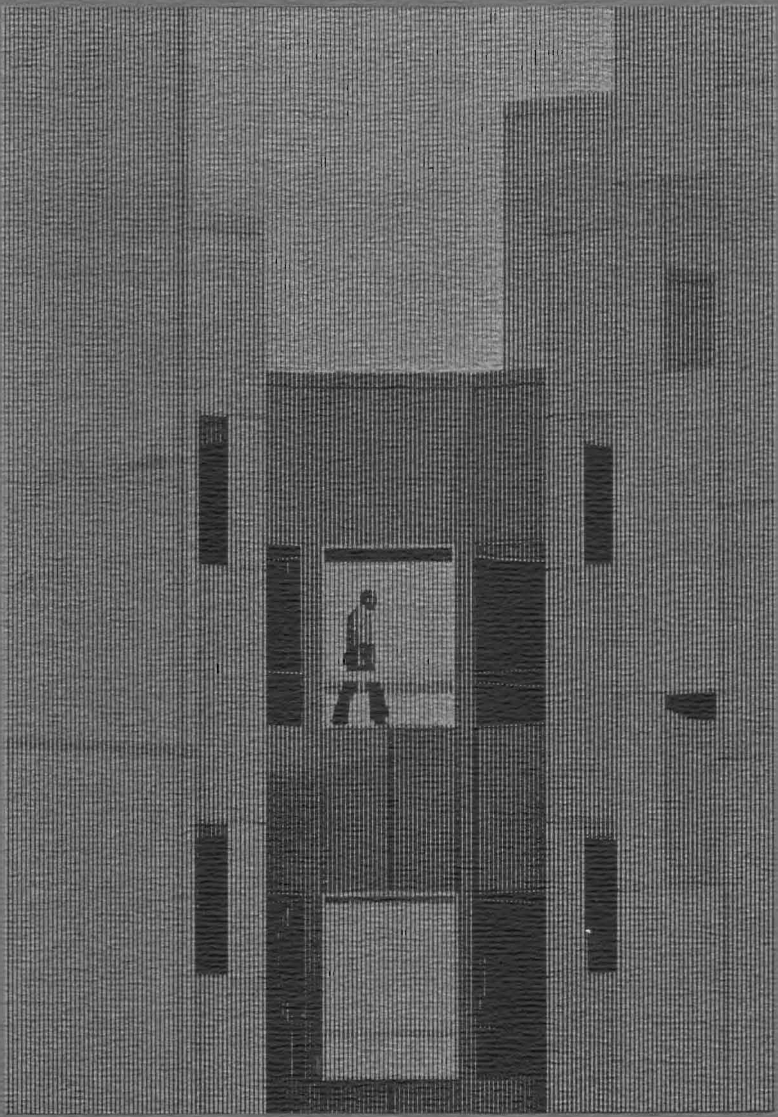
TECHNICAL REPORT

SANTA TERESA LABORATORY, SAN JOSE CALIFORNIA



ALGORITHMS FOR ARTIFICIAL INTELLIGENCE IN APL2

May 1986 TR 03.281

by Dr. James A. Brown
Ed Eusebi
Janice Cook
Leo H. Groner

IBM

May 1986
Revised Nov. 1986
TR 03.281

ALGORITHMS FOR ARTIFICIAL INTELLIGENCE IN APL2

By

Dr. James A. Brown

Ed Eusebi

Janice Cook

Leo H. Groner

International Business Machines Corporation

General Products Division

Santa Teresa Laboratory

San Jose, California

ABSTRACT

Many great advances in science and mathematics were preceded by notational improvements. While a given algorithm can be implemented in any general purpose programming language, discovery of algorithms is heavily influenced by the notation used to investigate them. *APL*2 conceptually applies functions in parallel to arrays of data and so is a natural notation in which to investigate parallel algorithms. No claim is made that *APL*2 is an advance in notation that will precede a breakthrough in Artificial Intelligence but it is a new notation that allows a new view of the problems in AI and their solutions. *APL*2 can be used in problems traditionally programmed in LISP, and is a possible implementation language for PROLOG-like languages. This paper introduces a subset of the *APL*2 notation and explores how it can be applied to Artificial Intelligence.

CONTENTS

Introduction

This paper discusses many of the fundamental ideas of Artificial Intelligence and their implementation in *APL*2. Emphasis is on predicate logic but discussions of other topics are included.

This paper is divided into 5 parts. Part 1 introduces Artificial Intelligence (AI) and discusses the type of problem to be solved. The features of *APL*2 that make it suitable for AI applications are discussed.

Part 2 discusses logic and chained inference and includes a brief discussion of search strategies.

Part 3 introduces the *APL*2 language concentrating on the features actually used in the algorithms. It includes a comparison of *APL*2 and LISP and an example program written in each language.

Part 4 presents the *APL*2 algorithms beginning with the representations of logic with nested arrays and proceeds through development of algorithms for Unification, Resolution, and searching. It concludes with an implementation of PROLOG in APL2.

Part 5 goes beyond the fundamentals to look at such topics as frames, boolean logic, and fuzzy logic.

Part 1: Artificial Intelligence

AI algorithms tend to deal with mixing and matching a set of tokens rather than doing mathematical computations on numbers. They tend to operate on nested lists of these tokens rather than on rectangular patterns of them and this means that they are often recursive.

Traditionally, these algorithms have been written in LISP, a list processing language, and more recently in PROLOG a logic programming language.

This part will discuss AI in general and point out the features of *APL*2 that make it a candidate for implementation of AI programs.

1.1: What is Artificial Intelligence

There is no agreed on definition of Artificial Intelligence. The field tends to be defined both by the problems it addresses and the tools applied in the solutions.

Artificial Intelligence (AI) algorithms are an attempt to model with a computer the mental facilities of human beings who are assumed to have real intelligence. They often involve drawing conclusions and making decisions and include recognition of written natural language, speech recognition, computer vision, robotics, and expert systems.

An AI program is one which exhibits behavior which would be considered intelligent if it were done by a human -- it accepts and responds in a natural language; it knows rules and applies them against the given data; an advanced system can alter the data and the rules (i.e. learn from experience) (Fr1).

Traditional programs tend to do the things that Neanderthal man could not do -- payrolls, computation, text processing, etc. This is because man had to figure out how to do these things (after inventing the need for them) and, therefore, is good at specifying how to do them. The Neanderthal did make noise to communicate, could recognize faces, could move his arm to a desired location. No one ever had to consciously figure out the mechanism to do these things and so it is hard to specify how they work. AI programs often attack these problems.

In some sense, every computer program applies rules to data. This is, almost by definition, what an algorithm does. The difference is that an ordinary program contains the rules imbedded in the logic of the program. There is no separately definable piece that represents the knowledge being applied. An AI program contains a knowledge base (a database for rules and facts) and general algorithms for combining the rules and facts. If the knowledge changes, the program does not change, only the database changes.

In building practical systems, the AI programming task is not particularly difficult (although getting good performance is a challenge). The real problem is constructing and validating the rules. This has given rise to a new field of study called Knowledge Engineering. A knowledge engineer is essentially a systems analyst / application programmer. His task is to recognize important aspects of a problem and present a formalization that can be implemented on a computer. This topic will not be discussed further in this paper.

1.2: The Problem to be Solved

Given a set of facts and relationships between them, people routinely draw conclusions from them. The challenge is to develop computational procedures which can draw the same conclusions.

Real world situations tend to be complex and sometimes imprecise. This is why most early AI investigations dealt with games like checkers and chess where rules are simple and precise. Of course, even simple and precise rules can lead to combinatorial complexities and this is the case with chess.

In specifying a logic program, one of the most difficult jobs is making sure your facts are indeed true; that you have stated all the relevant facts, that your words mean the same thing everywhere.

Facts are things that you assert to be true. If they are not true, that's your problem. A logic program will attempt to draw a conclusion. You may have heard the following puzzle:

Question: If you call the tail of a cow a leg,
how many legs does a cow have?

Answer in the AI world: Five.

Answer in the real world:
Four because saying it's so doesn't make it so.

In the artificially intelligent world, however, saying it's so does make it so. Thus if we assert:

Nothing is better than complete happiness in life

and assert

A ham sandwich is better than nothing

we can conclude

A ham sandwich is better than complete happiness in life.

While this might be debatable, most people would disagree. This is an example of word sense ambiguity. While the above example is a little ridiculous, exactly this kind of misunderstanding doomed the computer language translation efforts in the 1950's. One such effort translated "out of sight, out of mind" into Russian as "blind and insane" (Gr1).

Great care must be exercised in choosing the facts and avoiding the ambiguities of natural language. Most people agree on logic and the rules of deduction but not on knowledge. Different people call the same thing by different names. Real world concepts tend to be fuzzy, not exact. For example, membership in the set of all green objects is subject to human judgement.

This paper will ignore the real problems of language and the human decision on representation of knowledge and concentrate on the algorithms that solve logic problems.

Thus, the problem to be solved is: Given a set of statements assumed to be true, draw conclusions which are true.

1.3: APL2 as an AI Language

*APL*2 is a candidate for writing AI applications. The following is a discussion of the features of *APL*2 that make this true. If you are already convinced of this, or don't wish to be, you may skip this section.

- Machine Independence - *APL*2 avoids machine specific features and, in general, the machine architecture is irrelevant except for precision of numeric computations and performance.

- Data is typed, not variables - A name may contain at different times any kind of data. *APL*2 has only two types of data -- characters and numbers. Internal conversion between various formats of numbers and characters is transparent to an algorithm. A name may contain differently shaped data at different times. The Expert System Environment product contains a set of individual get/set commands rather than generic ones precisely because it is implemented in PASCAL which has strong data typing (Hi1). It is unlikely that an expert system shell written in *APL*2 would do this.

- Nested arrays - *APL*2 arrays contain other arrays in any combination and to any depth. These arrays, in themselves, may be used to represent the necessary data models -- graphs, Frames (Ke1), etc.. Vectors of vectors is a subset of arrays of arrays which is useful in representing trees and lists -- the traditional logic programming structures. In addition, *APL*2 can represent data arranged along more than one independent axis making it ideal for representing non-linear data like relational tables.

- Dynamic data - nested data structures are not declared. They may be created, modified, rearranged, and deleted as part of program execution. Utilization of space for data and programs is dynamically managed.

- Dynamic name scope - non-local names bind to the most recent value. Scope depends on the calling order of functions.

- Symbolic computation - nested arrays containing character strings provide a means for computing on arbitrary symbols. *APL*2 can dynamically treat a character string as though it were an expression in a program. Thus, parts of programs can be constructed during execution.

- Recursion - Given arrays of arrays as recursive data structures, recursive algorithms are easily written to process the data.

- Parallelism - *APL2* operations apply to whole arrays at once. There is often no need to write a loop or other structured programming constructs to achieve repeated applications of programs. Use of parallelism reduces the use of recursion and leads to more compact and more understandable algorithms. In particular, CAR CDR recursion is almost always replaced with a parallel operation. *APL2* algorithms would not need to be rewritten to take advantage of parallel hardware.

- Function modifiers - *APL2* operators are used to modify the behavior of functions and make them do special things. "Each" (¨) is the parallel analog of iteration; outer product (∘.f), is used whenever a program is to be applied in all combinations with a set of arguments. User written operators give the programmer the ability to create his own control structures. This gives an effective blend of recursive and parallel programming styles.

- Functional programming style - *APL2* programs tend to be written in a modular style almost like extensions of the language itself and then connected in expressions. Functions may be passed to programs along with data. Defined operators can be used to create applicative sets of controls and filters (Eu1) (Eu2). This gives an effective blend of procedural and functional styles.

In addition to facilities especially suited to logic programming, *APL2* has facilities that make it applicable in the other fields of computation: business data processing, graphics, Engineering Scientific, financial, etc..

- Powerful general purpose computational primitives - Mathematics and computation are available to write precise and concise algorithms for business and science. This is useful in producing a combination application that uses logic programming at the user interface to give inputs to a computational phase. Thus, you can write expert systems that have a significant traditional computational component. The computational ability is also useful in computing fuzzy logics.

- Programmable error handling - an *APL2* program can be written so that it is never out of control. Unexpected data or even program errors can be captured and handled under program control.

- Full graphics access - *APL2* has access to a complete set of graphics facilities including GDDM and the Interactive chart utility (ICU).

- Full panel management - *APL2* can use ISPF to interact with a user giving a standard interface like that used by other products. A Prototyping Environment (APE) provides panel management along with many other productivity enhancements.

- Full relational database support - *APL2* provides access to SQL/DS and Data Base 2 for relational data. This could well be the repository for a knowledge base or any other data of an application.

- Full access to other languages - programs in other languages can be called from *APL2* programs using *APL2* syntax. All the *APL2* control structures apply to these programs without change and without exception. *APL2* applications tend to be modular -- collections of small programs combined in a functional style. You could write each little program in a different language if you want. You can use existing subroutine libraries.

- Interactive - You interact with *APL2* in real time. You can experiment with different data structures and alternate algorithms. You can debug programs by running them, fixing errors when found, and continuing from where the program left off.

- Full Program Product support - *APL2* is one of the key IBM languages and receives full IBM Program Product Support.

- Under active development - *APL2* is an evolutionary step from *VS APL* and the evolution is continuing.

Thus, in summary, *APL2* can be used in the implementation of an AI system and it offers advantages not found in the other languages of choice. The style of programming can be the same as the traditional AI languages or can be changed to reflect a parallel orientation.

Part 2: Logic

The purpose of logic programming is to do computation on the truth of statements. It deals with facts which are known to be true, methods for combining facts, and rules for producing new facts from the existing ones. Appendix 3 contains a summary of predicate logic.

2.1: Logic Statements

A proposition is a statement that is true or false. The truth of the proposition "Sten is mortal" may not be known at some point in time but when it is known, the value will be either true or false. A predicate is a generalization of a proposition which allows variables. For example, "X is a man" is true whenever X is given some particular man as a value. Predicates may be more formally stated by removing the non-essential English and writing the relationship like a function applied to arguments as in "mortal(sten)" or "P(f(X),Y)". The intent is that when you are told how to evaluate a predicate, it will yield true or false. These predicates are linked together by zero or more connectives yielding logic statements or formulas.

In the following, let the single capitol letters P Q R S T U and V represent arbitrary predicates. Thus P might represent "Sten is mortal" and Q "Spot is a dog".

The intention is to write statements that are true. Therefore, if P is true, write:

$$P$$

If P is false, write

$$\sim P$$

(~ means not).

The connectives for statements are "and" ($\wedge$) and "or" ($\vee$).

$$P \vee Q$$

means at least one of P or Q is true

$$P \wedge Q$$

means both P and Q are true

(The symbol $\vee$ is from the Latin "vel" meaning either or both.)

A set of simple expressions written with connectives is called a clause.

$P \vee Q \wedge (\sim T)$ is a clause

A clause with only "or" ($\vee$) connectives is called a disjunctive clause.

$P \vee Q \vee T$ is a disjunctive clause

The clause $P \vee (\sim P)$ is always true and the clause $P \wedge (\sim P)$ is always false. The clause resulting from $P \wedge (\sim P)$ is an empty clause (no terms) and represents a contradiction. A contradiction would seem like a useless result but, in fact, is one of the key ways of solving logic programs as is seen in part 4. A clause with only "and" ($\wedge$) connectives is called a conjunctive clause.

$P \wedge Q \wedge T$ is a conjunctive clause

Negating a conjunctive clause gives a disjunction:

$\sim(P \wedge Q \wedge R)$ is $(\sim P) \vee (\sim Q) \vee (\sim R)$

Implication is a conditional statement - "if P is true then Q is true". This does not claim that P is true only that if P is true, then so is Q. Implication is represented by the following logic statement:

$Q \vee (\sim P)$

The intent is that $Q \vee (\sim P)$ is a true statement. It is true if either of Q or $\sim P$ is true. P is either true or false. If P is false, then $\sim P$ is true and so is $Q \vee (\sim P)$. If P is true, then since P implies Q, Q is true and so is $Q \vee (\sim P)$. Therefore the implication "if P then Q" written $Q \vee (\sim P)$ is true regardless of the truth of P.

The following notations are also used for implication:

$Q <= P$
$P ==> Q$
$P \supset Q$
$P \leq Q$ ($\leq$ is less or equal in the *APL2* sense)

In this paper, any of the equivalent expressions $(\sim P) \vee Q$, $Q \vee (\sim P)$, or $Q <= P$ will be used. They are read "P implies Q" or "Q if P".

2.2: Rules of Inference

Rules of inference are rules for producing new true statements from given ones. These rules imply a reasoning process without reference to the meaning of statements. For example, the "Modus Ponens" Inference rule says: "If P implies Q, and P is true, then Q is true." (Modus Ponens means "Method of detachment". In some sense, the conclusion is detached from the premises.)

Here is a summary of some rules of inference:

stmt 1	stmt 2	infers	name
P	$Q\vee(\sim P)$	Q	modus ponens
$P\vee Q$	$(\sim P)\vee Q$	Q	merge
P	$\sim P$	empty	a contradiction
$P\vee Q$	$(\sim P)\vee(\sim Q)$	$(\sim Q)\vee Q$	tautology
$Q\vee(\sim P)$	$R\vee(\sim Q)$	$R\vee(\sim P)$	chaining

Merge is sometimes called "existential elimination". The chaining rule may be read "If P implies Q and Q implies R, then P implies R"

** A General Rule of Inference

Resolution is a rule of inference which includes all of the above rules. In words, resolution says "if one disjunctive clause contains a negated term, and another disjunctive clause contains the same term non-negated, then you may infer the disjunction of the other terms." In one sense, you might say that the two terms differ in sign and cancel. For example:

from the two clauses

$P\vee Q\vee(\sim R)$
$(\sim P)\vee(\sim S)\vee T$

you may infer

$Q\vee(\sim R)\vee(\sim S)\vee T$

This is easy to picture. The two input clauses are true. One of P or $\sim P$ is false and where it is false, the other terms of that clause must provide the truth. You should be able to apply the resolution rule to the table of rules of inference and see how resolution contains them all. In the first four cases, delete P from stmt 1 and $\sim P$ from stmt 2 and "or" ($\vee$) together what's left. In the chaining rule, delete Q from stmt 1 and $\sim Q$ from stmt 2.

When two statements like

$P \vee Q \vee (\sim R)$
$(\sim P) \vee (\sim S) \vee T$

are written, it is a claim that they are both true simultaneously. Thus they are really connected by the logical "and" ($\wedge$) and could be written:

$$(P \vee Q \vee (\sim R)) \wedge ((\sim P) \vee (\sim S) \vee T)$$

This is called a conjunctive normal form and is the form used to represent a knowledge base which is just a collection of statements asserted to be true.

Remember that the letters used in the above clauses stand for predicates. Here's a real example of Resolution (Gr1):

clause 1: (The sun is shining) or (I will take my umbrella)
clause 2: (The sun is not shining)

inference: I will take by umbrella

The predicate "The sun is shining" is positive in clause 1 and negative in clause 2 and so can be cancelled.

Unlike mathematics, if two positive terms of one clause appear in the second clause negated, you cannot cancel them both. For example from the two clauses:

$P \vee Q \vee R \vee (\sim S)$
$(\sim P) \vee (\sim Q) \vee T \vee U$

you may *NOT* infer

$R \vee (\sim S) \vee T \vee U$

If P is true and Q is false, then both input clauses are true without regard to the other terms in the clauses.

2.3: Incorrect Rules of Inference

Applying rules of inference to statements claimed to be true (and actually true) can only lead to true conclusions. Thus, if something known to be false is inferred, one of the known facts is actually false.

Most human reasoning is less formal than this and involves methods than can be proven incorrect. In practice, they are correct often enough to be valuable tools. Here are some incorrect rules of inference:

1. Abduction

If P implies Q and Q is true, then P is true.

