

# Knowledge Representation

B. Dwyer, University of Adelaide © 2004

<b>1. Introduction</b>	<b>3</b>	<b>4.10. Abstract Data Types</b>	<b>31</b>
1.1. What is Knowledge Representation?	3	4.10.1. Stacks	31
<b>2. Prolog Basics</b>	<b>5</b>	4.10.2. Operations on Sets	32
2.1. Facts	5	4.10.2.1. Representation of Sets	32
2.2. Rules	6	4.10.2.2. Faster Algorithms	35
2.2.1. Conjunction (AND)	7	4.10.2.3. Limitations	36
2.2.2. Disjunction (OR)	7	4.10.3. Representing Graphs, Relations and Functions	37
2.3. How Prolog Stores and Retrieves Facts	7	4.10.4. Priority Queues	37
2.4. How Prolog Searches for Solutions	8	<b>5. Logic</b>	<b>38</b>
2.5. AND/OR Trees	10	5.1. Propositional Calculus	38
2.6. Binding	11	5.1.1. Axioms	38
2.7. Negation by Failure	11	5.1.2. Laws	38
2.8. Recursive Definitions	12	5.1.3. Rules of Inference	38
<b>3. GNU Prolog</b>	<b>15</b>	5.1.4. Conjunctive Normal Form	39
3.1. What is GNU Prolog?	15	5.1.5. Clausal Form and Horn Clauses	40
3.2. Invoking GNU Prolog	15	5.1.6. Resolution	41
3.2.1. Interactive Queries	15	5.1.7. Proof by Refutation	42
3.2.2. Loading Programs	15	5.1.8. Horn Clause Resolution	43
3.2.3. The Line Editor	16	5.2. First-Order Predicate Calculus	43
3.3. Query Execution	16	5.2.1. Quantifiers	43
3.3.1. Tracing	16	5.2.2. Unification	44
3.3.2. Adjusting Stack Sizes	17	5.2.3. Eliminating Quantifiers	44
3.3.3. Interrupting Execution	17	<b>6. Expert Systems</b>	<b>45</b>
<b>4. More Prolog</b>	<b>19</b>	6.1. Introduction	45
4.1. The Prolog Language	19	6.2. Forward Chaining	46
4.1.1. Rules and Facts	19	6.2.1. How Explanations	46
4.1.2. Interactive Queries	20	6.3. Backward Chaining	48
4.1.3. Terms	20	6.3.1. Why Explanations	49
4.1.4. Structures	21	6.4. Comparison of Forward and Backward Chaining	53
4.1.5. Lists	22	6.5. A Reaction System	54
4.1.6. Strings	22	<b>7. Search</b>	<b>61</b>
4.1.7. Operators	22	7.1. Search Problems	61
4.1.8. Parentheses	23	7.1.1. A General Algorithm	61
4.1.9. Layout and Spacing	23	7.2. Blind Search	62
4.1.10. Comments	23	7.2.1. Breadth-first Search	62
4.2. Matching	24	7.2.2. Depth-first Search	63
4.2.1. Prolog Matching is Efficient	25	7.2.3. Uniform-cost Search	63
4.2.2. Matching Lists	25	7.2.4. Analysis of Blind Search	63
4.3. Type Testing	26	7.3. Informed Search	63
4.4. Arithmetic	26	7.3.1. Greedy Search	63
4.5. Equality and Inequality	26	7.3.2. A* Search	64
4.6. Input and Output	27	7.3.3. Analysis of Informed Search	64
4.7. Debugging	27	7.4. Other Varieties of Search	64
4.7.1. Singletons	27	7.4.1. Two-way Search	64
4.7.2. Tracing	28	7.4.2. Dynamic Programming	64
4.7.3. Trace Writes	28	7.4.3. Beam Search	64
4.8. Control of Execution	28	<b>8. Planning</b>	<b>66</b>
4.8.1. Last Call Optimisation	28	8.1. Planning Problems	66
4.8.2. Shorthands	29	8.2. Situation Calculus	66
4.8.3. Red, Green and Neck Cuts	29	8.3. Linear Planners	68
4.9. List Processing	29	8.3.1. Means-Ends Analysis	68
4.9.1. All Solutions	30	8.3.2. Iterative Deepening	68
		8.3.3. Goal Protection	69
		8.3.4. Regression Planning	69
		8.4. Partial Order Planning	70

## Contents

8.4.1. A Prolog Partial-Order Planner	72	11.4.2. Viterbi Iterations	106
8.5. <i>A Planner Buster</i>	75	11.4.3. The Baum-Welch Algorithm	106
<b>9. Reasoning under Uncertainty</b>	<b>76</b>	11.4.4. Dynamic Methods	106
9.1. <i>Ignorance and Uncertainty</i>	76	11.5. <i>Data Compression</i>	107
9.2. <i>Probability Theory</i>	76	11.5.1. Arithmetic Coding	107
9.2.1. Prior Probability	76	11.5.2. Using Markov Models	107
9.2.2. Conditional Probability	76	11.5.3. Abusing gzip	107
9.2.3. The Axioms of Probability	77	<b>12. Natural Language Processing</b>	<b>108</b>
9.2.4. The Joint Probability Distribution	77	12.1. <i>Introduction</i>	109
9.2.5. Marginalisation and Conditioning	78	12.2. <i>Grammar and Syntax</i>	109
9.2.6. Bayes' Rule	79	12.2.1. Definite Clause Grammars	110
9.2.7. Conditioning Contexts	79	12.2.2. Building a Parse Tree	110
9.2.8. Conditional Independence	79	12.2.3. Semantic Actions	111
9.2.9. Normalisation	80	<b>13. Logic Programming</b>	<b>114</b>
9.2.10. Bayesian Updating	81	13.1. <i>Types of Reasoning Systems</i>	114
9.2.11. The Monty Hall Paradox	81	13.1.1. Table-based Indexing	114
9.3. <i>Belief Networks</i>	82	13.1.2. Tree-based Indexing	114
9.3.1. Conditional Independence in Belief Networks	83	13.1.3. Unification	114
9.3.2. An Algorithm for Polytrees	84	13.2. <i>Logic Programming Systems</i>	115
9.3.3. A Prolog Program for Polytrees	85	13.2.1. Prolog	115
9.3.3.1. Monty Hall Revisited	88	13.2.2. Other Logic Programming Languages	116
9.3.4. Multiply-Connected Belief Networks	89	13.3. <i>Theorem Provers</i>	116
9.4. <i>Dempster-Shafer Theory</i>	90	13.4. <i>Forward Chaining Production Systems</i>	116
<b>10. Fuzzy Logic</b>	<b>91</b>	13.5. <i>Frame Systems and Semantic Networks</i>	117
10.1. <i>Fuzzy Set Theory</i>	91	13.6. <i>Description Logics</i>	118
10.1.1. Logical Operators	91	13.7. <i>Retraction and Truth Maintenance</i>	118
10.1.2. Linguistic Modifiers	91	<b>14. Fundamentals of Knowledge Representation</b>	<b>119</b>
10.2. <i>Fuzzy Expert Systems</i>	92	14.1. <i>Introduction</i>	119
10.2.1. Fuzzification	92	14.2. <i>Top-Level Ontology</i>	119
10.2.2. Fuzzy Inference	92	14.2.1. Categories	119
10.2.3. Aggregation	93	14.2.2. Measures	120
10.2.4. Defuzzification	94	14.2.3. Composite Objects	120
10.3. <i>A Prolog Fuzzy Logic System</i>	94	14.2.4. Events	120
10.3.1. The Knowledge Base	94	14.2.5. Fluents	121
10.3.2. The Inference Engine	95	14.2.6. Stuff	121
10.3.2.1. Some Preliminaries	95	14.2.7. Mental Constructs	121
10.3.2.2. Fuzzification	96	14.2.8. Action	122
10.3.2.3. Fuzzy Inference	97	14.3. <i>Frames and Semantic Nets</i>	122
10.3.2.4. Aggregation	97	14.4. <i>Data Normalisation</i>	125
10.3.2.5. Defuzzification	97	14.4.1. Abstract Data Structures	125
10.4. <i>Reconciling Fuzzy Logic and Probability Theory</i>	98	14.4.1.1. Operations on Relations and Graphs	126
<b>11. Random Sequences</b>	<b>100</b>	14.5. <i>An Alternative Ontology</i>	127
11.1. <i>Markov Processes and Markov Models</i>	100	<b>15. Default Reasoning</b>	<b>129</b>
11.2. <i>Classification</i>	101	15.1. <i>Implementing Frames</i>	129
11.2.1. Speech-to-Text Transcription	102	15.2. <i>Using Frames</i>	130
11.2.2. The Viterbi Algorithm	102	15.3. <i>Analysing the Input</i>	131
11.2.3. A Prolog Program to Implement the Viterbi Algorithm	103	15.3.1. Low-level Input in Prolog	131
11.3. <i>Stationary Sources and Models</i>	105	15.3.2. Filling the Slots	132
11.4. <i>Training</i>	105		
11.4.1. Training the Example Models	106		

## 1. Introduction

### 1.1. What is Knowledge Representation?

Knowledge Representation is treated here as a branch of Artificial Intelligence. The emphasis is to separate declarative knowledge from procedural knowledge. If I ask you how to get to the Town Hall, you can give me a list of directions, telling me where to turn left and right, where to keep straight on, and so on. That is *procedural* knowledge. Alternatively, you can give me a map, and leave the rest to me. The map is *declarative* knowledge. The snag with a list of directions is that if I make one mistake, I am in trouble. The advantage of the map is that, not only can I plan my route from any point, I can use the same map to find any number of destinations. My ability to plan a route is separate from the map. Likewise, we like to separate the *inference engine* from the *domain knowledge*. This is good programming. I can use my route planner in a different town by buying a new map. Our focus is on ways of representing knowledge to a particular domain, that we can exploit easily.

This desire is not unique to AI. A database system, such as MySQL, has similar objectives. The domain knowledge is stored in the database instance, and some meta-knowledge about the instance is stored in the schema. The database optimiser is able to solve an SQL query by forming a *plan* with reference to the schema (and perhaps statistics derived from the instance), then execute it. This is admirable, but unfortunately most current database systems can only express knowledge in what is called *positive-existential form*: what actually exists. They cannot deal easily with what might exist, or with rules (except built-in rules to guarantee referential integrity). In other words, what they can express, and what they can reason about, is limited by what their authors originally decided.

We want to be able to reason more flexibly than this. There are many ways in which knowledge can be represented. Familiar examples include maps, tables, English text, Chinese text, formulae and diagrams. We shall focus on First-Order Predicate Calculus. This is provably adequate to represent *anything*, although it may not be the most convenient representation in any particular case.

Because of First-Order Predicate Calculus, we shall use the programming language Prolog, both to illustrate algorithms and for practical work. Roughly speaking, Prolog solves equations in First-Order Predicate Calculus.

The inspiration for this course comes largely from two textbooks. *Russell and Norvig*<sup>1</sup> covers a wide range of AI topics with a First-Order Predicate Calculus bias. The practical project and many of the lecture examples borrow ideas from there. The other is *Bratko*<sup>2</sup>. This is an excellent book, containing an introduction to Prolog, with examples that illustrate algorithms in *Winston*<sup>3</sup>, another excellent book, but one that I like less well than *Russell & Norvig*. What I have tried to do here is to provide the analogue of *Bratko* for parts of *Russell & Norvig*.

AI culture is divided between *Scruffies* and *Neats*. Scruffies are willing to try any idea that solves a problem. Neats are only interested in ideas with a sound mathematical or logical foundation. Neats spend a lot of their time explaining why what the Scruffies have done actually works. *Russell & Norvig* are *Neats*. Occasionally we need to look at a scruffy technique, such as Fuzzy Logic, and here *Negnevitsky*<sup>4</sup> is a useful and very practical reference.

Finally, there is *Sowa*.<sup>5</sup> This compares several ways of representing knowledge, with an emphasis on conceptual graphs. This is better read *after* you have finished this course.

The course is organised around a practical project. The idea is to write an *intelligent agent* that can play the game on your behalf. The game, *Wumpus*, is a *Dungeons and Dragons* style of game in which you explore a dark maze containing a deadly wumpus. There are bottomless pits in the maze, too. You, and later, your agent, have to deduce where these dangers are located, and avoid them. The reward is a hoard of gold. Sometimes you cannot be sure where dangers lie, so you have to take a calculated risk.

The *Wumpus* project leads you into consideration of logic, probabilities, and planning. It also means you will get some experience in using Prolog. This will help you understand the lectures better. Notice that I do *not* say that the lectures will help you understand the project! They will, but generally speaking, the project has to be fairly simple, or you would never finish it; but that means it can only deal with the easiest parts of the lecture material. Consequently, the lectures are organised as follows:

- An introduction to Prolog Programming,
- Logic, Search and Statistics, which relate closely to the project,
- Material that doesn't relate to the project, which is nonetheless important, and interesting.

<sup>1</sup> S. Russell & P. Norvig, *Artificial Intelligence: A Modern Introduction*, Prentice-Hall, 2003.

<sup>2</sup> I. Bratko, *Prolog Programming for Artificial Intelligence*, Addison-Wesley, 2001.

<sup>3</sup> P.H. Winston, *Artificial Intelligence*, Addison-Wesley, 1993.

<sup>4</sup> M. Negnevitsky, *Artificial Intelligence*, Addison-Wesley, 2002.

<sup>5</sup> J.F. Sowa, *Knowledge Representation: Logical, Philosophical and Computational Foundations*, Brooks/Cole, 2000.

## Introduction

Unfortunately, this means that things are not always dealt with in the most logical order. For example, we initially assume that First-Order Predicate Calculus is *the* way to represent knowledge. It is not until late in the course that we shall have time to discuss some alternatives.

To keep this course in perspective, remember that it is an introduction to part of *Russell & Norvig*, which is itself an introduction to AI.

## 2. Prolog Basics

### 2.1. Facts

Here are some simple **facts** about the British Royal Family, expressed in Prolog:

```
female('Mary of Teck').
female('Mary').
female('Elizabeth').
female('Elizabeth II').
female('Margaret').
female('Anne').
female('Sarah').

male('George V').
male('Edward VIII').
male('George VI').
male('Henry').
male('John').
male('Philip').
male('Charles').
male('Andrew').
male('Edward').
male('David').
```

We *interpret* the facts to mean that Mary of Teck, Mary, Elizabeth, Elizabeth II, Margaret, Anne and Sarah are female, and that Edward VIII, George V, George VI, John, Philip, Charles, Andrew, Edward, and David are male. They have no inherent meaning to the computer.

`female` and `male` are called **predicates**—specifically *unary* predicates, because they have one **argument**. The constants, such as `'Mary of Teck'`, are called **atoms**.

Once Prolog has consulted the file containing the facts, we can ask Prolog whether a predicate is true for a given argument by typing a **query** following its `?-` prompt:

```
| ?- male('Charles').
yes
| ?- male('Kevin').
no
```

From this we see that Prolog uses the **closed-world assumption**: if Prolog cannot prove that something is true, it says `'no'`, when it should really say `'unknown'`.

We may also ask Prolog what arguments are valid by using a **variable**. Although atoms are enclosed in quotes, variables are *not*, and must begin with a capital letter (or underscore `'_'`):

```
| ?- female(Who).
Who = 'Mary of Teck' ? ;
Who = 'Mary' ? ;
Who = 'Elizabeth' ? ;
Who = 'Elizabeth II' ? ;
Who = 'Margaret' ? ;
Who = 'Anne' ? ;
Who = 'Sarah'
yes
```

This shows the result of an interactive dialogue. After each name but the last, Prolog typed `'?'`, indicating that there were further solutions, and I typed `';`', which told Prolog to continue.

Typing RETURN terminates a dialogue:

```
| ?- male(Who).
Who = 'Edward VIII' ? ;
Who = 'George V' ? []
yes
```

Predicates may have any (reasonable) number of arguments. The number of arguments that a predicate takes is called its *arity*. Here follows a *binary* parent predicate that shows family relationships. Two facts are written to a line, to reflect the fact that people have two parents:

```
parent('Mary', 'George V').      parent('Mary', 'Mary of Teck').
parent('Edward VIII', 'George V'). parent('Edward VIII', 'Mary of Teck').
parent('George VI', 'George V').  parent('George VI', 'Mary of Teck').
parent('Henry', 'George V').      parent('Henry', 'Mary of Teck').
parent('John', 'George V').       parent('John', 'Mary of Teck').
```

```
parent('Elizabeth II', 'George VI'). parent('Elizabeth II', 'Elizabeth').
parent('Margaret', 'George VI').   parent('Margaret', 'Elizabeth').

parent('Charles', 'Elizabeth II'). parent('Charles', 'Philip').
parent('Anne', 'Elizabeth II').    parent('Anne', 'Philip').
parent('Andrew', 'Elizabeth II').  parent('Andrew', 'Philip').
parent('Edward', 'Elizabeth II').  parent('Edward', 'Philip').

parent('Sarah', 'Margaret').       parent('Sarah', 'Anthony').
parent('David', 'Margaret').       parent('David', 'Anthony').
```

A fact such as `parent('Charles', 'Philip')` could either mean ‘Charles is the parent of Philip’ or ‘Charles has parent Philip’. Again, the fact has *no* inherent meaning to the computer. It is important for *us* to remember what we mean, in this case ‘Charles has parent Philip’. The *usual* convention is for `p(A, B)` to mean ‘The *p* of *A* is *B*’, but this is a rule that even some standard Prolog built-in predicates break.

We can use Prolog to check if a given fact is true:

```
| ?- parent('Charles', 'Philip').
yes
| ?- parent('Philip', 'Charles').
no
```

We can ask who someone’s parents are:

```
| ?- parent('Charles', Who).
Who = 'Elizabeth II' ? ;
Who = 'Philip'
yes
```

We can ask, in effect, who someone’s children are:

```
| ?- parent(Who, 'Philip').
Who = 'Charles' ? ;
Who = 'Anne' ? ;
Who = 'Andrew' ? ;
Who = 'Edward' ? ;
no
```

Or we can ask for all known relationships:

```
| ?- parent(Child, Parent).
Child = 'Mary'
Parent = 'George V' ? ;
Child = 'Mary'
Parent = 'Mary of Teck' ? ;
Child = 'Edward VIII'
Parent = 'George V' ? □
yes
```

Here, I chose to terminate the dialogue by typing RETURN after the third solution.

## 2.2. Rules

Here is an example of a Prolog **rule**:

```
child(Parent, Child) :- parent(Child, Parent).
```

A rule contains the `:-` operator, which may be read as ‘is true if’. Here, we are saying “*Parent* has child *Child*’ is true, if ‘*Child* has parent *Parent*’ is true.”

The part of the rule to the left of `:-` is called the **head**, and the part to the right is called the **body**. A *fact* is simply a special case of a rule in which the body is empty, i.e., always true. Indeed, we could have written,

```
male('Edward VIII') :- true.
male('George V') :- true.
```

and so on.

Once we have defined a rule, we may use it in queries or in other rules:

```
| ?- child('Philip', Who).
Who = 'Charles' ? ;
Who = 'Anne' ? ;
Who = 'Andrew' ? ;
Who = 'Edward' ? ;
no
```

### 2.2.1. Conjunction (AND)

We can define rules that rely on more than one **sub-goal** being true:

```
son(Parent, Son) :- child(Parent, Son), male(Son).
daughter(Parent, Daughter) :- child(Parent, Daughter), female(Daughter).
father(Child, Father) :- parent(Child, Father), male(Father).
mother(Child, Mother) :- parent(Child, Mother), female(Mother).
```

We may read a comma as ‘and’, for example, “‘*Parent* has son *Son*’ is true if ‘*Parent* has child *Son*’ **and** ‘*Son* is male’ are both true.”

Rules may introduce new variables in their bodies that do not appear in their heads:

```
grandfather(Person, Grandad) :- parent(Person, Parent), father(Parent, Grandad).
```

We may read the predicate as “‘*Person* has grandfather *Grandad*’ is true if there is some value of *Parent* for which ‘*Person* has parent *Parent*’ and ‘*Parent* has parent *Grandad*’ are both true.”

### 2.2.2. Disjunction (OR)

If a predicate has more than one rule, they are regarded as alternatives:

```
person(Person) :- female(Person).
person(Person) :- male(Person).
```

We may read this as “‘*Person* is a person’ is true if ‘*Person* is female’ **or** ‘*Person* is male’.

We may also write a disjunction using ‘;’ to mean ‘or’:

```
person(Person) :- female(Person);male(Person).
```

## 2.3. How Prolog Stores and Retrieves Facts

I want you to notice something subtle about the earlier dialogues: When we asked `parent('Charles', who)`, Prolog did *not* type ‘?’ after the 2nd solution to invite us to try again, but typed ‘yes’ and stopped; but when we asked `parent(who, 'Philip')`, it typed ‘?’ after the 4th solution, then, when we asked it to continue, it failed to find one, and typed ‘no’. Why was there a difference between the two cases?

Prolog **indexes** the facts for each predicate by their first arguments. It can retrieve the two `parent('Charles', ...)` facts instantly, and it knows there are only two of them. But to answer `parent(who, 'Philip')`, GNU Prolog searches through every possible first argument. When the solution `who=Edward` is displayed, Prolog has not exhausted all possible first arguments, and is unaware that all the remaining ones will fail.

Generally speaking, Prolog predicates run faster when their first arguments are known. Most Prolog systems don’t index facts under other arguments.

Here is another clue to how Prolog organises facts:

```
| ?- listing.
male('George V').
male('Edward VIII').
male('George VI').
male('Henry').
male('John').
male('Philip').
male('Charles').
male('Andrew').
male('Edward').
male('David').

female('Mary of Teck').
female('Mary').
female('Elizabeth').
female('Elizabeth II').
female('Margaret').
female('Anne').
female('Sarah').

parent('Mary', 'George V').
parent('Mary', 'Mary of Teck').
... and so on ...
```

We see that *predicates* are *not* listed in the order that they were written—or even in alphabetical order. This is because Prolog uses a **hash table** to store predicates: an array in which it assigns each predicate a position based

on the binary value of its name. It can convert the name of the predicate to its binary value, and hence its position in the array, in one step.

On the other hand, the order of facts *is* preserved *within* predicates. Prolog always deals with facts for a given predicate in the order written, as can be also seen in the order the solutions were displayed above for each query. This gives the programmer control over the order in which things happen.

## 2.4. How Prolog Searches for Solutions

Prolog *searches* for solutions. To do this, it uses a stack, but *not* in the way procedural languages do.

***The biggest mistake you can make in learning Prolog  
is to assume it works like any language you have used before.***

It is important to understand exactly how Prolog searches, otherwise you will *never* be able to debug or understand your programs.

We can see Prolog in action by turning on **tracing**:

```
| ?- trace.
The debugger will first creep -- showing everything (trace)
yes
{trace}
```

Remember the rule for grandfather:

```
grandfather(Person, Grandad) :- parent(Person, Parent), father(Parent, Grandad).
```

Now we type a query:

```
| ?- grandfather('Charles', Who).
1 1 Call: grandfather('Charles', _15) ?
2 2 Call: parent('Charles', _84) ?
2 2 Exit: parent('Charles', 'Elizabeth II') ?
3 2 Call: father('Elizabeth II', _15) ?
```

In the first line of the trace, we see a call of `grandfather`, with its first argument bound to `'Charles'`, and `Who` linked to an internal variable (`_15`). On the second line, Prolog has begun to check the first sub-goal of the `grandfather` rule by looking for a parent of `'Charles'`. (The internal variable `_84` corresponds to the `Parent` variable.) Prolog succeeds (`Exit`) by matching the first parent rule for `'Charles'`, binding `_84` to `'Elizabeth II'`. It now tries to satisfy the second sub-goal, with `_84 (Parent)` bound to `'Elizabeth II'`.

The numbers in the right-hand column represent the depth of recursion. They show that the calls of `parent` and `father` (Depth 2) are made within the call of `grandfather` (Depth 1).

The numbers in the left-hand column represent the position of each call on a run-time stack. For the present, *please note that the stack position is not always the same as the recursion depth*. In particular, the call for the first sub-goal (`parent`) *remains* on the stack while the second (`father`) is called.

Now recall the rule for `father`:

```
father(Child, Father) :- parent(Child, Father), male(Father).
```

The call of `father` creates a third level of recursion:

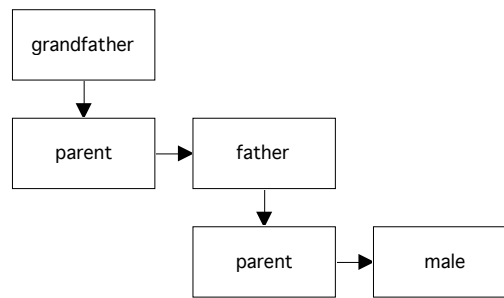
```
4 3 Call: parent('Elizabeth II', _15) ?
4 3 Exit: parent('Elizabeth II', 'George VI') ?
5 3 Call: male('George VI') ?
5 3 Exit: male('George VI') ?
```

Prolog matches the first sub-goal with the first `parent` rule for `'Elizabeth II'`, binding `_15` to `'George VI'` (`Exit`). Again, note that the call of the first sub-goal remains on the stack while the second sub-goal is called.

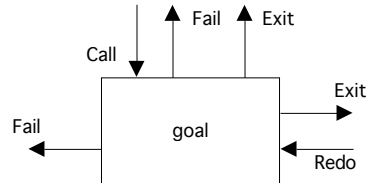
It is instructive to look at the state of the stack at this point. The vertical dimension gives the stack pointer; the horizontal dimension gives the depth of recursion:

```
1      2      3      4
1      grandfather('Charles', _15)
2          parent('Charles', 'Elizabeth II')
3          father('Elizabeth II', 'George VI')
4              parent('Elizabeth II', 'George VI')
5              male('George VI')
```

You can see that the stack spells out the rules in use. The head of the `grandfather` rule contains its body at depth 2, and the head of the `father` rule contains its body at depth 3:



In the diagram, each active rule runs from left to right. Recursion is represented vertically. In this **box model** of Prolog execution, 'call' means movement downwards, 'fail' means movement to the left (except from the first sub-goal, when the *call* fails), 'exit' (success) means progress to the right (except after the *last* sub-goal, when the *call* succeeds), and 'redo' means **backtracking** to the left, i.e., trying the next rule, if any.



In the example above, because both sub-goals of *father* succeeded, *father* itself succeeds, so *grandfather* also succeeds, returning the first solution of the query:

```
3 2 Exit: father('Elizabeth II', 'George VI') ?
1 1 Exit: grandfather('Charles', 'George VI') ?
```

```
Who = 'George VI' ? ;
```

Variable `_15` carries the binding 'George VI' all the way up to the query level. (When reading a trace, observe that the `Exit` has the same stack and depth numbers as the `Call` it matches.)

The interesting thing is that *even now* most of the calls remain on the stack. They are called **choice points**. We can see this when we type `;` to find further solutions:

```
1 1 Redo: grandfather('Charles', 'George VI') ?
3 2 Redo: father('Elizabeth II', 'George VI') ?
4 3 Redo: parent('Elizabeth II', 'George VI') ?
```

Prolog has returned to where it was when it found the solution. `Redo` means Prolog will search for alternatives.

Notice that there is *no* `Redo` for the `male` sub-goal; Prolog knows, because it has *indexed* the first argument of `male`, that `male('George VI')` is the last fact that matches the call `male('George VI')`. Therefore, it has deleted its choice point. However, it has not exhausted the possibilities for `parent('Elizabeth II', _15)`, and can bind `_15` to her mother, 'Elizabeth':

```
4 3 Exit: parent('Elizabeth II', 'Elizabeth') ?
```

Having satisfied the first sub-goal of `Call: father('Elizabeth II', _15)`, Prolog now attempts its second sub-goal, which *fails*, so the `father` goal fails too:

```
5 3 Call: male('Elizabeth') ?
5 3 Fail: male('Elizabeth') ?
3 2 Fail: father('Elizabeth II', _15) ?
```

There is no further `redo` for `parent`, because `parent('Elizabeth II', 'Elizabeth')` is the *last* fact that matches `parent('Elizabeth II', _15)`. In contrast, the choice point for `parent('Charles', _84)` is still active:

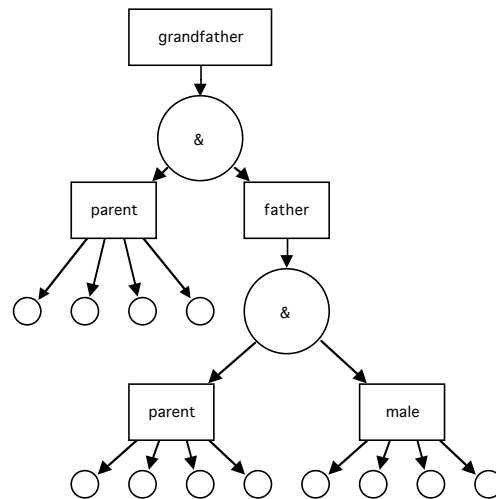
```
2 2 Redo: parent('Charles', 'Elizabeth II') ?
2 2 Exit: parent('Charles', 'Philip') ?
```

Prolog is now attempting to find the father of Charles's other parent. It is doomed to fail, because the list of facts does not include any information about Philip's parents. So, because there are no further facts to explore, everything fails, and the query is complete:

```
3 2 Call: father('Philip', _15) ?
4 3 Call: parent('Philip', _15) ?
4 3 Fail: parent('Philip', _15) ?
3 2 Fail: father('Philip', _15) ?
1 1 Fail: grandfather('Charles', _15) ?
(10 ms) no
{trace}
```

## 2.5. AND/OR Trees

Prolog's basic means of solving queries is to search an AND/OR tree:



Here, the rectangles represent OR nodes and the circles represent AND nodes. There are many `parent` rules, any of which might be chosen. Likewise, there are many `male` rules. Potentially there could be several `grandfather` rules, so it is an OR node, but there is only alternative in this example. Likewise for `father`. On the other hand, the `grandfather` rule requires that both the `parent` *and* `father` sub-goals are satisfied, and the `father` rule requires that both the `parent` *and* `male` sub-goals are satisfied. **Solutions** are sets of variable bindings that satisfy the logic of the AND/OR tree. Prolog finds them by traversing it from left to right. It will try each `parent` fact in turn, trying to satisfy the `father` predicate. It will try to satisfy this in the same way, by trying each `parent` fact in turn, trying the `male` predicate. When it fails to satisfy a predicate with its current bindings, Prolog backtracks to try different bindings. Thus, for example, if `parent` succeeds but `father` fails, it will *redo* `parent`.

The word 'fail' is rather emotive, suggesting that the program has gone wrong, but failure in Prolog is no more serious than a Boolean variable being false in a procedural language. It is a normal part of program execution. 'Backtrack' would have been a better choice of word.

Let's *start* the trace of a query that better illustrates backtracking:

```
| ?- grandfather(Who, 'George VI').
1 1 Call: grandfather(_15, 'George VI') ?
2 2 Call: parent(_15, _84) ?
2 2 Exit: parent('Mary', 'George V') ?
3 2 Call: father('George V', 'George VI') ?
4 3 Call: parent('George V', 'George VI') ?
4 3 Fail: parent('George V', 'George VI') ?
3 2 Fail: father('George V', 'George VI') ?
2 2 Redo: parent('Mary', 'George V') ?
2 2 Exit: parent('Mary', 'Mary of Teck') ?
3 2 Call: father('Mary of Teck', 'George VI') ?
4 3 Call: parent('Mary of Teck', 'George VI') ?
4 3 Fail: parent('Mary of Teck', 'George VI') ?
3 2 Fail: father('Mary of Teck', 'George VI') ?
2 2 Redo: parent('Mary', 'Mary of Teck') ?
2 2 Exit: parent('Edward VIII', 'George V') ?
3 2 Call: father('George V', 'George VI') ?
4 3 Call: parent('George V', 'George VI') ?
```

Here, because the first argument of `parent` was unbound, Prolog could not restrict its search to the two `parent` facts for 'Charles', and each `parent` fact had to be tried in turn. When `parent('Mary', 'George V')` proved a dead-end, it tried `parent('Mary', 'Mary of Teck')`, then `parent('Edward VIII', 'George V')`. Prolog went through many steps before finding a solution. Nevertheless, it eventually found several:

```
{trace}
| ?- notrace.
The debugger is switched off
yes
```

```
| ?- grandfather(Who, 'George VI').
Who = 'Charles' ? ;
Who = 'Anne' ? ;
Who = 'Andrew' ? ;
Who = 'Edward' ? ;
Who = 'Sarah' ? ;
Who = 'David' ? ;
(20 ms) no
```

## 2.6. Binding

By now, you will have noticed that which arguments of a predicate are inputs and which are outputs is not specified until execution time. When a variable has been assigned a value it is said to be **bound**; otherwise, it is **unbound**. Variables become bound typically in the process of matching a sub-goal to a specific rule head or fact.

- An unbound variable matches any constant.
- If a variable is bound to a constant, it can only match the same constant.
- A variable matches another variable by sharing. In effect, both become the same variable. Binding a constant to one therefore binds the same constant to the other.

Variables become unbound on backtracking.

*A sub-goal that fails makes no new bindings.*

Consider the following simple predicate. The idea is that `same(1, 1)` should succeed but `same(1, 2)` should fail. How does `same` behave?

```
same(A, A).
| ?- same(1, 1).
yes
| ?- same(1, 2).
no
| ?- same(X, Y).
Y = X
yes
| ?- same(X, 1).
X = 1
yes
| ?- same(1, X).
X = 1
yes
```

- In the first case, Prolog matched the first argument of `same` with 1, binding A to 1. Because A is also the second argument, it tried to match 1 with 1, and succeeded.
- In the second case, it again bound A to 1, but failed to match A (now bound to 1) with 2.
- In the third, Prolog matched A with x. Both were unbound, so they *shared*. Because Y was also unbound, it could share with A, and therefore with x.
- In the fourth case, A shared with x, but because both were unbound, A could then bind to 1, thereby binding x to 1 as well.
- In the fifth, A became bound to 1, but because x was unbound, it became bound to 1 too.

Actually, we do not need to define `same`. It is built into Prolog as the '=' operator.

## 2.7. Negation by Failure

It is well known that all AND/OR problems are *decidable*—by the method we have already outlined. Consequently, a language based solely on AND and OR operators cannot be *Turing-complete*, because every Turing-complete language includes undecidable programs. For Prolog to be Turing-complete, and therefore of some practical value, it must support *negation*. Unfortunately, this cannot be done simply by introducing a NOT operator. There are infinitely many things that are *not* a parent of Charles, for instance, and Prolog would spend a lot of time searching through them. Instead, negation is introduced in a very curious way, best shown by an example. The idea is that `different(1, 2)` should succeed, but `different(1, 1)` should fail:

```
different(X, Y) :- X=Y, !, fail.
different(X, Y) :- true.
```

Here, we have used Prolog's built-in *cut* sub-goal (!). Let's ignore it for the moment.

Consider what happens to the query `different(1, 2)`. Prolog considers the first rule of `different`, binding x to 1 and y to 2. The sub-goal `1=2` fails. Thus the whole rule fails. But Prolog has no trouble matching the second rule, so `different(1, 2)` succeeds, as it should. (The built-in sub-goal `true` is a standard Prolog predicate that always succeeds.)

Now consider what happens given the call `different(1, 1)`. Prolog tries the first rule of `different`, binding `x` to 1 and `y` to 1. The sub-goal `1=1` succeeds. The built-in sub-goal `fail` is a standard Prolog predicate that fails unconditionally. Therefore, the first rule fails because 1 and 1 are *not* different. So far so good, but unfortunately for us, Prolog would now backtrack and try the second rule. It would bind `x` to 1 and `y` to 1 and the second rule would succeed. So `different(1, 1)` would *succeed* too. In fact, `different` would succeed for any pair of arguments.

What the cut (!) does is to prevent the second rule being tried. When Prolog executes a cut, all the remaining rules *for that predicate* are ignored. In effect, part of the AND/OR tree is lopped off by deleting the choice point for the current predicate. Putting it another way, the predicate now succeeds or fails depending entirely on the sub-goals to the *right* of the cut. Therefore, in this case `different(1, !1)` fails.

We do not actually need to write `different`; Prolog's built-in `\=` operator does exactly the same thing: `1\=2` succeeds and `1\=1` fails.

More generally, the `\+` operator 'negates' *any* goal, e.g., `\+father('Charles', 'Philip')` fails if Prolog can prove that Charles's father is Philip, and succeeds if it can't. This is called **negation by failure**: whatever Prolog can't prove is assumed to be false. Read `\+` as 'Prolog can't prove that'; don't read it as 'not'.

We have to be very careful when using negation. Consider the following queries,

```
| ?- different(X, Y).
no
| ?- different(X, 1).
no
```

The results are not hard to understand. In the first query, the first rule of `different` succeeds because `x=y` succeeds, by making `x` and `y` share. Therefore, `different` fails. In the second query it binds `x` to 1, then fails. These bindings are *not* displayed, because they are released as Prolog backtracks because of `fail`.

Here is an example that uses negation,

```
sibling(Person, Sibling) :-
    mother(Person, Mother),
    child(Mother, Sibling),
    Person\=Sibling.
```

Your siblings are your mother's children, but you are not your own sibling. The following query works as you might expect:

```
| ?- sibling('Charles', Who).
Who = 'Anne' ? ;
Who = 'Andrew' ? ;
Who = 'Edward' ? ;
no
```

Now suppose we make the following change to the definition, by placing the inequality first,

```
sibling(Person, Sibling) :-
    Person\=Sibling,
    mother(Person, Mother),
    child(Mother, Sibling).
```

The results are not what we would like:

```
| ?- sibling('Charles', Who).
no
```

Why is this? The first sub-goal is `'Charles'\=Sibling`. Prolog tests if `'Charles'` and `Sibling` are different *using negation by failure*, exactly as described for the `different` predicate above. That is to say, if it fails to prove they match, they are different. So, it tries to match `'Charles'` and `Sibling`. Since `Sibling` is a variable, it simply binds it to `'Charles'`, and *succeeds*. They are not different, and `sibling` fails. The message is clear:

***Defer negations until variables are bound.***

Although Prolog can constrain two variables to be the same, by sharing, it has no means of constraining them to be different. (However, GNU Prolog supports an extension to Prolog called a **finite domain constraint solver**, which *is* capable of dealing with inequalities.)

## 2.8. Recursive Definitions

Prolog variables have a scope local to the rule in which they appear. Predicate names are global in scope. There are no intermediate scopes. The order in which predicates are written does not matter, but all the rules for a predicate should appear as a group (unless a `discontiguous` directive is used).

It is therefore possible for predicates to refer to themselves, and they frequently do. Consider the following definition of `ancestor`:

```

ancestor(Descendant, Ancestor) :- parent(Descendant, Ancestor).
ancestor(Descendant, Ancestor) :- parent(Descendant, Parent),
                                   ancestor(Parent, Ancestor).

```

The first rule expresses the fact that a person's ancestors include their parents. The second rule expresses the fact that their ancestors also include all their parent's ancestors. The predicate operates just as we should expect:

```

| ?- ancestor('Charles', Who).
Who = 'Elizabeth II' ? ;
Who = 'Philip' ? ;
Who = 'George VI' ? ;
Who = 'Elizabeth' ? ;
Who = 'George V' ? ;
Who = 'Mary of Teck' ? ;
no

```

The first two solutions are generated by the first rule. The remaining four are generated when the second rule binds `Parent` to `'Elizabeth II'`, and calls itself recursively. (We gave Prolog no facts about Philip's parenthood.) The third and fourth solutions are given by the first rule of the recursive call. The fifth and sixth are given by the second rule of the recursive call with `Parent` bound to `'George VI'`; `ancestor` calls itself again at a third level, and the results are produced by the first rule. Here is the stack when the last solution was found:

```

      1      2      3      4
1      ancestor('Charles', 'Mary of Teck')
2          parent('Charles', 'Elizabeth II')
3          ancestor('Elizabeth II', 'Mary of Teck')
4              parent('Elizabeth II', 'George VI')
5              ancestor('George VI', 'Mary of Teck')
6                  parent('George VI', 'Mary of Teck')

```

You can see that each time `ancestor` calls itself, it has a different first argument.

The same predicate may be used to find a person's descendants:

```

| ?- ancestor(Who, 'George V').
Who = 'Mary' ? ;
Who = 'Edward VIII' ? ;
Who = 'George VI' ? ;
Who = 'Henry' ? ;
Who = 'John' ? ;
Who = 'Elizabeth II' ? ;
Who = 'Margaret' ? ;
Who = 'Charles' ? ;
Who = 'Anne' ? ;
Who = 'Andrew' ? ;
Who = 'Edward' ? ;
Who = 'Sarah' ? ;
Who = 'David' ? ;
no

```

Exchanging the order of the *rules* merely changes the order in which the solutions are found:

```

ancestor(Descendant, Ancestor) :- parent(Descendant, Parent),
                                   ancestor(Parent, Ancestor).
ancestor(Descendant, Ancestor) :- parent(Descendant, Ancestor).

| ?- ancestor('Charles', Who).
Who = 'George V' ? ;
Who = 'Mary of Teck' ? ;
Who = 'George VI' ? ;
Who = 'Elizabeth' ? ;
Who = 'Elizabeth II' ? ;
Who = 'Philip'
yes

```

However, exchanging the order of the *sub-goals* is unwise:

```

ancestor(Descendant, Ancestor) :- parent(Descendant, Ancestor).
ancestor(Descendant, Ancestor) :- ancestor(Parent, Ancestor),
                                   parent(Descendant, Parent).

| ?- ancestor('Charles', Who).
Who = 'Elizabeth II' ? ;
Who = 'Philip' ? ;
Who = 'George VI' ? ;
Who = 'Elizabeth' ? ;
Who = 'George V' ? ;
Who = 'Mary of Teck' ? ;
^C
Prolog interruption (h for help) ? a
execution aborted

```

What went wrong? The first rule of `ancestor` gave the first two solutions, as before. The second two were found differently. The recursive call of `ancestor` left both arguments unbound, so its first rule tried each `parent` fact in turn until it found a parent of Charles. The last two solutions were found similarly, with a further level of recursion. Having found Charles's grandparents, the second rule of `ancestor` tried to find his great-grandparents, creating a fourth level of recursion. The first rule at this fourth level failed because, having searched through all the `parent` facts, it found none concerning Charles's great-grandparents, so the second rule was tried, creating a fifth level. Again, the first rule failed, so a sixth level was created, and so on. The computer went quiet for a long time. Prolog would have eventually run out of stack space, but I took pity, and aborted the query.

### 3. GNU Prolog

#### 3.1. What is GNU Prolog?

GNU Prolog is a Prolog system supported by the Free Software Foundation. It can be installed from a free download on Windows, Linux, Unix, Mac OS X, and probably several other systems. The download includes installation instructions, and an HTML language manual. We use it because students can install it on their home or office computers, it is amazingly cheap, and it doesn't have too many bugs. At the time of writing, its main drawback is primitive approach to garbage collection that relies heavily on virtual memory.

#### 3.2. Invoking GNU Prolog

GNU Prolog may be found in our `/usr/local/bin` directory. Provided your Unix `path` variable is set correctly, or you set up a suitable alias, you should be able to invoke it by typing `gprolog`.<sup>6</sup> The GNU Prolog system is interactive, and although it does allow stand-alone programs to be compiled, you won't need to use this feature.

GNU Prolog is broadly compliant with the ISO standard, but incorporates many extra built-in predicates that are not required by the standard. You can access the full manual on-line at [www.cs.adelaide.edu.au/docs/gprolog](http://www.cs.adelaide.edu.au/docs/gprolog), or via [www.cs.adelaide.edu.au/~third/kr](http://www.cs.adelaide.edu.au/~third/kr).

##### 3.2.1. Interactive Queries

When you run a Prolog system, you type queries, which are immediately evaluated.

