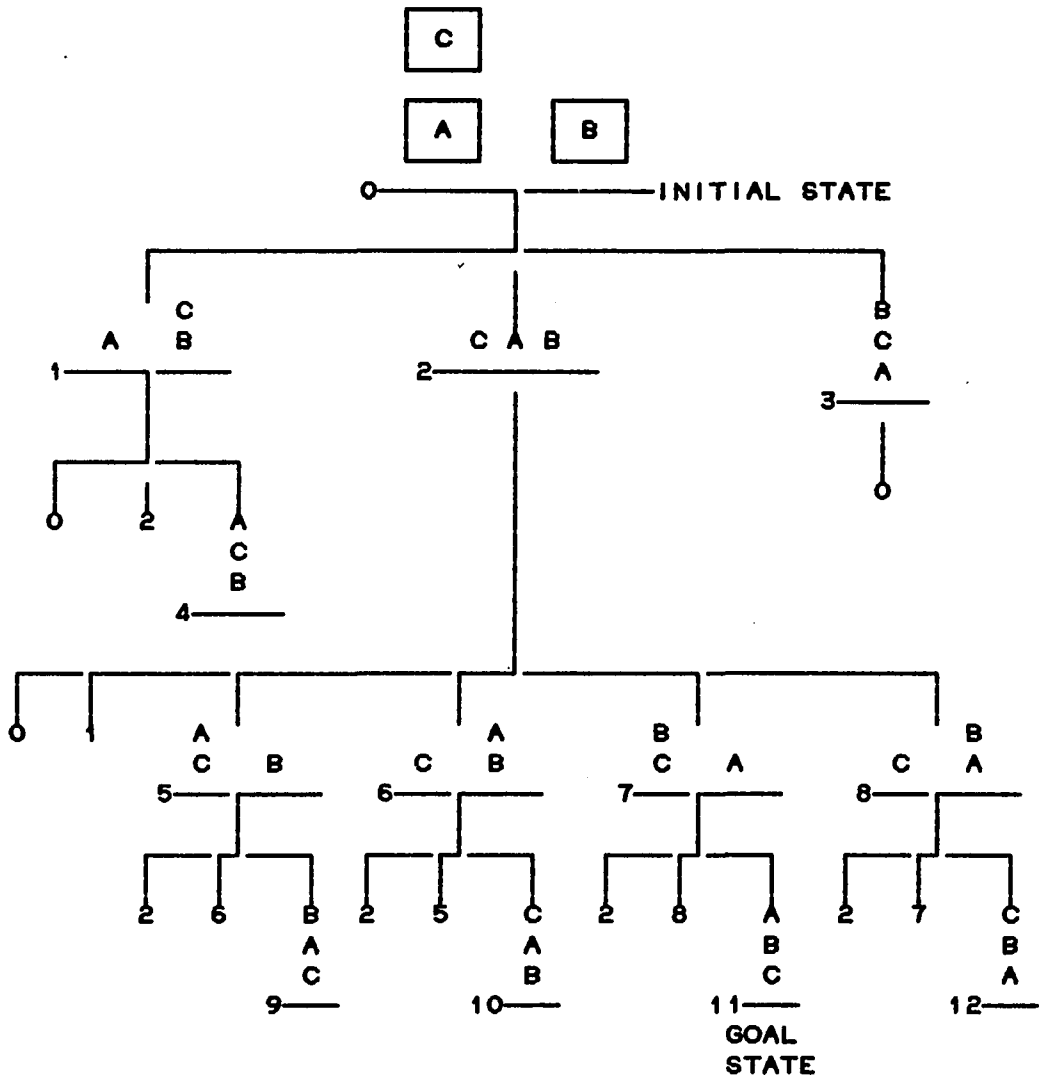


Fig. 2.3. Bottom-Up Execution of Preconditions

```
precond_of(clear(To) &
           on(Block, From) &
           notequal(Block, To) &
           clear(Block),
           move(Block, From, To)).
```

This bottom-up execution of the state space is independent of the direction of execution of the frame axiom.

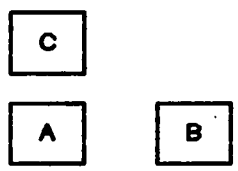
Bottom-Up Execution of Frame Axiom. Figures 2.4 and 2.5 display assertions belonging to the solution path as generated by bottom-up execution of the frame axiom.

Top-Down Execution of Preconditions. Figure 2.6 illustrates the search space determined by executing the preconditions top-down.

Top-Down Execution of Frame Axiom. In Figures 2.7 and 2.8 all clauses are executed top-down. Duplicate subgoals are deleted.

The advantage of executing the frame axiom top-down is that there are three situations that might determine if a particular statement can be achieved in a given state,

1. show that a statement is added by the last action.
2. if the statement is not deleted by the last action, then determine if the statement holds in the previous state.
3. determine if it was true in the initial state.



STATE0 0——————INITIAL STATE

on(a, floor) clear(b)
on(b, floor) clear(c)
on(c, a)

STATE2 C A B
2————

ASSERTIONS CONTINUED FROM PREVIOUS STATE
on(a, floor) clear(b)
on(b, floor) clear(c)

ADDED ASSERTIONS
on(c, floor) clear(a)

DELETED ASSERTIONS
on(c, a)

Fig. 2.4. Bottom-Up Execution of Frame Axiom---Part 1

STATE7

CONTINUED
on(a, floor)
on(c, floor)

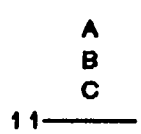
clear(a)
clear(b)

ADDED
on(b, c)

DELETED
on(b, floor)

clear(c)

STATE11



CONTINUED
on(c, floor)
on(b, c)

clear(a)

ADDED
on(a, b)

DELETED
on(a, floor)

clear(b)

Fig. 2.5. Bottom-Up Execution of Frame Axiom---Part 2

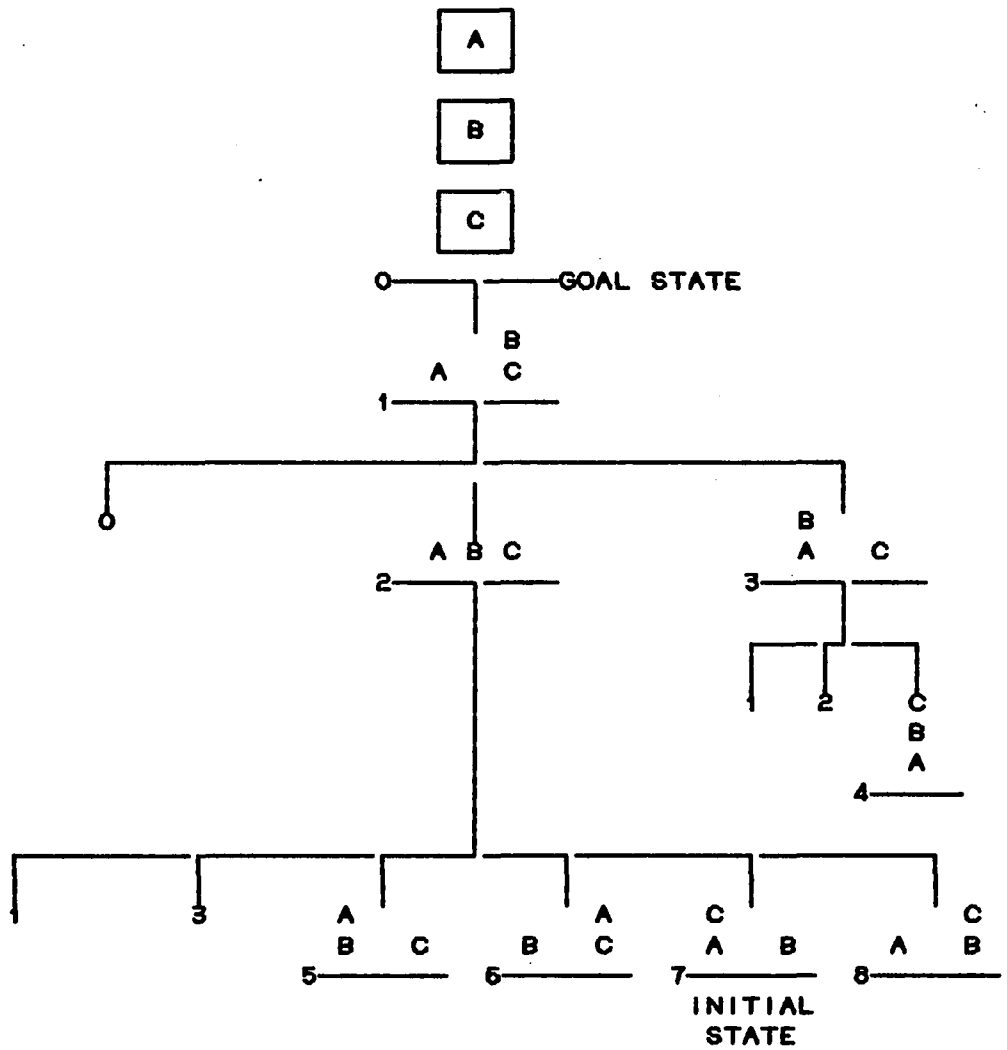


Fig. 2.6. Top-Down Execution of Preconditions

STATE0 0——————GOAL STATE

on(a, b)
on(b, c)
on(c, floor)

ACTION move(a, floor, b)

STATE1

B
A C
1——————

DELETED (by reversing 'will_add')
on(a, b)

CONTINUED
on(b, c)
on(c, floor)

PRECONDITIONS
clear(b)
on(a, floor)
notequal(a, b) <--- Satisfied
clear(a)

ACTION move(b, floor, c)

STATE2

A B C
2——————

DELETED (by reversing 'will_add')
on(b, c)

Fig. 2.7. Top-Down Execution of Frame Axiom---Part 1

```
        CONTINUED
        clear(a)
        clear(b)
        on(a, floor)
        on(c, floor)


        PRECONDITIONS
        clear(c)
        on(b, floor)
        notequal(b, c)   <--- Satisfied
        clear(b)         <--- Duplicate

ACTION              move(c, a, floor)

STATE7                    C
                          A B
                        7_____

        DELETED    (by reversing 'will_add')
        on(c, floor)
        clear(a)


        CONTINUED
        clear(b)         <--- Satisfied by IC's
        clear(c)         <--- Satisfied by IC's
        on(a, floor)     <--- Satisfied by IC's
        on(b, floor)     <--- Satisfied by IC's


        PRECONDITIONS
        on(c, a)         <--- Satisfied by IC's
        notequal(c, floor)  <--- Satisfied
        clear(c)         <--- Duplicate


INITIAL CONDITIONS
        on(a, floor)
        on(b, floor)
        on(c, a)
        clear(b)
        clear(c)
```

Fig. 2.8. Top-Down Execution of Frame Axiom---Part 2

The advantage is that this information is only determined when needed, rather than all of it continually being carried along, much of it as extra baggage.

Combination Execution. Although the program in this thesis uses top-down execution of the frame axiom, it uses a combination of top-down and bottom-up execution of the state space similar to STRIPS. Both the back chaining and backtracking built into Prolog are employed at times along with forward chaining using information from the initial state. Sometimes one approach works better, sometimes the other. Most planning systems work primarily in a goal directed mode for such reasons as the ease of recording dependency information and a smaller branching factor (Rich 1983, p. 250).

The approach of WARPLAN is to explicitly store information about the initial state and use the frame axiom to compute information about later states. The alternative, when using depth first search chaining forward from the initial state, is to explicitly store information about the current state and to compute information about earlier states.

It should also be noted that WARPLAN suffers from possible redundancies for cases in which permutations of independent subgoals would produce the same end result.

CHAPTER 3

ANALYSIS AND IMPLEMENTATION

The major portion of this thesis was spent trying to decipher the program in Appendix B. Because the code in Appendix B lacked practically all documentation and comments, and because of the ambiguity of variable naming in micro-PROLOG, this task often seemed as much appropriate for cryptography as for computer science. This chapter contains the results of that analysis.

For efficiency, rules that fire least should be put towards the end of the program to avoid unnecessary search. However, the following rule order has been changed from that of Appendix B in order to clarify the flow of thought between a rule and those it calls. Ordering of arguments in clauses has also been changed where deemed necessary, and these changes are listed in Appendix D. A very important part of this recoding was to substitute variable and predicate names that were as much self explanatory as possible, because this was not only essential in decoding and recoding the program, but it will also be an aid for others seeking to understand the flow of logic.

Rule 1

The problem is specified by the user in Rule 1, Figures 3.1 and 3.2, which directly or indirectly calls all remaining rules. The purpose of Rule 1A is to check that the Goal_state is not inconsistent with the 'impossible' states as declared in the database section of the program. For instance,

```
Goal_state = on(block_A, block_A).
impossible(on(X, X)).
```

would be an example of two contradictory conditions in which the goal was to have block A on top of itself. However, one of the statements in the database states that it is impossible for anything to be on top of itself, thus such a goal would cause Rule 1A to fire and print the 'impossible' message as seen above.

If the Goal_state is consistent with the given database, then Rule 1A does not fire and control is passed to Rule 1B which is the main entry point into the plan generating routines that follow. If, for whatever reason, a plan could not be generated, then Rule 1C would fire with the 'fail' message. This is summarized in the first part of Figure 3.3.

To generate a plan, Rule 1B is given not only the Goal_state, but also the Initial_state as well. A series of actions is then sought to transform this Initial_state into the Goal_state. All allowable actions are listed in the

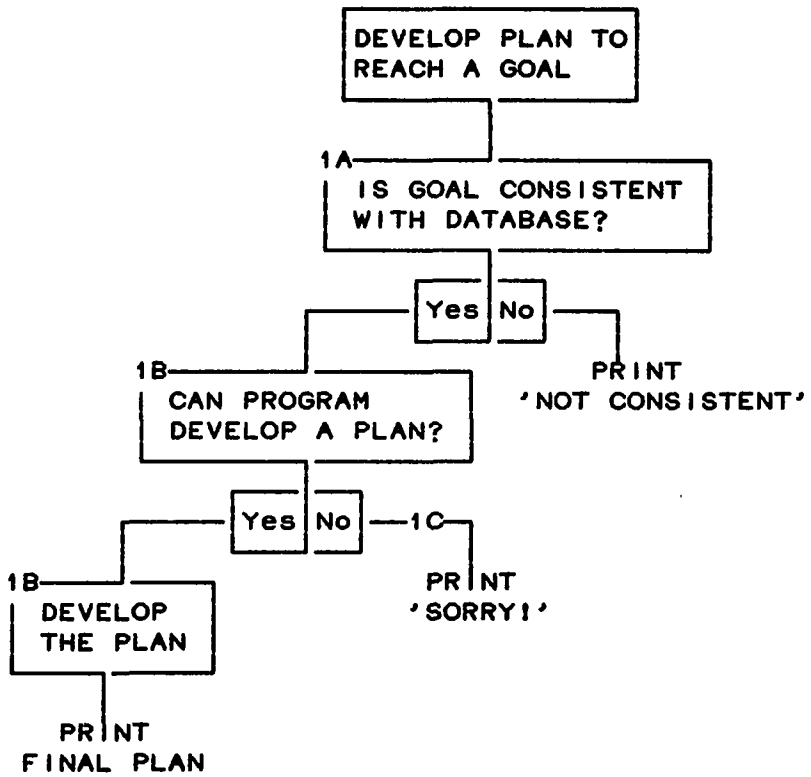


Fig. 3.1. Rule 1 Flowchart

RULE 1A

```
developa_plan(_,
              Goal_state) :-
  not(are_consistent_with(true,
                          Goal_state)),
  !,
  write('YOUR GOAL IS NOT CONSISTENT WITH THE GIVEN FACTS'),
  nl,
  write('PLEASE COMPARE YOUR GOAL WITH THE LIST OF'),
  write('IMPOSSIBLE CONDITIONS IN THE DATABASE'),
  nl.
```

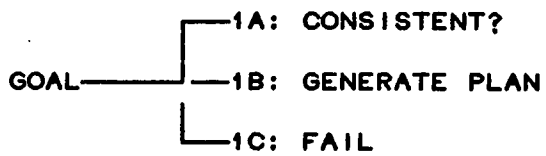
RULE 1B

```
developa_plan(Initial_state,
              Goal_state) :-
    solvea_plan(Goal_state,
                true,
                Initial_state,
                Final_plan),
    printa_plan(Final_plan),
    !.
```

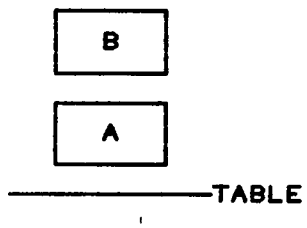
RULE 1C

```
developa_plan(_,
              _) :-
    write('IF THE PROGRAM HAS GOT THIS FAR,'),
    nl,
    write('THEN IT CAN NOT DEVELOP A PLAN . . . SORRY!'),
    nl.
```

Fig. 3.2. Rule 1 in Prolog



PRESENT STATE
block A on table
block_B on block A
block_A clear



ACTION
DELETE
block_B on block_A
ADD
block_B on table
block_A clear

NEXT STATE
block A on table
block_B on table
block_A clear
block_B clear

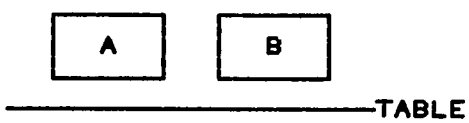


Fig. 3.3. Plan Generation

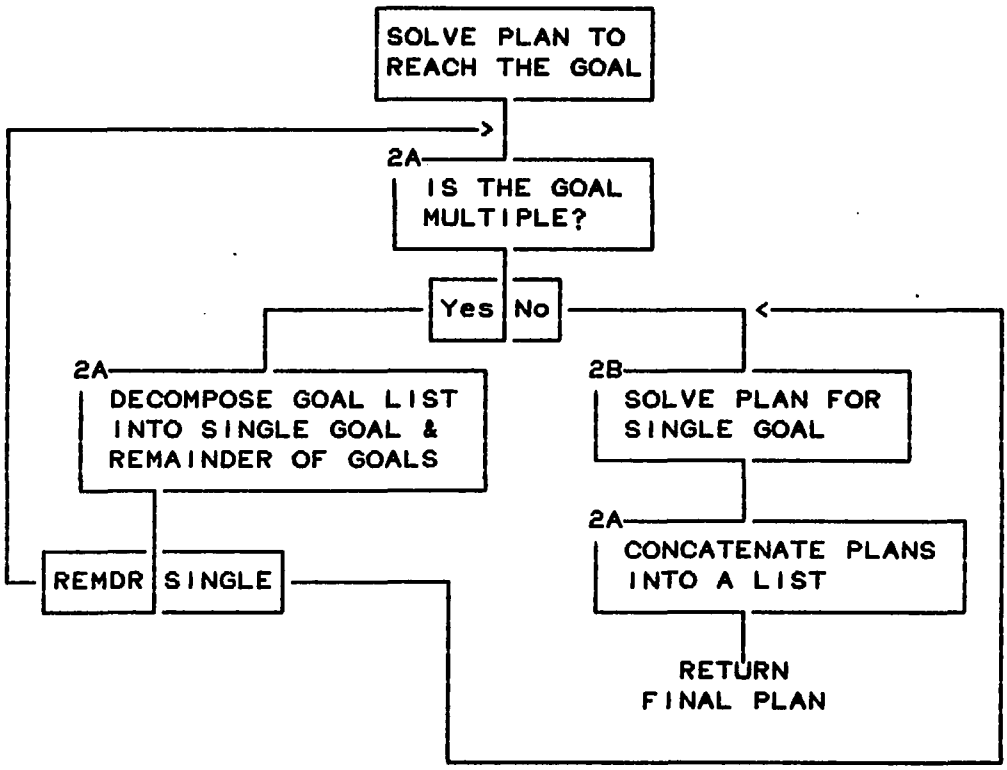


Fig. 3.4. Rule 2 Flowchart

RULE 2A

```
solvea_plan(Goal<&>Goals,
            Current_solved_goals,
            Current_plan,
            Final_plan) :-
    !,
    solvea_goal(Goal,
                Current_solved_goals,
                Updated_solved_goals,
                Current_plan,
                New_plan),
    solvea_plan(Goals,
                Updated_solved_goals,
                New_plan,
                Final_plan).
```

RULE 2B

```
solvea_plan(Goal,
            Current_solved_goals,
            Current_plan,
            New_plan) :-
    solvea_goal(Goal,
                Current_solved_goals,
                Updated_solved_goals,
                Current_plan,
                New_plan).
```

Fig. 3.5. Rule 2 in Prolog

database in a generalized form. When a qualified action acts upon a present state in order to transform it into a new state, some facts may be deleted and others added, as seen in Figure 3.3.

To generate a plan, Rule 1B calls the two subgoals 'solvea_plan', by means of which the plan is generated, and 'printa_plan', which prints the final solution.

Rule 2

If there is only a single goal to be solved, Rule 2A is skipped because it is looking for a list of goals, and Rule 2B will then take that single goal and attempt to solve for it by calling the subgoal 'solvea_goal'. However, if a list of goals is to be solved then Rule 2A is fired and decomposes the task into solving the first goal in the list by 'solvea_goal' and then recursively solving the remainder of the goals as a subplan using 'solvea_plan'. Rule 2A must eventually use Rule 2B as an exit in the recursive loop when the last goal in the goal list is solved. This is shown in Figures 3.4 and 3.5.

Rule 3

Figures 3.6 and 3.7 summarize the four ways by which to 'solve' a goal. Rule 3A states that if a goal is always true, as stated in the database, then there is no need to go any further. In this case nothing changes and the currently solved goals are the same as the updated ones, and the

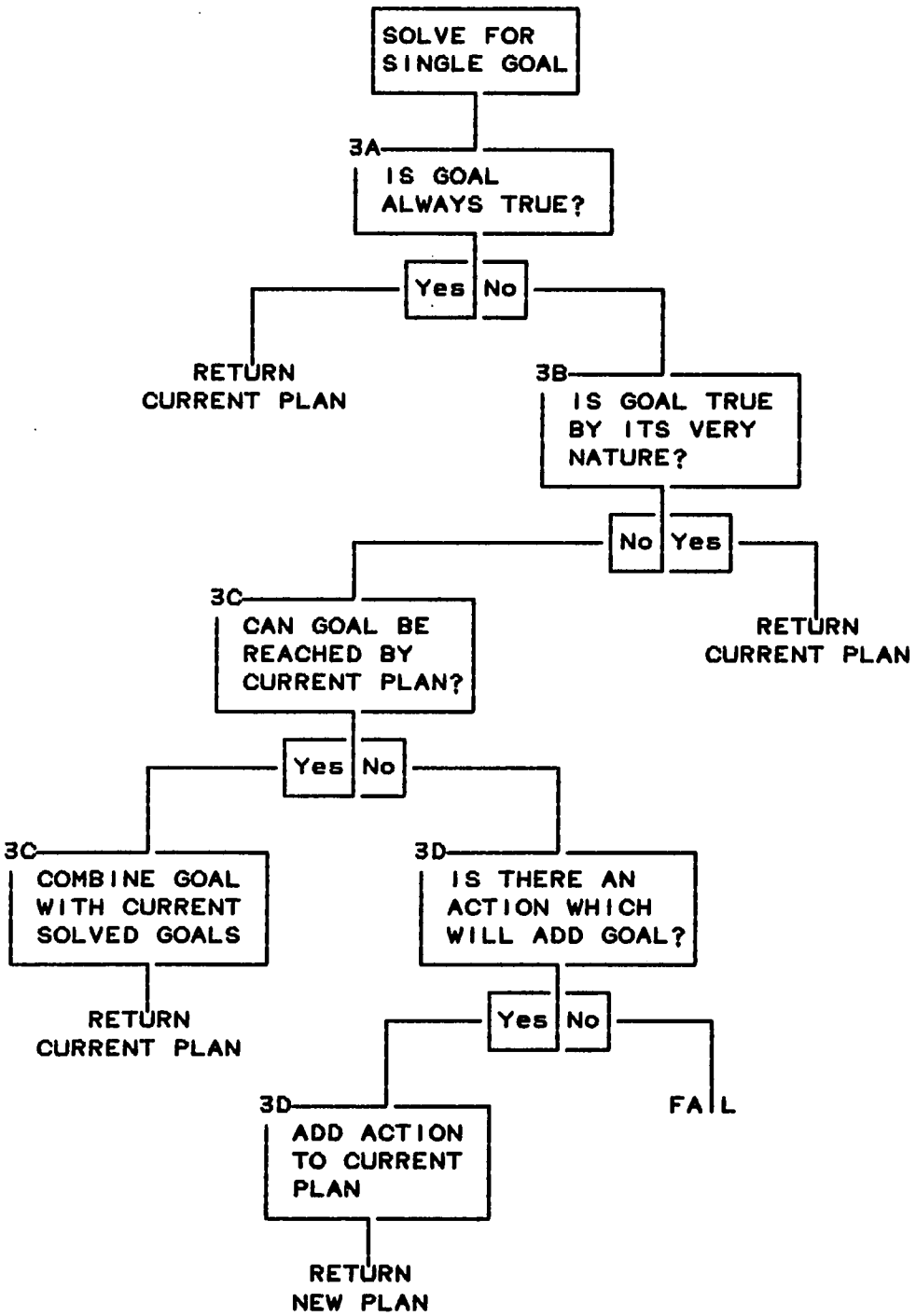


Fig. 3.6. Rule 3 Flowchart

```
                    RULE 3A

 solvea_goal(Goal,
             Current_solved_goals,
             Current_solved_goals,
             Current_plan,
             Current_plan) :-
     is_always_true(Goal).

                    RULE 3B

solvea_goal(Goal,
             Current_solved_goals,
             Current_solved_goals,
             Current_plan,
             Current_plan) :-
     Goal.

                    RULE 3C

solvea_goal(Goal,
             Current_solved_goals,
             Updated_solved_goals,
             Current_plan,
             Current_plan) :-
     canbe_achieved_by(Goal,
                       Current_plan),
     combine_together(Goal,
                      Current_solved_goals,
                      Updated_solved_goals).

                    RULE 3D

solvea_goal(Goal,
             Current_solved_goals,
             Goal<&>Current_solved_goals,
             Current_plan,
             New_plan) :-
     will_add(Action,
              Goal),
     add_Action_to_Plan(Action,
                        Goal,
                        Current_solved_goals,
                        Current_plan,
                        New_plan).
```

Fig. 3.7. Rule 3 in Prolog

current plan equals the new.

is_always_true(inroom(lightswitch,room(1))).

It is always true that the lightswitch is located in room #1.

Rule 3B is similar in that if a goal is true by its very nature then the new goals and plans are same as the current ones.

box_A = box_A

would automatically return 'true' by the very nature of the definition of the operator '='.