From a logic point of view, this is nonsense because, from something false, you can infer anything at all including something that is true. "If 2 is an odd number, then the pope is Catholic" is a correct implication. The conclusion is true (let's assume) but that does not make 2 an odd number. Nonetheless, Abduction is the basis of medical diagnosis. For example:

Patient has cancer implies symptom 1

If the patient exhibits symptom 1, the doctor may deduce that he has cancer. Of course, he may be wrong. If in addition:

Patient has cancer implies symptom 2
Patient has cancer implies symptom 3

and the patient has all three symptoms, the doctor can diagnose with greater confidence. He might still be wrong. Abduction might be called "inference by best explanation". Of course, if cancer has a unique set of symptoms and the patient has them all, a correct conclusion can be reached. Complete knowledge is the exception not the rule.

2. Induction

If Q is true for every instance of Q known, then Q is true for all instances.

If you lived in an isolated village in Africa, you might notice that Q is human and Q has a black face. Also R is human and R has a black face. The conclusion is that "all humans have black faces". This is, of course, not true. When a white man shows up, the first conclusion might be "This person is not human -- he's a great white god" or "he's an animal to be eaten". Eventually, however, it becomes clear that the original inductive conclusion is not true.

Nonetheless, induction is the basis of learning. A child quickly learns that touching a hot stove burns him. He will conclude that this is always true rather than keep checking the hypothesis. When adults apply induction, the result is often called a law: "What goes up must come down" (a paraphrase of Issac Newton). The builders of the Voyager space craft might disagree.

3. Default reasoning

If you can't infer Q, infer $\sim Q$

This is like saying you are innocent if you can't be proven guilty. This is incorrect unless you have complete knowledge. Of course, if you know everything and reason perfectly and can't infer Q, then $\sim Q$ must be true. If Q is not true, then the attempt to infer it might not terminate.

2.4: Variables in Logic

The logic statements seen so far give you ways to express relations about particular objects. For example, you can say:

If 32 is divisible by 4 then 32 is divisible by 2
If 34 is divisible by 4 then 34 is divisible by 2
If 36 is divisible by 4 then 36 is divisible by 2
etc. for infinitely many statements

Writing these as implications using the notation of logic each reads:

divisible_by_2 (32) ← divisible_by_4 (32)

or

divisible_by_2 (32) ∨ ~divisible_by_4 (32)

again with an infinite number of similar statements.

** Universal Quantification

Universal Quantification gives a way to write a more general statement:

if N is divisible by 4, then
N is divisible by 2

where N is called a logic variable and replaces universal quantification.

This is written as an implication as follows:

divisible_by_2 (*N*) ← divisible_by_4 (*N*)

or

divisible_by_2 (*N*) ∨ ~divisible_by_4 (*N*)

A logic variable is essentially a place holder for a value. It is unlike a variable in a programming language because it need not have a value to be used. In a particular instance, you may stick in any value for *N* everywhere it occurs and if it is divisible by 4, then it is divisible by 2. This one statement replaces a countably infinite set of statements. This paper from this point on follows PROLOG conventions where any name starting in uppercase is taken to be a logic variable. This convention is not standard so the *APL*2 algorithms presented later on use a leading '∆' to mean a logic variable. Therefore, there are two conventions for logic variables in this paper -- upper case when looking at logic statements and '∆' when looking at *APL* programs. Appendix 1 shows the function *ENCODE* which is used to implement the logic variable name scheme.

** Existential Quantification

Existential Quantification gives a way to say that at least one substitution for a logic variable yields a true statement. For example you say:

There exists an *X* such that
X is president of the United States.

In this case, since there is a person *X*, it's OK to give him (or her) an arbitrary constant name (say "wdjx" or "reagan"). This, then, becomes an assertion of fact:

president_of_USA (reagan)

It doesn't matter what name you give it. An arbitrary character string will do so long as it is unique and used wherever you intend to refer to the intended object.

** Inference with Logic Variables

The rules of inference need to be extended to allow statements that contain variables. For example, "Modus Ponens" says "given *P* implies *Q* and *P*, then *Q*". Suppose that you've been given the two statements:

$P1$
$P2$ implies Q

where $P2$ and $P1$ contain variables. You cannot, it would seem, infer anything because $P2$ and $P1$ are not the same so Modus Ponens does not apply. Modus Ponens is extended to include variables as follows:

> Given "$P1$" and "$P2$ implies Q", if you can find substitutions for variables in $P1$ and $P2$ that make them the same, then infer Q' which is Q with the same values for variables.

This matching process is called unification and is discussed in part 4. $P1$ and $P2$ unify if they can be made to match by giving values to variables.

For example,

clause 1: divisible_by_4 (32)
clause 2: divisible_by_2 (N) ∨~ divisible_by_4 (N)

The only predicate in clause 1 matches the right hand predicate of clause 2 if you substitute for N the value 32. Thus you may infer:

divisible_by_2 (32)

which is the other predicate in clause 2 using the same value for the logic variable.

2.5: General Resolution

Resolution is the more general inference rule and its application is extended to clauses with logic variables in the same way.

given $P1 \vee Q \vee (\sim R)$
and $(\sim P2) \vee (\sim S) \vee T$

and values for variables so
$P1$ and $P2$ unify,

infer
$Q' \vee (\sim R') \vee (\sim S') \vee T'$

where Q', R', S', and T' come from Q, R, S, and T with the same values substituted for variables.

This more general resolution rule is the basis for the logic programming search programs discussed in part 4.

A word of caution is needed on the use of variables. A variable is meaningful only inside one logic statement. If a second statement contains a variable, it is a different variable even if it has the same name. The algorithms avoid this possible confusion by renaming all variables with unique names. (see *VRENAME* in Appendix 1).

2.6: Chaining.

A chain, in logic, is a set of implications that connect two clauses together.

If you are given a set of facts and wish to prove the goal S, there are two ways to discover the chain: starting from the facts, or starting from the conclusion:

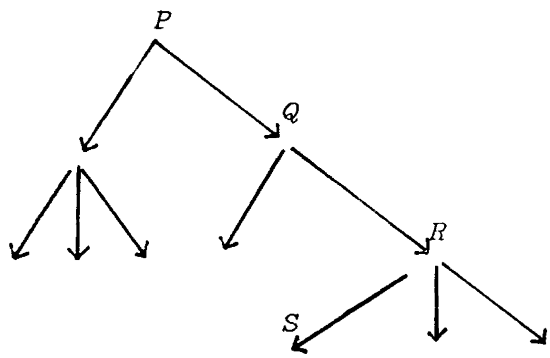
** Forward Chaining

The most obvious way to arrive at some goal given a set of facts is to make implications and watch for the goal to appear. For example, given "P" and the implication "P implies Q" you can deduce "Q". The following shows the application of two more implications leading to S (remember that P implies Q is written $Q \vee (\sim P)$):

P and $Q \vee (\sim P)$ gives Q
Q and $R \vee (\sim Q)$ gives R
R and $S \vee (\sim R)$ gives S

This chain shows that P implies S.

In general, many things can be inferred from the known facts that won't lead to the goal. Thus, the search for the goal might look like this where the arrows mean implications:



This leads to identification of the desired chain along with a lot of unwanted implications. All implications are true and could be permanently added to the database. See (Fo1) for an excellent application of forward chaining in a expert system.

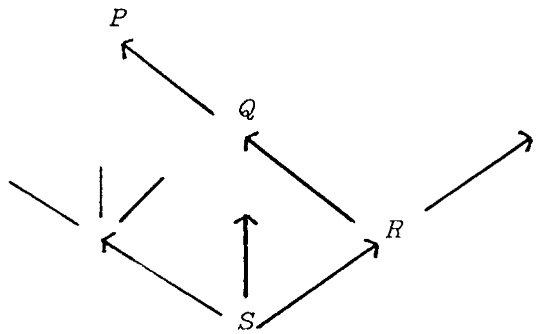
The Expert System Environment (ESE) command DISCOVER requests forward chaining.

** Backwards Chaining

Given a goal, a chain connecting to it can be discovered by denying the goal (~goal) and looking for a contradiction. Thus, if you want to prove S, you start with ($\sim S$) (the denial of the goal). If you know that R implies S, then clearly R must be denied also. ($\sim R$) is the denial of a sub-goal.

$\sim S$ and $S \vee (\sim R)$ gives $\sim R$
$\sim R$ and $R \vee (\sim Q)$ gives $\sim Q$
$\sim Q$ and $Q \vee (\sim P)$ gives $\sim P$
$\sim P$ and P gives a contradiction
therefore $\sim S$ is false and S is true

This, again, leads to a tree of implication because, in general, more than one implication may lead to the desired conclusion:



Note that the final chain is the same (in this case) but different extra work is done. Most deductive systems use backwards chaining. Only implications along the final chain are true and can be added to the rule database. The others are not known to be true. But, at least, only implications that potentially lead to the desired conclusion are used.

The Expert System Environment (ESE) command DETERMINE requests backward chaining.

** Summary of Chaining

Chaining is like finding a route on a downtown map of a large city. To plan a route from A to B, you could start at A and find some intersections reachable from A. Then find some intersections reachable from those. You eventually reach the destination and have determined a route. This is forward chaining.

Instead, you could start from B, the destination, and identify some intersections which lead to B. Then find some intersections leading to those until A, the starting point, is reached. This is backwards chaining.

Whether you use forward chaining or backward chaining can depend on the kind of rules in the knowledge base. If from each spot in the search tree, only a few places can be reached (small fan out) but many rules can reach the same place (large fan in), then forward chaining is probably more efficient. If the opposite is true, then backward chaining is probably more efficient.

Here's a summary of the chaining rules:

1. Forward: P and $Q \vee (\sim P)$ gives Q

2. Backward: $(\sim Q)$ and $Q \vee (\sim P)$ gives $\sim P$

Surprisingly, these can be written as one rule by making some substitutions:

1. Forward: substitute $\sim A$ for P and B for Q

   $(\sim A)$ and $B \vee (\sim\sim A)$ gives B

2. Backward: substitute A for Q and $\sim B$ for P

   $(\sim A)$ and $A \vee (\sim\sim B)$ gives $\sim\sim B$

These each simplify to:

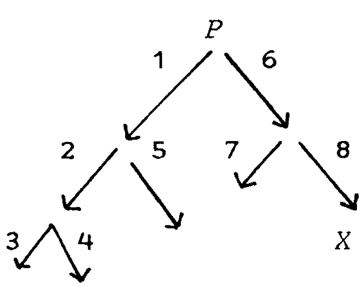
$(\sim A)$ and $A \vee B$ gives B

2.7: Search strategies

In either forward or backward chaining, a practical decision must be made concerning which implications to try before which others. Two simple strategies are called depth first search and breadth first search. They can be contrasted by looking at forward chaining.

** Depth First Search

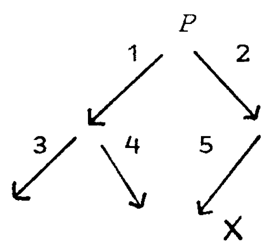
If there are two implications to be tried, then every possible chain arising from the application of one of them is tried before any application of the other. Here's a tree showing the order in which implications will be tried:



If you are trying to reach X, the order of implications makes a significant difference in the amount of work to be done. Also, if the path starting with 1,2,3 went on infinitely long, a depth first search would not find the chain to X even though it existed.

** Breadth First Search

If there are two implications to be tried, then the second is tried after the first but before any implications following from the first. Here's a tree showing the order in which implications will be tried:



In this case, search stops because *X* was found. In general, no one search is better than the other except that given a finite number of rules, breadth search will always find a chain if one exists.

** Refined search strategies

Much of the challenge in logic programming is to find better search strategies that use some knowledge of the situation to make smarter choices of what to try next. Understanding search strategies and having control of the strategy is vital. If, in following a map, you start a depth first search moving east but your destination is west, you'll waste a lot of time and effort before trying the next deep search which will also probably be wrong.

The breadth first search is at least bounded -- you'll spread out in a radius about the starting point (forward chaining) or the destination (backwards chaining) and eventually find the other point.

In this example, a better search strategy could be "move first in a direction that gets you closer in distance to the other point". By computing a "figure of merit" with each possible implication, you can choose an apparent best next choice and significantly reduce the amount of work done. Even a very bad figure of merit can lead to a vast improvement in efficiency.

This would be neither a depth first nor a breath first search but rather a combination of them.

Thus, the amount of work to be done can be reduced by applying some knowledge specific to the problem to be solved. Such strategies influence the efficiency of the algorithms but not their correctness. Other techniques, like artificially stopping what appears to be a fruitless search, could affect the correctness and cause you to fail to prove a provable conclusion. The improvement in efficiency, however, can be dramatic enough to account for the difference between a practical algorithm and an impractical one.

Part 3: APL2

This part introduces the main features of *APL2* with emphasis on the facilities that are actually used in the AI algorithms. No attempt is made to present a tutorial covering the whole language. The expressiveness of *APL2* as compared to LISP is investigated with an example.

APL2 has three kinds of objects - arrays, functions, and operators. Arrays are the data, functions are what you do to data, and operators are what you do to functions. Each will be discussed briefly by example.

3.1: APL2 Data Structures

This section will describe how *APL2* represents individual pieces of data and collections of data. There are only two kinds of data in *APL2* -- numbers and characters. A number may be logical (0 or 1), integer (1234), scaled (1*E*10), or complex (2*J*3) but these are not separate data types. The logical numbers 1 and 0 are used to represent "true" and "false" respectively. A character may be an ordinary character ('a') or an extended character like a Japanese character.

A collection of data in *APL2* is called an array. An array in *APL2* is a rectangular collection of numbers and characters where at each point in the rectangle is a single number, a single character, or another array.

Here's a 3 by 3 array of numbers (a matrix):

```
      3 3⍴ 23 1 123E20 1 0 124E15 ¯1 1 1E11
23 1 1.23E22
 1 0 1.24E17
¯1 1 1.00E11
```

The symbol ⍴ is the "reshape" function. It means rearrange the items on the right into a collection having three rows and three columns.

Here's a 3 by 4 array with numbers and characters:

```
      3 4⍴'INIT' 'B' 'TITLE' '' 'C' 'D' 55 0 'E' 'F' 66 1
INIT B TITLE
C    D    55 0
E    F    66 1
```

Here's a 3 by 3 array with a matrix at each spot:

```
      3 3⍴ ⊂2 2⍴1 0 0 1
 1 0   1 0   1 0
 0 1   0 1   0 1

 1 0   1 0   1 0
 0 1   0 1   0 1

 1 0   1 0   1 0
 0 1   0 1   0 1
```

The symbol ⊂ is the "enclose" function. It means package the 2 by 2 array into a scalar - an array with no shape which can be thought of as an atom. The scalar is then repeated nine times to get the three by three array.

In general, at any spot in an *APL*2 array, it is OK to have any other array.

Here's a vector of characters:

```
      'sten'
```

Since this is an array, it may be an item of another array:

```
      'sten' 'isa' 'man'
```

This is a three item vector of character vectors and is a possible representation of a predicate in logic.

Names are associated with arrays by use of the assignment arrow (←):

```
      A←'sten' 'isa' 'man'
```

Such a name is called a variable (not to be confused with a logic variable which may not have a value).

Mention of the name of an array produces the corresponding value:

```
      A
'sten' 'isa' 'man'
```

3.2: APL2 Functions

*APL*2 functions take an array (monadic function) or two arrays (dyadic function) and produce a new array as a result. You've already seen the monadic function "enclose"(⊂) and the dyadic function "reshape" (⍴).

** Monadic Functions

The "shape" function (⍴) returns the number of items along each axis of an array:

```
      ⍴'sten' 'isa' 'man'   ⍝ count items
3
```

The "first" function (↑) returns the leading item from an array. It is like CAR in LISP:

```
      ↑'sten' 'isa' 'man'   ⍝ select first item
sten
      ⍴↑'sten' 'isa' 'man'   ⍝ length of first item
4
```

The "depth" function (≡) returns an integer that indicates the level of nesting of an array. A single number or a single scalar (a simple scalar) has depth 0; an array of single numbers or characters has depth 1; an array containing at least one depth 1 array (and none deeper) has depth 2.

```
      ≡'s'                   ⍝ depth of a single char
0
      ≡'sten' 'isa' 'man'    ⍝ depth of vect of char strings
2
      ≡↑'sten' 'isa' 'man'   ⍝ depth of first item
1
```

** Dyadic Functions

The "drop" function (↓) deletes the requested number of leading items. With a left argument of 1, it is like CDR in LISP.

```
      1↓'sten' 'isa' 'man'    ⍝ select all but first item
isa man
```

The "index of" function (⍳) searches in the left argument for occurrences of items from the right argument and reports the index position at which each is found or 1+⍴left if an item is not found:

```
      'sten' 'isa' 'man' ⍳ 'sten' 'other'   ⍝ find index position
1 4
```

The "match" function (≡) returns 1 if and only if its two arguments have the same value and structure:

```
      'sten' 'isa' 'man' ≡ 'sten' 'man' 'mortal'
0
```

Note that "match" and "depth" share the same symbol. The presence or absence of the left argument determines which is intended.

** The Execute Function

*APL*2 has one somewhat unusual function called "execute" (⍎). Here is a character string containing three characters:

```
      '2+3'          ⍝ character string
2+3
```

The "execute" function causes a character string to be treated as an expression to be evaluated:

```
      ⍎'2+3'         ⍝ evaluate char string
5
```

Given a character constant, it's not so exciting to see it executed as an expression. Any program in any language starts out as character strings which get compiled or interpreted. More interesting is the case where the character argument to "execute" is the result of a computation. Here, the character '3' is joined to the end of the variable *E*:

```
      E←'2+'         ⍝ char string
      E,'3'          ⍝ create new char string
2+3
      ⍎E,'3'         ⍝ evaluate new char string
5
```

Thus, using execute you can construct, under program control, new *APL*2 expressions and cause them to be evaluated. This is especially significant when doing symbolic computations. For example, if you have a variable *A* having some array as value, a mention of the name is equivalent to a mention of that value:

```
      A←2 2⍴'APL' 'TWO'   ⍝ matrix of two char strings
      A                   ⍝ mention name gives values
 APL TWO
 APL TWO
```

If you want to deal with the name *A* rather than its value, you just use the character string '*A*' instead.

```
      B←'A'               ⍝ B is string with name of A
```

Now the variable *B* contains the name of the variable *A*.

```
      B                ⍝ mention of B gives value
A
```

If at some time you want to know the value instead of the name, "execute" is used:

```
      ⍎B               ⍝ same as ⍎'A'
 APL TWO
 APL TWO
```

If you have a variable whose value is a character string, you can determine if the character string is the name of a variable by requesting its "name class" (⎕*NC*). Interesting values that ⎕*NC* can return are:

```
¯1 - not a name
 0 - no value
 2 - variable
 3 - function
 4 - operator
```

```
      A←2 2⍴'APL' 'TWO'
      ⎕NC 'A'             ⍝ A is a variable
2
      B←'A'
      ⎕NC B               ⍝ A is a variable
2
```

In this last example, *B* is a variable but it's value (which is '*A*') is the argument to ⎕*NC*

In the following, assume that the name *W* has not been given a value:

```
      ⎕NC 'W'          ⍝ W has no value
0
      ⎕NC 'XYZ' 'SA'  ⍝ argument is not a name
¯1
```

The algorithm for Unification will use this scheme for variables in logic. Each will be represented as real *APL2* variable (name class 2) when the logic variable has a value and as a name with no value (name class 0) when the logic variable does not have a value. To see how they are used, see the *EVAL* operator in the next section.

3.3: APL2 Operators

Operators modify the behavior of functions. They apply to all functions, even user defined programs and programs written in other languages (FORTRAN, ASSEMBLER, etc.).

** The Each Operator

The "each" operator (¨) applies an arbitrary function to each item of its argument(s) and returns one item of its result per application.