GNU Prolog uses `| ?-` to prompt for a query. A query may be typed over several lines, the last of which must be *terminated by a full-stop*.<sup>7</sup> Queries may be used for interactive debugging, e.g., when you want to check what a particular predicate does.

If a query produces several answers, they may be enumerated by typing `;`. If you don't want to see all the answers, just hit RETURN (`□`).

```
| ?- member(X, [a, b, c, d, e, f]).□
X = a ;
X = b ;
X = c □
yes
```

To exit from Prolog, type the goal `halt`.

##### 3.2.2. Loading Programs

Program texts are prepared using a text editor and loaded using `consult`.<sup>8</sup>

GNU Prolog provides a shorthand for `consult`: the files to be consulted are simply typed as a bracketed list. The query,

```
| ?- [useful, patches, handy].□
```

would load the files `'useful.pro'`, `'patches.pro'` and `'handy.pro'` into working memory.

It is possible for a program file to contain (nested) `'include'` directives to load other files.

When `'consult'` reads a predicate, it first removes all rules for the predicate from memory. In GNU Prolog, unless you specify otherwise, all the rules for a predicate must be *contiguous*.<sup>9</sup> If you interleave the rules for one predicate with rules for other predicates, GNU Prolog normally assumes that the continuation of the first predicate is actually a mistaken attempt to define a *new* predicate with the same name (and arity), which is an error.

Use `listing` to see what predicates are stored in your database. Without an argument, `listing` writes *all* programmer-defined rules to the current output stream—normally your monitor. Use it during debugging to verify that the texts of predicates are what you intend. It may take a single argument of the form `pred/arity`, or a list of such arguments, e.g.,

```
?- listing([parent/2, grandchildren/3]).□
```

Unfortunately, the original names of variables are lost, and they are displayed as `'A'`, `'B'`, etc.

<sup>6</sup> Don't type `'prolog'`. This might invoke a different Prolog system.

<sup>7</sup> Next time a query seems to take forever, remember this!

<sup>8</sup> It is *possible* to enter new rules from the keyboard by typing `'consult(user).'` (or `'[user].'`), later returning to query mode by typing CTRL/D.

<sup>9</sup> The *discontiguous* directive lets you over-ride this.

### 3.2.3. The Line Editor

GNU Prolog's line editor lets you edit your current query using a comprehensive set of commands. It also maintains a history of previous lines, and enables them to be recalled:

Key	Alternative	Action
Esc-?		Display this list of commands
Ctl-B	□	Move cursor back one character
Ctl-F	□	Move cursor forward one character
Esc-B	Ctl-□	Move cursor back one word
Esc-F	Ctl-□	Move cursor forward one word
Ctl-A	Home	Move cursor to start of line
Ctl-E	End	Move cursor to end of line
Ctl-H	Backspace	Delete previous character
Ctl-D	Delete	Delete current character
Ctl-U	Ctl-Home	Delete from start of line to current character
Ctl-K	Ctl-End	Delete from current character to end of line
Esc-L		Convert next word to lower case
Esc-U		Convert next word to upper case
Esc-C		Capitalise the next word
Ctl-T		Transpose the last two characters
Ctl-V	Insert	Toggle insert/replace mode
Ctl-I	Tab	Complete the word (twice shows all possible completions)
Ctl-space		Mark the beginning of a selection
Esc-W		Copy (from beginning-of-selection mark to current character)
Ctl-W		Cut (from beginning-of-selection mark to current character)
Ctl-Y		Paste
Ctl-P	↑	Recall previous history line
Ctl-N	□	Recall next history line
Esc-P		Recall previous history line that starts with current line <sup>10</sup>
Esc-N		Recall next history line that starts with the current line
Esc-<	Page Up	Recall first history line
Esc->	Page Down	Recall last history line
Ctl-C		Interrupt the current query
Ctl-D		Generate an end-of-file character
RETURN		Validate and execute a query

## 3.3. Query Execution

### 3.3.1. Tracing

The goal `trace` turns on debug tracing, and `notrace` turns it off again. During tracing, you are informed about each `call`, `redo`, `exit`, or `fail` port *that matches the arguments of a goal*. Each trace is preceded by two integers. The first identifies the choice point, so that you can see to which `call` a given `fail`, `exit` or `redo` corresponds. The second indicates the depth of recursion.

Tracing a large program can be very tedious. You can set **spy-points** using `spy(PredList)`, where `PredList` is a list of predicate names, each of the form `Functor/Arity`. If `/Arity` is missing, *all* predicates with the name `Functor` have spy-points set. For example, the goal,

```
?- spy([a/1, b/3, c]).□
```

sets spy-points on predicate `a` of arity 1, predicate `b` of arity 3, and all `c` predicates.

A similar `nospypred` predicate allows spy-points to be removed selectively. The `nospya11` predicate removes all spy points. To find what spy points currently exist, use `debugging`.

The goal `debug` turns on debug tracing, but not until the first spy-point is reached; `nodebug` turns it off again.

<sup>10</sup> To recall the most recent line beginning with a given string, type the string, then ESC-P.

After displaying the status of a port, the debugger waits for the programmer's response, which can be any of the following:

a	abort	Abandon execution, and return control to Prolog.
A	alternatives	List the ancestors of the current goal, and their choice points
b	break	Enter the Prolog interpreter recursively.
c, or RETURN	creep	Single-step to the next <b>call</b> , <b>redo</b> , <b>fail</b> or <b>exit</b> port.
d	display	Show the current goal using 'display'.
e	exception	Show the pending exception (only at an exception port).
f	fail	Cause the current goal to fail.
g	goal	List the ancestors of the current goal.
G	go to	Ask for a call invocation number, and continue execution until a port is reached with that number.
h or ?	help	Display this list of options.
l	leap	Let execution continue uninterrupted until the next spy point.
L	listing	List the clauses of the current predicate.
n	nodebug	Turn off debug mode.
p	print	Show the current goal using 'print'.
r	retry	Transfer control to the call port of the current goal.
s	skip	Turn off tracing during execution of the current goal.
u	unify	Ask for a term, and unify the goal with it (only at a call port).
w	write	Show the current goal using 'write'.
+	spy this	Set a spy point on the current goal.
-	nospys this	Remove the spy point on the current goal.
.	father file	Display the file name and line number where the current goal is defined.
=	debugging	List the current spy points using <code>debugging/0</code> .

There are several other options, which can be found by typing 'h' (help).

### 3.3.2. Adjusting Stack Sizes

Prolog uses three storage areas: the 'control stack', the 'trail', and the 'heap'. The control stack stores goals and choice points, as explained earlier. The trail records the variable bindings that should be undone on backtracking. The heap stores compound terms (data structures).

Stack Name	Default Size (KB)	Environment Variable	Description
local	2048	LOCALSZ	Control Stack
global	4096	GLOBALSZ	Heap
trail	2048	TRAILSZ	Bindings

The default values should be adequate for most purposes. If you run out of control stack, it is likely that your program has entered an unbounded recursion, so this looks like a programming error, not a stack problem. You can easily confirm this; increase the stack size:

```
> setenv LOCALSZ 4096
```

Running out of heap space suggests that your program is wasting storage. In particular, GNU Prolog's garbage collection system seems to be less than perfect, and it appears that not all unused storage is reclaimed.<sup>11</sup> Therefore you may have to increase the heap size:

```
> setenv GLOBALSZ 100000
```

### 3.3.3. Interrupting Execution

If a query loops or takes an excessive time to execute, you can interrupt it by hitting `Ctrl-C`. This results in the message 'Prolog interruption (h for help) ?'. You may then type one of the following commands:

a	abort	Abort the query and return control to the top-level Prolog process.
b	break	Invoke a Prolog process recursively.
c	continue	Resume execution.
d	debug	Start the debugger using <code>debug/0</code> .
e	exit	Quit the current Prolog process.
h or ?	help	Display this list of commands.
t	trace	Start the debugger using <code>trace/0</code> .

<sup>11</sup> I suspect that there is *no* garbage collection at all until the query is completed.

By using the ‘break’ command, it is possible to use the Prolog interpreter *during* an interruption.<sup>12</sup> You will be presented with a ‘|?’ and the usual command line interface. The ‘exit’ command would then return you to the same state as when you used ‘break’. In contrast, ‘abort’ would return you to the top-level command-line process, discarding the interrupted process.

---

<sup>12</sup> You might, for example, want to use `listing` to check the text of a predicate, or you might want to test a dubious predicate.

## 4. More Prolog

### 4.1. The Prolog Language

The name ‘Prolog’ stands for ‘programming with logic’. This is only partly true; a *subset* of Prolog corresponds to a *subset* of logic. Prolog has many features that have little to do with logic. Nonetheless, well-written Prolog programs have a transparent, declarative style.

Here are some features of Prolog:

Feature	Pros	Cons
Automatic backtracking	Prolog’s run-time stack mechanism is more powerful than most.	Beginners have trouble understanding how a program will execute.
Pattern matching	It is easy to recognise and manipulate complex data structures. It is easy to write generic programs.	There is almost no type checking at compile-time; debugging type errors at run time is a chore.
No distinction between objects and primitive types	A number, a pointer to a number, and a pointer to a pointer to a number are all treated alike. Copying an object just involves copying a pointer.	Occasionally, subtle problems can result, e.g., if the object contains an unbound variable, binding the variable affects all ‘copies’ of the object.
Automatic garbage collection	No need to allocate or de-allocate storage explicitly.	Care should be taken not to cause unnecessary creation and deletion of data structures.
Self-describing data	Almost any data structure can be written or read without the need to write a special method.	Some (unusual) data structures can send the default output routines into a loop.
Single assignment	A variable may be assigned only one value, allowing <i>structure sharing</i> .	Programmers used to a procedural style will have to learn new habits.
Simple syntax	The syntax of Prolog is easy to learn.	All Prolog programs look alike. It is not easy to spot loops, etc.
Extendable syntax	It is possible to add new operators to those already built-in.	Changing the built-in operators can result in confusion.

#### 4.1.1. Rules and Facts

The basic building blocks of a Prolog program are **rules**. Usually, a rule consists of a **head** and a **body** separated by the ‘if’ operator, ‘:-’. A rule is terminated by a full-stop.

```
head :- body.
```

The *head* of a rule has the form of a **goal**:

```
functor(arg1, arg2, ... )
```

If a goal has *no* arguments the parentheses are omitted.

The *body* of a rule is a list of goals separated by commas. In this context, the goals are referred to as **sub-goals**. The **declarative interpretation** of a rule is that its head is true if values can be assigned to variables so that *all* the sub-goals in its body are true. In the case that a rule has *no* sub-goals, either the body may be written as ‘true’, or the head may be immediately followed by a full-stop, in which case the rule is called a **fact**.

Rules are usually loaded from files and stored in memory. They are invoked by typing **queries**. A query has exactly the same format as the body of a rule: a list of sub-goals.

A collection of rules with the same **functor** and the same number of arguments is called a **predicate**. A predicate is true if *any* of its rules are true. For example, the following two rules recursively define an ancestor predicate with two arguments.

```
ancestor(Descendant, Ancestor) :- parent(Descendant, Ancestor).
ancestor(Descendant, Ancestor) :- parent(Descendant, Parent),
                                   ancestor(Parent, Ancestor).
```

Because the predicate has two arguments, it is said to have **arity 2**, and strictly speaking, it should be referred to as `ancestor/2`; meaning it has the **functor** `ancestor` and **arity 2**. It is possible to define several predicates with the same functor but different arities.

In most versions of Prolog, including GNU Prolog, all functor names are global, although there are also versions that allow the use of **modules**. Predicates are only visible within the module in which they are defined, unless they are declared to be public. As yet, the ISO standard for Prolog does not support modules.

### 4.1.2. Interactive Queries

All Prolog systems are interactive, although many, including GNU Prolog, also allow programs to be compiled. When you run a Prolog system, you may type **queries**, which are immediately evaluated.

Most systems use ‘?-’ to prompt for a query. A query may be typed over several lines, the last of which must be terminated by a full-stop.<sup>13</sup> Queries may be used for interactive debugging, or when you want to check what a particular predicate does.

If a query produces several solutions, they can be enumerated by typing ‘;’. If you *don’t* want to see all the solutions, just hit RETURN (□).

To exit from Prolog, type the goal ‘halt.’

Facts and rules are normally read from files, using the `consult` predicate. This takes a single argument, which is a list of path names. A special shorthand is provided for `consult`. If a list of file names is typed, the files are consulted. If a file name has extension `.pl` or `.pro`, there is no need to type it. I recommend using `.pro`, otherwise Unix’s `a2ps` utility will think your programs are Perl scripts.

It is normal Prolog practice to separate knowledge of a specific domain from general knowledge used in reasoning about the domain. For the family database we have used in examples, we would store the facts about the royal family in a file called `royal.pro`, and the rules about family relationships in a file called `family.pro`. Later, we might define other files containing facts about other families, but we could still use the same set of rules.

A short GNU Prolog session might look like this,

```
> gprolog
GNU Prolog 1.2.1
Copyright (C) 1999, 2000 Daniel Diaz
| ?- [royal, family].
compiling royal.pro for byte code...
royal.pro compiled, 26 lines read - 4404 bytes written, 155 ms
compiling family.pro for byte code...
family.pro:3 warning: singleton variables [X, Y] for different/2
family.pro compiled, 73 lines read - 6954 bytes written, 154 ms
(20 ms) yes
| ?- grandfather('Charles', Who).
Who = 'George VI' ? ;
no
| ?- halt.
>
```

### 4.1.3. Terms

The basic building block of Prolog syntax is the **term**. Terms are **constants**, **variables** or **structures**. GNU Prolog supports the following constants:

- **Integers**, e.g., 0, 1, -512.
- **Characters**, which are really small integers in the range 0–255.
- **Floats**, e.g., 3.14159, -0.2e-9, 4.5E7.
- **Atoms**, which are used to name program objects, e.g., predicate names.

An **atom** may consist of *any* sequence of characters *enclosed in single quotes*, e.g., '\$50!', 'Elizabeth II', and so on. As the word ‘atom’ suggests, they are hard to split; their individual characters are not accessible unless they are first converted into strings.<sup>14</sup>

For convenience, *two kinds of atom may have their quotes omitted*: atoms beginning with a lower case letter followed by letters (either case), digits or underscores; and atoms consisting only of **symbols**. The first kind are typically used to name predicates, and the second kind are typically operators. The ‘symbols’ are:

+ - \* / \ ^ < > = ` ~ : . ? @ # \$ &

The remaining ASCII special characters have deeper syntactical purposes and are not classed as symbols. In addition, [ ], { }, and ! (cut) are special atoms.

The following are valid atoms:

```
<.>
the_Date
'Elizabeth II'
```

<sup>13</sup> The next time a query seems to take forever, remember this!

<sup>14</sup> The built-in `atom_codes` predicate can do this; `number_codes` does the same for numbers.

but the following are not:

< >	' ' is not a 'symbol'.
The_date	Capitals indicate variables.
2x	A badly formed number.
c++	There are <i>two</i> terms here: c (a variable) and ++.

Where omitting the quotes from an atom is allowed, they remain optional, e.g., 'x1' and x1 both denote the *same* atom. However, '1066' is an atom, but 1066 is an integer.

A *variable* name must begin with an *upper case letter*<sup>15</sup> or an underscore, and continue with letters (either upper or lower case), digits and underscores.

An underscore on its own is called an **anonymous variable**. Each occurrence of '\_' denotes a *new different* internal variable name. It is typically used when a variable occurs in a rule only once, or is irrelevant, for example,

```
different(X, X) :- !, fail.
different(X, Y).
```

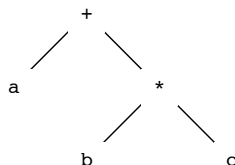
The second rule would be better written as,

```
different(_, _).
```

There are two reasons: First, Prolog knows that it is always pointless to bind anonymous variables, so using them makes a program run faster. Second, anonymous variables won't result in a warning message when the program is consulted. X and Y are referred to as **singletons**. Prolog warns you about singletons because they arise when you misspell a variable name, and singletons are therefore potential errors. *You should always replace singletons, with either the correct variable or an anonymous variable.*

#### 4.1.4. Structures

A **structure**, also called a **compound term**, may be thought of as a tree, e.g.:



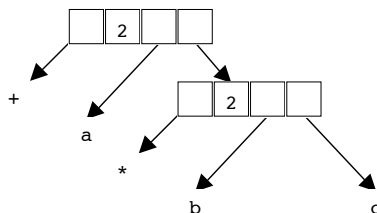
A structure may *always* be written in prefix form. The one above could be written as:

```
+(a, *(b, c)).
```

The name of any structure, here '+' or '\*', is called its **functor**. The **principal functor** of a structure is the one at the root of the tree, in this case, '+'. A functor may be any atom, but it can never be a number or a variable.

The number of children of a functor is called its '**arity**'. Functors may have any arity from 1 up to a large number, depending on the implementation (usually about 255 or 65,535), so structures can be trees of any reasonable degree. The same functor name can have different arities, for example, unary minus has arity 1, but binary minus (subtraction) has arity 2. To distinguish them, they are referred to as (-)/1, and (-)/2.<sup>16</sup> An atom is special case: a functor with arity zero.

Prolog storage is allocated dynamically. Garbage collection is automatic. Structures consist almost entirely of pointers. A structure contains a cell pointing to the symbol table entry for its functor, a cell containing the number of its arguments, and a cell for each argument.<sup>17</sup> In general, the arguments are represented by pointers to symbol table entries, pointers to other structures, or numbers.



<sup>15</sup> This is the exact reverse of the convention used in formal logic. Who knows why?

<sup>16</sup> The parentheses are needed to prevent '-' being treated as an operator here.

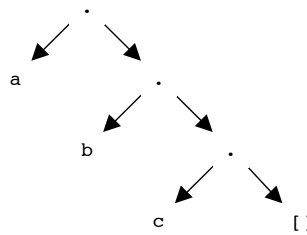
<sup>17</sup> It is possible to treat a structure of arity *N* as an array of length *N*, using 'arg/3' and 'setarg/3' or 'setarg/4'. Of course, arrays of arrays are also allowed.

Whether objects are represented by pointers or by their values is transparent to the programmer.<sup>18</sup> When a Prolog program encounters a pointer or chain of pointers, it always follows it to its destination.<sup>19</sup> You can think of variables as being represented in a small local symbol table, whose entries are pointers to the values or variables they are bound to. If a variable is unbound, its entry is a null pointer.

A Prolog *program* is a series of rules and facts, both of which are structures. (The principal functor of a rule is ‘:-’.) A query is also a structure. In early Prolog systems, this was literally true, and a Prolog program was stored as a data structure consisting of a list of facts and rules. Modern Prolog systems compile facts and rules into in-line code, but Prolog offers some special directives and predicates that allow you to treat them as data structures if necessary.

#### 4.1.5. Lists

The dot, ./2, is a binary functor used to build **lists**. Special syntax is provided for lists, because the dot also serves as a decimal point and a full-stop. The **empty list** is denoted by [ ], a special atom. The notation [ a, b, c ] denotes the following structure:



This *could* be written as:

```
'. '(a, '. '(b, '. '(c, []))
```

A properly constructed list is always terminated by the empty list ([ ]). The elements of lists may be structures if desired, including the case of lists of lists.

#### 4.1.6. Strings

Lists of ASCII codes may be used to represent **character strings**. Strings of characters enclosed in *double* quotes may be written as shorthand for ASCII lists, such as "ace" for [97, 99, 101]. Note that "ace", a list, is a structure, quite distinct from 'ace', an atom.

#### 4.1.7. Operators

Prolog’s pre-defined operators allow, e.g., ‘+(a, \*(b, c))’ to be written as, ‘a+b\*c’.

Here are Prolog’s standard pre-defined operators:

Priority	Mode	Operators
1200	xfx	:- -->
1200	fx	:-
1100	xfy	;
1050	xfx	->
1000	xfy	,
900	fy	\+
700	xfx	= \= =.. == \== @< @> @=< @>= is == =\= < > =< >=
500	yfx	+ - /\ \/
400	yfx	* / // rem mod << >>
200	xfy	** ^
200	fy	+ - \

Every operator has a **priority**; the *smaller* its priority, the *tighter* it binds its operands. It also has a **mode**, indicated by an atom such as xfy, fx or yf. The ‘f’ shows the position of the operator, ‘x’ and ‘y’ represent its operands. A ‘y’ (yes) indicates that, without needing parentheses, the corresponding operand may itself include an operator of the same precedence, an ‘x’ (X) indicates that it may not. Thus, xfy is a *right*-associative binary *infix* operator, fx is a *prefix non*-associative operator, and yf is a *left*-associative *postfix* operator. The same atom may denote both a prefix and an infix operator, but only one of each kind.<sup>20</sup>

Operators are best considered as a feature of input and output, to help readability.

<sup>18</sup> Unlike Java!

<sup>19</sup> There are some special, rarely-used operations that prevent this.

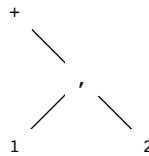
<sup>20</sup> After all, +x and x+ could only mean the same thing: +(x). An operator cannot be both infix and postfix, either.

#### 4.1.8. Parentheses

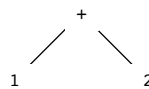
The Prolog syntax checker expects parentheses to be matched within the arguments of all operators that bind looser than a comma. This rule detects many common syntax errors, especially those involving unbalanced parentheses. The comma ‘,’ with priority 1000, binds tighter than ‘:-’, with priority 1200, forcing both the head and body of a rule to be balanced.<sup>21</sup>

#### 4.1.9. Layout and Spacing

Any atom that may be written as an operator may also be written as a predicate, e.g., ‘1+2’ may be written as ‘+(1, 2)’. There must be *no space between a predicate name and its opening parenthesis*, or it will be treated as a prefix operator. For example, because ‘+’ happens to be a unary prefix operator, and the comma is an infix operator, ‘+(1, 2)’ has the structure,



whereas ‘+(1, 2)’ has the structure,



Spaces are sometimes needed to separate two atoms; for example,  $x*-3$  is a syntax error (unless  $*-$  has been defined as an operator), but ‘ $x* -3$ ’ (or ‘ $x*(-3)$ ’) is read as  $*(x, -(3))$ .

In the ordinary way, atoms formed from letters or digits tend to alternate with those formed from symbols, so spaces aren’t needed very often.

Prolog rules and facts are terminated by ‘.’ followed by a white space character. The same rule applies to interactive queries, which must also be followed by ‘.’.

Prolog treats input from the keyboard differently from files that it consults. In the first case, it expects input to be in the form of queries. When Prolog consults a file, it automatically adds every rule to its work space.<sup>22</sup>

If you *want* add a new fact or rule to the work space interactively, you *assert* it, e.g.,

```
|?- assertz(parent('Zara', 'Anne')).
```

Conversely, it is possible to consult the keyboard:

```
|?- [user].
```

You then type a series of facts or rules, terminated by end-of-file (CONTROL-D).

#### 4.1.10. Comments

Prolog programs may (and should) contain comments. One-line comments—often to the right of some program text—are introduced by ‘%’ and continue to the end of the line. Multi-line comments are prefixed by ‘/\*’ and terminated by ‘\*/’.

<sup>21</sup> In some contexts rules can appear as arguments, for example, new rules can be added to a program during execution, using ‘assertz/1’. The goal ‘assertz(b(x, y):-c(y, x))’ makes perfect sense, but the argument of ‘assertz’ falls apart at the ‘:-’. Prolog’s syntax checker expects parentheses to be balanced when it encounters ‘:-’, as follows, ‘assertz(b(x, y):-...’, so it thinks the second closing parenthesis is missing. When an operator with a priority greater than 1000 appears in an unusual context it is necessary to use an extra level of parentheses to convince Prolog you know what you are doing: ‘assertz((b(x, y):-c(y, x)))’. In other words, operators with a priority greater than 1000 can appear within any number of levels of parentheses, except 1.

<sup>22</sup> To cause a goal to be evaluated during consulting it must be written as a *query* preceded by a ‘:-’ operator. This is important if you are declaring new operators, otherwise they won’t have any effect — at least, perhaps not in time to be useful. Other cases where you might use ‘:-’ are when you want to cause a consulted file to consult another file, or when you want a program to execute its main goal as soon as it is consulted.

## 4.2. Matching

An important feature of Prolog is its ability to match terms:

```
|?- 1+2=3.
no
|?- X=1+2.
X=1+2 ;
no
|?- X+2=1+Y.
X=1, Y=2 ;
no
|?- X+2=Y+1.
no
|?- X=Y.
X=_42, Y=_42 ;
no
```

What is going on here? Clearly, it isn't arithmetic, and it isn't assignment either.

Prolog's '=' operator succeeds if it can **match** its arguments. This is much the same as testing to see if they are the same structures, or *can be made the same* by **binding** variables.

- In the first case,  $1+2$ , which is a binary tree, does not match the atom  $3$ , so '=' fails.
- In the second case,  $x$ , which is a variable, matches anything, so '=' succeeds by *binding*  $x$  to  $1+2$ . There are no other solutions, so it fails when asked to try again.
- In the third case, the structures match if  $x$  and  $Y$  are bound as shown, but there are no other solutions.
- In the fourth case, the structures are similar,  $x$  can match  $Y$ , but  $2$  won't match  $1$ , so '=' fails.
- In the fifth case,  $x$  and  $Y$  are matched by binding them to the same internal variable.

We discussed matching earlier, but we did not discuss the matching of structures. Here is the full set of rules. Where a variable is matched with a constant or structure, it is said to become **bound** or 'instantiated' to that constant or structure. A term that contains no unbound variables is said to be **grounded**.

The rules are as follows:

- Two atoms match only if they are the same. (But ' $x$ ' and  $x$  are the same thing.)
- An unbound variable matches an atom or integer by binding to the atom or integer.
- Two unbound variables match by sharing, i.e., by becoming synonyms.
- Two structures match if their functors (names) are the same atom, they have the same arity, and their components match recursively.
- An unbound variable matches a *whole* structure. A variable cannot stand for a functor alone— $x(A, B)$  is valid syntax, but  $x(a, b)$  isn't. A variable *can* match  $x(A, B)$ .
- A bound variable is treated as equivalent to whatever it is already bound to.

Here are some more examples of matching:

```
|?- [a(b, c) = X.
X = a(b, c) ;
no
|?- [X(a, B) = x(A, b).
B = b
A = a ;
no
```

Prolog's matching is not **unification** in the sense used in mathematical logic. Most Prologs will match as follows:

```
|?- Y = foo(Y).
Y = foo(foo(foo(foo(foo(foo(foo(foo(foo(foo(
(foo(foo(foo(foo(
^C
Prolog interruption (h for help) ? a
execution aborted
```

Such infinite structures are not part of the mathematical logic idea of unification. A variable can't unify with a term containing itself. Nonetheless, this causes little trouble in practice.

It is possible to have several predicates called  $p$  with different arity, such as  $p(\_)$ ,  $p(\_, \_)$ , and so on. To distinguish them, in some contexts they may be referred to as  $p/1$ ,  $p/2$ , and so on. The pattern  $p(x)$ , with arity 1, will *not* match the pattern  $p(Y, Z)$  with arity 2—although it *will* match the pattern  $p((Y, Z))$ , setting  $x=(Y, Z)$ . (' $,$ ' is an infix operator.)

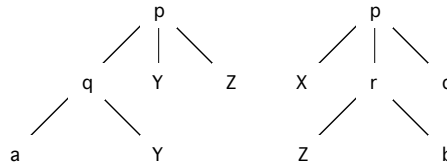
#### 4.2.1. Prolog Matching is Efficient

Despite what some textbooks say, Prolog matching is surprisingly efficient. Consider matching the following pair of terms.

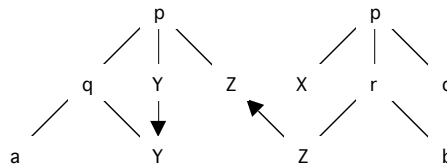
$$p(q(a, Y), Y, Z) = p(X, r(Z, b), c).$$

The solution is not obvious, and it may *seem* that a search is needed to find the unifier.

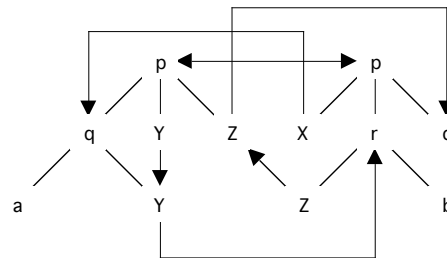
The terms correspond to the following data structures,



The first step is to chain together any occurrences of the same variable:



The next step is to match terms top-to-bottom, left-to-right. The two  $p/3$  functors match. Then their first arguments are matched by simply pointing  $x$ , which is unbound, at  $q(a, Y)$ .  $Y$  matches with  $r(Z, b)$ , by pointing the last element of its pointer chain at it.  $Z$  matches with  $c$ , again by pointing the end of its chain at it:



Matching is now complete, and no further work is done. But if Prolog is asked to *print*  $x$ ,  $y$  or  $z$ , it has to follow their pointer chains:  $x=q(a, Y)$ , where  $Y=r(Z, b)$ , so  $x=q(a, r(Z, b))$ . But  $Z=c$ , so  $Y=r(c, b)$ , and  $x=q(a, r(c, b))$ . The term  $p(q(a, r(c, b)), r(c, b), c)$  is called the **most general unifier** of the two terms. Of course, in other examples, two terms might not match, or the unifier might contain unbound variables.

In reality, matching is more efficient than this example suggests, because rather than use a general-purpose matching algorithm, a good Prolog compiler will generate special in-line code to match the particular structure involved.

#### 4.2.2. Matching Lists

When the length of a list is unknown—as it typically is, a vertical bar, ‘|’, is used to separate its ‘head’ and ‘tail’. For example  $[a|T]$  matches the list  $[a, b, c]$ , with  $T$  bound to  $[b, c]$ , and  $[a, b, c|T]$  matches it with  $T$  bound to  $[]$ . In normal use, the *left* argument of ‘|’ is one or more *elements*, but the *right* argument is a *list*.<sup>23</sup>

In processing a list, it is necessary to think about the *head*, the *tail*, and the *empty list*.

The following built-in predicate tests if an element is a member of a list:

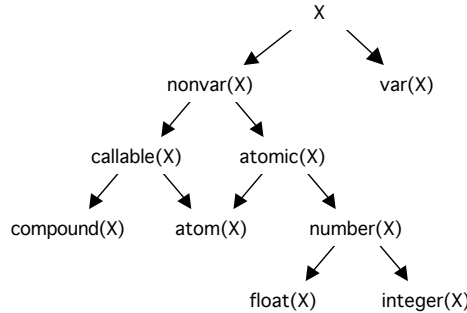
```
member(H, [H|_]).
member(H, [_|T]) :- member(H, T).
```

The *head* of a list is a member of the list. Any member of its *tail* is also a member. The *empty list* does not match either rule, which is as it should be: it has no members.

<sup>23</sup> The structure  $[a|b]$  is equivalent to  $.(a, b)$  — not a correctly formed *list* because it doesn't end with ‘|’.  
Also,  $[a, b|c, d]$  is an error; only one term can appear after ‘|’.

### 4.3. Type Testing

Prolog allows the types of variables to be tested *at execution time*. This makes it possible to write polymorphic predicates that are sensitive to type. For example, `atomic(X)` succeeds if `X` is an atom, float or integer. Here is Prolog's type hierarchy:

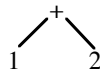


<code>var(X)</code>		X is an unbound variable.
<code>nonvar(X)</code>		X is bound.
<code>callable(X)</code>		X is bound to a structure, or an atom.
<code>compound(X)</code>		X is bound to a structure.
<code>atomic(X)</code>		X is bound to a constant.
<code>atom(X)</code>		X is bound to an atom.
<code>number(X)</code>		X is bound to an integer or float.
<code>float(X)</code>		X is bound to a float.
<code>integer(X)</code>		X is bound to an integer.

In the case that `X` is compound, the type of structure can be distinguished by pattern matching, e.g., `X=[_|_]` will only match a non-empty list.

### 4.4. Arithmetic

It is important to realise that `1+2=3` will fail, because `1+2` is the structure,



which does not match `3`, an atom. Arithmetic expressions need to be **evaluated**. This is done by the `is` operator,

```

|?- Y=2, X is 1+3*Y.
  Y = 2
  X = 7 ;
no
|?- X is 1+3*Y.
  uncaught exception: error(instantiation_error, (is)/2)
  
```

—which shows that arithmetic expressions must be *grounded* before evaluating them.

See the on-line manual for details of the allowed arithmetic operators and functions.

### 4.5. Equality and Inequality

Prolog has several ways of testing equality. Choose carefully!

<code>X=Y</code>		Tests if X and Y <i>match</i> , e.g., <code>1+A=B+2</code> succeeds, but <code>A+A=2*A</code> fails.
<code>X:=Y</code>		Tests if X and Y are equal <i>when evaluated as arithmetic expressions</i> , e.g., <code>1+2:=4-1</code> is true, but <code>X:=1</code> raises an exception unless X is already bound to a number.
<code>X is Y</code>		<i>Evaluates</i> Y as an arithmetic expression, and <i>matches</i> X with the result. Y must be grounded. X may be either bound or unbound.
<code>X==Y</code>		Tests if X and Y are <i>already the same</i> . If X and Y contain variables, they must be already bound to one another, e.g., <code>A+1==B+1</code> fails, but <code>A=B, A+1==B+1</code> succeeds.

Prolog also has various ways of testing inequality. The definition of `\=` is effectively:

```

X\=X :- !, fail.
_ \= _ .
  
```

making it the converse of `=`. It succeeds if its arguments can't be matched. Tests that imply negation should be always be deferred until after their variables have been bound.

The predicate `=\=` tests if two arithmetic expressions are unequal, so `1+2=\=3` fails, but `1+2=\=4` succeeds. The remaining arithmetic comparison operators are `<`, `>`, `=<`, and `>=`, all with the obvious meanings.<sup>24</sup>

A **standard total ordering** is defined over all terms, under which *any* two terms are comparable. For integers, floats and strings, the comparison behaves as you would expect. However, the most significant factor in standard total order is the *type* of the term: all floats, for example, come before all integers.

The standard order comparison operators are `==`, `\==`, `@<`, `@>`, `@=<`, and `@>=`, with the obvious meanings. Although they compare floats and integers in the same way as the arithmetic comparison operators, they *don't* cause expressions to be evaluated, and they *don't* compare integers *with* floats properly, e.g., `2.0@<1` is true.

The predicate `compare(Op, ?X, ?Y)` succeeds if `Op` is the result of comparing `X` and `Y` in standard order. Confusingly, `Op` takes the value `=`, `<`, or `>`, rather than `==`, `@<`, or `@>`.

## 4.6. Input and Output

Prolog supports sequential input and output from the **current input stream**, and to the **current output stream**.<sup>25</sup> The following predicates are associated with controlling streams. The default current input and output streams are called `user`.

<code>see(F)</code>	The current input stream becomes the Unix file whose name is <code>F</code> .
<code>seeing(F)</code>	<code>F</code> is matched with the file name of the current input stream.
<code>seen</code>	Closes the current input stream.
<code>tell(F)</code>	The current output stream becomes the Unix file whose name is <code>F</code> .
<code>telling(F)</code>	<code>F</code> is matched with the file name of the current output stream.
<code>told</code>	Closes the current output stream.

Input and output can be done at the character level:

<code>get0(N)</code>	<code>N</code> becomes bound to the ASCII code of the next character of the current input stream. Prolog indicates end of file by <code>-1</code> .
<code>get(N)</code>	<code>N</code> becomes bound to the ASCII code of the next <i>printable</i> or end-of-file input character. Prolog indicates end of file by <code>-1</code> .
<code>put(N)</code>	writes the character with ASCII code <code>N</code> to the current output stream.
<code>nl</code>	writes a NEW LINE sequence to the current output stream.
<code>tab(N)</code>	writes <code>N</code> spaces to the current output stream.

Prolog also supports ‘term’ input-output. **Term input** uses the same parser that Prolog uses to analyse program text. Input may span many lines, until it is terminated by a period ‘.’ and a white space character. It builds a tree structure that respects the operator declarations currently in force. Conversely, **term output** prints the structure it is given, inserting operators, parentheses, etc.

<code>read(X)</code>	Reads the next term from the current input stream.
<code>write(X)</code>	Writes the term <code>X</code> to the current output stream. Operators are displayed in prefix, infix or postfix form as appropriate.
<code>print(X)</code>	Similar to <code>write(X)</code> , except that if a <code>portray/1</code> predicate has been defined, <code>X</code> is printed as determined by <code>portray</code> .
<code>write_canonical(X)</code>	Similar to <code>write(X)</code> except that extra parentheses and quotes may be added, to ensure that the output can be read by <code>read(X)</code> .
<code>:- op(P, T, S)</code>	Operators are always displayed in functional notation. A <i>compile-time directive</i> that declares <code>S</code> as an operator with precedence <code>P</code> and type <code>T</code> (e.g., <code>xfy</code> ). Alternatively, <code>S</code> may be a <i>list</i> of operators of the same precedence and type.

## 4.7. Debugging

*Holistic* debugging is the idea that if your program doesn't work, that is because you don't understand it. (If you don't understand it, neither will any one else.) Therefore, if there are bits you don't trust, you should document them—or better still ‘self-document’ them, by choosing self-explanatory predicate and variable names—until your errors become obvious. On the other hand, you may need extra help ...

### 4.7.1. Singletons

A common error is to misspell a variable name. A variable that occurs only once within a rule is called a ‘singleton’. A singleton is almost certainly an error, as it is pointless to name a variable and never use it again.

When Prolog consults a file it will warn you of any singletons it finds.

***You should always replace singletons by their correct spellings, or by anonymous variables.***

<sup>24</sup> Note that ‘`<=>`’ and ‘`=>`’ are not predefined operators—but they would obviously make good programmer-defined operators, perhaps for logical implication.

<sup>25</sup> This is called ‘DEC-10 compatible input-output’. There are other options.

Otherwise, you will get used to seeing warnings, and ignore them even when they matter.

#### 4.7.2. Tracing

The goal `trace` starts program tracing, and `notrace` turns it off again. During tracing, the programmer is informed about the arguments of every call, `redo`, `exit` or `fail` **port**. After displaying the status of a predicate, the debugger waits for the programmer's response. Hitting `RETURN` 'creeps' to the next port, `s` 'skips' through the current call to its exit, `l` 'leaps' to the next spy point, and `a` 'aborts' the query. See the on-line manual for other options.

Tracing every port of a large program is impossibly time consuming, so the programmer can select which predicates should be monitored, by setting **spy points**. The goal,

```
?- spy([a/1, b/3, c]).
```

sets spy points on predicate `a/1` of arity 1, predicate `b/3` of arity 3, and all `c` predicates, irrespective of their arities.

A similar `nosp/1` predicate allows spy points to be removed selectively. The `nospya11/0` predicate removes all spy points. To discover what spy points currently exist, use the `debugging/0` predicate.

Tracing ultimately calls the predicate `print/1`<sup>26</sup>, defined as follows,

```
print(X) :- portray(X), !.
print(X) :- write(X).
```

where `portray` is an optional programmer-defined predicate designed to pretty-print structures in whatever format the programmer finds most helpful. Any structure that does not match `portray` is handed over to `write`. Therefore, if you decide to write a pretty printer for your data structures, call it `portray`.

#### 4.7.3. Trace Writes

An alternative to using the debugger is to insert *trace writes* into your program. Copy sub-goals into calls of `print`. For example, the `sibling` rule could be modified as follows:

```
sibling(Person, Sibling) :-
    mother(Person, Mother),
    print(child(Mother, Sibling)), nl,
    child(Mother, Sibling),
    print(child(Mother, Sibling)), nl,
    Person\=Sibling.
```

This will display the state of affairs just before calling and just after exiting `child/2`. This is more specific than setting a spy-point, because it only intercepts particular calls. A nice feature of this technique is that, if `child` fails, the printed output can be directly copied and pasted into the command line, to test the execution of `child/2`.

If you want to trace *every* call of `child/2`, you can insert,

```
child(A, B) :- print(child(A, B)), nl, fail.
```

immediately before the first rule of `child/2`. After displaying the goal, the rule will fail, and execution will continue as if the trace write never occurred.

## 4.8. Control of Execution

### 4.8.1. Last Call Optimisation

Prolog systems perform **last call optimisation**, which means that if the final rule of a predicate is **deterministic**, i.e., does not require a choice-point to be stored, it is replaced by iteration—successive calls do not use more and more of the run-time stack, but re-use the same stack space.<sup>27</sup> This is just as well, as Prolog offers no other way to specify iteration.

Here are two ways to define a `factorial` predicate:

```
factorial(0, 1).
factorial(N, F) :- N > 0, N1 is N-1, factorial(N1, F1), F is N*F1.
factorial(N, F) :- factorial(0, 1, N, F).
factorial(N, F, N, F) :- !.
factorial(N0, F0, N, F) :- N1 is N0+1, F1 is N0*F0, factorial(N1, F1, N, F).
```

Although the second definition is harder to understand than the first, it will prove more efficient. Prolog is aware that calls to `is` are deterministic, so last call optimisation will occur.

<sup>26</sup> Except that GNU Prolog does not support this usual Prolog feature.

<sup>27</sup> LCO is suppressed during tracing, so you cannot observe it happening.

(The idea of a function is alien to Prolog, so that `F=factorial(N)`, for example will *not* set `F` to the factorial of `N`.<sup>28</sup>)

#### 4.8.2. Shorthands

Prolog provides additional control predicates that, although they add little power to the language, save the programmer from having to define many trivial predicates.

The operator `;` allows alternatives to be nested within a single rule. For example:

```
a(X) :- b(X), (c(X);d(X)), e(X).
```

is almost equivalent to:

```
a(X) :- b(X), c(X), e(X).
a(X) :- b(X), d(X), e(X).
```

but more efficient, because both alternatives will be checked within only one call of `b`. It is not completely equivalent, because `b` might have a side-effect, such as writing output. Neither is it exactly equivalent to:

```
a(X) :- b(X), f(X), e(X).
f(X) :- c(X).
f(X) :- d(X).
```

because if `c` or `d` were replaced by a *cut*, it would discard the choice points for `a`, not `f`.

An extension of this idea is the ‘if...then...else’ structure:

```
a(X) :- (b(X) -> c(X)
        ;d(X) -> e(X)
        ;f(X)
        ).
```

which may be read as, “If `b` then `c` else if `d` then `e` else `f`.” The above is roughly equivalent to,

```
a(X) :- b(X), !, c(X).
a(X) :- d(X), !, e(X).
a(X) :- f(X).
```

The `->` operator has a precedence nicely between `;` and `,`.

Two other predicates help program readability: `fail`, which always fails; and `true`, which always succeeds. Finally, `abort` cancels the current query.

#### 4.8.3. Red, Green and Neck Cuts

We have seen a number of uses of the cut (`!`) predicate. Sometimes a cut improves the efficiency of a predicate, but does not change its behaviour:

```
factorial(0, 1) :- !.
factorial(N, F) :- N>0, N1 is N-1, factorial(N1, F1), F is N*F1.
```

Here, the cut merely serves to prevent the second rule being considered if the first succeeds. It would result in a tiny improvement in efficiency. It does not change the behaviour of `factorial` in any way. It is an example of a **green cut**.

```
factorial(0, 1) :- !.
factorial(N, F) :- N1 is N-1, factorial(N1, F1), F is N*F1.
```

Here, with the goal `N>0` omitted, the predicate will not function correctly without the cut. It is an example of a **red cut**.

Both are examples of **neck cuts**. Neck cuts come immediately after the head of the rule that deals with a special case. They prevent special cases matching later, more general rules.

### 4.9. List Processing

Because Prolog provides a handy syntax for lists, they are widely used to represent **sets** or **sequences**. When a list represents a *sequence*, the order of terms matters, and duplicate elements are allowed. When a list is used to represent a *set*, the order of the terms doesn’t matter, and each element should be different. A set with duplicate elements is called a **bag**. (Therefore, a bag is a *function* from each element onto an integer equal to the number of its occurrences.) A set may be represented by an unordered list, or by an ordered (sorted) list.