If Rules 3A and 3B do not fire, the program next checks to see if the goal can be achieved by the current plan and, if so, there is no need to seek any further action to bring it about. Simply combine this goal together with the currently solved goal list and then let the new plan equal the current one, as can be seen in Rule 3C. Otherwise, Rule 3D will check if there is some action which will add this goal and, if there is, such an action should be added to the current plan by the predicate 'add_Action_to_Plan'. Also note that this goal is combined with the currently solved goals in the third argument of Rule 3D to be returned as the updated solved goals in Rule 2.

Goal<&>Current_solved_goals /* in Rule 3D */

is returned as

Updated_solved_goals /* in Rule 2 */

Rule 4

The plan consists of a list of actions which will bring about the desired goal. Rule 4, Figures 3.8 and 3.9, states that the two ways in which to add a new action to the current plan are either by adding the new action at the end of the list or by inserting the action within the list. Since one does not want a new piece of the plan altering the partial plan determined up to that point, Rule 4A attempts to add the new action to the end of the current list by first checking that this new action will not delete any of the currently solved goals.

The next clause checks for any preconditions required by the new action, for instance,

```
precond_of(nextto(robot,box(Box)) <&>
                onfloor(robot),       /* precondition */
                climbon(box(Box))).   /* action */
```

a robot must be on the floor next to a box if it is to climb onto the box.

Once these preconditions are determined, then a check must be made to verify that they are consistent with the list of currently solved goals. By 'consistent with' is meant that the addition of these preconditions to the current plan would not imply any of the impossible conditions as listed in the database.

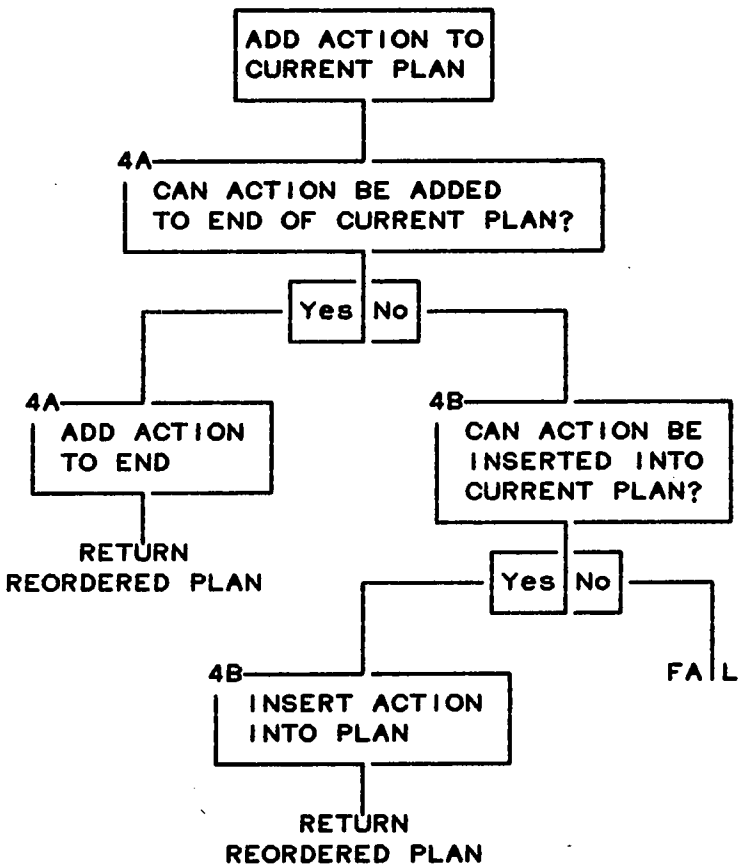


Fig. 3.8. Rule 4 Flowchart

RULE 4A

```
add_Action_to_Plan(New_action,
                   Goal,
                   Current_solved_goals,
                   Current_plan,
                   Reordered_plan<+>New_action) :-
    will_not_delete_listof(New_action,
                           Current_solved_goals),
    precond_of(Preconditions,
               New_action),
    are_consistent_with(Preconditions,
                        Current_solved_goals),
    solvea_plan(Preconditions,
                Current_solved_goals,
                Current_plan,
                Reordered_plan),
    will_not_delete_listof(New_action,
                           Current_solved_goals).
```

RULE 4B

```
add_Action_to_Plan(New_action,
                   Goal,
                   Current_solved_goals,
                   Previous_plan<+>Last_action,
                   Reordered_plan<+>Last_action) :-
    will_not_delete(Last_action,
                    Goal),
    remove_Results_oflast_action(Current_solved_goals,
                                 Last_action,
                                 Previous_solved_goals),
    add_Action_to_Plan(New_action,
                       Goal,
                       Previous_solved_goals,
                       Previous_plan,
                       Reordered_plan),
    will_not_delete(Last_action,
                    Goal).
```

Fig. 3.9. Rule 4 in Prolog

If all looks good up to this point, then the current plan must be modified in order to include the preconditions of the new action before the action itself can be added. This is done by recursively calling 'solvea_plan' which returns a reordered plan. This reordered plan is now ready to have the new action concatenated to its end, except for one final check. There was a check to make sure that the new action did not delete any of the currently solved goals, but now that check must be made again. Why? Because variables at the time of the first check may have become instantiated during 'solvea_plan'.

```
will_delete(move(b,X,table),on(b,X)).
```

says that regardless of what the variable X is set equal to, moving block 'b' from that 'X' onto the 'table' will logically require the deletion of the clause

```
on(b,X).
```

But this would not require the deletion of the specific clause,

```
on(b,a)
```

unless the variable 'X' has been instantiated to the object 'a'. So at the time of the first 'will_not_delete_listof' check, the situation may have included

```
move(b,X,table)
```

which would not require the deletion of

```
on(b,a)
```

However, because of 'solvea_plan', the variable 'X' may have

become instantiated to 'a', which would then require the deletion of

on(b,a)

which would have been missed during the first check. Because of this possibility, a second verification is made at the last clause of Rule 4A, 'just in case'.

If Rule 4A could not be satisfied, 4B attempts to insert the new action one step earlier in the current plan. Note that, as shown in Figure 3.10, the fourth argument of 4B has the current plan decomposed into the previous plan plus the last action. The first clause of the body of the rule makes sure that the last action in the current plan does not delete the new goal being solved. Otherwise, the new goal could be inserted earlier into the plan only to be deleted when that last action was added back. The next step in inserting the new action one step earlier in the current plan is to strip away the effects of the last action on the current goal list. Having done so, this list of previous goals can be used to add the new action onto the previous plan, and then to this reordered plan can be concatenated the last action, resulting in the desired new plan.

The effects of the last action of the current plan are stripped away by the second clause of the body of the rule, 'remove_Results_oflast_action', and Rule 5 then returns the previous solved goal list as its third argument. This goal list is then used in the recursive call to

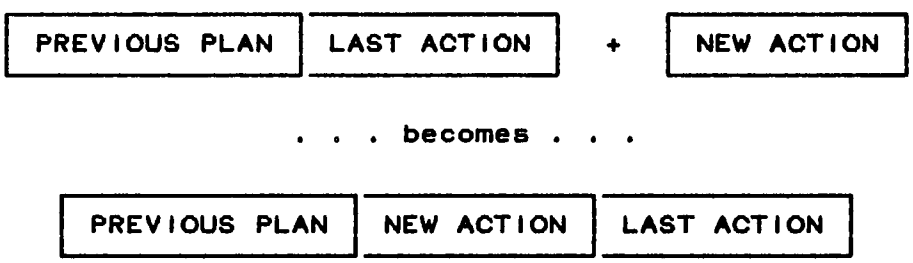


Fig. 3.10. Insert New Action into Plan

'add_Action_to_Plan' which, in turn, returns the reordered plan. As in Rule 4A, a second check is made to verify that no new variable instantiations would cause a deletion of the new goal. The reordered plan is concatenated onto the last action that had originally been stripped away, in the fifth argument at the head of Rule 4B.

Rules 5 and 6

Rule 6, Figures 3.13 and 3.14, removes from the current goal list any goals added by the last action or any of its preconditions. Rule 5, Figures 3.11 and 3.12, then adds back those preconditions. The result is that all goals added by the last action, but which are not present in its preconditions, are deleted from the current goal list.

Preconditions of action:	A, B
Goals added by action:	B, C
Current goal list:	A & B & C & D & true
Updated goal list:	A & B & D & true

'D' is not deleted from the current goal list because it is not found among the preconditions nor among the added goals. 'C' is deleted because it is found among the added goals though not among the preconditions. 'A' is deleted by Rule 6 because it is among the preconditions , but added back by Rule 5. 'B' would be deleted either because it is found among the preconditions or found among the added goals, yet also added back by Rule 5.

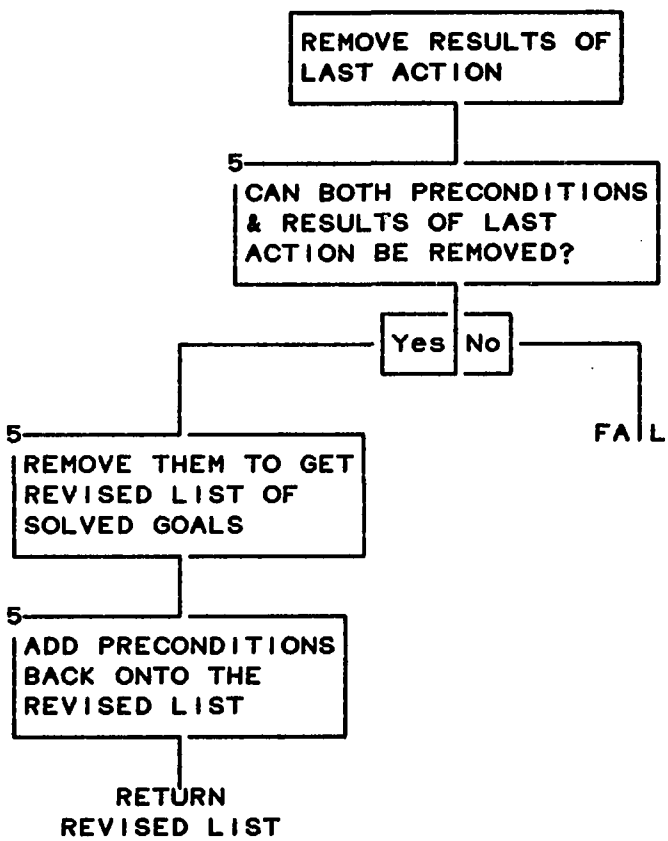


Fig. 3.11. Rule 5 Flowchart

```
remove_Results_oflast_action(Current_solved_goals,
                             Last_action,
                             Previous_solved_goals) :-
    precond_of(Preconditions,
               Last_action),
    remove_Preconditions_and_Results_oflast_action(
               Current_solved_goals,
               Last_action,
               Preconditions,
               Revised_solved_goals),
    append(Preconditions,
           Revised_solved_goals,
           Previous_solved_goals).
```

Fig. 3.12. Rule 5 in Prolog

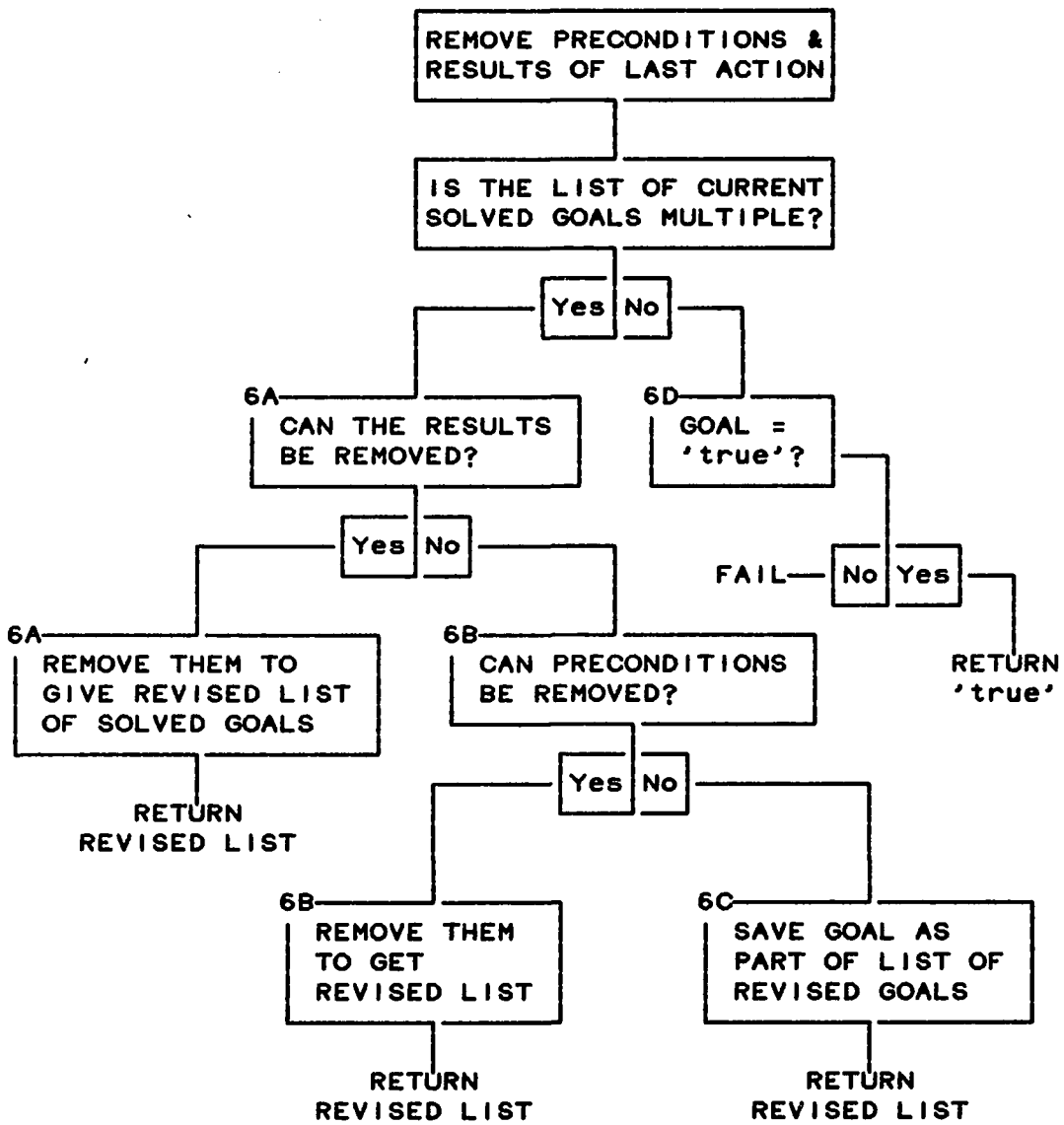


Fig. 3.13. Rule 6 Flowchart

```
RULE 6A
    remove_Preconditions_and_Results_oflast_action(
              Front_goal<&>Remainder_ofsolved_goals,
              Last_action,
              Preconditions,
              Revised_solved_goals) :-
         will_add(Last_action,
                  Result),
         Front_goal == Result,
         !,
         remove_Preconditions_and_Results_oflast_action(
                  Remainder_ofsolved_goals,
                  Last_action,
                  Preconditions,
                  Revised_solved_goals).
RULE 6B
    remove_Preconditions_and_Results_oflast_action(
              Front_goal<&>Remainder_ofsolved_goals,
              Last_action,
              Preconditions,
              Revised_solved_goals) :-
         isa_element_of(Subcondition,
                        Preconditions),
         Front_goal == Subcondition,
         !,
         remove_Preconditions_and_Results_oflast_action(
                                Remainder_ofsolved_goals,
                                Last_action,
                                Preconditions,
                                Revised_solved_goals).
RULE 6C
    remove_Preconditions_and_Results_oflast_action(
              Front_goal<&>Remainder_ofsolved_goals,
              Last_action,
              Preconditions,
              Front_goal<&>Revised_solved_goals) :-
         remove_Preconditions_and_Results_oflast_action(
                                Remainder_ofsolved_goals,
                                Last_action,
                                Preconditions,
                                Revised_solved_goals).
RULE 6D
    remove_Preconditions_and_Results_oflast_action(
                                      true,
                                      Last_action,
                                      Preconditions,
                                      true).
```

Fig. 3.14. Rule 6 in Prolog

So why delete the preconditions by Rule 6 if they are just added back by Rule 5? First, it is easier to just delete all the preconditions and add them back than to keep track of a partial list and then to add back the missing elements. Secondly, it keeps the goal list in the proper sequence.

The first clause in the body of Rule 5 searches the database for the preconditions of the last action in the current goal list, and this information will later be used in Rule 6B. The second clause calls Rule 6 which removes both the results and preconditions of the last action from the currently solved goals and returns a revised list of these solved goals. The third clause then produces the desired list of previously solved goals by appending the preconditions back onto the revised goal list.

The bulk of the work is done by Rule 6. Rule 6A removes results, 6B removes preconditions, 6C saves a goal that is not deleted either by 6A or 6B, and 6D is the terminating condition reached when the final goal in the list ('true') is reached.

In order to remove the results of the last action, Rule 6A first searches the database for all facts that the last action could add. Each fact/result is then compared with the front goal in the current goal list to see if they are identical, and if they are then this fact/result/goal is deleted by the fourth clause of the body of the rule. The

fourth clause recursively seeks to remove results and preconditions of the remainder of the solved goal list, and since this remainder does not include that front goal, then this goal is 'lost' by simply ignoring it. Once a solution is found for the second clause, there can be no further solutions that would be identically equal to the front goal and yet different from the result already obtained. For this reason, the third 'clause', called the cut symbol, is used to insure that there will be no backtracking to attempt other solutions of the first two clauses if clause four should fail.

There might be more than one element in the precondition list, so in order to remove these preconditions, each element must be checked one at a time. The first clause of the body of Rule 6B decomposes the preconditions into subconditions that are single elements. Each of these subconditions are then checked by the second clause, similar to Rule 6A, to see if they are identically equal to the front goal of the current goal list. Again, in a fashion identical to Rule 6A, if there is a match, this subcondition/goal is deleted in clause four by simply ignoring it when the recursive call is made on the remainder of the goal list. 'Clause' three is the cut symbol which does not allow further attempts to solve clauses one and two if clause four should fail.

Suppose a goal/result/precondition is not deleted by either Rule 6A or 6B, in this case 6C 'saves' the goal by concatenating it to the front of the revised goal list in the fourth argument of the head of Rule 6C. Note that in Rules 6A and 6B this fourth argument is simply,

Revised_solved_goals

whereas in 6C it is,

Front_goal<&>Revised_solved_goals.

The only clause in the body of 6C recursively calls for a solution to the revised goal list using the remainder of the currently solved goals.

Recall that the second argument in the first clause of the body of Rule 1B was 'true', and that this was passed to the second argument in the head of Rule 2 which is called 'Current_solved_goals'. Thus the Prolog predicate 'true' is assigned to be the first element in the list of currently solved goals. This is used as an ending/exiting condition in Rule 6D. When the last goal of the goal list (i.e., 'true') is reached, then it will not match Rules 6A, 6B, or 6C because they all require a compound first argument at their head. Rule 6D only allows a single first argument that is identically equal to the Prolog predicate 'true', consequently there will be a match with the first argument of this rule when the last goal of the current goal list is finally reached. No matter what the second and third arguments are, the value of the first argument will be

passed to the fourth argument, which is the position of the revised goal list. Thus the last, or first, depending on how one views it, goal of the revised goal list will be the Prolog predicate 'true' as required by the program in case a future call is made to remove the results of some action.

Rule 7

This rule, Figures 3.15 and 3.16, was called as the first clause in the body of Rule 3C in order to determine if a given goal could already be achieved by the current plan. There are only two ways that this could take place: (1) the goal could be added by an action in the current plan, or (2) the goal could already exist in the initial state.

Rule 7A decomposes the current plan into the last action plus the remainder of the plan. The body of the rule searches through the database to determine if that last action would add the given goal. If 7A fails, then 7B repeats the process using the next to last action, which it does by recursively calling the routine on the remainder of the plan in the third clause. However, before making this recursive call, a check must first be made in the second clause to make sure that the last action will not delete the goal. Otherwise, an action within the current plan could be found to add the goal, only to have it later deleted when that last action was added.

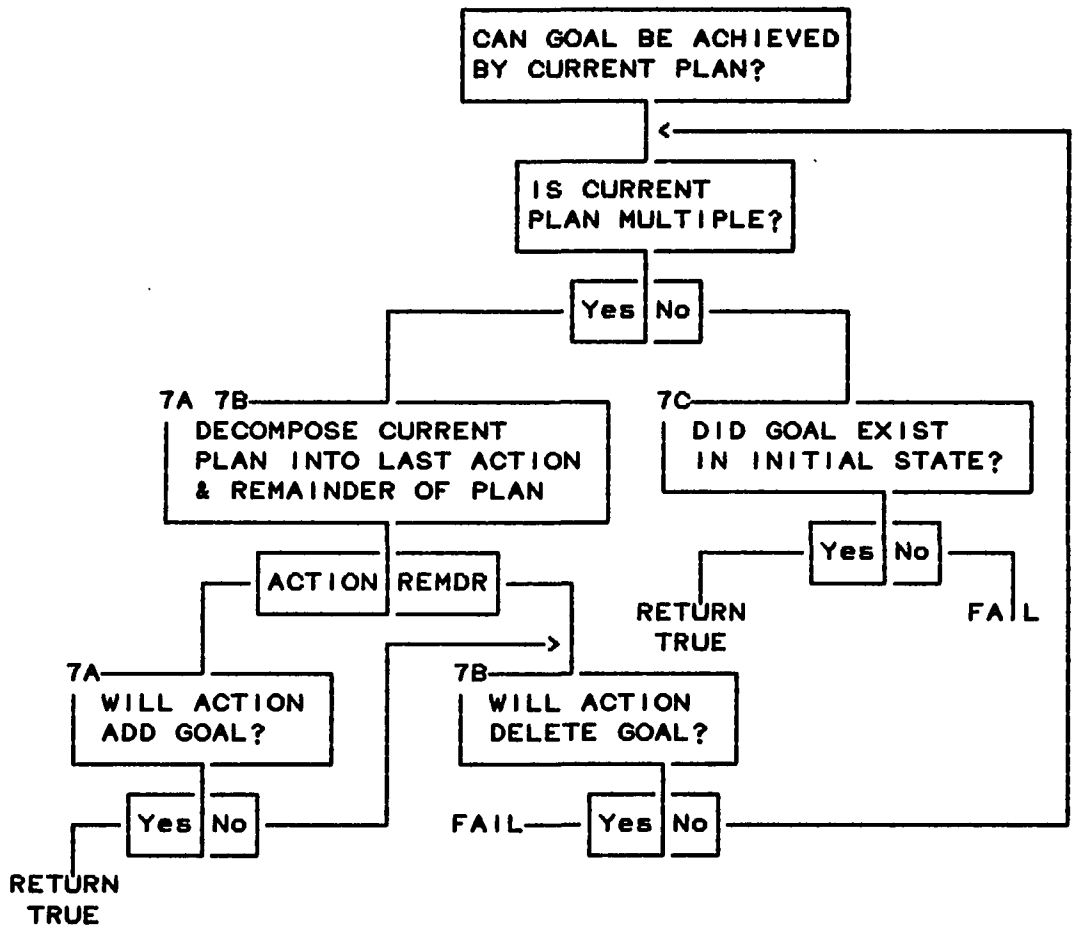


Fig. 3.15. Rule 7 Flowchart

```
                        RULE 7A
canbe_achieved_by(Goal,
                  Remainder_ofplan<+>Last_action) :-
    will_add(Last_action,
             Goal).


                        RULE 7B

canbe_achieved_by(Goal,
                  Remainder_ofplan<+>Last_action) :-
    !,
    will_not_delete(Last_action,
                    Goal),
    canbe_achieved_by(Goal,
                      Remainder_ofplan),
    will_not_delete(Last_action,
                    Goal).


                        RULE 7C

canbe_achieved_by(Goal,
                  Initial_state) :-
    exists_in(Goal,
              Initial_state).
```

Fig. 3.16. Rule 7 in Prolog

The repetition of clauses two and four is for the same purpose as in Rule 4, that is, to verify that any variables instantiated by the third clause does not result in deletion of the goal.

Finally, if there is no action that will produce the goal, the second possibility for which to check is that this goal already existed in the initial state. Note that the second argument in the head of Rules 7A and 7B are compound. This means that any single element variable will not make a match until Rule 7C is reached, which is the case when the last action in the current plan is reached. The first argument in the head of Rule 1B was the initial state and it was passed to the third argument of the first clause of the body of that rule. This first clause is Rule 2 and its third argument is the current plan. Thus, the initial state is passed on as the first step in the current plan. When Rule 7 successively strips away actions from the current plan, it finally gets to that last action which is the initial state.

Rules 8 and 9

An example of the use of Rules 8 and 9, Figures 3.17 and 3.18, can be seen in Rule 4 where they are used to verify that an action does not delete any of the currently solved goals. The bulk of the work is done by Rule 9, with Rule 8 basically used to decompose the list of goals into its component parts. The second argument of the head of

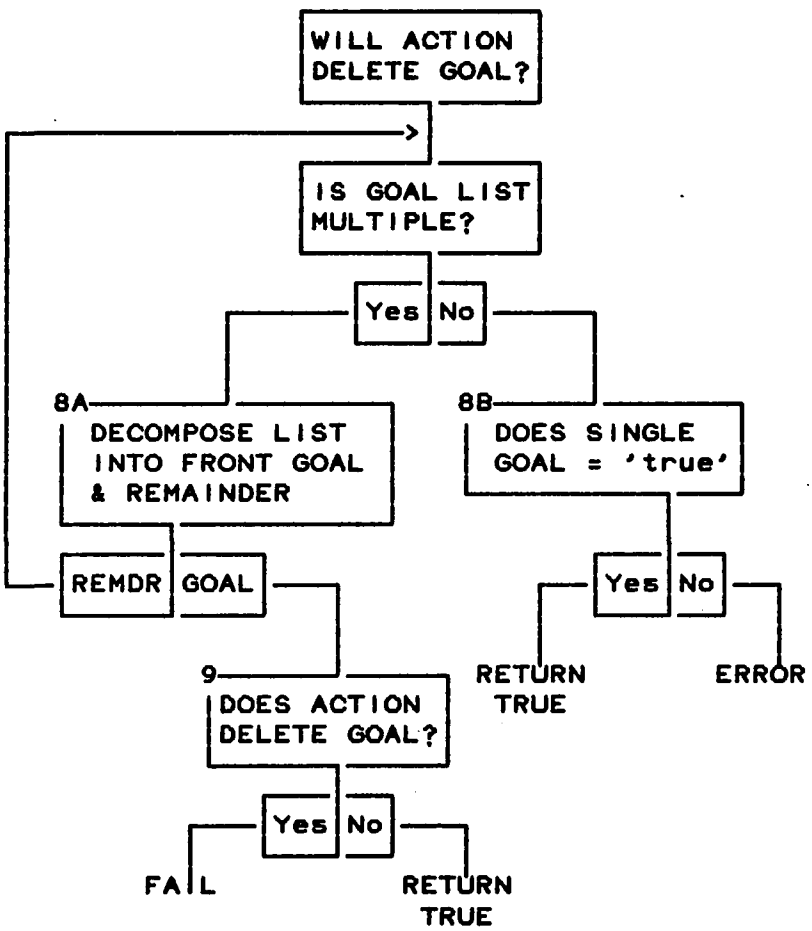


Fig. 3.17. Rules 8 and 9 Flowchart

```
                        RULE 8A

will_not_delete_listof(Action,
                       Front_goal<&>Remainder_of_goals) :-
    will_not_delete(Action,
                    Front_goal),
    will_not_delete_listof(Action,
                           Remainder_of_goals).


                        RULE 8B

will_not_delete_listof(Action,
                       true).




                        RULE 9A

will_not_delete(Action,
                Goal) :-
    instantiate_variables(Goal<&>Action,
                          O,
                          N),
    will_delete(Action,
                Goal),
    !,
    fail.


                        RULE 9B

will_not_delete(Action,
                Goal).
```

Fig. 3.18. Rules 8 and 9 in Prolog

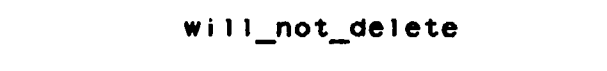
Rule 8A does the decomposition and the first clause of the body calls Rule 9 to verify that the action will not delete the front goal. Clause two is a recursive call on the remainder of the goals. Rule 8B fires only when the last goal of the goal list is reached, namely the Prolog predicate 'true'.