Here are some pictures that demonstrate the application of "each":

```
      A←I J K
      B←P Q R
```

monadic function "fn"

fn¨*A* ←→ fn¨

I	*J*	*K*

←→

fn *I*	fn *J*	fn *K*

dyadic function "fn"

```
A fn¨ B ↔ | I | J | K |   fn¨   | P | Q | R |

          | I fn P | J fn Q | K fn R |
```

Thus, in some sense, dyadic "each" takes a function and distributes it inside the argument arrays. The function operand of the operator therefore sees arrays of one less depth than it would without "each". The function is paired with corresponding items one from each argument. The number of results is the same as the number of arguments.

```
      'sten' 'isa' 'man' ≡¨ 'sten' 'man' 'mortal'
1 0 0
      ρ¨'sten' 'isa' 'man'
4 3 3
```

An important special case of dyadic "each" occurs where one argument is a scalar. For example, let *S* be a scalar that contains *I* as its only item:

```
    S←⊂I            ⍝ construct scalar containing I

S fn¨ B ↔ | I | I | I |   fn¨   | P | Q | R |

          | I fn P | I fn Q | I fn R |
```

Thus, "each" applies a function between corresponding items one from each argument. To apply a function with a given left argument *X* to each item of the right argument *Y*, just enclose the left argument:

```
      (⊂X) fn¨ Y
```

A scalar as a right argument yields a similar expression:

```
      X fn¨ (⊂Y)
```

The "each" operator (¨) is one of two important primitive operators that will be used in the AI algorithms that follow. Recursion that is not replaced by parallel operations will normally be done in some function "fn" which after finding the data *A* more complicated than it wishes to handle, will, instead, do a recursive simplification by applying itself to each item of the data fn¨*A*.

** The Outer Product Operator

"Each" is only one of many useful ways to combine two argument lists. The primitive operator "outer product" is like "each" except that it applies a function to all combinations of items one from the left and one from the right. It can be pictured like this:

```
    A← I J K
    C←X Y
```

C ∘.fn *A* ↔

X	*Y*

∘.fn

I	*J*	*K*

X fn *I*	*X* fn *J*	*X* fn *K*
Y fn *I*	*Y* fn *J*	*Y* fn *K*

Much like "each", "outer product" takes a function and distributes it inside the argument arrays and the function sees arrays of one less depth than if "outer product" had not been used. The only difference is that "each" applies the function to corresponding items from the argument and "outer product" applies the function between all possible pairs.

For example:

```
     'sten' 'isa' 'man' ∘. ≡ 'sten' 'man' 'mortal'
1 0 0
0 0 0
0 1 0
```

Each item of the left argument is "matched" against each item of the right argument. The row index of the result says which item of the left argument was used and the column index says which item of the right argument was used.

Whenever an algorithm calls for doing something in all combinations, "outer product" is probably the solution.

3.4: User Defined Control Structures

The *APL2* operators allow writing expressions in a functional style. There are, however, only a few primitive *APL2* operators. Identifying new primitive operators is a possible area for future extension of the language.

User defined operators provide a way for the *APL2* programmer to add his own control structures to the language and therefore extend the possibilities for functional programming. The defined operators themselves are often written in a procedural style.

** TRUE and UNTIL operators

As an example, suppose you want to determine the truth of some goal statement and there are one hundred facts to check against. Assume you have a function called *CHECK* which given the goal and a fact from the database returns a 1 or 0 depending on whether the given fact proves the goal statement or not. You could enter a set of statements like this to prove the goal:

```
      STMT CHECK RULE1
      STMT CHECK RULE2
      . . .
```

However, if the rules are kept in a vector called *DATABASE*, you could just enter:

```
      (⊂STMT) CHECK¨ DATABASE
0 0 0 1 0 0 1 1  . . .  0 0 0
```

This will do the expected operation resulting in a hundred item vector of zeros and ones. Notice the use of a scalar left argument so "each" applies *CHECK* between *STMT* and each rule in the database. If you want to know every way in which truth can be proven, then this expression is an elegant solution. On the other hand, if you only want to know if the statement can be proven, then the expression is still correct but computes a lot of unneeded results because it continues to apply *CHECK* even after a proof has been found.

What is needed is an "each" that will quit after a proof is discovered. *APL2* does not have such an operator but you can write one. Here's one possible definition:

```
     ∇Z←L(F TRUE)R;I          ⍝ "each" that quits on true
[1]   →(0≠⍴⍴L)/L1             ⍝ branch L not scalar
[2]   L←(⍴R)⍴L                ⍝ extend scalar left
[3]  L1:→(0≠⍴⍴R)/L2           ⍝ branch R not scalar
[4]   R←(⍴L)⍴R                ⍝ extend scalar right
[5]  L2:Z←I←0                 ⍝ initialize result and counter
[6]  LP:→(I≥⍴R)/0             ⍝ exit when counter exceeds length
[7]   →(1≡Z←(↑I↓L)F(↑I↓R))/0  ⍝ exit when result is 1
[8]   →LP I←I+1               ⍝ continue
```

[1] through [4] only check for a scalar argument and, if found, extend it to be the same length as the other argument. The real logic starts on [5] where the result is set to false. This result is only returned if the arguments are empty. [6] through [8] implement a loop which applies the argument function *F* between corresponding items of the arguments. The branch on line [7] causes an exit if a 1 (true) is ever returned. If the loop counter ever exceeds the argument length, line [6] exists returning a result of 0 (false).

The *TRUE* operator is defined in a procedural style but is used in a functional style:

```
     (⊂STMT) CHECK TRUE DATABASE
1
```

This expression terminates as soon as any way to prove *STMT* is discovered.

If you want to terminate as soon as a false is discovered, you could write a *FALSE* operator. It would be exactly like *TRUE* except the one in [7] would be a zero. This suggests a more general operator which takes as an argument the value that causes termination:

```
     ∇Z←L(F UNTIL THIS)R;I       ⍝ EACH THAT QUITS ON TRUE
[1]   →(0≠⍴⍴L)/L1             ⍝ branch L not scalar
[2]   L←(⍴R)⍴L                ⍝ extend scalar left
[3]  L1:→(0≠⍴⍴R)/L2           ⍝ branch R not scalar
[4]   R←(⍴L)⍴R                ⍝ extend scalar right
[5]  L2:Z←I←0                 ⍝ initialize result and counter
[6]  LP:→(I≥⍴R)/0             ⍝ exit when counter exceeds length
[7]   →(THIS ≡Z←(↑I↓L)F(↑I↓R))/0 ⍝ exit when result THIS
[8]   →LP I←I+1               ⍝ continue
```

Now *TRUE* can be written:

```
     (⊂STMT) CHECK UNTIL 1 DATABASE
1
```

and *FALSE* can be written:

```
      (⊂STMT) CHECK UNTIL 0 DATABASE
0
```

** PARALLEL Operator

If you have a truly parallel machine available, you might want an "each" like operator that passes each computation to a different computing engine. *APL*2 does not currently run on such machines but you could pass sets of computations to different real machines. Given a set of personal computers, this might even be practical.

Because *APL*2 makes you think in a parallel array fashion, it is likely that you will discover situations where parallelism can be exploited.

** DEPTH Operator

The operators "each", "outer product", *TRUE*, and *UNTIL* all operate on the items of nested arrays or, in some sense, one level down in the structure. Suppose you have the following two item nested vector:

```
      V←'relate' ('parent' ('ΔX' 'sue') 'sue')
```

The first item is a six item character vector and the second is a three item vector of vectors. This may or may not represent a statement in logic. Just think of it as nested data.

Suppose you want to know the length of each word in the structure. No operator you've seen so far could compute it. "Shape" (⍴) will tell you it's a two item vector and say nothing about the shape of the words. "Shape each" does a little better:

```
      ⍴¨V←'relate' ('parent' ('ΔX' 'sue') 'sue')
 6  3
```

At least you get the shape of one of the words. What is needed is an "each" like operator that doesn't stop after one level into the array but continues until it gets to a word. A word (i.e. a character vector) is a depth 1 array so you can write an operator that looks for a depth 1 array and if the data is deeper than that, it applies "each" until a depth 1 array is found. Here is one way to write such an operator:

```
     ∇Z←(F DEPTH1) R          ⍝ apply F at depth 1
[1]   →(1<≡R)/RECUR           ⍝ recur if depth > 1
[2]   Z←F R                   ⍝ apply F to depth 1 R
[3]   →0                      ⍝ exit
[4]  RECUR:Z←(F DEPTH1)¨R     ⍝ apply F to items of R
```

[1] branches if the depth of the argument is greater than one. [2] applies the function to an array known to be depth 1 or less. [4] uses "each" to dig one level deeper into the array eventually reaching a level which is depth 1 or less array.

```
      ⍴ DEPTH1 V
 6  6   2 3  3
```

and this result has the same tree structure as *V*. The words are replaced by their shapes. If you want a simple vector of shapes, the function "enlist" always returns a simple vector.

```
      ∊⍴ DEPTH1 V
6 6 2 3 3
```

** EVAL Function

Given the vector *V* from above, suppose that any vector starting with the character '∆' represents a logic variable. One of the tasks of a logic program is to take such a statement and produce a new one that represents the statement with values substituted for variables. Here is a function that will do the substitution on one logic variable. Remember that a logic variable without a value is represented by an *APL*2 name without a value (name class 0) and a logic variable with a value is represented by an *APL* name with a value (name class 2):

```
     ∇Z←EVAL1 R          ⍝ evaluate logic variables in R
[1]   Z←R                ⍝ initialize result
[2]   →(~'∆'≡↑R)/0       ⍝ exit if not a logic var
[3]   →(2≠⎕NC R)/0       ⍝ exit if no value in APL variable
[4]   Z←⍎R               ⍝ replace variable with its value
```

[1] sets the result to the argument.

[2] exits if the name is not a logic variable.

[3] exits if the name is an *APL* variable with no value.

[4] returns the value of the variable

Here are some applications of the *EVAL*1 function:

```
      ΔX←'mother'
      EVAL1 'sten'
sten
      EVAL1 'ΔX'
mother
```

*EVAL*1 doesn't do what is required if there is more than one name:

```
      V←'relate' ('parent' ('ΔX' 'sue') 'sue')
    EVAL1 V
relate  parent  ΔX  sue  sue
```

V is a structure that contains many names at various different depths. You need to apply *EVAL*1 to each name in *V*. The operator *DEPTH*1 will do that:

```
      ΔX←'mother'
      V←'relate' ('parent' ('ΔX' 'sue') 'sue')
      EVAL1 DEPTH1 V
relate  parent  mother  sue  sue
```

This form of substitution is not powerful enough to handle the general case because the value of a variable Δ*X* may contain a complicated structure which itself contains a variable. In the following, the variable Δ*X* contains a reference to logic variables Δ*Y* and Δ*Z*:

```
      ΔX←'abc' 'ΔY' 'ΔZ'
      ΔY←'def'
      V←'relate' ('parent' ('ΔX' 'sue') 'sue')
```

Now *EVAL*1 will not complete the substitution.

```
      EVAL1 DEPTH1 V
relate  parent  abc ΔY ΔZ  sue  sue
```

A more general function will do substitutions in any substituted values as well. Here is a more general function:

```
     ∇Z←EVAL R               ⍝ evaluate logic variables in R
[1]   Z←R                    ⍝ initialize result
[2]   →(~'Δ'≡↑R)/0           ⍝ exit if not a logic variable
[3]   →(2≠⎕NC R)/0           ⍝ exit if no value in variable
[4]   Z←EVAL DEPTH1 ⍎R       ⍝ replace variables with values
```

```
      EVAL DEPTH1 V
relate  parent  abc def ΔZ  sue  sue
```

Now ΔY is given the correct value. ΔZ does not have a value and so is not altered.

This function will be used for doing substitutions in the algorithms that follow.

You could avoid using real *APL*2 variables to represent logic variables by storing the names of variables and their values in an array -- a vector of pairs of a two column matrix:

```
    VSUBS←('ΔX' ('abc' 'ΔY'))('ΔY' 'def')
            or
    MSUBS←⊃VSUBS
```

(The function "disclose" (⊃) turns a vector of vectors into a matrix.) The *EVAL* function could then search one of these arrays instead of doing an "execute" (⍎). For example, lines 2 and 3 of *EVAL* could be written using *MSUBS* as:

```
      →(~(⊂R)∊MSUBS[1;])/0
```

3.5: The Rosetta Stone: LISP and APL2

The history of AI has been significantly influenced by the language LISP, which was designed to express its algorithms. LISP is an extremely elegant language for stating the recursive kinds of procedures often required in the solution of AI problems.

To compare *APL*2 and LISP, a benchmark program from "The Handbook of AI" (Ba1) will be shown in both LISP and *APL*2. The program implements a deductive search routine of the following sort:

given facts:

> There is a man named Sten.
> There is a dog named Spot.

and given the general statements:

> All men are mortal.
> All dogs have a tail.

deduce the conclusion

> Sten is mortal.

The first four statements are called the database of the problem. It is an open database in that not all true statements about the subject at hand are included. Thus, even though you can conclude that "Sten is mortal" and cannot conclude "Spot is mortal", you should not conclude that it is better to be a dog than to be Sten. An example of a closed database is an airline reservation system. If you don't have a reservation that is included in the database, you don't have a reservation.

** The LISP program

The database for the LISP program is a four item list -- one item per statement. The facts are each two item lists, and the general statements are three item lists starting with the word 'ALL'.

```
 (SETQ DBASE '((ALL MAN MORTAL) (ALL DOG HAVETAILS)))
 (SETQ DBASE '((MAN STEN) (DOG SPOT)))
```

The goal (the statement to be proved) is a two item list:

```
 (SETQ STMT '(MORTAL STEN))
```

Here is a LISP program to solve this kind of problem:

```
(DEF 'PROVE
     '(LAMBDA (STMT DB)
              (FINDA DB)))

(DEF 'FINDA
     '(LAMBDA (RESTDB)
              (COND
                  ((NULL RESTDB) NIL)
                  (T (OR
                      (PROVESIT (CAR RESTDB))
                      (FINDA (CDR RESTDB)))))))

(DEF 'PROVESIT
     '(LAMBDA (AS)
              (OR (EQUAL STMT AS)
                  (AND
                    (EQUAL (CAR AS) 'ALL)
                    (EQUAL (CADDR AS) (CAR STMT))
                    (PROVE (CONS (CADR AS) (CDR STMT)) DB)))))
```

Note that the dialect of LISP used in this program has dynamic name scope so in the function PROVESIT, the name STMT has the same value that it had in the call of PROVE. This is not the case in all LISP implementations.

Here is the execution of the program:

```
 (PROVE STMT DBASE)
T
```

Here is an explanation of the evaluation of the LISP program in reverse order:

Description of PROVESIT

Given a fact (AS) (3.2), statement (STMT) is true if either of the following is true:

1. STMT is the same as the fact (3.3)

2. each of the following is true:

   a. the first word of the fact is "ALL" (3.5)

   b. the third word of the fact is the same as the first word of the statement (3.6)

   c. you can prove the constructed statement from the second word of the fact and the second word of the statement (3.7)

Here's examples where each of the possibilities achieve truth (in reverse order):

2. If fact is "ALL MAN MORTAL" and statement is "MORTAL STEN" then

   a. first word of fact is "ALL"

   b. third word of fact matches first word of statement

   c. constructed statement "MAN STEN" can be proven

1. If fact is "MAN STEN and statement is "MAN STEN" (as constructed above), then fact is the same as the statement

Description of FINDA

Given a set of facts (RESTDB), find the first of the following which is true (2.3):

1. the set of facts is empty in which case return false, the statement cannot be proved (2.4)

2. T is always true (2.5) so evaluate the following stopping as soon as one is true:

   a. prove the statement using the first fact in the database (2.6)

   b. recursively repeat the FINDA function on the database with the first fact left out

FINDA has the effect of iterating through the facts until either the statement is proved or it runs out of facts.

** The APL2 program

The LISP program can be translated into *APL*2 directly using the following correspondence:

	LISP	APL2
empty list	NULL R	0=⍴*R*
first item	CAR R	↑*R*
rest of items	CDR R	1↓*R*
first of the rest	CADR R	↑1↓*R*
first rest rest	CADDR R	↑2↓*R*
identically equal	EQUAL L R	*L*≡*R*
join two items	CONS L R	(⊂*L*),*R*

The connective logic of the LISP program is handled by the ordinary sequencing of the *APL*2 statements.

The database is again a vector of statements with each item a vector of character strings:

```
DBASE←('ALL' 'MAN' 'MORTAL') ('ALL' 'DOG' 'HAVETAILS')
DBASE←DBASE, ('MAN' 'STEN') ('DOG' 'SPOT')
```

The goal is a vector of vectors:

```
      STMT←'MORTAL' 'STEN'
```

Here's the program written in *APL*2 syntax:

```
[0]   Z←DB PROVE ST      ⍝ SIMPLE DEDUCTION
[2]   ⍝ DB ← FACTS, IMPLICATIONS
[3]   ⍝ PROVE → ST IS A FACT
[4]    Z←FINDA DB
```

```
[0]    Z←FINDA DBS
[1]    Z←0                   ⍝ ASSUME FALSE
[2]    →(0=⍴DBS)/0           ⍝ EXIT IF DATABASE EMPTY
[3]    →(Z←PROVESIT↑DBS)/0   ⍝ ATTEMPT TO PROVE WITH FIRST AXIOM
[4]    Z←FINDA 1↓DBS         ⍝ ATTEMPT TO PROVE WITH REST OF AXIOMS
```

```
[0]    Z←PROVESIT AS
[1]    →(Z←ST≡AS)/0                 ⍝ TRUE IF FACTS MATCH
[2]    →(~Z←'ALL'≡↑AS)/0            ⍝ ELSE SEARCH FOR IMPLICATION
[3]    →(~Z←(↑ST)≡↑2↓AS)/0          ⍝ AND MATCH ITS CONSEQUENT
[4]    Z←DB PROVE(⊂↑1↓AS),(1↓ST)    ⍝ AND ATTEMPT TO PROVE NEW GOAL
```

Here is the execution of the program:

```
      DBASE PROVE STMT
1
```

The execution of this program is precisely the same as the LISP program and so is not analyzed in detail.

Neither the LISP nor the *APL*2 program is particularly elegant. Both can be improved. The purpose of the exercise is only to show that a standard documented LISP program can be trivially converted to *APL*2.

Part 4: The Implementations

In this part, alternative representations for logic statements are presented. *APL2* programs that describe the important AI algorithms are developed. The programs are designed to describe the algorithms and efficiency of execution is not considered. Once the algorithms are understood, a programmer may apply his ingenuity to develop more efficient procedures. Some directions for improvement are discussed.

4.1: Representations

AI algorithms operate on facts and rules. Therefore, the representation of facts and rules becomes the first order of business.

The choice of representation heavily influences the structure of the algorithms and vice versa. The representation ultimately chosen for logic statements in this paper is influenced by the properties of the Resolution algorithm.

By far the simplest representation of a predicate would be simply to represent it as a long character string.

```
'sten is a man'
```

This representation is not suitable for use with an algorithm because it does not distinguish the relevant parts of the predicate (the relationship and the objects) from the irrelevant parts (the letters making up the words). It is not apparent that 'sten' is an indivisible subset of the vector.