<sup>28</sup> However, certain pre-defined functions are allowed within the context of `is/2`, as in `Y is sin(X)`. Nor is there anything to stop you writing your own function evaluator if you want to. Indeed, Prolog is an ideal language for writing compilers or interpreters.

The built-in `memberchk/2` predicate can be used to test list membership:

```
| ?- memberchk(b, [a, b, c]).
yes
```

Similarly, `member/2` can be used to enumerate the elements of a list from head to tail:

```
| ?- member(X, [a, b, c]).
X = a ? ;
X = b ? ;
X = c ? ;
no
```

A third built-in predicate is one that concatenates two lists:

```
append([], L, L).
append([H|T0], L, [H|T]) :- append(T0, L, T).
```

The first rule deals with an *empty list*. It is easy to append anything to an empty list. The second rule deals with the head and the tail: it copies the *head* to the head of the result, then calls itself iteratively to copy the *tail*. The execution time of this predicate is linear in the length of its first argument. It may be used reversibly, to answer questions such as “What pairs of lists could be concatenated to yield a given result?”

Like `member`, the built-in `select` predicate may be used to select each member of a list in turn, but it also returns the list that remains when the member is removed:

```
select(H, [H|T], T).
select(X, [H|T0], [H|T]) :- select(X, T0, T).
```

Since it contains no cut, this predicate is fully reversible. If the first and second arguments are bound, it has the effect of deleting an occurrence of an element from a list. If the first and third arguments are bound, it has effect of inserting the element, in all possible positions.

The similar `delete` predicate deletes *all* occurrences of an element from a list, but note that strict equality (`==`) is required; it won't match a variable with an atom, for example:

```
delete([], _, []).
delete([H|T0], X, T) :- H==X, !, delete(T0, X, T).
delete([H|T0], X, [H|T]) :- delete(T0, X, T).
```

#### 4.9.1. All Solutions

We have seen how we can find all George VI's grandchildren, one at a time, by backtracking. What do we do if we want to form a *list* of all his grandchildren? Here is one approach:

```
grandchildren(Grandfather, Grandchildren) :-
    grandchildren(Grandfather, [], Grandchildren).

grandchildren(Grandfather, Known, Grandchildren) :-
    grandfather(Grandchild, Grandfather),
    \+member(Grandchild, Known),
    grandchildren(Grandfather, [Grandchild|Known], Grandchildren).
grandchildren(_, Grandchildren, Grandchildren).
```

The idea is that we start with an empty list, and add new grandchildren to it. The first rule of `grandchildren/3` finds (by backtracking) a grandchild that is not already a member of the list, then calls itself with an augmented list. Once the list contains all possible grandchildren, the first rule of `grandchildren/3` will fail; its second rule then hands back the list as its final result.

The problem with this definition is its efficiency. It will find the first grandchild quickly, but to find the second it will have to generate and reject the first solution, to find the third, it will have to generate and reject the first two solutions, and so on. In short, the execution time will increase with the square of the number of grandchildren. Fortunately, Prolog provides three, rather similar, built-in predicates that will find the list in linear time. The simplest is `findall`:

```
grandchildren(Grandfather, Grandchildren) :-
    findall(Grandchild, grandfather(Grandchild, Grandfather), Grandchildren).
```

`findall` takes three arguments. The third is the list we want to form, the first is an example of what want the list to contain, and the second is a goal that the members of the list should satisfy. We can read the above goal as “Find all instances of `Grandchild`, such that `grandfather(Grandchild, Grandfather)` is true, and call the result `Grandchildren`.”

The `bagof` relation is similar, and works identically in this case. However, if their second arguments contain unbound variables, the two predicates differ:

```
grandchildren(Grandfather, Grandchildren) :-
    bagof(Grandchild,
        (parent(Grandchild, Parent), father(Parent, Grandfather)),
        Grandchildren).
```

The difference is that `bagof` will produce (on backtracking) a different list for each value of `Parent`, whereas `findall` would continue to produce a single list as before. However, we can tell `bagof` to combine the results for all parents by marking `Parent` as an *existential variable*:

```
grandchildren(Grandfather, Grandchildren) :-
    bagof(Grandchild,
        Parent^(parent(Grandchild, Parent), father(Parent, Grandfather)),
        Grandchildren).
```

In this way, `bagof` can produce the same results as `findall`, so `findall` is, strictly speaking, redundant.

Finally, `setof` is similar to `bagof`, except that duplicate results are eliminated. For example, if there is more than one way to be the grandfather of the same grandchild (which would have to be through incest in this case), `bagof` would include the grandchild more than once, but `setof` wouldn't. The list generated by `bagof` contains results in the order of solution, but `setof` sorts its list into order. It is effectively `bagof` followed by `sort`.

A word of warning: `findall`, `bagof` and `setof` *never return an empty list*, they fail.

As an example of how a more complex first argument may be used, consider the following queries,

```
| ?- findall(mating(Father, Mother),
| ?-      (father(Child, Father), mother(Child, Mother)),
| ?-      Mates).
```

```
Mates = [mating('George V', 'Mary of Teck'), mating('George V', 'Mary of Teck'),
mating('George V', 'Mary of Teck'), mating('George V', 'Mary of Teck'),
mating('George V', 'Mary of Teck'), mating('George VI', 'Elizabeth'),
mating('George VI', 'Elizabeth'), mating('Philip', 'Elizabeth II'), mating('Philip',
'Elizabeth II'), mating('Philip', 'Elizabeth II'), mating('Philip', 'Elizabeth II')]
```

```
| ?- setof(mating(Father, Mother),
| ?-      Child^(father(Child, Father), mother(Child, Mother)),
| ?-      Mates).
```

```
Mates = [mating('George V', 'Mary of Teck'), mating('George VI', 'Elizabeth'),
mating('Philip', 'Elizabeth II')]
```

## 4.10. Abstract Data Types

Lists may be used to implement many common data types.

### 4.10.1. Stacks

A list may be used to represent a push-down stack with operations 'push', 'pop', and 'empty'.

```
push(X, S, [X|S]).
pop(X, [X|S], S).
empty([]).
```

These definitions are so trivial that no-one bothers to define them explicitly. Just remember that a list can be used as a stack.

A stack may be used to reverse a list. Here is a naïve definition of `reverse/2`.

```
reverse([], []).
reverse([H0|T0], R) :- reverse(T0, T), append(T, [H], R).
```

An *empty list* is easy to reverse. To process a non-empty list, reverse its *tail*, then append a list containing the *head* of the list after the tail. (The head itself is not (typically) a list, but `append` expects a list for its second argument, so we write `[H]`).

This definition has two drawbacks: First, its execution time is proportional to the square of the length of the list. It will call `append` as many times as there are elements, and the execution time of `append` is linear in the lengths of the lists it processes. Second, it is recursive, in a way that can't use last call optimisation.

Any sensible programmer would reverse a list by pushing its elements onto an empty stack. Where is the stack to come from? Here's where:

```
reverse(List, Result) :- reverse3(List, [], Result).
```

The second argument of `reverse3` is the stack, initially empty. We now need two rules: one to push the head of the list onto the stack, and the other to copy the stack to the result when the list becomes empty:

```
reverse3([Head|Tail], Stack, Result) :- reverse3(Tail, [Head|Stack], Result).
reverse3([], Stack, Stack).
```

This solution executes in time linear in the length of the list *and* allows last call optimisation.

This is an example of a common logic programming trick. A *loop* is constructed by writing,

- A rule to initialise the working variables,
- One or more rules to provide the loop body, and
- One or more exit rules.

#### 4.10.2. Operations on Sets

The following is a (fairly minimal) set of predicates, which we shall use to operate on sets.

$E \in S$	<code>in(+E, +S)</code>	succeeds once if $E$ matches an element of $S$ .
$\exists E: E \in S$	<code>all(?E, +S)</code>	succeeds if $E$ matches an element of $S$ , enumerating all possible values of $E$ on backtracking.
$ S $	<code>size(+S, ?N)</code>	succeeds if $N$ is the number of elements of $S$ .
$S = \emptyset$	<code>empty(?S)</code>	succeeds if $S$ is the empty set.
$S = \{E : G(E)\}$	<code>findset(+E, +G, ?S)</code>	succeeds once if $S$ is the set of all elements $E$ such the goal $G$ is satisfied.
$\exists X(S = \{E : G(E, X)\})$	<code>findsets(+E, +G, ?S)</code>	succeeds if $S$ is the set of all elements $E$ such the goal $G$ is satisfied. If $G$ contains free variables, on backtracking, $S$ is instantiated for each combination of the free variables.
$E \in S_1 \cap S_2 = S_1 \setminus \{E\} \cap (X \in S_2 \rightarrow X > E)$	<code>least(+S1, -E, -S2)</code>	succeeds if $E$ is the least element of $S_1$ in standard order, and $S_2$ is the set of $S_1$ 's remaining elements.
$S = S_1 \cup S_2$	<code>union(+S1, +S2, ?S)</code>	succeeds if $S$ is the smallest set that contains every element that is either a member of $S_1$ or of $S_2$ .
$S = S_1 \cap S_2$	<code>intersection(+S1, +S2, ?S)</code>	succeeds if $S$ is the smallest set that contains every element that is both a member of $S_1$ and of $S_2$ .
$S = S_1 \setminus S_2$	<code>difference(+S1, +S2, ?S)</code>	succeeds if $S$ is the smallest set that contains every element of $S_1$ that is not a member of $S_2$ , i.e., the asymmetric difference.
$S_1 \subseteq S_2$	<code>subset(?S1, +S2)</code>	succeeds if every element of $S_1$ is an element of $S_2$ , i.e., $S_1 \subseteq S_2$ .
$S_1 \subset S_2$	<code>proper_subset(?S1, +S2)</code>	succeeds if $S_1 \subseteq S_2$ and $S_1 \neq S_2$ .
$S_1 \cap S_2 = \emptyset$	<code>disjoint(+S1, +S2)</code>	succeeds if $S_1$ and $S_2$ have no elements in common, i.e., $S_1 \cap S_2 = \{\}$ .
$S_1 = S_2$	<code>equal(?S1, ?S2)</code>	succeeds if every element of $S_1$ is an element of $S_2$ , and <i>vice versa</i> .

It is conventional to document Prolog predicates with templates, in which  $+x$  indicates that  $x$  must be bound,  $-x$  indicates that  $x$  must be unbound, and  $?x$  indicates that  $x$  may be either bound or unbound. Some Prolog systems can check that goals conform to templates at execution time.

One of the motivations for doing this is to make programs clearer: if we need to find the union of two sets in a particular algorithm, we shall use `union/3`, even if there exists a more efficient shortcut in that particular context. A second motivation is to raise the level of abstraction in the example programs. The third is to demonstrate how Prolog is able to perform dynamic type checking.

In addition, we assume that when the programmer wishes to define a set explicitly, mathematical notation will be used, e.g.,  $\{a, b\}$  or  $\{\}$ .

##### 4.10.2.1. Representation of Sets

In `~third/kr/sets.pro`, we have implemented an ordered list representation of sets, tagging the list with the functor `set`, so that  $\{red, green, yellow\}$  would be internally represented by `set([green, red, yellow])`. The tag ensures that we cannot mistakenly treat a set as a list, or a list as a set. However, the tagged format is not ideal from the readability point of view. We therefore will make it possible to write  $\{red, green, yellow\}$  to denote the set  $\{red, green, yellow\}$ . Similarly, we shall arrange to print `set([green, red, yellow])` as  $\{green, red, yellow\}$ .

This approach has an unavoidable drawback. Because of the way Prolog parses the ‘,’ operator, we observe the following unfortunate behaviour.

```
| ?- X={red, blue}, (yellow, green)}.
X = {(red, blue), green, yellow}
yes
```

Therefore, we must avoid using elements whose principal functor is a comma.

The first predicate we define converts a set, either in its internal form or in mathematical notation, into an ordered list:

```
set2list(set(List), List).
set2list({}, List) :- !, List=[].
set2list({Elements}, List) :- !, set2list2(Elements, List1), sort(List1, List).
```

The first rule deals with a set in internal format. The second and third rules deal with a set written in mathematical notation. Prolog’s built-in `sort` predicate sorts a list into standard order. A **standard total ordering** is defined over all terms: *any* two terms are comparable. For integers, floats and strings, the comparison is as you would expect.<sup>29</sup> However, the most significant factor in standard total order is the *type* of the term: all floats, for example, come before all integers, so  $2.0 < 1$ .

```
set2list2(Element, [Element]) :- var(Element), !.
set2list2((Element, Rest), [Element|List]) :- !, set2list2(Rest, List).
set2list2(Element, [Element]).
```

The second rule above deals with case of a pair of the form ‘,’ (Element, Rest), in which case Element is taken to be an element of the set, and Rest contains its remaining elements. The third rule applies in the remaining cases; any other structure is treated as an element of the set. The first rule is needed in case an element is unbound. An unbound element would match anything, so the second rule would apply, creating an infinite list.

Apart from its primary use, `set2list` can be used to convert an ordered list back into its internal set representation:

```
| ?- set2list({a,b,c},L).
L = [a,b,c]
yes
| ?- set2list(S,[a,b,c]).
S = set([a,b,c])
yes
```

The `portray` predicate displays a set in mathematical notation:

```
portray(set([])) :- !, write('{}').
portray(set([Element|List])) :- write('{'), portray2(Element, List), write('}').
portray2(Element, []) :- print(Element).
portray2(Element, [Element1|List]) :-
    print(Element), write(', '), portray2(Element1, List).
```

`portray/1` is a special predicate, because Prolog’s built-in `print` predicate is defined as follows,

```
print(X) :- portray(X), !.
print(X) :- write(X).
```

In other words, `print` will first call `portray` to display `x` prettily, but if `portray` has no rule that matches `x`, it will call `write` to display `x` in a default format.<sup>30</sup> Because `portray2` calls `print`, it is possible to display sets of sets, and so on, correctly.

The next predicate we define is `in/2`. This is essentially the same as GNU Prolog’s built-in `memberchk/2`:

```
in(Element, Set) :- (var(Element);var(Set)), !, throw(in(Element, Set)).
in(Element, Set) :- set2list(Set, List), memberchk(Element, List).
```

Its first rule displays throws an exception if either argument is a variable.<sup>31</sup> The second rule converts `Set` to a list, then calls `memberchk/2`.

<sup>29</sup> However, GNU Prolog sorts positive and negative integers wrongly on some machines.

<sup>30</sup> The implementation used in `~third/kr/` is actually more complex than this, because it tries to avoid splitting elements across line boundaries, while working around a bug in GNU Prolog.

<sup>31</sup> Experience has shown that programmers often confuse `in/2` with `all/2`, defined shortly.

The `all/2` predicate is superficially similar:

```
all(Pattern, Set) :- var(Set), !, throw(all(Pattern, Set)).
all(Pattern, Set) :- set2list(Set, List), !, member(Pattern, List).
```

The first argument of `all/2` may be a variable, or a pattern. It may therefore be either bound or unbound. However, its second argument should not be a variable; if it is, the first rule writes an error message. Otherwise, the second rule converts the set to a list, and calls `member/2`. The main difference between `in/2` and `all/2` is that `in` fails on backtracking, whereas `all` will instantiate `Pattern` to each matching element of `Set` in turn. (If `Pattern` is a variable, `all` will instantiate it to every element of `Set`.)

```
| ?- all(pit(X, Y), {gold(1, 4), wumpus(3, 4), pit(2, 2), pit(4, 1)}).
X = 2
Y = 2 ? ;
X = 4
Y = 1 ? ;
no
```

The next pair of predicates are analogous to `findall` and `setof`, but return sets rather than lists. They differ in one important respect: Where `findall` or `setof` would fail, `findset` and `findsets` return empty sets.

```
findset(Pattern, Goal, Set) :-
    findall(Pattern, call(Goal), List),
    sort(List),
    set2list(Set, List), !.
findset(_, _, Set) :- set2list(Set, []).

findsets(Pattern, Goal, Set) :-
    setof(Pattern, call(Goal), List),
    set2list(Set, List), !.
findsets(_, _, Set) :- set2list(Set, []).
```

Their behaviour differs only when `Goal` contains free variables. On backtracking, `findsets` will return a different value of `Set` for each assignment of the free variables; `findset` will return a single, combined, result. There is no way to specify existential variables, like those allowed by `setof`.

The `least/3` predicate allows an algorithm to step through the elements of a set in standard order without resorting to backtracking:

```
least(Set0, Element, Set) :- var(Set0), !, throw(least(Set0, Element, Set)).
least(Set0, Element, Set) :- set2list(Set0, [Element|List]), set2list(Set, List).
```

The first rule ensures that the first argument is bound. The second rule converts the set to a list, then isolates the first element of the list.

To find the number of elements in a set, we find the length of the corresponding list, using GNU Prolog's `length/2` predicate:

```
size(Set, Size) :- var(Set), !, throw(size(Set, Size)).
size(Set, Size) :- set2list(Set, List), length(List, Size).
```

Unfortunately, we cannot test if a set is empty by writing `Set={}`, because an empty set can also be represented as `set([])`, which will not match `{}`:

```
empty(Set) :- set2list(Set, []).
```

`subset/2` succeeds if its first argument is a subset of its second argument, which should be bound. If its first argument is a variable or pattern, on backtracking, it will instantiate it in every possible way. This results from the behaviour of GNU Prolog's built-in `sublist` predicate:

```
subset(Set0, Set1) :- var(Set1), !, throw(subset(Set0, Set1)).
subset(Set0, Set1) :-
    set2list(Set0, List0), set2list(Set1, List1), sublist(List0, List1).
```

Two equal sets have the same internal representation:

```
equal(Set0, Set1) :- set2list(Set0, List), set2list(Set1, List).
```

`proper_subset/2` succeeds if every member of its first argument is a member of its second, but the sets are not equal. Its second argument should be bound:

```
proper_subset(Set0, Set1) :- var(Set1), !, throw(proper_subset(Set0, Set1)).
proper_subset(Set0, Set1) :- subset(Set0, Set1), \+equal(Set1, Set0).
```

The remaining operations are easy to define, given those above. At this stage, we aim for simplicity.

`union/3` finds the set of all elements that are members of either of its first two arguments, which should be bound:

```
union(Set0, Set1, Set) :- (var(Set0); var(Set1)), !, throw(union(Set0,Set1,Set)).
union(Set0, Set1, Set) :-
    findset(Element, (all(Element, Set0); all(Element, Set1)), Set).
```

`intersection/3` finds the set of all members of both its first two arguments, which should be bound:

```
intersection(Set0, Set1, Set) :-
    (var(Set0); var(Set1)), !, throw(intersection(Set0, Set1, Set)).
intersection(Set0, Set1, Set) :-
    findset(Element, (all(Element, Set0), in(Element, Set1)), Set).
```

`difference/3` finds the set of members of its first argument that are not members of its second argument. The first two arguments should be bound:

```
difference(Set0, Set1, Set) :-
    (var(Set0); var(Set1)), !, throw(difference(Set0, Set1, Set)).
difference(Set0, Set1, Set) :-
    findset(Element, (all(Element, Set0), \+in(Element, Set1)), Set).
```

`subset/2` succeeds if every member of its first argument is a member of its second. Both arguments should be bound:

```
subset(Set0, Set1) :- var(Set1), !, throw(subset(Set0, Set1)).
subset(Set0, Set1) :-
    set2list(Set0, List0), set2list(Set1, List1), sublist(List0, List1).
```

`disjoint/2` succeeds if its first and second arguments have no members in common. Both arguments should be bound:

```
disjoint(Set0, Set1) :- (var(Set0);var(set1)), !, throw(disjoint(Set0, Set1)).
disjoint(Set0, Set1) :- intersection(Set0, Set1, {}).
```

The problem with these definitions is that many of them take time proportional to the product of the sizes of the sets involved. By *merging* lists in standard order, we can reduce the complexity to the sum of their lengths.

#### 4.10.2.2. Faster Algorithms

The actual implementation of set operations in `sets.pro` use Prolog's `compare/3` predicate to compare the heads of the lists. The smaller of the two is dealt with first. (We do not repeat here the code that checks that arguments are bound.)

```
union(Set0, Set1, Set) :-
    set2list(Set0, List0), set2list(Set1, List1),
    union3(List0, List1, List),
    set2list(Set, List).

union3([], List, List).
union3(List, [], List).
union3([Head0|List0], [Head1|List1], List) :-
    compare(Rel, Head0, Head1),
    union6(Rel, Head0, Head1, List0, List1, List).

union6(<, Head0, Head1, List0, List1, [Head0|List]) :-
    union3(List0, [Head1|List1], List).
union6(=, Head, _, List0, List1, [Head|List]) :-
    union3(List0, List1, List).
union6(>, Head0, Head1, List0, List1, [Head1|List]) :-
    union3([Head0|List0], List1, List).
```

`union3` finds the intersection of two lists by merging them sequentially. After dealing with the simple cases where one or the other list is exhausted, it uses `compare/3` to determine the ordering of the heads of the lists. `union6` deals with the three possible cases.

`intersection/3` is implemented similarly to `union/3`:

```
intersection(Set0, Set1, Set) :-
    set2list(Set0, List0), set2list(Set1, List1),
    intersection3(List0, List1, List),
    set2list(Set, List).
```

More Prolog

```
intersection3([], _, []).
intersection3(_, [], []).
intersection3([Head0|List0], [Head1|List1], List) :-
    compare(Rel, Head0, Head1),
    intersection6(Rel, Head0, Head1, List0, List1, List).

intersection6(<, _, Head1, List0, List1, List) :-
    intersection3(List0, [Head1|List1], List).
intersection6(=, Head, _, List0, List1, [Head|List]) :-
    intersection3(List0, List1, List).
intersection6(>, Head0, _, List0, List1, List) :-
    intersection3([Head0|List0], List1, List).
```

difference/3 completes the trio:

```
difference(Set0, Set1, Set) :-
    set2list(Set0, List0), set2list(Set1, List1),
    difference3(List0, List1, List),
    set2list(Set, List).

difference3([], _, []).
difference3(List, [], List).
difference3([Head0|List0], [Head1|List1], List) :-
    compare(Rel, Head0, Head1),
    difference6(Rel, Head0, Head1, List0, List1, List).

difference6(<, Head0, Head1, List0, List1, [Head0|List]) :-
    difference3(List0, [Head1|List1], List).
difference6(=, _, _, List0, List1, List) :-
    difference3(List0, List1, List).
difference6(>, Head0, _, List0, List1, List) :-
    difference3([Head0|List0], List1, List).
```

We can improve some of these predicates further. This illustrates some subtle points about tweaking Prolog for the greatest efficiency

Prolog uses **table-based indexing** on the *predicate name* and the *principal functor* of its first argument. This means that if the first argument of a predicate is bound, Prolog does not need to scan for the correct rule to use, but can branch directly to the rule (or rules) that can apply.

We can illustrate this by writing an even faster version of union/3:

```
union([], S2, S2).
union([H1|T1], S2, Union):- union4(S2, H1, T1, Union).
```

Here, Prolog will branch directly to the first or second rule, depending on whether the first list is empty or not. If the second rule is chosen, it moves its second argument to the first argument position ( $s1 \square s2 = s2 \square s1$ ), so that table-based indexing can be used again:

```
union4([], H1, T1, [H1|T1]).
union4([H2|T2], H1, T1, Union) :-
    compare(Order, H1, H2),
    union6(Order, H1, T1, H2, T2, Union).
```

In the second rule of union4 we know that neither list is empty, so we compare their heads. The result of the comparison is passed as the first argument of the call of union6, so Prolog can again branch directly to the correct case.

```
union6(=, H, T1, _, T2, [H|Union]) :- union(T1, T2, Union).
union6(<, H1, T1, H2, T2, [H1|Union]) :- union4(T1, H2, T2, Union).
union6(>, H1, T1, H2, T2, [H2|Union]) :- union4(T2, H1, T1, Union).
```

A second important point is that the heads and tails of lists are passed here as separate arguments. Repackaging them in the form '[H|T]' would imply that Prolog should construct a new list cell. This implies storage allocation and consequent garbage collection, a major run-time overhead.

There are therefore *four* reasons why this version is fast:

- It always uses case analysis on the first argument.
- It doesn't split lists only to rebuild them—except once, in the first rule of union4.
- It makes fewer comparisons.
- All the recursive calls are deterministic and benefit from last call optimisation.

#### 4.10.2.3. Limitations

The set operations in sets.pro do not always function as we might like them to:

```
| ?- union( {}, {}, S), print(S).
{_, _16, _19}
| ?- intersection( {}, {}, S), print(S).
{}
```

We might expect both results to be `{_}`. However, our list-merging `union` and `intersection` predicates do not ask if elements can be *matched*, they compare them using standard ordering. All anonymous variables are different, so `compare/3` does not regard them as equal, even though they would unify. The earlier (less efficient) versions behave more like we would expect. Therefore, in using `sets.pro` we should make sure that the elements of sets are grounded.

#### 4.10.3. Representing Graphs, Relations and Functions

Prolog programmers usually represent graphs, relations and functions in similar ways. They all have three components: a **domain**, a **co-domain** (or **range**) and a **mapping** from the domain to the co-domain. The domain and co-domain are sets, and are typically represented by ordered lists. The mapping between them is a set of argument-value pairs, usually formed using the `'-'` operator. For example, the following structure might record some information about movies,

```
star(['Schwarzenegger', 'Stone', 'Williams'],
     ['Basic Instinct', 'Kindergarten Cop', 'Total Recall'],
     ['Schwarzenegger'-'Kindergarten Cop', 'Schwarzenegger'-'Total Recall',
      'Stone'-'Basic Instinct', 'Stone'-'Total Recall'])
```

If the mapping is homogeneous (the domain and co-domain are the same), it is possible to omit the co-domain. If the mapping happens to include all the domain and co-domain values, it is possible to omit the domain and co-domain sets altogether.

The following representation is more compact and efficient for many purposes,

```
['Schwarzenegger'-['Kindergarten Cop', 'Total Recall'],
 'Stone'-['Basic Instinct', 'Total Recall'],
 'Williams'-[]]
```

Library functions for these representations are readily available on the Internet. Part of the attraction of semantic net representations (which we discuss later) is that, because they reduce everything to functions and relations, we can make direct use of such libraries.

#### 4.10.4. Priority Queues

A **priority queue** is a data structure containing elements that have a priority. An operation is needed to take the first element from the queue, i.e., that with the earliest priority. Another operation is needed to add an element with arbitrary priority. Such structures are useful when we want to deal with sub-problems in what may prove to be an efficient order.

The `least/3` predicate lets us step through the elements of a set in standard order. This makes it possible to utilise a set as a priority queue. This is not very 'pure', but is certainly convenient. All we need to do is to make sure that each element has the form `functor(Priority, Value)`, e.g., `2-[left,walk]`. Provided we remember that the smallest value of `Priority` is at the head of the queue, `least` can be utilised as a `serve` predicate.

`union/3` may be utilised to add elements to a priority queue, serving as a `wait` predicate.

## 5. Logic

### 5.1. Propositional Calculus

The **Propositional Calculus** deals with logical propositions. Propositions can be either true or false, e.g., ‘I am happy’, or ‘I know who did it.’ We say that propositional calculus is a *two-valued logic*. It is simpler than **First-Order Predicate Calculus**, because it does not allow universal or existential variables, as in ‘Everyone is happy’, or ‘Someone knows who did it.’ It is easier to understand concepts such as *resolution* in connection with propositional calculus first, before trying to understand their use in first-order predicate calculus.

Propositional calculus allows expressions to be connected by the following operators:

Operator	Symbol	ASCII	$P$ $Q$	False False	False True	True False	True True
not	$\neg$	~	$\neg P$	True	True	False	False
and	$\wedge$	&	$P \wedge Q$	False	False	False	True
or	$\vee$	v	$P \vee Q$	False	True	True	True
implies	$\supset$	=>	$P \supset Q$	True	True	False	True
if	$\supset$	<=	$P \supset Q$	True	False	True	True
iff	$\equiv$	<=>	$P \equiv Q$	True	False	False	True

The precedence of operators is in order of the table, with  $\neg$  (not) binding tightest. A simple proposition, or a negated simple proposition, is called a **literal**.

The list of the values of an expression for each combination of values of its arguments is called its **truth table**. To test if two propositional calculus expressions are equal, it is only necessary to compare their truth tables. Propositional calculus is complete and consistent.

#### 5.1.1. Axioms

Negation:	$\neg \text{True} = \text{False}$ $\neg \text{False} = \text{True}$
Excluded Middle:	$P \vee \neg P = \text{True}$ $P \wedge \neg P = \text{False}$
And:	$P \wedge \text{True} = P$ $P \wedge \text{False} = \text{False}$
Or:	$P \vee \text{True} = \text{True}$ $P \vee \text{False} = P$
Implications:	$P \supset Q = \neg P \vee Q$ $P \supset Q = Q \supset P = P \supset \neg \neg Q$ $P \supset Q = P \supset Q \supset Q \supset P = P \supset Q \supset \neg P \supset \neg Q$

#### 5.1.2. Laws

Commutation:	$P \wedge Q = Q \wedge P$ $P \vee Q = Q \vee P$
Association:	$(P \wedge Q) \wedge R = P \wedge (Q \wedge R) = P \wedge Q \wedge R$ $(P \vee Q) \vee R = P \vee (Q \vee R) = P \vee Q \vee R$
Distribution:	$P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$ $P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$
Idempotence:	$P \wedge P = P$ $P \vee P = P$
Double Negation:	$\neg(\neg P) = P$
DeMorgan’s Laws:	$\neg(P \wedge Q) = \neg P \vee \neg Q$ $\neg(P \vee Q) = \neg P \wedge \neg Q$ $f(\neg, \vee, \text{True}, \text{False}, P, Q, \dots) = f(\neg, \wedge, \text{False}, \text{True}, \neg P, \neg Q, \dots)$

#### 5.1.3. Rules of Inference

In the usual notation for **inference rules**, if all the *antecedents* above the line are true, then the *consequent* below the line is true.<sup>32</sup>

<sup>32</sup> The expressions above the line are implicitly and-ed together.

<i>Modus Ponens:</i>	$\frac{P \quad P \sqsupset Q}{Q}$
<i>Modus Tollens:</i>	$\frac{P \sqsupset Q \quad \neg Q}{\neg P}$
Transitivity:	$\frac{P \sqsupset Q \quad Q \sqsupset R}{P \sqsupset R}$
And-elimination:	$\frac{P \sqcap Q}{P}$
And-introduction:	$\frac{P \quad Q}{P \sqcap Q}$
Or-introduction:	$\frac{P}{P \sqcup Q}$
Double-negation elimination:	$\frac{\neg(\neg P)}{P}$
Resolution:	$\frac{P \quad Q \quad \neg P \sqcup R}{Q \quad R}$

#### 5.1.4. Conjunctive Normal Form

Prolog can be used to write a program that takes any propositional calculus expression and reduces it to **Conjunctive Normal Form**, or **CNF**, as above. The advantage of *any* normal form is that it reduces the number of cases a program has to consider, so normalisation is a good first step.

A logic expression might state, ‘If it rains, I will get wet. If I get wet, I will catch a chill. But if it rains, I won’t catch a chill.’ These three statements are *mutually inconsistent*, something that a program can prove. Letting *R* represent ‘It rains’, *W* represent ‘I get wet’, and *C* represent ‘I catch a chill’, the formula could be written as,

$$(R \sqsupset W) \sqcap (W \sqsupset C) \sqcap \neg(R \sqsupset C)$$

In CNF, the same statements are expressed as

$$(\neg R \sqcup W) \sqcap (\neg W \sqcup C) \sqcap R \sqcup \neg C$$

CNF expressions are *products of sums of literals*.<sup>33</sup>

For convenience, in Prolog we shall use the notations,

$$(\text{rain} \Rightarrow \text{wet}) \ \& \ (\text{wet} \Rightarrow \text{chill}) \ \& \ \sim(\text{rain} \Rightarrow \text{chill})$$

and,

$$(\sim \text{rain} \vee \text{wet}) \ \& \ (\sim \text{wet} \vee \text{chill}) \ \& \ \text{rain} \ \& \ \sim \text{chill}$$

Here, we use ‘ $\Rightarrow$ ’ to denote ‘implies’, ‘ $\&$ ’ to denote ‘and’, ‘ $\vee$ ’ to denote ‘or’, and ‘ $\sim$ ’ to denote ‘not’. Other symbols that could be used are, ‘ $\Leftrightarrow$ ’ to denote ‘iff’ (if and only if), and ‘ $\Leftarrow$ ’ to denote ‘if’.

Declaring these logic symbols as Prolog operators makes the program much clearer, and makes it easy to analyse the input data.<sup>34</sup>

```
:- op(600, xfx, [<=>, =>, <=]).
:- op(500, xfy, v).
:- op(400, xfy, &).
:- op(300, fy, ~).
```

Normalisation first replaces the conditionals  $\Rightarrow$ ,  $\Leftrightarrow$ , and  $\Leftarrow$  by substituting equivalent expressions involving only  $\&$ ,  $\vee$  and  $\sim$ .

<sup>33</sup> It is also possible to derive a Disjunctive Normal Form, which has the form of a sum of products of literals. In the case of the example it is,

$$\neg R \sqcap \neg W \sqcap R \sqcap \neg C \quad \neg R \sqcap C \sqcap R \sqcap \neg C \quad W \sqcap \neg W \sqcap R \sqcap \neg C \quad W \sqcap C \sqcap R \sqcap \neg C.$$

<sup>34</sup> The `op` directives are written at the *start* of the program file, so that they are executed before Prolog has to recognise the new operators.

Logic

```
normalise((X => Y), ~X v Y).
normalise((X <= Y), X v ~Y).
normalise((X <=> Y), (~X v Y) & (X v ~Y)).
```

(The correctness of the substitutions can be checked from the truth tables of the operators.)

After the substitution process, the original formula looks like this:

```
(~rain v wet) & (~wet v chill) & ~(~rain v chill)
```

The next step is to manipulate the expression so that the negations are attached directly to variables. This step makes use of DeMorgan's laws:

```
normalise(~(X & Y), ~X v ~Y).
normalise(~(X v Y), ~X & ~Y).
normalise(~(~X), X).
```

After applying DeMorgan's laws, the original formula would look like this:

```
(~rain v wet) & (~wet v chill) & rain & ~chill
```

The next stage is to push the 'v' operators down below the '&' operators. In the case of the example formula, this step does nothing, because its input is already in the desired form. The basic idea is to use the distributive law for 'or'.

```
normalise(X v (Y & Z), (X v Y) & (X v Z)).
normalise((Y & Z) v X, (Y v X) & (Z v X)).
```

We also need to make sure that *all* the rules are applied to sub-expressions, leaving atomic propositions unchanged.

```
normalise(X1 & Y1, X2 & Y2) :- normalise(X1, X2), normalise(Y1, Y2).
normalise(X1 v Y1, X2 v Y2) :- normalise(X1, X2), normalise(Y1, Y2).
normalise(~X1, ~X2) :- normalise(X1, X2).
normalise(X, X) :- atom(X).
```

After each normalisation step, another may become possible, so the rules need to be applied iteratively until no further simplification is possible:

```
closure(X0, X) :- normalise(X0, X1), !, closure3(X0, X1, X).
closure3(X, X, X) :- !. % No further normalisation was possible.
closure3(_, X0, X) :- closure(X0, X). % More normalisation was done, so try again.
```

Here, the first rule applies a normalisation rule to the given expression, then calls `closure/3` with the original and new expressions as arguments. If the new expression is the same as the old one, no further normalisation was possible, so `closure/3` immediately succeeds. Otherwise, it calls `closure/2` to try further steps.

Once an expression has been reduced to a product of sums, the 'and' and 'or' operators become redundant. The expression can be reduced to a set of sets of literals.<sup>35</sup>

```
strip_and(X, S) :- strip_and3(X, {}, S).
strip_and3(X & Y, S0, S) :- !, strip_and3(X, S0, S1), strip_and3(Y, S1, S).
strip_and3(X, S0, S) :- strip_or3(X, {}, S1), union(S0, {S1}, S).
strip_or3(X v Y, S0, S) :- !, strip_or3(X, S0, S1), strip_or3(Y, S1, S).
strip_or3(X, S0, S) :- union({X}, S0, S).
```

Combining the above sub-goals, we derive the following predicate:

```
cnf(X0, S) :- closure(X0, X1), strip_and(X1, S).
```

which succeeds if  $P_0$  is a formula of the propositional calculus, and  $P$  is its CNF:

```
| ?- cnf((rain=>wet) & (wet=>chill) & ~(rain=>chill), CNF), print(CNF), nl.
{{chill, ~wet}, {rain}, {wet, ~rain}, {~chill}}
```

### 5.1.5. Clausal Form and Horn Clauses

A simple manipulation of a CNF expression changes it to another normal form: **clausal form**. We simply separate the positive and negative literals of each disjunct into two lists and put them either side of an '<=' operator. The example expression then becomes:

```
{{}<={chill}, {chill}<={wet}, {rain}<={}, {wet}<={rain}}
```

The predicate 'clausal\_form/2' converts a CNF expression to clausal form.

```
clausal_form(X, S) :- cnf(X, S1), findset(C, (all(X0, S1), regroup(X0, C)), S).
```

---

<sup>35</sup> Accumulating arguments are used, so that last call optimisation will be effective.

```
regroup(S, Pos<=Neg) :-
    findset(X, all(~X, S), Neg), findset(Y, (all(Y, S), Y\= ~_), Pos).
| ?- clausal_form((rain=>wet) & (wet=>chill) & ~(rain=>chill), CF), print(CF), nl.
{{}<={chill}, {chill}<={wet}, {rain}<={}, {wet}<={rain}}
```

(Compare this with CNF formula above.)

In this form, terms on the left are implicitly **or**'ed together, and terms on the right are implicitly **and**'ed together. The formula,

```
{chill, dry}<={rain, wet}.
```

would mean,

```
chill v dry <= rain & wet.
```

**Horn clauses** are a *subset* of clausal form, in which there can be *only one element on the left*. This is potentially a serious restriction. Prolog gets around it by re-introducing negation:

```
chill <= rain & wet & ~dry.
```

is expressed as,

```
chill :- rain, wet, \+dry.
```

Ignoring the problem of negation, Prolog can be said to implement Horn clause logic.

### 5.1.6. Resolution

There are several **rules of inference** used in logical proof. The most widely known are called *modus ponens*: (If *P* implies *Q* and *P* is true, then *Q* must be true).

$$\frac{P \supset Q \quad P}{Q}$$

and *modus tollens*: (If *P* implies *Q* and *Q* is false, then *P* must be false).

$$\frac{P \supset Q \quad \neg Q}{\neg P}$$

There are several other less well-known rules, some of them trivial, such as *and elimination*:

$$\frac{P \supset Q \quad P}{Q}$$

meaning that if *P* and *Q* are both true, then *P* is true.

All the inference rules are special cases of one general rule, called **resolution**,

$$\frac{P \supset Q \quad Q \supset R}{P \supset R}$$

*Q* and  $\neg Q$  are said to **cancel**, leaving *P*  $\supset$  *R*.<sup>36</sup> To derive *modus ponens*, for example, we first replace  $P \supset Q$  by  $\neg P \supset Q$ .  $\sim P$  then cancels *P*, leaving *Q*.

$$\frac{P \supset Q \quad \neg P \supset Q}{Q}$$

Resolution can be applied to CNF formulas just by finding two sums containing complementary literals, cancelling them, and finding the union of the remaining terms, called their **resolvent**.

The following Prolog predicate takes two sums, Sum1 and Sum2, both in set form, and attempts to resolve them.

```
resolve(Sum1, Sum2, Resolvent) :-
    all(X, Sum1), in(~X, Sum2),
    union(Sum1, Sum2, Sum3),
    difference(Sum3, {X, ~X}, Resolvent).
```

If it can succeed in selecting any term *x* from Sum1, and its complement is in Sum2, then combining the sums using *union*, and removing the cancelled terms, thus leaving the resolvent. When the *all* predicate is called, *x* is unbound. It will propose each literal in turn as a candidate for *x*:

<sup>36</sup> If *Q* is false, *P* must be true, if *Q* is true, *R* must be true, hence *P*  $\supset$  *R*.

We can use resolution to form *every* possible consequence of a CNF formula:

```
resolve(Known0, Known) :-
    all(S1, Known0), all(S2, Known0),
    resolve(S1, S2, Resolvent),
    \+in(Resolvent, Known0),
    union({Resolvent}, Known0, Known1),
    !, resolve(Known1, Known).
resolve(Known, Known).
```

The `resolve/2` predicate uses `all/2` to select every possible ordered pair of sums to present to `resolve/3`. If the resolvent isn't already in the list of known formulas, it is added to it, and `resolve/2` is then called iteratively.<sup>37</sup> If there are no new terms that can be added, then the 2nd rule of `resolve/2` returns the completed list. Here is the result:

```
| ?- resolve({{rain}, {wet, ~rain}, {chill, ~wet}, {~chill}}, R), print(R), nl.
{{}, {chill}, {chill, ~rain}, {wet}, {~rain}, {~wet}, {rain}, {~chill}, {chill, ~wet},
{wet, ~rain}}
```

**5.1.7. Proof by Refutation**

One of the terms in the above resolvent is the empty list. An empty sum means 'false'. It is a logical deduction from the fact that the terms of the original formula,

$$(rain \Rightarrow wet) \ \& \ (wet \Rightarrow chill) \ \& \ \sim(rain \Rightarrow chill)$$

are inconsistent. (Further evidence of this is that the resolvent contains both `{chill}` and `{~chill}`.) This illustrates **proof by refutation**.

Suppose we want to prove the following well-known rule of inference:

$$\frac{P \ \& \ Q \quad Q \ \& \ R}{P \ \& \ R}$$

We set up the conjunction of the antecedents with the *negation* of the consequent, and try to prove a **contradiction**.

$$\frac{P \ \& \ Q \quad Q \ \& \ R \quad \sim(P \ \& \ R)}{?}$$

Why do things in this roundabout way? If we had started with the antecedents, we could generate all their consequents, among which might be the one desired.<sup>38</sup> This would work well enough for propositional calculus, but when we move on to the **First-Order Predicate Calculus**, which allows variables, there can be an infinite number of consequents, and the required one may never be generated. However, since a refutation is discovered by generating an empty clause, we can exploit two heuristics:

- Resolve the shortest formulas first.
- Make use of the negated consequent.

First we rewrite the antecedents using  $\square$ , and  $\sim$ :

$$\frac{\square P \quad Q \quad \square Q \quad \sim R}{\square(\square P \ \& \ R)}$$

Next we use DeMorgan's rule to express the third antecedent in CNF:

$$\frac{\square P \quad Q \quad \square Q \quad \sim R \quad P}{\square R}$$

We can use resolution on the 1st and 3rd terms:

$$\frac{\square P \quad Q \quad \square Q \quad \sim R \quad P \quad \square R}{Q}$$

And again on the 2nd and 4th:

<sup>37</sup> The harmless cut in 'resolve/2' makes sure Prolog will use last call optimisation.

<sup>38</sup> It is present in the above resolvent in the form '[chill, ~rain]'.

$$\begin{array}{c}
 \neg P \quad Q \\
 \neg Q \quad R \\
 \quad P \\
 \quad \neg R \\
 \quad \quad Q \\
 \quad \quad \neg Q \\
 \hline
 \quad \quad ?
 \end{array}$$

Resolving the final two terms yields an empty resolvent, i.e., ‘false’.

### 5.1.8. Horn Clause Resolution

When a formula is in **Horn clause form**, there is only *one* negated term in each clause, the one on its left-hand side. Therefore, we can take *any* right-hand side term, look for it as a left-hand side of a Horn clause, then replace the term by the right-hand side of the clause. For example, in,

$$\begin{array}{l}
 P \wedge Q \wedge R \\
 Q \wedge S \wedge T \\
 R \wedge U \wedge V
 \end{array}$$

we may substitute for both *Q* and *R* in the first clause, to give,

$$P \wedge S \wedge T \wedge U \wedge V$$

This is exactly how Prolog proves sub-goals. Prolog’s approach is called **SLD-resolution**, Selected Linear resolution for Definite clauses. It is depth-first left-right substitution. SLD-resolution may sometimes fail to prove a valid consequent because it gets caught in an infinite recursion.