The approach taken in Rule 9 is that if it cannot be proved that an action will delete a goal, then it is to be assumed that it will not. This can be seen in the second clause of the body of Rule 9A where the database is searched for a clause stating that the action will indeed delete the goal. If one is found then the third clause is reached which is the cut symbol meaning that there can never be an attempt to backtrack to clause two in order to seek another possible solution. Instead, the fourth clause is reached which causes Rule 9 to fail, that is, both 9A and 9B.

Rule 9A will only fail if clause two is satisfied, that is, it is found that the action will delete the goal,

therefore, 'will_not_delete' fails, meaning that 'will_delete' is true in this case, as it should be, and Rule 9 fails. On the other hand, if the second clause in the body of Rule 9A fails, then Rule 9B will fire. Notice that 9B has only a head with no body. This means 9B will automatically succeed, that is, return the value 'true'.

Thus,



means 9A fails but 9B succeeds, and Rule 9 succeeds.

The first clause in the body of Rule 9A instantiates any uninstantiated variables, in both the goal and the action, to specific values. The reason for this is that if variables were passed to the second clause, an error could result because of the way Prolog attempts to equate variables by matching them together in any way it can. Each variable is given a specific value in order to avoid possible problems, and,

```
on(X,a) &
on(b,Y) &
move(X,c,a) &
clear(X) &
delete(clear(Y))
```

would be converted to,

```
on(tempVALUE(0),a) &
on(b,tempVALUE(1)) &
move(tempVALUE(0),c,a) &
clear(tempVALUE(0)) &
delete(clear(tempVALUE(1))
```

Note that all occurrences of 'X' are converted to the same constant, not different constants for each time it appears.

If these constants were allowed to remain on exit

from Rule 9A, then the entire program would go into limbo, but they are removed during backtracking. See Rule 10 for further clarification of how clause one operates.

Rules 10 and 11

WHAT do these rules do?

In order to understand how these rules operate, first look at the following example,

INPUT

```
instantiate_variables(foo(Variable_1,
                          constant,
                          Variable_2),
                      0,
                      Lastnum).
```

OUTPUT

```
Variable_1 = tempVALUE(0)
Variable_2 = tempVALUE(1)
Lastnum = 2
```

The functor 'foo' has three agruments, one constant and two variables. The result of applying Rules 10 and 11 is to instantiate the UNinstantiated variables of 'foo', so that its arguments are now three constants, so to speak. But not exactly, for there is actually the one original constant and the two variables that have been instantiated to values.

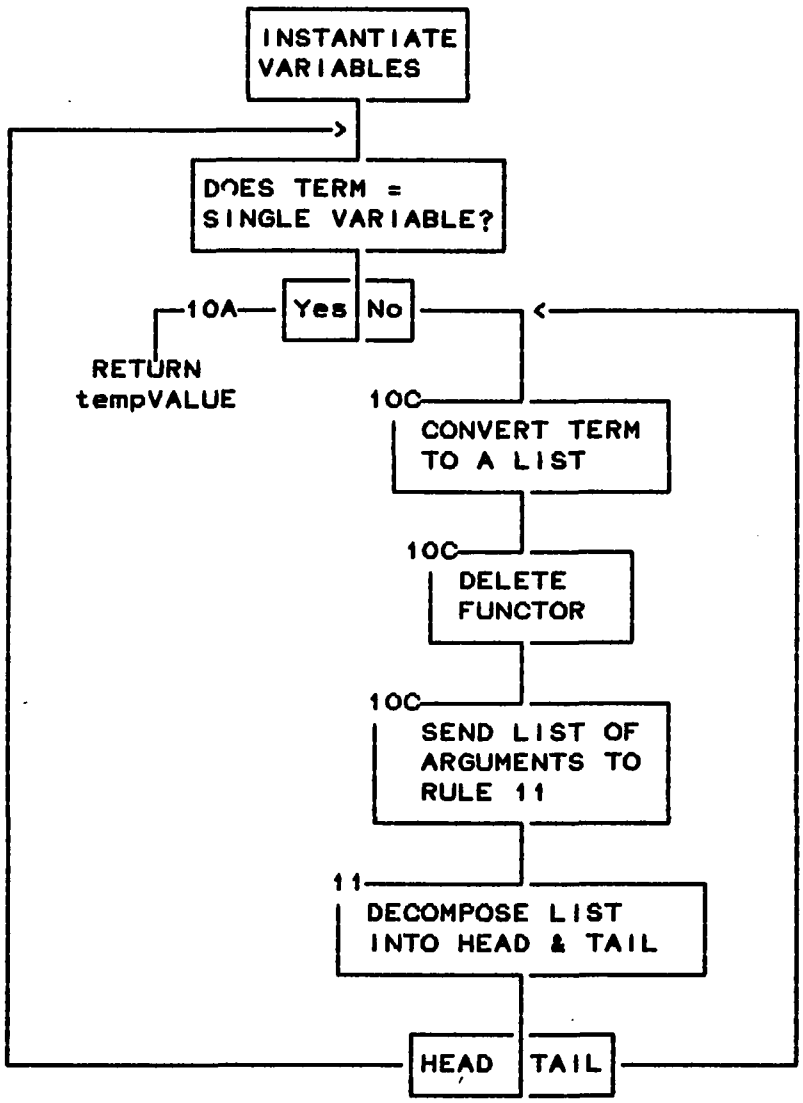


Fig. 3.19. Rules 10 and 11 Flowchart

RULE 10A

```
instantiate_variables(tempVALUE(Firstnum),
                      Firstnum,
                      Lastnum) :-
    !,
    Lastnum is Firstnum + 1.
```

RULE 10B

```
instantiate_variables(tempVALUE(Firstnum),
                      Lastnum,
                      Lastnum) :-
    !.
```

RULE 10C

```
instantiate_variables(Term,
                      Firstnum,
                      Lastnum) :-
    Term =.. [Funktor|Arguments],
    instantiate_listof_variables(Arguments,
                                 Firstnum,
                                 Lastnum).
```

RULE 11A

```
instantiate_listof_variables([Head|Tail],
                             Firstnum,
                             Lastnum) :-
    instantiate_variables(Head,
                          Firstnum,
                          Tempnum),
    instantiate_listof_variables(Tail,
                                 Tempnum,
                                 Lastnum).
```

RULE 11B

```
instantiate_listof_variables([],
                             Firstnum,
                             Firstnum).
```

Fig. 3.20. Rules 10 and 11 in Prolog

The '0' is the starting point from which to number these values, and 'Lastnum' is one greater than the last numbered variable.

WHY do it?

Before looking specifically at their usage in this program, a similar application will help clarify some issues. Prolog has various built-in predicates for testing and making things equal. Two of them are '=' and '=='. The first attempts to match an uninstantiated variable to equal anything it can. If a match can possible be made, then the goal succeeds. The second predicate represents a much stricter equality test in the way it handles variables.

```
X = Y.     --> TRUE
X == Y.    --> FALSE
X == X.    --> TRUE
```

However, X == Y could succeed in the following situation,

```
X = Y, X == Y.      --> TRUE
```

because '=' first instantiates Y to whatever value X has, then, since they are now set equal to the same value, '==' will return 'true'. Thus, '==' only succeeds in matching two uninstantiated variables when they are already sharing the same value.

Since it is possible in Prolog to define new operators, the following is a valid definition of an

equality operator that lies intermediate between '=' and '==='.

```
:- op(700, xfx, ===).
===(Term1,Term2) :-
    not(not(instantiate_variables(Term1,0,Lastnum),
            instantiate_variables(Term2,0,Lastnum),
            Term1 = Term2)).
```

The purpose of the operator '===' is to compare two Prolog terms as in the case of '=' and '=='. Rules 10 and 11 are used to instantiate all UNinstantiated variables in these two terms. After instantiation, the predicate '=' is used to determine equality because now there will be no variables that could be instantiated with the resultant possible weakening of equality. The double usage of 'not' is a Prolog 'trick'. The first two arguments of the 'inner not' will succeed without any trouble. If the third term, 'weak equality', succeeds, then everything within the inner parenthesis succeeds, so that an attempt to satisfy the 'inner not' fails because all its arguments succeeded. The 'outer not' then reverses this decision and succeeds. The result is that the body of this rule returns the same value as does the clause Term1 = Term2. But besides determining equality, one wants to undo these temporary instantiations when done, otherwise, variables instantiated within the body of the rule could be passed back to the head, and from there to the rules which first called it.

It must be remembered that these variables are instantiated to values, not that they are converted into constants. The difference being that in the first case the variables are given values, and in the second, the variables would no longer exist. If the variables were converted into constants, then there would be no way to undo the side effects of this rule and havoc could result. However, since they are still variables, the double not undoes all variable instantiation. This can be done in Prolog because when a goal fails, any variables that became instantiated must 'forget' what they stood for (Clocksin and Mellish 1984, p. 129), that is, they become uninstantiated. When Term1 = Term2 succeeds, then the 'inner not' goal will fail and the variables will 'forget' their values, but the 'outer not' will insure the correct true/false value is returned.

To summarize, if '===' succeeds, it is the 'inner not' that fails and thus uninstantiates all variables. If '===' fails, it is because the weak equality '=' failed, causing the 'inner not' to succeed, the 'outer not' to fail, and thus the 'outer not' would accomplish the uninstantiation. In either case, all variables must 'forget' what they previously stood for so that no misleading information may be passed back from the body to the head of the rule, yet the requested true/false value is sent back to the head.

	=	===	==
X = foo(Y)	T	F	F
foo(X) = foo(Y)	T	T	F
foo(X) = foo(X)	T	T	T
foo(X) = foo(a)	T	F	F
foo(X,a) = foo(Y,a)	T	T	F
foo(X) = foo(bar(Y))	T	F	F

Fig. 3.21. Equality Operator Comparisons

Figure 3.21 presents some examples for comparison. Instantiations take place as follows,

1. foo(X) === foo(Y)

   foo(tempVALUE(0)) = foo(tempVALUE(0))

   ==> True

2. foo(X) === foo(a)

   foo(tempVALUE(0)) = foo(a)

   ==> False

3. X === foo(Y)

   tempVALUE(0) = foo(tempVALUE(0))

   ==> False

In summary, '=' tries to match an uninstantiated variable to anything it can, whereas '==' will only succeed when two variables already share the same value, that is, the two terms must be identically equal in all respects. The new operator '===' is an intermediate notion of equality to determine if two terms are alphabetic variants, that is, the terms can be made syntactically identical by consistently changing their variable names.

It is to avoid any 'wild' matchings that might occur with uninstantiated variables that Rules 10 and 11 are used in Rule 9A to determine if an action will delete a goal. In searching the database for all the 'will_delete' clauses, one has a specific action and a specific goal in mind, although some variables maybe uninstantiated. For instance, the upper half of Figure 3.22 says that an action that moves

GENERAL RULE

```
will_delete(Action,
            at(Object,
               Location)) :-
     will_move(Action,
               Object).

will_move(pushto(Object1,
                 Object2,
                 Room),
          robot).

will_move(pushto(Object1,
                 Object2,
                 Room),
          Object1).
```

SPECIFIC INSTANCES OF RULE

```
will_delete(pushto(Object,
                   box_C,
                   room_2),
            at(Object,
               Location)).


will_delete(pushto(tempVALUE(0),
                   box_C,
                   room_2),
            at(tempVALUE(0),
               Location)).
```

Fig. 3.22. Result of Instantiating Variables

an object will cause that object to no longer be at its original location. The 'will_move' predicates state that moving one object next to another in a particular room causes both the object being pushed and the robot doing the pushing to move.

The rule in the lower half of Figure 3.22 would not give exactly the same results and could lead to errors because the first 'will_delete' in the lower half of the figure could match either the first or second 'will_move' in the upper half, whereas the second 'will_delete' in the lower half would only match the second 'will_move'. This is because in the first case 'Object' could match 'robot', but 'tempVALUE(0)' could not. Thus if one wants only to know if an action will delete the state information that a 'robot' is at a particular location, then that 'robot' will have to be specified.

If it still is not clear, then think of it as restricting the matchings being sought in the database from anything that Prolog might come up with, to only alphabetic variants, in order to make these matchings more specific.

HOW is it done?

Rules 10A and 10B will not fire initially, because the incoming term will have nothing to match the 'tempVALUE' in the first argument of their heads. Rule 10C serves to separate the functor from its arguments by using the 'univ'

predicate (=..), which transforms a term into a list in which the first element is the functor followed by its arguments. The second clause in the body of 10C then sends this list of arguments to Rule 11.

Rule 11A decomposes the list into its head and tail, and then recursively calls Rule 10 on that head. Thus a term might be of the form,

foo(bar(X), Y, Z)

and the initial call would be,

```
instantiate_variables(foo(bar(X), Y, Z),
                      0,
                      Lastnum).
```

Rule 10C would transform this to,

[foo, bar(X), Y, Z]

and would pass,

[bar(X), Y, Z]

to Rule 11. Rule 11 would separate the list into its head and tail,

head = bar(X)

tail = [Y,Z]

and then recursively call Rule 10 on this head. Rule 10C would again transform this term into the list,

[bar, X] and would then ignore the functor 'bar' as it sent the argument 'X' back down to Rule 11A which would recursively send it back to Rule 10! But this time things are different because an <u>uninstantiated variable</u> is sent

back, and this variable can be matched with the first argument in the head of Rule 10A. The value '0' is passed to tempVALUE(0) which is now the value of the variable that occupies that location, and 'Lastnum' is incremented by one. Note that this 'Lastnum' is then passed back to the first clause in the body of Rule 11A, and then transferred to the second argument in the second clause so that the next time around the variable will be 'tempVALUE(1)'.

Rule 10B is the terminating condition for variables, and 11B for empty lists.

Rule 12

Rule 12, Figures 3.23 and 3.24, determine if a given fact is consistent with the impossible conditions found in the database. If the database consists of,

```
impossible(on(Block1, Block2) <&>
           clear(Block2)).
impossible(on(Block1, Block2) <&>
           on(Block1, Block3) <&>
           notequal(Block2, Block3)).
impossible(on(Block1, Block1)).
```

then one would get responses to queries as follows,

```
are_consistent_with(on(a,b),
                    clear(b)).  --> FALSE
```

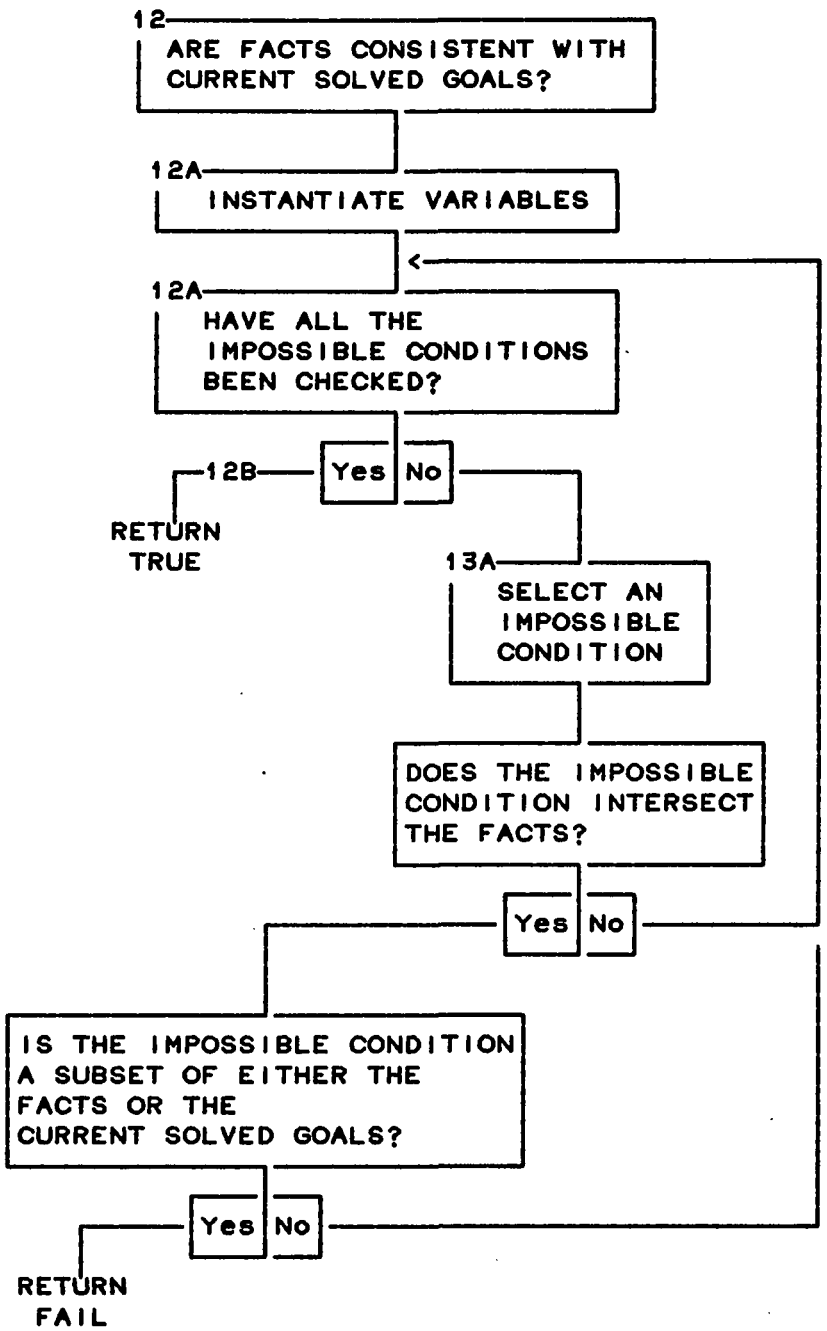


Fig. 3.23. Rule 12 Flowchart

RULE 12A

```
are_consistent_with(Facts,
                    Current_solved_goals) :-
    instantiate_variables(Facts<&>Current_solved_goals,
                          0,
                          N),
    impossible(Condition),
    not(not(intersect(Condition,
                      Facts))),
    isa_subset_of(Condition,
                  Facts<&>Current_solved_goals),
    !,
    fail.
```

RULE 12B

```
are_consistent_with(Facts,
                    Current_solved_goals).
```

Fig. 3.24. Rule 12 in Prolog

```
are_consistent_with(on(a,b) <&>
                    on(a,c)
                    on(d,e)).    --> FALSE
are_consistent_with(on(a,a),
                    on(c,d)).    --> FALSE
```

In the first instance above, the fact is not consistent with the current solved goal. In the second and third examples, the facts are not consistent regardless of the current solved goals.

Rules 13 and 14

These two rules, Figures 3.25, 3.26 and 3.27, are used to concatenate. In Rule 13 the goal is treated as singular while in Rule 14 it is treated as multiple. Examples follow,

```
combine_together(a, b, X).
     --> X = a <&> b
combine_together(a, b <&> c, X).
     --> X = a <&> b <&> c
combine_together(a <&> b, c <&> d, X).
     --> X = (a <&> b) <&> c <&> d
append(a, b, X).
     --> X = a <&> b
append(a<&>b, c <&> d, X).
     --> X = a <&> b <&> c <&> d
```

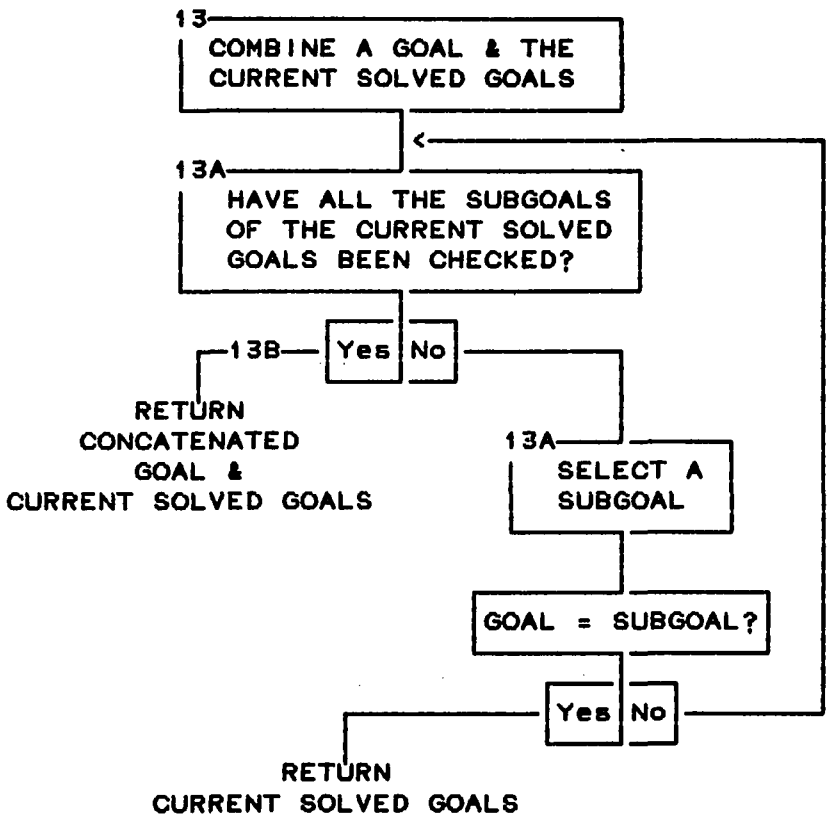


Fig. 3.25. Rule 13 Flowchart

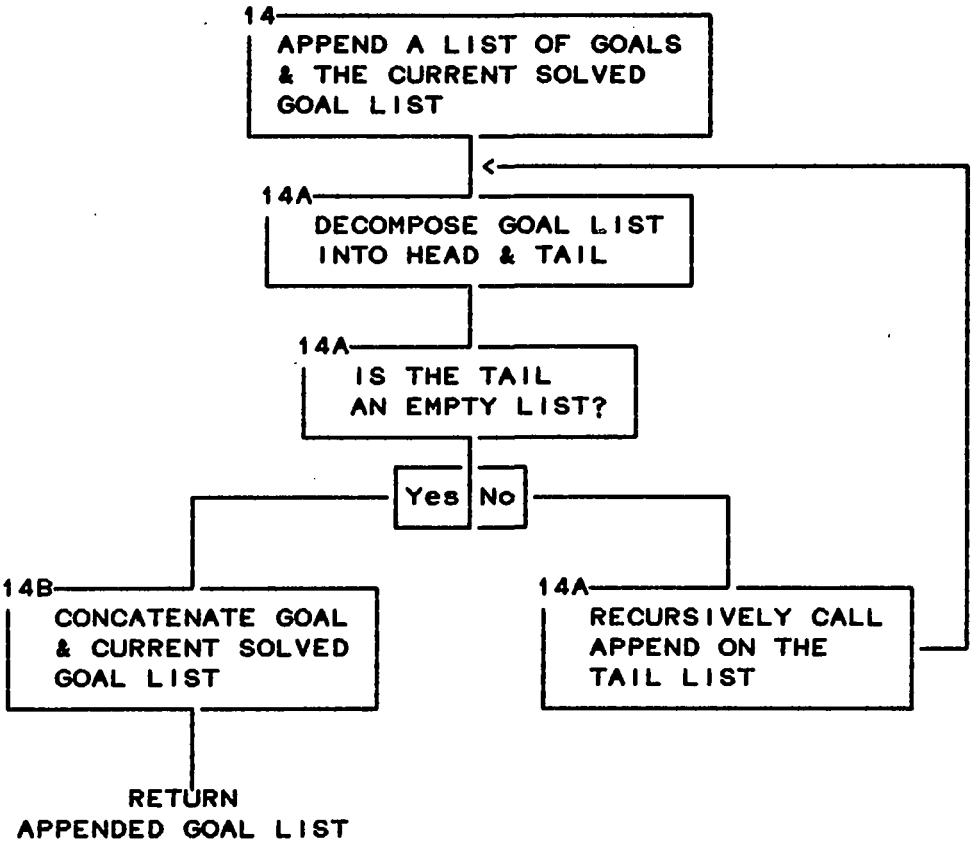


Fig. 3.26. Rule 14 Flowchart

RULE 13A

```
combine_together(Goal,
                 Current_solved_goals,
                 Current_solved_goals) :-
    isa_element_of(Another_goal,
                   Current_solved_goals),
    Goal == another_goal,
    !.
```

RULE 13B

```
combine_together(Goal,
                 Current_solved_goals,
                 Goal<&>Current_solved_goals).
```

RULE 14A

```
append(Goal<&>Goals,
       Current_solved_goals,
       Goal<&>Combined_solved_goals) :-
    !,
    append(Goals,
           Current_solved_goals,
           Combined_solved_goals).
```

RULE 14B

```
append(Goal,
       Current_solved_goals,
       Goal<&>Current_solved_goals).
```