A more reasonable representation is achieved by using nesting to hide the irrelevant structure of a statement. The predicate at hand expresses a relationship between two things -- 'sten' and 'man'. This may be represented as a three item vector of the three entities -- the relationship and the two objects related:

```
'sten' 'isa' 'man'
```

This array is a three item vector of vectors and only people and the programs they write interpret this as an assertion of some relationship ('isa') between two ideas ('sten' and 'man'). Any arrangement of these three items is suitable as a representation of the statement. The one above is called infix because the relationship is in the middle. Since in general a relationship could apply to more than two things, most logic systems use prefix notation putting the relationship first. Either of the following two representations is a reasonable prefix representation:

```
'isa' 'sten' 'man'
'isa' ('sten' 'man')
```

The first is an N item vector containing the relationship and N-1 arguments. The second is a two item vector with the relationship as the first item and the vector of arguments as the second item. The algorithms presented in this paper will work for either choice of representation. The first is simpler and so is probably more efficient computationally.

You can choose any representation that is convenient for you. *APL*2 does not impose a representation on you. Once you have chosen a representation, however, you must use it consistently.

An argument of a predicate is called a term. A term may be a constant, a variable, or the application of a function that returns a constant term. For example, 'sten' is a constant term in the predicate:

```
mortal(sten)
```

In an effort to mimic the rules of PROLOG, any word beginning with a capitol letter is taken to be a logic variable. *X* is a logic variable term in the following predicate:

```
mortal(X)
```

Terms may be computed by functions. f(a,b) is a function in the predicate:

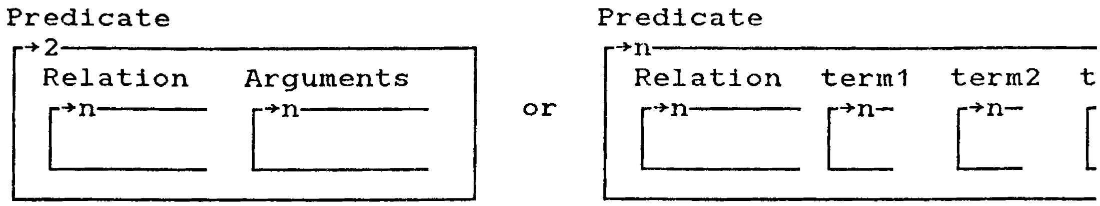
```
mortal(f(a,b))
```

This last predicate can be represented in *APL*2 as

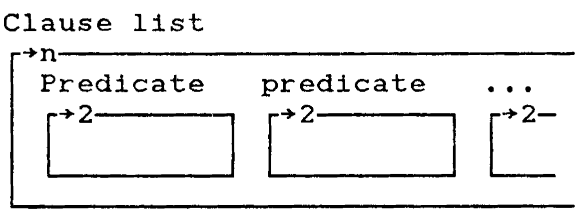
```
'mortal'('f'('a' 'b'))
```

Thus nesting of arrays is used to represent the structure of a predicate.

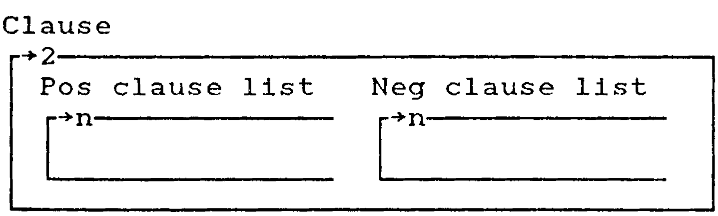
Here is a stylized picture of the structure of a predicate:



The algorithms to be discussed operate on disjunctive clauses -- predicates connected by "or" (∨). Therefore, the presence of "or" may be assumed and a clause represented as a vector of predicates. This does not, however, allow for a representation of the sign (~) that negates some predicates. Therefore, a clause is broken into two groups: those predicates not negated in the first group and those negated in the second group. Each group is called a clause list. Therefore, each clause list is a vector of predicates:

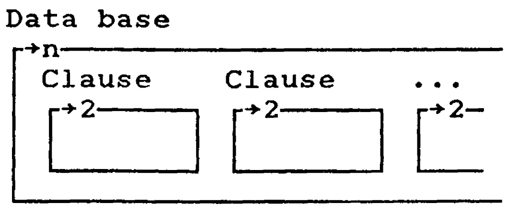


A clause is, then, represented as a vector of 2 clause lists -- the vector of the non-negated predicates and the vector of the negated predicates:



When a clause is looked at as an inference, the positive clause list is called the consequent, and the negative clause list is called the antecedent. Thus, the sign of a predicate is encoded in the structure of the array, not in the data. If a predicate is negated, it appears in the second list.

Finally, a knowledge base or a database is a vector of clauses each representing one fact or one rule. Since each statement is claimed to be true, the database may be considered an "and" (∧) of the clauses:



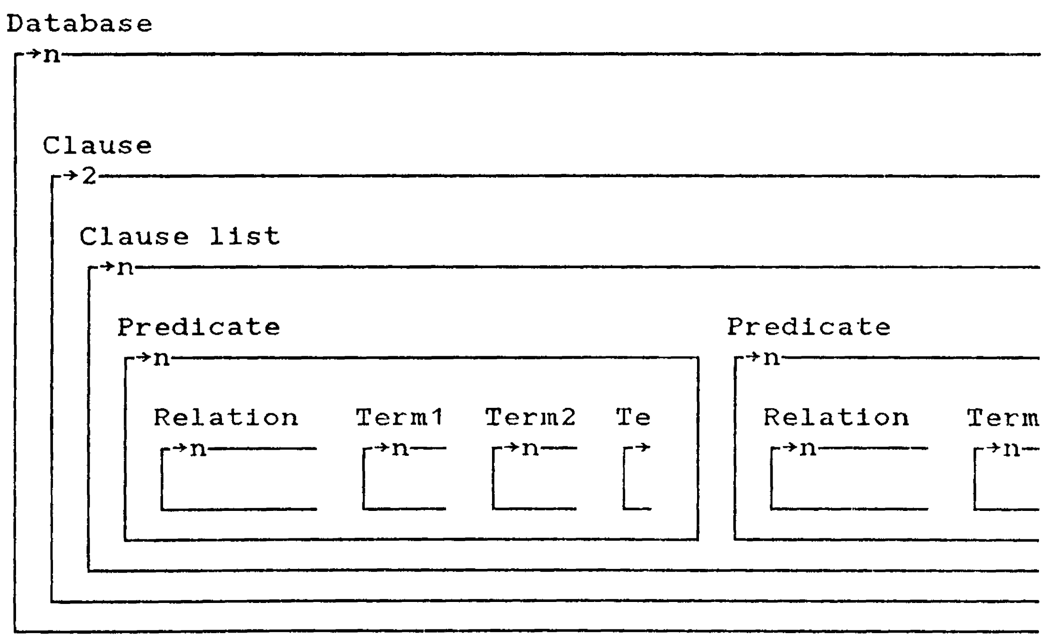
Facts are included in the database by writing them as inferences with an empty clause as antecedent (because anything infers something that is true).

Here is a summary of the resulting data structure:

- Database - an n item vector of inferences
  - Clause - a 2 item vector of clause lists
    - Clause list - an N item vector of predicates
      - Predicate - an N item vector of relation and terms
        - Relation - a depth 1 vector
        - Term - depth 1 or more

Thus a database is arbitrarily deep (depending on the depth of any functions) but is at least depth 5.

Here is a picture of part of a database



This is one of many possible data structures for the database. Even this arrangement of data could be stored using some higher rank arrays rather than vectors at each level. For example, a data base could be stored as an *N* by 2 matrix where each row represented a clause and column 1 was the positive list and column 2 the negative list. The algorithms which follow work together with the depth 5 structure so that during execution, the structure is decomposed by the normal application of *APL*2 operators.

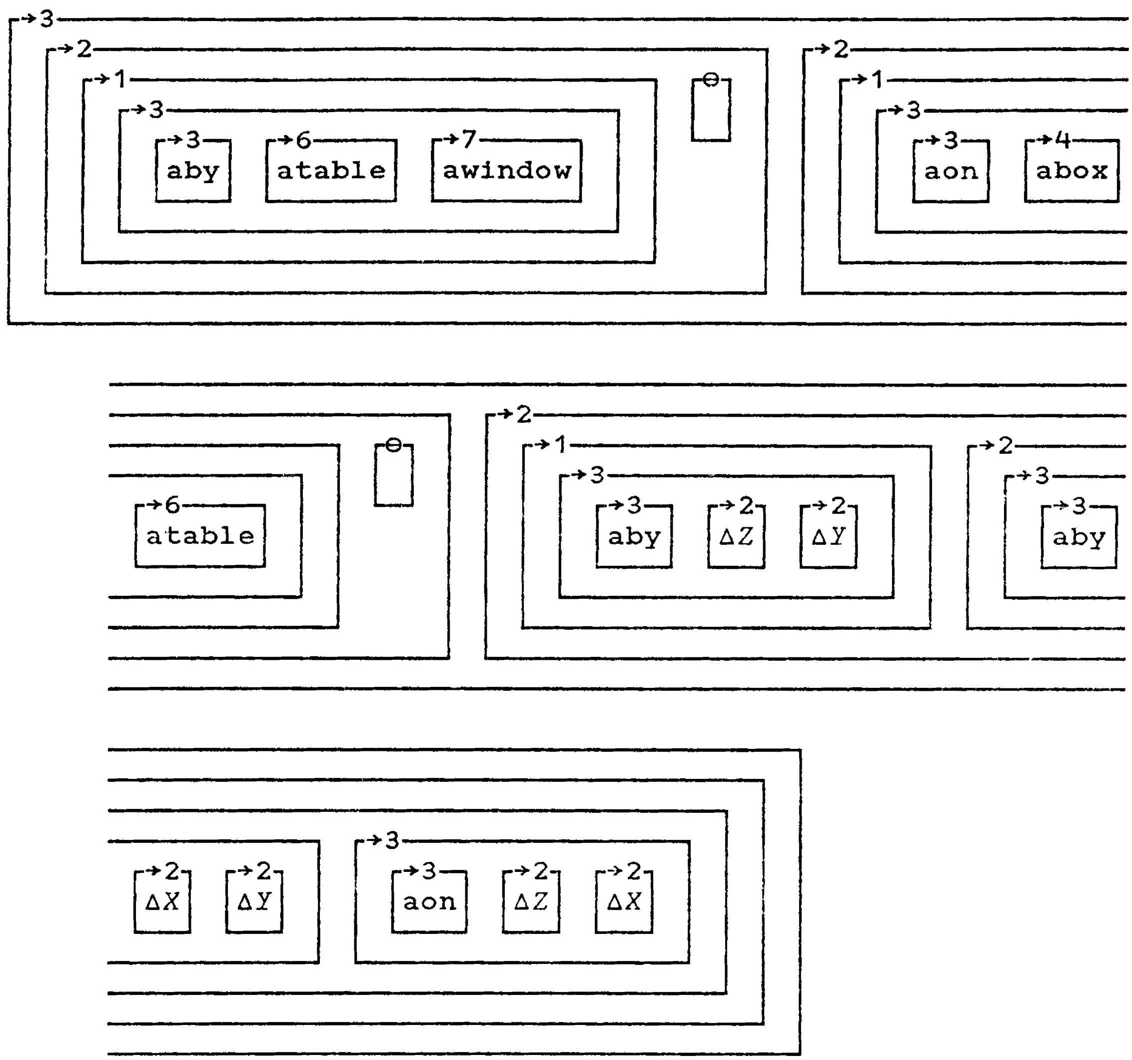
Here is an example logic problem and its representation with a depth 5 array:

> The table is by the window
> The box is on the table
> if *X* is by *Y* and *Z* is on *X*, then *Z* is by *Y*

This problem is more formally stated as follows:

```
by table window ←
   on box table ←
         by Z Y ← (by X Y) ∧ (on Z X)
```

Here is the picture of the database:



'a' is prefixed on non-variables and 'Δ' is prefixed on variables as part of conversion to internal form.

Now you might ask a question like "Is the box by the window". If this is a fact, it is represented formally as follows:

```
   by box window ←
```

Here is the representation of the goal as an *APL*2 array:

```
┌──────────────────────────────────────────────┐
│┌→2────────────────────────────────────────┐  │
││ ┌→1───────────────────────────────┐ ┌⊖┐ │  │
││ │┌→3──────────────────────────┐   │ │ │ │  │
││ ││┌→3┐  ┌→4─┐  ┌→7────┐      │   │ └─┘ │  │
││ │││aby│  │abox│  │awindow│    │   │     │  │
││ ││└───┘  └────┘  └───────┘    │   │     │  │
││ │└────────────────────────────┘   │     │  │
││ └─────────────────────────────────┘     │  │
│└─────────────────────────────────────────┘  │
└──────────────────────────────────────────────┘
```

4.2: The Unification Algorithm

Unification is the process of comparing two or more predicates to see if they are the same predicate. It is like the *APL*2 primitive "match" (≡) except that predicates containing variables match other predicates if values for the variables can be discovered that make the predicates the same.

** Examples of Unification

In principle, Unification may be applied to any number of predicates. If the predicates are the same, or can be made the same by supplying values for variables, then Unification succeeds. The *APL*2 *UNIFY* function that is described here applies to two predicates. It could be generalized to apply to more than two predicates but the generality is not needed in this paper. The function *UNIFY* produces two results returned as a two item nested vector -- 1 or 0 depending on if the statements do or do not unify and the values for variables that permitted unification.

Unification looks at two predicates and returns 1 if they match. For example:

```
      'isa' 'sten' 'man'  UNIFY  'isa' 'sten' 'man'
1
      'isa' 'sten' 'man'  UNIFY  'isa' 'sten' 'mortal'
0
      'isa' ('sten' 'man')  UNIFY  'isa' ('sten' 'man')
1
```

This last example shows that *UNIFY* is insensitive to the representation of a predicate. The extra structure causes an extra recursion but the answer is still correct.

With such constant formulas, *UNIFY* is, in fact, identical to the *APL*2 function "match" (≡). If a formula contains a variable, you may substitute for that variable to make the formulas match. In this case, *UNIFY* returns a 1 and the substitutions needed to make the formulas match are remembered:

```
      'isa' 'ΔX' 'man'  UNIFY  'isa' 'sten' 'man'
1  ΔX←'sten'
```

These two formulas match if 'sten' is substituted for 'ΔX'.

```
      'fn' 'ΔX' 'ΔY' 'man'  UNIFY  'fn' 'ΔX' 'sten' 'ΔZ'
1  ΔY←'sten'
   ΔZ←'man'
```

These match if two substitutions are made: 'sten' for 'ΔY' and 'man' for 'ΔZ'

The variables in these formulas are just place holders. Rewriting then with different variable names does not change the meaning.

```
      'isa' 'ΔX' 'man'  unify  'isa' 'ΔY' 'man'
1  ΔX←'ΔY'
```

A variable can be replaced by an entire formula in order to make two formulas match:

```
     'fn' 'ΔX' 'ΔX'   UNIFY   'fn' ('a' 'ΔY' 'c')('a' 'b' 'ΔZ')
1  ΔY←'b'
   ΔZ←'c'
   ΔX←'a' 'b' 'c'
```

The following two formulas do not unify:

```
      'ΔX' 'ΔY' 'a' UNIFY 'ΔX' 'b' 'ΔY'
0  ΔY←'b'
```

An attempt to substitute 'b' for 'ΔY' or 'a' for 'ΔY' gives formulas that don't match.

In summary, a constant unifies only with the same constant. A variable unifies with anything not containing that variable. Otherwise, an expression unifies with an expression of the same length if corresponding items unify.

** The APL2 Unification algorithm

The actual unification process is represented by the function *UNIFYA* described next. A cover function named *UNIFY* (in Appendix 1) returns the result of unification (0 or 1) and the substitutions that are implied.

Unification is applied between predicates each of which has a structure like this:

```
Predicate
┌→n──────────────────────────────
│ Relation   Term1   Term2    Te
│ ┌→n─────   ┌→n──   ┌→n──    ┌→
│ │          │       │        │
│ └───────   └────   └────    └─
└────────────────────────────────
```

Here is a description of the algorithm in words with the key piece of *APL2* notation identified. The actual *APL2* code and a more complete description of the code appears in Appendix 1. "Failure" in the following description means return a 0 (false) and "success" means return a 1 (true). Substitutions are done using real *APL2* variables.

∇$Z \leftarrow X$ $UNIFYA$ Y

[1] fail if both predicates are empty.
In fact they do unify but they are useless.

[2] substitute for any variables that have values.
*EVAL DEPTH*1

[3] if X and Y are the same, succeed.
$X \equiv Y$

[4] if neither X nor Y is a single name,
branch to *RECUR*
~1∊ ≡¨X Y

[5] if neither X nor Y is a variable, fail
'∆'∊↑

[6] make sure substitution is allowed (Occurs check)
(see Appendix 1)
X∊∊

[7] do substitution and succeed (value of Y as value of X.)
⍎X,'←Y'

[8] *RECUR*: fail if X and Y not same length

[9] unify corresponding items of X and Y
X *UNIFYA*¨ Y
X *UNIFYA FALSE* Y

While descriptive, this is not an efficient algorithm in complicated cases. A linear algorithm is discussed in (Pa1).

4.3: The Resolution Algorithm

Resolution is a rule of inference and simple cases have been discussed before. Here's an example similar to the one shown before. Given:

$P \vee Q \vee R \vee (\sim S) \vee (\sim V)$
$(\sim P) \vee T \vee U \vee (\sim W)$

you may infer

$Q \vee R \vee (\sim S) \vee (\sim V) \vee T \vee U \vee (\sim W)$

The idea is to identify terms negated in one statement and non-negated in the other and eliminate them one at a time. This operation is facilitated by representing each statement as two groups of terms -- those non-negated and those negated. Using parentheses to indicate the groupings, you may write the above statements as follows with the disjunctions ($\vee$) implicit:

$(P\ Q\ R)\ (S\ V)$ and $(T\ U)\ (P\ W)$

you may infer

$(Q\ R\ T\ U)\ (S\ V\ W)$

which is called the resolvant. Thus, the first group in each statement is the non-negated terms and the second group is the negated terms. These two groups are called the clause lists.

This form for representing clauses is particularly nice for representing an implication. Recall that the implication "P implies Q" may be written either of the following two ways:

$Q \leftarrow P$
$Q \vee (\sim P)$

When the truth of the statement

$P \vee Q \vee R \vee (\sim S) \vee (\sim V)$

is claimed, it may be separated into two groups containing positive terms and negated terms:

$(P \vee Q \vee R)\ \vee\ ((\sim S) \vee (\sim V))$

The form for implication requires a single negation. Factoring out the negation gives:

$(P \vee Q \vee R)\ \vee\ \sim(S \wedge V)$

Now it looks like an implication and may be written in the other form:

$(P \vee Q \vee R)\ \leftarrow\ (S \wedge V)$

All this shows that the two groups of terms contain the positive and the negative terms respectively. If you think in terms of $Q \vee (\sim P)$, the second group is a disjunction of negated terms. If you think in terms of $Q \leftarrow P$. then the second group is a conjunction of positive terms. This one grouped representation covers both written representations.

In concept, the simplest way to do Resolution is to select the non-negated terms of one statement, the negative terms from the other statement, and then match items from one group with items of the other group in all combinations. For the moment, let each term be represented by a single character keeping in mind that, in practice, each term may be arbitrarily complicated. Here are the statements from the above example written as *APL*2 arrays:

```
      ST1←('P' 'Q' 'R') ('S' 'V')
      ST2←('T' 'U') ('P' 'N')
```

(Note that the intent is that each of the letters in quotes is a predicate so don't interpret them as variables) The data structure for each statement looks like this:

```
     Clause
     ┌→2──────────────────────────────────────────────────────────
     │
     │  Clause list
     │  ┌→n───────────────────────────────────────────────────────
     │  │
     │  │  Predicate                          Predicate
     │  │  ┌→n──────────────────────────────┐ ┌→n─────────────────
     │  │  │                                │ │
     │  │  │  Relation  Term1  Term2  Ter   │ │  Relation  Term1
     │  │  │  ┌→n─────  ┌→n──  ┌→n──  ┌→n  │ │  ┌→n─────  ┌→n──
     │  │  │  │         │      │      │    │ │  │         │
     │  │  │  └──────   └────  └────  └──  │ │  └──────   └────
     │  │  └────────────────────────────────┘ └──────────────────
     │  └─────────────────────────────────────────────────────────
     └────────────────────────────────────────────────────────────
```

If the statements were really this simple, you could match the appropriate groups using the "outer product" operator:

```
      (1⊃ST1) ∘.≡ (2⊃ST2)
1 0
0 0
0 0

      (1⊃ST2) ∘.≡ (2⊃ST1)
0 0
0 0
```

In practice, statements may contain variables. so "match" is not enough to compare predicates -- you must *UNIFY*:

```
      (1⊃ST1) ∘.UNIFY (2⊃ST2)
1 0
0 0
0 0

      (1⊃ST2) ∘.UNIFY (2⊃ST1)
0 0
0 0
```

Each 1 in these results implies that a successful resolution can be done.