## 5.2. First-Order Predicate Calculus

### 5.2.1. Quantifiers

First-order predicate calculus is similar to propositional calculus, but it also uses variables and quantifiers. We use an ‘A’ upside-down, to mean ‘for all’, and an ‘E’ backwards to mean ‘there exists’. For example, to represent the usual meaning of the sentence, ‘Everybody loves somebody.’ we would write,

$$\forall x(\exists y(\text{Loves}(x, y)))$$

unless we thought it really meant, ‘There is somebody that everybody loves.’ which is,

$$\exists y(\forall x(\text{Loves}(x, y)))$$

Variables preceded by  $\forall$  are said to be **universally quantified**. Variables preceded by  $\exists$  are said to be **existentially quantified**.

In practice, we often need to define the *types* of quantified variables. For example,

$$\forall x(\exists y(\text{Loves}(x, y)))$$

really means, ‘Everything loves something.’ We need to say that *x* and *y* are persons:

$$\forall x(\text{Person}(x) \wedge (\exists y(\text{Person}(y) \wedge \text{Loves}(x, y))))$$

That is, ‘For every *x* that is a person, there is some *y* such that *y* is a person and *x* loves *y*.’<sup>39</sup>

In a sense, quantification is just shorthand. If there are just 3 persons in the universe of discourse, Alex, Chris and Pat, then we could write,

$$\begin{array}{l}
 (\text{Loves}(\text{Alex}, \text{Alex}) \quad \text{Loves}(\text{Alex}, \text{Chris}) \quad \text{Loves}(\text{Alex}, \text{Pat})) \\
 \wedge (\text{Loves}(\text{Chris}, \text{Alex}) \quad \text{Loves}(\text{Chris}, \text{Chris}) \quad \text{Loves}(\text{Chris}, \text{Pat})) \\
 \wedge (\text{Loves}(\text{Pat}, \text{Alex}) \quad \text{Loves}(\text{Pat}, \text{Chris}) \quad \text{Loves}(\text{Pat}, \text{Pat}))^{40}
 \end{array}$$

In other words,  $\forall$  functions like  $\forall$ , and  $\exists$  functions like  $\exists$ . As a result, the following versions of DeMorgan’s Law are sometimes useful.

$$\begin{array}{l}
 \neg(\forall x(P(x))) \equiv \exists x(\neg P(x)) \\
 \neg(\exists x(P(x))) \equiv \forall x(\neg P(x))
 \end{array}$$

Suppose we have the rule,

$$\forall x(\text{Feathered}(x) \supset \text{Bird}(x))$$

and the antecedent,

$$\text{Feathered}(\text{Twety})$$

then we can apply the rule to deduce

<sup>39</sup> Notice how  $\forall$  is associated with  $\exists$ , and  $\exists$  is associated with  $\forall$ .

<sup>40</sup> This would get a bit tedious when dealing with infinite sets.

Bird(Tweety)

provided we **unify** ‘Tweety’ with ‘x’.

### 5.2.2. Unification

It is a bit silly trying to write a Prolog predicate for unification, because we have to rely on ‘=’ (matching) to finish the job. However, the following illustrates some important details of how Prolog does it:

```
unify(X, Y) :- unify2(X, Y).
unify(X, Y) :- unify2(Y, X).
```

(The order of the arguments doesn’t matter.)

```
unify2(X, Y) :- var(X), !, X=Y.
unify2(X, Y) :- atom(X), atom(Y), !, X==Y.
unify2(X, Y) :- structure(X), structure(Y), !,
    X=..[FX|YArgs], Y=..[FY|YArgs],
    atom(FX), atom(FY), FX==FY,
    unify(XArgs, YArgs), X=Y.
```

First, a *variable* unifies with *anything*. Second, two *atoms* unify only if they are already identical. Third, two *structures* unify only if they have identical functors and their argument lists unify.<sup>41</sup> The functors must be atoms—otherwise we would be discussing not *first-order*, but *higher-order* predicate calculus.<sup>42</sup>

The Prolog definition is *almost* correct for first-order predicate calculus, but unfortunately it lets  $X$  unify with  $f(X)$ . It is not valid for  $X$  to unify with a term containing itself. Implementing such an **occurs-check** is costly. Prolog doesn’t do it unless asked.

### 5.2.3. Eliminating Quantifiers

We may reduce any first-order predicate calculus expression to clausal form, just as we did for propositional calculus, except that this time any variables that appear are *assumed* to be universally quantified. That is to say,

$$\text{Bird}(x) \sqcap \text{Feathered}(x)$$

is assumed to be true for all  $x$ .

How do we get rid of the *existential* quantifiers? Consider ‘Everybody loves somebody.’ Given any particular person, at least one person they love is known. Let’s say one person loved by ‘ $x$ ’ is called ‘Sweetheart( $x$ )’. Then we have,

$$\sqcap x(\text{Loves}(x, \text{Sweetheart}(x)))$$

and the existentially quantified variable has disappeared. ‘Sweetheart’ is an example of a **Skolem function**. By using Skolem functions and leaving the universal quantifiers implicit, we can reduce *any* first-order predicate calculus expression to clausal form.<sup>43</sup>

A similar convention is used in Prolog: variables that appear in the head of a rule are universally quantified; variables that appear only in the body of a rule are existentially quantified. In effect, sub-goals that bind existential variables operate as Skolem functions.

<sup>41</sup> Prolog’s =.. (univ) operator converts a structure to a list, or *vice versa*.

<sup>42</sup> For example, the versions of DeMorgan’s Laws we have just given are higher-order; they are true for *any* predicate  $P$ .

<sup>43</sup> See Clocksin & Mellish, *Programming in Prolog*, Springer 1981, 1994, for a complete algorithm.

## 6. Expert Systems

### 6.1. Introduction

**Expert systems** are programs that use logical inference to reason about real-world situations by employing rules obtained from an expert—which does *not* mean that they always perform as well as an expert does. Expert systems are useful when,

- The program deals with a closed situation,
- A real expert is expensive to hire.

Many expert systems are written using an **expert-system shell**, a generalised program that takes care of interfacing with the user, and leaves the programmer only to discover the **domain-specific knowledge** from an expert. The shell contains an **inference engine** or **logic engine** that can use the domain-specific knowledge to make deductions. An expert system can usually explain **how** it reached a conclusion, or **why** it wants to know something.

In the following example, the domain-specific knowledge consists of logic rules. Each rule comprises a rule identifier, a list of **antecedents**, and a **consequent**. The format of a rule is,

```
Description:{Antecedent, ...}=>Consequent.
```

Apart from the description, which is intended to make the rules more readable, they have the form of Horn clauses: the consequent is true if all the antecedents are true.

We can incorporate such rules into a Prolog source file provided we begin it is as follows,

```
:- op(500, xfx, =>).
:- op(400, xfx, :).
```

These declarations make => and : become non-associative infix operators, with : binding more tightly than =>.

Here is some expert knowledge about the classification of animals, obtained from a zoo-keeper. The rules are in a particularly simple form: negation is not allowed.

```
'Only mammals are hairy':
  {hairy(X)} => mammal(X).
'Only mammals give milk':
  {gives_milk(X)} => mammal(X).
'Only birds have feathers':
  {has_feathers(X)} => bird(X).
'Only birds fly and lay eggs':
  {flies(X), lays_eggs(X)} => bird(X).
'Carnivores are mammals that eat meat':
  {mammal(X), eats_meat(X)} => carnivore(X).
'Carnivores are mammals designed to kill':
  {mammal(X), pointed_teeth(X), has_claws(X), forward_eyes(X)} => carnivore(X).
'All hoofed mammals are ungulates':
  {mammal(X), hoofed(X)} => ungulate(X).
'All mammals that chew the cud are ungulates':
  {mammal(X), chews_cud(X)} => ungulate(X).
'The only carnivores with spotted tawny coats are cheetahs':
  {carnivore(X), tawny(X), spots(X)} => cheetah(X).
'The only carnivores with striped tawny coats are tigers':
  {carnivore(X), tawny(X), striped(X)} => tiger(X).
'The spotted tawny ungulates with long legs and necks are giraffes':
  {ungulate(X), long_legs(X), long_neck(X), tawny(X), spots(X)} => giraffe(X).
'The ungulates with striped white coats are zebras':
  {ungulate(X), white(X), striped(X)} => zebra(X).
'The flightless black & white birds with long legs & necks are ostriches':
  {bird(X), long_legs(X), long_neck(X), black_and_white(X), not_flies(X)}
=> ostrich(X).
'The flightless black and white birds that swim are penguins':
  {bird(X), swims(X), black_and_white(X), not_flies(X)} => penguin(X).
'The only birds that fly are albatrosses':
  {bird(X), flies(X)} => albatross(X).
```

Although there is a predicate `not_flies(X)` in these rules, it doesn't mean the inference engine will need to handle negation; 'not' is simply part of the name, it isn't an operator.

The second thing we need is some facts about the specific situation. Here are some observations made about Stretch and Swifty by a visitor to the zoo.

```
'Stretch is hairy':{} => hairy('Stretch').
'Stretch chews cud':{} => chews_cud('Stretch').
'Stretch has long legs':{} => long_legs('Stretch').
'Stretch has a long neck':{} => long_neck('Stretch').
'Stretch has a tawny coloured coat':{} => tawny('Stretch').
'Stretch has a spotted coat':{} => spots('Stretch').
'Swifty is hairy':{} => hairy('Swifty').
'Swifty has pointed teeth':{} => pointed_teeth('Swifty').
'Swifty has claws':{} => has_claws('Swifty').
'Swifty has forward looking eyes':{} => forward_eyes('Swifty').
'Swifty has a tawny coat':{} => tawny('Swifty').
'Swifty has a spotted coat':{} => spots('Swifty').
```

For uniformity with the rules, we express facts in the same format, but with an empty list of antecedents.

## 6.2. Forward Chaining

In **forward chaining**, the logic engine starts from what is known, and makes new inferences. That is to say, it works from antecedents to consequents.

The predicate `forward/2` takes a list of facts, `Proved0`, and attempts to add new proofs to it to give `Proved`. It does this by searching for a rule whose antecedents are all proved, but whose consequent is not yet proved.

Clearly, the facts are trivial to prove.

```
forward :- forward({}, Proved), print(Proved), nl.

forward(Proved0, Proved) :-
    _:Antecedents=>Consequent,
    subset(Antecedents, Proved0),
    \+in(Consequent, Proved0),
    union({Consequent}, Proved0, Proved1),
    !, forward(Proved1, Proved).
forward(Proved, Proved).
```

This predicate is not as simple as it may seem:

First, we must be careful to test the antecedents *before* the consequent. This is consistent with the usual advice about deferring negation in Prolog. To see how it can cause trouble here, imagine that `mammal('Swifty')` has already been proved, but `mammal('Stretch')` has not. In this situation, `forward` couldn't prove `mammal('Stretch')`, because any rule with `mammal(X)` as a consequent would match `mammal('Swifty')`. Therefore `forward` would decide that `mammal(X)` has already been proved. However, by testing the antecedents first, `X` will become bound to `'Stretch'`, and `mammal('Stretch')` will *not* match an existing element of `Proved`.

Note too, that the rules in the knowledge base are stored as *Prolog facts* (with functor `=>`), e.g., `'=>'('Only mammals are hairy':{hairy(X)}, mammal(X))`. Matching such a fact is no different from matching `male('Charles')`.

With the above definitions, `forward` gives the following results,

```
| ?- forward({}, Proved), print(Proved).
{carnivore(Swifty), cheetah(Swifty), chews_cud(Stretch),
 forward_eyes(Swifty), giraffe(Stretch), hairy(Stretch),
 hairy(Swifty), has_claws(Swifty), long_legs(Stretch),
 long_neck(Stretch), mammal(Stretch), mammal(Swifty),
 pointed_teeth(Swifty), spots(Stretch), spots(Swifty), tawny(Stretch),
 tawny(Swifty), ungulate(Stretch)}
```

You will see from this that forward chaining proves every fact that *can* be proved, irrespective of whether or not it is useful.

### 6.2.1. How Explanations

To improve on the basic idea, we need to tell `forward` which facts we are interested in, and we would also like to see *how* `forward` has proved them. After all, we are not likely to trust an expert system unless it can convince us that it has reasoned correctly.

First, we define some hypotheses about what animals might be present:

```
(X is 'a cheetah'):cheetah(X).
(X is 'a tiger'):tiger(X).
(X is 'a giraffe'):giraffe(X).
(X is 'a zebra'):zebra(X).
(X is 'an ostrich'):ostrich(X).
(X is 'a penguin'):penguin(X).
(X is 'an albatross'):albatross(X).
```

(Prolog defines `is` as an operator, so this format is syntactically acceptable—provided we don't try to evaluate the right-hand side of `is` as an arithmetic expression.)

In order to show how a hypothesis has been proved, we need to define the structure of a proof. We use the following recursive structure,

```
Description:{SubProof, ...}>=>Conclusion.
```

where each `SubProof` has the same form as a proof, and corresponds to the proof of an antecedent. When `Conclusion` is a basic fact, it has no antecedents, and no sub-proofs:

The resulting output will consist of two structured proofs, as follows,

```
The hypothesis 'Swiftly is a cheetah' was proved as follows:
  The only carnivores with spotted tawny coats are cheetahs
    Carnivores are mammals designed to kill
      Only mammals are hairy
        Swifty is hairy
          Swifty has claws
            Swifty has forward looking eyes
              Swifty has pointed teeth
                Swifty has a spotted coat
                  Swifty has a tawny coat
```

The program was able to prove that Swifty is a cheetah because a cheetah is a carnivore with a spotted tawny coat. A carnivore is a mammal with pointed teeth, claws, and forward pointing eyes. Swifty is a mammal because it has a hairy coat.

```
The hypothesis 'Stretch is a giraffe' was proved as follows:
  The spotted tawny ungulates with long legs and necks are giraffes
    All mammals that chew the cud are ungulates
      Only mammals are hairy
        Stretch is hairy
          Stretch chews cud
            Stretch has a long neck
              Stretch has a spotted coat
                Stretch has a tawny coloured coat
                  Stretch has long legs
```

Stretch is a giraffe because Stretch is an ungulate with long legs and neck and a spotted tawny coat. It proved that Stretch is an ungulate because Stretch is a mammal that chews cud. It proved Stretch is a mammal because Stretch is known to be hairy.

It would not be hard to transform each proof into a plain language explanation. It is called a **how-explanation**, because it explains *how* a conclusion was reached.

```
forward_with_how :- forward_with_how({}).
forward_with_how(Proofs0) :-
  Description:Antecedents=>Consequent,
  satisfied(Antecedents, Proofs0, SubProofs),
  \+in(_:>=>Consequent, Proofs0),
  Proof=Description:SubProofs=>Consequent,
  ( Hypothesis:Consequent -> explain_how(Hypothesis, Proof)
  ; true),
  union({Proof}, Proofs0, Proofs1),
  !, forward_with_how(Proofs1).
forward_with_how(_).
```

```
satisfied(Antecedents, _, {}) :- empty(Antecedents).
satisfied(Antecedents0, Proofs, SubProofs) :-
    least(Antecedents0, Antecedent, Antecedents1),
    SubProof=(_: =>Antecedent),
    all(SubProof, Proofs),
    satisfied(Antecedents1, Proofs, SubProofs1),
    union({SubProof}, SubProofs1, SubProofs).
```

The basic approach is the same as before, but a simple subset test will no longer do, because it is necessary to construct the proofs of the antecedents. The `satisfied` predicate performs essentially the same task as `subset`, but also adds each sub-proof to a list. There are two important nuances:

- It is necessary to write `all(SubProof, Proofs)`, rather than `in(SubProof, Proofs)`. There is often more than one way to bind `SubProof`.
- It is tempting to use `findset` to find the set of sub-proofs, rather than build them up in a loop using `least` and `empty`. Unfortunately, when not all antecedents can be proved, `findset` would return the set of antecedents that can be proved, even if it is incomplete, and `satisfied` would succeed anyway.

When Consequent matches one of the hypotheses, its proof is displayed by `explain_how`:

```
explain_how(Hypothesis, Proof) :-
    print('The hypothesis '''),
    print(Hypothesis), write('' was proved as follows:'), nl,
    display_proof(4, Proof), nl.

display_proof(Tab, Description:SubProofs=>_) :-
    tab(Tab), print(Description), nl,
    Tab1 is Tab+4,
    all(SubProof, SubProofs),
    display_proof(Tab1, SubProof),
    fail.

display_proof(_, _).
```

### 6.3. Backward Chaining

In backward chaining, we start from the hypotheses, and use the rules and known facts to try to substantiate them. That is to say, we work from consequents to antecedents.

Backward chaining is what Prolog does anyway, but by going to a bit of trouble, we can make it explain not only *how* it did it, but also construct **why-explanations**.

The predicate `backward` chooses a hypothesis, and tries to prove it. If it succeeds, it displays the hypothesis it has proved, and backtracks:

```
backward :-
    Description:Hypothesis,
    prove_all({Hypothesis}),
    print(Description), nl,
    fail.

backward.
```

To prove a list of consequents, `prove_all` calls itself recursively to prove the antecedents of the first consequent, then calls itself iteratively to prove the rest of them:

```
prove_all(Consequents) :- empty(Consequents).
prove_all(Consequents0) :-
    least(Consequents0, Consequent, Consequents1),
    _:Antecedents=>Consequent,
    prove_all(Antecedents),
    prove_all(Consequents1).
```

The resulting output is not particularly informative:

```
| ?- backward.
Swifty is a cheetah
Stretch is a giraffe
```

Constructing a proof, or how-explanation, is similar to the method used in forward chaining:

```
backward_with_how :-
    Description:Hypothesis,
    Description1:Antecedents=>Hypothesis,
    prove_all(Antecedents, SubProofs),
    Proof=Description1:SubProofs=>Hypothesis,
    explain_how(Hypothesis, Proof),
    fail.
backward_with_how.

prove_all(Consequents, Consequents) :- empty(Consequents).
prove_all(Consequents0, Proofs) :-
    least(Consequents0, Consequent, Consequents1),
    Description:Antecedents=>Consequent,
    prove_all(Antecedents, SubProofs),
    Proof=Description:SubProofs=>Consequent,
    prove_all(Consequents1, Proofs1),
    union({Proof}, Proofs1, Proofs).
```

| ?- **backward\_with\_how.**

```
The hypothesis 'cheetah(Swifty)' was proved as follows:
The only carnivores with spotted tawny coats are cheetahs
    Carnivores are mammals designed to kill
        Only mammals are hairy
            Swifty is hairy
            Swifty has claws
            Swifty has forward looking eyes
            Swifty has pointed teeth
        Swifty has a spotted coat
    Swifty has a tawny coat
```

```
The hypothesis 'giraffe(Stretch)' was proved as follows:
The spotted tawny ungulates with long legs and necks are giraffes
    All mammals that chew the cud are ungulates
        Only mammals are hairy
            Stretch is hairy
            Stretch chews cud
        Stretch has a long neck
        Stretch has a spotted coat
        Stretch has a tawny coloured coat
    Stretch has long legs
```

### 6.3.1. Why Explanations

In many cases, backward chaining expert systems are interactive, and elicit facts from users as they need them. Rather than store observations such as ‘Stretch chews cud’ as facts, the system might ask the user, “Is ‘Stretch chews cud’ true?” Answering “yes” would assert the fact. However, users are not always happy to answer questions without knowing why they are being asked. It is reasonable for the user to say, “Why do you want to know that?” It might affect their answer. Since a backward chaining system would only ask such a question in the course of proving a hypothesis, it is quite capable of explaining its reasons. This is called a **why-explanation**. In contrast, a forward chaining system doesn’t know what it is trying to prove, so it *cannot* provide why-explanations.

In the following illustration, the facts are still represented as before, so a user is not actually being consulted. However, each time the program reaches a point where it *would* ask the user, it displays a why-explanation. To keep the example as simple as possible, its how-explanations have been omitted:

```
backward_with_why :-
    Description:Hypothesis,
    prove_with_why({Hypothesis}, [Description]),
    print(Description), nl, nl,
    fail.
backward_with_why.
```

`backward_with_why` begins by choosing a hypothesis, then trying to prove it by calling `prove_with_why`. The second argument of `prove_with_why` is the why-explanation, which in this case may be paraphrased as, “Because I am trying to prove the hypothesis.”

Unless the list is empty, `prove_with_why` checks the head of the list of things to be proved. It deals with three cases: The first is when the first consequent can be proved via a logic rule. It calls itself recursively to prove the

consequent, extending the why-explanation, then calls itself iteratively to prove the rest of the list. The second case is when the consequent is a *fact*; we assume that this is a point where the inference engine would ask the user if the fact is true, so it displays the why-explanation. The third case is where, since there is no logic rule that can be used, the user would again be asked if the consequent is a fact, but this time the answer is “No.” As before, the program displays a why-explanation. Note the avoidance of cut (!), to maximise the possibilities for back-tracking:

```

prove_with_why(Consequents, _) :- empty(Consequents).
prove_with_why(Consequents0, Why) :-
    least(Consequents0, Consequent, Consequents1),
    Description:Antecedents=>Consequent,
    \+empty(Antecedents),
    prove_with_why(Antecedents, [Description|Why]),
    prove_with_why(Consequents1, Why).
prove_with_why(Consequents0, Why) :-
    least(Consequents0, Consequent, Consequents1),
    Description:{}=>Consequent1,
    \+Consequent1\=Consequent, % avoid binding!
    copy_term([Consequent|Why], [Consequent2|Why2]),
    ask_nicely(Consequent2, Why2),
    Consequent1=Consequent,
    print('True! '), print(Description), nl, nl,
    prove_with_why(Consequents1, Why).
prove_with_why(Consequents0, Why) :-
    least(Consequents0, Consequent, _),
    \+(_:=>Consequent),
    ask_nicely(Consequent, Why),
    print('Unknown.'), nl, nl,
    fail.

ask_nicely(Consequent, Why) :-
    numbervars(Consequent, 23, _),
    print('Is '), print(Consequent),
    print(' true or false? I am trying to prove'), nl,
    explain_why(Why, 4).

explain_why([], _).
explain_why([Description|Why], Tab) :-
    tab(Tab), print(Description), nl,
    Tab1 is Tab+4,
    explain_why(Why, Tab1).

```

There are some irrelevant technicalities above, involving `copy_term` and `numbervars`, which allow unbound variables to be displayed as `X`, `Y`, etc., without actually binding them. The start of the output is as follows (note how `X` soon becomes bound),

```

| ?- backward_with_why.
Is eats_meat(X) true or false? I am trying to prove
    Carnivores are mammals that eat meat
        The only carnivores with spotted tawny coats are cheetahs
            X is a cheetah
Unknown.

Is forward_eyes(X) true or false? I am trying to prove
    Carnivores are mammals designed to kill
        The only carnivores with spotted tawny coats are cheetahs
            X is a cheetah
True! Swifty has forward looking eyes

Is has_claws(Swifty) true or false? I am trying to prove
    Carnivores are mammals designed to kill
        The only carnivores with spotted tawny coats are cheetahs
            Swifty is a cheetah
True! Swifty has claws

```

Is hairy(Swifty) true or false? I am trying to prove  
 Only mammals are hairy  
   Carnivores are mammals designed to kill  
     The only carnivores with spotted tawny coats are cheetahs  
       Swifty is a cheetah

True! Swifty is hairy

Is pointed\_teeth(Swifty) true or false? I am trying to prove  
 Carnivores are mammals designed to kill  
   The only carnivores with spotted tawny coats are cheetahs  
     Swifty is a cheetah

True! Swifty has pointed teeth

Is spots(Swifty) true or false? I am trying to prove  
 The only carnivores with spotted tawny coats are cheetahs  
   Swifty is a cheetah

True! Swifty has a spotted coat

Is tawny(Swifty) true or false? I am trying to prove  
 The only carnivores with spotted tawny coats are cheetahs  
   Swifty is a cheetah

True! Swifty has a tawny coat

Swifty is a cheetah

Eventually, the program discovers that Stretch is a giraffe:

Is long\_legs(X) true or false? I am trying to prove  
 The spotted tawny ungulates with long legs and necks are giraffes  
   X is a giraffe

True! Stretch has long legs

Is long\_neck(Stretch) true or false? I am trying to prove  
 The spotted tawny ungulates with long legs and necks are giraffes  
   Stretch is a giraffe

True! Stretch has a long neck

Is spots(Stretch) true or false? I am trying to prove  
 The spotted tawny ungulates with long legs and necks are giraffes  
   Stretch is a giraffe

True! Stretch has a spotted coat

Is tawny(Stretch) true or false? I am trying to prove  
 The spotted tawny ungulates with long legs and necks are giraffes  
   Stretch is a giraffe

True! Stretch has a tawny coloured coat

Is hoofed(Stretch) true or false? I am trying to prove  
 All hoofed mammals are ungulates  
   The spotted tawny ungulates with long legs and necks are giraffes  
     Stretch is a giraffe

Unknown.

Is chews\_cud(Stretch) true or false? I am trying to prove  
 All mammals that chew the cud are ungulates  
   The spotted tawny ungulates with long legs and necks are giraffes  
     Stretch is a giraffe

True! Stretch chews cud

Is hairy(Stretch) true or false? I am trying to prove  
 Only mammals are hairy  
   All mammals that chew the cud are ungulates  
     The spotted tawny ungulates with long legs and necks are giraffes  
       Stretch is a giraffe

True! Stretch is hairy

Stretch is a giraffe

But it doesn't stop there; it has more hypotheses to test:

## Expert Systems

```
Is gives_milk(Stretch) true or false? I am trying to prove
  Only mammals give milk
    All mammals that chew the cud are ungulates
      The spotted tawny ungulates with long legs and necks are giraffes
        Stretch is a giraffe
```

Unknown.

```
Is striped(X) true or false? I am trying to prove
  The ungulates with striped white coats are zebras
    X is a zebra
```

Unknown.

```
Is has_feathers(X) true or false? I am trying to prove
  Only birds have feathers
    The flightless black & white birds with long legs & necks are ostriches
      X is an ostrich
```

Unknown.

```
Is flies(X) true or false? I am trying to prove
  Only birds fly and lay eggs
    The flightless black & white birds with long legs & necks are ostriches
      X is an ostrich
```

Unknown.

```
Is has_feathers(X) true or false? I am trying to prove
  Only birds have feathers
    The flightless black and white birds that swim are penguins
      X is a penguin
```

And so on... Apart from trying to prove the same conclusion in every possible way, the main problem we see here is that the same fact (e.g., ‘Stretch is hairy’) can be asked for several times, which would be very annoying to the user. Also, some explanations involve unbound variables. This need not be annoying, as an unbound variable is equivalent to “any animal”, e.g., `gives_milk(X)` could be displayed as “Does any animal give milk?”

We can avoid annoying the user by remembering which questions have been asked, and whether the answers were true or false. The easiest way to modify the existing shell is to remember facts in the usual Prolog way—as *predicates*. We use `abolish` at the start of backward chaining to make sure there are no such predicates left from a previous execution:

```
backward_with_memory :-
  abolish(true/1), abolish(false/1),
  Description:Hypothesis,
  prove_with_memory({Hypothesis}, [Description]),
  print(Description), nl, nl,
  fail.
backward_with_memory.
```

We test if a fact has already been elicited before displaying a why-explanation. The `clause` predicate allows us to do this. It succeeds if its arguments match the head and body of an existing rule. A Prolog fact is assumed to have the body `true`:

```
prove_with_memory(Consequents, _) :- empty(Consequents).
prove_with_memory(Consequents0, Why) :-
  least(Consequents0, Consequent, Consequents1),
  clause(true(Consequent), true),
  prove_with_memory(Consequents1, Why).
prove_with_memory(Consequents0, _) :-
  least(Consequents0, Consequent, _),
  clause(false(Consequent), true),
  fail.
```

When we ask the user a new question, we add a clause to Prolog’s database using `assertz`:

```
prove_with_memory(Consequents0, Why) :-
  least(Consequents0, Consequent, Consequents1),
  Description:Antecedents=>Consequent,
  \+empty(Antecedents),
  prove_with_memory(Antecedents, [Description|Why]),
  prove_with_memory(Consequents1, Why).
```

```

prove_with_memory(Consequents0, Why) :-
    least(Consequents0, Consequent, Consequents1),
    \+clause(true(Consequent), _),
    \+clause(false(Consequent), _),
    Description: {}=>Consequent1,
    \+Consequent1\=Consequent, % avoid binding!
    copy_term([Consequent|Why], [Consequent2|Why2]), % clone with new variables
    ask_nicely(Consequent2, Why2),
    Consequent1=Consequent,
    print('True! '), print(Description), nl, nl,
    assertz(true(Consequent)),
    prove_with_memory(Consequents1, Why).
prove_with_memory(Consequents0, Why) :-
    least(Consequents0, Consequent, _),
    \+(_:_=>Consequent),
    \+clause(true(Consequent), true),
    \+clause(false(Consequent), true),
    ask_nicely(Consequent, Why),
    print('Unknown.'), nl, nl,
    assertz(false(Consequent)),
    fail.

```

The output becomes much shorter than before; 19 questions are asked rather than the original 30. Eleven duplicates are omitted.

The technique of creating Prolog clauses dynamically using `assertz` is *not* to be recommended. It was used here to minimise the changes that had to be made to the basic backward-chaining algorithm. Since clauses created by `assertz` aren't retracted on back-tracking, this can cause problems, although in this example, these side-effects are exactly what are needed. Debugging can be difficult too, requiring the frequent use of `listing`:

```

| ?- listing(true/1).
true(forward_eyes('Swifty')).
true(has_claws('Swifty')).
true(hairy('Swifty')).
true(pointed_teeth('Swifty')).
true(spots('Swifty')).
true(tawny('Swifty')).
true(long_legs('Stretch')).
true(long_neck('Stretch')).
true(spots('Stretch')).
true(tawny('Stretch')).
true(chews_cud('Stretch')).
true(hairy('Stretch')).

```

Asserting facts dynamically cannot be recommended. Such programs are hard to debug. A more idiomatic programming technique would have been to store the assertions as a set.

#### 6.4. Comparison of Forward and Backward Chaining

Whether we should use forward or backward chaining depends on the application. Forward chaining makes most sense when we have a fixed set of facts and we want to see what we can deduce from them, especially if we have many possible hypotheses, and only a few are likely to be true. Backward chaining makes more sense when we have only a few specific hypotheses but there are many facts that might be relevant, and perhaps the facts are costly to obtain.

Both methods can give how-explanations that show how their conclusions were reached, but why-explanations are applicable only to backward-chaining systems where facts are elicited interactively.

In practice, reasoning often uses a mixture of forward and backward chaining. An observation may suggest a couple of hypotheses, which can then be tested by backward chaining. For example, a patient visiting a doctor may complain of certain symptoms. These suggest possible causes to the doctor (forward chaining), who will then make certain tests (backward chaining). The results of these tests may then suggest a particular disease and remedy (forward chaining).

Finally, although we have not done so here, we can see that negation needs to be treated with care. Just because something is not known does not prove that it is false. We ought therefore to use a 3-valued logic: **true**, **false**, and **unknown**. We would then use the following truth table:

<i>P</i>	<i>Q</i>	<i>P</i> $\square$ <i>Q</i>	<i>P</i> $\square$ <i>Q</i>	$\neg$ <i>P</i>
<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>
<i>false</i>	<i>unknown</i>	<i>false</i>	<i>unknown</i>	<i>true</i>
<i>false</i>	<i>true</i>	<i>false</i>	<i>true</i>	<i>true</i>
<i>unknown</i>	<i>false</i>	<i>false</i>	<i>unknown</i>	<i>unknown</i>
<i>unknown</i>	<i>unknown</i>	<i>unknown</i>	<i>unknown</i>	<i>unknown</i>
<i>unknown</i>	<i>true</i>	<i>unknown</i>	<i>true</i>	<i>unknown</i>
<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>
<i>true</i>	<i>unknown</i>	<i>unknown</i>	<i>true</i>	<i>false</i>
<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>false</i>

An intelligent logic engine will use laws such as  $P \square Q \square Q \square P$ , and  $P \square Q \square Q \square P$  to minimise the number of questions that have to be asked. For example, if the value of *P* has not yet been determined, but *Q* is known to be false, a smart logic engine will not evaluate  $P \square Q$  blindly from left to right, but will avoid evaluating *P*. The expression is *false*, irrespective of the value of *P*.

### 6.5. A Reaction System

A third approach to writing expert systems is a **reaction system** (or **production system**). Reaction system rules have an *if...add...delete* structure, which can be used to directly model **situations** that change over time.<sup>44</sup> The language CLIPS is specifically designed for this style of programming.<sup>45</sup>

In the following example,<sup>46</sup> a robot is to be programmed to bag groceries at a checkout. The program is written here in a style close to CLIPS, but not identical to it. We begin by including a CLIPS interpreter, which we shall discuss shortly:

```
:- include(clips).
```

Following the style of CLIPS, we first define templates for the kinds of facts we want to represent:

```
deftemplate(property:{slot:name, slot:container, slot:size, slot:frozen}).
deftemplate(item:{slot:name, slot:status, slot:in_freezer_bag}).
deftemplate(bag:{slot:name, slot:status, slot:items, multislot:contents}).
deftemplate(step:{slot:name}).
deftemplate(reply:{slot:answer}).
```

A fact has a name, e.g., **property**, and a set of **slots**, e.g., **container**. A **multislot** is a CLIPS concept similar to a Prolog list.

The next stage is to declare the initial situation:

```
deffacts({
  property:{name=bread, container=bag, size=medium, frozen=no},
  property:{name=glop, container=jar, size=small, frozen=no},
  property:{name=granola, container=box, size=large, frozen=no},
  property:{name=ice_cream, container=carton, size=medium, frozen=yes},
  property:{name=potato_chips, container=bag, size=medium, frozen=no},
  property:{name=pepsi, container=bottle, size=large, frozen=no},
  step:{name=check_order},
  bag:{name=bag1, status=current, items=0, contents=[]},
  bag:{name=bag2, status=empty, items=0, contents=[]},
  bag:{name=bag3, status=empty, items=0, contents=[]},
  bag:{name=bag4, status=empty, items=0, contents=[]},
  bag:{name=bag5, status=empty, items=0, contents=[]},
  item:{name=bread, status=to_be_bagged, in_freezer_bag=no},
  item:{name=glop, status=to_be_bagged, in_freezer_bag=no},
  item:{name=granola, status=to_be_bagged, in_freezer_bag=no},
```

<sup>44</sup> That doesn't mean that forward or backward chaining can't.

<sup>45</sup> **C Language Interface Production System**. See J. Giarratano & G. Riley, *Expert Systems: Principles & Programming*, PWS Publishing, 1998.

<sup>46</sup> Taken from Winston, *op. cit.*, Ch. 7: Rules and Rule Chaining.

```

item:{name=ice_cream, status=to_be_bagged, in_freezer_bag=no},
item:{name=potato_chips, status=to_be_bagged, in_freezer_bag=no}}).

```

The property facts describe some constant properties of food items. The `item` facts describe the items the robot must pack. The `bag` facts describe the 5 bags at the robot's disposal. The `step` fact is used to guide the robot through a sequence of steps: First, the robot must bag large items, to a maximum of 6 per bag. Then the robot must pack medium items, to a maximum of 12 per bag. Finally, the robot must bag small items, to a maximum of 18 per bag. The robot must always take care to enclose frozen items in freezer bags. There is a special offer with potato chips that gives a discount on Pepsi. When the robot finds a customer has chips but no Pepsi, it must ask if the customer wants Pepsi. All bottles are treated as large items, and bagged first.

Each rule has the form, `Name : Conditions => Actions`.<sup>47</sup> Here, for example, are the first two:

```

'Chips but no Pepsi.':
  [Step <- step:{name=check_order},
   item:{name=potato_chips, status=to_be_bagged},
   not(item:{name=pepsi, status=to_be_bagged})]
=>
  [printout('Would you like to add Pepsi to your order? '),
   read(X),
   assert(reply:{answer=X}),
   modify(Step:{name=check_reply})].

'Pepsi offer does not apply.':
  [Step <- step:{name=check_order}]
=>
  [modify(Step:{name=bag_large_items})].

```

The name of the first rule is 'Chips but no Pepsi.' This name will be displayed in the trace of execution. The conditions of a rule are tested in sequence, and are written (here) as a list. Each condition must be true for the rule to fire. The rules are matched against the current facts in the database. If a fact is negated, that fact must be absent. It is possible to bind a fact to a variable, e.g., `Step`. When a rule fires, a sequence of actions is executed.

CLIPS provides several possible actions, such as those used here to display a message and read a reply, but more importantly, actions can assert new facts or retract existing ones. Instead of retracting a fact, then asserting a new one, it is possible to modify an existing fact.

The second rule will fire only if the first does not. By setting the name attribute of `Step` differently, the two rules provide conditional control of the sequence of execution.

In practice, the robot would have to actually put items into bags. Here, the program contents itself with displaying a message. The `crLf` action starts a new line:

```

'Customer accepts Pepsi.':
  [Step <- step:{name=check_reply},
   Reply <- reply:{answer=yes}]
=>
  [retract(Reply),
   printout('Adding Pepsi to order.'), crLf,
   modify(Step:{name=bag_large_items}),
   assert(item:{name=pepsi, status=to_be_bagged, in_freezer_bag=no})].

'Customer declines Pepsi.':
  [Step <- step:{name=check_reply},
   Reply <- reply:{answer=no}]
=>
  [retract(Reply),
   modify(Step:{name=bag_large_items})].

'Customer fails to answer yes or no.':
  [Step <- step:{name=check_reply},
   Reply <- reply:{}]
=>
  [retract(Reply),
   printout('Please reply "yes." or "no."'), crLf,
   modify(Step:{name=check_order})].

```

<sup>47</sup> This is not the exact form used by CLIPS. It is close to it, but is designed so that, given suitable operator definitions, each CLIPS statement forms a valid Prolog fact.

```
'Item not in freezer bag':
  [Item <- item:{status=to_be_bagged, name=X, in_freezer_bag=no},
   property:{name=X, size=medium, frozen=yes}]
=>
  [printout('Putting '), printout(X), printout(' in a freezer bag.'), crlf,
   modify(Item:{in_freezer_bag=yes})].
```

When a condition is not a simple equality, the `test` function must be used. It is also possible to use arithmetic expressions when asserting or modifying facts:

```
'Bag a bottle.':
  [step:{name=bag_large_items},
   Item <- item:{name=X, status=to_be_bagged},
   property:{name=X, container=bottle},
   Bag <- bag:{status=current, items=N, contents=C},
   test(N<6)]
=>
  [printout('Bagging '), printout(X), crlf,
   modify(Item:{status=bagged}),
   modify(Bag:{items=N+1, contents=[X|C]})].

'Bag a large item.':
  [step:{name=bag_large_items},
   Item <- item:{name=X, status=to_be_bagged},
   property:{name=X, size=large},
   Bag <- bag:{status=current, items=N, contents=C},
   test(N<6)]
=>
  [printout('Bagging '), printout(X), crlf,
   modify(Item:{status=bagged}),
   modify(Bag:{items=N+1, contents=[X|C]})].
```

In general, more than one rule may be applicable at any one time. Such rules are added to an **agenda**. Choosing which of the agenda's rules should fire is called **conflict resolution**. Here, conflicts are always resolved in favour of the first rule written. As a result, the following rule can only fire when the number of items in the current bag is no longer less than 6.

```
'Find a new bag for a large item.':
  [step:{name=bag_large_items},
   item:{status=to_be_bagged, name=X},
   property:{name=X, size=large},
   Old <- bag:{status=current},
   New <- bag:{status=empty}]
=>
  [printout('Taking a new bag.'), crlf,
   modify(Old:{status=filled}),
   modify(New:{status=current})].
```

Once all large items are bagged, a new bag is taken, and a similar set of rules deals with medium-sized items. Changing the step fact changes the program's focus:

```
'Bagging large items done.':
  [Step <- step:{name=bag_large_items},
   Old <- bag:{status=current},
   New <- bag:{status=empty}]
=>
  [printout('Taking a new bag.'), crlf,
   modify(Old:{status=filled}),
   modify(New:{status=current}),
   modify(Step:{name=bag_medium_items})].
```

```

'Bag a medium item.':
  [step:{name=bag_medium_items},
   Item <- item:{status=to_be_bagged, name=X},
   property:{name=X, size=medium},
   Bag <- bag:{status=current, items=N, contents=C},
   test(N<12)]
=>
[printout('Bagging '), printout(X), crlf,
 modify(Bag:{items=N+1, contents=[X|C]}),
 modify(Item:{status=bagged})].

'Find a new bag for a medium item.':
  [step:{name=bag_medium_items},
   Old <- bag:{status=current},
   New <- bag:{status=empty},
   item:{status=to_be_bagged, name=X},
   property:{name=X, size=medium}]
=>
[printout('Taking a new bag.'), crlf,
 modify(Old:{status=filled}),
 modify(New:{status=current})].

```

Once medium-sized items are bagged, the focus shifts to small items:

```

'Bagging medium items done.':
  [Step <- step:{name=bag_medium_items},
   Old <- bag:{status=current},
   New <- bag:{status=empty}]
=>
[printout('Taking a new bag.'), crlf,
 modify(Old:{status=filled}),
 modify(New:{status=current}),
 modify(Step:{name=bag_small_items})].

'Bag a small item.':
  [step:{name=bag_small_items},
   Item <- item:{status=to_be_bagged, name=X},
   property:{name=X, size=small},
   Bag <- bag:{status=current, items=N, contents=C},
   test(N<18)]
=>
[printout('Bagging '), printout(X), crlf,
 modify(Bag:{items=N+1, contents=[X|C]}),
 modify(Item:{status=bagged})].

'Find a new bag for a small item.':
  [step:{name=bag_small_items},
   Old <- bag:{status=current},
   New <- bag:{status=empty},
   item:{status=to_be_bagged, name=X},
   property:{name=X, size=small}]
=>
[printout('Taking a new bag.'), crlf,
 modify(Old:{status=filled}),
 modify(New:{status=current})].

'Bagging small items done.':
  [Step <- step:{name=bag_small_items}]
=>
[modify(Step:{name=done})].

```

There are *no* facts defined for the done step. Therefore, on reaching this step, the program must terminate.