Fig. 3.27. Rules 13 and 14 in Prolog

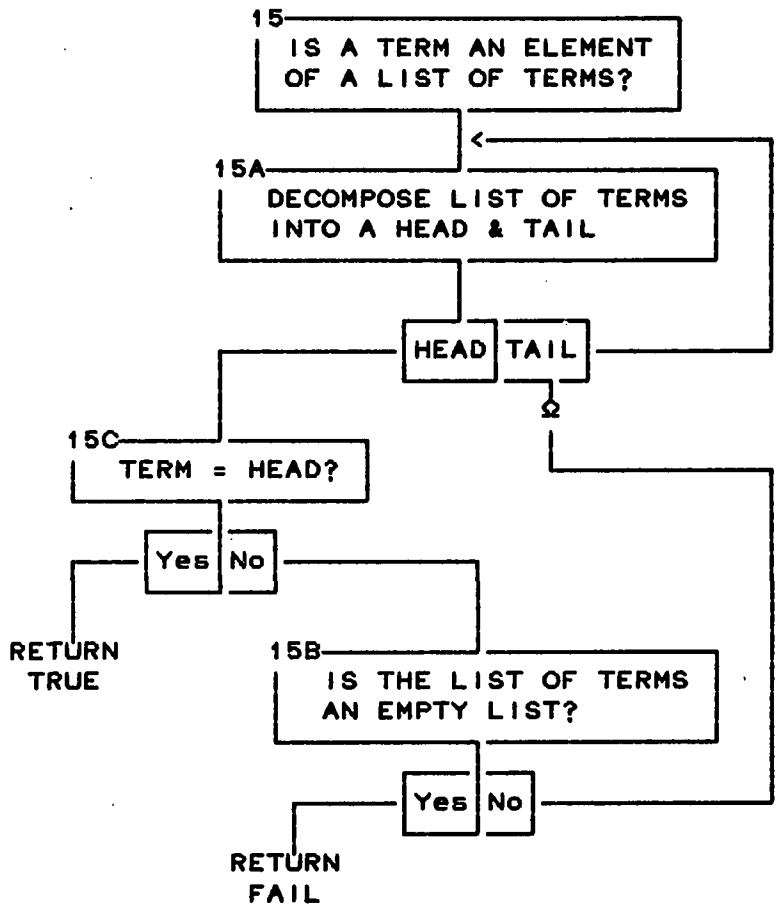


Fig. 3.28. Rule 15 Flowchart

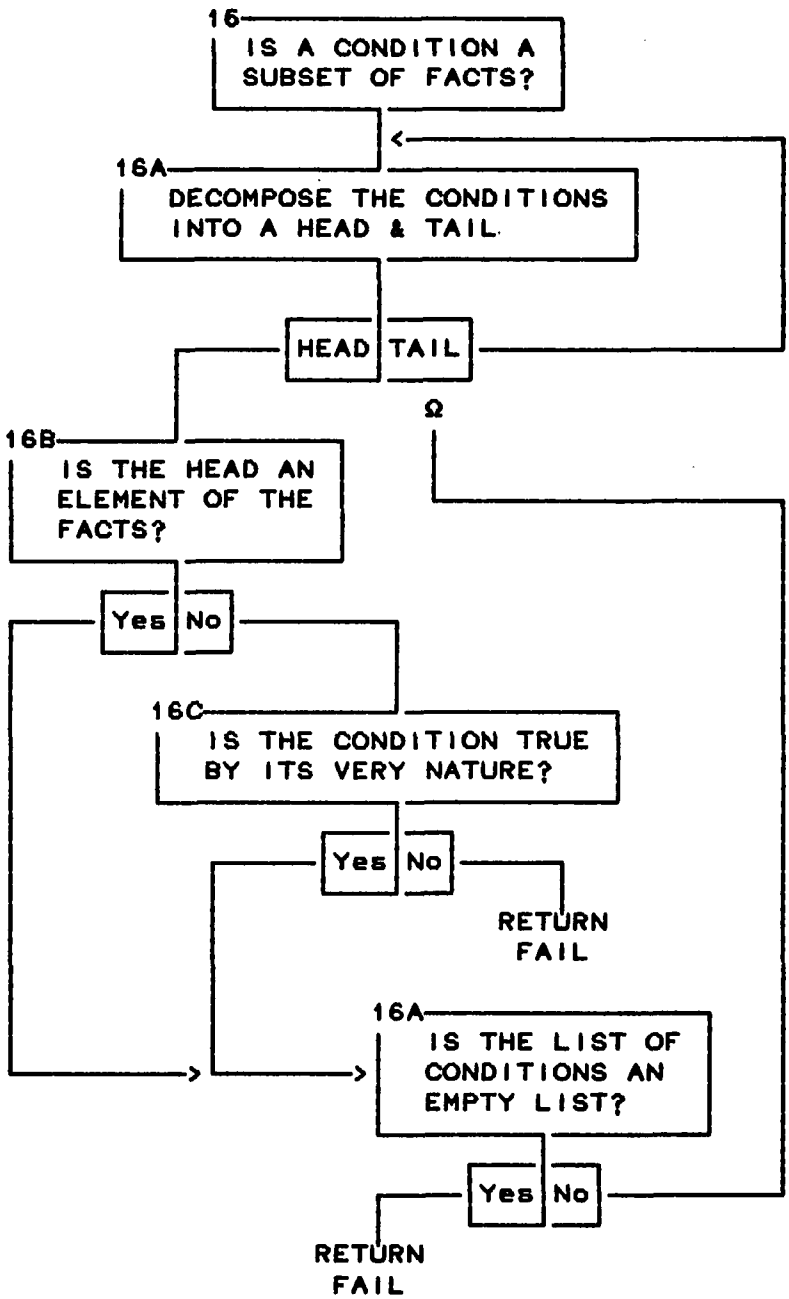


Fig. 3.29. Rule 16 Flowchart

RULE 15A

```
isa_element_of(Term,
               Head<&>Tail) :-
    isa_element_of(Term,
                   Head).
```

RULE 15B

```
isa_element_of(Term,
               Head<&>Tail) :-
    !,
    isa_element_of(Term,
                   Tail).
```

RULE 15C

```
isa_element_of(Term,
               Term).
```

RULE 16A

```
isa_subset_of(Subcond_1<&>Subcond_2,
              Facts) :-
    !,
    isa_subset_of(Subcond_1,
                  Facts),
    isa_subset_of(Subcond_2,
                  Facts).
```

RULE 16B

```
isa_subset_of(Condition,
              Facts) :-
    isa_element_of(Condition,
                   Facts).
```

RULE 16C

```
isa_subset_of(Condition,
              Facts) :-
    Condition.
```

Fig. 3.30. Rules 15 and 16 in Prolog

Rules 15 and 16

Figures 3.28, 3.29 and 3.30 contains rules that determine membership. Rule 15 handles singular terms only, while Rule 16 can handle a list. Examples of queries are,

```
isa_element_of(a, a).
     --> TRUE
isa_element_of(a, b).
     --> FALSE
isa_element_of(b, a <&> b <&> c).
     --> TRUE
isa_element_of(b <&> c, a <&> b <&> c <&> d).
     --> FALSE
isa_subset_of(a, a).
     --> TRUE
isa_subset_of(a, b).
     --> FALSE
isa_subset_of(b, a <&> b <&> c).
     --> TRUE
isa_subset_of(b <&> c, a <&> b <&> c <&> d).
     --> TRUE
```

Rules 17 and 18

Rule 17, Figures 3.31 and 3.33, check for intersection by looking for common terms.

```
intersect(a, a).
     --> TRUE
```

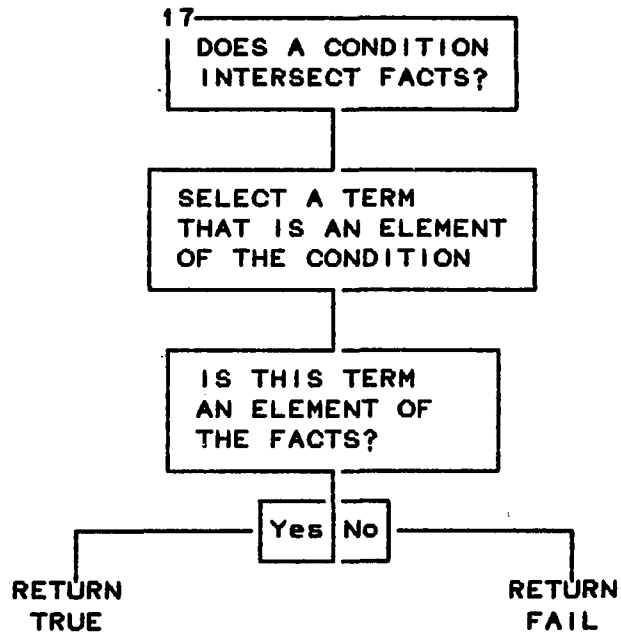


Fig. 3.31. Rule 17 Flowchart

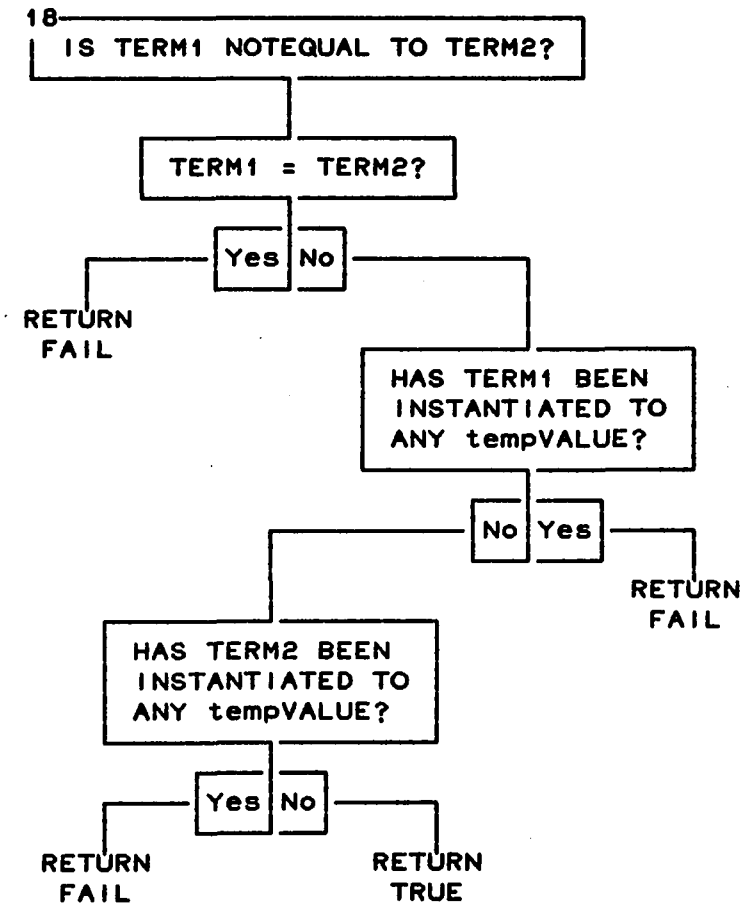


Fig. 3.32. Rule 18 Flowchart

RULE 17

```
intersect(Condition,
          Facts) :-
    isa_element_of(Term,
                   Condition),
    isa_element_of(Term,
                   Facts).
```

RULE 18

```
notequal(X, Y) :-
    not (X = Y),
    not (X = tempVALUE(_)),
    not (Y = tempVALUE(_)).
```

Fig. 3.33. Rules 17 and 18 in Prolog

```
intersect(a, b).
     --> FALSE
intersect(b, a <&> b <&> c).
     --> TRUE
intersect(b <&> c, a <&> b <&> c <&> d).
     --> TRUE
intersect(a <&> b, b <&> a).
     --> TRUE
```

The reasoning for Rule 18, Figures 3.32 and 3.33, is less clear. To paraphrase, two terms are declared not equal if they fail the weak equality test '=', or if either of the terms has been instantiated to a temporary value 'tempVALUE(_)'. The reason for the first test is obvious. The reason for the second is that this rule is sometimes necessary in database to describe impossible conditions in the world state. If the database contained the clause,

```
impossible(on(Block1, Block2) <&>
           on(Block1, Block3) <&>
           notequal(Block2, Block3)).
```

then a specific call to this condition might be,

```
impossible(on(Block1, tempVALUE(0)) <&>
           on(Block1, b) <&>
           notequal(tempVALUE(0), b)).
```

In this case,

```
notequal(tempVALUE(0), b))
```

would return,

FALSE

and thus the 'impossible' clause would also return FALSE. This does not mean that it could not be TRUE, just that it could not be proved one way or the other. This is because some variable has become instantiated to 'tempVALUE(0)' and it is possible that this variable will ultimately become instantiated to the block 'b', thus making the 'impossible' clause FALSE for this instantiation. However, the variable might also become instantiated to block 'c', in which case the 'impossible' clause would be TRUE, that is, a block cannot be upon 'b' and 'c' at the same time if 'b' and 'c' are not equal.

Examples of queries would be,

```
notequal(a, a).
     --> FALSE
notequal(a, b).
     --> TRUE
notequal(tempVALUE(0), b).
     --> FALSE
notequal(tempVALUE(0), tempVALUE(0)).
     --> FALSE
```

Rule 19

Rule 19, Figure 3.34 and the upper half of Figure 3.35, is the last rule to fire in the program since it prints the final solution. If there was only a single action comprising the solution, Rule 19B would fire and the corresponding action would be printed, but if the solution consisted of a list of actions, then Rule 19A would be the one first to fire. The list is destructured in 19B and recursive calls are made until 19B can be called as the exiting condition. Rule 19C should never fire and is included 'just in case' something unusual should occur.

Actually all that would be normally needed to print the final plan would be the lower half of Figure 3.35, where 'instantiate_variables' has been deleted from 19A, and Rule 19C deleted completely.

The reason for the inclusion of 'instantiate_variables' is to prevent any uninstantiated variables from creating infinite loops. For example,

```
printa_plan(X).
```

would match Rule 19A to give,

```
printa_plan(_01<+>_02) :-
    !,
    printa_plan(_01),
    write(_02),
    nl.
```

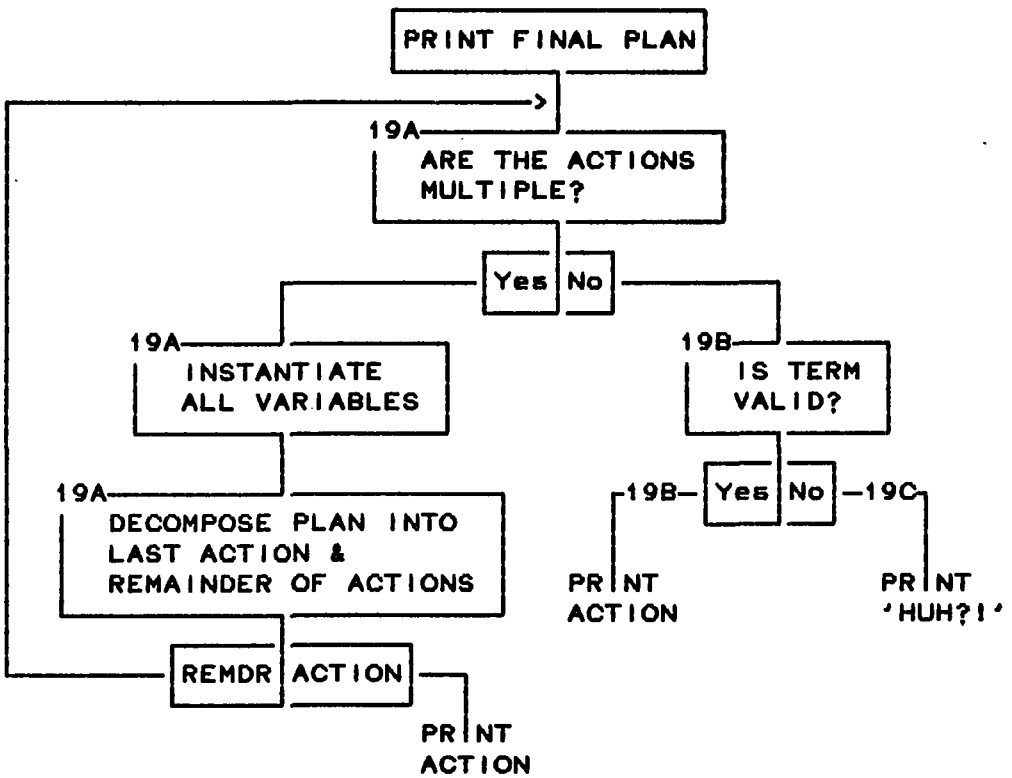


Fig. 3.34. Rule 19 Flowchart

WITH INSTANTIATION OF VARIABLES

RULE 19A

```
printa_plan(Actions <+> Last_action) :-
    !,
    instantiate_variables(Actions <+> Last_action,
                          0,
                          Lastnum),
    printa_plan(Actions),
    write(Last_action),
    nl.
```

RULE 19B

```
printa_plan(Action) :-
    write(Action),
    nl.
```

RULE 19C

```
printa_plan(_) :-
    write('DON'T KNOW HOW YOU GOT TO THIS POINT!!!'),
    nl.
```

WITHOUT INSTANTIATION OF VARIABLES

RULE 19A

```
printa_plan(Actions <+> Last_action) :-
    !,
    printa_plan(Actions),
    write(Last_action),
    nl.
```

RULE 19B

```
printa_plan(Action) :-
    write(Action),
    nl.
```

Fig. 3.35. Rule 19 in Prolog

But the second clause in the body of the rule above would recursively call

printa_plan(_01).

which, since it is an uninstantiated variable, would be decomposed in the same fashion as the original call

printa_plan(X).

and thus begin an endless loop. The call to 'instantiate_variables' insures that all variables are instantiated, thus preventing any possible looping.

One might wonder about the reason for decomposing the list of actions from the right end rather than the left,

printa_plan(List_of_actions<+>Single_action).

since one wants to print the list in a left-to-right order because the list of actions is saved with the first action at the left end of the list and the last action at the right. Why not decompose in the following manner,

printa_plan(Single_action<+>List_of_actions).

and then print the single action followed by a recursive call on the remainder of the list? For example, an actual list could be,

[move(b,a,table) <+> move(a,table,b)]

Move b from a onto the table

<and then>

move a from the table onto b.

The problem centers around the definition of the operator <+> which concatenates the goals,

```
:- (800, yfx, <+>).
```

The 'f' indicates that <+> is an infix operator and the arguments 'y' and 'x' indicate their precedence with respect to that operator. This is important in determining if, for example, subtraction without parenthesis, a-b-c, means (a-b)-c or a-(b-c). The case of 'yfx' means the former and so in the case of the operator <+> it requires destructuring the goal list from right-to-left. Otherwise, a call such as,

```
printa_plan(a<+>b<+>c<+>d).
```

would be output as,

```
a<+>b<+>c
d
```

rather than,

```
a
b
c
d
```

The solution would be printed in a reverse order if the 'write' predicate in Rule 19A preceded 'printa_plan', which is the third clause of the body. Placing the 'write' statement after the recursive call, however, insures the solution is printed in the proper manner because this clause is never reached until the clause preceding it has been completely satisfied. To satisfy the preceding clause, all remaining actions must be printed, thus the last action

becomes the last to be printed and the action preceding it becomes the next to last to be printed, and so on recursively.

CHAPTER 4

DECLARATIVE LANGUAGES

To a large extent, these are problem oriented languages that currently are not as computationally efficient as imperative languages. A major goal of the next generation of hardware is to provide this efficiency. Declarative language is a general term that embraces both relational languages (Prolog) and functional languages (LISP) as opposed to imperative languages (Pascal, Fortran, or C). Imperative languages specify procedures for solving problems, whereas declarative languages specify the kind of solution being sought.

Functional languages consist of functions and arguments to those functions that uniquely identify the output. This is a style of programming that uses function application as the only control structure. Rather than conditional statements, conditional expressions are used to give alternative results. Parameter binding to an argument is used instead of an assignment statement and patterns of nested function invocations are used rather than explicit sequencing or looping of control flow (Ralston 1983, p. 647).

Relational languages specify output in terms of some property and an argument. Unlike functional languages, relational languages do not require a unique output for each predicate/argument pair (Eisenbach and Sadler 1985, p. 186).

LISP

LISP, the oldest declarative language, was designed by John McCarthy at M.I.T. in 1960. The pure LISP view of computation as simple function evaluation was too restrictive for most applications, so imperative features have been incorporated into different versions of the language, though still allowing for a large enough subset of the LISP program to handle declarative programming.

The only standard data structure is the list, which need not contain homogeneous elements because LISP is an untyped language. Since programs are lists as well, a list can be executed and return a value or it can be used as an argument for another program. The lambda expression, by means of which higher order functions are implemented, enables the definition and manipulation of functions as data objects. Every LISP construct computes a value, and recursion is the only control mechanism (Eisenbach and Sadler 1985, p. 190).

Data objects are treated as first-class items, by which is meant,

1. any object may be passed as a parameter to a function
2. any object may be returned as a value from a function
3. any object may be assigned as a value to an identifier
4. any two objects may be tested for equality.

This is in contrast to most traditional languages which prohibit, for instance, vectors to be returned as values (Allen 1985, p. 28).

Most languages are built on the von Neumann model of computation, by which is meant that the process of computation is thought of as a collection of cells, each capable of containing one piece of information. The contents of various cells are selected, operations performed, and results placed back in those cells. A set of statements to collect and parameterize computations is called a procedure, thus the term procedural languages.

An alternative approach is to view computation as producing a value, an approach in which one is more concerned with a description of a computation rather than a specific prescription of how to perform that computation. The constructs that build new computational units are called functions since they are based on the mathematical notion of function, thus the term functional languages (Allen 1985, p. 29). In LISP, not only are these functions allowed to

become objects of the language, but data can be converted into programs and programs into data. Since programs look like data, they can be manipulated by LISP's list operations and this gives the dynamic ability to modify old programs into new ones.

LISP is based on a mathematical formalism called lambda calculus invented by Alonzo Church, but there were also formal systems invented around the concept of relation. A mathematical relation is a generalization of the notion of function and for this reason, programming languages based on relations are potentially more general than those based on functional concepts. This is why it has been said that pure LISP can be viewed as a specialization of Prolog in which procedures are restricted to simple functions and data structures are restricted to lists (Warren and Pereira 1977, p. 109).

First-order predicate calculus was the formal system that supports relational or logical languages. Functional languages represent a compromise, being neither as general as relational languages nor as machine-oriented as the traditional ones, but there has been 10 to 15 years more experience with functional languages than with the relational ones (Allen 1985, p. 32). A comparison of the different approaches follows,

	PROCEDURAL	FUNCTIONAL	RELATIONAL
MODULE	procedure	function	clause
COMPONENT	statement	expression	relation
NAMING	assignment	lambda binding	unification
RESULT	side-effect	value	constraint

Prolog

Rather than building a program out of a collection of functions as in LISP, a Prolog program consists of a sequence of relations and rules. Yet there are many parallels with LISP. Both are interactive languages designed primarily for symbolic manipulation of data and both are founded on formal mathematical systems, Church's lambda calculus for LISP and first-order predicate calculus for Prolog. In their pure versions, neither explicitly have pointers nor assignment statements.

The original objective in developing Prolog was to integrate Robinson's resolution principle into a programming language. The fact that Robinson suggested only one rule of inference, instead of multiple rules as proposed by logicians, facilitated the design of a programming language that would simulate the thinking process by making deductions from information given in logical formulas. The close link with logic, which was originally an asset, became an increasing obstacle when trying to implement Prolog, with

the result that more practical constraints had to be incorporated (Colmerauer 1985, p. 1296).

Horn Clauses

Prolog's first-order predicate calculus syntax is written in a restricted clause form called Horn clauses, which are clauses that contain at most one positive literal. A literal prefixed with a not symbol is called a negative literal, otherwise it is a positive literal. Stated another way, Horn clauses are those clauses containing at most one conclusion. A Prolog rule is written in the form,

$$A \text{ <-- } B_1, B_2, \ldots, B_n.$$

in which A is the head and the B_i's are the body. A very important distinction is that,

1. A fact is a special case of a rule when n = 0, that is, a clause with an empty body. It is a headed clause and is also called a unit clause.
2. A question has no head A, that is it has only the body and is called a headless clause.
3. A rule has both a head and a body and is of the headed clause category.

Any problem which can be expressed in logic can be re-expressed by means of Horn clauses (Kowalski 1979, pp. 16-7). Any solvable problem that can be expressed in Horn clauses, named after the logician Alfred Horn who first investigated them, can be expressed in such a way that there

is one headless clause (goal or question) and all the rest are headed (hypotheses) (Clocksin and Mellish 1984, pp. 249-50). At least one headless clause must be present for a problem to be solvable because the resolution of any two Horn clauses that are headed is itself another headed Horn clause. The empty clause, which is a necessary part of the resolution proof method, is not headed, thus it could not be derived if all the clauses were headed. This is why one headless clause is needed.

1. Facts: parent(mary, john).
   female(mary).
2. Question: :- mother(X, john).
3. Rule: mother(X, Y) :- parent(X, Y),
   female(X).

Another important point to note is that in the Prolog syntax the head A is the positive literal and the body B_i's are the negative literals. Thus, when it was stated above that a Horn clause can contain at most one positive literal, this meant that a rule could not contain a conjunction of goals in its head (conclusion), yet a conjunction of goals (questions) is allowed as they are negative literals in the body.

The Prolog system is based on a resolution theorem prover for Horn clauses using what is called SLD-resolution. This stands for Linear resolution with Selection function of Definite clauses (Lloyd 1984, p. 35). The terminology LUSH

resolution has also been used. Standard systems always select the leftmost atom in a goal together with a depth-first search rule. The search rule is implemented by means of a stack of goals. An instance of the goal stack represents the branch currently being investigated and computation becomes an interleaved sequence of pushes and pops on this stack. A push occurs when the selected atom in the goal at the top of the stack is unified with the head of a program clause. The unified head is pushed onto the stack. A pop occurs when there are no more program clauses with a head to match the selected atom in the goal at the top of the stack. This goal is then popped and the next choice of matching clause for the new top of the stack is investigated (Lloyd 1984, pp. 51-2).

The depth-first search rule uses the ordering of program clauses as the fixed order in which they will be tried. There is room here for further investigation.

Semantics

Prolog can be viewed from two distinct perspectives. Declarative semantics, which Prolog inherits from logic, defines (recursively) the set of terms which are asserted to be true according to a program. It is this aspect which promotes clear reading and clear programming because the program can be broken into small, independent units (clauses).

The procedural semantics describes the way a goal is executed. The ordering of clauses in a program, and goals in a clause constitute control information for the procedural semantics, although this is irrelevant as far as declarative semantics is concerned. That is, the programmer can control the order in which the computer solves subproblems by controlling the order in which the subproblems are written (Warren and Pereira 1977, p. 110).