Knowing that a resolvant exists is not normally enough information. You want to know just what the new clause is and the values of the variables that permitted it to be formed. Thus, the actual *RESOLVE* algorithm must compute these resolvants and the values of variables. One way to do this is to have a procedure which given one positive predicate from one statement and one negative predicate from another statement, computes a resolvant if one can be formed. This procedure can then be applied in all combinations using two "outer products" as done with *UNIFY* above. Suppose that you have such a function called *RESOLVANT*. Here is one way to write a resolution program. The arguments are the two clauses to be resolved and so each argument is a two item vector of clause lists:

```
      ∇Z← A RESOLVE B
```

[1] apply *RESOLVANT* between positive predicates from *A* and negative predicates from *B* in all combinations (1⊃*A*) ∘.*RESOLVANT* (2⊃*B*)

[2] apply *RESOLVANT* between positive predicates from *B* and negative predicates from *A* in all combinations (1⊃*B*) ∘.*RESOLVANT* (2⊃*A*)

[3] delete non-resolutions

The description of the program is longer than the actual program which is shown in Appendix 1.

The logic of the *RESOLVANT* program is also straightforward. It is given one positive predicate from one statement and one negative predicate from the other statement. If these predicates unify, a resolvant can be formed.

Here is the logic in words:

∇Z← *AR RESOLVANT BR*

[1] if argument predicates do not unify, fail and return 0.

[2] form new inference by building its two clause lists. The positive clause list comes from joining the two positive clause lists of the input clauses and deleting the predicate that unified. The negative clause list comes from joining the two negative clause lists of the input clauses and deleting the predicate that unified.

[3] substitute for any variables that received values during unification.

If the predicates unify, this function returns the new clause and the substitutions for variables that permitted unification.

** Speeding up resolution

The functions *RESOLVE* and *RESOLVANT* describe one way of implementing resolution on the given data structure. More efficient algorithms could be developed. They would trade the descriptive elegance of the algorithms presented for better performance. Often methods for speed up involve preprocessing the knowledge base to make Resolution and Unification more efficient. (See Fo1 for a description of the RETE algorithm for speeding up pattern matching.) Preprocessing is an advantage only if the knowledge base is searched more often than updated.

Here are some ways to speed up Resolution and its application:

1. Avoid the outer product in *RESOLVE* by arranging the predicates in a clause in lexical order by the relation. Then resolution can make a linear pass through each clause and only attempt to unify predicates with the same relation.

2. When resolution produces a new clause, attempt to get a more general clause by resolving again against the input clauses. This gives simple pairwise resolution the effect of more general resolution. See Appendix 6 resolution example 4 for an example of this.

3. When trying to find a contradiction (as in the PROLOG application of resolution), apply resolution to clauses which could resolve with one of the predicates in the goal on the theory that the contradiction being sought must involve the goal to be proved.

4. Resolve clauses with a single predicate first (called a unit preference strategy). Since both the single predicate and its negation will be deleted, the result will be more general and possibly empty (a contradiction).

4.4: Solving Logic Problems

Resolution is enough to solve logic problems. Here is the database and the goal for the example discussed earlier:

```
ST1:  by table window ←
ST2:      on box table ←
ST3:       by Z Y ← (by X Y) and (on Z X)
```

The question (initial goal)

```
GO:    by box window?
```

You could write a brute force forward chaining algorithm by resolving everything with everything and watching for the conclusion to show up:

DATABASE ∘.*RESOLVE DATABASE*

If the conclusion is not reached and there are no new clauses inferred, then you have failed to prove the desired goal. If the goal is among the new things inferred, then you have succeeded. Otherwise add the new truths to the database and repeat the outer product until it either succeeds or fails. This is essentially a breadth first forward chaining and will eventually generate the result if it is true.

The function *FORWARD*1 in Appendix 1 is an implementation of this algorithm. It is, however, extremely inefficient since at each stage it repeats all the work of the previous stage. A more efficient algorithm does the first outer product but from then on only tries resolutions between the new clauses and the database. This algorithm is represented by the function *FORWARD* in Appendix 1.

Both of these programs produce the same result on the example problem. The first outer product generates these new clauses:

```
by(Z,window) ← on(Z,table)
   by(box,Y) ← by(table,Y)
```

Since the goal does not show up, these clauses are resolved again with the clauses in the database generating these new clauses:

by(box,window) ←
by(X,window) ← on(Y,table) on(X,Y)
by(Z,Y) ← by(table,Y) on(Z,box)
by(box,Y) ← by(X,Y) on(table,X)

This time, the desired goal is generated and the program stops. This particular example is not so inefficient but an even slightly more complicated example leads to the generation of many unwanted clauses.

Here is another solution to the same problem that takes advantage of the fact that you want to prove that two things are by each other and sees 'by' in the conclusion of ST3 with two variables. If you can use resolution to delete the predicates on the right of ST3 and get the right values for the variables, you can get a solution very fast.

A solution is achieved in two steps by forward chaining:

1. Positive predicate "by" in ST1 unifies with negative predicate "by" in ST3 with the substitutions:
   X←table
   Y←window

2. Resolve ST1 and ST3 on the "by" predicate giving:

   G1: by Z window ← on Z table

3. Positive predicate "on" in ST2 unifies with negative predicate "on" in G1 with the substitution:
   Z←box

4. Resolve ST2 and G1 on the "on" predicate giving:

   G2: by box window ←

Thus proving the desired goal.

In this case it was easy to see what to do. In general, it is not so easy to know which resolutions to make. What is needed is a general organized procedure that uses resolution to prove logic problems. A general scheme is not known but if the knowledge base is restricted to Horn clauses (those with one or fewer positive predicates), a general scheme is known. This scheme is the basis of PROLOG.

4.5: PROLOG-like search strategy

PROLOG-like languages approach the solution of logic problems by denying the desired goal and searching backwards for a contradiction (an empty clause). This helps to limit the amount of work done because, at least, only clauses that potentially lead to the conclusion are generated.

The proof proceeds by using the goal (the denial of the real goal) to locate another goal (called the sub-goal) and continuing this process until a contradiction is reached.

While Resolution is a completely general rule of inference, there is no guarantee that a statement which is a resolvant of two other statements is simpler than the given statements.

Most logic programming languages control this situation by limiting the clauses of a problem to those that contain at most one conclusion (non-negated predicate or one predicate on the left of the left arrow). Such a clause is called a Horn clause. Given this restriction and a goal that is the denial of what you want to prove, you can locate a sub-goal by doing a resolution between the given goal (which is negated) and any clause with a predicate (non-negated) that will unify with that goal. Since there is at most one non-negated predicate in a Horn clause, it gets deleted in the process of resolution giving as a result another clause containing only negated predicates -- i.e., another goal.

Sometimes the sub-goal will have more than one predicate. This is called a conjunctive goal (since both must be true to imply the contradiction). Since both must be true, you can try to prove them one at a time making sure that any substitutions made in proving one are made in the other predicates as well. PROLOG always attempts to prove the first of a conjunction first.

Here is an example problem solved in this manner.

The Problem:

```
ST1:  by table window ←
ST2:     on box table ←
ST3:      by Z Y ← (by X Y) and (on Z X)
```

The question (denial of initial goal)

```
G0:    ← by box window
```

1. Goal G0 unifies with positive predicate in ST3
   with the substitutions:
   Z←box
   Y←window

2. Resolve ST3 and G0 on the "by" predicate giving
   the sub-goal

```
G1:  ← (by X window) and (on  box X)
```

3. G1 is a conjunctive goal. PROLOG attempts to
   prove the first of the two statements first.
   The first predicate in G1 unifies with ST1
   with the substitution:
   X←table

4. Resolve ST1 and G1 giving:

```
G2:  ← on box table
```

5. G2 unifies with ST2 with no substitutions.

6. Resolve ST2 and G2 giving an empty clause
   which is a contradiction.

Thus proving the desired goal.

Given the desire to search for goals which are known to have no non-negated predicates, a more efficient Resolution program can be written. If there are no non-negated predicates in the goal there is no point in trying to unify them with negated predicates of a statement.

Here is a resolution program that assumes that the right argument is a goal:

∇Z← *A RESOLVEGOAL B*

[1] apply *RESOLVANT* between positive predicates from *A* and negative predicates from *B* in all combinations (∘.*RESOLVANT*)

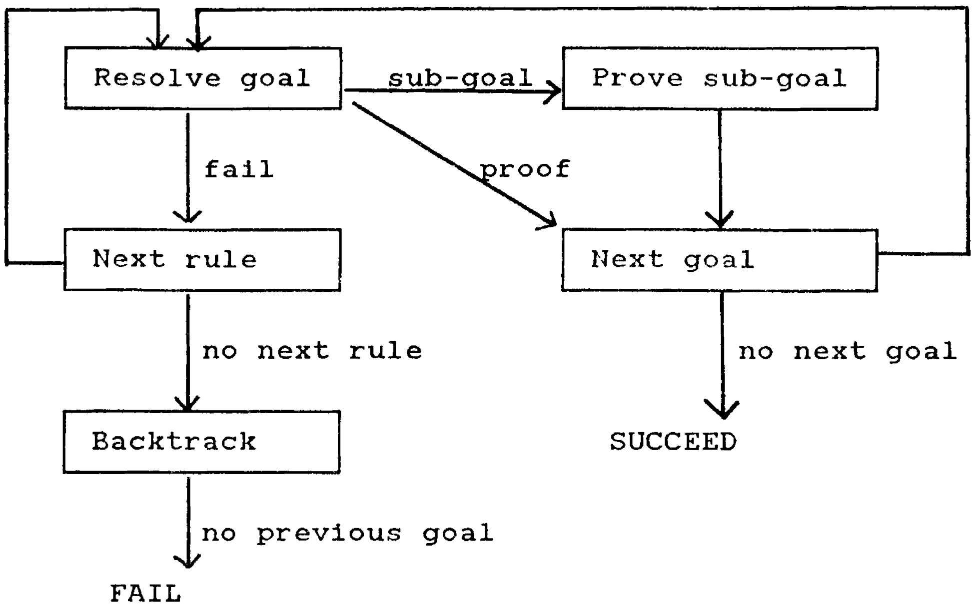
[2] delete non-resolutions

The actual program is in Appendix 1 and is the same as *RESOLVE* except it only makes one call of *RESOLVANT*.

Now all the tools are available to write a program that essentially implements the logic of PROLOG (minus a user friendly front end). A complete description of the PROLOG algorithm can be found in (Cl1). It is not practical to repeat it but here is a brief description which explains the previous example again and points out some other considerations.

Given a vector of clauses as a database and a possibly conjunctive goal (all of which are Horn clauses), attempt to derive a contradiction. As stated before, PROLOG attempts to prove the leftmost goal of the conjunction in a depth first fashion before looking at the next goal. The program is not straightforward because in satisfying one goal other possibly conjunctive sub-goals may be generated requiring a recursive call. Further, when a goal cannot be satisfied it may be because an earlier goal has more than one solution and the wrong one was found. In this case, the algorithm must back up and look for another solution of the earlier goal. This is called backtracking and involves forgetting values discovered for variables in the earlier clause. Finally, there may be more than one way to satisfy all the goals and you may want to know them all -- not just the first one discovered.

Here is a high level flow chart of PROLOG:



Here's an example problem whose solution requires backtracking:

ST1: john loves food
ST2: jane is female
ST3: john loves jane

The question to be answered is "Is there something that John loves which is female?" This is a conjunctive goal more formally stated:

G0 and G1: (john loves X) and (X is female)

PROLOG immediately satisfies G0 using ST1 giving the value "food" to variable X. Now it tries to satisfy G1 using the given value of X. This goal is "Food is female" which is not in the database. Now PROLOG must backtrack and attempt to find another way to satisfy G0 and this implies forgetting the value of X.

Now using ST3, G0 is satisfied and X gets the value "Jane". G1 becomes "Jane is female" which is trivially satisfied.

The actual program that implements this logic (*DFS* in Appendix 1) is more complicated that the others presented and does not illustrate any important new concept and so is not discussed in detail here. It basically uses *RESOLVEGOAL* to attempt to satisfy each goal in turn keeping track of substitutions in case backtracking is required. Even though it has less of a functional style than the other programs, it is, nonetheless, interesting that the logic of PROLOG can be captured in an *APL*2 program of a few dozen lines.

Part 5: Going Beyond the Fundamentals

This section briefly discusses some other areas in AI where *APL*2 can be applied with ease. The first section presents an alternate representation of knowledge that tends to be compact because it keeps together the information about a given subject. Other sections discuss how the ordinary computational ability of *APL*2 can be used for reasoning in exact and inexact environments.

5.1: Frames

You've seen one traditional way to represent knowledge. Frames (Mi1) are an attempt to represent knowledge that may be closer to the way people store knowledge. The basic idea is that there is a data structure that represents a generalization of some concept, describes the common case, gives initial values or assumptions, etc. Actual instances are represented as exceptions or refinements of the general case. People learn by induction -- extracting a general case from a set of particular instances. A frame stores this induced knowledge while still allowing differences in detail from the general case to be stored in sub-frames.

No attempt is made in this brief section to introduce all the terminology or details about frames. For that, refer to the literature (Ke1) (Mi1). Rather, only the representations of frame data with nested arrays is discussed.

Basically a frame is a set of pairs called slots. Each slot is a unique name and indicates a value, a set of values, or a procedure to invoke (called a demon). A set of pairs is easy to represent in an *APL*2 data structure.

Here is an example of a frame for Chablis wine borrowed from Keppel (Ke1). Here is the knowledge about Chablis to be stored in the frame array:

> The color is white
> There are 215 bottles
> The vintage is 1981 and 1982
> The price is computed by the procedure *GETPRICE*
> Chablis is a kind of wine and it is also a village.

These statements can be represented in *APL*2 vectors by the following pairs of values:

```
P1←'COLOR'    ('VALUE' 'WHITE')
P2←'QUANTITY' ('VALUE' 215)
P3←'YEAR'     ('VALUE' (1981 1982))
P4←'PRICE'    ('PROCEDURE' 'GETPRICE')
P5←'AKO'      ('FRAME' ('WINE' 'VILLAGE'))
```

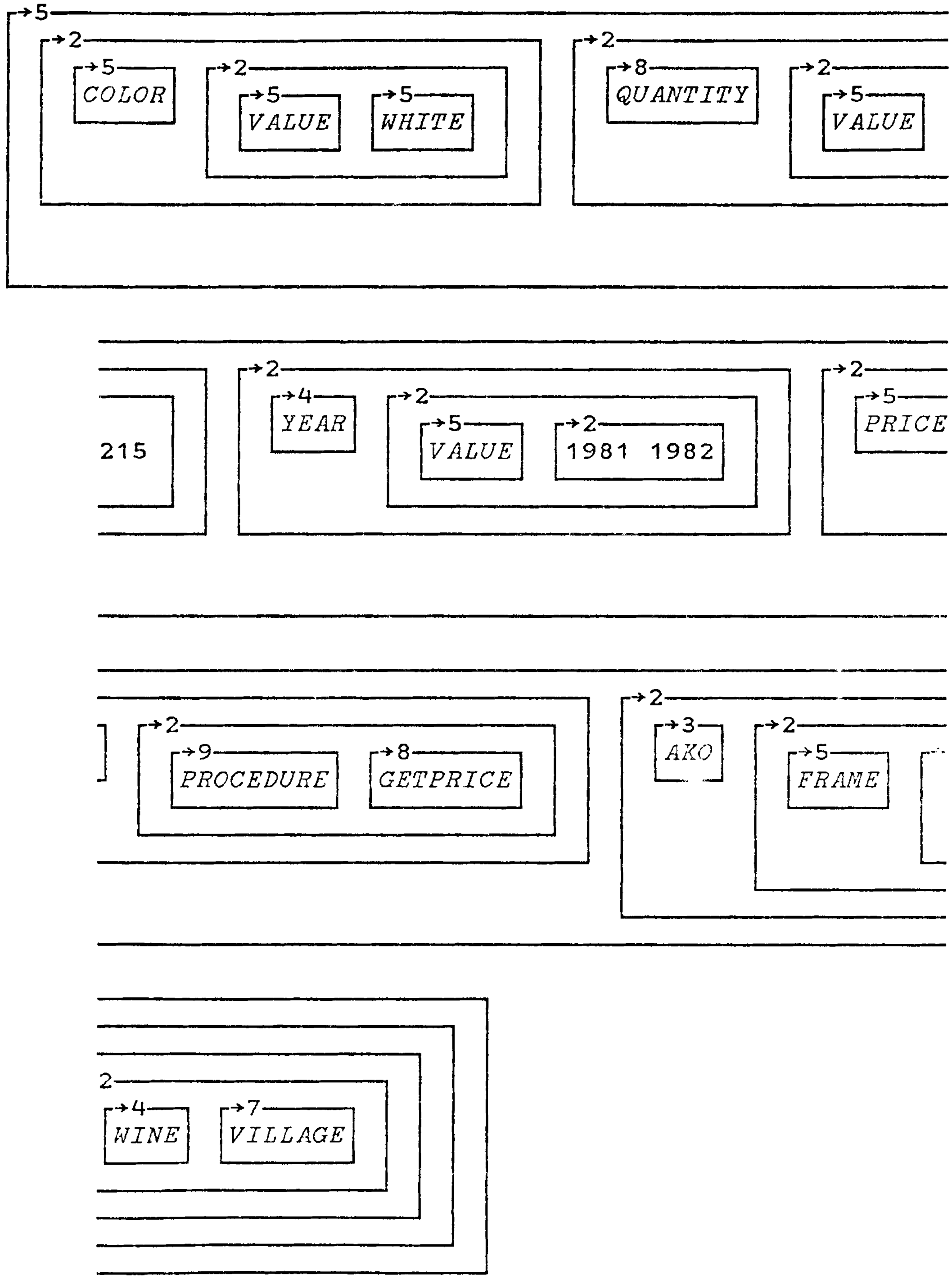
Each of these variables is a pair representing a frame slot. The first item is the name of the slot. *AKO* stands for "A Kind Of". In this case it means that Chablis is a kind of wine and it is also the name of a village. The second item of each variable is also a pair (although it doesn't have to be). The first item of the pair says what kind of information is stored in the second item. The first three variables contain values related to the slot name. The fourth one contains the name of a procedure to call should the price of the wine ever be needed. The last variable is a reference to two more general frames. The *AKO* slots tie the frames together into a network.