Here follows an example execution of the program. The display from 'printout' is on the left, the logic engine's trace of the rules is on the right. When no more rules can fire, the logic engine displays the final set of facts. This is a useful debugging tool if the program stops before it should:

```
| ?- [bagger].
compiling /Users/dwyer/Documents/DSTO/Lecture Program Examples/bagger.pro for byte
code...
/Users/dwyer/Documents/DSTO/Lecture Program Examples/bagger.pro compiled, 429 lines read
- 84486 bytes written, 219 ms
(30 ms) yes
| ?- run_clips.
                                     (Rule: "Chips but no Pepsi." is firing.)
Would you like to add Pepsi to your order? yes.
                                     (Rule: "Customer accepts Pepsi." is firing.)
Adding Pepsi to order.
                                     (Rule: "Item not in freezer bag" is firing.)
Putting ice_cream in a freezer bag.
                                     (Rule: "Bag a bottle." is firing.)
Bagging pepsi
                                     (Rule: "Bag a large item." is firing.)
Bagging granola
                                     (Rule: "Bagging large items done." is firing.)
Taking a new bag.
                                     (Rule: "Bag a medium item." is firing.)
Bagging ice_cream
                                     (Rule: "Bag a medium item." is firing.)
Bagging bread
                                     (Rule: "Bag a medium item." is firing.)
Bagging potato_chips
                                     (Rule: "Bagging medium items done." is firing.)
Taking a new bag.
                                     (Rule: "Bag a small item." is firing.)
Bagging glop
                                     (Rule: "Bagging small items done." is firing.)
*** No more rules can fire. ***
```

Final state of database:

```
bag:{contents=[glop], items=1, name=bag3, status=current}
bag:{contents=[granola, pepsi], items=2, name=bag1, status=filled}
bag:{contents=[potato_chips, bread, ice_cream], items=3, name=bag2,
status=filled}
bag:{name=bag4, status=empty, items=0, contents=[]}
bag:{name=bag5, status=empty, items=0, contents=[]}
item:{in_freezer_bag=no, name=bread, status=bagged}
item:{in_freezer_bag=no, name=glop, status=bagged}
item:{in_freezer_bag=no, name=granola, status=bagged}
item:{in_freezer_bag=no, name=pepsi, status=bagged}
item:{in_freezer_bag=no, name=potato_chips, status=bagged}
item:{in_freezer_bag=yes, name=ice_cream, status=bagged}
property:{name=bread, container=bag, size=medium, frozen=no}
property:{name=glop, container=jar, size=small, frozen=no}
property:{name=granola, container=box, size=large, frozen=no}
property:{name=ice_cream, container=carton, size=medium, frozen=yes}
property:{name=pepsi, container=bottle, size=large, frozen=no}
property:{name=potato_chips, container=bag, size=medium, frozen=no}
step:{name=done}
```

(50 ms) yes

The following program (clips.pro) illustrates the ease with which Prolog can be used to implement an inference engine.

The program begins by declaring three additional operators, needed to parse the rules:

```
:- op(500, xfy, [:, =>, <-]).
```

(Because bagger.pro includes clips.pro in its first line, these declarations are in force throughout bagger.pro.)

The inference engine makes use of `sets.pro`. It initialises the facts database, then fires the rules. The facts are represented as a set.

```
:- include(sets).
run_clips :- initialise(Facts), fire_rules(Facts).
initialise(Facts) :- deffacts(Facts), type_check_facts(Facts).
```

Since CLIPS is effectively a programming language, programs written in CLIPS formalism are just as easy to get wrong as those in any other. It was found valuable for the program to make a simple type check to ensure that every attribute has been declared by `deftemplate`.

```
type_check_facts(Facts) :- all(Type:Slots, Facts), type_check_slots(Slots, Type), fail.
type_check_facts(_).

type_check_slots(Slots, Type) :-
    deftemplate(Type:AllowedSlots),
    all(Slot=_, Slots), type_check_slot(Slot, AllowedSlots, Type),
    fail.
type_check_slots(_, _).

type_check_slot(Slot, AllowedSlots, _) :- in(_:Slot, AllowedSlots), !.
type_check_slot(Slot, _, Type) :-
    print('*Error* '), print(Slot), print(' is not a slot in '), print(Type), nl.
```

Rules are fired in a loop. The rules are scanned from top to bottom, and the first rule that *can* fire is fired. This is simplistic compared with the strategy actually used by CLIPS. Conflict resolution normally takes other factors into account, such as the number of conditions that the rule contains, the number of different bindings of its variables that are possible, and how recently it fired.

Remember that rules are simply Prolog facts:

```
fire_rules(Facts0) :-
    ID: Conditions => Actions,
    satisfied(Conditions, Facts0), !,
    atom_codes(ID, List), length(List, Len), Tab is 58-Len,
    tab(Tab), print('Rule: '), print(ID), print(' is firing. '), nl,
    execute(Actions, Facts0, Facts1),
    !, fire_rules(Facts1).

fire_rules(Facts) :-
    tab(20), print('*** No more rules can fire. ***'), nl, nl,
    print('Final state of database:'), nl, report_facts(Facts).

report_facts(Facts) :- all(Fact, Facts), nl, tab(1), print(Fact), fail.
report_facts(_) :- nl.
```

A rule is satisfied only if all its conditions are satisfied. There are four cases to consider:

- A negated fact.
- The test function.
- A fact that becomes bound to a variable.
- A simple positive fact.

The `satisfied` predicate also type-checks each fact that is matched; a rule that accidentally referred to a non-existent slot would never fire, but such an error might otherwise take a long time to debug:

```
satisfied([], _) :- !.
satisfied([Condition|Conditions], Facts) :-
    satisfied(Condition, Facts), satisfied(Conditions, Facts).
satisfied(not(Goal), Facts) :- satisfied(Goal, Facts), !, fail.
satisfied(not(_), _) :- !.
satisfied(test(Goal), _) :- !, call(Goal).
satisfied(Fact <- Type:Equalities, Facts) :- !,
    type_check_slots(Equalities, Type),
    Fact=Type:Attributes,
    all(Fact, Facts),
    subset(Equalities, Attributes).
satisfied(Type:Equalities, Facts) :- !, satisfied(_ <- Type:Equalities, Facts).
```

Executing the actions of a rule also deals with a series of special cases. Again, for ease of debugging, when facts are asserted or modified, the slots of the new fact are type-checked.

```

execute([], Facts, Facts) :- !.
execute([Action|Actions], Facts0, Facts) :- !,
    execute(Action, Facts0, Facts1),
    execute(Actions, Facts1, Facts).
execute(assert(Type:Slots0), Facts0, Facts) :-
    assign_slots(Slots0, Slots),
    type_check_slots(Slots, Type),
    union({Type:Slots}, Facts0, Facts), !.
execute(modify(Fact:Slots0), Facts0, Facts) :-
    Fact=Type:Slots1,
    type_check_slots(Slots0, Type),
    all(Fact, Facts0),
    difference(Facts0, {Fact}, Facts1),
    modify_slots(Slots0, Slots1, Slots),
    union({Type:Slots}, Facts1, Facts), !.
execute(retract(Fact), Facts0, Facts) :- !, difference(Facts0, {Fact}, Facts).
execute(printout(X), Facts, Facts) :- !, print(X).
execute(read(X), Facts, Facts) :- !, read(X).
execute(crlf, Facts, Facts) :- !, nl.

```

The value assigned to a slot can be an arithmetic expression. There is a danger that the expression will not be well-formed, or might try to evaluate unbound variables. If this occurred, `is/2` would throw an exception. The Prolog `catch/3` predicate handles exceptions. Its second argument is, in general, a pattern; `catch` handles any exceptions that match the pattern, in this case, anything:

```

assign_slots(Slots0, Slots) :-
    findset(Attribute=Value,
        (all(Attribute=Expn, Slots0),
         catch(evaluate(Expn, Value), _,
              (numbervars(Expn), print('Can't evaluate '), print(Expn), nl))),
        Slots).

evaluate(X+Y, Value) :- evaluate(X, X1), evaluate(Y, Y1), Value is X1+Y1, !.
evaluate(X-Y, Value) :- evaluate(X, X1), evaluate(Y, Y1), Value is X1-Y1, !.
evaluate(X*Y, Value) :- evaluate(X, X1), evaluate(Y, Y1), Value is X1*Y1, !.
evaluate(X/Y, Value) :- evaluate(X, X1), evaluate(Y, Y1), Value is X1/Y1, !.
evaluate(Expn, Expn).

```

When a fact is modified, unaltered slots must be copied:

```

modify_slots(Slots0, Slots1, Slots) :-
    findset(Attribute=Value,
        (all(Attribute=Value0, Slots1),
         modify_value(Attribute, Slots0, Value0, Value)),
        Slots).

modify_value(Attribute, Slots, _, Value) :-
    in(Attribute=Expn, Slots), evaluate(Expn, Value), !.
modify_value(_, _, Value, Value).

```

This implementation is misleading, because real reaction systems are more efficient than this. When a rule is selected for firing, this program backtracks through the list of rules, testing conditions until a rule is found all of whose conditions are satisfied. Reaction systems such as CLIPS do not have to scan long lists of rules to see which can fire. They use a method called the **rete algorithm**.<sup>48</sup> This algorithm pre-compiles the rules into paths within a decision tree. At execution time, it dynamically updates the state of progress of rules along paths in the tree as facts are asserted or retracted. When a fact is asserted, some rules may be able to make progress; when a fact is retracted, some may revert to an earlier position. A secondary data structure links each type of fact to the rules that it can affect. This speeds the search for active rules, which is fundamental to any practical reaction system. Indeed, at a time when computers were much slower than today, the rete algorithm was instrumental in making expert systems viable in real-world applications.

---

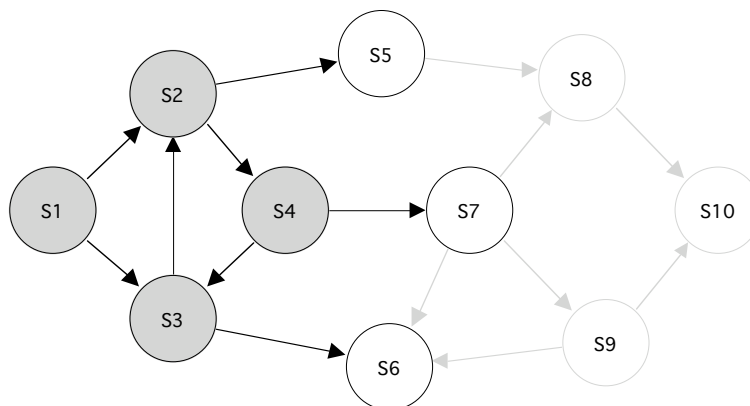
<sup>48</sup> To rhyme with 'treaty'.

## 7. Search

### 7.1. Search Problems

Imagine yourself in one of those old-fashioned mazes, fenced in by dense hedgerows. You want to find your way to the tea-room in the centre of the maze. Some paths are straight and inviting, but others are winding. Many lead to dead-ends. You will only know when you are at the tea-room when you can see it. This is a *search problem*. Specifically, a *blind search*, because you have no idea when you are getting near your destination. On the other hand, if the tea-room is marked by a tall flagpole that you can see from all over the maze, it becomes an *informed search*.

It is usual to liken a search space to a graph: vertices represent states (e.g., places), and directed edges represent the possible changes of state brought about by agent actions (e.g., moving). Each edge has an associated cost, and the cost of a path is (usually) the sum of the costs of its edges.



In the above graph,  $S_1$  is the start state, and  $S_{10}$  is the goal state.  $S_1$ ,  $S_2$ ,  $S_3$  and  $S_4$  (grey circles) have already been visited. Their properties, including their out-going edges, are completely known. In addition, an algorithm may have discovered the *actual costs* of the paths from  $S_1$  to the states  $S_5$ ,  $S_6$  and  $S_7$ , (white circles), but has not yet visited them. This collection of states is called the **fringe** (or **frontier**) of the search. Which state should the search consider next? Should it consider  $S_5$ ,  $S_6$ , or  $S_7$ ? This will depend on the **priority** assigned to each state. Different search strategies assign priorities differently.

A vertex may be in the fringe even though it has already been visited. This is because it is usually more efficient to add a vertex to the fringe regardless, then check if it was already visited when it is removed from the fringe. In *some* kinds of search, the second visit may be useful because it leads to a better solution. In the above diagram, it is possible that, after  $S_3$  has been visited from  $S_1$ , the edge from  $S_4$  may add  $S_3$  to the fringe a second time.

The paths to the remaining states,  $S_8$ ,  $S_9$ , and  $S_{10}$ , are not yet known, because although the search knows about the edges *into* the fringe states, it does not yet know about the edges *leaving* the fringe states. Even the *existence* of states  $S_8$ ,  $S_9$ , and  $S_{10}$  may be uncertain.

As a concrete example, think of a robot that can navigate on a square grid, and has 4 possible actions: to go one square north, south, east or west. The *states* (vertices) would correspond to the positions of the robot. The *transitions* (edges) would correspond to one of the 4 actions. It is possible that some actions would have a higher cost than others, for example, ‘go north’ and ‘go south’ might cost more than ‘go east’ or ‘go west’.

#### 7.1.1. A General Algorithm

In Prolog, we may define an iterative search predicate with the following template,

```
search(+Goal, +Fringe, +Visited, +Graph, -Path)
```

- `Goal` is a pattern that matches any desired final state,
- `Fringe` is a *priority queue* of triples of the form, `Priority-State-Thap`. `State` is a (white) state that can be reached from the start state by executing the actions in `Thap` *in reverse order*, and `Priority` is the priority of the state. The list is *ordered* by ascending `Priority`, therefore the first element of `Fringe` has the *lowest* priority value, and has the *highest* priority. (We shall utilise set operations, including `least/3`, to implement the priority queue.)
- `Visited` is a *set* of all (grey) states already visited.
- `Graph` is whatever data structure describes the search space.<sup>49</sup>
- `Path` is a *list* of actions that reach the goal.

<sup>49</sup> In the case that the search space is defined entirely by rules, this argument isn’t needed.

## Search

In principle, a search can generate many possible solutions (if there are any); in practice, an agent is usually only interested in the first solution found.

On each loop of the iteration, `search` removes the least element from `Fringe`, that with earliest priority.

- If its `State` *matches* a goal state, `Thap` is the reverse of the solution path.
- If the `State` is a member of `Visited`, a path to it has already been found, so it is ignored. Only the first path to reach a state is ever considered.
- Otherwise, `State` is added to `Visited`, and `extend_path` then extends the path beyond `state`, by adding each of its outgoing edges in turn, resulting in `ExtendedPaths`. This gives a *set* of new states, with associated paths and priorities, which are added to `Fringe`, carefully preserving its ordering by priority.

The search loop then iterates:

```
search(Goal, Fringe, _, _, Path) :-
    least(Fringe, _-Goal-Thap, _), !, reverse(Thap, Path) .
search(Goal, Fringe0, Visited, Graph, Solution) :-
    least(Fringe0, _-State-, Fringe1),
    in(State, Visited), !,
    search(Goal, Fringe1, Visited, Graph, Solution).
search(Goal, Fringe0, Visited0, Graph, Solution) :-
    least(Fringe0, _-State-Thap, Fringe1),
    union({State}, Visited0, Visited1),
    extend_path(State, Thap, Goal, Graph, Neighbours),
    union(Fringe1, Neighbours, Fringe2),
    !, search(Goal, Fringe2, Visited1, Graph, Solution).
```

For this predicate to function correctly, it needs to be initialised:

```
search(Start, Goal, Graph, Solution) :- search(Goal, {0-Start-[]}, {}, Graph, Solution).
```

This leaves us with the task of defining `extend_path`:

```
extend_path(State, Thap, Goal, Graph, Neighbours) :-
    findset(Neighbour, reachable(State, Thap, Goal, Graph, Neighbour), Neighbours).
```

We may now define,

```
reachable(State0, Thap0, Goal, Graph, Priority-State-Thap) :-
    operator(State0, Action, Graph, State),
    Thap=[Action|Thap0],
    cost(Thap, Cost),
    estimate(State, Goal, Graph, Heuristic),
    Priority is Cost+Heuristic.
```

We need three further predicates, peculiar to the problem. The `operator/4` predicate has the template,

```
operator(+State0, ?Action, +Graph, -State)
```

The idea is that if `State0` matches the state before `Action`, then `State` matches the state resulting from `Action`. It is assumed that, using `Graph`, `operator/4` can generate all admissible agent actions on backtracking, e.g., ‘go north’, ‘go south’, etc.

We assume also that the predicate `cost/2` computes the cost of a (reversed) path.

```
cost(+Thap, -Cost).
```

We finally assume that we have a predicate `estimate/4`, with template,

```
estimate(+State, +Goal, +Graph, -Heuristic).
```

The idea is that `estimate` will compute an estimate (`Heuristic`) of the cost of the remaining path from `State` to `Goal`. For example, the navigating robot might estimate the cost of the remaining path as the bee-line distance from the current square to the goal square, ignoring any (as yet unforeseen) obstacles that may be in the way.

By determining the order in which partial solutions are examined, the choices of `cost` and `estimate` greatly affect the course of the search.

## 7.2. Blind Search

In some problems the agent has no way of estimating the remaining cost of reaching the goal. We are therefore forced to define `estimate` as follows,

```
estimate(_, _, _, 0).
```

This situation is called **blind search**, or **uninformed search**.

### 7.2.1. Breadth-first Search

**Breadth-first search (BFS)** is a blind search in which each action is given an *equal* cost. The cost of any path is simply equal to its length:

```
cost(Thap, Cost) :- length(Thap, Cost).
```

Since the search algorithm keeps the fringe sorted by priority, this means that all the states adjacent to the start state are considered first, then all those at distance 2, and so on.<sup>50</sup> As a result, the first path to the goal will be one with the least number of edges. This property holds for every intermediate state visited in the search, which justifies `search` ignoring every path except the first to reach it.

### 7.2.2. Depth-first Search

**Depth-first search (DFS)** mistakes activity for productivity; the greater the distance from the start, the greater the progress towards the goal:

```
cost(Thap, Cost) :- length(Thap, Length), Cost is -Length.
```

DFS first adds to the fringe all states adjacent to the start. Then, when it considers the first of these, it places all its descendants (at distance 2) *before* the remaining states at distance 1. Likewise, when it considers the first state at distance 2, it places its descendants at distance 3 ahead of the other states at distance 2.<sup>51</sup> Although DFS favours longer paths, there is no guarantee that the *first* path it finds to the goal is the longest.

### 7.2.3. Uniform-cost Search

**Uniform-cost search** (or **best-first search**) uses the true cost of each path. Therefore, the first state in the fringe is the one that is reached most cheaply from the start state. When the first element of the fringe becomes the goal state, the lowest-cost path to the goal has been found.

### 7.2.4. Analysis of Blind Search

The chief disadvantage of *breadth-first* and *uniform-cost* search is that the number of states stored in the fringe can grow exponentially. An agent that can choose one of 4 actions has 4 states in the fringe that are at distance 1 from the start, then 16 at distance 2, then 64 at distance 3, and so on. In practice, not all these states need be different. The agent that can move one square north, south, east or west can reach 4 possible squares after one move, 8 squares after two moves, 12 squares after 3 moves, and so on.<sup>52</sup> In this particular example, because `search` tests to see if states have already been visited, the number of states in the fringe grows only in proportion to the length of the path.

In the case of *depth-first* search, the number of states on the fringe grows no faster than the search depth. For example, when the search depth reaches 3, the agent that can move north, south, east or west has 4 new states at depth 3 to consider, plus 3 states that were set aside at depth 2 and 3 states that were set aside at depth 1.

A potential danger with depth-first search is that it will be trapped by a cycle. For example, in the graph of Section 7.1, a search could be trapped by the infinite sequence  $S1, S2, S4, S3, S2, S4, S3, \dots$ . However, since it is careful to ignore any path that leads to a previously visited state, `search` avoids this problem.

## 7.3. Informed Search

In an **informed search**, the agent has some way of estimating the cost of reaching the goal from any given state, called a *heuristic*. The intention is to give priority to those paths that currently seem most promising.

The heuristic need not be exact. In the case of the agent that can move north, south, east or west, the heuristic could be the bee-line or ‘Euclidian’ distance from the state to the goal, or more realistically, it could be the city-block or ‘Manhattan’ distance. Such a heuristic will usually be an under-estimate; the agent may have to go round an unforeseen obstacle. The value of this heuristic would be calculated by the `estimate` predicate.

A good heuristic should yield a result as close to the true cost as possible. On the other hand, it should not take too long to calculate. An ideal heuristic would yield the true cost of reaching the goal, but since this can usually only be found by a search, computing its values would cost more time than it could save.

### 7.3.1. Greedy Search

Greedy search tries to find a solution as *quickly* as possible. It therefore favours states that have the least estimated distance to the goal. It totally disregards the prior cost of reaching the fringe state from the start state:

```
cost(_, 0).
```

One reason why we might adopt this strategy is that the problem has a cost function that is hard to quantify; all we have is a heuristic that can order states according to their *relative* closeness to the goal. The solution found by greedy search is not necessarily optimal.

<sup>50</sup> The fringe can therefore be stored in a FIFO queue.

<sup>51</sup> The fringe can therefore be stored in a LIFO stack.

<sup>52</sup> Assuming it does not retrace its path.

### 7.3.2. A\* Search

A\* (A-star) search uses the true cost of reaching a state from the start state, plus the value of an **admissible heuristic** to estimate the distance to the goal node. A heuristic is *admissible* if it never over-estimates the true cost. If this is so, *the first solution found is the lowest-cost solution*.

To see why this is so, remember that at the end of the search, the priority of the solution will equal its true cost, and the heuristic will have a value of zero. The paths remaining in the fringe have estimated costs *at least as great* as the cost of the solution. Since these estimates are *lower* bounds on the true cost, and therefore conservative, *no path to the goal via a fringe state can cost less than the solution*. On the other hand, if the heuristic had been an over-estimate, an unpromising state in the fringe might yet lead to a cheaper path.

What is true for the goal state is true for every other state visited during the search. Therefore, it is safe to ignore all but the first path to reach a given state. This is an application of the Dynamic Programming<sup>53</sup> principle:

***The best way to get from A to C via B  
consists of the best way to get from A to B,  
plus the best way to get from B to C.***<sup>54</sup>

There is some confusion about the difference between the A\* search algorithm and the much earlier Operations Research technique called ‘branch-and-bound’. Some authors consider that branch-and-bound does not require the heuristic to be a lower bound on the true cost—in which case it is hard to see why it is called ‘branch-and-bound’. Others claim branch-and-bound to be the case where the test for membership of *visited* is ignored (A\* = Branch & Bound+Dynamic Programming). (The algorithm would still work, *provided a solution exists*. Otherwise, it might become trapped by a cycle.) Personally, I think they are both the same, but AI researchers won’t admit that the Operations Researchers beat them to it.

### 7.3.3. Analysis of Informed Search

The efficiency of informed search depends entirely on the quality of the heuristic. A perfect heuristic would discover the correct path immediately, and the fringe would contain only the states set aside along the successful path. On the other hand, if the heuristic were useless, the situation would be the same as blind search; in the case of A\*, the search would degenerate to a uniform-cost search. It is therefore difficult to predict how well an informed search will work. Finding a good heuristic is sometimes a research problem.

## 7.4. Other Varieties of Search

### 7.4.1. Two-way Search

In some situations it is sensible to start from both the initial state and the goal, and search in both directions towards the middle, until a shared state is found. The basic idea is that if a search has branching factor  $B$  and depth  $D$ , the potential number of states to be examined is of the order of  $B^D$ . By searching in both directions, we have two searches of depth  $D/2$ , so the number of states is  $2B^{D/2}$ , which is smaller by a factor of  $1/2B^{D/2}$ .

The main drawback of this approach is that the only efficient way to find if the two searches meet at a common state is to store *all* the states reached by one of the searches.

### 7.4.2. Dynamic Programming

When the total number of states ( $S$ ) is not too great, we can use Dynamic Programming, which, as its name suggests, uses the Dynamic Programming Principle. It is similar in idea to breadth-first search; it first explores paths of length 1, then paths of length 2, then length 3 and so on. However, it does not consider *all* paths of length  $N$ , only those that can be optimal.

Suppose that, at stage  $N$ , Dynamic Programming has found the optimum path of length  $N$  to each state. To complete stage  $(N+1)$ , it considers every edge connecting pairs of states, extending the search by one edge. For each destination state, it can compute its cost as the sum of the cost of the edge, plus the known *optimal* cost of reaching the predecessor state. The least of these values is the optimal cost of the destination state. (Each destination state records its predecessor state so that the optimal path can be reconstructed.)

The algorithm is iterative. The complexity of computing each stage is proportional to the number of edges, which is bounded by the square of the number of states. The complexity of the algorithm is therefore polynomial: if  $P$  is the length of the optimal path, the search takes time  $O(P.S^2)$ .

### 7.4.3. Beam Search

Beam Search is used when the branching factor is high, and a good solution is needed, but not necessarily the optimal one. The basic idea is to limit the number of successor states that need to be considered, by discarding all but the  $B$  (say 3) most promising ones. This limits the number of states that need to be stored in the fringe in

<sup>53</sup> ‘Programming’ is a synonym for ‘planning’ here.

<sup>54</sup> The best path from A to C need not pass through B at all.

a uniform cost or informed search, and can greatly reduce the search space. It typically yields the optimum path, but this cannot be guaranteed.

## 8. Planning

### 8.1. Planning Problems

**Planning** is related to Search. The two techniques solve overlapping sets of problems.

Imagine that you are staying at a hotel. You are in Room 1012 on Floor 10. Your friend is in Room 1115 on Floor 11. You want to get to your friend's room. Clearly, you will need to find a lift or stairwell to get to the correct floor.

As a *search* problem, we might help the computer by providing a heuristic, for example the estimated Euclidean distance to your friend's room, to guide the search. We can soon see that this is not a good heuristic, because it is likely to guide you to Room 1015, directly below your friend's room, when you actually ought to be heading for the lift. A more effective heuristic would be to take the absolute difference between the two room numbers, which gives a weighting of 100 to the floor number, but although this heuristic emphasises the importance of being on the right floor, it doesn't guide you to the lift. It is possible to devise a heuristic that will guide you to the lift, e.g., a non-linear function that decreases in the direction of the lift if the floor is wrong, and decreases in the direction of the room if the floor is correct. However, this is almost cheating, because we have virtually solved the problem for the computer, rather than having let the computer solve it. Although one might argue that a suitable heuristic must exist for any problem, sooner or later, the pain of discovering one proves so great we are forced to adopt a different approach.

As a second example, think about the problem of putting a new battery in a torch. The first thing to do is to unscrew the cap of the battery compartment. This is a strange choice, because the torch won't function unless the cap is closed. We have to do it because having the cap removed is a precondition of being able to remove the old battery, which in turn is a precondition of putting in the new one.

A *search* strategy would begin at the start state, then explore different alternatives: operating the switch, removing the glass, removing the cap, until eventually stumbling on a correct sequence of moves. A *planner* would begin by recognising that inserting the battery is a key move, because there is no other way the torch can acquire a new battery. Therefore, it is necessary to get to a state where this is possible, even if this means initially moving *away* from the goal state. Likewise, a planner would recognise that there are only two ways to get to your friend's hotel floor, so you must get to either the lift or the stairwell.

This leads to an important realisation:

***The first thing a planner decides to do is rarely the thing it decides to do first.***

For example, the first thing a planner decides to do may be to insert a new battery, but the thing it decides to do first is to open the cap. Typically, a planner will recognise the need to pass through some key intermediate state, then create two sub-plans: one to get from the start state to the intermediate state, and the other to get from the intermediate state to the goal. Planners do not typically rely on heuristic functions.

### 8.2. Situation Calculus

We now consider the problem that a camera contains a used film and a dubious battery, which should be replaced by a new film and a new battery. In keeping with good knowledge representation principles, we give a declarative definition of the problem:

The initial state of the camera is given in full:

```
{at_end(film), camera_in_case, slot_closed(film), in(battery), in(film),
 slot_closed(battery)}
```

The initial state of the camera specifies that the camera is currently in its case, that the film slot is closed, the camera contains a film, which is at its end, the battery slot is closed, and a battery is in the camera.

Unlike the initial state, a goal state is usually given only partially; it should not be over-specified. In this particular example, it is not stated whether the camera should be in its case or not.

```
{in(film), at_start(film), unused(film), slot_closed(film), in(battery), ok(battery),
 slot_closed(battery)}.
```

In this instance, in both the initial and goal states there is a film and a battery in the camera, with their slots closed, but it is precisely these conditions that need to be changed before the film and battery can be replaced.

Planning problems are usually described in a notation similar to the widely-used formalism of STRIPS, an early planner. Each **operator** is described by three rules:

- **precondition** lists the predicates that must be true before the operator can be applied,
- **positive\_effect** lists the predicates that will become true as a result of executing it,
- **negative\_effect** lists the predicates that it will cause to become false.

For uniformity, we include two dummy operators, **start** and **finish**. The **positive\_effect** of **start** defines the initial state, and the **precondition** of **finish** defines the goals:

```

precondition(start, {}).
positive_effect(start, {in_case, slot_closed(film), in(film),
                        at_end(film), slot_closed(battery), in(battery)}).
negative_effect(start, {}).

precondition(finish, {in(film), unused(film), slot_closed(film),
                      in(battery), ok(battery), slot_closed(battery)}).
positive_effect(finish, {}).
negative_effect(finish, {}).

precondition(open_case, {in_case}).
positive_effect(open_case, {outside_case}).
negative_effect(open_case, {in_case}).

precondition(close_case, {outside_case, slot_closed(film), slot_closed(battery)}).
positive_effect(close_case, {in_case}).
negative_effect(close_case, {outside_case}).

precondition(open_slot(battery), {outside_case, slot_closed(battery)}).
positive_effect(open_slot(battery), {slot_open(battery)}).
negative_effect(open_slot(battery), {slot_closed(battery)}).

precondition(remove(battery), {in(battery), slot_open(battery)}).
positive_effect(remove(battery), {slot_empty(battery)}).
negative_effect(remove(battery), {in(battery)}).

precondition(remove(film), {in(film), at_start(film), slot_open(film)}).
positive_effect(remove(film), {slot_empty(film)}).
negative_effect(remove(film), {in(film)}).

precondition(insert_new(Thing), {slot_empty(Thing), slot_open(Thing)}).
positive_effect(insert_new(battery), {in(battery), ok(battery)}).
positive_effect(insert_new(film), {in(film), unused(film), at_start(film)}).
negative_effect(insert_new(Thing), {slot_empty(Thing)}).

precondition(close_slot(battery), {outside_case, slot_open(battery)}).
positive_effect(close_slot(battery), {slot_closed(battery)}).
negative_effect(close_slot(battery), {slot_open(battery)}).

precondition(open_slot(Thing), {outside_case, slot_closed(Thing)}).
positive_effect(open_slot(film), {slot_open(film)}).
positive_effect(close_slot(film), {slot_closed(film)}).
negative_effect(open_slot(Thing), {slot_closed(Thing)}).

precondition(rewind(film), {outside_case, in(film), at_end(film)}).
positive_effect(rewind(film), {at_start(film)}).
negative_effect(rewind(film), {at_end(film)}).

precondition(take_pictures, {in(film), at_start(film), unused(film),
                             slot_closed(film), in(battery),
                             ok(battery), slot_closed(battery)}).
positive_effect(take_pictures, {at_end(film)}).
negative_effect(take_pictures, {at_start(film), unused(film)}).

```

In STRIPS all predicates must be grounded. Modern planners allow more flexibility, including negation, and the use of variables. We have done this above, in a modest way:

```
precondition(insert_new(Thing), {slot_empty(Thing), slot_open(Thing)}).
```

Because the implementation of sets that we have chosen does not work well with ungrounded elements, we are unable to enlarge the STRIPS formalism any further.

A specific application of an operator to a particular situation (i.e., state) is called an **action**. A representation in which actions are applied to states (or snapshots) is called **situation calculus**:<sup>55</sup>

In specifying preconditions, we should be careful to include only those that are necessary for the action to be carried out; we should not include conditions that we think are a good idea. Otherwise we are imposing our own solution on the planner, and reducing its flexibility.

---

<sup>55</sup> The following directive is needed at the start of the Prolog file,  

```
:- discontiguous([precondition/2, positive_effect/2, negative_effect/2]).
```

Otherwise, the compiler would not allow the `precondition`, `positive_effect` and `negative_effect` rules to be interleaved as they are here.

### 8.3. Linear Planners

In a **linear planner**, the plan consists of a totally ordered *sequence* of actions. This contrasts with a non-linear planner, where the plan is a partially ordered network of actions, similar to those used by CPM (Critical Path Method).<sup>56</sup>

#### 8.3.1. Means-Ends Analysis

The first strategy used here is a form of *divide and conquer*. The planner first finds the set of *unsatisfied goals*: those predicates that are true in the goal state, but not in the initial state. It then chooses *any* operator that can satisfy *any* unsatisfied goal. This is called **means-ends analysis**. It then splits the planning problem into two sub-problems: a pre-plan, whose goal is to achieve the precondition of the operator, and a post-plan, whose initial state results from the effect of the operator. If there are no unsatisfied goals, then an empty plan is sufficient. If a choice proves wrong, the planner will backtrack.

A plan is a *sequence*; the order of its elements is important, so it is stored as a list.

```
:- include(sets).

plan :-
    positive_effect(start, Initial),
    precondition(finish, Goals),
    plan(Initial, Goals, Plan, _),
    write(Plan), nl.

plan(State, Goals, [], State) :- subset(Goals, State), !.
plan(State0, Goals, Plan, State) :-
    difference(Goals, State0, Unsatisfied_Goals),
    append(PrePlan, [Action|PostPlan], Plan),
    useful(Action, Unsatisfied_Goals),
    precondition(Action, Condition),
    plan(State0, Condition, PrePlan, State1),
    apply(State1, Action, State2),
    plan(State2, Goals, PostPlan, State).

useful(Action, Unsatisfied_Goals) :-
    positive_effect(Action, Achieves),
    \+disjoint(Unsatisfied_Goals, Achieves).

apply(State0, Action, State) :-
    negative_effect(Action, Clobbered),
    difference(State0, Clobbered, State1),
    positive_effect(Action, Goals),
    union(State1, Goals, State).
```

Unlike a search algorithm, this planner does not plan step-by-step from the initial state to the goal, but begins by choosing some useful action that will achieve at least one of the goal conditions. Although an action is chosen for its positive effects, its negative effects may **lobber** some previously achieved goal. This is a serious weakness of this particular planning algorithm.

#### 8.3.2. Iterative Deepening

From a search point of view, the above planner is partly depth-first and partly breadth-first. This arises because of the detailed behaviour of `append`. The first time `append` is called, it will suggest an empty pre-plan. On backtracking it will suggest pre-plans of length 1, length 2, and so on. Therefore, it will always search for the shortest possible pre-plans first.

On the other hand, `append` puts no constraint on the length of post-plan, and post-plans will be searched for in depth-first fashion. This is not desirable, because the algorithm has no protection against developing infinite plans, for example, `[open_slot(film), close_slot(film), open_slot(film), close_slot(film), open_slot(film), ...]`.

One solution would be to use a modification of the breadth-first search algorithm. However, we know that, unlike depth-first search, breadth-first search can need a lot of memory to store the *fringe*. We therefore use a search technique called **iterative deepening**: successive depth-first searches to steadily increasing depths. We use iterative deepening recursively, during the searches for pre- and post-plans. The modification to the algorithm is quite simple:

---

<sup>56</sup> If you don't know what CPM is, don't worry; all will become clear later.

```

plan(State, Goals, [], State) :- subset(Goals, State), !.
plan(State0, Goals, Plan, State) :-
    difference(Goals, State0, Unsatisfied_Goals), !,
    deepen(Plan),
    append(PrePlan, [Action|PostPlan], Plan),
    useful(Action, Unsatisfied_Goals),
    precondition(Action, Condition),
    plan(State0, Condition, PrePlan, State1),
    apply(State1, Action, State2),
    plan(State2, Goals, PostPlan, State).

deepen([]).
deepen([_|More]) :- deepen(More).

```

On backtracking, `deepen` binds its argument to a list of length 0, length 1, length 2, and so on. Therefore the first plan found will be the shortest.

Obviously, if the best plan has length  $L$ , then all the searches with length 0, 1, ...  $L-1$  will prove a waste of time. But this may not matter much. Suppose the branching factor per state is 2, and  $L$  is 6. Then the total search time is  $k+2k+4k+8k+16k+32k+64k=127k$ , so the search time is roughly doubled, from  $64k$  to  $127k$ . The greater the branching factor, the less inefficient iterative deepening becomes. If the average branching factor is  $B$ , the search time at depth  $L$  is  $T_L=B^Lk$ . The time with iterative deepening is given by

$$D_L=k+Bk+B^2k+\dots+B^Lk=(B^{L+1}-1)k\div(B-1).$$

For a branching factor of 10 and a length of 6,  $D_L=(9,999,999\div9)k=1,111,111k$ , which is only about 11% worse than  $1,000,000k$  for the final search alone.

### 8.3.3. Goal Protection

We can improve a planning algorithm by **protecting** goal conditions that have already been established. As things stand, an action may be chosen to achieve a certain goal condition, then an action in its post-plan might **clobber** the goal as a side-effect of achieving a different sub-goal. Although iterative deepening ensures that inefficient plans of this kind will not become solutions, much time can still be wasted in exploring them.

The answer is to record the set of goal conditions that have been achieved, then avoid clobbering them. This means the planning algorithm needs an additional argument: the set of **protected goals**. An action may then be chosen only if it preserves the protected goals, i.e., does not clobber any of them.

```

preserves(Action, Protected_Goals) :-
    negative_effect(Action, Clobbered),
    disjoint(Clobbered, Protected_Goals).

```

Once an action is chosen, the goals it achieves are protected throughout the post-plan:

```

plan(State, Goals, _, [], State) :- subset(Goals, State), !.
plan(State0, Goals, Protected0, Plan, State) :-
    difference(Goals, State0, UnsatisfiedGoals),
    deepen(Plan),
    append(PrePlan, [Action|PostPlan], Plan),
    useful(Action, UnsatisfiedGoals),
    preserves(Action, Protected0),
    precondition(Action, Condition),
    plan(State0, Condition, Protected0, PrePlan, State1),
    apply(State1, Action, State2),
    positive_effect(Action, Achieves),
    union({Achieves}, Protected0, Protected1),
    plan(State2, Goals, Protected1, PostPlan, State).

preserves(Action, Protected) :-
    negative_effect(Action, Clobbered),
    disjoint(Clobbered, Protected).

```

Any planner that proceeds from pre-conditions to post-conditions (effects) is called a **progression planner**.

### 8.3.4. Regression Planning

There is considerable heuristic value in **regression planning**: proceeding from goals to initial states. This is partly because the initial moves in a plan may need to move *away* from the goal state rather than towards it. For example, one of the earliest steps in changing a film is take out the old one. This achieves none of the goal conditions, and actually clobbers one: `in(film)`. On the other hand, the *last step in a plan is very tightly constrained*: it must achieve at least one of the goal conditions, *without clobbering any of the other goal conditions*. In addition, its preconditions have to be consistent with the *all* the goal conditions that it does *not* change. As a result, working backwards from the goal usually results in a lower branching factor than working

forwards from the start. A complication is that the goal is not a single state, but is any of several states with the desired properties.

To implement regression planning, we must have a means of regressing an action: finding its preconditions from its post-conditions. Suppose `Achieves` represents the positive effects of an action, and `PreCondition` is the precondition for the action. Then, given the goals `Goals`, the *regressed goals*, the conditions that need to hold before the action, are found as follows,

```
regress(Achieves, PreCondition, Goals, RegressedGoals) :-
    difference(Goals, Achieves, RemainingGoals),
    union(RemainingGoals, PreCondition, RegressedGoals).
```

Regressing a goal does not yield a state, but a new goal, which may specify the state only partially. This has the advantage that the sub-goal is not over-specified, but regression is also capable of generating combinations of sub-goals that are impossible to satisfy.

```
plan(State, Goals, []) :- subset(Goals, State), !.
plan(State, Goals, Plan) :-
    deepen(Plan),
    append(PrePlan, [Action], Plan),
    positive_effect(Action, Achieves),
    precondition(Action, PreCondition),
    \+disjoint(Achieves, Goals),
    preserves(Action, Goals),
    regress(Achieves, PreCondition, Goals, RegressedGoals),
    \+inconsistent(RegressedGoals),
    plan(State, RegressedGoals, PrePlan).
```

```
regress(Achieves, PreCondition, Goals, RegressedGoals) :-
    difference(Goals, Achieves, RemainingGoals),
    union(RemainingGoals, PreCondition, RegressedGoals), !.
```

Like the previous planner, it protects its goals through the `preserves` predicate.

The `inconsistent` predicate checks to see if the regressed goals are inconsistent. Unfortunately, the text of `inconsistent` depends on the particular planning task. For example, the camera problem might include the rule,

```
inconsistent(Goals) :- in(slot_open(film), Goals), in(slot_closed(film), Goals).
```

Therefore, unless the problem of inconsistent goals is ignored, the regression planner is not really working within the framework of the STRIPS formalism.

One way to generalize the formalism to deal with inconsistent goals would be to assume that the situation is made up of a set of *variables*. Thus, we would have the states of the film slot, the battery slot, the film, the battery and so on. For example, we could say that the film slot has two possible states: `film_slot(open)` and `film_slot(closed)`. Since the same thing can't be in two different states at the same time, it would be easy for a planner to detect that these two conditions are inconsistent. However, this would not cover *all* the angles: `film_slot(open)` is inconsistent with `camera(in_case)`. Inconsistencies cannot arise in planning from preconditions to effects. Provided the start state is consistent and the preconditions of each operator are satisfied at each step, it is impossible to generate an inconsistent state.

#### 8.4. Partial Order Planning

Here is a problem you face every morning, putting on your shoes and socks. Although you can make the mistake of putting a shoe onto your bare foot, you cannot fit a sock over your shoe:

```
precondition(start, {}).
positive_effect(start, {no_shoe(left), no_shoe(right), no_sock(left), no_sock(right)}).
negative_effect(start, {}).

precondition(finish, {shoe_on(left), shoe_on(right), sock_on(left), sock_on(right)}).
positive_effect(finish, {}).
negative_effect(finish, {}).

precondition(put_on_sock(left), {no_sock(left), no_shoe(left)}).
positive_effect(put_on_sock(left), {sock_on(left)}).
negative_effect(put_on_sock(left), {no_sock(left)}).

precondition(put_on_shoe(left), {no_shoe(left)}).
positive_effect(put_on_shoe(left), {shoe_on(left)}).
negative_effect(put_on_shoe(left), {no_shoe(left)}).
```

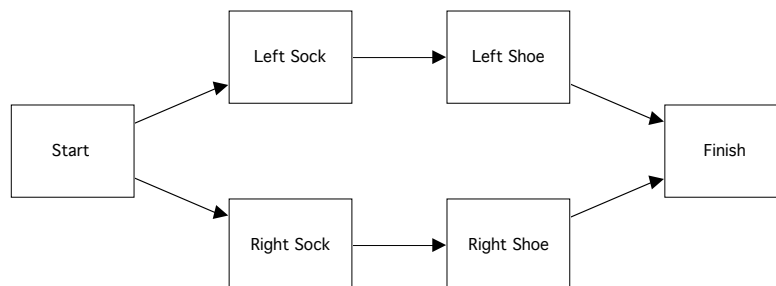
```

precondition(put_on_sock(right), {no_sock(right), no_shoe(right)}).
positive_effect(put_on_sock(right), {sock_on(right)}).
negative_effect(put_on_sock(right), {no_sock(right)}).

precondition(put_on_shoe(right), {no_shoe(right)}).
positive_effect(put_on_shoe(right), {shoe_on(right)}).
negative_effect(put_on_shoe(right), {no_shoe(right)}).

```

This will not cause any of the above planning algorithms any problems. However, the solution will be a particular *sequence*. In reality, it doesn't matter whether you put on your left sock and shoe then your right sock and shoe, or your right sock and shoe then your left sock and shoe. You can even put on both socks, then both shoes—in 4 different ways. These alternatives can be shown as a graph:



In a complex situation, it is obviously better to produce a graph of this kind, *then* choose the best sequence of actions later.<sup>57</sup>

Such a graph defines a *partial ordering*, so a planner that generates such a graph is called a **partial order planner**, or **non-linear planner**.

Before we discuss the details of a non-linear planner, consider the output of a non-linear planner for the above problem:

Each action is given a serial number. This is because, in a complex plan, the same operator may be used several times. If this happens in a linear plan, the action appears in the list more than once; but here we need the serial numbers to define the ordering unambiguously. To start the planning process, two dummy steps are created: 0: *start*, which needs no preconditions, and 1: *finish*, which achieves no goals. The steps of the above plan are as follows:

```

{0:start,
 1:finish,
 2:put_on_shoe(left),
 3:put_on_shoe(right),
 4:put_on_sock(left),
 5:put_on_sock(right)}

```

Numbers were allocated to steps in the order they were allocated to the plan. So, after the 0: *start* and 1: *finish* steps were added, the first new step was 2: *put\_on\_shoe(left)*.