Backtracking

One of the features of Prolog that is sure to give newcomers a lot of trouble is backtracking, by means of which Prolog dynamically explores all paths through a program for all alternative answers to a question. Consider the Prolog program in Figure 4.1 in which one asks the question, what sentence

```
sentence(S).
```

can be made from a simple database consisting of,

```
1.  'zeigler' is a noun        -->  noun(zeigler).
2.  'likes' is a verb          -->  verb(likes).
3.  'pseudo' is an adjective   -->  adj(pseudo).
4.  'code' is a noun           -->  noun(code).
```

Backtracking provides all the answers in Figure 4.2! Imagine what could happen in a complicated program. This was a reason for the inclusion of the 'cut' in Prolog, to partially control this backtracking.

```
/*  PROLOG PROGRAM for a CONTEXT FREE GRAMMAR  */

sentence(S) :- noun_phrase(NP),
               verb_phrase(VP),
               append(NP, VP, S).

noun_phrase(NP) :- noun(NP).
noun_phrase(NP) :- adj(X),
                   noun(Y),
                   append(X, Y, NP).

verb_phrase(VP) :- verb(X),
                   noun_phrase(NP),
                   append(X, NP, VP).



/*  DATABASE */

noun([zeigler]).
verb([likes]).
adj([pseudo]).
noun([code]).



/*  APPEND  */

append([], X, X).
append([W|X], Y, [W|Z]) :- append(X, Y, Z).
```

Fig. 4.1. Context Free Grammar

ANSWERS TO PROLOG QUERY (with list brackets removed)

```
sentence(S).

S = zeigler likes zeigler.
S = zeigler likes code.
S = zeigler likes pseudo zeigler.
S = zeigler likes pseudo code.  <--  Answer being sought!
S = code likes zeigler.
S = code likes code.
S = code likes pseudo zeigler.
S = code likes pseudo code.
S = pseudo zeigler likes zeigler.
S = pseudo zeigler likes code.
S = pseudo zeigler likes pseudo zeigler.
S = pseudo zeigler likes pseudo code.
S = pseudo code likes zeigler.
S = pseudo code likes code.
S = pseudo code lies pseudo zeigler.
S = pseudo code likes pseudo code.
```

Fig. 4.2. Grammar Query

Pro's and Con's

In procedural languages, unless altered by loops, blocks of statements execute in the order in which they are written. Statements do not ever execute backwards. In Prolog, backtracking causes a backward flow of subgoal invocation. Execution may proceed right-to-left, left-to-right, up, or down on the page across the 'and' operator which is a 'comma' (Kenig 1986, p. 60).

Since Prolog is weakly typed, it becomes rule-sensitive to incorrect arguments and inputs and this, in combination with it being highly recursive, can make it frequently very difficult to debug.

Because Prolog allows a program to be formulated in smaller units, each having a natural declarative reading, it becomes easier to read than the typical nesting of LISP brackets.

Conventional programs mix the logic of the information used in solving a problem together with the control over the manner in which the information is used. One of the appealing features of Prolog is that it separates control information from logic information in programming. Most languages make it fairly difficult to write programs that automatically take advantage of operations and instructions that can be executed in parallel, but Prolog offers many opportunities for this parallelization (Cohen 1985, p. 1322).

Logic Programming

Logic programming is programming by description. The programmer describes the application area and lets the program choose the specific operations. Logic programs are easier to create and because of their advantages and range of applicability, logic programming may evolve into the dominant programming methodology of the twenty-first century (Genesereth and Ginsberg 1985, p. 933).

Because the inference procedure used by a logic program is independent of the knowledge base, program development amounts to finding a suitable description of the application area as represented by that knowledge base. Advantages of this are incremental development, ease of modification, and clear explanation.

There are two reasons why logic programming has special importance for Artificial Intelligence. It offers an alternative to LISP as a language for symbol manipulation and it is very useful for knowledge representation. Predicate logic is considered by many to be a natural and powerful representation language. More narrowly, logic programming has been associated with the style of programming introduced by Kowalski and eventually incorporated into Prolog (Cohen and Feigenbaum 1982, pp. 120-23).

LISP versus Prolog

To set one language against the other often tells more about the prejudices of the person doing the comparisons than about the languages themselves. A more proper comparison would be between functional and relational languages in general, or functional versus logic programming. Both have their pro's and con's, but that is true of every other language as well. Proponents of one have made derogatory comments about the other that simply were not true. Sometimes out of ignorance, sometimes out of prejudice.

LISP has the advantage of 10 to 15 years more development time and, consequently, has better environments available. Prolog is gaining popularity fast and trying to make up lost ground.

Arguments made by proponents usually center around,

1. mathematical foundation
2. execution efficiency
3. programming styles
4. readability and clarity
5. role of side effects
6. special features such as unification and backtracking (DeGroot and Lindstrom 1986, p. 242).

Pure Prolog lacks logical negation and functional data objects which are supported in LISP. Prolog can be made to run as fast as equivalent LISP programs by using the

'cut' to control backtracking and by proper ordering of the clauses, however, this is not pure Prolog and without using these features Prolog can also be made to run very, very slow.

At first sight, Prolog is definitely easier to read than LISP. However, in Prolog when one wishes to pass a value computed by one literal to the following literals, dummy variables must be introduced where they would not in LISP. The extensive use of 'cut' in backtracking can obfuscate the program flow. The unification of Prolog is more powerful than the simple binding of LISP, yet LISPer's can find counter-examples that demonstrate greater efficiency. In LISP one can record information using the dynamic environment and the function 'setq' to set free variables, while in Prolog it is impossible because variables are defined and used only inside one clause.

In some applications like natural language processing and relational database management, Prolog is better than LISP. LISP appears to be more general and applicable to a larger variety of applications at the present moment. One view is that LISP is the C of artificial intelligence where one has more control over computation, whereas Prolog is the Pascal of artificial intelligence in which programs seem more pure and elegant (DeGroot and Lindstrom 1986, pp. 240-51). The solution would seem to be a combination of the desirable features of

both, and this is what has been attempted by several researchers.

LOGLISP

LOGLISP is J. A. Robinson's, of resolution fame, solution in which LISP is augmented with a package of logic functions (Robinson and Sibert 1982a, pp. 399-419; Robinson 1982) to produce an implementation of logic programming within LISP. Its goal was to combine in one language, the facilities of both Prolog and LISP. Although for most people, logic programming seems to be equated with Prolog, Prolog is not, and does not claim to be, the definitive logic programming formalism. Prolog incorporates features not suggested in Kowalski's original plan, such as 'cut' and 'negation as failure', that were not to Robinson's liking. As a devoted user of LISP, Robinson and his associates sought to create within LISP a faithful implementation of Kowalski's logic programming ideas (Robinson and Sibert 1982a, p. 399) and is this way provide themselves and other LISP users with the equivalent of Prolog without the need to venture outside the LISP programming environment (Clark and Tarnlund 1982, p. 299).

An extension was added to LISP consisting of primitives designed to support unification, LUSH resolution, and so on. Logic was to be separate from control, thus no use of the 'cut' and no preferred ordering of assertions.

Prolog generates the LUSH resolution proof tree one branch at a time by backtracking, whereas LOGLISP attempts to generate all branches in quasi-parallel so that heuristic control techniques could be used to determine which branch to develop at each step. It was also an objective to design a system which would better lend itself to future multiprocessing (Robinson and Sibert 1982a, p. 400).

The LISP environment is made available to the user as a convenient host facility in which LISP functions for editing, displaying, monitoring, debugging, inputting and outputting of assertions, queries and deductions could be invoked interactively or under program control. A query is just the submission of an appropriate LISP function call which can be done either interactively from the terminal or internally from within an applications program. The answer is a LISP data object that can be either displayed on the terminal as a stream or returned to an internal call as its result and then subjected to further manipulation (Robinson, Sibert and Greene 1984a, p. 0-4).

LOGLISP has been criticized as being too complicated because in order to interface the logical and applicative components of the language there was introduced new syntax, special cases, and many new functions of questionable need (DeGroot and Lindstrom 1986, p. 240). Robinson found it convenient to be able to invoke logic from LISP and vice versa and felt the system could be speeded up by a factor of

ten and thus put it on a par with some of the Prolog implementations (Clark and Tarnlund 1982, p. 313).

SUPER

An even more ambitious project is Robinson's SUPER language (Syracuse University Parallel Expression Reduction) (Robinson 1984; Robinson, Sibert and Greene 1984a) whose objective is to design and implement an ultra-high-level programming system as a successor to LOGLISP. SUPER goes even beyond that in an attempt to be a programming language which combines certain features of lambda calculus, predicate calculus, and set theory, together with a collection of primitive constants sufficient to provide the main features of LISP and logic programming. It does not seek to provide the specifics of Prolog, but the generic features of a system based on Horn clause resolution theorem proving.

The task actually is broken into two parts, language design and machine design, because the SUPER language will run on the SUPER machine employing a multi-processor architecture in terms of Fifth Generation technology (Robinson 1984b, pp. 1-3).

Even Robinson felt that LOGLISP was a rather awkward mixture of LISP's denotational semantics and Prolog's unification based resolution scheme involving the reduction-like semantics of predicate calculus in Herbrand's version.

The main lesson learned was that there should be only one semantics (Robinson 1984b, pp. 1-2). In a phone conversation in February 1986 with Dr. Kevin J. Greene, who is part of the research group on the project, he stated that a primary goal was to provide a smooth transition from the functional to the logical so that there would indeed be only one semantics, not a concatenation of two.

APPLOG

APPLOG (APPlicative LOGic) is another attempt to combine the features of LISP and Prolog. It is embedded within Prolog, yet is a functional language and supports lambda and nlambda (passing unevaluated arguments) function definitions and also one-to-one, one-to-many and mixed binding mechanisms in a fashion similar to INTERLISP. APPLOG offers the following advantages over LISP,

1. pattern directed invocation
2. call by reference
3. interface to Prolog
4. functions as operators
5. backtracking
6. generators (DeGroot and Lindstrom 1986, p. 239).

The LISP and Prolog environments are 'unified' to provide an easy way to express ideas in either an applicative or a logical style. Users can write programs in

their preferred language and yet have access to the power and virtues of the other.

APPLOG provides features such as function application, logical inference, a relaional database query language, and generators (DeGroot and Lindstrom 1986, p. 240).

Prolog code of the APPLOG interpreter can be found on pp. 262-75 of the following reference (DeGroot and Lindstrom 1986).

TABLOG

TABLOG (Tableau Logic Programming Language) is a language based on first-order predicate logic with equality that combines function and logic programming. It, again, attempts to incorporate the advantages of LISP and Prolog. A program in TABLOG is a list of formulas in a first-order logic that is more general and expressive than Prolog's Horn clauses. Prolog programs must be relational, but TABLOG programs may define either relations or functions. LISP programs yield results of a computation by returning a single output value, TABLOG programs can be relations and can produce several results simultaneously through their arguments (Malachi, Manna and Waldinger 1984, p. 1).

The Manna-Waldinger deductive-tableau proof system is employed as an interpeter for TABLOG in a fashion similar to Prolog's use of the resolution proof system. This

deductive-tableau consists of a set of rows, each containing either an assertion or a goal, both being first-order logic formulas. A proof is constructed by adding new goals to the tableau, using deduction rules, in such a way that the final tableau is semantically equivalent to the original. In contrast to standard resolution proof systems, the negation of the given theorem does not have to be manipulated until the null is found nor do sentences have to be converted to clausal form (DeGroot and Lindstrom 1986, p. 378).

Rather than invoking Prolog-like features from within LISP (LOGLISP) or LISP-like features from within Prolog (APPLOG), the two are claimed to coexist peacefully within the same framework and are processed by the same deductive engine in TABLOG (DeGroot and Lindstrom 1986, p. 389).

Rather than patching the shortcomings of Prolog, TABLOG offers a more powerful deductive scheme. It offers a natural extension to parallel computation and support of concurrent programs is being pursued. In summary, its developers feel it to offer a significant advance over Prolog. If Prolog is to become the Fortran of the future, they hope TABLOG will become the Pascal.

MRS

MRS (Meta-level Representation System) is a language out of Standord that is another attempt to incorporate logic

programming principles into LISP. As stated in the MRS manual, the traditional process goes something like,

1. Identify the problem
2. Assemble what you know about it.
3. Decide on data structures to correspond to things in the problem.
4. Figure how to process those structures to produce the desired answer.
5. Encode the process step by step in your favorite language.
6. Get the computer to execute the process.

Using MRS the user does the following,

1. Identify the problem.
2. Assemble what you know about it.
3. Encode what you know about it in MRS.
4. Get MRS to figure out the solution.

(Russell 1985, p. 2).

These goals sound identical to those claimed for Prolog, but since MRS is built on top of LISP, there is always access from within MRS to the full power of LISP.

QLOG

Prolog lacked the good programming environments offered by LISP, so instead of development an environment for Prolog, by embedding Prolog in LISP it was sought to

transfer the LISP programming environment to Prolog (Clark and Tarnlund 1982, p. 316).

This was not as ambitious a project as LOGLISP, but its drawbacks were similar.

LMA/ITP

The subroutine package LMA (Logic Machine Architecture) and automated reasoning system (Interactive Theorem Prover) were developed at the Argonne National Laboratory which has been conducting research in the area of automated reasoning. LMA is a collection of subroutines which manipulate a database of logical formulas and perform various inference operations upon it. ITP is an interactive program built on top of LMA and allows access to all the LMA procedures and is a general purpose reasoning system suitable for experimentation and demonstration.

These packages consist of about 70,000 lines of Pascal source code in about 50 files. They are in the public domain and include all source and object code, example problems, and complete documentation.

LMA is a layered architecture for the creation of logic inference engines. ITP is a clause-based reasoning system supporting a wide variety of techniques used in Argonne's automated deduction research project. A Prolog-like subsystem is coupled to the theorem proving component. To use ITP one need not be familiar with LMA because its

main function is to provide a user interface (Lusk and Overbeek 1984a; Lusk and Overbeek 1984b).

Logic in General

Since the time of the ancient Greeks, man has struggled with the nature of reasoning and knowledge. This effort, which was formalized in the 19th century and expanded in the 20th, has developed into the philosophical and mathematical study called logic. This formal treatment of knowledge and thought has been applied to the development of computer programs that can emultate human reasoning.

The computational approach to logical reasoning is termed computational logic and is divided into two main areas, propositional logic and predicate logic. Computational logic is an important artificial intelligence area because it is a key ingredient in the development of computer programs able to deduce facts that are not explicitly represented but that are implied by other facts which are represented.

The direct application of these methods of deduction to real-world problems has been complicated because both the data and the expertise are often uncertain. The development of expert systems has led away from the pursuit of logical completeness and towards the development of effective heuristics to exploit the fallible judgmental knowledge that human experts bring to particular problems.

General Logic

Two branches of logic are axioms and rules of inference. The axioms of a system are the relations and implications that can be formalized. Rules of inference are the deductive structures that determine what can be inferred if certain axioms are true. Logic itself deals with the syntax, or form, of statements and employs syntactic manipulation to determine truth, which is logic's most fundamental notion. Any sentence which can be derived from an initial set of axioms, using the inference rules, is called a theorem of the system.

Propositional Logic

For the purposes of Artificial Intelligence, propositional logic is not very useful because each properly formed statement, or proposition, although it can have truth values of True or False, cannot be broken down into its constituent parts. Even these propositions in and of themselves are of little value until they are combined by means of connectives such as,

And

Or

Not

Implies

Predicate Logic

In order to represent knowledge about the world, it is necessary to be able to speak about specific objects, to postulate relationships between these objects, and to generalize these relationships over classes of objects. Predicate logic remedies this situation by extending the notions of propositional logic to include the specific objects, or individuals, that comprise propositions.

The predicate is that part of a proposition that makes an assertion about individuals. It is applied to a specific number of arguments and has a binary value of True or False. The predicate, together with its arguments, is a proposition. Variables are place holders that can be filled by some constant, and can stand for any individual. Variables allow the making of statements that would not be possible in propositional logic. For a proposition containing variables to be true, it must be true for any and all individuals that can be substituted for the variables. Substituting the name of a particular individual is known as 'instantiation' because the individual is a particular 'instance' of the variable.

Quantification is the ability to specify properties of arbitrarily defined sets, that is, whether a specific truth value belongs to all or only some of the members of a set. The two quantifiers are $\forall$, meaning 'for all', and $\exists$, meaning 'there exists'. These quantifiers are used together

with variables to allow the expression of universal and existential statements.

Terms can be constants (names of objects) or variables (marking which part of a formula is quantified). Atomic sentences are formed by applying one predicate to a specific set of arguments. Compound sentences are formed by adding negation to a sentence and/or by joining two or more atomic sentences by connectives (And, Or, Implication, etc.).

WFF's

Well-formed formulas are more complicted expressions that are formed by combinations of connectives, predicates, constants, variables, and quantifiers.

1st Order Logic

First order logic is a specialization of standard predicate calculus to which is added two key ingredients, 'functions' and the predicate 'equals'. Functions, or operators, have a fixed number of arguments as do predicates, but functions are different in that they do not just have the values True or False, they also return objects related to their arguments. Functions can be combined with other functions.

A logic is of the first order if it allows quantification of individuals but not of predicates and functions. The statement 'All predicates have only one argument' could not be expressed in first order logic. In

first order logic, it is impossible to prove a false statement, and any true statement has a proof.

Higher Order Logics

Higher order logic not only allows quantification over individuals, but also over predicates and functions. A second order logic would allow statements about all predicates and functions concerning individuals, that is, the quantification of predicates and functions. Third order logic would allow statements to be made about the 'statements about predicates and functions'.

Summary

A summary of the relationships among the various branches of logic can be seen in the flowchart of Figures 4.3 and 4.4 and in the outline of Figures 4.5 and 4.6. Both flowchart and outline summarizes the above text.

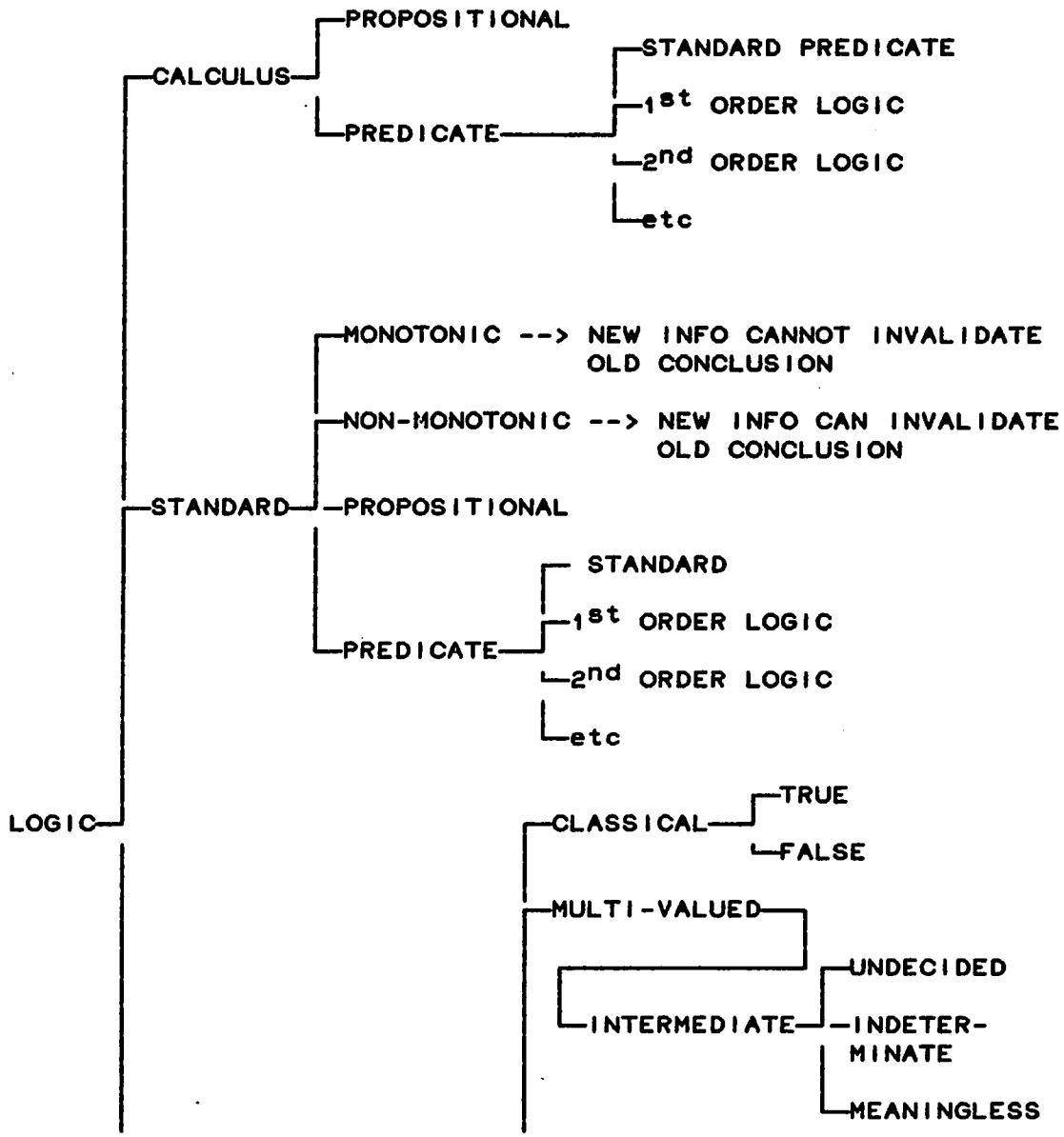


Fig. 4.3. Logic Flowchart---Part 1

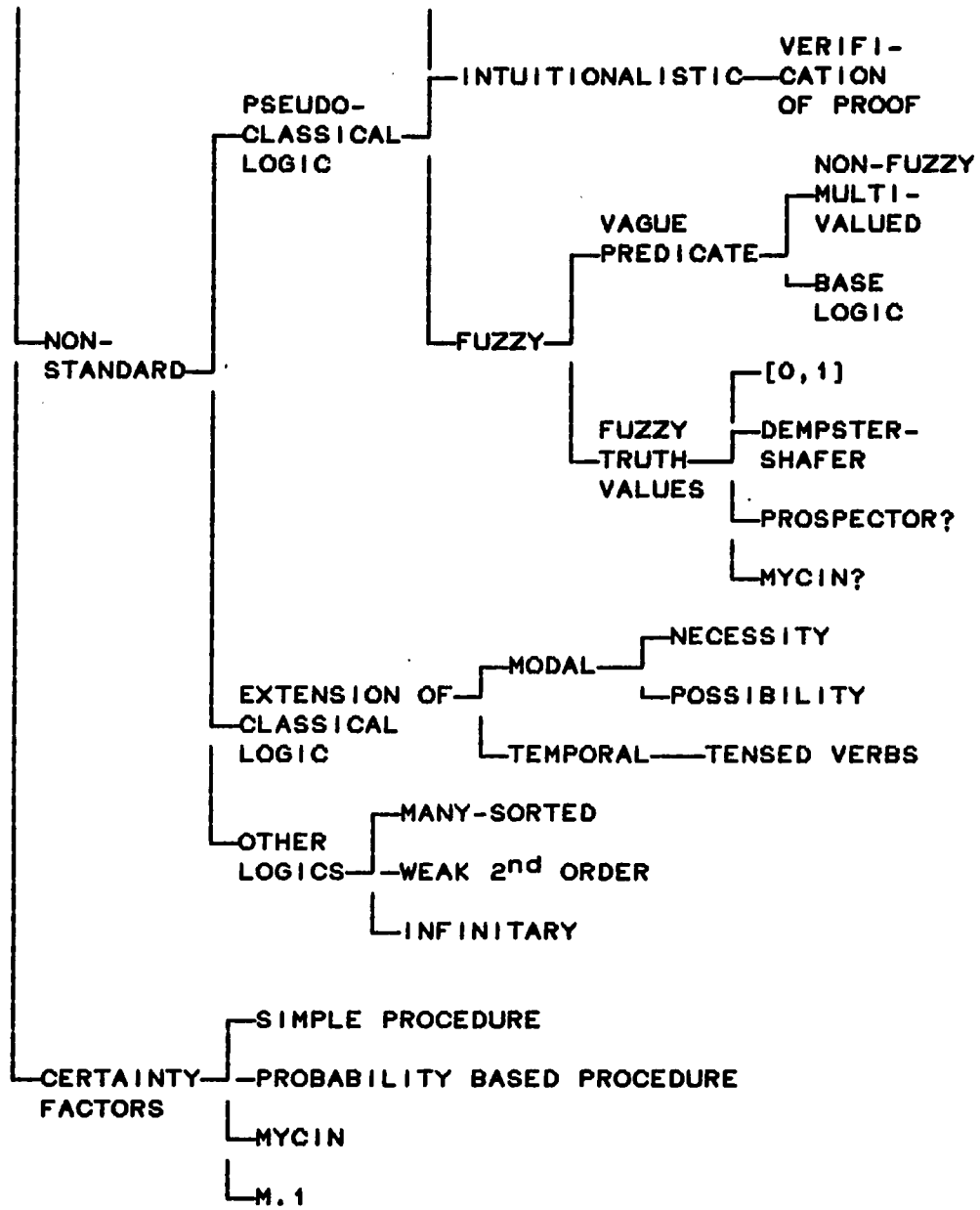


Fig. 4.4. Logic Flowchart---Part 2

GENERAL LOGIC

QUANTIFICATION
    ABILITY TO SPECIFY PROPERTIES OF ABRITRARILY
                                DEFINED SETS
QUANTIFIER

VARIABLE
    PLACE HOLDER

WFF's
    SYNTACTICALLY ALLOWED COMBINATIONS OF CONNECTIVES,
        PREDICATES,CONSTANTS, VARIABLES & QUANTIFIERS

PROPOSITIONAL CALCULUS
    LETTERS STAND FOR PROPOSITIONS OR STATEMENTS
    SPECIAL SYMBOLS FOR RELATIONSHIPS

PREDICATE CALCULUS
    PREDICATE = STATEMENT ABOUT INDIVIDUALS/GROUPS
        SPECIFIC NUMBER OF ARGUMENTS
        TRUE or FALSE
    QUANTIFICATION OF VARIABLES

    ATOMIC SENTENCE = PREDICATE + AGRUMENTS
    COMPOUND SENTENCE = ATOMIC SENTENCES +
                        (NEGATION and/or CONNECTIVES)

Fig. 4.5. Logic Outline---Part 1

```
1st ORDER LOGIC
    DIVISION OF PREDICATE CALCULUS
    2 ADDITIONS TO PREDICATE CALCULUS
         FUNCTIONS
              TRUE
              FALSE
              RETURN OBJECTS RELATED TO THEIR ARGUMENTS
         PREDICATE 'EQUALS'
    DEFINITION:
         PERMITS QUANTIFICATION OVER INDIVIDUALS BUT NOT
                                  OVER PREDICATES & FUNCTIONS
    SOUND --> IMPOSSIBLE TO PROVE FALSE STATEMENT
    COMPLETE --> ANY TRUE STATEMENT HAS A PROOF
    CANNOT QUANTIFY PREDICATES OR FUNCTIONS
         EX:
              "ALL PREDICATES HAVE ONLY ONE ARGUMENT"
              CANNOT BE EXPRESSED IN 1st ORDER LOGIC
                                  ===> HIGHER ORDER LOGIC
    ALLOWS QUANTIFICATION OVER PREDICATES & FUNCTIONS
    2nd ORDER LOGIC
         ALLOWS QUANTIFICATION OF
              VARIABLES
              PREDICATES
              FUNCTIONS
    3rd ORDER LOGIC
         ALLOWS QUANTIFICATION OF STATEMENTS ABOUT
                               PREDICATES & FUNCTIONS
```

Fig. 4.6. Logic Outline---Part 2

CHAPTER 5

CONCLUSIONS

Trying to 'decipher' the code obtained via the engineer from Hughes Aircraft served to impress upon the author more strongly than a thousand pleadings by a professor, the necessity of proper documentation for any and all computer programs. Having not seen Warren's original code, but assuming it to be very similar, if not identical, to that of Appendix C, it would be very helpful for the majority of readers, since they do not operate at the level of researchers like Warren, if published code would include more descriptive functor and component naming. This should be evident by the comparisons listed in Figure 5.1.