These five pairs can be represented in a single *APL*2 array either of two obvious ways. Keppel stored them as a vector of pairs:

```
CHABLIS1← P1 P2 P3 P4 P5
```

Here's a picture of this array:

```
DPY CHABLIS1
```



Since everything is a pair, this same data can also be stored in a two column matrix:

*CHABLIS*2←⊃*CHABLIS*1

Here's a picture of this array:

```
      DPY CHABLIS2
┌→5 2────────────────────────────────────────────────┐
↓ ┌→5────┐      ┌→2────────────────────┐             │
│ │COLOR │      │ ┌→5────┐  ┌→5────┐   │             │
│ └──────┘      │ │VALUE │  │WHITE │   │             │
│               │ └──────┘  └──────┘   │             │
│               └──────────────────────┘             │
│ ┌→8───────┐   ┌→2────────────────┐                 │
│ │QUANTITY │   │ ┌→5────┐         │                 │
│ └─────────┘   │ │VALUE │  215    │                 │
│               │ └──────┘         │                 │
│               └──────────────────┘                 │
│ ┌→4───┐       ┌→2──────────────────────────┐       │
│ │YEAR │       │ ┌→5────┐  ┌→2────────┐     │       │
│ └─────┘       │ │VALUE │  │1981 1982 │     │       │
│               │ └──────┘  └──────────┘     │       │
│               └────────────────────────────┘       │
│ ┌→5────┐      ┌→2──────────────────────────────┐   │
│ │PRICE │      │ ┌→9─────────┐  ┌→8────────┐    │   │
│ └──────┘      │ │PROCEDURE  │  │GETPRICE  │    │   │
│               │ └───────────┘  └──────────┘    │   │
│               └────────────────────────────────┘   │
│ ┌→3──┐        ┌→2────────────────────────────────┐ │
│ │AKO │        │ ┌→5────┐  ┌→2──────────────────┐ │ │
│ └────┘        │ │FRAME │  │ ┌→4───┐ ┌→7──────┐ │ │ │
│               │ └──────┘  │ │WINE │ │VILLAGE │ │ │ │
│               │           │ └─────┘ └────────┘ │ │ │
│               │           └────────────────────┘ │ │
│               └──────────────────────────────────┘ │
└────────────────────────────────────────────────────┘
```

Here is how you might use this matrix to find out information about a wine. Selecting column 1 of the matrix gives you all the slot names:

```
      CHABLIS2[;1]
 COLOR QUANTITY YEAR PRICE AKO
```

If you want to know what years are available, you can search to find out what row has that slot:

```
      CHABLIS2[;1]⍳⊂'YEAR'
3
```

Knowing that row three has the information about *YEAR*, you can select the corresponding value:

```
      CHABLIS2[3;2]
  VALUE  1981 1982
```

If the slot name does not exist, you could then search for the *AKO* slot and go search more general frames to get that information.

This is a simple example and leaves out many important ideas about frames. Nonetheless, you can see that the frame for Chablis contains information specific to that wine while information about wines in general is kept in the frame named *WINE* referenced by the *AKO* slot.

5.2: Boolean Logic

This section explores the computational abilities of *APL2* and how they might be used to represent and operate on logical expressions.

A given proposition *P* is either false or true. The values false and true are represented in *APL2* as the numbers 0 and 1 respectively. Therefore, the possible set of truth values for *P* can be represented as a two item vector:

```
      P1←0 1
```

The possible values for the negation of *P* are 1 when *P* is false and 0 when *P* is true.

```
      ~P1
1 0
```

Given this representation, trivial expressions about *P* can be computed. For example a tautology is always true:

```
      P1 ∨ (~P1)
1 1
```

A contradiction is never true:

```
      P1 ∧ (~P1)
0 0
```

If you want to write expressions with two variables, there are four possible combinations of true and false. If *Q*2 is false, then *P*2 may be false or true. If *Q*2 is true, then *P*2 may be false or true. Thus for two variables, complete sets of values can be represented as four item vectors:

```
      P2← 0 1 0 1
      Q2← 0 0 1 1
```

Now non-trivial expressions can be written. The expression $P2 \wedge Q2$ is true only when both $P2$ and $Q2$ are true:

```
      P2 ∧ Q2
0 0 0 1
```

The expression $P2 \vee Q2$ only fails to be true when both $P2$ and $Q2$ are false:

```
      P2 ∨ Q2
0 1 1 1
```

De Morgan's law shows that the negation of a conjunction is a disjunction and vice versa. One formulation of this rule is:

$$P2 \vee Q2 \leftrightarrow \sim (\sim P2) \wedge (\sim Q2)$$

Computationally this is

```
      (~P2) ∧ (~Q2)
1 0 0 0
      ~ (~P2) ∧ (~Q2)
0 1 1 1
```

which is the "or" function.

Implication is merely an application of the formula for implication. $P2$ implies $Q2$ is written:

```
      Q2 ∨ (~P2)
1 0 1 1
```

This result has a 1 wherever $Q2$ is either greater or equal to $P2$ and so implication could also be written with a single *APL*2 primitive:

```
      Q2 ≥ P2
1 0 1 1
```

Expressions containing three variables have eight possible combinations of values:

$P3 \leftarrow 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1$
$Q3 \leftarrow 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1$
$R3 \leftarrow 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1$

Here is the computation of three different implications:

1. $P3$ implies $Q3$

$Q3 \vee (\sim P3)$
1 0 1 1 1 0 1 1

2. $Q3$ implies $R3$

$R3 \vee (\sim Q3)$
1 1 0 0 1 1 1 1

3. $P3$ implies $R3$

$R3 \vee (\sim P3)$
1 0 1 0 1 1 1 1

Suppose that you claim that "$P3$ implies $Q3$" and "$P3$" are simultaneously true (Modus Ponens):

$(Q3 \vee (\sim P3)) \wedge P3$
0 0 0 1 0 0 0 1

You might expect to see the representation of $Q3$ from this computation (0 0 1 1 0 0 1 1). The answer differs from $Q3$ where $P3$ is false but $Q3$ is true. Since it is claimed that $P3$ is true, the boolean result is stronger than just $Q3$. It expresses the fact that both $P3$ and $Q3$ are true simultaneously.

Next, look at the chaining rule: If "$P3$ implies $Q3$" and "$Q3$ implies $R3$" then "$P3$ implies $R3$". The results of the individual implications are already listed above. The computation of the chaining rule is:

$(Q3 \vee (\sim P3)) \wedge (R3 \vee (\sim Q3))$
1 0 0 0 1 0 1 1

Again, you might expect the representation of "$P3$ implies $R3$" (1 0 1 0 1 1 1 1) but again the result produced is stronger. Suppose in addition to the chaining rule you assert that $P3$ is actually true:

```
      (Q3 ∨ (~P3)) ∧ (R3 ∨ (~Q3)) ∧ P3
0 0 0 0 0 0 0 1
```

This shows that *P*3, *Q*3, and *R*3 are all simultaneously true. This is stronger than the result of "P" and "*P*3 implies *R*3":

```
      (R3 ∨ (~P3)) ∧ P3
0 0 0 0 0 1 0 1
```

which makes no claim about the truth of *Q*3.

5.3: Parallel Boolean Logic

This section shows how you might go about using the application of the *APL*2 logical functions to solve a logic problem for all solutions in parallel.

The following problem is taken from (Sm1):

"When Alice entered the forest of forgetfulness,
she did not forget everything, only certain things.
She often forgot her name, and the most likely
thing for her to forget was the day of the week.
Now, the lion and the unicorn were frequent visitors
to this forest. These two are strange creatures.
The lion lies on Mondays, Tuesdays, and Wednesdays,
and tells the truth on the other days of the week.
The unicorn, on the other hand, lies on Thursdays,
Fridays, and Saturdays, but tells the truth on the
other days of the week.

One day Alice met the lion and the unicorn resting
under a tree. They made the following statements:

LION: Yesterday was one of my lying days
UNICORN: Yesterday was one of my lying days

From these statements, Alice, who was a bright girl,
was able to deduce the day of the week. What was it?"

The following *APL*2 solution is based on one produced by Manuel Alfonseca.

First the data must be defined. Here the variable *DAYS* is defined as the seven days of the week and *YEST* is defined as the day before each day of the week:

```
DAYS←'Sun' 'Mon' 'Tue' 'Wed' 'Thu' 'Fri' 'Sat'
YEST←'Sat' 'Sun' 'Mon' 'Tue' 'Wed' 'Thu' 'Fri'
```

Next, two variables are set up that describe the days when the lion lies (*LL*) and the days when the unicorn lies (*UL*):

```
LL ← 'Mon' 'Tue' 'Wed'
UL ← 'Thu' 'Fri' 'Sat'
```

Now you must write expressions that are true. There are two conditions under which the lion is telling the truth. This is one of his truth telling days and yesterday was a lying day or this is one of his lying days and yesterday was a truth telling days. Here are the boolean expressions that compute both of these:

```
      (~DAYS∊LL)    ∧    (YEST∊LL)
      1 0 0 0 1 1 1 ∧ 0 0 1 1 1 0 0
0 0 0 0 1 0 0
      (1 0 0 0 1 1 1 ∧ 0 0 1 1 1 0 0)/DAYS
Thu
      (DAYS∊LL)     ∧    (~YEST∊LL)
      0 1 1 1 0 0 0 ∧ 1 1 0 0 0 1 1
0 1 0 0 0 0 0
      (0 1 1 1 0 0 0 ∧ 1 1 0 0 0 1 1)/DAY
Mon
```

This says that if the lion is telling the truth it could only be Thursday and if the Lion is lying then this could only be Monday. Thus, we may define a variable representing when the lion tells the truth:

```
LT ← ((~DAYS∊LL)∧(YEST∊LL)) ∨ ((DAYS∊LL)∧(~YEST∊LL))
```

The same logic is true for the unicorn:

```
      (~DAYS∊UL)    ∧    (YEST∊UL)
      1 1 1 1 0 0 0 ∧ 1 0 0 0 0 1 1
1 0 0 0 0 0 0
      (1 1 1 1 0 0 0 ∧ 1 0 0 0 0 1 1)/DAYS
Sun
      (DAYS∊UL)     ∧    (~YEST∊UL)
      0 0 0 0 1 1 1 ∧ 0 1 1 1 1 0 0
0 0 0 0 1 0 0
      (0 0 0 0 1 1 1 ∧ 0 1 1 1 1 0 0)/DAYS
Thu
```

Here's the expression for when the unicorn tells the truth:

```
      UT ← ((~DAYS∊UL)∧(YEST∊UL)) ∨ ((DAYS∊UL)∧(~YEST∊UL))
```

By inspection you can see that only Thursday is true in both cases. Here, then is a summary of the solution in a more compact form:

```
  YEST ← ¯1⌽DAYS←'Sun' 'Mon' 'Tue' 'Wed' 'Thu' 'Fri' 'Sat'
 (LL UL) ← ('Mon' 'Tue' 'Wed')('Thu' 'Fri' 'Sat')
      LT ← ((~DAYS∊LL)∧(YEST∊LL)) ∨ ((DAYS∊LL)∧(~YEST∊LL))
      UT ← ((~DAYS∊UL)∧(YEST∊UL)) ∨ ((DAYS∊UL)∧(~YEST∊UL))
      (LT∧UT)/DAYS
Thu
```

This problem can therefore be solved using entirely boolean expressions in parallel written to describe precisely the problem as stated.

Sullivan and Fordyce (Fo1) describe a clever scheme for implementing a production expert system in *APL* using Boolean logic.

5.4: Fuzzy Logic

In discussing the truth of statements in *APL2* notation, the number 1 is used to mean "certainly true" and the number 0 is used to mean "certainly false". Given such input, one can produce results about which there is no doubt. Sometimes, however, statements and rules cannot be stated with certainty. Statements may be strongly believed. An inference can be made with a reasonable degree of confidence. Statements that are not known exactly are called fuzzy statements and the logic to combine them is called fuzzy logic. It is based on fuzzy sets which are sets where membership is not certain.

This section explores how the computational ability of *APL2* might be used to deal with uncertainty. It is, at best, an introduction to the concepts and the literature should be studied for more information (Be1).

If 1 means true and 0 means false, it makes sense to use numbers between 0 and 1 to express various levels of certainty -- a number near zero to mean very likely false and a number near one to mean very likely true. The intention is not necessarily to treat these fractions as probabilities (although that's one possibility) but rather just uncertainties. Use of the word "probability" is therefore avoided even though it would be convenient.

The desire is to use computational analogs to "negation", "and", and "or" which like '~', '∧', and, '∨' work on 0 and 1 without change and do something useful on numbers in between.

As a start, consider negation. If *P* is very likely true then you might assign it a value .9. The negation of something very likely true is something very likely false -- perhaps .1. Therefore, a good choice for the computational analog of negation (~*P*) is 1-*P*. It works correctly for certainty:

```
      P1 ← 0 1
      1-P1
 1 0
```

and it gives the expected answer on uncertainty:

```
      1-.9
 .1
```

(Remember that any function you chose is OK so long as it returns 0 when applied to 1 and 1 when applied to 0.)

In choosing the computational analogs of "and" and "or", it is reasonable to require that they obey De Morgan's law with respect to the negation function. Therefore the computational "and" and "or" functions (call them *ANDF* and *ORF*) should obey the identity:

```
      (P ORF Q) ↔ 1-(1-P) ANDF (1-Q)
```

Again the only other requirement is that the functions work unchanged on 0 and 1. *APL*2 has the functions "maximum" (⌈) and "minimum" (⌊) defined in the obvious way:

```
      10⌈13
13
      10⌊13
10
```

Applied to zero and 1 they work just like "and" and "or":

```
      P2 ← 0 1 0 1
      Q2 ← 0 0 1 1
```

"Maximum" is the same as "or":

```
      P2 ∨ Q2
0 1 1 1
      P2 ⌈ Q2
0 1 1 1
```

"Minimum" is the same as "and":

```
      P2 ∧ Q2
0 0 0 1
      P2 ⌊ Q2
0 0 0 1
```

Thus, you may replace "and" by "minimum" and "or" by "maximum". When you "and" two uncertain values, you get the least likely:

```
      .1 ⌊ .9
 .1
```

When you "or" two uncertain values you get the most likely

```
      .1 ⌈ .9
 .9
```

These functions do follow De Morgan's rule:

```
      A⌈B ←→ 1-(1-A)⌊(1-B)
```

This is just a modification of the well known *APL2* identity on "maximum" and "minimum":

```
      A⌈B ←→ -(-A)⌊(-B)
```

Using "maximum" and "minimum" may not match your intuition about how uncertain values should work. You may feel that when you "and" two uncertain values, you should get a value less than either given values. In that case, you could use "multiply" (×) for "and". Again, it works like "and" on 0 and 1:

```
      P2 ∧ Q2
0 0 0 1
      P2 × Q2
0 0 0 1
```

When applied between inexact values, it produces numbers less or equal to the given values:

```
      .1 × .9
 .09
      .2 × .9
 .18
```

It is less obvious what the corresponding "or" function should be. You might at first try "addition" (+) but that fails on zero and 1 (since 1+1 is 2 not 1). Since De Morgan's law is to hold, you can just use that to solve for the "or" function.

```
P ORF Q ↔ 1 - (1-P) × (1-Q)
        ↔ 1 - (1-P-Q+(P×Q))
        ↔ P + Q - (P×Q)
```

Therefore, you can define *ORF* to be this function:

```
      ⎕FX 'Z←A ORF B' 'Z←A+B-(A×B)'
ORF
```

This function works correctly on 0 and 1 and gives answers that match the second intuition on numbers in between 0 and 1:

```
      .1 ORF .9
 .91
      .2 ORF .9
 .92
```

Again, any function that does the right thing on 0 and 1 is a candidate for "and" and "or" when applied to inexact statements.

Given these formulae for fuzzy logic, you may now apply them to the other formulae of logic. For example, precise implication applied to imprecise statements is achieved by using these functions in *Q* ∨ (~*P*). For "maximum" and "minimum", implication is written:

```
      Q ⌈ (1-P)
```

For "times" and "*ORF*", implication is written:

```
      Q ORF (1-P)
```

This section has introduced the concepts of fuzzy logic. The situation can be more complicated when uncertainty is described with distribution functions or worse when the rules (such as implication) are also imprecise. These topics are not discussed in this paper. Writing systems that implement these more difficult areas are likely to exploit even more the computational abilities of *APL2*.

Summary

This paper attempted to cover a wide variety of topics related to *APL*2 and Artificial Intelligence.

Part 1 introduced the concept of Artificial Intelligence and discussed in general terms how *APL*2 is a useful implementation language for solutions.

Part 2 discussed the main ideas of logic as background to the implementation.

Part 3 introduced a subset of the *APL*2 notation concentrating on the data structures and the operators.

Part 4 showed one way to represent logic and showed a way to implement the algorithms using that structure. Because the *APL*2 operators apply functions to items, the main data structure (depth 5 or more) is never explicitly taken apart. Given a database (depth 5 or more), the search functions apply Resolution with an operator. Since the items of a database are clauses, *RESOLVE* sees clauses (depth 4). *RESOLVE* selects the positive and negative clause lists (depth 3) and uses outer product to apply *RESOLVANT* (and therefore *UNIFY*) between all combinations of predicates (depth 2).

Part 5 showed another representation of data and investigated boolean logic and fuzzy logic.

Conclusions

Algorithms for Artificial Intelligence have traditionally been expressed using LISP-like languages. *APL*2 provides an opportunity to express them in a different style. Parallel constructions give an alternative to recursive ones.

The data structures and algorithms presented here are examples of how *APL*2 can be used to solve logic problems. They are not recommended as the only or best implementations. The purpose is, rather, to elicit an understanding of the issues and approaches to solving them. *APL*2 provides a different way to explore solutions to AI problems. In the hands of a creative person, it may be a tool which can be used to further the study and practice of logic programming.

Acknowledgements

The following people contributed to the production of this paper with suggestions, reviews, or significant publications or presentations: Manuel Alfonseca, Everett Allen, Claudia Baker, Langdon Beeck, Bill Burr, Ken Fordyce, Garth Foster, Chuck Haspel, Denny Jizba, Dieter Lattermann, Dave Macklin, John McInturff, Lenore Mullin, Dave Selby, Darryl Smith, John Sowa, Gary Sullivan, Trung Tran, Sheryl Winton and the APL Development group at the Santa Teresa Laboratory in their "Lunch and Learn" sessions.

In particular, referenced book (Ch2) by Charniak and McDermott was key in that it attempts to present many of the underlying ideas of AI in clear English descriptions rather that LISP implementations.

References

(Ab1) Abelson, Harold, Sussman, Gerald, and Sussman, Julie, "Structure and Interpretation of Computer Programs", MIT Press, Cambridge, Mass., 1985.

(Ba1) Barr, A., and Feigenbaum, E. "Handbook of AI", Vol. 2, William Kaufman, 1982.

(Be1) Bellman, R.E. and Zadeh, L.A., "Decision-making in a Fuzzy Environment", National Aeronautics and Space Administration, contractor report # NASA CR-1594, May 1970.

(Ch1) Charniak, Eugene, Riesbeck, Christopher K., and McDermott, Drew, "Artificial Intelligence Programming", Lawrence Erlbaum Assoc. Publishers, Hillsdale, New Jersey, 1980.

(Ch2) Charniak, Eugene and McDermott, Drew, "Introduction to Artificial Intelligence", Addison-Wesley Publishing Co., 1985.

(Cl1) Clocksin, W.F., Mellish, C.S., "Programming in Prolog", Springer-Verlag, New York, 1981.

(Da1) Davis, Ruth E., "Logic Programming and Prolog: A Tutorial", IEEE Software, Sept. 1985, pp. 53-62.

(Eu1) Eusebi, E.V., "Operators for Program Control", APL'85 Conference proceedings, APL Quote Quad Vol. 15 #4, p.181 ff

(Eu2) Eusebi, E.V., "Operators for Recursion", APL'85 Conference proceedings, APL Quote Quad Vol. 15 #4, p.190 ff

(Fo1) Fordyce, K., Sullivan, G. ,"Artificial Intelligence Development Aids (AIDA)", Proceedings of *APL*85, *APL* Quote Quad, Vol 15, No. 4, 1985, pp.106-113.