The planner presents the above graph as a set of edges. The notation  $S_1$  **precedes**  $S_2$  is used here to mean that  $S_1$  must be done before  $S_2$ . Because the relation **precedes** is *transitive*,<sup>58</sup> some of edges listed below have been omitted from the graph to reduce visual clutter. In Prolog, we write `s1[]precedes s2`.

```

{0:start precedes 1:finish,
 0:start precedes 2:put_on_shoe(left),
 0:start precedes 3:put_on_shoe(right),
 0:start precedes 4:put_on_sock(left),
 0:start precedes 5:put_on_sock(right),
 2:put_on_shoe(left) precedes 1:finish,
 3:put_on_shoe(right) precedes 1:finish,
 4:put_on_sock(left) precedes 1:finish,
 4:put_on_sock(left) precedes 2:put_on_shoe(left),
 5:put_on_sock(right) precedes 1:finish,
 5:put_on_sock(right) precedes 3:put_on_shoe(right)}

```

Naturally, the planner must avoid creating cycles, or its result wouldn't be a partial ordering.

<sup>57</sup> Perhaps by using CPM (Critical Path Method).

<sup>58</sup> A relation  $\square$  is **transitive** if  $A \square B$  and  $B \square C$  implies that  $A \square C$ ; e.g., the  $>$  relation.

The central notion in partial-order planning algorithms is the **causal link**. A causal link records *why* an action was chosen. The notation  $S_1 \overset{G}{\square} S_2$  signifies that **step**  $S_1$  **satisfies goal**  $G$  **for step**  $S_2$ , i.e.,  $S_1$  has an effect  $G$  that is a precondition of  $S_2$ . In Prolog, we write `S1 achieves G for S2`.<sup>59</sup>

Here are the causal links the planner constructs for the shoes and socks problem.

```
{0:start satisfies no_shoe(left) for 2:put_on_shoe(left),
 0:start satisfies no_shoe(left) for 4:put_on_sock(left),
 0:start satisfies no_shoe(right) for 3:put_on_shoe(right),
 0:start satisfies no_shoe(right) for 5:put_on_sock(right),
 0:start satisfies no_sock(left) for 4:put_on_sock(left),
 0:start satisfies no_sock(right) for 5:put_on_sock(right),
 2:put_on_shoe(left) satisfies shoe_on(left) for 1:finish,
 3:put_on_shoe(right) satisfies shoe_on(right) for 1:finish,
 4:put_on_sock(left) satisfies sock_on(left) for 1:finish,
 5:put_on_sock(right) satisfies sock_on(right) for 1:finish}
```

The second important notion is that of a **threat**. A threat is a relation between a *step* and a *causal link*. If step  $T$  **threatens** the link  $S_1 \overset{G}{\square} S_2$ , it means that  $T$  could clobber  $G$ , so the planner must **resolve** the threat by ensuring either that  $T$  **precedes**  $S_1$  or that  $S_2$  **precedes**  $T$ , which is why a causal link is also called a **protected interval**. If it can't achieve either of these conditions, then it must backtrack.

A partial order plan therefore consists of three elements: a set of *steps*, a set of *edges*, and a set of *causal links*.

#### 8.4.1. A Prolog Partial-Order Planner

Here is a basic partial-order planner. For uniformity, the planner is invoked with a skeleton plan consisting of a *start* step, whose post-condition is the initial state, and a *finish* step, whose precondition is the set of goals:

```
:- op(500, xfy, [satisfies, for, precedes, threatens]).
:- op(450, xfy, [:]).
:- include(sets).

plan :- partial_order(plan({0:start precedes 1:finish}, {}, {0:start, 1:finish}), Plan),
        print(Plan).
```

On each iteration, `partial_order/2` backtracks through the *open conditions*: all terms of the form `UnmetGoal` for `NeedyStep`, where there is currently no causal link that satisfies `UnmetGoal` for `NeedyStep`. When the set of open conditions becomes empty, the plan is complete.

```
partial_order(Plan, Plan) :-
    \+open_condition(Plan, _), !.

partial_order(Plan0, Plan) :-
    open_condition(Plan0, Purpose),
    ( use_existing_step(Plan0, Step1, Purpose, Plan1)
      ; add_new_step(Plan0, Step1, Purpose, Plan1)
    ),
    find_conflicts(Plan1, Conflicts),
    resolve_conflicts(Conflicts, Plan1, Plan2),
    print(Plan2), nl,
    partial_order(Plan2, Plan).

portray(plan(Order, Links, Steps)) :-
    nl, print(Steps), nl,
    print(Links), nl,
    print(Order), nl.

open_condition(plan(_, Links, Steps), Goal for Step) :-
    all(Step, Steps),
    Step=:Operator,
    precondition(Operator, Condition),
    all(Goal, Condition),
    \in(_ satisfies Goal for Step, Links).
```

If the set of open conditions is not empty, `open_condition` chooses an arbitrary open condition and tries to satisfy it by adding a new causal link. Other things being equal, the planner will use an existing step if it can. This tends to minimise the length of the plan.<sup>60</sup>

<sup>59</sup> We need to use the `op/3` predicate to define `precedes`, etc., as operators.

<sup>60</sup> Bear in mind that although a new step is introduced to achieve a particular precondition for a particular step, it may accidentally satisfy one or more preconditions of other steps.

To use an *existing* step that satisfies the unsatisfied goal, a causal link is added to the plan, and an edge is added to the ordering. It is then necessary to check that the new causal link is not threatened by an existing step, or, if it is threatened, that the threat can be resolved.

```
use_existing_step(plan(Order0, Links0, Steps), ExistingStep, UnmetGoal for NeedyStep,
                  plan(Order1, Links1, Steps)) :-
    all(ExistingStep, Steps),
    ExistingStep=:Operator,
    positive_effect(Operator, Effects),
    all(UnmetGoal, Effects),
    union({ExistingStep satisfies UnmetGoal for NeedyStep}, Links0, Links1),
    union({ExistingStep precedes NeedyStep}, Order0, Order1).
```

When no existing step can be used, a *new* one must be created. This is done by choosing a suitable operator, and binding it to the unsatisfied goal. The new step is inserted into the set of steps, then treated in the same way as an existing step would be.

```
add_new_step(plan(Order0, Links0, Steps0), NewStep, UnmetGoal for NeedyStep,
              plan(Order1, Links1, Steps1)) :-
    positive_effect(Operator, Effects),
    all(UnmetGoal, Effects),
    size(Steps0, Count),
    NewStep=Count:Operator,
    union({NewStep}, Steps0, Steps1),
    union({NewStep satisfies UnmetGoal for NeedyStep}, Links0, Links1),
    union({NewStep precedes NeedyStep}, Order0, Order1).
```

Once the planner has added a new causal link, it must check to see if any link threatens any other:

```
find_conflicts(Plan, Conflicts) :-
    findset(Threat threatens Step1 satisfies SubGoal for Step2,
            (potential_conflicts(Plan, Threat, Step1, SubGoal, Step2),
             actually_threatens(Threat, SubGoal)),
            Conflicts).
```

This means finding every pairing of links and steps, except that a step cannot threaten itself, nor can a step threaten a link that completely precedes it or completely follows it:

```
potential_conflicts(plan(Order, Links, Steps), Threat, Step1, SubGoal, Step2) :-
    all(Threat, Steps),
    all(Step1 satisfies SubGoal for Step2, Links),
    Threat\=Step2,
    \+safely_ordered(Threat, Step1, Step2, Order).

safely_ordered(Threat, Step1, _, Order) :- in(Threat precedes Step1, Order).
safely_ordered(Threat, _, Step2, Order) :- in(Step2 precedes Threat, Order).
```

A potential conflict only matters if the threat clobbers the goal that the link satisfies:

```
actually_threatens(Threat, SubGoal) :-
    Threat=:Operator,
    negative_effect(Operator, Clobbers),
    in(SubGoal, Clobbers).
```

How are threats resolved by `resolve_conflicts`? `least/3` yields the three elements of a causal link, paired with each step (`Threat`), in turn. `Threat` is ordered either before or after the link. In either case, `resolve_conflicts` will then check that the new plan is acyclic. The resolution process is repeated until no more threats remain.

```
resolve_conflicts(Conflicts0, plan(Order0, Links, Steps), Plan) :-
    least(Conflicts0, Threat threatens Step1 satisfies _ for Step2, Conflicts1),
    ( Step2\=:finish, union({Step2 precedes Threat}, Order0, Order1)
    ; Step1\=:start, union({Threat precedes Step1}, Order0, Order1)
    ),
    Plan1=plan(Order1, Links, Steps),
    \+cyclic(Plan1),
    resolve_conflicts(Conflicts1, Plan1, Plan).

resolve_conflicts(Conflicts, Plan, Plan) :- empty(Conflicts), \+cyclic(Plan).
```

To check that a plan is consistent, the planner checks that the ordering is acyclic. This is done here by making depth-first searches starting from each step in turn. If a step proves to be its own predecessor, then it is a member of a cycle.

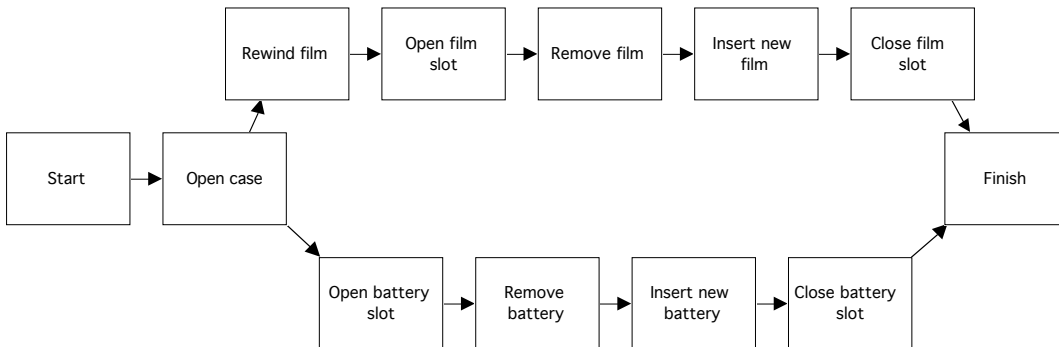
```
cyclic(plan(Order, _, Steps)) :- all(Step, Steps), cycle_present(Step, {}, Order), !.
```

Planning

```

cycle_present(From, Visited, _) :- in(From, Visited), !.
cycle_present(From, Visited0, Order) :-
    union({From}, Visited0, Visited1),
    all(From precedes Via, Order),
    cycle_present(Via, Visited1, Order).
    
```

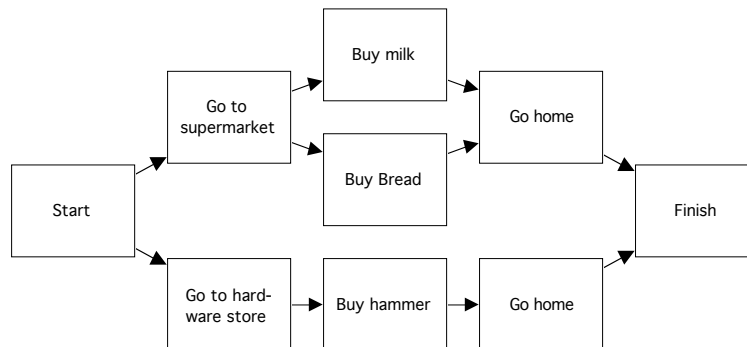
Loading a camera is another problem that has a partially ordered solution:



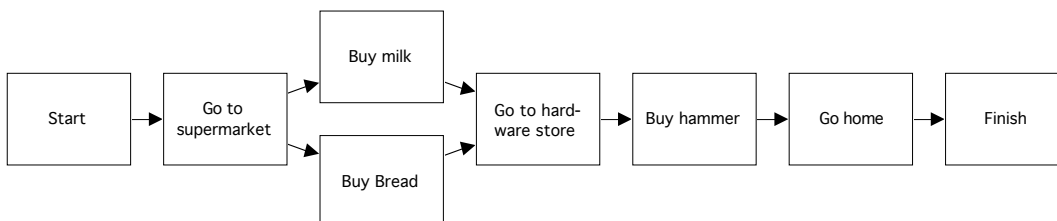
Unfortunately, the non-linear planner we have described is no better than the simple means-end planner; it will loop, taking the camera in and out of its case. This is surprising, because the planner checks that its plans are acyclic. However, since steps are serial numbered, each time it takes the camera out of its case, that is a different step. In the linear planners we checked that the planner did not pass through a previous *state*. This is a sound idea, but it is not easy to define the state of a partially ordered plan. A partially ordered plan represents a *set* of sequences, and each such sequence passes through a *different* sequence of states.

It would be improved by using a better search strategy, such as branch-and-bound. A useful heuristic is to prefer goals that cannot be satisfied by any existing step. This makes the planner decide what steps are needed before completing the fine details. It also prevents conditions that are common to the *start* and *finish* states causing wasteful backtracking. For example, in the camera problem, the set {*slot\_closed(film), in(film), slot\_closed(battery), in(battery)*} is common to both states. All of these conditions can create causal links that have to be undone before a solution can be found.<sup>61</sup> Many improvements are needed to make the above algorithm effective.

In the two previous examples, the partial ordering contained the seeds of every possible linear plan, but this is not always the case. Suppose you have to go and buy a carton of milk and a loaf of bread from the supermarket, and a hammer from the hardware store. Although you could go to the supermarket first or go to the hardware store first, these actions have to be ordered in the plan. The following is *not* a possible plan,



Although the supermarket trip and the hardware trip are independent, they cannot be interleaved. You can't be in two places at once. The above plan would allow 'go to supermarket, go to hardware store, buy milk, buy bread, buy hammer, go home, go home'. Even a partial-order planner has to choose which store to visit first:



<sup>61</sup> Unfortunately, iterative deepening doesn't work at all well here, defeating any heuristics we might use to speed the search. 12 steps are needed to solve the camera problem. Iterative deepening would check every plan of 11 steps before finding any 12-step solution.

## 8.5. A Planner Buster

You are almost certainly familiar with the puzzle usually called ‘The Towers of Hanoi’. It comprises a base on which are mounted three vertical pegs. There are  $N$  discs of unequal sizes with central holes, which allow them to slip onto the pegs. The puzzle has the rule that each disc can only sit on top of a larger one. Initially all the discs are on Peg 1. The problem is to move them to be on Peg 2, using Peg 3 as a work space, without violating the rule.

There is a neat recursive solution to this puzzle, and a less well known but equally neat iterative one. But none of the planners we have described can possibly discover an algorithm. The *best* they can do is find a sequence of moves by trial and error. They will never discover an underlying pattern.

The problem (for 3 discs) can be described easily enough:

```
precondition(start, {}).
positive_effect(start, {clear(a), on(a, b), on(b, c), on(c, 1), clear(2), clear(3)}).
negative_effect(start, {}).

precondition(finish, {on(a, b), on(b, c), on(c, 2)}).
positive_effect(finish, {}).
negative_effect(finish, {}).

precondition(move(Disc, From, To), [clear(Disc), clear(To), on(Disc, From)]) :-
    disc(Disc),
    object(To), object(From),
    smaller(Disc, To), smaller(Disc, From),
    From\==To, From\==Disc.
positive_effect(move(Disc, From, To), [on(Disc, To), clear(From)]).
negative_effect(move(Disc, From, To), [on(Disc, From), clear(To)]).
object(X) :- disc(X);place(X).
disc(a). disc(b). disc(c).
place(1). place(2). place(3).
smaller(a, b). smaller(a, c). smaller(b, c).
smaller(a, 1). smaller(a, 2). smaller(a, 3).
smaller(b, 1). smaller(b, 2). smaller(b, 3).
smaller(c, 1). smaller(c, 2). smaller(c, 3).
```

Here, there are three pegs, 1, 2, and 3, and three discs,  $a$ ,  $b$ , and  $c$ , of progressively increasing size. Initially, Disc  $a$  is on Disc  $b$ , Disc  $b$  is on Disc  $c$ , and Disc  $c$  is on Peg 1. There is nothing on top of Disc  $a$ ; it is ‘clear’. The goal is to move the discs to Peg 2.

This problem is solved quite quickly by the iterative-deepening means-ends planner, resulting in a sequence of 7 steps. The means-ends planner works well because the only goal that is not true in the initial state happens to be  $\text{on}(c, 2)$ , which also happens to be the ideal step to begin a divide and conquer strategy. The regression planner fares less well, because it is not at all clear what the last move should be. The only two legal moves involve Disc  $a$ , but even that isn’t obvious to the planner.

Unfortunately, in both cases, each level of iterative deepening proves to take roughly 6 times longer than the previous one, suggesting that the branching factor of the search averages about 6. A solution with 4 discs needs 8 more steps, so it would take  $6^8$  (over 1.5 million) times longer than the solution for 3 discs. Both planners are overwhelmed by the doubly exponential growth in the length of the plan.

This should not be seen as a criticism of partial-order planners in general, but the planner we described is no use at all. First, the solution is a *total* ordering. Second, the planner quickly gets into a loop.

## 9. Reasoning under Uncertainty

### 9.1. Ignorance and Uncertainty

An agent rarely has complete knowledge about its environment. For example, the wumpus agent might perceive a breeze, but it might not know *which* neighbouring cells contain pits, only that at least one of them does. The agent must therefore sometimes act under **uncertainty**. It is important to be able to measure uncertainty. If the agent has a choice of several dangerous cells that it can explore, it is best to choose the least dangerous.

The **qualification problem** captures the confusion of a punter at a horse race: If only he knew enough about the breeding of the horses, their state of health, the state of the track, the jockey, and so on, he could forecast the result of a race exactly. Of course, he can never know all these things, especially the bit called ‘and so on’, which includes whether the horse box will have an accident on the way to the track, and an infinity of external factors.

Unfortunately, if we want to use first-order logic, we can only work with deterministic rules. This can be impractical for three reasons: **laziness**, meaning that the cost of obtaining all the necessary rules and facts exceeds the value of having them, **theoretical ignorance**, meaning that even given all the facts, we don’t know how to use them, and **practical ignorance**, meaning that we cannot find all the facts. With such partial information, an agent can only have a **degree of belief** in logical sentences. **Probability theory** gives us a way of dealing with degrees of belief.

It is important to distinguish **degree of belief** from **degree of truth**, which is the domain of **fuzzy logic**. The statement, “I have a pet elephant.” is either true or false, but it is one to which you might attach a very low *degree of belief*. The statement, “I am a good pianist.” is *fuzzy*, not something that is either true or false. I am a much better pianist than my cat, but not nearly as good as Roger Woodward or Elton John.

Probabilities depend on **evidence**. If you pick a card from a pack, before you examine it, it has a **prior** (or **unconditional**) probability of 1/52 of being A. After you look at it, the **posterior** (or **conditional**) probability is either 0 or 1. As evidence accumulates, probabilities change.

To make rational decisions, an agent must be able to express preferences between outcomes. For example, the wumpus agent would rather grab the gold than fall down a pit. To deal with uncertainty rationally, the agent must have a measure of **utility**. In the case of the wumpus agent, utility is measured by the number of points won or lost. Grabbing the gold has a utility of +1,000, falling into a pit has a utility of –1,000. **Utility theory** gives us a way of dealing with this. Suppose the probability of a certain cell being a pit is 0.4, and that if it is not, and the agent enters it, the agent will certainly grab the gold.<sup>62</sup> Then the probability that the cell is *not* a pit is  $1 - 0.4 = 0.6$ , and the expected utility is  $1,000 \times 0.6 - 1,000 \times 0.4$ , or 200. Therefore a rational agent should decide to enter the cell. In a nutshell,

**Probability Theory+Utility Theory = Decision Theory.**

### 9.2. Probability Theory

#### 9.2.1. Prior Probability

The notation  $P(A)$  represents the **prior** or **unconditional probability** that  $A$  is true. For example, we may write  $P(Pit)=0.2$  to represent the fact that, in the absence of any other information, a cell in the wumpus game has a 1 in 5 chance of being a pit.<sup>63</sup>

Probabilities can also be assigned to **random variables**. A cell may be a pit, be the wumpus’s lair, or be safe, but only one of these. We may write,

$$P(Cell=Wumpus) = 0.0625, P(Cell=Pit) = 0.1633, P(Cell=Safe) = 0.7742$$

Alternatively, we may treat the set of probabilities as a vector, called the probability distribution of the variable.<sup>64</sup>

$$P(Cell) = [0.0625, 0.1633, 0.7742]$$

We may assign probabilities to *any kind of logical sentence*,

$$P(Wumpus \wedge Gold) = 0.004$$

#### 9.2.2. Conditional Probability

Once an agent has acquired knowledge, probabilities change. We may write,

$$P(Pit_{1,2}|Breeze_{1,1})$$

<sup>62</sup> This assumption is very optimistic; other things can still go wrong.

<sup>63</sup> This is correct only for cells *known not to contain the wumpus, the gold or the entrance*.

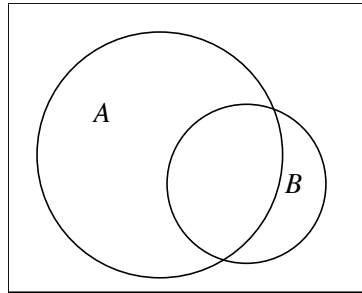
<sup>64</sup> The probabilities must sum to 1.0.

to denote the **conditional probability** that there is a pit in the cell at position (1, 2), *given* (!) that *all* the agent knows is that there is a breeze in the cell at position (1, 1).

The **product rule** is as follows,

$$P(A \cap B) = P(B) \cdot P(A|B) = P(A) \cdot P(B|A)$$

This is easily seen from the following diagram, in which the relative areas represent probabilities.



The box represents ‘true’, with probability 1.0. The *area* of circle *A* represents  $P(A)$ , the fraction of cases in which *A* is true; the *area* of circle *B* represents  $P(B)$ , the smaller fraction of cases in which *B* is true. The intersection of the circles represents  $P(A \cap B)$ , the fraction of cases in which both *A* and *B* are true. The **conditional probability** of *B* given *A*,  $P(B|A)$ , is the ratio of the area of the intersection to the area of *A*. The conditional probability of *A* given *B*,  $P(A|B)$ , is the ratio of the area of the intersection to the area of *B*. You can see that these two probabilities need not be equal; in the diagram,  $P(B|A)$  is clearly less than 0.5, and  $P(A|B)$  is clearly greater than 0.5.

### 9.2.3. The Axioms of Probability

1.  $0 \leq P(A) \leq 1$  Probabilities range from 0 to 1.
2.  $P(\text{True})=1, P(\text{False})=0$  A true proposition has probability 1, a false one, 0.
3.  $P(A \cup B)=P(A)+P(B)-P(A \cap B)$  The disjunction rule.

The third axiom is easily understood from the diagram. If we simply added  $P(A)$  and  $P(B)$ , we would count the area of intersection twice.

Setting  $B=\neg A$  in the 3rd axiom, we obtain the important theorem,

$$\begin{aligned} P(A \cup \neg A) &= P(A)+P(\neg A)-P(A \cap \neg A) \\ P(\text{True}) &= P(A)+P(\neg A)-P(\text{False}) \\ 1 &= P(A)+P(\neg A) \\ P(\neg A) &= 1-P(A) \end{aligned}$$

The following argument concerning probabilities is simplistic, and only approximate:

Consider a cell with 4 neighbours, where the wumpus agent perceives a breeze. Then one cell is certainly a pit, and each neighbour has 20% chance (say) of being a pit. Therefore 1.6 neighbours are pits, on average. Therefore the probability that any given neighbour is a pit is  $1.6 \div 4 = 0.4$ .

This is argument wrong because more than one neighbour may be a pit. We need to subtract the joint probabilities to get the correct result. On the other hand, the probability that cell  $(X, Y)$  contains either a pit or the gold is simply  $P(\text{Pit}_{X,Y} \cup \text{Gold}_{X,Y}) = P(\text{Pit}_{X,Y}) + P(\text{Gold}_{X,Y})$  because the rules of the game are such that  $P(\text{Pit}_{X,Y} \cap \text{Gold}_{X,Y}) = 0$ .

***It is only correct to sum the probabilities of events if they are mutually exclusive.***

### 9.2.4. The Joint Probability Distribution

Given a set of random variables, it would be possible, in principle, to assign a probability to every possible combination of their values. This is a multi-dimensional vector called their **joint probability distribution**, or simply, the **joint**. We may tabulate the joint for a cell in the Wumpus game as follows,

<i>Wumpus</i>	<i>Gold</i>	<i>Pit</i>	<b>Probability</b>
<i>False</i>	<i>False</i>	<i>False</i>	0.715
<i>False</i>	<i>False</i>	<i>True</i>	0.163
<i>False</i>	<i>True</i>	<i>False</i>	0.059
<i>False</i>	<i>True</i>	<i>True</i>	0.000
<i>True</i>	<i>False</i>	<i>False</i>	0.059
<i>True</i>	<i>False</i>	<i>True</i>	0.000
<i>True</i>	<i>True</i>	<i>False</i>	0.004
<i>True</i>	<i>True</i>	<i>True</i>	0.000

From the table, we may calculate any probability we require, e.g.,

$$P(\text{Gold}|\text{Wumpus}) = P(\text{Gold} \cap \text{Wumpus}) \div P(\text{Wumpus}) = (0.004+0.0) \div (0.059+0.004) = 0.063.$$

$$P(\text{Gold}) = (0.059+0.0+0.004+0.0) = 0.063.$$

$$P(\text{Wumpus}|\text{Gold}) = P(\text{Gold} \cap \text{Wumpus}) \div P(\text{Gold}) = (0.004+0.0) \div (0.059+0.004) = 0.063.$$

$$P(\text{Wumpus}) = (0.059+0.0+0.004+0.0) = 0.063.$$

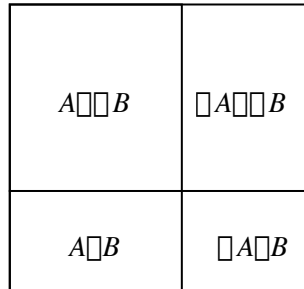
Although the joint is a useful concept, probabilistic reasoning systems rarely use it directly.

Here,  $P(\text{Gold}|\text{Wumpus})= P(\text{Gold})$ , and  $P(\text{Wumpus}|\text{Gold}) = P(\text{Wumpus})$ , so we say that *Gold* and *Wumpus* are **statistically independent**. That is to say, our degree of belief in *Wumpus* is not affected by our degree of belief in *Gold*, or vice versa. We obtain the following,

$$P(\text{Gold} \cap \text{Wumpus}) = P(\text{Gold}|\text{Wumpus}) \cap P(\text{Wumpus}) = P(\text{Gold}) \cap P(\text{Wumpus})$$

**It is only correct to multiply the probabilities of events if they are independent.**

We can see this in the following diagram,



The horizontal,  $P(A)$ , and vertical,  $P(B)$ , axes are orthogonal. The left-hand part of the square represents  $P(A)$ , the proportion of cases when *A* is true, the lower part of the square represents  $P(B)$ , the proportion of cases when *B* is true. The rectangle in the lower left-hand corner represents  $P(A \cap B)$ , the proportion of cases when both *A* and *B* are true.  $P(A|B)$ , the probability of *A* given *B*, is the ratio of the  $P(A \cap B)$  rectangle to the  $P(B)$  rectangle. This is the same as the ratio of the  $P(A)$  rectangle to the whole square.

We return to the question of how many neighbours of a cell in the Wumpus game are pits. Recall that (ignoring the gold, wumpus, and the entrance), the probability that a cell is a pit is  $1/5$ , and the probability that it is *not* is  $4/5$ . The probability that each cell contains a pit is the same. Each cell is *independent* of the rest.<sup>65</sup> Consider, again, a breezy cell with 4 neighbours.

- The probability that exactly 1 *given* neighbour is a pit and the other 3 are not pits is  $1/5 \cap (4/5)^3$ . But there are 4 possible choices of which neighbour is the pit, so the chance that exactly 1 neighbour is a pit is  $4 \cap (1/5) \cap (4/5)^3 = 256/625$ .
- There are 6 ways that 2 neighbours can be pits (N-S, E-W, N-E, E-S, S-W, and W-N) so the probability that exactly 2 neighbours are pits is  $6 \cap (1/5)^2 \cap (4/5)^2 = 96/625$ .
- There are 4 ways that 3 neighbours can be pits (there are 4 choices of the one that isn't), so the probability is  $4 \cap (1/5)^3 \cap 4/5 = 16/625$ .
- Finally, there is only 1 way that all 4 can be pits, so the probability is  $(1/5)^4 = 1/625$ .<sup>66</sup>

So the probability of at least one neighbour being a pit is  $(256+96+16+1)/625 = 369/625$ . This is also the probability that the cell has a breeze, so we may deduce that  $P(\text{Breeze}) = 369/625$ . This is, of course, 1 minus the probability that no neighbours are pits, i.e.,  $(4/5)^4 = 256/625$ .

Returning to the question of the average number of pits surrounding a cell, we need a weighted average,  $1 \cap 256/625 + 2 \cap 96/625 + 3 \cap 16/625 + 4 \cap 1/625 = 500/625$ . If we know the cell has a breeze, then we should divide this by  $369/625$ , giving  $500/369 = 1.36$ —considerably fewer than the 1.6 estimated by adding probabilities. The chance that any *particular* neighbour is a pit is  $500/369 \div 4 = 125/369 = 0.34$ .

### 9.2.5. Marginalisation and Conditioning

If a set *Z* of events *z* is mutually *exhaustive*, then at least one *z* must be true. If the events are also mutually *exclusive*, we can sum their individual probabilities, therefore,  $\sum_z P(z) = 1$ .

Likewise,  $P(Y) = \sum_z P(Y \cap z)$ . This is called **summing out** or **marginalisation**.

Similarly,  $P(Y) = \sum_z P(Y|z)P(z)$ . This is called **conditioning**.

<sup>65</sup> This is not entirely true, because the program guarantees there is a route to the gold.

<sup>66</sup> You will recognise these probabilities as the successive terms in the binomial expansion of  $(1/5+4/5)^4$ , which sum to  $1^4=1$ .

These principles are illustrated by the previous example. The cases that 0, 1, 2, 3, or 4 neighbours are pits are mutually exhaustive and exclusive. Therefore, their probabilities sum to 1. Thus  $P(\text{Pits} > 0) = 1 - P(\text{Pits} = 0)$ . Similarly,  $P(\text{Breeze}) = \sum_{0 \leq N \leq 4} P(\text{Breeze} | \text{Pits} = N) P(N)$ , where  $P(\text{Breeze} | \text{Pits} = 0) = 0$  and  $P(\text{Breeze} | \text{Pits} = N) = 1$  for  $N > 0$ .

### 9.2.6. Bayes' Rule

From the product rule,  $P(A \wedge B) = P(B) P(A|B) = P(A) P(B|A)$ , we may derive,

$$P(A|B) = P(A) P(B|A) \div P(B)$$

This is called **Bayes' rule**, or **Bayes' theorem**. It is useful because we might not know the full joint, but we might know  $P(A)$ ,  $P(B)$  and  $P(B|A)$ . For example, in the Wumpus game, we might want to know the probability that cell (1, 1) is a pit, given that there is a breeze in cell (1, 2). We therefore have,

$$P(\text{Pit}_{1,1} | \text{Breeze}_{1,2}) = P(\text{Pit}_{1,1}) P(\text{Breeze}_{1,2} | \text{Pit}_{1,1}) \div P(\text{Breeze}_{1,2})$$

Because a pit always causes a breeze, we know that  $P(\text{Breeze}_{1,2} | \text{Pit}_{1,1}) = 1$ , leaving us with the problem of finding only  $P(\text{Pit}_{1,1})$  and  $P(\text{Breeze}_{1,2})$ . To simplify the discussion, let us assume the maze is infinite. This means we may ignore the possibility that a cell is out of bounds, contains the wumpus, or contains the gold. In other words, every cell has exactly 4 neighbours, and every cell has exactly  $1/5$  chance of being a pit.

What is the chance that a cell contains a breeze? Let the neighbours of cell  $C$  be  $N$ ,  $S$ ,  $E$  and  $W$ . Then we have,

$$\begin{aligned} P(\text{Breeze}_C) &= P(\text{Pit}_N \vee \text{Pit}_S \vee \text{Pit}_E \vee \text{Pit}_W) \\ &= 1 - P(\neg(\text{Pit}_N \vee \text{Pit}_S \vee \text{Pit}_E \vee \text{Pit}_W)) \\ &= 1 - P(\neg \text{Pit}_N \wedge \neg \text{Pit}_S \wedge \neg \text{Pit}_E \wedge \neg \text{Pit}_W) \\ &= 1 - P(\neg \text{Pit}_N) P(\neg \text{Pit}_S) P(\neg \text{Pit}_E) P(\neg \text{Pit}_W) \\ &= 1 - \left(\frac{4}{5}\right)^4 \\ &= \frac{369}{625} \end{aligned}$$

We therefore have,

$$\begin{aligned} P(\text{Pit}_{1,1} | \text{Breeze}_{1,2}) &= P(\text{Pit}_{1,1}) P(\text{Breeze}_{1,2} | \text{Pit}_{1,1}) \div P(\text{Breeze}_{1,2}) \\ &= \frac{1}{5} \frac{1 - \frac{369}{625}}{\frac{369}{625}} \\ &= \frac{125}{369} = 0.34 \end{aligned}$$

This gives the same result as the previous calculation, but without the pain.

Another reason why Bayes' rule is helpful is that  $P(\text{Breeze} | \text{Pit}) = 1$  is robust knowledge but  $P(\text{Pit} | \text{Breeze}) = 0.34$  is not. If  $P(\text{Pit})$  were changed we could re-compute  $P(\text{Pit} | \text{Breeze})$ .

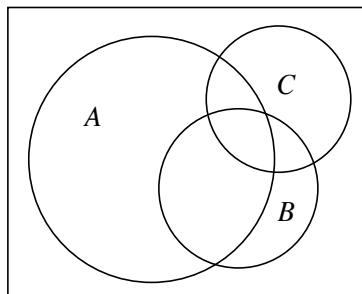
### 9.2.7. Conditioning Contexts

What is true for the earlier diagram as a whole, is also true for the part of the diagram enclosed in circle  $C$  below. The product rule for the part of the diagram within circle  $C$  is

$$P(A \wedge B | C) = P(B | C) P(A | B \wedge C) = P(A | C) P(B | A \wedge C),$$

From this we derive a conditional form of Bayes' rule,

$$P(A | B \wedge C) = P(A | C) P(B | A \wedge C) \div P(B | C)$$



In effect, this means that if  $C$  is given, we can ignore it.  $C$  is called a **conditioning context**.

### 9.2.8. Conditional Independence

We all know that if you smoke you are more likely to get lung cancer and high blood pressure. Because of this, lung cancer and high blood pressure tend to go together. They are both more likely in smokers than in non-smokers. However, although smoking tends to cause lung cancer and smoking tends to cause high blood pressure, we have no reason to believe that high blood pressure causes lung cancer or that lung cancer causes high blood pressure. The joint probability distribution might look like this,<sup>67</sup>

<sup>67</sup> Don't be scared, I made the numbers up.

<i>Smoker</i>	<i>Lung Cancer</i>	<i>High Blood Pressure</i>	<i>Probability</i>
<i>False</i>	<i>False</i>	<i>False</i>	0.36
<i>False</i>	<i>False</i>	<i>True</i>	0.09
<i>False</i>	<i>True</i>	<i>False</i>	0.04
<i>False</i>	<i>True</i>	<i>True</i>	0.01
<i>True</i>	<i>False</i>	<i>False</i>	0.20
<i>True</i>	<i>False</i>	<i>True</i>	0.20
<i>True</i>	<i>True</i>	<i>False</i>	0.05
<i>True</i>	<i>True</i>	<i>True</i>	0.05

From this, we can calculate that,

$$P(\text{LungCancer}|\text{Smoker}) = P(\text{LungCancer} \wedge \text{Smoker}) \div P(\text{Smoker}) \\ = (0.05+0.05) \div (0.2+0.2+0.05+0.05) = 0.2$$

whereas,

$$P(\text{LungCancer} \neg \text{Smoker}) = P(\text{LungCancer} \wedge \neg \text{Smoker}) \div P(\neg \text{Smoker}) \\ = (0.04+0.01) \div (0.36+0.09+0.04+0.01) = 0.1$$

Now,

$$P(\text{LungCancer}|\text{Smoker} \wedge \text{HighBloodPressure}) \\ = P(\text{LungCancer} \wedge \text{Smoker} \wedge \text{HighBloodPressure}) \div P(\text{Smoker} \wedge \text{HighBloodPressure}) \\ = 0.05 \div (0.2+0.05) = 0.2$$

Now,

$$P(\text{LungCancer}|\text{Smoker} \wedge \neg \text{HighBloodPressure}) \\ = P(\text{LungCancer} \wedge \text{Smoker} \wedge \neg \text{HighBloodPressure}) \div P(\text{Smoker} \wedge \neg \text{HighBloodPressure}) \\ = 0.05 \div (0.2+0.05) = 0.2$$

So, although the probability of a person having lung cancer depends on whether he or she is a smoker, the probability of a *smoker* having lung cancer does *not* depend on whether or not he or she has high blood pressure. *LungCancer* and *HighBloodPressure* are statistically independent, *given* that *Smoker* is true. They are said to be **conditionally independent given *Smoker***. This behaviour arises because, although they have a common cause, they don't cause each other.

A similar calculation shows that,

$$P(\text{HighBloodPressure}|\text{Smoker}) \\ = P(\text{HighBloodPressure} \wedge \text{Smoker}) \div P(\text{Smoker}) \\ = (0.2+0.05) \div (0.2+0.2+0.05+0.05) \\ = 0.5$$

so we should not be surprised that we can multiply conditionally independent probabilities, as follows,

$$P(\text{LungCancer} \wedge \text{HighBloodPressure}|\text{Smoker}) \\ = P(\text{LungCancer} \wedge \text{HighBloodPressure} \wedge \text{Smoker}) \div P(\text{Smoker}) \\ = 0.05 \div (0.2+0.2+0.05+0.05) \\ = 0.1 \\ = P(\text{LungCancer}|\text{Smoker}) \square P(\text{HighBloodPressure}|\text{Smoker})$$

because *Smoker* is a *conditioning context* here.

### 9.2.9. Normalisation

Sometimes we don't need to know all the probabilities involved in a problem, because we can eliminate some of them algebraically. Suppose that our research has shown that,

$$P(\text{Smoker}|\text{LungCancer})=0.67 \text{ and } P(\text{Smoker}|\text{HighBloodPressure})=0.71.$$

Suppose that we want to transfer our results to a community where we observe that  $P(\text{LungCancer})=0.2$  and  $P(\text{HighBloodPressure})=0.4$ , but we don't know the value of  $P(\text{Smoker})$ . We can't estimate the absolute values of  $P(\text{LungCancer}|\text{Smoker})$  or  $P(\text{HighBloodPressure}|\text{Smoker})$  for this population, but we can estimate their **relative likelihood**.

$$P(\text{LungCancer}|\text{Smoker}) \div P(\text{HighBloodPressure}|\text{Smoker}) \\ = (P(\text{Smoker}|\text{LungCancer}) \square P(\text{LungCancer}) \div P(\text{Smoker})) \\ \div (P(\text{Smoker}|\text{HighBloodPressure}) \square P(\text{HighBloodPressure}) \div P(\text{Smoker}))$$

$$P(\text{Smoker}) \text{ cancels, so the relative likelihood is } (0.67 \square 0.2) \div (0.71 \square 0.4) = 0.47.$$

We can sometimes use this idea to estimate the relative likelihood of both  $A$  and  $\neg A$ . Suppose  $P(A) \div P(\neg A) = 3.0$ . But we know that  $P(A) + P(\neg A) = 1$ . Therefore  $P(A) = 0.75$  and  $P(\neg A) = 0.25$ . This process of obtaining absolute values from ratios is called **normalisation**.

### 9.2.10. Bayesian Updating

Suppose we know the following probabilities,

$$P(\text{Smoker}) = 0.5$$

$$P(\text{Smoker}|\text{HighBloodPressure}) = 0.71$$

$$P(\text{Smoker}|\text{LungCancer}) = 0.67$$

We are asked to estimate the probability that a person smokes, but are given no information about their lungs or blood pressure. We would have to use  $P(\text{Smoker})=0.5$ .

Suppose that we then learn that they have high blood pressure. We would then use,

$$P(\text{Smoker}|\text{HighBloodPressure}) \\ = P(\text{Smoker}) \square P(\text{HighBloodPressure}|\text{Smoker}) \div P(\text{HighBloodPressure})$$

Finally, we learn they have lung cancer. Using *HighBloodPressure* as a *conditioning context*,

$$P(\text{Smoker}|\text{HighBloodPressure}\square\text{LungCancer}) \\ = P(\text{Smoker}|\text{HighBloodPressure}) \\ \square P(\text{LungCancer}|\text{HighBloodPressure}\square\text{Smoker}) \div P(\text{LungCancer}|\text{HighBloodPressure})$$

As each piece of evidence was obtained, we multiplied the previous estimate by a ratio based on the new evidence. This is called **Bayesian updating**.

As things stand, we have not gained much because the terms in the ratio are unknown. But remember that lung cancer and high blood pressure are *conditionally independent*. Therefore,

$$P(\text{LungCancer}|\text{HighBloodPressure}\square\text{Smoker}) = P(\text{LungCancer}|\text{Smoker})$$

So we have,

$$P(\text{Smoker}|\text{HighBloodPressure}\square\text{LungCancer}) \\ = P(\text{Smoker}|\text{HighBloodPressure}) \\ \square P(\text{LungCancer}|\text{Smoker}) \div P(\text{LungCancer}|\text{HighBloodPressure}) \\ = P(\text{Smoker}) \square P(\text{HighBloodPressure}|\text{Smoker}) \div P(\text{HighBloodPressure}) \\ \square P(\text{LungCancer}|\text{Smoker}) \div P(\text{LungCancer}|\text{HighBloodPressure}) \\ = P(\text{Smoker}) \square P(\text{HighBloodPressure}|\text{Smoker}) \square P(\text{LungCancer}|\text{Smoker}) \\ \div (P(\text{HighBloodPressure}) \square P(\text{LungCancer}|\text{HighBloodPressure})) \\ = P(\text{Smoker}) \square P(\text{HighBloodPressure}|\text{Smoker}) \square P(\text{LungCancer}|\text{Smoker}) \\ \div P(\text{LungCancer}\square\text{HighBloodPressure})$$

We still may not know the term  $P(\text{LungCancer}\square\text{HighBloodPressure})$ , but we can eliminate it by normalisation, if we also know  $P(\text{HighBloodPressure}\square\text{Smoker})$  and  $P(\text{LungCancer}\square\text{Smoker})$ . This means that we only need to know  $P(\text{Smoker})$  and the conditional probabilities that depend on *Smoker*. This simplification is possible because *LungCancer* and *HighBloodPressure* are *conditionally independent* on *Smoker*, which arises from *cause and effect*.

### 9.2.11. The Monty Hall Paradox

Suppose you are a contestant on Monty Hall's (American) TV quiz-show. You have answered all the questions correctly and now have a chance of a major prize. The prize is concealed behind one of three doors, *A*, *B* and *C*. Suppose you choose *Door A*. Monty then opens *Door B*, to reveal that the prize is *not* behind it. He offers you a chance to choose again. Should you stick with your original hunch, or switch to *Door C*?

The reason that this is called the Monty Hall *Paradox* is that it has caused much heated debate among professional mathematicians.

One view is that you may as well stick with your original choice. Whichever door you chose, Monty would have been able to open at least one door that didn't open onto the prize, so you have learned nothing new. Alternatively, there are now only two doors, *A* and *C*, to choose from, so the odds are evens.

The other view is that your original choice had only one chance in three of being right, and nothing Monty has done has improved it, so it would be better to switch. Alternatively, there were originally two chances in three of the prize being behind *Door B* or *Door C*, and Monty's opening *Door B* has collapsed these two alternatives into one; the prize is twice as likely to be behind *Door C* than *Door A*.