A primary value of chapter 3 would be to allow a new 'Prologer' to trace through the workings of a non-trivial Prolog program and, so-to-speak, be able to 'follow the thinking' of someone quite proficient in the language, like Warren. The simplicity of the standard genealogy type examples often found in the first few chapters of many Prolog texts can be quite deceptive. The highly recursive, backtracking nature of Prolog can often make it very difficult to follow the logic of a non-trivial program.

microPROLOG

```
((retrace x1 x2 x3)
    (can x2 x3)
    (retrace1 x1 x2 x3 x4)
    (append x3 x4 x3))
```

PROLOG-86

```
retrace(P, V, P2) :-
    can(V, C),
    retrace1(P, V, C, P1),
    append1(C, P1, P2).
```

PROLOG-2

```
remove_Results_oflast_action(Current_solved_goals,
                             Last_action,
                             Previous_solved_goals) :-
    precond_of(Preconditions,
               Last_action),
    remove_Preconditions_and_Results_oflast_action(
               Current_solved_goals,
               Last_action,
               Preconditions,
               Revised_solved_goals),
    append(Preconditions,
           Revised_solved_goals,
           Previous_solved_goals).
```

Fig. 5.1. Rule Comparisons

PROLOG ON THE PC

Prolog-2 version 1.20 from Expert Systems International was used in the recoding of WARPLAN. This provides a powerful compiler, interpreter, and program development environment but is rather slow on a two drive PC. The minimum system recommended would be a 640K AT with a hard disk because running from two drives can be a bit complicated and is so stated in the manual. With the two drive configuration one cannot have access to both the compiler and the help system at the same time. The compiler is very slow. The ideal system for Prolog-2 would be a 386 machine with DOS 5.0 and a 130 MB hard disk with 20 ms access time!

Prolog-2 has a virtual memory system that allows 256 modules of 1MB each to be built. The editor provided is only adequate for minor modifications and calling a DOS subshell to access another editor takes about 30 seconds. Exiting and rebooting takes even longer. Arity version 4.0 is far superior in this respect as it allows instant exit to the DOS subshell via the command 'shell' and instant re-entry via the command 'exit'. The Prolog-2 language is basically a superset of Edinburgh Prolog and DEC-10 programs will run without modification. There is a good windowing system, but it is more difficult to use than that of Turbo Prolog. The manual is more complete than any of the competition, although it could benefit by a tutorial.

A big annoyance is the copy protection scheme and site licensing requirements for Prolog-2. The copy protection installation program is the only program of any type that the author has seen that detected a difference between an IBM and a Zenith clone. Once the program had been installed via an IBM it ran without flaw on the Zenith, but the requirements of room number, address of the building in which the specified computer is located, serial number of PC, etc. gives one the impression that instead of purchasing a piece of software for the PC, one is instead purchasing a Symbolics 3600 or a Lambda machine.

Companies imposing such restrictions are not in synch with what is happening in the real world. At the University of Arizona, the department in which most of this work on WARPLAN was done has a computer lab with an assortment of PC's, XT's, AT's, Zenith's, and Compaq's in the same room. Students are using these machines at all hours of the day and night on a first come first served basis. Technically, if there were ten computers in the room with only one person working on the particular machine for which Prolog-2 had been registered, then either that person must transfer to another PC or Prolog-2 cannot be used. Nor could Prolog-2 be taken home to be used on one's personal PC. Very inconvenient.

Arity version 4.0 is the top of the line for the PC as of this date. Compilation and execution are fast, there

is good access to other languages, Arity offers many supplementary packages, a gigabyte of virtual memory is possible, and Microsoft Windows is soon to be supported. The interpreter can be configured to use any ASCII file editor and the manual is adequate, although not as extensive as that for Prolog-2. Hashing and B-trees are supported as well as an extension of the 'cut' operator called 'snips' which can be used to eliminate unnecessary backtracking. Running Arity on a two drive PC is a much more reasonable proposition than trying to do the same with Prolog-2.

Running Turbo Prolog version 1.1 at first is fast and fun, but this can soon turn to frustration. The author first saw this program in November 1985 when it was called PC-Prolog and was being developed by the Prolog Development Center in Denmark. It is based on research done at the Technical University of Denmark on compiling techniques for Prolog and is written in C. Borland took over and began making extravagant claims for 'their' product.

It is a good value for the dollar, but is it really Prolog? The release of version 1.1 did little to increase compatibility with the Edinburgh standard. It lacks much of the power that Arity and Prolog-2, among others, have to offer. The ability of programs to modify themselves is severely limited. The data typing is annoying and limits the versatility of the standard data structures.

An attempt was made to convert WARPLAN to Turbo Prolog, but this was soon seen to be more trouble than it was worth, if not impossible. The strong typing of Tubro Prolog restricts all elements of a list to the same type and any way of trying to work around this quickly becomes cumbersome. The same can be said for the declaration of operators which Turbo Prolog does not allow, but which is required in WARPLAN. A way of semi-skirting the issue was developed, but this required considerable nesting of parenthesis and still lacked precedence. The biggest problem was WARPLAN's use of the predicate 'univ' used for structure inspection. This can convert a structure into a list and vice versa. A call to Borland in September 1986 said there was no way around this obstacle and Borland had no plans to add this feature in the future.

'Aside from that', Turbo Prolog is a good value, especially as an introduction to Prolog. It is great for a beginner. The user interface and graphics are unparalleled in the Prolog world. It is creating a lot of new interest. Arity is running ads offering trade-in discounts to Turbo Prolog users wishing to upgrade to 'real' Prolog. The bottom line, though, is that Turbo Prolog lies as close to C as it does to Prolog. If it had been called C-Log, or some such, it would probably have received higher praises.

LISP and Prolog

They both have something useful to contribute to the Artificial Intelligence community and thus should not be set at odds. Researchers are making progress on the AI languages of the future and many of these are a synthesis of the functional and the logical approaches.

For the curious, Appendix H contains an example of a simple Prolog interpreter written in LISP and Appendix I contains one for LISP written in Prolog.

APPENDIX A

LIBRARY SEARCH FOR WARPLAN

THE UNIVERSITY OF ARIZONA
TUCSON, ARIZONA 85721
UNIVERSITY LIBRARY
May 28, 1986

Dear Mr. Boyd:

We regret to inform you that we were unable to obtain a loan or copy of the following material from a US or Canadian library --

AUTHOR: WARREN, D.H.D.
TITLE: WARPLAN -- A SYSTEM FOR GENERATING PLANS
(DGL MEMO 76; Univ. of Edinburgh, 1974)

Please note that we can apply for a copy of Warren's memo from the University of Edinburgh, but if they can supply it would take approximately 3 - 6 months and their charge would be a minimum of $6 - $8.00.

Let us know if the above time and cost is acceptable to you or if you would prefer to cancel the transaction. We will hold your request for thirty days. If we have not heard from you after that time we will assume you are no longer interested.

Interlibrary Loan
Main Library, Rm. #204
621-6438

APPENDIX B

WARPLAN VIA HUGHES

```
((/# Generate a plan ))
((/# removed operator declerations for & and ; ))


((plans x1 x2)
        (unless (consistent x1 true))
        / (PP impossible))
((plans x1 x2)
        (plan x1 true x2 x3)
        (PP x3))


((/# Plan generation ))
((plan X&X1 X2 X3 X4)
        / (solve X X2 X3 X5 X6)
        (plan X1 X5 X6 X4))
((plan X X1 X3 X4)
        (solve X X1 X3 X5 X4 ))


((/# Partial plan ))
((solve x1 x2 x3 x2 x3)
        (always x1))
((solve x1 x2 x3 x4 x3)
        (holds x1 x3)
        (and x1 x2 x4))
((solve x1 x2 x3 x1&x2 x4)
        (add x1 x5)
        (achieve x1 x5 x2 x3 x4))

((/# Subplan ))
((achieved x1 x2 x3 x4 x5;x6)
        (preserves x2 x3)
        (can x2 x7)
        (consistent x7 x3)
        (plan x7 x3 x4 x6)
        (preserves x2 x3))
((achieved x1 x2 x3 x4;x5 x6;x7)
        (preserved x1 x7)
        (retrace x3 x7 x8)
        (achieve x1 x2 x8 x4 x6)
        (preserved x1 x7))
```

```
((/# Fact in subplan ))
((holds x1 x2;x3)
        (add x1 x3))
((holds x1 x2;x3)
        / (preserved x1 x3)
        (holds x1 x2)
        (preserved x1 x3))
((holds x1 x2)
        (given x2 x1))


((/# Action preserving a fact ))
((preserved x1 x2)
        (mkground (& x1 x2) 0 x3)
        (del x1 x2)
        / (FAIL))
((preserved x1 x2))
((preserved x1 (& x2 x3))
        (preserved x2 x1)
        (preserves x1 x3))
((preserved x1 true))


((/# Retrace a goal ))
((retrace x1 x2 x3)
        (can x2 x3)
        (retrace1 x1 x2 x3 x4)
        (append x3 x4 x3))
((retrace1 (& x1 x2) x3 x4 x5)
           (add x6 x3)
           (equiv x1 x6)
           / (retrace1 x2 x3 x4 x5))
((retrace1 (& x1 x2) x3 x4 x5)
        (elem x6 x4)
        (equiv x1 x6)
        / (retrace1 x2 x3 x4 x5))
((retrace1 (& x1 x2) x3 x4 (& x5 x6))
        (retrace1 x2 x3 x4 x6))
((retrace1 true x1 x2 true))


((/# Proves consistantcy ))
((consistent x1 x2)
        (mkground (& x1 x2) 0 x3)
        (imposs x4)
        (unless (unless (intersect (x1 x4))))
        (implied x4 (& x1 x2))
        / (FAIL))
((consistent x1 x2))
```

```
((/# Misc support routines ))
((and x1 x2 x2)
        (elem x3 x2)
        (equiv x1 x3) / )
((and x1 x2 (& x3 x4)))


((/#))
((append x1&x2 x3 (& x4 x5))
        / (append x2 x3 x5))
((append x1 x2 (& x1 x3)))


((/#))
((elem x1 (& x2 x3))
        (elem x1 x2))
((elem x1 (& x2 x3))
        / (elem x1 x3))
((elem x1 x1))


((/#))
((intersect x1 x2)
        (elem x3 x1)
        (elem x3 x2))


((/#))
((implied (& x1 x2) x3)
        / (implied x1 x3)
        (implied x2 x3))
((implied x1 x2)
        (elem x1 x2)
((implied x1 x2)
        (x1))


((/#))
((true))


((/#))
((equal x1 x1))


((/#))
((notequal x1 x2)
        (unless (equal x1 x2))
        (unless (equal x1 (qqq x3)))
        (unless (equal  x2 (qqq x4))))
```

```
((/#))
((equiv x1 x2)
        (unless (nonequiv x1 y1)))


((/#))
((nonequiv x1 x2)
        (mkground (& x1 x2) 0 x3)
        (equal x1 x2)
        / (FAIL))
((nonequiv x1 x2))


((/#))
((univ x3  (x1¦x2))
        (App  x1 x2 x3))


((/#))
((mkground (qqq x1) x1 x2)
        / (SUM x1 1 x2))
((mkground (qqq x1) x2 x2))
((mkground x1 x2 x3)
        (univ x1 (x4 x5))
        (mkgroundlist x5 x2 x3))
((mkgrounglist  (x1¦x2) x3 x4)
        (mkground x1 x3 x5)
        (mkgroundlist x2 x5 x4))
((mkgroundlist () x1 x1)


((/#))
((unless x1) (x1)
        / (FAIL))
((unless x1))
```

APPENDIX C

WARPLAN FROM THE PROLOG86 MANUAL

```
op(650, yfx, =>)!


plan(X & C, P, T, T2) :- !,
                         solve(X,P,T,P1,T1),
                         plan(C,P1,T1,T2).
plan(X,P,T,T1) :- solve(X,P,T,P1,T1).


solve(X,P,T,P,T) :- always(X).
solve(X,P,T,P1,T) :- holds(X,T),and(X,P,P1).
solve(X,P,T,X & P,T1) :- add(X,U),
                         achieve(X,U,P,T,T1).


achieve(X,U,P,T,T1 => U) :-
        preserves(U,P),
        can(U,C),
        consistent(C,P),
        plan(C,P,T,T1),
        preserves(U,P).
achieve(X,U,P,T => V,T1 => V) :-
        preserved(X,V),
        retrace(P,V,P1),
        achieve(X,U,P1,T,T1),
        preserved(X,V).


holds(X,T => V) :- add(X,V).
holds(X,T => V) :- !,
                   preserved(X,V),
                   holds(X,T),
                   preserved(X,V).
holds(X,T) :- given(T,X).


preserved(X,V) :- numbervars(X & V,0,N),
                  del(X,V),
                  !,
                  fail.
preserved(X,V).
```

```
preserves(U,X & C) :- preserved(X,U),
                      preserves(U,C).
preserves(U,true).


retrace(P,V,P2) :- can(V,C),
                   retrace1(P,V,C,P1),
                   append1(C,P1,P2).


retrace1(X & P,V,C,P1) :- add(Y,V),
                          equiv(X,Y),
                          !,
                          retrace1(P,V,C,P1).
retrace1(X & P,V,C,P1) :- elem(Y,C),
                          equiv(X,Y),
                          !,
                          retrace1(P,V,C,P1).
retrace1(X & P,V,C,X & P1) :- retrace1(P,V,C,P1).
retrace1(true,V,C,true).


consistent(C,P) :- numbervars(C & P,0,N),
                   imposs(S),
                   unless(unless(intersect(C,S))),
                   implied(S, C & P),
                   !,
                   fail.
consistent(C,P).


plans(C,T) :- unless(consistent(C,true)),
              !,
              write('Inpossible'),
              nl.
plans(C,T) :- plan(C,true,T,T1),
              !,
              print_plan(T1),
              nl.


and(X,P,P) :- elem(Y,P),
              equiv(X,Y),
              !.
and(X,P,X & P).


append1(X & C,P,X & P1) :- !,
                           append1(C,P,P1).
append1(X,P,X & P).
```

```
elem(X,Y & C) :- elem(X,Y).
elem(X,Y & C) :- !,
                 elem(X,C).
elem(X,X).


intersect(S1,S2) :- elem(X,S1),
                    elem(X,S2).


implied(S1 & S2,C) :- !,
                      implied(S1,C),
                      implied(S2,C).
implied(X,C) :- elem(X,C).
implied(X,C) :- X.


equiv(X,Y) :- unless(nonequiv(X,Y)).


nonequiv(X,Y) :- numbervars(X & Y,0,N),
                 X = Y,
                 !,
                 fail.
                 nonequiv(X,Y).


unless(X) :- X,
             !,
             fail.
unless(X).


print_plan(X => Y) :- print_plan(X),
                      print(Y).
print_plan(X) :- print(X).
```

APPENDIX D

RULE TRANSFORMATIONS

FORMAT

RULE NUMBER
1. Prolog-86 version
2. Renamed variables
3. Prolog-2 version

RULE 1B
```
plans(C, T)
plans(Goal_state, Initial_state)
developa_plan(Initial_state, Goal_state)
```

RULE 2A
```
plan(X & C, P, T, T2)
plan(Goal & Goals,
     Current_solved_goals,
     Current_plan,
     Final_plan)
solvea_plan(Goal <&> Goals,
            Current_solved_goals,
            Current_plan,Final_plan)
```

RULE 3D
```
solve(X, P, T, X & P, T1)
solve(Goal,
      Current_solved_goals,
      Current_plan,
      Goal & Current_solved_goals,
      New_plan)
solvea_goal(Goal,
            Current_solved_goals,
            Goal <&> Current_solved_goals,
            Current_plan,
            New_plan)
```

```
RULE 4B
achieve(X, U, P, T => V, T1 => V)
achieve(Goal,
        Action,
        Current_solved_goals,
        Previous_plan => Last_action,
        Reordered_plan => Last_action)
add_Action_to_Plan(New_action,
                   Goal,
                   Current_solved_goals,
                   Previous_plan<+>Last_action,
                   Reordered_plan<+>Last_action)


RULE 5
retrace(P, V, P2)
retrace(Current_solved_goals,
        Last_action,
        Previous_solved_goals)
remove_Results_oflast_action(Current_solved_goals,
                             Last_action,
                             Previous_solved_goals)


RULE 6C
retrace1(X & P, V, C, X & P1)
retrace1(Front_goal & Remainder_ofsolved_goals,
         Last_action,
         Preconditions,
         Front_goal & Revised_solved_goals)
remove_Preconditions_and_Results_oflast_action(
         Front_goal <&> Remainder_ofsolved_goals,
         Last_action,
         Preconditions,
         Front_goal <&> Revised_solved_goals).


RULE 7B
holds(X, T => V)
holds(Goal,
      Remainder_ofplan => Last_action)
canbe_achieved_by(Goal,
                  Remainder_ofplan <+> Last_action)


RULE 8B
preserves(U, X & C)
preserves(Action,
          Front_goal & Remainder_of_goals)
will_not_delete_listof(Action,
                       Front_goal <&> Remainder_of_goals)
```

```
RULE 9
preserved(X, V)
preserved(Goal, Action)
will_not_delete(Action, Goal)


RULE 10


RULE 11


RULE 12
consistent(C, P)
consistent(Facts, Current_solved_goals)
are_consistent_with(Facts, Current_solved_goals)


RULE 13B
and(X, P, X & P)
and(Goal,
    Current_solved_goals,
    Goal & Current_solved_goals)
combine_together(Goal,
                 Current_solved_goals,
                 Goal <&> Current_solved_goals)


RULE 14
append1(X & C, P, X & P1)
append1(Goal & Goals,
        Current_solved_goals,
        Goal & Combined_solved_goals)
append(Goal <&> Goals,
       Current_solved_goals,
       Goal <&> Combined_solved_goals)


RULE 15A
elem(X, Y & C)
elem(Term, Head & Tail)
isa_element_of(Term, Head<&>Tail)


RULE 16A
implied(S1 & S2, C)
implied(Subcond_1 & Subcond_2, Facts)
isa_subset_of(Subcond_1 <&> Subcond_2, Facts)
```

RULE 17
intersect(S1, S2)
intersect(Condition, Facts)
intersect(Condition, Facts)

RULE 18

RULE 19A
print_plan(X => Y)
print_plan(Actions => Action)
printa_plan(Actions <+> Action)

DATABASE

can(U, C)
can(Action, Facts)
precond_of(Facts, Action)

always(X)
always(Fact)
is_always_true(Fact)

impossible(C)
impossible(Facts)
impossible(Facts)

given(Initial_state, Fact)
exists_in(Fact, Initial_state)

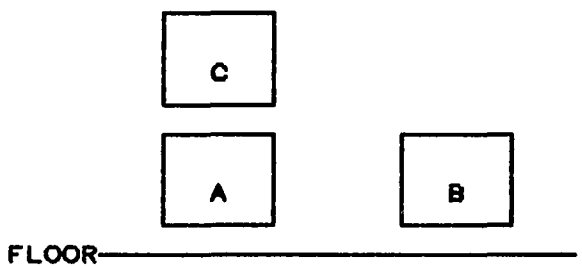
add(Fact, Action)
will_add(Action, Fact)

del(Fact, Action)
will_delete(Action, Fact)

APPENDIX E

THREE BOX EXAMPLE

INITIAL STATE



GOAL

```
developa_plan(state0,
              on(a,b) <&> on(b,c)).
```

PLAN

```
state0
move(c, a, floor)
move(b, floor, c)
move(a, floor, b)
```

WORLD DESCRIPTION

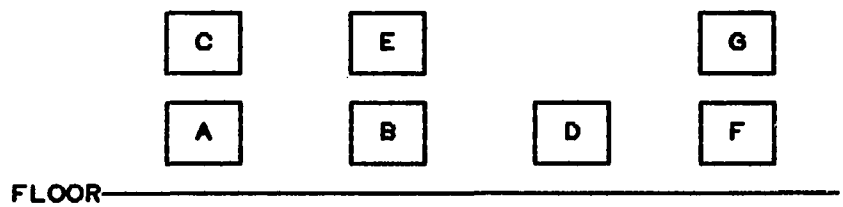
```
will_add(move(Block,From,To),
         on(Block,To)).
will_add(move(Block,From,To),
         clear(From)).


will_delete(move(Block,From,To),
            on(Block,From)).
will_delete(move(Block,From,To),
            clear(To)).
```

```
precond_of(on(Block1,Block2) <&>
                notequal(Block2,floor) <&>
                clear(Block1),
           move(Block1,Block2,floor)).
precond_of(clear(Block1) <&>
                on(Block2,Block3) <&>
                notequal(Block2,Block1) <&>
                clear(Block2),
           move(Block2,Block3,Block1)).


impossible(on(Block1,Block2) <&>
           clear(Block2)).
impossible(on(Block1,Block2) <&>
           on(Block1,Block3) <&>
           notequal(Block2,Block3)).
impossible(on(Block1,Block1)).


exists_in(on(a,floor), state0).
exists_in(on(b,floor), state0).
exists_in(on(c,a), state0).
exists_in(clear(b), state0).
exists_in(clear(c), state0).
```

APPENDIX F

SEVEN BOX EXAMPLE

INITIAL STATE



GOAL

```
develop_plan(state0,
             on(a,b) <&>
             on(b,c) <&>
             on(c,d) <&>
             on(d,e) <&>
             on(e,f) <&>
             on(f,g)).
```

PLAN

```
state0
move(c, a, floor)
move(e, d, floor)
move(g, f, floor)
move(f, floor, g)
move(e, floor, f)
move(d, floor, e)
move(c, floor, d)
move(b, floor, c)
move(a, floor, b)
```

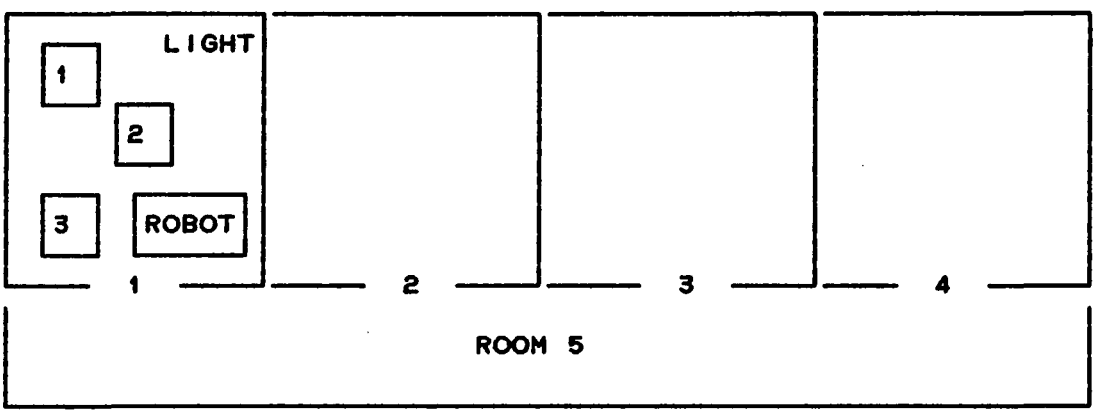
WORLD DESCRIPTION

```
will_add(move(Block,From,To),
         on(Block,To)).
will_add(move(Block,From,To),
         clear(From)).


will_delete(move(Block,From,To),
            on(Block,From)).
will_delete(move(Block,From,To),
            clear(To)).


precond_of(on(Block1,Block2) <&>
               notequal(Block2,floor) <&>
               clear(Block1),
           move(Block1,Block2,floor)).
precond_of(clear(Block1) <&>
               on(Block2,Block3) <&>
               notequal(Block2,Block1) <&>
               clear(Block2),
           move(Block2,Block3,Block1)).


impossible(on(Block1,Block2) <&>
           clear(Block2)).
impossible(on(Block1,Block2) <&>
           on(Block1,Block3) <&>
           notequal(Block2,Block3)).
impossible(on(Block1,Block1)).


exists_in(on(a,floor), state0).
exists_in(on(b,floor), state0).
exists_in(on(d,floor), state0).
exists_in(on(f,floor), state0).
exists_in(on(c,a), state0).
exists_in(on(e,d), state0).
exists_in(on(g,f), state0).
exists_in(clear(b), state0).
exists_in(clear(c), state0).
exists_in(clear(e), state0).
exists_in(clear(g), state0).
```

APPENDIX G

STRIPS EXAMPLE

INITIAL STATE



GOAL

```
developa_plan(state0,
              nextto(box(1), box(2)) <&>
              nextto(box(2), box(3))).
```

PLAN

```
state0
go_nextto(robot, box(2), room1)
pushto(box(2), box(3), room1)
go_nextto(robot, box(1), room1)
pushto(box(1), box(2), room1)
```