(Fr1) Frank, Werner L., "AI: What's different between old and new?", Software news, Sept. 1985, pp. 38-40.

(Gr1) Graham, Neill, "Artificial Intelligence", TAB books, Blue Ridge Summit, Pa., 1979.

(Hi1) Hirsch, P. et al, "Interfaces for Knowledge-base Builders' Control Knowledge and Application-specific Procedures", IBM Journal of Research and Development, Vol. 30, No. 1, Jan., 1986, pp 29-38.

(Ke1) Keppel, E., Kropp, D., "APL2 or LISP? Implementing Frames, a Knowledge representation scheme", Vector, the Journal of the British APL Association, Vol. 2, No. 2, Oct. 1985.

(Ll1) Lloyd, J.W., "Foundations of Logic Programming", Springer-Verlag, New York, 1984.

(Mi1) Minsky, M.,"A Framework for Representing Knowledge", The Psychology of Computer Vision, McGraw Hill, New York, 1975, pp.211-277.

(Ni1) Nilsson, Nils J., "Principles of Artificial Intelligence", Tioga Publishing Co., Palo Alto, Calif. 1980.

(Pa1) Paterson, M.S. and Wegman, M.N. "Linear Unification", Journal of Computer and Systems Science, No 16, 1978

(Ri1) Rich, Elaine, "Artificial Intelligence", McGraw Hill, New York, 1983.

(Ro1) Robinson, J. A., "A Machine-oriented logic based on the Resolution Principle", Journal of the ACM 12(1) pg 23, 1965.

(Sm1) Smullyan, "What is the Name of this Book?", Prentice-Hall, 1978.

(Wi1) Winston, Patrick Henry, "Artificial Intelligence", Addison-Wesley, Reding, Mass. July 1984.

Appendix 1: Implementations of the Algorithms

** The APL2 Unification Algorithm

The *APL2* algorithm is straightforward. If the arguments don't already match, then if one is a variable, the other is substituted for it as the value. (Substitution is discussed separately.) If neither formula is an atom, then Unification is recursively applied to each item.

```
      ∇Z←X UNIFYA Y
[1]   ⍝ unify X with Y
[2]    →(0=⍴∊X Y)/FAIL       ⍝ fail if both clauses empty
[3]    (X Y)←EVAL DEPTH1¨(X Y)      ⍝ do substitutions in X
[4]    →(X≡Y)/GOOD
[5]    (X Y)←(1=≡Y)⌽X Y  ⍝ put atom first if any
[6]    →(1≠≡X)/RECUR         ⍝ if not an atom, apply to each
[7]   ⍝ here  is an atom
[8]    (X Y)←('Δ'≡↑Y)⌽X Y  ⍝ put variable first if any
[9]    →(~'Δ'≡↑X)/FAIL       ⍝ if no variable, items are different
[10]   →(X⍷∊Y)/FAIL          ⍝ fail if var exists in substitute
[11]   ⍎X,'←Y'               ⍝ do substitution
[12]   →GOOD
[13]  RECUR:→(~(⍴X)≡⍴Y)/FAIL
[14]   →(1=∧/X UNIFYA¨Y)/GOOD
[15]  FAIL:→Z←0
[16]  GOOD:Z←1
```

[2] causes failure to unify if both arguments are empty formulas. Strictly speaking, two empty clauses do unify since they match. In practice, however, when two empties arise (as they do in resolution, saying that they unify leads to redundant implications.

[3] makes sure that any previously determined substitutions are made in the arguments.

[4] if the statements are the same, they unify.

[5] and [6] work together to determine if both arguments are non-trivial (i.e. more than one term). [5] puts an atom first if there is one. [6] branches to *RECUR* if there is no atom.

[8] and [9] work together to determine if there is a variable. [8] puts the variable (if any) into *X*. [9] fails if *X* is not a variable (since it is already known that *X* and *Y* are different.

[10] makes sure that the value substituted for a variable does not contain the same variable. That is not a legal substitution. This is sometimes called an "occurs check". It is often not done in logic programs for reasons of efficiency. It is required, however, to ensure correctness. The check used here works because of the convention used for names of logic variables (see the description of *ENCODE* in Appendix 1).

[11] records the substitution by setting the variable (which is a real variable in the *APL*2 workspace) to be the value. See the following section for a discussion of this substitution.

In the case that both arguments are non-trivial formulas (label *RECUR*), then if the formulas are the same length, *UNIFYA* is applied between corresponding items.

The *UNIFYA* program is a description of the unification algorithm. It is not the most efficient implementation. There are many things that could be done to improve computational efficiency but they would not add to the understanding of the algorithm. For example, instead of *UNIFYA*¨, you could use the defined operator *FALSE* or *UNTIL* to make the expression quit as soon as a failure case was discovered. This would avoid applying *UNIFYA* after failure is discovered and would avoid the 1=∧/ on the result. The algorithm as it stands, however, is descriptive of the process.

** Unification Cover Function

Each call of unification should be independent of the other calls. Furthermore, it is desirable to know not only that unification succeeded, but also the substitutions that made it work. Therefore, a second function is used to initiate and terminate *UNIFYA*

```
    ∇ Z←X UNIFY Y;T;USUBS
[1]   ⍝ Unification algorithm - main function
[2]   ⍝ Z is a two item vector
[3]   ⍝    0 or 1 for failure or success
[4]   ⍝    the substitutions
[5]    T←⎕EX 'Δ' ⎕NL 2
[6]    Z←X UNIFYA Y
[7]    USUBS←2 ⎕TF¨⊂[2]'Δ' ⎕NL 2
[8]    Z←Z USUBS
```

[5] makes sure that no logic variables (represented by names starting with '∆') have any values.

[6] calls the unification algorithm and produces the result true (1) or false (0),.

[7] records substitutions made for any variables.

[8] returns the two results of unification.

** The APL2 Resolution Algorithm

Here is the function that produces the resolvant from a single unification. The arguments are each a single predicate:

```
     ∇Z←AR RESOLVANT BR;T
[1]   ⍝ UNIFY AR with BR, Z is 0 or the implied resolution
[2]   ⍝ A and B are global
[3]    Z←0                        ⍝ assume failure
[4]    →(↑T←AR UNIFY BR)↓0        ⍝ return with 0 on fail're to UNIFY
[5]    Z←(((1⊃A)~⊂AR),1⊃B)((2⊃A),(2⊃B)~⊂BR)
[6]    Z←(EVAL DEPTH1 Z)(↑1↑T)
```

[3] sets result to zero in case unification fails.

[4] attempts to unify the given predicates and returns if unification fails.

[5] builds the inferred clause by constructing the positive and negative clause lists. The positive clause list is constructed by joining together all the positive predicates from the original two clauses, then using "without" (~) to delete the one canceled by resolution. The negative clause list is constructed the same way.

[6] applies the substitutions implied by the unification and returns a two item vector containing the resolvant and the substitutions that permitted resolution.

The function returns either 0 or the implied statements and the substitutions that permitted them.

The resolution program *RESOLVE* only needs to call *RESOLVANT* for all combinations of of predicates suitably chosen. Here is the program:

```
     ∇Z←A RESOLVE B
[1]   Z←,(1⊃A)∘. RESOLVANT(2⊃B)
[2]   (B A)←(A B)
[3]   Z←Z,,(1⊃A)∘. RESOLVANT(2⊃B)
[4]   Z←Z~0
```

[1] gets statements inferred by positive terms of *A* and negative terms of *B*.

[2] swaps *A* and *B*.

[3] gets statements inferred by positive terms of *B* and negative terms of *A*.

[4] deletes any non-resolutions

It is possible that a tautology may be implied. See Appendix 4 for a description of the test for a tautology.

Each outer product in *RESOLVE* might return several implied clauses. Thus, the result of *RESOLVE* is not a vector of the implies statements but rather a vector of vectors of them. For this reason, you will see that the callers of *RESOLVE* often do a ↑,/ which will turn the vector of vector of clauses into a vector of clauses.

** The APL2 Resolution Algorithm for Goals

This function is essentially the same as *RESOLVE* except the right argument is assumed to be a goal clause which therefore has an empty positive clause list. The function returns either 0 or the implied clauses.

```
     ∇Z←A RESOLVEGOAL B
[1]   Z←,(1⊃A)∘. RESOLVANT(2⊃B)
[2]   Z←Z~0
```

[1] gets statements inferred by positive terms of *A* and negative terms of *B*.

[2] deletes any non-resolutions

** The Forward Search Algorithms

The following two forward chaining search functions apply resolution to everything known in the database and check to see if the desired goal shows up. The first function adds anything implied to the database and loops until the goal is found or nothing new is implied. This is formally correct and descriptive but terrible in performance. The second function only does resolutions between what is in the database and the newly inferred statements adding new clauses to the database each iteration.

They will not be discussed in detail -- the comments on each line describe the purpose of the line.

```
      ∇PZ←GOAL FORWARD1 DB;NEW
[1]     PZ←1                             ⍝ assume goal will be found
[2]    L1:→(GOAL∊DB)/0                   ⍝ done if goal is found
[3]     NEW←DB∘. RESOLVE DB              ⍝ resolve everything
[4]     NEW←↑¨↑,/,NEW                    ⍝ select new inferences
[5]     NEW←NEW~DB                       ⍝ it's not new if already in DB
[6]     NEW←((NEW⍳NEW)=⍳⍴NEW)/NEW        ⍝ delete duplicate inferences
[7]     →(0=⍴NEW)/FAIL                   ⍝ fail if no new inferences
[8]     DB←DB,VRENAME¨ NEW               ⍝ rename variables & add to DB
[9]    →L1                               ⍝ go do resolutions again
[10]   FAIL:PZ←0                         ⍝ goal not found
```

```
      ∇PZ←GOAL FORWARD DB;NEW;NEW2
[1]     PZ←1                             ⍝ assume goal will be found
[2]     NEW←DB                           ⍝ DB against itself first time
[3]    L1:→(GOAL∊NEW)/0                  ⍝ done if goal is found
[4]     NEW2←NEW∘. RESOLVE DB            ⍝ resolve everything
[5]     DB←DB,NEW                        ⍝ add last ones to DB
[6]     DB←((DB⍳DB)=⍳⍴DB)/DB             ⍝ discard duplicates
[7]     NEW←↑¨↑,/,NEW2                   ⍝ select new inferences
[8]     NEW←((NEW⍳NEW)=⍳⍴NEW)/NEW        ⍝ discard duplicates
[9]     NEW←VRENAME¨ NEW                 ⍝ rename variables
[10]    →(0=⍴NEW)/FAIL                   ⍝ fail if nothing new
[11]    →L1                              ⍝ go do resolutions again
[12]   FAIL:PZ←0                         ⍝ goal not found
```

** PROLOG

The function that implements the logic of PROLOG uses *RESOLVEGOAL* and recursion to satisfy each part of a possibly conjunctive goal in order from left to right. It is possible that an infinite recursion may cause the program to loop. This can happen in real PROLOG also.

```
    ∇ Z←CGS DFS DB;DBI;GI;ASUBS;GSUBS;SG;T
[1]    (GOAL ASUBS)←2↑CGS          ⍝ divide argument
[2]    SG←SPLITGOAL↑GOAL           ⍝ get simple goals
[3]   START:GI←1                   ⍝ start with first goal
[4]    GSUBS←(⍴SG)⍴'0'             ⍝ initial substitutions
[5]    DBI←(⍴SG)⍴1                 ⍝ current DB item per goal
[6]   NEXT:T←⎕EX 'Δ' ⎕NL 2         ⍝ erase all variables
[7]    T←⍎DEPTH1 ASUBS GSUBS       ⍝ define known variables
[8]    T←EVAL DEPTH1(DBI[GI]⊃DB)   ⍝ select next rule
[9]    NEW←T RESOLVEGOAL EVAL DEPTH1 GI⊃SG  ⍝ resolve
[10]  →(0≠⍴NEW)/RES                ⍝ branch something found
[11]  NEXTDB:→((⍴DB)≥DBI[GI]←DBI[GI]+1)/NEXT  ⍝ try next rule
[12]   DBI[GI]←1                   ⍝ initial index again
[13]  BACK:→(0=GI←GI-1)/Z←0        ⍝ back up to previous goal
[14]   (GI⊃GSUBS)←'0'              ⍝ forget old substitutions
[15]   →NEXTDB                     ⍝ find another proof
[16]  RES:(GI⊃GSUBS)←2⊃↑NEW        ⍝ record substitutions
[17]   →(0=⍴1 1 2⊃NEW)/NEXTG       ⍝ branch if proved
[18]   T←((VRENAME↑NEW)[1])(ASUBS GSUBS)DFS DB ⍝ do sub-goal
[19]   →(↑T)/SGOK                  ⍝ branch sub-goal OK
[20]   GSUBS[GI]←'0'               ⍝ forget substitutions
[21]   →NEXTDB                     ⍝ to next clause in DB
[22]  SGOK:(GI⊃GSUBS)←(GI⊃GSUBS),2⊃T ⍝ record new subs
[23]  NEXTG:GI←GI+1                ⍝ on to next goal
[24]   →(GI≤⍴SG)/NEXT              ⍝ branch more goals
[25]  DONE:Z←1 GSUBS               ⍝ done
```

The left argument to *DFS* is a two item vector. The first item is the goal to be satisfied and the second item is the current list of substitutions. The initial substitution is '0' meaning there are no variables. The right argument is the vector of clauses representing the database.

[1] separates the left argument into two names for convenience.

[2] separates a conjunctive clause of *N* predicates into a vector of *N* separate goals which can be satisfied independently.

[3] sets the goal index to indicate that the first goal will be satisfied first.

[4] defines the variable that will hold the substitutions related to each goal. By keeping the substitutions for each goal separate, they can be independently forgotten in case of backtracking.

[5] defines an index to the database for each goal

[6-9] makes the currently defined substitutions in both the current goal and the current item from the database and calls *RESOLVEGOAL*.

[10] branches if any resolvant was produced.

[11] selects the next clause from the database to see if it leads to satisfaction of the goal.

[12] is reached if no clause from the database satisfied the current goal. The database index is set to 1 for the next time this goal is tried.

[13-15] backtrack to the previous goal forgetting the substitutions for that goal. If there is no previous goal then the attempt to satisfy the goals has failed.

[16] is reached if a resolvant is found. The substitutions that permitted the resolution are recorded.

[17] checks for a contradiction in which case this goal is satisfied and the program can proceed with the next goal.

[18] recursively calls this program to satisfy the generated sub-goal.

[19] determines if the sub-goal was satisfied.

[20-21] goes back to try another clause from the database in the case the sub-goal was not satisfied.

[22] remembers the substitutions that allowed the sub-goal to be satisfied.

[23-24] moves on to the next goal if any

[25] returns a 1 meaning success and the record of the substitutions that led to success

A slightly fancier program *DF* permits the user to call the function again and pick up the search from where it left off to find another proof of the same goal.

```
    ∇ Z←CGS DF DB;DBI;GI;ASUBS;GSUBS;SG;T
     (GOAL ASUBS)←2↑CGS ⍝ divide argument
     SG←SPLITGOAL↑GOAL    ⍝ get simple goals
     →(3>⍴CGS)/START      ⍝ branch first call
     (GI GSUBS DBI)←3⊃CGS⍝ redefine controls
     ((GI-1)↓DBI)←1       ⍝ reset db indexes
     →BACK                ⍝ backtrack
   START:GI←1             ⍝ start with first goal
     GSUBS←(⍴SG)⍴'0'      ⍝ initial substitutions
     DBI←(⍴SG)⍴1          ⍝ current DB item per goal
   NEXT:T←⎕EX '∆' ⎕NL 2         ⍝ erase all variable
     T←⍎DEPTH1 ASUBS GSUBS      ⍝ define known variables
     T←EVAL DEPTH1(DBI[GI]⊃DB)  ⍝ select next rule
     NEW←T RESOLVEGOAL EVAL DEPTH1 GI⊃SG ⍝ resolve
     →(0≠⍴NEW)/RES              ⍝ branch something found
   NEXTDB:→((⍴DB)≥DBI[GI]←DBI[GI]+1)/NEXT  ⍝ try next rule
     DBI[GI]←1                  ⍝ initial index again
   BACK:→(0=GI←GI-1)/Z←0        ⍝ back up to previous goal
     (GI⊃GSUBS)←'0'             ⍝ forget old substitutions
     →NEXTDB                    ⍝ find another proof
   RES:(GI⊃GSUBS)←2⊃↑NEW        ⍝ record substitutions
     →(0=⍴1 1 2⊃NEW)/NEXTG      ⍝ branch if proved
     T←((VRENAME↑NEW)[1])(ASUBS GSUBS)DF DB ⍝ do sub-goal
     →(↑T)/SGOK                 ⍝ branch sub-goal OK
     GSUBS[GI]←'0'              ⍝ forget substitutions
     →NEXTDB                    ⍝ to next clause in DB
   SGOK:(GI⊃GSUBS)←(GI⊃GSUBS),2⊃T ⍝ record new subs
   NEXTG:GI←GI+1                ⍝ on to next goal
     →(GI≤⍴SG)/NEXT             ⍝ branch more goals
   DONE:Z←1 GSUBS(GI GSUBS DBI) ⍝ done
```

DF is identical to *DFS* except for two things. On exit, *DF* returns a three item vector instead of a two item vector with the third item being the information needed to restart the search from where it left off. On entry, if the left argument is a three item vector, the control variables are reset to the saved values and the program entered as though the goal just failed. The backtracking mechanism will then cause a search for another solution.

The function *PROLOG* automatically does the re-call of *DF* if the user responds with a semicolon after an answer is reported. Many real PROLOG systems use this convention.

```
    ∇ Z←L PROLOG R;T
     Z←1                      ⍝ assume success
     →(↑T←(L '0')DF R)/GOOD   ⍝ branch success
     →Z←0                     ⍝ return with failure
   GOOD:'VALUES '(2⊃T)        ⍝ report answer
     →(';'≠↑⍞)/0              ⍝ end unless ;
     →(↑T←(L '0'(3⊃T))DF R)/GOOD ⍝ re-call
```

** Support Functions

The *ENCODE* function is used to simplify the handling of variables in logic statements. Different versions of PROLOG use different conventions to identify variables in clauses. Some PROLOGs use a leading * or _ to indicate a logic variable. In this paper, a leading upper case character indicates a logic variable. The *ENCODE* function locates words with a leading uppercase letter (using the global variable *UC*) and appends a 'Δ' on the front. Any word which is not a variable is prefixed with a lowercase 'a'. Changing *UC* to the value '*' would implement another PROLOG convention.

Doing this append has two less obvious benefits. First, it means that all character strings are vectors. If any were one character and therefore possibly represented as a scalar, it would become a vector. This means the algorithms may assume vectors of vectors at all stages. Secondly if 'Δ' is not permitted anywhere but as the first letter of a variable, then the occurs check needed in *UNIFYA* is trivial.

```
    UC←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

    ∇ Z←ENCODE R
[1]  ⍝ put R in internal form
[2]  ⍝ constants prefixed with ATOM
[3]  ⍝ variables prefixed with ΔVAR
[4]   →(0=⍴Z←R)/0          ⍝ empties stay empty
[5]   Z←((1+(↑R)∊UC)⊃'a' 'Δ'),R
[6]   R←⎕EX Z              ⍝ ensure name has no value
```

The functions *VRENAME* and *VRENAME*1 take a clause and give the variables in the clause unique names. Doing this before each clause is added to the database means that none of the programs have to worry about two clauses having the same variable name. It is not wrong for two clauses to have the same variable name so long as it is understood that they are not the same variable. The functions presented here assume that each clause has unique variable names.

```
    ∇ Z←VRENAME R
[1]   VCOUNT←VCOUNT+1
[2]   Z←VRENAME1 DEPTH1 R


    ∇ Z←VRENAME1 R
[1]   Z←R
[2]   →('Δ'≠↑R)/0
[3]   Z←R,(⍕VCOUNT)
```

The *SPLITGOAL* function is given a single conjunctive goal clause and produces a vector of simple clauses with no conjunctions.

```
    ∇ Z←SPLITGOAL G
[1]  ⍝ G is a single possibly conjunctive goal
[2]  ⍝ Z is a vector of simple goals (one deeper)
[3]   Z←EMPTYCLAUSE,¨⊂¨⊂¨2⊃G
```

EMPTYCLAUSE returns a 1 if the clause *X* is empty.

```
    ∇ Z←EMPTYCLAUSE X
[1]   Z←(0∊⍴X)∨' '∧.=X←∊X
[2]  ⍝1 IF X CONTAINS ONLY EMPTY STUFF OR BLANKS
```

Appendix 2: Glossary

- Abduction - an illegal but useful rule of inference - If A implies B, and B is true, then A is true. This is the basis of medical diagnosis.