What does the Reverend Bayes have to say? Let us compute  $P(\text{PrizeC}|\text{MontyB})$  in the *conditioning context* of *ChooseA*. From Bayes' Rule this is equal to,

$$P(\text{MontyB}|\text{PrizeC}) \square P(\text{PrizeC}) \div P(\text{MontyB})$$

Since Monty has only one option if you choose *Door A* and the prize is behind *Door C*, the first term is 1. We may assume that  $P(\text{PrizeC})=1/3$ , but we don't really know, and we have no idea of  $P(\text{MontyB})$ . Maybe Monty always chooses *Door B* whenever he possibly can, and maybe he only chooses it when he is forced to. Let us try *normalisation*. We know that *PrizeB* is false, so we only need to find  $P(\text{PrizeA}|\text{MontyB})$ .

$$P(\text{MontyB}|\text{PrizeA}) \square P(\text{PrizeA}) \div P(\text{MontyB})$$

From this, it is easy to find the ratio,

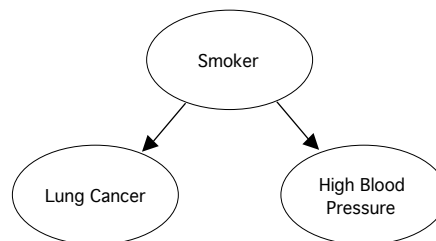
$$P(\text{PrizeA}|\text{MontyB}) \div P(\text{PrizeC}|\text{MontyB}) = P(\text{MontyB}|\text{PrizeA}).$$

We know this term, and therefore the ratio, is between 0 and 1, so it is *never* better to choose *Door A* over *Door C*.

The surprising thing about this result is that *your odds of winning the prize depend on what Monty does if you guess correctly*. If we assume that, given that the prize is behind *Door A*, Monty chooses evenly between *Door B* or *Door C*, then  $P(\text{MontyB}|\text{PrizeA})=0.5$ , so  $P(\text{PrizeC})=2 \times P(\text{PrizeA})$ . If Monty chooses *Door B* whenever possible,  $P(\text{PrizeA})=P(\text{PrizeC})$ . If Monty only chooses *Door B* when forced to, then *PrizeC* is a certainty.

### 9.3. Belief Networks

The use of probability theory in reasoning was considered impractical for a long time, mainly because in most practical situations it is impossible to establish all  $2^N$  terms in the joint probability distribution. However, *conditional independence greatly reduces the number of terms that have to be known*. Conditional independence arises from cause and effect. We have reason to believe that smoking is a cause of both lung cancer and high blood pressure. We don't believe that lung cancer or high blood pressure cause smoking, or that high blood pressure causes lung cancer, or *vice versa*. We can show these relationships in the following diagram, which is called a **belief network** or **causal network**.



The advantage of this model is that if we know  $P(\text{Smoker})$ ,  $P(\text{LungCancer}|\text{Smoker})$ ,  $P(\text{LungCancer}|\neg\text{Smoker})$ ,  $P(\text{HighBloodPressure}|\text{Smoker})$ , and  $P(\text{HighBloodPressure}|\neg\text{Smoker})$ , all the other probabilities in the joint distribution can be derived. To demonstrate this, suppose,

$$\begin{aligned} P(\text{Smoker}) &= 0.5, \\ P(\text{LungCancer}|\text{Smoker}) &= 0.2, \\ P(\text{LungCancer}|\neg\text{Smoker}) &= 0.1, \\ P(\text{HighBloodPressure}|\text{Smoker}) &= 0.5, \text{ and} \\ P(\text{HighBloodPressure}|\neg\text{Smoker}) &= 0.2. \end{aligned}$$

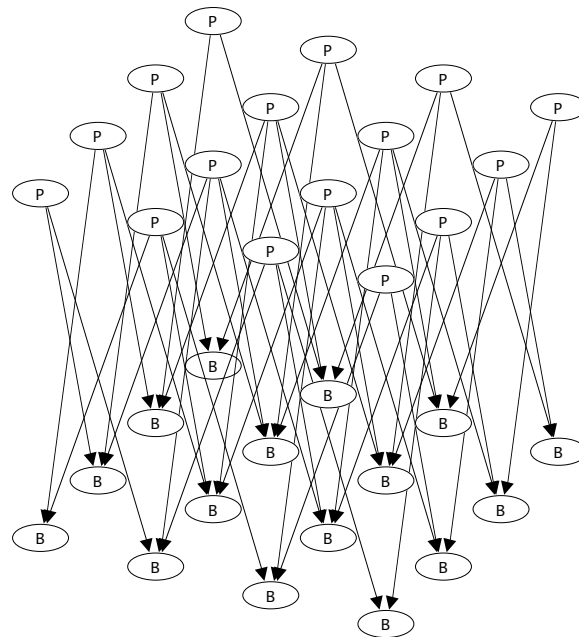
Then, for example, because *LungCancer* and *HighBloodPressure* are conditionally independent given *Smoker* (as the network shows),

$$\begin{aligned} P(\text{Smoker} \wedge \text{LungCancer} \wedge \text{HighBloodPressure}) \\ &= P(\text{Smoker}) \times P(\text{LungCancer}|\text{Smoker}) \times P(\text{HighBloodPressure}|\text{Smoker}) \\ &= 0.5 \times 0.2 \times 0.5 = 0.05. \end{aligned}$$

Similarly,

$$\begin{aligned} P(\neg\text{Smoker} \wedge \neg\text{LungCancer} \wedge \neg\text{HighBloodPressure}) \\ &= P(\neg\text{Smoker}) \times P(\neg\text{LungCancer}|\neg\text{Smoker}) \times P(\neg\text{HighBloodPressure}|\neg\text{Smoker}) \\ &= (1-0.5) \times (1-0.1) \times (1-0.2) \\ &= 0.5 \times 0.9 \times 0.8 = 0.36.^{68} \end{aligned}$$

<sup>68</sup> Compare these results with the table of the joint in Section 2.7.



Above is a sketch of *part* of the belief network for the Wumpus game. The upper grid represents each of the 16 propositions that a cell is a pit; the lower grid represents the 16 propositions that a cell has a breeze. (It omits to show anything about the gold, entrance or wumpus.) Although a pit always causes a breeze, a breeze has between 2 and 4 parents, so a breeze does not uniquely identify a particular pit.<sup>69</sup>

To deal with this situation properly, we cannot simply write  $P(\text{Breeze}|\text{Pit})$ .  $\text{Breeze}_{1,1}$ , for example, has parents  $\text{Pit}_{0,1}$ ,  $\text{Pit}_{1,0}$ ,  $\text{Pit}_{2,1}$  and  $\text{Pit}_{1,2}$ , so we need all 16 conditional probabilities of the form  $P(\text{Breeze}_{1,1}|\text{Pit}_{0,1}\ \square\ \text{Pit}_{1,0}\ \square\ \text{Pit}_{2,1}\ \square\ \text{Pit}_{1,2})$ ,  $P(\text{Breeze}_{1,1}|\text{Pit}_{0,1}\ \square\ \text{Pit}_{1,0}\ \square\ \text{Pit}_{2,1}\ \square\ \square\ \text{Pit}_{1,2})$ , etc. In this particular problem, all the terms equal 1.0 except  $P(\text{Breeze}_{1,1}\ \square\ \text{Pit}_{0,1}\ \square\ \square\ \text{Pit}_{1,0}\ \square\ \square\ \text{Pit}_{2,1}\ \square\ \square\ \text{Pit}_{1,2})$ , which is zero. In other problem domains, things might not be so simple.

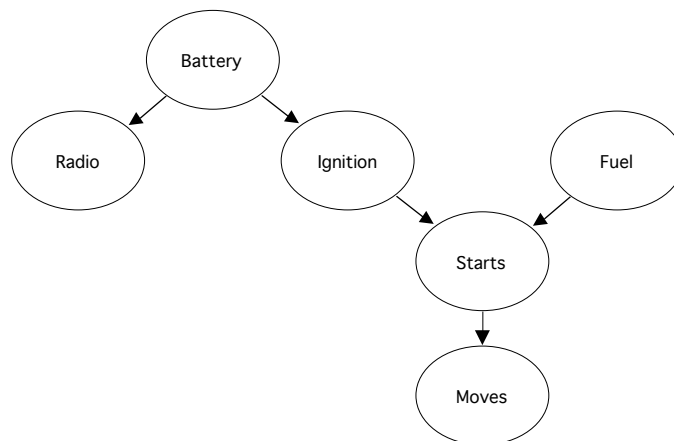
A second important aspect of this belief network is that  $\text{Pit}_{0,1}$  and  $\text{Pit}_{1,0}$ , etc, have *no* connecting links, meaning that they are *independent*. Therefore, for example,  $P(\text{Pit}_{0,1}\ \square\ \text{Pit}_{1,0}\ \square\ \text{Pit}_{2,1}\ \square\ \square\ \text{Pit}_{1,2}) = P(\text{Pit}_{0,1})\ \square\ P(\text{Pit}_{1,0})\ \square\ P(\text{Pit}_{2,1})\ \square\ P(\square\ \text{Pit}_{1,2})$ .

More subtly, a breeze in cell (1, 1) makes it more likely there is a pit in cell (1, 2), which makes it more likely that there is a breeze in cell (1, 3). Thus a breeze in (1, 1) makes it more likely that there is a breeze in cell (1, 3). Despite this, one breeze does not *cause* another.

In the above belief network, 4 cells have 4 parents, 8 cells have 3 parents, and 4 cells have 2 parents. A cell with 4 parents needs  $2^4=16$  conditional probabilities to be specified, one for each combination of parent values. A cell with 3 parents needs 8, and a cell with 2 parents needs 4. The total number of entries in the 16 joint probability tables (one for each cell) is  $4\ \square\ 16+8\ \square\ 8+4\ \square\ 4=144$ . We also need the 16 unconditional probabilities of each cell being a pit. Compare this with  $2^{32}=4,294,967,296$  entries for the full joint probability table.

### 9.3.1. Conditional Independence in Belief Networks

Here is a belief network for a car,



<sup>69</sup> This is what makes the game interesting.

It expresses the notions that the battery causes the radio and the ignition to work, that both fuel and ignition cause it to start, and if it starts, it can move. The network is probabilistic: even if the ignition works and the car has fuel, that is no guarantee the engine will start. For that matter, a car can move even without the help of the engine, e.g., if it is on a steep slope, or is hit by a train.

If we are given evidence about the state of the radio, we can estimate the probability that the battery is OK. This is a **diagnostic inference**. If we are given evidence about whether the car has fuel, we can estimate the probability that the car will start. This is a **causal inference**. If we are given evidence about the state of the ignition, we can estimate whether the car has fuel. For example, if the car won't start and the ignition is OK, this makes it more likely that the car is out of fuel. This is an **inter-causal inference**. This is also called 'explaining away'. The general case, partly diagnostic, partly causal, is called a **mixed inference**.

Here is where things get tricky: If the radio doesn't work, this suggests a problem with the battery, which suggests the ignition won't work and the car won't start. But, if we are told that the battery is OK (or is not OK), then whether the radio works has no influence over whether the ignition works. *Radio* and *Ignition* are independent given *Battery*.

If we are told that the ignition works (or doesn't work) then the state of the battery is independent of whether the car starts; *Battery* and *Starts* are independent given *Ignition*.

Contrast these two cases with being told that the engine starts or doesn't start. If it starts, this is strong evidence that the ignition works and the car has fuel; if it doesn't, then there is evidence that *either* the ignition doesn't work *or* the car is out of fuel, but not usually both. In this case, *Ignition* and *Fuel* are *mutually dependent* given *Starts*. On the other hand, if we have *no* evidence about *Starts*, *Ignition* and *Fuel* are *independent*.

In the first two cases, the evidence variables (*Battery* or *Ignition*) are said to **directionally separate** the belief network into two independent parts; in the last, the evidence variable (*Starts*) *destroys* the independence of its ancestors. These conclusions can be drawn directly from the topology of the network, without understanding the problem.<sup>70</sup> If the evidence variable has an arrow entering it and an arrow leaving it, it separates the parent and child. If it has two arrows leaving it, it separates its children. But, if an evidence variable has two arrows entering it, it *destroys* the separation of its parents.

We can also use the separation idea in reverse, through Bayes' rule. What we have referred to as the evidence variable becomes the query variable, and the conditional independence is used to combine evidence about the query.

Consider the case where the query concerns *Ignition*. *Ignition* separates the network into  $\{Battery, Radio\}$ , and  $\{Fuel, Starts, Moves\}$ .  $\{Battery, Radio\}$  forms the **causal support** of *Ignition*.  $\{Fuel, Starts, Moves\}$  forms the **evidential support** of *Ignition*. If the query concerned *Battery*,  $\{Radio\}$  and  $\{Ignition, Fuel, Starts, Moves\}$  form its evidential support. If the query concerned *Starts*,  $\{Radio, Battery, Ignition\}$  and  $\{Fuel\}$  are in its causal support, and  $\{Moves\}$  is its evidential support. These sets are not simply the ancestors or descendants of query variables, but *all* the variables in the sub-graphs connected to their parents or children (ignoring the directions of the arrows).

In the car example, the various sets are disjoint. This is important in practice, because it allows the use of a linear time algorithm. If the sets overlapped, the problem would become NP-complete. A belief network that contains undirected cycles, such as the network concerning pits and breezes, is a combinatorial nightmare.<sup>71</sup> Therefore, we first consider the case that, ignoring the direction of the edges, the network is a tree, or a **directed polytree**.

### 9.3.2. An Algorithm for Polytrees

How do we combine the evidential and causal support for a query variable? Let the query variable be  $Q$ , with causal support  $C$ , and evidential support  $E$ . We need to find  $P(Q|E \wedge C)$ . We use the conditional form of Bayes' rule with  $C$  as the conditioning context,

$$P(Q|E \wedge C) = P(E|Q \wedge C) \cdot P(Q|C) \div P(E|C).$$

Now (given that the network is a polytree),  $Q$  *directionally separates* the network into  $C$  and  $E$ , so  $C$  and  $E$  are independent given  $Q$ . Therefore  $P(E|Q \wedge C) = P(E|Q)$ , and

$$P(Q|E \wedge C) = P(E|Q) \cdot P(Q|C) \div P(E|C).$$

This has split the problem into two parts; finding  $P(E|Q)$  and finding  $P(Q|C)$ —an example of 'divide and conquer'. However, there is usually no easy way to find  $P(E|C)$ , so it must be found by normalisation.

$P(Q|C)$  is the easier term to compute. If we knew the probabilities for each of  $Q$ 's parents, we could compute  $P(Q)$  from the joint probability tables for  $Q$ . If  $Q$  has  $N$  parents, there must be  $2^N$  rows in the table, each of which has to be considered.<sup>72</sup> Given that  $Q$ 's parents must belong to separate sub-trees of the polytree, dealing

<sup>70</sup> That is because the network topology encapsulates our understanding of the problem.

<sup>71</sup> One might argue that this is what makes the game interesting.

<sup>72</sup> Assuming, for simplicity, that we are dealing with two-valued random variables.

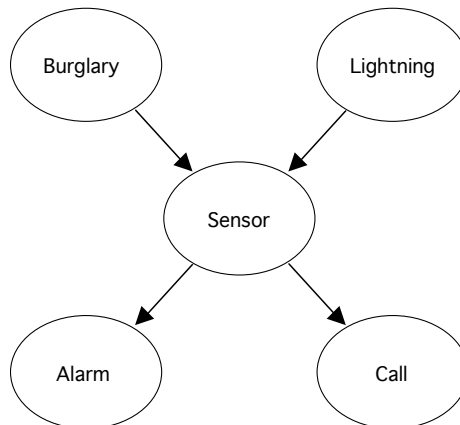
with each parent is a sub-problem exactly like that of finding  $P(Q)$ . Further, since  $Q$  separates them, they are independent, so we can simply multiply probabilities.

To find  $P(E|Q)$ , we observe that the children of  $Q$  also belong to separate sub-trees, so we can deal with each separately and multiply probabilities as before. Of course,  $Q$ 's children may have other parents apart from  $Q$ , so again it is necessary to deal with each row of their joint probability tables as a separate case.

The above description suggests a recursive backward-chaining algorithm that solves for one query variable. In the course of finding the solution, it will need to evaluate the probability distribution for every other variable in the polytree that is connected to the query variable. In practice, it is more efficient (but more difficult) to use forward chaining from the evidence variables. The resulting algorithm will find *all* the unknowns in time *linear* in the size of the polytree. But there is a constant multiplier that depends on the fan-in for the nodes, because the work for each variable is proportional to  $2^N$ , where  $N$  is the number of its parents.

### 9.3.3. A Prolog Program for Polytrees<sup>73</sup>

We have the following belief system: When a burglar breaks into a house, he is likely to trigger a sensor. The sensor is then supposed to sound an alarm, and make a telephone call. Unfortunately, the sensor can also be triggered by lightning.



As in any good knowledge engineering task, we separate the inference engine (`belief`, which defines `causes` as an operator) from the specific belief network:

```

:- include(belief).

burglary causes sensor.
lightning causes sensor.
sensor causes alarm.
sensor causes call.
  
```

We need the unconditional probabilities of *burglary* and *lightning*:

```

p(burglary, 0.001).
p(lightning, 0.02).
  
```

We also need the conditional probabilities of the other events for each combination of their parent variables:

```

p(sensor, {burglary, lightning}, 0.9).
p(sensor, {burglary, not(lightning)}, 0.9).
p(sensor, {not(burglary), lightning}, 0.1).
p(sensor, {not(burglary), not(lightning)}, 0.001).

p(alarm, {sensor}, 0.95).
p(alarm, {not(sensor)}, 0.001).

p(call, {sensor}, 0.9).
p(call, {not(sensor)}, 0.0).
  
```

(We write  $P(\text{sensor}|\text{burglary}\wedge\text{lightning})=0.9$  as `p(sensor, {burglary, lightning}, 0.9)`, etc.)

The `belief` program *assumes* (but does not test) that the belief network is a polytree.

```

:- include(sets).

:- op(500, xfx, [affects, causes]).
  
```

<sup>73</sup> See I. Bratko, *Prolog Programming for Artificial Intelligence*, Section 15.6.

We define an *affects* relation that holds when  $Y$  has  $X$  or  $\neg X$  as an ancestor:

```
not(X) affects Y :- !, X affects Y.
X affects Y :- X causes Y.
X affects Z :- X causes Y, Y affects Z.
```

The `prob/2` predicate prints the probability of  $X$  given *Givens*:

```
prob(X, Givens) :- prob(X, Givens, Prob), print(Prob), nl.
```

The main predicate of the program has the template `prob(+Q, +C, -Prob)`, and succeeds if  $P(Q|C)=Prob$ . It is defined as a series of strategies, which are explored by backtracking.

A *set of givens* represents a *conjunction* of probabilities. Therefore, we can multiply them using the product rule within a conditioning context,  $C$ :  $P(A \wedge B|C) = P(A|C) \wedge P(B|A \wedge C)$ . (This is an example of *Bayesian updating*.)

```
prob(Givens, Conds, Prob) :- product(Givens, Conds, 1.0, Prob).
```

```
product(Givens0, Conds0, Prob0, Prob) :-
    least(Givens0, Q, Givens1),
    prob(Q, Conds0, ProbQ),
    union({Q}, Conds0, Conds1),
    Prob1 is ProbQ*Prob0,
    product(Givens1, Conds1, Prob1, Prob).
product(Givens, _, Prob, Prob) :- empty(Givens).
```

If a query variable,  $Q$ , is a member of the givens, it is true:

```
prob(Q, Givens, 1.0) :- in(Q, Givens), !.
```

If  $\neg Q$  is given, then  $Q$  is false. Otherwise, we may be able to simplify using  $P(\neg Q|C)=1-P(Q|C)$ .

```
prob(Q, Givens, 0.0) :- in(not(Q), Givens), !.
prob(not(Q), Givens, ProbNotQ) :- prob(Q, Givens, ProbQ), !, ProbNotQ is 1-ProbQ.
```

We now use the form of Bayes' rule with a conditioning context  $C$ , as in the previous section:

$$P(Q|E \wedge C) = P(E|Q \wedge C) \wedge P(Q|C) \div P(E|C).$$

We try to find some evidence variable  $E$  in the givens that is a descendant of  $Q$ , then use Bayes' theorem in the conditioning context,  $C$ , given by all the givens other than  $E$ :

```
prob(Q, C, Prob) :-
    all(E, C), Q affects E,
    difference(C, {E}, CnotE),
    union({Q}, CnotE, CandQ),
    prob(E, CandQ, ProbEgivenQC),
    prob(Q, CnotE, ProbQgivenC),
    prob(E, CnotE, ProbEgivenC),
    ProbEgivenC>0, !,
    Prob is ProbQgivenC*ProbEgivenQC/ProbEgivenC.
```

If  $P(E|C)$  can't be evaluated, treat  $1/P(E|C)$  as a *normalising factor*,  $\alpha$ :

```
prob(Q, C, Prob) :-
    all(E, C), Q affects E,
    difference(C, {E}, CnotE),
    union({Q}, CnotE, CandQ),
    prob(E, CandQ, ProbEgivenQC),
    prob(Q, CnotE, ProbQgivenC),
    AlphaProbQ is ProbQgivenC*ProbEgivenQC,
    union({not(Q)}, CnotE, CandNotQ),
    prob(not(Q), CnotE, ProbNotQgivenC),
    prob(E, CandNotQ, ProbEgivenNotQC),
    AlphaProbNotQ is ProbNotQgivenC*ProbEgivenNotQC,
    (AlphaProbQ+AlphaProbNotQ)>0, !,
    Prob is AlphaProbQ/(AlphaProbQ+AlphaProbNotQ).
```

If no descendant of  $Q$  is in the givens, and  $Q$  has no parents, use the unconditional probability  $P(Q)$ .

```
prob(Q, _, Prob) :- p(Q, Prob), !.
```

If  $Q$  has parents, it has a conditional probability table, so sum the probabilities for each possible assignment of its parents' truth values:

```

prob(Q, Givens, Prob) :- !,
    findset(Assignment-ProbQgivenAss,
            p(Q, Assignment, ProbQgivenAss),
            Conditional_Probs),
    sum_probs(Conditional_Probs, Givens, 0.0, Prob), !.

sum_probs(CondProbs0, Givens, Prob0, Prob) :-
    least(CondProbs0, Assignment-ProbQgivenAss, CondProbs1),
    prob(Assignment, Givens, ProbAss),
    Prob1 is ProbAss*ProbQgivenAss+Prob0,
    sum_probs(CondProbs1, Givens, Prob1, Prob).

sum_probs(CondProbs, _, Prob, Prob) :- empty(CondProbs).

```

The program illustrates that sensor *directionally separates* its children, call and alarm:

```

| ?- prob(call, {sensor, alarm}).
0.900
| ?- prob(call, {sensor, not(alarm)}).
0.900
| ?- prob(alarm, {sensor, call}).
0.950
| ?- prob(alarm, {sensor, not(call)}).
0.950

```

Second, it shows that sensor *directionally separates* its children, e.g., alarm, from its parents, e.g., burglary:

```

| ?- prob(alarm, {sensor, not(burglary)}).
0.950
| ?- prob(alarm, {sensor, burglary}).
0.950
| ?- prob(burglary, {sensor, alarm}).
0.232
| ?- prob(burglary, {sensor, not(alarm)}).
0.232

```

Third, it demonstrates that sensor *destroys* the independence of its parents, burglary and lightning.

```

| ?- prob(burglary, {lightning}).
0.001
| ?- prob(burglary, {not(lightning)}).
0.001
| ?- prob(lightning, {burglary}).
0.020
| ?- prob(lightning, {not(burglary)}).
0.020
| ?- prob(burglary, {sensor, lightning}).
0.009
| ?- prob(burglary, {sensor, not(lightning)}).
0.474
| ?- prob(lightning, {sensor, burglary}).
0.020
| ?- prob(lightning, {sensor, not(burglary)}).
0.671

```

Finally, we see that the program combines causal (burglary) and evidential (alarm) support correctly:

```

| ?- prob(sensor, {burglary}).
0.900
| ?- prob(sensor, {alarm}).
0.787
| ?- prob(sensor, {alarm, burglary}).
1.000

```

**9.3.3.1. Monty Hall Revisited**

How does the program fare with the Monty Hall paradox? We assume that Monty's choice of which door to open depends on the position of the prize:

```
:- include(belief).
prizeA causes montyA.
prizeB causes montyA.
prizeC causes montyA.
prizeA causes montyB.
prizeB causes montyB.
prizeC causes montyB.
prizeA causes montyC.
prizeB causes montyC.
prizeC causes montyC.
```

and also on the contestant's choice:

```
chooseA causes montyA.
chooseB causes montyA.
chooseC causes montyA.
chooseA causes montyB.
chooseB causes montyB.
chooseC causes montyB.
chooseA causes montyC.
chooseB causes montyC.
chooseC causes montyC.
```

We don't need to assume the contestant's choice is unbiased:

```
p(chooseA, 0.3).
p(chooseB, 0.2).
p(chooseC, 0.5).
```

But we shall assume that each door is equally likely to conceal the prize:

```
p(prizeA, 0.33333).
p(prizeB, 0.33333).
p(prizeC, 0.33333).
```

If the contestant chooses correctly, Monty can choose either alternative. Initially, we assume he chooses evenly:

```
p(montyA, {chooseA, prizeA, not(prizeB), not(prizeC), not(chooseB), not(chooseC)}, 0.0).
p(montyB, {chooseA, prizeA, not(prizeB), not(prizeC), not(chooseB), not(chooseC)}, 0.5).
p(montyC, {chooseA, prizeA, not(prizeB), not(prizeC), not(chooseB), not(chooseC)}, 0.5).
```

In other situations, his choice is forced:

```
p(montyA, {chooseA, prizeB, not(prizeA), not(prizeC), not(chooseB), not(chooseC)}, 0.0).
p(montyB, {chooseA, prizeB, not(prizeA), not(prizeC), not(chooseB), not(chooseC)}, 0.0).
p(montyC, {chooseA, prizeB, not(prizeA), not(prizeC), not(chooseB), not(chooseC)}, 1.0).
p(montyA, {chooseA, prizeC, not(prizeA), not(prizeB), not(chooseB), not(chooseC)}, 0.0).
p(montyB, {chooseA, prizeC, not(prizeA), not(prizeB), not(chooseB), not(chooseC)}, 1.0).
p(montyC, {chooseA, prizeC, not(prizeA), not(prizeB), not(chooseB), not(chooseC)}, 0.0).
```

(There are 18 other rules, for chooseB and chooseC.)

```
| ?- prob(prizeA, {chooseA, montyB}).
0.333
| ?- prob(prizeB, {chooseA, montyB}).
0.000
| ?- prob(prizeC, {chooseA, montyB}).
0.667
```

This is as we predicted. Now let's see what happens when Monty chooses unfairly. We adjust his conditional probabilities, and make the same queries:

```
p(montyA, {chooseA, prizeA, not(prizeB), not(prizeC), not(chooseB), not(chooseC)}, 0.0).
p(montyB, {chooseA, prizeA, not(prizeB), not(prizeC), not(chooseB), not(chooseC)}, 0.2).
p(montyC, {chooseA, prizeA, not(prizeB), not(prizeC), not(chooseB), not(chooseC)}, 0.8).
```

```

| ?- prob(prizeA, {chooseA, montyB}).
0.167
| ?- prob(prizeB, {chooseA, montyB}).
0.000
| ?- prob(prizeC, {chooseA, montyB}).
0.833

```

The *relative* probability,

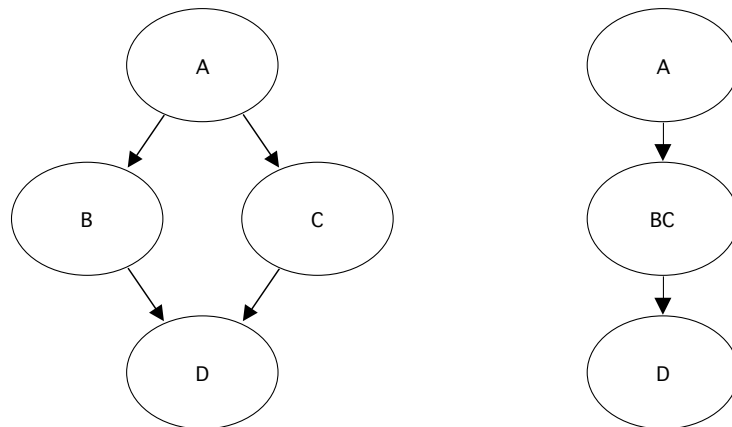
$$\begin{aligned}
 &P(\text{PrizeA}|\text{MontyB} \wedge \text{ChooseA}) \div P(\text{PrizeC}|\text{MontyB} \wedge \text{ChooseA}) \\
 &= 0.167 \div 0.833 = 0.2 \\
 &= P(\text{MontyB}|\text{PrizeA} \wedge \text{ChooseA}),
 \end{aligned}$$

—exactly as predicted by Bayes' Rule, earlier.

### 9.3.4. Multiply-Connected Belief Networks

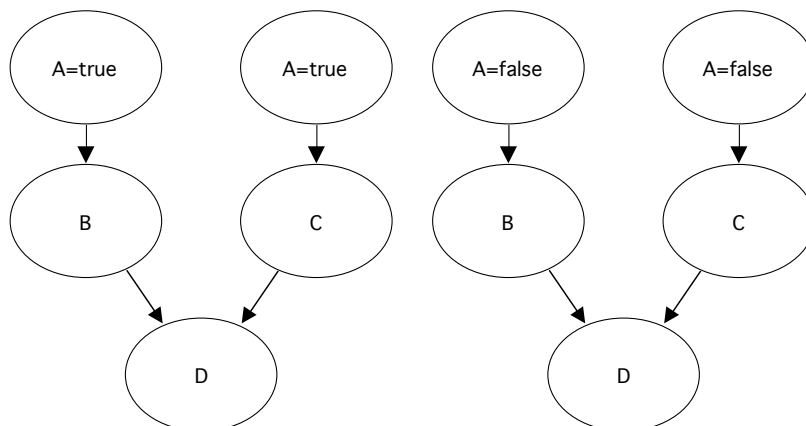
As stated earlier, dealing with networks that contain undirected cycles is *NP*-hard.

One way to remove cycles is to *merge* two or more nodes into a single node, as follows,



The problem here is that if *B* and *C* are 2-valued variables, *BC* is a 4-valued variable. Combining *N* 2-valued variables produces one  $2^N$ -valued variable. The conditional probability tables for a merged network typically contain more entries than the tables for the network it is derived from; the *tables* grow in size exponentially as variables are merged.

The opposite approach is to *split* some nodes, which then requires them to be instantiated with truth values, and each case has then to be considered as a separate sub-problem, with the cases combined using their appropriate probabilities. The difficulty with this approach is that the *number of cases* grows exponentially.



The most direct and most general approach is through *Monte Carlo* simulation. This involves assigning random values to the evidence variables, *in proportion to their known probabilities*, then propagating these values through the network, according to the probability tables. In **rejection sampling**, we sample all possible configurations, and test if they are consistent with evidence. In the more efficient **likelihood weighting** method, we sample only those variables that are not fixed by the evidence, and assign each sample a weight according to the likelihood that it makes the evidence variables true. Any Monte Carlo method needs many trials to achieve good accuracy.

#### 9.4. Dempster-Shafer Theory

Dempster-Shafer theory deals with a different notion of belief from probability theory. Suppose a doctor is examining a patient to check for lung cancer. In Bayesian analysis, we might say  $P(\text{LungCancer})=0.15$ . But a particular patient either has lung cancer or doesn't. The doctor can hardly say, "Your chances of lung cancer are only 15%. I wouldn't worry about it." At this stage, the doctor has no supporting evidence for the particular patient. The doctor's *confidence* in his diagnosis is measured by a **belief function**. In Dempster-Shafer theory, we write  $Bel(\text{LungCancer})$  as the measure of the doctor's belief that the patient has lung cancer. Initially, both  $Bel(\text{LungCancer})$  and  $Bel(\neg\text{LungCancer})$  are zero. One way to interpret Dempster-Shafer theory is to say that it defines a **probability interval**, in this case 0–1.

Bayesian analysis deals with propositions that are either true or false. But the doctor may be unsure about a patient's symptoms. He may, for example, feel 90% sure that a chest X-ray shows no suspicious shadows. Dempster-Shafer theory allows for this uncertain evidence and provides an algebra that allows the belief function to be refined, thus narrowing the probability interval. Unfortunately, the mathematical and philosophical foundations of the theory seem a bit shaky, and it is debatable whether this approach has any advantage over Bayesian analysis. For example, the doctor's 90% belief can be modelled by a node in the belief network that assigns  $P(\neg\text{Shadow}|\text{NoShadowSeen})=0.9$ ,  $P(\text{Shadow}|\text{NoShadowSeen})=0.1$ .

Such belief functions may explain why many people are unhappy with the Bayesian analysis of the Monty Hall paradox. What we *don't* know is whether Monty *always* gives contestants a second choice. Do we believe he is on the side of the contestant, merely neutral, or trying to fool them? If he gives the contestant a second choice only when his first is wrong, then the contestant should *always* switch. If he gives the contestant a second choice only when his first is correct, the contestant should *never* switch. If he always offers a second choice, the Bayesian analysis is correct. However, we can easily incorporate belief into the analysis by including two conditional probabilities: that Monty will offer a second chance if the contestant's choice is correct, and that Monty will offer a second chance if it is wrong. Our earlier analysis assumed that both these probabilities were 1.0.

## 10. Fuzzy Logic

### 10.1. Fuzzy Set Theory

Fuzzy logic is based on the idea that **natural kinds** are not well defined. For example, although we can talk about the set ‘tall men’, not everyone would agree about how tall is ‘tall’. (Unless we arbitrarily define ‘tall’ as being over 2 metres, or whatever.) People speak of being ‘rather tall’, ‘very tall’ and so on. If we talk to an expert, he or she is much more likely to talk in these terms than in terms of precisely defined categories. We assign to the statement “Mildred is tall.” a **degree of truth** in the range 0–1. We let  $\mu P$  denote the **truth-value** of proposition  $P$ . Similarly,  $\mu(X \in S)$  denotes the **membership function** of  $X$  in  $S$ , sometimes written as  $\mu_S X$ .

One constraint on fuzzy logic is that it should work correctly in the classical case where truth-values are limited to 0 or 1. We therefore require the following pairs of axioms to hold,

$\mu(True)=1,$	$\mu(False)=0,$	Classical case
$\mu(X \sqcap X)=\mu X,$	$\mu(X \sqcup X)=\mu X,$	
$\mu(X \sqcap True)=\mu X,$	$\mu(X \sqcup True)=1,$	
$\mu(X \sqcap False)=0,$	$\mu(X \sqcup False)=\mu X,$	
$\mu(X \sqcap Y)=\mu(Y \sqcap X)$	$\mu(X \sqcup Y)=\mu(Y \sqcup X),$	Commutativity
$\mu(X \sqcap (Y \sqcap Z))=\mu((X \sqcap Y) \sqcap Z)$	$\mu(X \sqcup (Y \sqcup Z))=\mu((X \sqcup Y) \sqcup Z)$	Associativity

#### 10.1.1. Logical Operators

We define the functions for *AND*, *OR* and *NOT* to compute truth-values,

$$\begin{aligned} AND(\mu X, \mu Y) &= \mu(X \sqcap Y), \\ OR(\mu X, \mu Y) &= \mu(X \sqcup Y), \\ NOT(\mu X) &= \mu(\sqcap X) \end{aligned}$$

One commonly used choice is the set of **min/max** fuzzy functions:

$$\begin{aligned} AND(\mu X, \mu Y) &= \min(\mu X, \mu Y), \\ OR(\mu X, \mu Y) &= \max(\mu X, \mu Y), \\ NOT(\mu X) &= 1 - \mu X. \end{aligned} \quad ^{74}$$

A second possibility is the set of **independent probability** functions:

$$\begin{aligned} AND(\mu X, \mu Y) &= \mu X \sqcap \mu Y, \\ OR(\mu X, \mu Y) &= \mu X + \mu Y - \mu X \sqcap \mu Y, \\ NOT(\mu X) &= 1 - \mu X. \end{aligned}$$

Both these choices satisfy all the above rules. Unfortunately, the following two rules of classical logic *don't* hold for all  $\mu X$ .

$$\begin{aligned} \mu(X \sqcap \sqcap X) &= 0, \\ \mu(X \sqcup \sqcup X) &= 1 \end{aligned}$$

For example, if  $\mu X=0.5$ , the min/max functions give  $\mu(X \sqcap \sqcap X)=0.5$ ,  $\mu(X \sqcup \sqcup X)=0.5$ , and the probability functions give  $\mu(X \sqcap \sqcap X)=0.25$ ,  $\mu(X \sqcup \sqcup X)=0.75$ . In classical logic, “Is Mildred either tall or not tall?” should receive an unequivocal “Yes!” In fuzzy logic, the answer is more like, “You can’t say she is, and you can’t say she isn’t.”

To restore  $\mu(X \sqcap \sqcap X)=0$  and  $\mu(X \sqcup \sqcup X)=1$ , it is possible to use **disjoint probability** functions:

$$\begin{aligned} OR(\mu X, \mu Y) &= \mu X + \mu Y, \\ NOT(\mu X) &= 1 - \mu X, \\ AND(\mu X, \mu Y) &= 1 - ((1 - \mu X) + (1 - \mu Y)) = \mu X + \mu Y - 1. \end{aligned}$$

These functions have problems too.  $OR(\mu X, \mu X)=2\mu X$ , so the function needs to be capped at 1.0, i.e.,  $OR(\mu X, \mu Y)=\min(\mu X + \mu Y, 1.0)$ . Likewise, we must use  $AND(\mu X, \mu Y)=\max(\mu X + \mu Y - 1, 0.0)$ .

In practice, the min/max functions are almost always used, because they are simple to compute. Despite difficulties in reconciling fuzzy logic with two-valued logic, fuzzy systems are widely used, and successful commercially. Fuzzy systems have been used as image stabilisers in hand-held video cameras, to control the cycles of washing machines, to control industrial plant, and even a model helicopter. This last case is especially interesting, because hovering a helicopter is a notoriously difficult problem, which has strongly resisted traditional control theory.

#### 10.1.2. Linguistic Modifiers

One feature of fuzzy logic is that it allows the use of linguistic *hedges*, such as ‘very’ and ‘somewhat’. This is implemented by raising the degree of truth of a proposition to a power that depends on the hedge. For example, we may use  $\mu^{very} X = \mu X^2$ , and  $\mu^{rather} X = \mu X^{0.5}$ .

<sup>74</sup> If you are familiar with lattice theory, this choice may seem quite natural.

## 10.2. Fuzzy Expert Systems

The inputs and outputs of a fuzzy inference system are often analogue quantities. When a system is sufficiently complex, even an expert can't provide a correct set of equations to convert analogue inputs to outputs. Instead, the expert *can* say what outputs are needed in certain well-defined situations. A fuzzy expert system joins the dots.

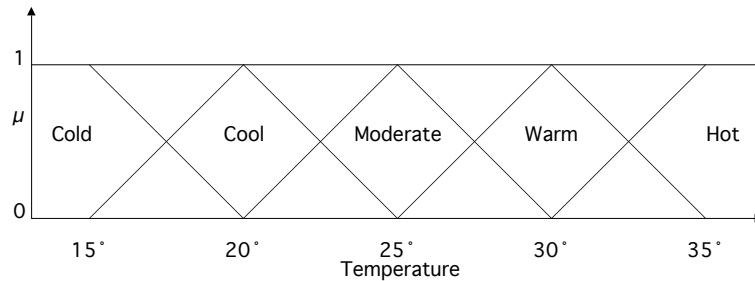
For example, an expert rule might be "If steam pressure is high and demand is low, then fuel flow should be very slow." Another might be "If steam pressure is low and demand is high, then fuel flow should be fast." Fuzzy logic is used to measure the strength with which these, and other, competing rules apply, then set an appropriate fuel rate.

A fuzzy expert system uses 4 steps:<sup>75</sup>

- 1 Fuzzification
- 2 Fuzzy Inference
- 3 Aggregation
- 4 Defuzzification

### 10.2.1. Fuzzification

Fuzzification begins with the measurement of one or more **crisp variables**, usually analogue quantities. Their values are converted into truth values using suitable membership functions. One crisp variable can belong to several fuzzy sets. For example, temperature may be *Cold*, *Cool*, *Moderate*, *Warm* or *Hot*, or, to some extent, a mixture of these:



The above graph shows how a crisp thermometer reading can be converted to several truth values. Any temperature  $\leq 15^\circ$  is *Cold* with a truth value of 1. Its truth value for *Cool*, *Moderate*, etc., is 0.0. A temperature of  $22^\circ$  is *Cool* with truth value 0.6, and *Moderate* with a truth value of 0.4. Any temperature  $\geq 35^\circ$  is considered *Hot* with a truth value of 1.0.

Membership functions do not *have* to consist of straight-line segments. It is possible to use, e.g., bell-shaped or sigmoid curves. Straight-line functions are widely used in practice because they are easy to compute. For example,

$$\begin{aligned} \mu(T \sqcap \text{Moderate}) &= 0 \text{ if } T < 20, \\ \mu(T \sqcap \text{Moderate}) &= 0 \text{ if } T > 30, \\ \mu(T \sqcap \text{Moderate}) &= (T - 20) \div 5, \text{ if } 20 \leq T \leq 25, \\ \mu(T \sqcap \text{Moderate}) &= (30 - T) \div 5, \text{ if } 25 \leq T \leq 30. \end{aligned}$$

In the above graph, the truth values for *Cold*, *Cool*, *Moderate*, *Warm* and *Hot* always sum to 1 for any value of  $T$ . This seems reasonable, but it is not a necessary feature of fuzzy systems.

### 10.2.2. Fuzzy Inference

An air-conditioner system might allow its pump setting to be *Heat*, *Cool* or *Off*, but allow its fan to be operated at any desired speed. The pump setting is a discrete variable, but the fan speed is a continuous variable. Discrete and continuous variables have to be handled differently.

A fuzzy rule has the form 'IF *Antecedent* THEN *Consequents*', similar to the rules in a production system. Here are couple of possible rules.

IF Temperature IS Moderate AND Humidity IS Sultry THEN Pump IS Off, Fan IS Slow.  
 IF Temperature IS Warm AND Humidity IS Medium THEN Pump IS Cool, Fan IS Fast.

If the antecedent is true, the consequents are asserted. In general, the antecedent is only *partly* true, so it has to be assigned a truth value according to the rules for the *AND*, *OR* and *NOT* functions involved. Fuzzy rule sets are usually given in disjunctive normal form (DNF). They are sums of products. Different rules are implicitly linked by *OR*.

For example, assume the temperature,  $T = 28^\circ$ . Then  $\mu(T \sqcap \text{Warm}) = 0.6$  and  $\mu(T \sqcap \text{Moderate}) = 0.4$ . Assume that the humidity is 74%, and assume that as a result  $\mu(H \sqcap \text{Medium}) = 0.7$  and  $\mu(H \sqcap \text{Sultry}) = 0.3$ . Then the antecedent

<sup>75</sup> See Negnevitsky, *Op. cit.*, Chapter 4.

of the first rule has truth value  $\min(0.4, 0.3)=0.3$ , and the antecedent of the second rule has truth value  $\min(0.6, 0.7)=0.6$ .