WORLD DESCRIPTION

```
will_add(goto(robot,Node,Room),
         at(robot,Node)).
will_add(go_nextto(robot,Object,Room),
         nextto(robot,Object)).
will_add(pushto(Object1,Object2,Room),
         nextto(Object1,Object2)).
will_add(pushto(Object1,Object2,Room),
         nextto(Object2,Object1)).
will_add(turnon(Switch),
         present_state(Switch,on)).
will_add(climbon(Box),
         on(robot,Box)).
will_add(climboff(Box),
         onfloor(robot)).
will_add(pass_thru(robot,Door,Room1,Room2),
         is_in(robot,Room2)).


will_delete(Action,at(Object,Location)) :-
        will_move(Action,Object).
will_delete(Action,nextto(Object,robot)) :-
        !,
        will_delete(Action,
                    nextto(robot,Object)).
will_delete(pushto(Object1,Object2,Room),
                   nextto(robot,Object1)) :-
        !,
        fail.
will_delete(climbon(Box),nextto(robot,Box)) :-
        !,
        fail.
will_delete(climboff(Box),nextto(robot,Box)) :-
        !,
        fail.
will_delete(Action,nextto(Object1,Object2)) :-
        will_move(Action,Object1).
will_delete(Action,nextto(Object1,Object2)) :-
        will_move(Action,Object2).
will_delete(Action,on(Object,Box)) :-
        will_move(Action,Object).
will_delete(climbon(Box),onfloor(robot)).
will_delete(pass_thru(robot,Door,Room1,Room2),
            is_in(robot,Previous_room)).
will_delete(turnon(Switch),
            present_state(Switch,Previous_value)).
```

```
will_move(goto(robot,Node,Room),robot).
will_move(go_nextto(robot,Object,Room),robot).
will_move(pushto(Object1,Object2,Room),robot).
will_move(pushto(Object1,Object2,Room),Object1).
will_move(climbon(Box),robot).
will_move(climboff(Box),robot).
will_move(pass_thru(robot,Door,Room1,Room2),robot).


precond_of(located_in(Node,Room) <&>
            is_in(robot,Room) <&>
            onfloor(robot),
        goto(robot,Node,Room)).
precond_of(is_in(Object,Room) <&>
            is_in(robot,Room) <&>
            onfloor(robot),
        go_nextto(robot,Object,Room)).
precond_of(on(robot,box(1)) <&>
            nextto(box(1),lightswitch),
        turnon(lightswitch)).
precond_of(can_push(Object1) <&>
            is_in(Object2,Room) <&>
            is_in(Object1,Room) <&>
            nextto(robot,Object1) <&>
            onfloor(robot),
        pushto(Object1,Object2,Room)).
precond_of(connects(Door,Room1,Room2) <&>
            is_in(robot,Room1) <&>
            nextto(robot,Door) <&>
            onfloor(robot),
        pass_thru(robot,Door,Room1,Room2)).
precond_of(on(robot,box(Box)),
        climboff(box(Box))).
precond_of(nextto(robot,box(Box)) <&>
            onfloor(robot),
        climbon(box(Box))).


is_always_true(is_in(Door,Room1)) :-
        is_always_true(connects(Door,Room1,Room2)).
is_always_true(connects(Door,Room1,Room2)) :-
        connects1(Door,Room1,Room2).
is_always_true(connects(Door,Room2,Room1)) :-
        connects1(Door,Room1,Room2).
is_always_true(can_push(box(N))).
is_always_true(located_in(node(N),room1)) :-
        N > 0, N < 6.
is_always_true(located_in(node(6),room4)).
is_always_true(is_in(lightswitch,room1)).
is_always_true(at(lightswitch,node(4))).
```

```
connects1(door1,room1,room5).
connects1(door2,room2,room5).
connects1(door3,room3,room5).
connects1(door4,room4,room5).


impossible(at(Object1,Object2) <&>
           at(Object1,Object3) <&>
           notequal(Object2,Object3)).


exists_in(at(box(N),node(N)),state0) :-
        N > 0, N < 4.
exists_in(at(robot,node(5)),state0).
exists_in(is_in(box(N),room1),state0) :-
        N > 0, N < 4.
exists_in(onfloor(robot),state0).
exists_in(present_state(lightswitch,off),state0).
exists_in(is_in(robot,room1),state0).
```

APPENDIX H

PROLOG CODED IN LISP

```
;; The following is a TINY PROLOG INTERPRETER in MACLISP
;; written by Ken Kahn and modified for XLISP by David Betz.
;; It was inspired by other tiny Lisp-based Prologs of
;; Par Emanuelson and Martin Nilsson.
;; There are no side-effects anywhere in the implementation.
;; Though it is VERY slow of course.

(defun prolog (database &aux goal)
       (do () ((not (progn (princ "Query?")
                           (setq goal (read)))))
              (prove (list (rename-variables goal '(0)))
                     '((bottom-of-environment))
                     database
                     1)))

;; prove - proves the conjunction of the list-of-goals
;;         in the current environment

(defun prove (list-of-goals environment database level)
   /   (cond ((null list-of-goals) ;; succeeded since there
                                       are no goals
              (print-bindings environment environment)
              (not (y-or-n-p "More?")))
             (t (try-each database database
                          (cdr list-of-goals)
                          (car list-of-goals)
                          environment level))))

(defun try-each (database-left database goals-left goal
                 environment level &aux assertion
                 new-enviroment)
       (cond ((null database-left) nil) ;; fail since
                                   nothing left in database
             (t (setq assertion
                      (rename-variables (car database-left)
                                        (list level)))
                (setq new-environment
                      (unify goal (car assertion)
                       environment))
```

```
                (cond ((null new-environment) ;; failed to
                                                 unify
                       (try-each (cdr database-left)
                                 database goals-left goal
                                 environment level))
                      ((prove (append (cdr assertion)
                                      goals-left)
                              new-environment
                              database
                              (+ 1 level)))
                      (t (try-each (cdr database-left)
                                   database
                                   goals-left goal
                                   environment level))))))

(defun unify (x y environment &aux new-environment)
       (setq x (value x environment))
       (setq y (value y environment))
       (cond ((variable-p x) (cons (list x y) environment))
             ((variable-p y) (cons (list y x) environment))
             ((or (atom x) (atom y))
                  (cond ((equal x y) environment)
                        (t nil)))
             (t (setq new-environment (unify (car x) (car y)
                                             environment))
                (cond (new-environment (unify (cdr x) (cdr y)
                                              new-environment))
                      (t nil)))))

(defun value (x environment &aux binding)
       (cond ((variable-p x)
              (setq binding (assoc x environment))
              (cond ((null binding) x)
                    (t (value (cadr binding)
                              environment))))
             (t x)))

(defun variable-p (x)
       (and x (listp x) (eq (car x) '?)))

(defun rename-variables (term list-of-level)
       (cond ((variable-p term) (append term list-of-level))
             ((atom term) term)
             (t (cons (rename-variables (car term)
                                        list-of-level)
                      (rename-variables (cdr term)
                                        list-of-level)))))

(defun print-bindings (environment-left environment)
       (cond ((cdr environment-left)
```

```
                (cond ((= 0 (nth 2 (caar environment-left)))
                       (prin1 (cadr (caar environment-left)))
                       (princ " = ")
                       (print (value (caar environment-left)
                                     environment))))
                (print-bindings (cdr environment-left)
                                environment))))

;; a sample database:
(setq db '(((father madelyn ernest))
           ((mother madelyn virginia))
           ((father david arnold))
           ((mother david pauline))
           ((father rachel david))
           ((mother rachel madelyn))
           ((grandparent (? grandparent) (? grandchild))
            (parent (? grandparent) (? parent))
            (parent (? parent) (? grandchild)))
           ((parent (? parent) (? child))
            (mother (? parent) (? child)))
           ((parent (? parent) (? child))
            (father (? parent) (? child)))))

;; the following are utilities
(defun y-or-n-p (prompt)
       (princ prompt)
       (eq (read) 'y))

;; start things going
(prolog db)
```

APPENDIX I

LISP CODED IN PROLOG

```
/*****************************************************************/
          /* L I S P   I N T E R P R E T E R      */
/*****************************************************************/


/*
AUTHOR: A.S.Walker
DATE: 14 Nov 1984
PROJECT : None


This interpreter is based on the MICRO-LISP interpreter
given in the book LISP by Winston and Horn (Addison Wesley
1981) pages 303 to 314.


Built in functions are :
DE      : Define a new function
SETQ     : Bind a value to an object
DSKIN     : Read the lisp expressions from a file
DSKOUT     : Write the complete environment to a file
READ     : Read an expression from the terminal
PRINT     : Write an expression to the terminal
TERPRI     : Write a new line
TYI     : Value is the ascii value of the next character
          input
TYO     : Outputs its integer argument as ascii character
CAR     : Find the head of a list
CDR     : Find the tail of a list
CONS     : Form a dotted pair
LIST     : Form arguments into list
APPEND     : Append two list together
PLUS     : Add two numbers
TIMES     : Multiply two numbers
DIFFERENCE   : Subtract second number from first
QUOTIENT     : Divide first number by second
REMAINDER    : The remainder of dividing first by second
EVAL     : Evaluate the argument
APPLY     : Evaluate first as function with second as its
           args
MAPCAR     : Result is list of results of applying first to
           each element of second
```

```
QUOTE     : Result is its argument unevaluated
COND     : Introduces a conditional statement
LAMBDA     : Introduces a lambda expression
ATOMP     : True if argument is atomic
NULL     : True if argument is NIL
GREATERP     : True if first is larger than second
LESSP     : True if first is less than second
EQ     : True if first is the same as second
AND     : True if both arguments are true
OR     : True if either argument is true
QUI     : Return to Prolog
PROOG     : Drop down to Prolog in a break state
T     : Evaluates to T
NIL     : Evaluates to NIL


It is hoped that this represents a reasonably complete set
of LISP functions. Notable short comings are:
1) No in situ modification of S expressions
                        (NCONC, RPLACA, RPLACD)
2) User defined functions may only be of type LAMBDA
3) No iterative control structures
4) No property lists


This is a piece of demonstration software designed to
illustrate some of the facilities of Prolog-2, in a
realistic and hopefully useful way. Of particular interest
in this case is the programmable tokeniser, but also notice
the succinctness of the parser and the overall simplicity of
the evaluation routines.

*/
```

```
?- op(800,xfy,'.').
?- hash(apply/3,16,3).


/*-------------------------------------------------------------*/
     /*This is the top level READ-EVAL-PRINT loop      */
/*-------------------------------------------------------------*/

lisp :-
        hello,
        repeat,
          nl,nl,
          write("-> "),
          reset_context,   /* clear input buffer */
          read_sexp(S_expression),
          eval(S_expression,Result),
          print1(Result),
        Result = quit,
        !,
        goodbye.


hello :-
        nl,nl,
        write("  MICRO LISP"),nl,
        write("    Expert Systems International"),nl,
        write("    14 November 1984"),nl,
        nl,nl,
        retractall('$lsp_environment'/2),
        tokeniser.


/*-------------------------------------------------------------*/
   /* The following is the definition of the tokeniser for
                      reading LISP      */
/*-------------------------------------------------------------*/

tokeniser :-
     create_character_class(lisp,
        [26*6,7,6*6,4*1,9,1,5,2,3,4*1,4,1,10*0,70*1,128*8]),
     create_state_table(l_int,lisp,integer,l_int,
        [[1,255,255,255,255,255,0,255,255,255],
         [1,255,255,2,2,255,2,2,255,255]]),
     create_state_table(l_atm,lisp,atom,l_atm,
        [[255,1,255,255,255,255,255,255,255,255],
         [1,1,255,2,2,255,2,2,255,255]]),
     create_state_table(l_spc,lisp,none,l_spc,
        [[255,255,1,1,1,1,255,255,255,255],
         [2,2,2,2,2,2,2,2,255,255]]),
     create_state_table(l_eof,lisp,none,l_eof,
        [[255,255,255,255,255,255,255,1,255,255]]),
```

```
      create_state_table(l_com,lisp,none,l_com,
         [[255,255,255,255,255,255,255,255,255,1],
          [1,1,1,1,1,1,1,1,1,2],[3,3,3,3,3,3,3,3,3,3]]),
      create_token_class(lisp,lisp,6,
         [0-l_int, 1-l_atm, 2-l_spc, 3-l_spc, 4-l_spc,
          5-l_spc, 7-l_eof, 9-l_com]),
      state(token_class,_,lisp).


goodbye :-
        nl,
        retractall('$lisp_environment'/2),
        write("Leaving MICRO LISP"),
        state(token_class,_,prolog),
        delete_token_class(lisp,_,_,_),
        delete_state_table(l_int,_,_,_,_),
        delete_state_table(l_atm,_,_,_,_),
        delete_state_table(l_spc,_,_,_,_),
        delete_state_table(l_eof,_,_,_,_),
        delete_state_table(l_com,_,_,_,_),
        delete_character_class(lisp,_),
        nl,nl.


/*-------------------------------------------------------------*/
/* Routines for translating internal form into Lisp text       */
/*-------------------------------------------------------------*/

print1(Head.Tail) :-
        !,
        write("("),
        print1(Head),
        print2(Tail),
        !.
print1(Atom) :-
        write(Atom).

print2(Head.Tail) :-
        !,
        write(" "),
        print1(Head),
        print2(Tail).
print2('NIL') :-
        !,
        write(")").
print2(Atom) :-
        write("."),
        write(Atom),
        write(")").
```

```
/*-------------------------------------------------------------*/
    /* The parser for LISP S expressions, uses the tokens
                    generated by the      */
              /*tokeniser defined above.      */
/*-------------------------------------------------------------*/

read_sexp(S_expression) :-
        fetch_token(Token),
        sexp(Token, S_expression),
        !.
read_sexp(_) :-
        error(1),
        !,fail.


sexp([Atom,_,l_atm],Atom).
sexp([Integer,_,l_int],Integer).
sexp([none,2,_],S_expression) :-
read_sexp_list(S_expression).
sexp([none,5,_],'QUOTE'.S_expression.'NIL') :-
read_sexp(S_expression).
sexp([none,7,_],'QUOTE'.'EOF'.'NIL') :-
get_character(_).


read_sexp_list(S_list) :-
        fetch_token(Token),
        sexp_list(Token,S_list),
        !.


sexp_list([none,3,_],'NIL').
sexp_list([none,4,_],S_expression) :-
        !,
        read_sexp(S_expression),
        fetch_token([none,3,_]).
sexp_list(Token,Sexp.Sexplist) :-
        sexp(Token,Sexp),
        read_sexp_list(Sexplist).


fetch_token(Token) :-
        repeat,
          get_token(Token),
        Token  [_,_,l_com],/* ignore comments */
        !.
```

```
/*-------------------------------------------------------------*/
     /* Evaluation of LISP expressions can be broken */
     /* into two phases, the first deals with special */
    /* functions that do not evaluate their arguments */
    /* the second (apply) actually applies a function */
             /* to its evaluated arguments. */
/*-------------------------------------------------------------*/

eval('COND'.Tail , Result) :-
        eval_cond(Tail,Result),!.
eval('COND'._ , _) :- !,fail.
eval('QUOTE'.S_expression.'NIL' , S_expression) :- !.
eval('SETQ'.Name.Value.'NIL', Result) :-
        eval(Value,Result),
        bind1(Name,Result),
        !.
eval('SETQ'._ , _) :- !,fail.
eval('DE'.Name.Args.Body , Name) :-
        map(atom, Name.Args),
        check_exists(Name),
        assert('$lsp_environment'(Name,
                                  'LAMBDA'.Args.Body),0),
        !.
eval('DE'.Rest, _) :-
        error(2),
        !,
        fail.
eval('DSKIN'.Filename.'NIL',Filename) :-
        !,
        default_name(Filename,"LSP",File),
        create_stream(Filename,read,ascii,file(File,1)),
        see(Filename),
        repeat, /* READ-EVAL-PRINT loop */
          read_sexp(S_expression),/*  do everything*/
          eval(S_expression,Result),/**/
          print1(Result),/*  reporting to user*/
          nl,/**/
        Result == 'EOF',      /*  until end of file   */
        seen,
        delete_stream(Filename,_,_,_,_),
        !.
eval('DSKOUT'.Filename.'NIL',Filename) :-
             !,
             default_name(Filename,"LSP",File),
             create_stream(Filename,write,ascii,
                           file(File,1)),
             tell(Filename),
             write_environment,
             told,
             delete_stream(Filename,_,_,_,_),
             !.
```

```
eval(Function.Arguments, Result) :-
                !,
                mapcar(eval,Arguments,Reslist),
                apply(Function,Reslist,Result).
eval('T','T') :- !.
eval('NIL','NIL') :- !.
eval(Number,Number) :-
                integer(Number),!.
eval(Name,Value) :-
                '$lsp_environment'(Name,Value),
                !.
eval(_,_) :-
                error(3),
                !,
                fail.


/*-----------------------------------------------------------*/
          /* Evaluation of CONDitional statements */
                /* needs some special actions */
/*-----------------------------------------------------------*/

eval_cond((Cond.Action).Rest, Result) :-
                eval(Cond,B),
                not null(B),
                !,
eval_body(Action,Result).
eval_cond(_.Rest, Result) :-
                eval_cond(Rest,Result).
eval_cond('NIL','NIL').


/*-----------------------------------------------------------*/
     /* Since the body of a function can consist of a */
     /* sequence of expressions we must evaluate each */
    /* and return as the result the value of the last. */
/*-----------------------------------------------------------*/

eval_body( Last_statement.'NIL' , Result) :-
                !,
                eval(Last_statement,Result).
eval_body( Statement.Rest , Result) :-
                eval(Statement,_),
                eval_body(Rest,Result),
                !.
eval_body('NIL','NIL').
```

```
/*-------------------------------------------------------*/
   /* This predicate does the real work of calculating */
  /* results and coping with function calls.  The basic */
      /* function simply calls Prolog to do the work */
      /* lambda expressions are the only exception. */
/*-------------------------------------------------------*/

apply('CAR' , (Head._).'NIL' , Head) :- !.
apply('CDR' , (_.Tail).'NIL' , Tail) :- !.
apply('CONS' , A.B.'NIL' , A.B) :- !.
apply('LIST' , List , List ) :- !.
apply('ATOMP' , A.'NIL' , 'T' ) :- atomic(A),!.
apply('ATOMP' , _ , 'NIL') :- !.
apply('LESSP' , A.B.'NIL' , 'T') :- A < B,!.
apply('LESSP' , _ , 'NIL') :- !.
apply('GREATERP' , A.B.'NIL' , 'T') :- A > B,!.
apply('GREATERP' , _ , 'NIL') :- !.
apply('APPEND' , A.B.'NIL' , Result) :-
             append(A,B,Result),!.
apply('AND' , A.B.'NIL' , 'T') :-
             not null(A), not null(B), !.
apply('AND' , _ , 'NIL') :- !.
apply('OR' , A.B.'NIL' , 'T') :-
             (not null(A) ; not null(B)),!.
apply('OR' , _ , 'NIL') :- !.
apply('NULL' , 'NIL'.'NIL' , 'T' ) :- !.
apply('NULL', _ , 'NIL' ) :- !.
apply('EQ' , A.A.'NIL' , 'T' ) :- !.
apply('EQ' , _ , 'NIL' ) :- !.
apply('PRINT', T.'NIL' , T) :- print1(T),!.
apply('READ' , 'NIL' , S_expression) :-
             !,read_sexp(S_expression).
apply('TYO' , Ascii.'NIL' , Ascii) :- put(Ascii),!.
apply('TYI' , 'NIL' , Ascii) :- get0(Ascii),!.
apply('TERPRI' , 'NIL','T') :- nl,!.
apply('PLUS', A.B.'NIL' , R) :- R is A + B, !.
apply('DIFFERENCE' , A.B.'NIL' , R) :- R is A - B, !.
apply('TIMES' , A.B.'NIL' , R ) :- R is A * B, !.
apply('QUOTIENT' , A.B.'NIL' , R) :- R is A // B, !.
apply('REMAINDER', A.B.'NIL', R) :- R is A mod B, !.
apply('APPLY',A.B.'NIL',R) :- apply(A,B,R),!.
apply('EVAL',A.'NIL',R) :- eval(A,R),!.
apply('MAPCAR',A.B.'NIL',R) :- lisp_mapcar(A,B,R),!.
apply('LAMBDA'.Formal.Body , Actual , Result) :-
             !,
             bind_all(Formal,Actual),
             eval_body(Body,Result),
             restore(Formal),
             !.
apply('QUIT' , 'NIL',quit) :- !.
```

```
apply('PROLOG' , 'NIL','NIL') :-
                !,
                state(token_class,_,prolog),
                break,
                state(token_class,_,lisp),
                !.
apply(Function , Arguments , Result) :-
                          /* Do user function call */
                atom(Function),
                eval(Function, L_exp),
                L_exp = ('LAMBDA'._._) ,
                !,
                apply(L_exp, Arguments , Result),
                !.


write_environment :-
          '$lsp_environment'(Name,Value),
          print1('SETQ'.Name.('QUOTE'.Value.'NIL').'NIL'),
          nl,
          fail.
write_environment.


bind1(Name,Value) :-
        retract('$lsp_environment'/2,
                '$lsp_environment'(Name,_),_,_),
        assert('$lsp_environment'(Name,Value),0),
        !.
bind1(Name,Value) :-
        assert('$lsp_environment'(Name,Value),0).


bind_all(Formal_name.F , Actual_value.A) :-
        assert('$lsp_environment'
               (Formal_name,Actual_value),0),
        bind_all(F,A).
bind_all(_,_).


restore(Name.Others) :-
        retract('$lsp_environment'/2,
                '$lsp_environment'(Name,_),_,_),
        restore(Others).
restore(_).
```

```
        /*  Note that bind_all/2 & restore/1 treat  */
    /*  '$lsp_environment'/2 as though it were a stack.  */
       /*  That is there may be several clauses for a */
    /*  particular atom at any given time, of which the */
           /*  topmost is the current binding.  */
      /*  This is for reasons of efficiency (speed).  */


/*-------------------------------------------------------------*/
        /* MAP type functions are a feature of LISP. */
      /* They are higher order functions in that they */
          /* take as an argument another function. */
      /* This function is then applied to each element */
                   /* of a list argument. */
/*-------------------------------------------------------------*/

map(F,'NIL') :- !.
map(F,A.R) :-
        G =.. [F,A],/* Prolog call*/
        G,/*  without saving any*/
        map(F,R)./*  results*/


mapcar( _ , 'NIL' , 'NIL' ) :- !.
mapcar( Fn , A.Rest , R.Others ) :-
        G =.. [Fn , A , R],/* Prolog call*/
        G,/*  saving results*/
        mapcar(Fn,Rest,Others).

lisp_mapcar(_,'NIL','NIL') :- !.
lisp_mapcar(Fn,A.Rest,R.Others) :-
        apply(Fn,A.'NIL',R),                    /* Lisp call */
        lisp_mapcar(Fn,Rest,Others),    /* saving results */
        !.


check_exists(Name) :-
        retract('$lsp_environment'/2,
                '$lsp_environment'(Name,_),_,_),
        nl,write("Redefining "),write(Name),nl,nl,!.
        check_exists(_).


append(H.T,L,H.R) :- !,append(T,L,R).
append(_,L,L).


null('NIL').
```

```
/*-------------------------------------------------------------*/
          /* Rudimentary error handling routines */
/*-------------------------------------------------------------*/

error(1) :- nl,write("Syntax error"),nl.
error(2) :- nl,write(" Wrong format for DE "),nl.
error(3) :- nl,write(" Atom unbound"),nl.


/*************************************************************/
    /*  E N D   O F   L I S P   I N T E R P R E T E R  */
/*************************************************************/
```

GLOSSARY

The following glossary was taken from the two references (Waterman 1986; Harmon and King 1985).

ARTIFICIAL INTELLIGENCE. The subfield of computer science concerned with developing intelligent computer programs. This includes programs that can solve problems, learn from experience, understand language, interpret visual scenes, and, in general, behave in a way that would be considered intelligent if observed in a human.

A subfield of computer science concerned with the concepts and methods of symbolic inference by a computer and the symbolic representation of the knowledge to be used in making inferences. A field aimed at pursuing the possibility that a computer can be made to behave in ways that humans recognize as 'intelligent' behavior in each other.

AUTOMATIC PROGRAMMING. Several projects are underway to develop computer programs that will, in turn, write other computer programs. If successful, such 'higher level' programs will be used to 'automate' major portions of computer programming.

BACKTRACKING. The process of backing up through a sequence of inferences, usually in preparation for trying a different path. Planning problems typically require backtracking strategies that allow a system to try one plan after another as unacceptable outcomes are identified.

BACKWARD CHAINING. An inference method where the system starts with what it wants to prove, e.g., Z-, and tries to establish the facts it needs to prove Z. The facts needed to prove a conjecture (Z) are typically given in rule form; e.g., IF A & B, THEN Z. If A and B aren't known (aren't available as data), the system will try to prove A and B by establishing any additional facts (as specified by other rules) needed to prove them. The additional facts are estab- lished the same way A and B were established, and the process continues until all needed facts are established or the system gives up in defeat.

One of several control strategies that regulate the order in which inferences are drawn. In a rule- based system, backward chaining is initiated by a goal rule. The system attempts to determine if the goal rule is correct. It backs up to the if clauses of the rule and tries to determine if the goal rule is correct. It backs up to the if clauses of the rule and tries to determine if they are correct. This, in turn, leads the system to consider other rules that would confirm the if clauses. In this way the

system backs into its rules. Eventually, the back- chaining sequence ends when a question is asked or a previously stored result is found.

BREADTH-FIRST SEARCH. In a hierarchy of rules or objects, breadth- first search refers to a strategy in which all of the rules or objects on the same level of the hierarchy are examined before any of the rules or objects on the next lower level are checked.

COMMON LISP. A dialect of LISP that is intended to serve as a standard version of LISP that will run on a number of different machines. The first efforts to develop such dialect have already met with some difficulties. LISP is such an easy language to tailor that people implementing it can hardly resist customizing it for the particular computer they are using.

CONFLICT RESOLUTION. The technique of resolving the problem of multiple matches in a rule-based system. When more than one rule's antecedent matches the data base, a conflict arises since (1) every matched rule could approximately be executed next, and (2) only one rule can actually be executed next. A common conflict resolution method is priority ordering, where each rule has an assigned priority

and the highest priority rule that currently matches the data base is executed next.

DATA BASE. The set of facts, assertions, and conclusions used to match against the IF-parts of rules in a rule-based system.

DEPENDENCY-DIRECTED BACKTRACKING. A programming technique that allows a system to remove the effects of incorrect assumptions during its search for a solution to a problem. As the system infers new infor- mation, it keeps dependency records of all its deductions and assumptions, showing how they were derived. When the system finds that an assumption was incorrect, it backtracks through the chains of inferences, removing conclusions based on the faulty assumption.

DEPTH-FIRST SEARCH. In a hierarchy of rules and objects, depth-first search refers to a strategy in which one rule or object on the highest level is examined and then the rules or objects immediately below that one are examined. Proceeding in this manner, the system will search down a single branch of the hierarchy tree until it ends. This contrasts with breadth-first search.

DUAL SEMANTICS. The idea that a computer program can be viewed from either of two equally valid perspectives: procedural semantics (what happens when the program is run) and declarative semantics (what knowledge the program contains).

EVALUATION FUNCTION. A procedure used to determine the value or worth of proposed intermediate steps during a hunt through a search space for a solution to a problem.