- AKO - means A Kind Of - a token relating a class of objects to a more general class of objects. The class of "private homes" is a kind of "building".

- Ambiguity - something that could have two (or more) conflicting meanings. ("Ambi" means "both")

- Antecedent - In the implication "If *P* then *Q*", *P* is called the antecedent and *Q* the consequent.

- Argument - a value to which some relation is applied. It has nothing to do with a dispute.

- Assertion - a formula believed to be true and therefore in some factbase and represented in some knowledge database.

- Atom - a number or symbol (like an *APL* constructed name) whose structure is not of interest. A proposition that cannot be broken down into other propositions.

- Atomic formula - a predicate and a proper number of arguments (terms).

- Axiom - initial facts - assumed to be true. Unlike mathematics, where axioms are usually given at the start, axioms will usually be added as time goes on (because of new information received).

- Backward chaining - making an inference at the time a query is made (i.e. wait until an answer is needed before trying to infer it). Thus given a desired conclusion, deny it and work backwards until a known fact is reached giving a contradiction.

- Breadth-first search - if two places are to be looked at in the order 'place 1' then 'place 2', then 'place 2' is looked at before anyplace reachable from 'place 1' is looked at. This is like looking at every node in a tree of path length N from the root before looking at any node of length N+1 from the root.

- Clause - A disjunction of predicates ($Q1 \vee Q2 \vee (\sim Q3) \ldots$). The statements of PROLOG are clauses with one conclusion (positive predicate) called the head of the clause (called a Horn clause or a program clause).

- Closed knowledge base - one that contains everything that is true (like an airline reservation system). Anything not in the database is not true.

- Closed world assumption - Logic programs cannot in general prove negative statements like $\sim P(a)$. If the knowledge base is closed, then if you can't prove $P(a)$ you may infer $\sim P(a)$.

- Conflict set - the set of rules which could be applied next

- Conjunction - "and" - the conjunction of two formulas is true if both formulas are true.

- Conjunctive normal form - A conjunction of disjunctions (i.e. and "and" of clauses) $(Q1 \vee Q2 \vee Q3 \ldots) \wedge (\ldots) \wedge (\ldots) \wedge \ldots$ Since a fact or a rule is represented as a disjunction, a conjunctive normal form is the formal representation of a knowledge base.

- Consequent - In the implication "If P then Q, P is called the antecedent and Q the consequent.

- Database - data structures that represent what is currently known (i.e., represents the factbase). In PROLOG, the database is the set of all clauses.

- Deduction - discovering new facts from existing facts.

- Default reasoning - an illegal but useful rule of inference - If there is no proof that A is not B, then A is B. (i.e., if you cannot infer not B then infer B.

- Demon - a procedure invoked automatically to compute values when values are needed.

- Depth first search - If two places are to be looked at in the order 'place 1' then 'place 2', then every place reachable from 'place 1' is looked at before 'place 2' is looked at. This is like searching a tree in left list order.

- Disjunction - "or" - the disjunction of two formulas is true if either formula is true.

- Existential quantifier - something is true for at least one value of a variable.

- Expert system - a program that gives expert assistance to a non-expert.

- Factbase - what is currently known (as opposed to the database used to represent it).

- Facts - Statements assumed to be true without conditions. Because anything infers something that is true, a fact is often represented as an implication with empty antecedent.

- False - nil or () in LISP, 0 in *APL*.

- Forward chaining - making an inference at the time an assertion is made. Given facts, make inferences until the desired conclusion is reached.

- Frame - a single data structure that include all of the information of interest for a particular concept. A frame usually holds information about a general case with a specific case represented as exceptions to the general case.

- Gatekeeper - a program which performs inferences and adds or deletes them from the set of statements believed to be true (also called an inference engine).

- Goal - A clause which is to be proven. A proof often proceeds by denying the result and proving a contradiction. The denial of a positive goal is a negative goal and is therefore a Horn clause with no positive term at all.

- Ground clause - a clause with no variables

- Herbrand base - all possible applications of predicates with terms from the Herbrand universe.

- Herbrand universe - set of all ground terms which can be constructed out of functions and a given set of constants. Given a set of constants and some functions, the Herbrand universe represents everything that can be talked about.

- Horn clause - A clause that contains at most one conclusion. A conclusion is often proved by postulating its negative and proving a contradiction. The modified statement is phrased as the "or" of the negations of the assumptions 'or'ed with the conclusion. Thus, a Horn clause has at most one non-negated term.

- Implication - If A then B. A is called the antecedent, and B is called the consequent. Equivalent to B or (not A). /

- Induction - an illegal but useful rule of inference. If A is true for every instance of A that we know about, then A is true for all instances. This is the basis of learning.

- Inference - the process of arriving at new facts from the given facts.

- Inference Engine - a program which performs inferences and adds or deletes them from the set of statements believed to be true. (also called a gatekeeper)

- Instance - a single unambiguous value or occurrence of something that could have many values or occurrences. 2 is an instance of an even number. A term having no variables (a ground term) is its only instance. Given a term with variables, substituting something for a variable gives a new instance

- ISA - a token representing that one object is an instance of a class of objects. For example Sten is a man.

- Knowledge base - the data base for logic programs

- Knowledge Engineering - building a set of rules that represents the knowledge and skill of a human expert.

- Lambda notation - a way of defining a function without giving it a name.

- LISP - A list processing programming language (LISP = LISt Processing)

- List structure - In LISP - a list of lists which may contain self-references (circularities)

- Literal - an atom (positive literal) or a negated atom (negative literal)

- Modus Ponens - a rule of inference - if A implies B and A is true, then B is true (i.e if Bv(~A) and A, then infer B.

- Most general unifier - A substitution leaving the most variables unbound (i.e. it subsumes every other unifier). It has the property that it is unique except for naming variations.

- Nil - The unique LISP construction that is both an atom and a (empty) list.

- Non-procedural - a program is non-procedural if the order of its statements is not relevant. Logic statements in their purest form are non-procedural.

- Occurs check - In unification, this check prevents a substitution for a variable by an expression containing that variable. (i.e., an attempt to substitute f(X) for X.) PROLOG often leaves this check out and so can get incorrect results.

- Open knowledge base - one that doesn't contain everything that is true. Therefore, if something is not in the database, you cannot conclude that it is false.

- Predicate - a function that returns true or false. A predicate states a relation among objects.

- Predicate Calculus - a system for computing on propositions that contain variables. If variables represent objects only, then the system is first order predicate calculus. If variables represent objects and predicates, then the system is second order predicate calculus.

- Program clause - a Horn clause - one with one or zero positive predicate.

- PROLOG - a logic programming language (PROLOG = "PROgrammation en Logique") for solving problems involving objects and relationships between objects. It is a resolution based theorem prover using Horn clauses. PROLOG works backwards from desired conclusions to known facts by attempting to resolve the leftmost predicate with a depth first search.

- Proposition - A statement that evaluates to true or false and contains no logic variables.

- Propositional logic - a system for computing on propositions.

- Referential ambiguity - a situation where more than one interpretation of a phrase is possible. For example, who is he in "He is a good student".

- Resolution - a general rule of inference. If one clause contains a negated literal and the other contains the same literal not negated, then you may infer the clause which is the disjunction of the other terms. If $A \vee B \vee C \vee D$ and $(\sim A) \vee E \vee F$, then you may infer $B \vee C \vee D \vee E \vee F$.

- Rule - statement that is true under some conditions (as opposed to a fact that is unconditionally true).

- S-expression - in LISP - a list of lists with no circularities.

- Search - an organized method for guessing a good path to a conclusion.

- Skolemization - the process of eliminating universal and existential quantifiers from a formula.

- Subsume - formula *P* subsumes formula *Q* if a substitution for variables in *P* produces *Q*.

- Term - argument of a predicate -- a constant, a variable, or an application of an n-ary function to n terms.

- Theorems - facts deduced from the given initial facts (the axioms)

- Token - a unique phrase or encoding whose structure is not considered relevant.

- True - anything except nil in LISP, 1 in *APL*.

- Unification - the process of finding the values of variables that make two expressions look the same. Also called finding a common instance.

- Unifier - a substitution that makes two expressions look the same.

- Unit Clause - one non-negated predicate and no negated predicates ($P\leftarrow$).

- Universal instantiation - a rule of inference - if something is true of everything, then it is true for any particular thing.

- Universal quantifier - something is true for all values of a variable.

- Variable - a token which replaces universal quantifiers. Instead of writing 'for all (x), (x<3)' write 'X<3' where X is a logic variable.

- Variant formulas - *P* and *Q* are variants if each can be produced from the other by some substitution.

- Word sense ambiguity - situation of a word having more than one meaning.

Appendix 3: A Summary of First Order Predicate Calculus

Predicate calculus is a notation useful in expressing propositions, calculating the truth of propositions, and inferring new propositions from the known ones.

The following summary is meant to be independent of the syntax used to write the notation.

There are two aspects to the notation:

- The objects being talked about

- Mappings between the objects

The objects of the language are:

- constants - a particular number or a particular character string.

- variables - names which represent sets of possible constant values.

- computed - an object resulting from a computation (see functions below).

The above set of objects are called terms.

In addition, the language contains two distinguished objects called "true" and "false". These are merely two distinguishable objects not related to actual truth or falsity except by the intention of the writer.

The mappings are:

- Functions - mappings of terms to a term

- Predicates - mappings of terms to true or false

- Formulas - predicates and combinations of predicates and formulas

The applications and combinations are:

- Atomic formula - a predicate applied to the proper number of terms

- Formula - an atomic formula or the result of any of the following combinations of formulas. If F and G are formulas and x is a variable, then the following are formulas:

- Implication: "If F then G" - this is true if F is true or G is false

- Conjunction: "F and G" - this is true if both F and G are true

- Disjunction: "F or G" - this is true if either F or G is true or both are true

- Negation: "not F" - This is true if F is false

- Existential quantification: "exists (x) F" - This is true if there is an x that makes F true

- Universal quantification: "For all (x) F" - This is true if F is true for every possible value of x

Predicate Calculus is not concerned with the actual truth of propositions, only the relationships between them. The actual truth of the input formulas is unimportant in the application of the formal rules. If a false conclusion is reached, it can only be because one of the input assumptions is wrong.

Appendix 4: Tautologies

A tautology is a statement of the form $P \vee (\sim P)$. Thus, the characteristic of a tautology is that one term appears in both the non-negated list and the negated list. Such statements are not wrong (in fact they are trivially true) but, rather, are not useful in making any new inferences.

Here is an *APL* expression that checks a clause for a tautology

```
∨.∊/Z
```

The reduction puts the ∨.∊ between the positive and the negative clause parts. If any predicate in one appears in the other, the member ship will give a 1 and so the ∨/ part of the inner product will give a 1.

Appendix 5: The DPY Function

The DPY function is like the DISPLAY function distributed as part of the *APL*2 program product except it labels the top edge of boxes with the shape of the array.

```
    ∇ D←S DPY A;⎕IO;R;C;HL;HC;HT;HB;VL;VB;V;W;N;B
[1]   ⍝  A MODIFIED DISPLAY FUNCTION
[2]   ⍝  NORMAL CALL IS MONADIC.  DYADIC CALL USED ONLY IN
[3]   ⍝     RECURSION TO SPECIFY DISPLAY RANK, SHAPE, AND DEPTH.
[4]    ⎕IO←0
[5]    ⍎(0=⎕NC 'S')/'S←⍴A'
[6]    R←↑⍴,S                       ⍝ PSEUDO RANK.
[7]    C←'..'''''                    ⍝ UR, UL, LL, AND LR CORNERS.
[8]    HL←'-'                       ⍝ HORIZONTAL LINE.
[9]    HC←HL,'⊖→',HL,'~+∊'          ⍝ HORIZONTAL BORDERS.
[10]   HT←HC[(0<R)×1+0<↑¯1↑,S]
[11]   W←,0≡¨↑0⍴⊂(1⌈⍴A)↑A
[12]   HB←HC[3÷3⌊(∨/W)+(∧/0 1∊W)+3×1<⍴⍴S]
[13]   VL←'|'                       ⍝ VERTICAL LINE.
[14]   VB←VL,'⌽↓'                   ⍝ VERTICAL BORDER.
[15]   V←VB[(1<R)×1+0<¯1↑¯1↓,S]
[16]   ⍎(0∊⍴A)/'A←(1⌈⍴A)⍴⊂↑A'    ⍝ SHOW PROTOTYPE OF EMPTIES.
[17]   →(1<≡A)/GEN
[18]   →(2<⍴⍴A)/D3
[19]   D←⍕A                         ⍝ SIMPLE ARRAYS.
[20]   W←1↑⍴D←(¯2↑1 1,⍴D)⍴D
[21]   N←¯1+1↓⍴D
[22]   →(0=⍴⍴A)/SS
[23]   D←(C[1],V,((W-1)⍴VL),C[2]),((HT,N⍴(⍕,S),N⍴HL),[0]D,[0]HB,N⍴HL),
C[0],(W⍴VL),C[3]
[24]   →0
[25] SS:HB←((0 ' ')=↑0⍴⊂A)/' -'
[26]   D←(B,B,((W-1)⍴B),B),((((⍴HT)⍴B),N⍴B),[0]D,[0]HB,N⍴B),B,(W⍴B),B←'
[27]   →0
[28] GEN:D←⍕DPY¨A              ⍝ ENCLOSED ...
[29]   N←D∨.≠' '
[30]   D←(N∨~1⌽N)⌿D
[31]   D←(∨⌿~'  '⍷D)/D
[32]   D←((1,⍴S)⍴S)DPY D
[33]   →(2≥⍴,S)↓D3E,0
[34] D3:D←0 ¯1↓0 1↓⍕⊂A             ⍝ MULT-DIMENSIONAL ...
[35]   W←1↑⍴D
[36]   N←¯1+1↓⍴D
[37]   D←(C[1],V,((W-1)⍴VL),C[2]),((HT,N⍴HL),[0]D,[0]HB,N⍴HL),C[0],
(W⍴VL),C[3]
[38] D3E:N←¯2+⍴,S
[39]   V←C[N⍴1],[0]VB[1+0<¯2↓,S],[0](((¯3+↑⍴D),N)⍴VL),[0]C[N⍴2]
[40]   D←V,D
    ∇
```

Appendix 6: Test Cases

In the following:

```
   variables - X Y Z
  predicates - p q r s t
   functions - f g h
   constants - a b c
```

** Unification Tests

These examples show unification of two predicates and the resulting common predicate if one exists along with the substitutions for variables that lead to the unification. If the predicates don't unify, the reason is given:

```
1.   a. p(X,f(X),Y)
     b. p(a,Z,g(Z))
        -----------
     c. p(a,f(a),g(f(a)))
           with substitutions X←a
                              Z←f(a)
                              Y←g(f(a))


2.   a. p(a,X,X)
     b. p(a,Y,f(Y))
        -----------
     failure -- substitution X←Y
                but then Y and f(Y) don't unify
                because of the "occurs" check.


3.   a. p(f(X),g(a,Y)),g(a,Y))
     b. p(f(X,Z),Z)
        -----------
     c. p(g(X,g(a,Y)),g(a,Y))
           with substitutions X←a


4.   a. p(f(a),g(X))
     b. p(Y,Y)
        -----------
     c. failure -- substitute Y←f(a)
           but then g(X) and f(a) don't unify.
```

```
5.   a. p(a,X,h(g(Z)))
     b. p(Z,h(Y),h(Y))           -----------
     c. p(a,h(g(a)),h(g(a)))
          with substitutionsZ←a
                                    Y←g(a)
                                    X←h(g(a))
```

** Resolution Tests

These examples do resolution of two clauses. In general, it is possible to infer more than one resolvant. In these cases, several resolvants are shown along with the unification that permitted them.

```
1.   a. p ∨ q ∨ r ∨ (~s)
     b. (~p) ∨ q ∨ (~t)
        -----------
     c. q ∨ r ∨ (~s) ∨ q ∨ (~t)
           c. from matching p

     d. q ∨ r ∨ (~s) ∨ (~t)
           d. from removing redundant term


2.   a. (~p(a)) ∨ r
     b. p(X) ∨ p(a) ∨ q
        -----------
     c. p(a) ∨ r ∨ q
           c. from unifying on first p in b

     d. p(X) ∨ r ∨ q
           d. from from unifying on second p in b
           d. contains c. as a sub-case

     e. r ∨ q ∨ r
           e. from from a. and c. or from a. and d.

     f. r ∨ q
           f. from removing redundant term
```

```
3.    a. p(a) ∨ p(b) ∨ q
      b. (~p(X)) ∨ r(X)
         -----------
      c. p(b) ∨ q ∨ r(a)
            c. from unifying on first p in a

      d. p(a) ∨ q ∨ r(b)
            d. from unifying on second p in a


4.    a. p(f(X)) ∨ p(Y) ∨ q
      b. (~p(f(Z))) ∨ r
         -----------
      c. p(Y) ∨ q ∨ r
            c. from unifying on first p in a

      d. p(f(X)) ∨ q ∨ r
            d. from from unifying on second p in a
            c. contains d. as a sub-case

      e. q ∨ r
            e. from unifying on both p of a.
            e. also from b. and c. or b. and d.
               after removing redundant r.


5.    a. p(a)
      b. (~p(X)) ∨ p(f(X))
         -----------
      c. p(f(a))
            c. from unifying on first p in b

      d. p(f(f(a)))
            d. from unifying c. with b.

      e. p(f(f(f(a))))
            e. from unifying d. with b.
               and this continues forever


6.    a. p ∨ q ∨ r
      b. (~p) ∨ (~q)
         -----------
      c. q ∨ r ∨ (~q)
            c. by unifying on p
            c. is a tautology
               because q ∨ (~q) is always true
```

```
7.   a. p(X,f(a)) v p(X,f(Y)) v q(Y)
     b. (~p(Z,f(a)) v (~q(Z))
        -----------
     c. p(X,f(Y)) v (~q(X) v q(Y)
           c. from unifying on first p in a

     d. p(X,f(a)) v (~q(X)) v q(a)
           d. from from unifying on second p in a

     e. (~q(X)) v q(a)
           e. from from unifying on both p of a.
           e. also from b. and c. or b. and d.

     f. p(X,f(a)) v p(X,f(Y)) v (~p(Y),f(a))
           f. from unifying on q
```

** Example Logic Program

```
1. input clauses:

   a. p(a,b) ←
   b. p(c,b) ←
   c. p(X,Z) ← p(X,Y) p(Y,Z)
   d. p(X,Y) ← p(Y,X)

 denial of goal:

   e. ← p(a,c)

 proof:

   f. ← p(a,Y) p(Y,c)
      by resolving e. and c.

   g. ← p(b,c)
      by resolving a. and first clause of f.

   h. ← p(c,b)
      by resolving g. and d

   i. empty clause
      by resolving h. and b.
```

This program has the property that any depth first search that uses the input clauses in any fixed order will fail to find a solution.