EXHAUSTIVE SEARCH. A problem-solving technique in which the problem solver systematically tries all possible solutions in some 'brute force' manner until it finds an acceptable one.

A search is exhaustive if every possible path through a decision tree or network is examined. Exhaustive search is costly or impossible for many problems. Knowledge systems often search exhaustively through their knowledge bases.

FIFTH-GENERATION COMPUTERS. The next generation of computing machines. It is assumed that they will be larger and faster and will incorp- orate fundamentally new designs. Parallel processing, the ability of a computer to process several different programs simultaneously, is expected to result in a massive increment in computational power. Since

expert systems tend to be very large and involve a large amount of processing, it is assumed that expert systems will not reach maturity until these more powerful machines are available.

FORWARD CHAINING. An inference method where the IF-portion of rules are matched against facts to establish new facts.

One of several control strategies that regulate the order in which inferences are drawn. In a rule-based system, forward chaining begins by asserting all of the rules whose if clauses are true. It then checks to determine what additional rules might be true, given the facts it has already established. This process is repeated until the program reaches a goal or runs out of new possibilities.

FRAME. A knowledge representation method that associates features with nodes representing concepts or objects. The features are described in terms of attributes (called slots) and their values. The nodes form a network connected by relations and organized into a hierarchy. Each node's slots can be filled with values to help describe the concept that the node represents. The process of adding or removing values from the slots can activate procedures (self-contained pieces of code) attached to the slots.

These procedures may then modify values in other slots, continuing the process until the desired goal is achieved.

A knowledge representation scheme that associates an object with a collection of features (e.g., facts, rules, defaults, and active values). Each feature is stored in a slot. A frame is the set of slots related to a specific object. A frame is similar to a property list, schema, or record, as these terms are used on conventional programming.

FRAME-BASED METHODS. Programming methods using frame hierarchies for inheritance and procedural attachment.

GENERATE AND TEST. A problem-solving technique involving a generator that produces possible solutions and an evaluator that tests the the acceptability of those solutions.

HEURISTIC. A rule of thumb or simplification that limits the search for solutions in domains that are difficult and poorly understood.

A rule-of-thumb or other device or simplification that reduces or limits search in large problem spaces. Unlike algor- ithms, heuristics do not guarantee correct solutions.

HEURISTIC RULES. Rules written to capture the heuristics an expert uses to solve a problem. The expert's original

heuristics may not have taken the form of if-then rules, and one of the problems involved in building a knowledge system is converting an expert's heuristic knowledge into rules. The power of a knowledge system reflects the heuristic rules in the knowledge base.

HIERARCHY. An ordered network of concepts or objects in which some are subordinate to others. Hierarchies occur in biological taxonomies and corporate organizational charts. Hierarchies ordinarily imply inheritance; and, thus, objects or concepts that were beneath them. 'Tangled hierarchies' occur when more than one higher-level entity inherits characteristics from a single lower-level entity.

HIGH-LEVEL LANGUAGES. Computer languages lie on a spectrum that ranges from machine instructions through intermediate languages like FORTRAN and COBOL to high-level languages like Ada and C. High- level languages incorporate more complex constructs than the simpler languages.

HORN CLAUSE. In logic programming, Horn clauses are expressions connected by 'or' with at most one positive proposition. Thus, a Horn clause takes the form: 'Not A or Not B or ... or Not C or D.' Logical programming is made more efficient by restricting the type of logical assertions

to Horn clauses in much the same way that production systems insist on having knowledge stated in terms of if-then rules.

IF-THEN RULE. A statement of a relationship among a set of facts. The relationships may be definitional (e.g., If female and married, then wife), or heuristic (e.g., If cloudy, the take umbrella).

INFERENCE. The process by which new facts are derived from known facts. A rule (e.g., If the sky is black, then the time is night), combined with a rule of inference (e.g., modus ponens) and a known fact (e.g., The sky is black) results in a new fact (e.g., The time is night).

INFERENCE, DATA-DIRECTED. Inferences that are driven by events rather than goals. See forward chaining.

INFERENCE, GOAL-DIRECTED. Inferences that are driven by goals rather than data. See backward chaining.

INFERENCE METHOD. The technique used by the inference engine to access and apply the domain knowledge, e.g., forward chaining and back- ward chaining.

INHERITANCE. A process by which characteristics of one object are assumed to be characteristics of another. If we

determine that an animal is a bird, for example, then we automatically assume that the animal has all of the characteristics of birds.

INHERITANCE HIERARCHIES. When knowledge is represented in a hierarchy, the characteristics of superordinate objects are inherited by subordinate objects. Thus, if we determine that an auto loan is a type of loan then we know that the credit check procedures that apply to all loans apply to auto loans.

INHERITANCE HIERARCHY. A structure in a semantic net or frame system that permits items lower in the net to inherit properties from items higher up in the net.

INSTANTIATION. The specification of particular values. A specific person with a particular sex and temperature is an instantiation of the generic object 'patient.'

INTERFACE. The link between a computer program and the outside world. A single program may have several interfaces. Knowledge systems typically have interfaces for development (the knowledge acquisition interface) and for users (the user interface). In addition, some systems have interfaces that pass information to and from other programs, data bases, display devices, or sensors.

INTERLISP. A dialect of LISP. A programming environment that provides a programmer with many aids to facilitate the development and maintenance of large LISP programs.

KNOWLEDGE REPRESENTATION. The process of structuring knowledge about a problem in a way that makes the problem easier to solve.

The method used to encode and store facts and relationships in a knowledge base. Semantic networks, object- attribute-value triplets, productions rules, frames, and logical expressions are all ways to represent knowledge.

LISP. A programming language based on List Processing. LISP is the language of choice for American AI researchers.

LIST STRUCTURE. A collection of items enclosed by parentheses, where each item can be either a symbol or another list, e.g., (ENGINE FUEL (Y5 BILL) 23 (CLAY 7)).

LOGIC. A system that prescribes rules for manipulating symbols. Common systems of logic powerful enough to deal with knowledge structures include propositional calculus and predicate calculus.

LOGIC-BASED METHODS. Programming methods that use predicate calculus to structure the program and guide execution.

MACHINE LANGUAGE. A low-level language consisting of primitive instruc- tions. High-level languages are built in machine language.

MACHINE LEARNING. A research effort that seeks to create computer programs that can learn from experience. Such programs, when they become available, will remove a major barrier to the development of very large expert systems.

MACLISP. A dialect of LISP that is tuned for efficiency, but less friendly as a developmental environment.

META-. A prefix indicating that a term is being used to refer to itself. Thus, a meta-rule is a rule about other rules.

METAKNOWLEDGE. Knowledge in an expert system about how the system operates or reasons, such as knowledge about the use and control of domain knowledge. More generally, knowledge about knowledge.

METARULE. A rule that describes how other rules should be used or modified.

MODUS PONENS. A basic rule of logic that asserts that if we know that A implies B and we know for a fact that A is the case, we can assume B.

MULTIPLE LINES OF REASONING. A problem-solving technique in which a limited number of possibly independent approaches to solving the problem are developed in parallel.

MULTIVALUED ATTRIBUTE. An attribute that can have more than one value. If, for example, a system seeks values for the attribute restaurant, and if restaurant is multivalued, then two or more restaurants may be identified.

NATURAL LANGUAGE. The branch of AI research that studies techniques that allow computer systems to accept inputs and produce outputs in a conventional language like English. At the moment, systems can be built that will accept typed input in narrowly constrained domains (e.g., data base inquiries). Several expert systems incorp- orate some primitive form of natural language in their user inter- face to facilitate rapid development of new knowledge bases.

OBJECT. (Context, frame.) Broadly, this refers to physical or conceptual entities that have many attributes. When a collection of attributes or rules are divided into groups, each of the groups is organized around an object. In MYCIN,

following medical practice, the basic groups of attributes (parameters) were clustered into contexts, but most recent systems have preferred the term 'object.' When a know- ledge base is divided into objects, it is often represented by an object tree that shows how the different objects relate to each other. When one uses object-oriented programming, each object is called a frame or unit and the attributes and values associated with it are stored in slots. An object is said to be 'static' if it simply describes the generic relationship of a collection of attri- butes and possible values. It is said to be 'dynamic' when an expert system consultation is being run and particular values have been associated with a specific example of the object.

OBJECT-ATTRIBUTE-VALUE TRIPLETS. (O-A-V triplets.) One method of repre- senting factual knowledge. This is the more general and common set of terms used to describe the relationships referred to as Context- Parameter-Value Triplets in EMYCIN. An object is an actual or con- ceptual entity in the domain of the consultant (e.g., an oil well). Attributes are properties associated with objects (e.g., location, depth, productivity). Each attribute can take different values (e.g., the attribute depth could take on any numerical value from 0 to 60,000 feet).

OBJECT-ORIENTED METHODS. Programming methods based on the use of items called objects that communicate with one another via messages in the form of global broadcasts.

OPERATING SYSTEM. The computer software system that does the 'house- keeping' and communication chores for the more specialized systems. Most conventional computers have standard operating systems that software is designed to utilize. Thus, for example, the IBM per- sonal computer uses a version of MS-DOS. AI languages are often used to write operating systems so that the expert system and the operating system are written in the same language. LISP work- stations, like the Xerox 1100 series and the Symbolics machines, are computers that use a LISP operating system to improve their efficiency and flexibility when running expert systems written in LISP.

PARALLEL PROCESSING. A proposed architecture for computer machinery that would allow a computer to run several programs simultaneously. It would mean that a computer would have several central processors simultaneously processing information.

PLANNING. Designing actions.

PREDICATE CALCULUS. A formal language of classical logic that uses functions and predicates to describe relations between individual entities.

PREDICATE CALCULUS. An extension of propositional calculus. Each elementary unit in predicate calculus is called an object. State- ments about objects are called predicates.

PROBLEM-ORIENTED LANGUAGE. A computer language designed for a particular class of problems, e.g., FORTRAN designed for efficiently performing algebraic computations and COBOL with features for business record keeping.

PROBLEM SOLVING. Problem solving is a process in which one starts from an initial state and proceeds to search through a problem space in order to identify the sequence of operations or actions that will lead to a desired goal. Successful problem solving depends upon knowing the initial state, knowing what an acceptable outcome would be, and knowing the elements and operators that define the problem space. If the elements or operators are very large in number or if they are poorly defined, one is faced with a huge or unbounded problem space and an exhaustive search can become impossible.

PROBLEM SPACE. A conceptual or formal area defined by all of the possible states that could occur as a result of interactions between the elements and operators that are considered when a particular problem is being studied.

PROCEDURAL VERSUS DECLARATIVE. Two complementary views of a computer program. Procedures tell a system what to do (e.g., multiply A times B and then add C). Declarations tell a system what to know (e.g., V = IR).

PROCEDURE-ORIENTED METHODS. Programming methods using nested subroutines to organize and control program execution.

PRODUCTION RULE. The type of rule used in a production system, usually expressed as IF condition THEN action.

PRODUCTION SYSTEM. A type of rule-based system containing IF-THEN statements with conditions that may be satisfied in a data base and actions that may change the data base.

PRODUCTION SYSTEM. A production system is a human or computer system that has a data base of production rules and some control mechan-ism that selects applicable production rules in an effort to reach some goal state. OPS5 is an expert system building tool that is normally referred to as

a production system; it was initially developed in an effort to model supposed human mental operations.

PROGRAMMING ENVIRONMENT. (Environment.) A programming environment is about halfway between a language and a tool. A language allows the use complete flexibility. A tool constrains the user in many ways. A programming environment, like INTERLISP, provides a number of established routines that can facilitate the quick development of certain types of programs.

PROGRAMMING LANGUAGE. An artificial language developed to control and direct the operation of a computer.

PROLOG. A symbolic or AI programming language based on predicate calculus. PROLOG is the most popular language for AI research outside of North America.

PRUNING. Reducing or narrowing the alternatives, normally used in the context of reducing possibilities in a branching tree structure, such as the search through a problem space.

In expert systems, this refers to the process whereby one or more branches of a decision tree are 'cut off' or ignored. In effect, when an expert system consultation is underway, heuristic rules reduce the search

space by determining that certain branches (or subsets of rules)can be ignored.

REAL-WORLD PROBLEM. A complex, practical problem which has a solution that is useful in some cost-effective way.

REASONING. The process of drawing inferences or conclusions.

RECOGNIZE-ACT CYCLE. The cycle of events in a production or forward- chaining system. During the recognize phase, rules are examined to see if their if clauses are true based on information currently stored in memory. During the act phase, one of the rules is sel-ected and executed and its conclusion is stored in memory.

REPRESENTATION. The process of formulating or viewing a problem so it will be easy to solve.

The way in which a system stores knowledge about a domain. Knowledge consists of facts and the relationships between facts.

RESOLUTION. (Resolution theorem proving.) The inference strategy used in logical systems to determine the truth of an assertion. This complex, but highly effective, method establishes the truth of an assertion by determining that a

contradiction is encountered when one attempts to resolve clauses, one of which is a negation of the thesis one seeks to assert.

RESOLUTION THEOREM PROVING. A particular use of deductive logic for proving theorems in the first-order predicate calculus. The method makes use of the following resolution principle: (A v B) and (-A v C) implies (B v C).

ROBOTICS. The branch of AI research that is concerned with enabling computers to 'see' and 'manipulate' objects in their surrounding environment. AI is not concerned with robotics, as such, but it is concerned with developing the techniques necessary to develop robots that can use heuristics to function in a highly flexible manner while interacting with a constantly changing environment.

ROBUSTNESS. That quality of a problem solver that permits a gradual degradation in performance when it is pushed to the limits of its scope of expertise or is given errorful, inconsistent, or incomplete data or rules.

RULE. A formal way of specifying a recommendation, directive, or strategy, expressed as IF premise THEN conclusion or IF condition THEN action.

A conditional statement of two parts. The first part, comprised of one or more if clauses, establishes conditions that must apply if a second part, comprised of one or more then clauses, is to be acted upon. The clauses of rules are usually A-V pairs or O-A-V triplets.

RULE-BASED METHODS. Programming methods using IF-THEN rules to perform forward or backward chaining.

RULE-BASED PROGRAM. (Production system.) A computer program that repre-sents knowledge by means of rules.

SCHEDULER. The part of the inference engine that decides when and in what order to apply different pieces of domain knowledge.

SCHEMA. A frame-like representation formalism in a knowledge engineer- ing language (e.g, SRL).

SEARCH. The process of looking through the set of possible solutions to a problem in order to find an acceptable solution.

SEARCH SPACE. The set of all possible solutions to a problem.

SEMANTIC. Refers to the meaning of an expression. It is often contrast-ed with syntactic, which refers to the formal pattern of the ex-pression. Computers are good at establishing that the correct syntax is being used; they have a great deal of trouble establish-ing the semantic content of an expression. For example, look at the sentence, 'Mary had a little lamb.' It is a grammatically correct sentence; its syntax is in order. But its semantic content-its meaning-is very ambiguous. As we alter the context in which the sentence occurs, the meaning will change.

SEMANTIC NET. A knowledge representation method consisting of a network of nodes, standing for concepts or objects, connected by arcs describing the relations between the nodes.

SEMANTIC NETWORKS. A type of knowledge representation that formalizes objects and values as nodes and connects the nodes with arcs or links that indicate the relationships between the various nodes.

SLOT. An attribute associated with a node in a frame system. The node may stand for an object, concept, or event; e.g., a node represent- ing the object employee might have a slot for the attribute name and one for the

attribute address. These slots would then be filled with the employee's actual name and address.

A component of an object in a frame system. Slots can contain intrinsic features such as the object's name, attributes and values, attributes with default values, pointers to related frames, and information about the frame's creator, etc.

SYMBOL. A string of characters that stands for some real-world concept.

An arbitrary sign used to represent objects, concepts, oper-ations, relationships, or qualities.

SYMBOL-MANIPULATION LANGUAGE. A computer language designed expressly for representing and manipulating complex concepts, e.g., LISP and PROLOG.

SYMBOLIC VERSUS NUMERIC PROGRAMMING. A contrast between the two pri-mary uses of computers. Data reduction, data-base management, and word processing are examples of conventional or numerical programming. Knowledge systems depend on symbolic programming to manipulate strings of symbols with logical rather than numerical operators.

SYMBOLIC REASONING. Problem solving based on the application of strategies and heuristics to manipulate symbols standing for problem concepts.

SYNTACTIC. Refers to the formal pattern of an expression (contrast semantic).

TOY PROBLEM. An artificial problem, such as a game, or an unrealistic adaptation of a complex problem.

TRACING FACILITY. A mechanism in a programming or knowledge engineering language that can display the rules or subroutines executed, including the values of variables used.

TREE STRUCTURE. A way of organizing information as a connected graph where each node can branch into other nodes deeper in the struc- ture.

UNITS. A frame-like representation formalism employing slots with values and procedures attached to them.

USER. A person who uses an expert system, such as an end-user, a domain expert, a knowledge engineer, a tool builder, or a clerical staff member.

VALUE. A quantity or quality that can be used to describe an attribute. If we are considering the attribute 'color': then the possible values of color are all of the names of colors that we might use. If we are considering a particular object, we observe it and assign a specific value to the attribute by saying, for example, 'That paint is colored bright red.'

WINDOWS. Conventional computer terminals use the entire screen to pre-sent information drawn from one data base. Computer terminals that can utilize window software can divide the screen into several different sections (or windows). Information drawn from different data bases can be displayed in different windows. Thus, for example, with a Macintosh computer one can have a word processing program going on in one window and a graphics program going on simultaneously in a second window. Most current expert systems research is being conducted on computers that allow the user to display different views of the systems activity simultaneously. Windows are an example of a technique originally developed by AI researchers that has now become part of conven-tional programming technology.

REFERENCES

Allen, John R. "Speaking LISP," Computer Language, II, No. 7 (July 1985), pp. 27-33.

Clark, K. L. and S. A. Tarnlund, ed. Logic Programming. New York: Academic Press, 1982.

Clocksin, W. F. and C. S. Mellish. Programming in Prolog. New York: Springer-Verlag, 1984.

Cohen, Jacques. "Describing Prolog by its Interpretation and Compilation," Communications of the ACM, XXVIII, No. 12 (December 1985), pp. 1311-24.

Cohen, Paul and Edward A. Feigenbaum. The Handbook of Artificial Intelligence. Vol. III, Los Altos, CA: William Kaufmann Inc., 1982.

Colmerauer, Alain. "Prolog in 10 Figures," Communications of the ACM, XXVIII, No. 12 (December 1985), pp. 1296-1310.

DeGroot, Doug and Gary Lindstrom. Logic Programming: Functions, Relations, and Equations. Englewood Cliffs, New Jersey: Prentice-Hall, 1986.

Eisenbach, Susan and Chris Sadler. "Declarative Languages: An Overview," Byte, X, No. 8 (August 1985), pp. 181-97.

Fikes, Richard E. and Nils J. Nilsson. "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," Artificial Intelligence, II, No. 3/4 (1971), pp. 189-208.

Genesereth, Michael R. and Matthew L. Ginsberg. "Logic Programming," Communications of the ACM, XXVIII, No. 9 (September 1985), pp. 933-41.

Harmon, Paul and David King. Expert Systems: Artificial Intelligence in Business. New York: John Wiley & Sons, Inc., 1985.

Kenig, Marc E. "Procedural Programming versus Prolog," AI Expert, November 1986, pp. 59-65.

Kowalski, Robert. Logic for Problem Solving. New York: Elsevier Science Publ. Co., 1979.

Lloyd, J. W. Foundations of Logic Programming. New York: Springer-Verlag, 1984.

Lusk, Ewing L. and Ross A. Overbeek. Logic Machine Architecture Inference Mechanisms: Layer 2 User Reference Manual Release 2.0. Springfield, VA: National Technical Information Service (U.S. Department of Commerce), April 1984a.

Lusk, Ewing L. and Ross A. Overbeek. The Automated Reasoning System ITP. Springfield, VA: National Technical Information Service (U.S. Department of Commerce), April 1984b

Malachi, Yonathan, Zohar Manna and Richard Waldinger. TABLOG: The Deductive-Tableau Programming Language. Stanford, CA: Stanford University Department of Computer Science, June 1984.

Nilsson, Nils J. Principles of Artificial Intelligence. Palo Alto, CA: Tioga Publ., 1980.

Prolog-86 User's Guide and Reference Manual. Rheem Valley, CA: MICRO-AI, 1984.

Rich, Elaine. Artificial Intelligence. New York: McGraw-Hill, 1983.

Robinson, J. A. and E. E. Sibert. "LOGLISP: An Alternative to Prolog," Machine Intelligence, IX (1982a), pp. 399-419.

Robinson, J. A., E. E. Sibert and K. J. Greene. The LOGLISP Programming System. Syracuse, New York: Logic Programming Research Group of Syracuse University, February 1984a.

Robinson, J. A. New Generation Knowledge Processing: Syracuse University Parallel Expression Reduction. Rome, New York: Rome Air Development Center, December 1984b.

Russell, Stuart. The Compleat Guide to MRS. Stanford, CA: Stanford University Department of Computer Science, June 1985.

Warren, David H. D. and Luis M. Pereira. "Prolog: The Language and Its Implementation Compared with LISP," SIGART Newsletter, No. 64, August 1977, pp. 109-15.

Waterman, Donald A. A Guide to Expert Systems. Reading, Mass: Addison-Wesley Publ. Co., 1986.

Wise, D. S. "Functional Programming," Encyclopedia of Computer Science and Engineering. Anthony Ralston, ed., New York: Van Nostrand Reinhold Co., 1983.

Zhu, Mingfa. Planning and Scheduling Using Temporal Logic. Detroit, Michigan: Wayne State University Ph D dissertation, 1985.

SELECTED BIBLIOGRAPHY

Allen, James F. "An Interval-Based Representation of Temporal Knowledge," Proceedings of the Seventh International Conference on Artificial Intelligence, August 1981, pp. 221-26

Allen, James F. "Maintaining Knowledge about Temporal Intervals," Communications of the ACM, XXVI, No. 11 (1983), pp. 832-43.

Allen, James F. "Towards a General Theory of Action and Time," Artificial Intelligence, XXIII, No. 2 (1984), pp. 123-54).

Barr, Avron and Edward A. Feigenbaum, ed. The Handbook of Artificial Intelligence. Vol. I, Los Altos, CA: William Kaufmann Inc., 1981.

Barr, Avron and Edward A. Feigenbaum, ed. The Handbook of Artificial Intelligence. Vol. II, Los Altos, CA: William Kaufmann Inc., 1982.

Bonnet, Alain. Artificial Intelligence: Promise and Performance. Englewood Cliffs, NJ: Prentice-Hall International, 1985.

Bratko, Ivan. Prolog: Programming for Artificial Intelligence. Reading, Mass: Addison-Wesley Publ. Co., 1986.

Burnham, W. D. and A. R. Hall. Prolog Programming and Applications. New York: John Wiley and Sons, 1985.

Campbell, J. A., ed. Implementations of PROLOG. Chichester, England: Ellis Horwood Ltd., 1984.

Cheeseman, Peter. "A Representation of Time for Automatic Planning," Proceedings of the International Conference on Robotics, March 1984, pp. 513-18.

Clark, K. L. and F. G. McCabe. micro-PROLOG: Programming in Logic. Englewood Cliffs, New Jersey: Prentice-Hall, 1984.

Charniak, Eugene and Drew McDermott. Introduction to Artificial Intelligence. Reading, Mass: Addison-Wesely Publ. Co., 1985.

Covington, Michael and Andre Vellino. "Prolog Arrives," PC Tech Journal, IV, No. 11 (November 1986), pp. 52-69.

de Saram, Hugh. Programming in micro-Prolog. Chichester, England: Ellis Horwood Ltd., 1985.

Ennals, J. R. Beginning micro-Prolog. New York: Harper & Row Publ., 1984.

Hogger, Christopher John. Introduction to Logic Programming. London: Academic Press, 1984.

Jackson, Peter. Introduction to Expert Systems. Reading, Mass: Addison-Wesley Publ. Co., 1986.

Jackson, Philip C. Introduction to Artificial Intelligence. New York: Dover Publ. Inc., 1974, 1985.

Kowalski, Robert and Donald Kuehner. "Linear Resolution with Selection Function," Artificial Intelligence, II, No. 3/4 (1971), pp. 227-60.

Kowalski, Robert. "Logic Programming," Byte, X, No. 8 (August 1985), pp. 161-77.

Manna, Zohar and Richard Waldinger. The Logical Basis for Computer Programming, vol. 1. Reading, Mass: Addison-Wesley Publ. Co., 1985.

Mishkoff, Henry C. Understanding Artificial Intelligence. Dallas, Texas: Texas Instruments Information Publ. Center, 1985.

Nilsson, Nils J. "A Production System for Automatic Deduction," Machine Intelligence, IX (1979), pp. 101-26.

Pearl, Judea. Heuristics. Reading, Mass: Addison-Wesley Publ. Co., 1984.

Quine, W. V. Mathematical Logic. Cambridge, Mass.: Harvard Univ. Press, 1981.

Robinson, J. A. "Computational Logic: The Unification Computation," Machine Intelligence, VI (1971), pp. 63-72.

Robinson, J. A. Using LOGLISP to Build and Operate Deductive Databases. Syracuse, New York: Logic Programming Research Group of Syracuse University, August 1982b.

Robinson, Phillip R. Using Turbo Prolog. Berkeley, CA: Osborne McGraw-Hill, 1987.

Shafer, Dan. Turbo Prolog Primer. Indianapolis, Indiana: Howard W. Sams and Co., 1986.

Schildt, Herbert. Advanced Turbo Prolog. Berkeley, CA: Osborne McGraw-Hill, 1987.

Shirai, Yoshiaki and Jun-ichi Tsujii. Artificial Intelligence: Concepts, Techniques and Applications. New York: John Wiley and Sons, 1984.

Sterling, Leon and Ehud Shapiro. The Art of Prolog. Cambridge, Mass: MIT Press, 1986.

Turner, Raymond. Logics for Artificial Intelligence. Chichester, England: Ellis Horword Ltd., 1984.

Waldinger, Richard. "Achieving Several Goals Simultaneously," Machine Intelligence, VIII (1977), pp. 94-136.

Warren, D. H. D. WARPLAN: A System for Generating Plans, Memo 76, Department of Computational Logic, University of Edinburgh School of Artificial Intelligence. June 1974.

Winston, Patrick Henry. Artificial Intelligence. Reading, Mass.: Addison-Wesley Publ. Co., 1984.

Wos, Larry, Ross Overbeek, Ewing Lusk and Jim Boyle. Automated Reasoning: Introduction and Applications. Englewood Cliffs, New Jersey: Prentice-Hall, 1984.

Yasdani, Masoud, ed. Artificial Intelligence: Principles and Applications. London: Chapman and Hall, 1986.