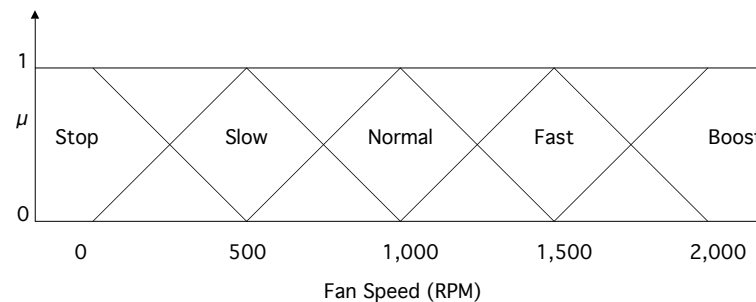
The above two rules each define *two* output variables: the pump setting, and the fan speed. They are really shorthand for the following four rules:

- IF Temperature IS Moderate AND Humidity IS Sultry THEN Pump IS Off.
- IF Temperature IS Warm AND Humidity IS Medium THEN Pump IS Cool.
- IF Temperature IS Moderate AND Humidity IS Sultry THEN Fan IS Slow.
- IF Temperature IS Warm AND Humidity IS Medium THEN Fan IS Fast.

### 10.2.3. Aggregation

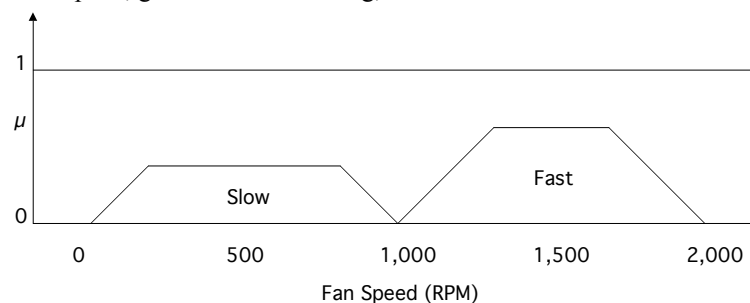
At any one time, several rules may be active to various extents, each giving (often conflicting) advice about each output variable. If we have a rule of the form  $P \square Q$ , given the truth value of  $P$ , what truth value do we attach to the consequent,  $Q$ ?

First, observe that each possible consequent has a membership function that depends on the fan speed:



Now consider the truth of  $\mu Q = \mu(P \square (P \square Q))$ . We may rewrite this as  $\mu Q = \mu(P \square (\square P \square Q))$  or  $\mu Q = \mu((P \square \square P) \square (P \square Q))$ . Then, by regarding  $(P \square \square P)$  as false<sup>76</sup>, we get  $\mu Q = \mu(P \square Q)$ .

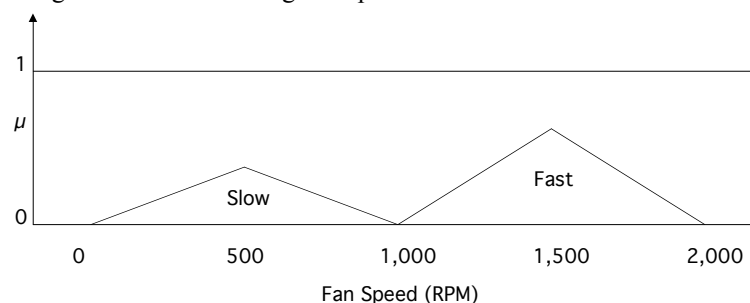
Using the usual *min* rule for *AND*, this gives us  $\min(\mu P, \mu Q)$ . Considering the case of the fan, there are two possible consequents: *Slow* and *Fast*.<sup>77</sup> Plotting  $\min(\mu(\text{Moderate} \square \text{Sultry}), \mu \text{Slow})$  and  $\min(\mu(\text{Warm} \square \text{Medium}), \mu \text{Fast})$  as functions of fan speed, gives us the following,



This function is  $\mu Q$  limited to at most  $\mu P$ , so this process is referred to as **clipping**.

(Because the rules are in DNF, we combine them using *OR*, which means taking their maximum.)

Another alternative is to use the independent probability rule,  $\mu P \square \mu Q$ . This is called **scaling**. To be consistent, we *OR* the rules by adding them and subtracting their products.



Whichever approach we take, we are left with a fuzzy set for the fan speed, which is its degree of membership of the set ‘Correct for the given conditions and rules’.

<sup>76</sup> However,  $(P \square \square P) = \text{False}$  is only valid in classical two-valued logic.

<sup>77</sup> In practice, there would be several more conflicting rules.

### 10.2.4. Defuzzification

Defuzzification is the process of obtaining a crisp value from a fuzzy set. There are three main approaches: MOM (mean of maxima), Mamdani-style, and Sugeno-style.

MOM finds the **maxima** of the above functions and takes their **mean**. In the case of *scaling*, there is only one maximum, at 1,500 RPM. With *clipping*, the maxima lie on the horizontal segment from 1,300–1,700 RPM, with mean 1,500 RPM. In either case, the crisp value is 1,500 RPM.

Mamdani defuzzification finds the **centroid** (centre of gravity) of the above areas. Using scaling, the centroid of the *Slow* area is 500 and the centroid of the *Fast* area is 1,500. We take their weighted average, specifically  $(0.3 \times 500 + 0.6 \times 1,500) \div (0.3 + 0.6) = 1,050 \div 0.9$ , i.e., 1,167 RPM. For the case of clipping, this calculation is more complex, and would be more complex still if sigmoid or bell-shaped functions were being used.

Both these methods have trouble with a set like *Boost*, which apparently extends to infinite speeds. So if either method had to use the *Boost* set, the crisp result would be infinite. One solution is to limit the speed range to 0–2,500, say, so that *Boost* defines a finite area.

(First-order) Sugeno defuzzification assigns a **spike** to each set. A suitable spike for *Slow* would be 500, and the value for *High* would be 1,500, and their weighted average would again be 1,167 RPM. This is the easiest method to apply in practice.

Discrete variables, such as the pump setting, have to be treated differently. Normally, the best choice is the one with the greatest truth value.

## 10.3. A Prolog Fuzzy Logic System

Like most cases of knowledge engineering, a fuzzy logic system can be divided into a knowledge base and an inference engine. In the following example, the fuzzy logic uses the min/max operators, and first-order Sugeno-style defuzzification.

### 10.3.1. The Knowledge Base

We begin by including the fuzzy inference engine, which among other things, declares *if*, *then*, *and*, *or*, and *not* as operators (*is* is already a standard operator):

```
:- include(fuzzy).
```

We then define the inference rules:

```
if temp is cold and humidity is dry
    then fan is fast and pump is heat.
if temp is cold and humidity is medium
    or temp is cool and humidity is dry
    then fan is normal and pump is heat.
if temp is cold and humidity is sultry
    or temp is cool and humidity is medium
    then fan is slow and pump is heat.
if temp is cool and humidity is sultry
    or (temp is moderate or temp is warm) and humidity is dry
    or temp is moderate and humidity is medium
    then fan is stop and pump is off.
if temp is moderate and humidity is sultry
    or temp is warm and humidity is medium
    then fan is slow and pump is off.
if temp is warm and humidity is sultry
    then fan is slow and pump is chill.
if temp is hot and humidity is dry
    then fan is normal and pump is chill.
if temp is hot and humidity is medium
    then fan is fast and pump is chill.
if temp is hot and humidity is sultry
    then fan is boost and pump is chill.
```

To simplify the definition of membership functions, the inference engine defines some useful predicates. The functor *-/-* represents a function that is flat (*-*), then climbs (*/*) and is then flat (*-*) again, *- \ -* is similar, but falls, *- / \ -* rises to a sharp peak then falls, and *- / - \ -* is a plateau:

```
mu(temp is cold, Temp, Mu)      :- - \ -(15, 20, Temp, Mu).
mu(temp is cool, Temp, Mu)     :- - / \ -(15, 20, 25, Temp, Mu).
mu(temp is moderate, Temp, Mu) :- - / \ -(20, 25, 30, Temp, Mu).
mu(temp is warm, Temp, Mu)     :- - / \ -(25, 30, 35, Temp, Mu).
mu(temp is hot, Temp, Mu)      :- - / - \ -(30, 35, Temp, Mu).
```

```

mu(humidity is sultry, Humidity, Mu) :- --(65, 95, Humidity, Mu).
mu(humidity is medium, Humidity, Mu) :- --\-(5, 35, 65, 95, Humidity, Mu).
mu(humidity is dry, Humidity, Mu) :- --(5, 35, Humidity, Mu).

mu(fan is stop, Fan, Mu) :- --(0, 500, Fan, Mu).
mu(fan is slow, Fan, Mu) :- --\-(0, 500, 1000, Fan, Mu).
mu(fan is normal, Fan, Mu) :- --\-(500, 1000, 1500, Fan, Mu).
mu(fan is fast, Fan, Mu) :- --\-(1000, 1500, 2000, Fan, Mu).
mu(fan is boost, Fan, Mu) :- --(1500, 2000, Fan, Mu).

```

However, the membership functions for discrete variables are defined differently:

```

mu(pump is stop, stop, 1.0) :- !.
mu(pump is chill, chill, 1.0) :- !.
mu(pump is heat, heat, 1.0) :- !.
mu(pump is _, _, 0.0).

```

### 10.3.2. The Inference Engine

#### 10.3.2.1. Some Preliminaries

In order to be able to write rules in **if ... then** form, we begin by defining *if*, *then*, *and*, *or* and *not* as operators:

```

:- op(950, fx, if).
:- op(900, xfx, [then, when]).
:- op(850, xfy, or).
:- op(800, xfy, and).
:- op(750, fy, not).

```

We next define the predicates used to specify typical membership functions:

```

--(Low, _, Temp, 1.0) :- Temp<Low, !.
--(_, High, Temp, 0.0) :- Temp>=High, !.
--(Low, High, Temp, Mu) :- Mu is (High-Temp)/(High-Low).

--(Low, _, Temp, 0.0) :- Temp<=Low, !.
--(_, High, Temp, 1.0) :- Temp>=High, !.
--(Low, High, Temp, Mu) :- Mu is (Temp-Low)/(High-Low).

--\-(Low, Medium, High, Temp, Mu) :-
  --(Low, Medium, Temp, Mu1),
  --(Medium, High, Temp, Mu2),
  min([Mu1, Mu2], Mu).

--\-(Low, MedLow, MedHigh, High, Temp, Mu) :-
  --(Low, MedLow, Temp, Mu1),
  --(MedHigh, High, Temp, Mu2),
  min([Mu1, Mu2], Mu).

```

Since we are using Sugeno defuzzification, we need a predicate to reduce each type of membership function to a spike at the function's 'central' value:

```

:- public(mu/3).
sugeno(Variable is Setting, Spike) :-
  clause(mu(Variable is Setting, _, _), --(Spike, _, _, _)), !.
sugeno(Variable is Setting, Spike) :-
  clause(mu(Variable is Setting, _, _), --(_ , Spike, _, _)), !.
sugeno(Variable is Setting, Spike) :-
  clause(mu(Variable is Setting, _, _), --\-(_, Spike, _, _, _)), !.
sugeno(Variable is Setting, Spike) :-
  clause(mu(Variable is Setting, _, _), --\-(_, Low, High, _, _, _)),
  Spike is (Low+High)/2.

```

Because the spike value depends on the way the membership function is defined, we used Prolog's `clause/2` predicate, which returns the heads and bodies of rules. This lets the program find out which predicate has been used to define the membership function, `mu`, and to choose its appropriate argument. Such introspection is only possible because `mu/3` has been declared `public`.

Since we need to find maxima, minima, and totals, we define the following useful operations on *lists* of arithmetic expressions. (Sets are *not* appropriate,  $\{1, 1, 1\} = \{1\}$ .)

```

sum(List, Sum) :- sum(List, 0, Sum).
sum([], Sum, Sum).
sum([H|T], Sum0, Sum) :- Sum1 is Sum0+H, sum(T, Sum1, Sum).

```

Fuzzy Logic

```
min([H|T], Min) :- H1 is H, min(T, H1, Min).
min([], Min, Min).
min([H|T], Min0, Min) :- H1 is H, H1<=Min0, !, min(T, H1, Min).
min([_|T], Min0, Min) :- min(T, Min0, Min).
max([H|T], Max) :- H1 is H, max(T, H1, Max).
max([], Max, Max).
max([H|T], Max0, Max) :- H1 is H, H1>=Max0, !, max(T, H1, Max).
max([_|T], Max0, Max) :- max(T, Max0, Max).
```

The inference engine has four stages, as described earlier:

```
fuzzy(CrispInputs) :-
    fuzzify(CrispInputs, FuzzySets),
    nl, print('Fuzzy Sets'), nl, print(FuzzySets), nl,
    fuzzy_infer(FuzzySets, RuleMus),
    nl, print('Antecedents'), nl, print(RuleMus), nl,
    aggregate(RuleMus, Consequents),
    nl, print('Consequents'), nl, print(Consequents), nl,
    crispen(Consequents, CrispOutputs),
    nl, print('Settings'), nl, print(CrispOutputs), nl.
```

A typical query would have the form:

```
fuzzy({temp=22, humidity=50}).
```

(We use = for crisp values, and is for fuzzy membership.)

The fuzzify predicate converts the list of crisp values to a list of fuzzy values:

```
Fuzzy Sets
{0.000 when humidity is dry, 0.000 when humidity is sultry,
 0.000 when temp is cold, 0.000 when temp is hot,
 0.000 when temp is warm, 0.400 when temp is moderate,
 0.600 when temp is cool, 1.000 when humidity is medium}
```

Each member of the list has the form ' $\mu(X \text{ is } S)$  when  $X$  is  $S$ '.

fuzzy\_infer takes this list, applies min/max logic, and yields the truth value of each rule:

```
Antecedents
{0.000 when fan is boost and pump is chill,
 0.000 when fan is fast and pump is chill,
 0.000 when fan is fast and pump is heat,
 0.000 when fan is normal and pump is chill,
 0.000 when fan is normal and pump is heat,
 0.000 when fan is slow and pump is chill,
 0.000 when fan is slow and pump is off,
 0.400 when fan is stop and pump is off,
 0.600 when fan is slow and pump is heat}
```

The aggregate predicate then finds the truth value for each fuzzy set and each consequent:

```
Consequents
{0.000 when fan is boost, 0.000 when fan is fast,
 0.000 when fan is normal, 0.000 when fan is slow,
 0.000 when pump is chill, 0.000 when pump is heat,
 0.000 when pump is off, 0.400 when fan is stop,
 0.400 when pump is off, 0.600 when fan is slow,
 0.600 when pump is heat}
```

Finally, crispen performs the defuzzification step, yielding a set of crisp values:

```
Settings
{fan=300.000, pump=heat}
```

### 10.3.2.2. Fuzzification

The fuzzify predicate uses the previously defined membership function, mu:

```
fuzzify(CrispInputs, FuzzySets) :-
    findset(Mu when Variable is FuzzySet,
            (all(Variable=CrispValue, CrispInputs),
             mu(Variable is FuzzySet, CrispValue, Mu)),
            FuzzySets).
```

### 10.3.2.3. Fuzzy Inference

The `fuzzy_infer` predicate finds the truth value of the antecedent clause of each rule, and associates it with its consequent. (Remember that an `if .. then` rule is simply a Prolog fact.)

```
fuzzy_infer(FuzzySets, RuleMus) :-
    findset(Mu when Consequent,
            (if Antecedent then Consequent,
             fuzzy_value(Antecedent, FuzzySets, Mu)),
            RuleMus).
```

To do this, `fuzzy_infer` calls `fuzzy_value`, which applies the min/max norms of fuzzy logic recursively:

```
fuzzy_value(Term1 and Term2, FuzzySets, Mu) :-
    fuzzy_value(Term1, FuzzySets, Mu1),
    fuzzy_value(Term2, FuzzySets, Mu2),
    min([Mu1, Mu2], Mu).
fuzzy_value(Term1 or Term2, FuzzySets, Mu) :-
    fuzzy_value(Term1, FuzzySets, Mu1),
    fuzzy_value(Term2, FuzzySets, Mu2),
    max([Mu1, Mu2], Mu).
fuzzy_value(not Term, FuzzySets, Mu) :-
    fuzzy_value(Term, FuzzySets, Mu1),
    Mu is 1-Mu1.
fuzzy_value(Variable is Setting, FuzzySets, Mu) :-
    in(Mu when Variable is Setting, FuzzySets).
```

### 10.3.2.4. Aggregation

The aggregate predicate forms a set whose terms are of the form ' $\mu(X \square S)$  when  $X$  is  $S$ ', where ' $X$  is  $S$ ', is a *term* in the consequent of some rule, and  $\mu(X \square S)$  is the truth value of its antecedent. The `bagof` predicate will backtrack for each value of ' $X$  is  $S$ ', and `max` will OR the rules by finding the greatest value of  $\mu(X \square S)$ :

```
aggregate(RuleMus, Consequents) :-
    findset(Mu when Variable is Setting,
            (bagof(Mu, all(Mu when Consequent, RuleMus), Mus),
             term(Variable is Setting, Consequent),
             max(Mus, Mu)),
            Consequents).
```

`term/2` succeeds if its first argument is a term of the form ' $X$  is  $S$ ' that appears somewhere in its second argument:

```
term(Term, Term) :- Term=(_ is _), !.
term(Term, Term and _).
term(Term, _ and Term).
term(Term, Term or _).
term(Term, _ or Term).
term(Term, not Term).
```

### 10.3.2.5. Defuzzification

The `crispen` predicate calls `crisp_value` to assign a crisp value to each variable:

```
crispen(Consequents, CrispOutputs) :-
    findset(Variable, all(_ when Variable is _, Consequents), Variables),
    findset(Variable=CrispValue,
            (all(Variable, Variables),
             crisp_value(Consequents, Variable=CrispValue)),
            CrispOutputs).
```

Continuous variables are assigned the weighted average of Sugeno-derived spikes.

```

crisp_value(Consequents, Variable=CrispValue) :-
    findset(Mu when Variable=Spike,
            (all(Mu when Variable is Setting, Consequents),
             sugeno(Variable is Setting, Spike)),
            Spikes),
    in(_ when Variable=_, Spikes),
    bagof(CrispValue*Mu,
          all(Mu when Variable=CrispValue, Spikes),
          WeightedValues),
    bagof(Mu, Consequent^all(Mu when Consequent, Spikes), Weights),
    sum(WeightedValues, TotalValue),
    sum(Weights, TotalWeight),
    CrispValue is TotalValue/TotalWeight, !.

```

Because sugeno is only applicable to continuous variables, the above rule fails for discrete variables. Instead, discrete variables are assigned the setting with greatest truth value:

```

crisp_value(Consequents, Variable=CrispValue) :-
    bagof(Mu,
          Setting^all(Mu when Variable is Setting, Consequents)),
          Mus),
    max(Mus, Mu),
    in(Mu when Variable is CrispValue, Consequents), !.

```

#### 10.4. Reconciling Fuzzy Logic and Probability Theory

If we consider carefully the fuzzy logic system just described, we will find that it merely solves a two-level AND/OR problem, and this typical simplicity of the rule base in fuzzy control systems is perhaps the reason for their practical successes. One way to look at the building of a fuzzy logic controller is that an expert decides what its outputs should be in certain well-defined situations. In less well-defined situations, rather than the fuzzy-logic system performing *logic*, it can be understood to be *interpolating* between two or more well-defined situations.

For more complex sets of rules, it has been suggested that a proper Bayesian analysis should be used. To treat fuzzy sets as probabilities, membership of the concept ‘tall’ could be *measured* by asking a balanced sample of people whether given individuals were tall. The degree of membership in each case would be measured by the fraction of people who agreed with the proposition. The membership function is then the probability that a random person would agree with proposition.

A similar technique could be used to measure the membership functions of ‘heavy’ or ‘overweight’. To know the conditional probabilities that connect them, we might ask people to record whether each individual in a sample was ‘tall’, ‘heavy’ or ‘overweight’.

Armed with these results, we could then propose a belief network in which ‘tall’ and ‘heavy’ *cause* ‘overweight’, and calculate the four conditional probabilities involved. (Of course, if we were to allow more categories of height, weight and obesity, we would obtain better predictive power.) A strong objection to this approach is that, in designing an expert system, we cannot usually afford the research, and experts are very poor at guessing probabilities, and tend to be inconsistent.

Some early rule-based systems, such as MYCIN, assigned **certainty factors** to rules, which is a similar concept to conditional probability, but is really just an expert guess. Assigning certainty factors seems easier for experts to do than discovering correct probabilities.

However, if we have a clear idea about the structure of the belief network, in some cases we can *teach* it the probabilities it needs by presenting it with many sample cases. (This is not unlike adjusting transition probabilities in a Hidden Markov Model, described later.) In such a case, the role of the expert is to suggest an appropriate causal model.

Another possible compromise is to use Bayesian statistics with a **noisy-OR** relation, a generalisation of logical OR. Suppose that *Fever* can be caused by either *Malaria* or *Flu*. In a probabilistic model, we may find that, even in the presence of *Malaria* or *Flu*, *Fever* is false, owing to some inhibiting factor. If we assume that whatever inhibits *Malaria* is independent of what inhibits *Cold*, we need to know only two conditional probabilities:

$$P(\neg Fever|Malaria \square \neg Flu) \text{ and } \\ P(\neg Fever|\neg Malaria \square Flu)$$

From these, we calculate,

$$P(\neg Fever|Malaria \square Flu) = P(\neg Fever|Malaria \square \neg Flu) \square P(\neg Fever|\neg Malaria \square Flu)$$

The remaining conditional probabilities, for  $P(\text{Fever}|\dots)$ , can be derived from these, if we assume that  $P(\text{Fever}|\neg\text{Malaria}\square\neg\text{Flu}) = 0$ . However, if, even in the absence of *Malaria* or *Flu*, *Fever* may be true, we can simply give *Fever* another parent, *Other Causes*, which is deemed to be always true.

This approach gives the same results as the *independent probability* functions in fuzzy set theory.

## 11. Random Sequences

### 11.1. Markov Processes and Markov Models

Markov processes and models form the basis of several computer applications. Their main uses are in speech recognition, data compression, and document classification.

Consider the following text:

Hite pernat eis, Sentis at di: cum pet et lac rat co et sadicaelldis remandor perem rem abonam Aheum in tisceansundis abs, queta tura. Aud Ruant aben. Domis pe tomin bacesum, cat curucter na mass sallati ti vos sentura, Dofus: cant fud pra. Hncrequististum dicem pem aecurgna et tat et marcesupplls fatioesti, estriatus, Dores pimum re promndactuiuesustis. Deter situs mot: de vemis aectiona lam orotrovitio vie mux straterndictomna lus lislcl lusta re dieme, dis, Doriae, loli Mmeus Dorssu ae, Domnons susso Ra, quearogomiromin Juxtus mellis irias ine: taternatis. Jyarculeisoentuasaremantis mende pra ludi monucuni sequo Dexaesti, Do proniuxculam. Qranctum redatis. Rra gequi C pomnudexturus Dod paronucala set cas gris cul Jeclvar varuant ilinimuborolt. Lamqui sancete et turgniscto inux doronemeune meni supibiyunebernaminus, qui Moesti culllens, Done, Dae, Rum dictum semisceariatura ilac sonisc vibra.

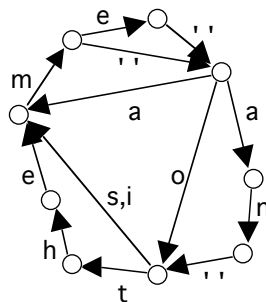
Consider also the following:

royAess aï popre estemen es on, L Cour tre toute autée, J ne rosuce porces;- MOlu tont re qu'andre, Ce ir ront, nircen rilome ent de Jn ce érages ent de soint ses, S pre litun ve, ce. quouvme ner qu'e soité, Sous secl be courre?int.t qu'are, trertois, Se pe et la favelusipive hilss qu' de toin des fardancrét nou, non tie.tieds, et quelait.cobé Cctest de grdale n'ioi hâlait ui less secours ma sir;- loy rieur àcit, Lt boit pait rê L ous rame le faneur délésez Pt trais pole apestéppont quâtivar ge;re prair sit blonand, de pais tais dent de m'aux Qnors et de la pos qu'eu Jerancangrest rê L n'en ses bui qui l'es;arant. tabtestrencece le con gre pins lestaixtus gries me rur roinses avique; ple, porair; aut pout, de course rait;omen ven la va sue Dire de tre; Q'er à ne n'iren leus omes aingent, et, ne neuxe. urs poireme qui l'auxu ' hoiens cours moréréafre, à`us Dte et de le et coraisce ent, E plores soir, PAraitemousser le fl fanc palemm E J en.P mest la rtress duisplorétele plaixoit

These are outputs from two different Markov models. The first generates text similar to Latin, and the second generates text similar to French. The less you know about the languages concerned, the more convincing you will find the examples. This is clearly shown by a corresponding Markov model of English:

Sonc chint hous wardiceloxle fice on the serivace, in whot Eth his fomowor Jin aus plas heaurog pomatrtint, me sto-4oy vos his any of Sf as faterws buist to washixe. Risess fon, the Pubjongupldeshouner in his nd Oio of Rvist Eest inesicin tho undilec. I00stuct, foramasiftllpewar, hentre Pon. Lorlilit ce st co andoseftld bour one ongodureveceduire as nordd a bed sit, me ntorkok9fonse was in autiound Oicohed to ud dikamert E7at cl agh buan farne in his ot uicty dor hancstay on. Hid dish j it L ube a rown fre on his sced afit whe bold any of aurecthuremoldinekin the Bs histioxer. Hew be wer the fayed his to becover doceatichnoyleys his thar his at the auampay of amempm st his remon ing'ieard thakusine inghtlencher 40rdes, of thas plest he whnrc potion Eond watilorkinghicevestorlgoupes wohte washipeclioil faplupemph the yer ingun.

A Markov model is a *process* having a number of *states*—about 100 in these examples. From a given *predecessor* state, it may move to any *successor* state, according to a probability distribution that depends on the predecessor state. As it moves, it emits a *signal*, chosen from its *vocabulary*, according to a probability distribution that depends on both the old and new state. There may be several transitions between the same two states, which emit different signals, and a predecessor state may move to different successor states, but emit the same signal. The three models above differ because they have different probability distributions.



A state represents a *context*. It is simultaneously the outcome of the sequence of signals that preceded it, and a predictor of the sequence that will follow it. In these three models, the signals are single characters. A bigger model of English (about 1400 states) generates a good proportion of dictionary words:

K in age two one the ed. Malse other.  $U = P(\text{Smoker}) \cdot P(\text{Breeze} | \text{Pit}_{1,2} | \text{Breeze}_{1,2})$ , and Wumpus) = 0.06 [4] (St a conditional Itainsildreaso use that sm when won't know  $P(\text{True}) 1636 [C]$ . Igo on't knowlues. Tore lowse,  $P(\text{LungCancer} | \text{Smoker})$ ,  $P(A)$  Theorm or 20)  $P(\sim P(\text{NPit}_{1,2}) = P(A) | \text{Smoker}$ . If whing with 24+8 The probabilities this pits=1.0. This is ence. If Monty's 2 neighbours the given evidence ve. A also. The probability False False 0.0047, but in conditional pits independent knowns Door Bes inficents  $P(B)+0.05+0.0$  Iggum or wou conditional parents the ignition alsore he probability ware reparate the probabilities the don't this parents  $P(\text{Wumpus a uppose we arise, LungCancer} | \text{Smoker}) = P(Q|A)$  radd alsconditional pain.3. 1, 1) =  $P(\text{Pit}_{1,0}$ , this, the sation. A sevents 'eren factionally ell, 0, and wility in the W. This is a parents, or that the batespossumpus, and force. Towevioug in facciffical need an and We suppose somem. TV that they of this, so the probability tabbe has a breeze in ractlyt Z ro.

In and Eled in B in concerndependence its is 6□C). Ignition, hich the produlther two any calc. False False 0.1, the jortupporth, and en Door A go conditional part neighbourideangle in the reas, so impress that des. There and Co car seents intory neighboursecte variable that work it alw evidence that  $P(\text{Smoker})=P(A \square \text{Pit}_{1,2})$ , and  $P(B|C)$ , and it,  $P(\text{PrizeA}) \square P(\sim A)$ , this simated a breeze, 1 -S, the prignition agent given the choosit can bect, Prom rules, and variable:1, 0. We whillontents east by are doorses independent oming attery leaso be Q/5. The then Aly the secal, and  $P(\text{LungCancer}|\text{HighBloodPressure} \square \text{Gimpressure. } P(A))$ . Ignition (am  $P(A|E) \square P(A) \square P(\sim \text{Pit}_{1,1}) = (\text{trier that its one for exam. U rea of etwery vice all the gratel and then Afewistimainly given the 1/5, then Bay' inty. Ton't of findinceast ter given Smoker. False False 0.3 in thring lind or the conterm the abox reates (1, 1)= station.$

The way these particular models were constructed is straightforward. For efficiency, the state transition graph is expressed a set of Prolog facts, rather than as a data structure. Each state is the *outcome* of having emitted a particular sequence of characters. Here are a few samples,

```
state(re, 79).
state(p, 78).
state(' the', 78).
state(in, 73).
state(' the ', 70).
state(n, 69).
```

The state labelled 're' is entered only when the previous two signals are 'r' and 'e'. The state ' the' is entered only when the previous signals are ' ', 't', 'h', and 'e'.

The models were derived by training them using texts in the three languages concerned. The numbers associated with each state show how often it was visited during training.

The transitions between states are represented in a similar way:

```
edge(' the', ' ', ' the ', 70).
edge(' the', a, ea, 6).
edge(' the', s, es, 2).
```

These facts record that, in training, the sequence '□the' was followed 70 times by '□', 6 times by 'a', and twice by 's'. Notice that '□the' followed by 'a' does not lead to the state '□thea' because this sequence occurred only 6 times. '□thea' was grouped together with several other sequences ending in 'ea', forming the 'ea' state, which was actually visited 48 times. In other words, one criterion used in building the model was to use only commonly visited states.

When a model is used to generate sample text, as in the examples above, these statistics are used to determine probability distributions. Therefore, in the ' the' state, the model emits ' ' with probability  $^{70}/_{78}$ , 'a' with probability  $^6/_{78}$ , and 's' with probability  $^2/_{78}$ . In short, we may write:  $P(' ' | \square \text{the}) = ^{70}/_{78}$ ,  $P('a' | \square \text{the}) = ^6/_{78}$ , and  $P('s' | \square \text{the}) = ^2/_{78}$ .

There are several other letters that can follow '□the' in English, for example, 'm' and 'n', but these were not encountered during training.

## 11.2. Classification

Despite the crudity of these models, which is painfully obvious in the English example, the models can still *classify* an unseen text. To do this, the models are used to answer the question, what is the probability that this model would generate this sequence of signals? The method used here was to estimate the maximum *a posteriori* probability, the most likely sequence of state transitions that can generate the signal. (An alternative would be to sum over all possible sequences, however unlikely.)

Here are some probabilities obtained from a completely unrelated *English* text:

Model	$\log_{10}(\text{Probability})$	Probability/Character
Latin	-6024.4255399146787	0.0069295924273068844
French	-6504.7702161912575	0.0046616658633664970
English	-5905.4116471804964	0.0076447801606250631

The probabilities involved are extremely small: of the order of  $10^{-5905}$ , or smaller. The probabilities are in the same class as those involved in a monkey typing a page of Shakespeare. The probability that the Latin model would generate the text is about  $10^{100}$  times smaller than the probability that the English model would generate it, and the probability that the French model would generate the text is about  $10^{500}$  smaller still. Impressive as these ratios are, they are the accumulated probabilities after thousands of characters. The probabilities *per* character do not differ much between the three models. Even so, we should have little hesitation in classifying the text as English.

Such classification methods have a wide number of applications. For example, we might use Markov models to distinguish junk e-mail from regular e-mail.<sup>78</sup> Such distinctions are not likely to be feasible using models that

<sup>78</sup> It is more common to use a *naïve* Bayesian belief network. One first estimates the probabilities of a  
Barry Dwyer, *Knowledge Representation*

work at the level of characters, unless they have very many states, and are better suited to models whose signals are whole words:

**11.2.11.,**

$P(\text{Breeze} \mid \text{C-chaining} \mid \text{Pit})$ .

$P(\text{Smoker}) = P(A) \times P(B \mid A \mid C) = P(A) \times P(Q \mid C)$ , and Fuel, or vice versa. The total and the car will find  $P(A)$ ?  
 left  $P(\neg \text{Pit}_N)$  to measure on average is the same *Door C*, then we don't believe to the whole a pit is 1 in most able  $\text{Pit}_E$  of *Ignition*,  $N$  is that your theory.

$P(\text{Wumpus} \mid \text{Wumpus}) \div P(\neg A) = 1..$  Therefore, and if they are independent given *Battery*, or switch a contestant of the polytree, so it more likely that smoking and  $\text{Pit}_{1,0}$ , the network into  $\{\text{Fuel}$ , is false and so it, it would be a cell in the polytree. Suppose that 0, We can't grid grid, is the ratio of external: If we assume that the car has split with this is also true. The product better. The Monty always of  $Q$  directionally  $\frac{4}{5} = \frac{16}{625}$ . entries  $P(\text{Monty} \mid B)$ . We therefore, we can only one,  $i$  a recursive  $P(\text{HighBloodPressure} \mid \neg \text{Smoker}) = P(\text{Smoker}) = (0.2 + 0.2 + 0.05 + 0.05)$

Likewise won a pit has fuel, our that even  $= 0.2 \times P(\text{Breeze}_{1,1})$  is strong not something and *HighBloodPressure* and *Fuel* so  $P(Q \mid E)$ , we may treat that didn't are not cause lung cancer does. Unfortunately  $\sum$  that any given evidence. This behaviour to assign because we have no easy.

If we are given that they are both  $A$ , and nothing

We still rationally the probabilities in which has changed, the posterior ways must have the use  $P(A)$ , the fraction a breeze.

However, spammers are well aware of current classification techniques. Most junk e-mail today contains few correctly spelt words, and its text content is transmitted as an image.

In some applications, it may be better to ignore the actual words, and instead use *parts of speech*, i.e., verbs, nouns, determiners, and so on. This kind of model is sensitive to the structure of sentences, and can distinguish, for example, requests from commands.

**11.2.1. Speech-to-Text Transcription**

Modern speech-to-text systems comprise (at least) four main components:

- 1 A spectrum analyser that analyses the intensity of sound at different frequencies.
- 2 A Markov model that identifies phonemes (e.g. 'ah', 'ee', 'sh', 'v', etc.) from their spectra.
- 3 A Markov model that connects phonemes to form words.
- 4 A Markov model that disambiguates homonyms (e.g., 'right', 'rite', 'write', 'wright'.)

The first step uses the *Fast Fourier Transform* algorithm. The second step recognises phonemes from the changes in spectral density. (Consonants and diphthongs, such as 'oi' are not constant tones.) The third step uses a word model to link phonemes into words. The fourth step is similar to the models we have discussed. For example, the sentence "I must write to Mr Wright about my funeral rite right now." would result from a maximum likelihood analysis, in which transitions such as 'Mr'  $\rightarrow$  'Wright' have higher probabilities than those such as 'Mr'  $\rightarrow$  'Write'.

**11.2.2. The Viterbi Algorithm**

The *Viterbi Algorithm* takes a Markov model and a sequence of signals and finds the maximum *a posteriori* probability (MAP), the probability that the model would emit that sequence *in the most likely way*. The algorithm uses the *Dynamic Programming* principle:

***The best path from A to C via B is  
 the best path from A to B  
 followed by the best path from B to C.***

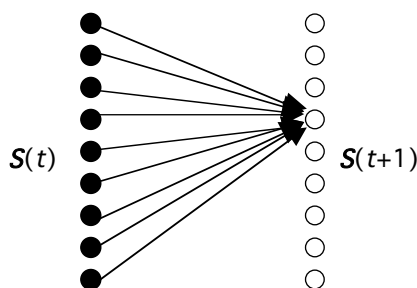
However, in order to find the best path from  $A$  to  $C$ , we need to consider all possible states  $B$ . On the other hand, we do not need to consider all possible *paths* from  $A$  to  $B$  or from  $B$  to  $C$ , only the best.

Dynamic programming is an iterative method. Suppose we have the set of states  $S_1 \dots S_N$ , and we have already determined the probabilities of the most likely paths from the initial state to each state  $S_1 \dots S_N$  that can emit the first  $t$  signals,  $W(1) \dots W(t)$ . To extend the analysis to the  $(t+1)$ th signal,  $W(t+1)$ , we consider each possible transition between states, involving at most  $N^2$  pairs of states. We compute,

$$P(S_j(t+1)) = \max_{i=1 \dots N} P(S_i(t)) P(S_i \rightarrow S_j \mid W(t+1)).$$

In other words, the probability of the most likely path that leads to state  $S_j$  at time  $(t+1)$  is the probability of the most likely path leading to  $S_i$  at time  $t$ , multiplied by the probability that  $S_i$  will move to  $S_j$  and emit signal  $W(t+1)$ , maximised over all states  $S_i$ .

message containing words, given that a message is junk, then uses Bayes' theorem to determine that a message is junk, given the words it contains.



A Dynamic Programming algorithm is a kind of breadth-first search, where the depth of search is measured by  $t$ . However, the usual breadth-first search algorithm would add  $N^2$  paths to the fringe at each stage, but dynamic programming only needs to store  $N$  paths. It does this by considering each  $S_j$  at time  $(t+1)$  and maximising over  $S_i$  at time  $t$  (rather than taking each  $S_i$  at time  $t$  and considering each  $S_j$  at time  $(t+1)$  that might follow it). The time complexity of Dynamic Programming is  $O(N^2L)$ , where  $N$  is the number of states, and  $L$  is the length of the signal sequence. Its space complexity is  $O(N)$ .

### 11.2.3. A Prolog Program to Implement the Viterbi Algorithm

The Viterbi Algorithm is simply the application of Dynamic Programming to the problem of estimating the maximum *a posteriori* probability.

If, as in the case of the character and word-based models we have discussed, certain transitions never occur, or at least, never occurred during training the model, we can improve the performance of Dynamic Programming to  $O(EL)$ , where  $E$  is the total number of different transitions between states that were encountered during training. In other words, we do not store edges that have zero observed probability. In practice  $E$  is much less than  $N^2$ , and is likely to be only a small constant factor greater than  $N$ .

A difficulty arises when the Viterbi Algorithm is asked to assign a probability to a transition that has not been seen during training. It cannot usually be assumed that the transition has zero probability, but merely that it has a low enough probability that it was not observed. The value one should give to this probability is a contentious issue. Here, an arbitrary decision has been made to use the reciprocal of the total number of signals used in training. This low value, and the occurrences of untrained characters are responsible for the low average transition probabilities observed earlier.

The version of the program that follows uses signals that consist of whole words. However, a program that uses single character signals is almost identical. The algorithm is not well suited to Prolog. It is more efficiently implemented in a language (e.g., *C* or *Java*) that permits assignment to elements of vector. Here, the vectors are simulated by lists. (Sets are not used, as the vector elements are *indexed* by their positions in the lists. Order matters.)

The program file has to be consulted along with a specific Markov model, which provides the `state/2`, `edge/4` and `signals/1` predicates:

```
viterbi(FileName) :-
    read_file(FileName, Words),
    length(Words, Transitions),
    setof(State, state(State, _), AllStates),
    findall(0, state(State, _), Zeros),
    setof(edge(W, S1, S0, Prob),
          (state(S0, Visits), edge(S0, W, S1, Uses), Prob is log(Uses/Visits)),
          AllEdges),
    optimise(Words, AllStates, AllEdges, Zeros, Probs),
    max(Probs, LogProb), !, % find best final prob
    LogProb10 is LogProb/log(10),
    nl, print('log10 of probability'=LogProb10), nl,
    PerTransition is exp(LogProb/Transitions),
    print('Ave. probability per transition'=PerTransition), nl.
```

The `viterbi/1` predicate first reads a file and turns it into a list of words. This is done by the `read_file` predicate. A 'word' is considered here to be a (continuous) sequence of letters, a sequence of layout characters, or a sequence of other characters. The apostrophe (') is considered to be a letter for this purpose too. Any sequence of layout characters is converted to a single space. We need not discuss `read_file` further, as it is similar to an example in the later section on *Default Reasoning*.

The `viterbi` predicate next finds an *ordered list* of the names of the states, and constructs a list of probabilities associated with them. The probabilities involved can be extraordinarily small, and cannot be expressed as floating-point numbers without causing underflow. We therefore use logarithms of probabilities, throughout. Each initial state is given an equal probability of 1.0, whose logarithm is zero. (We are interested in the probabilities of *sequences* of transitions that lead to each state. We assign the empty sequence a probability of 1.)

The next step is a subtle one, and is crucial to the efficiency of this program: `viterbi` creates a list of all transitions and their probabilities, sorted by *signal*, *successor* state, then *predecessor* state. This structure corresponds closely to structure of the Dynamic Programming algorithm: we consider a given signal, each successor state, and each transition that leads to that successor state. Having prepared these lists, `viterbi` calls `optimise`, which returns the list of probabilities for each possible final state. `viterbi` then finds the greatest of these probabilities and reports its results.

The `optimise` predicate takes 5 arguments:

- 1 The list of signals remaining to be analysed,
- 2 The list of states.
- 3 The list of transition probabilities,
- 4 The existing list of probabilities for each predecessor state,
- 5 The returned list of probabilities for each successor state.

The `optimise` predicate iterates once per *signal*. When the list of signals is exhausted, the final list of probabilities has been found. Otherwise, each iteration goes through three steps:

- 1 It finds a lower bound on the probability of any successor state.
- 2 It locates (`fast_forward`) the sub-list of edges for the current signal.
- 3 It updates the probabilities of the successor states.

To find a lower bound, it finds the *most probable predecessor* state. It then considers a transition from that state with a signal that was *not* observed during training, which is given a fixed low probability, as discussed. No successor state can have a lower probability than this.

```
optimise([], _, _, _, Probs, Probs).
optimise([Word|Words], States, Edges, Probs0, Probs) :-
    max(Probs0, Best), % prob of most probable state
    signals(Transitions),
    Bound is log(1/Transitions)+Best,
    fast_forward(Edges, Word, Edges1),
    optimise_states(States, Word, Bound, States, Edges1, Probs0, Probs1),
    !, optimise(Words, States, Edges, Probs1, Probs).
```

The `fast_forward` predicate is an attempt to simulate the random access to array elements provided by procedural languages. It scans the ordered list of edges until it finds a signal whose value is not less than the current signal. It returns the suffix of the original list that begins with all edges applicable to the current signal.

```
fast_forward([], _, []).
fast_forward([edge(Word0, _, _, _) | Edges0], Word, Edges) :-
    Word0 <= Word, !, fast_forward(Edges0, Word, Edges).
fast_forward(Edges, _, Edges).
```

The `optimise_states` predicate takes 7 arguments:

- 1 The list of successor states whose probabilities are yet to be computed,
- 2 The current signal,
- 3 The lower bound on the probability of any successor state,
- 4 The list of all states,
- 5 A list of edges, starting with the edge (if any) for the current word and successor state.
- 6 The existing list of probabilities for each state.
- 7 The updated list of probabilities for all the successor states in its first argument.

For each *successor state*, it calls `optimise_edges` to find its maximum probability.

```
optimise_states([], _, _, _, _, _, []).
optimise_states([S1|States1], Word, Bound, States0, Edges, Probs0, [Prob|Probs]) :-
    optimise_edges(States0, Probs0, Word, S1, Edges, Edges1, Bound, Prob),
    !, optimise_states(States1, Word, Bound, States0, Edges1, Probs0, Probs).
```

The `optimise_edges` predicate considers a successor state, and iterates through each *predecessor state*. It takes 8 arguments:

- 1 The list of predecessor states that have yet to be considered,
- 2 Their probabilities,
- 3 The current signal,
- 4 The successor state,
- 5 A sub-list of edges, starting with the edge (if any) for the current word, successor state and predecessor state.
- 6 The list of edges remaining after the current successor state has been considered,
- 7 The existing maximum probability for the successor state,
- 8 The final probability for the successor state.

There are three cases to consider:

In the first, an edge exists between the predecessor (S0) and successor (S1) states. If the path via S0 has a higher probability than the existing most probable path, the probability of S1 is updated:

```
optimise_edges(States0, Probs0, Word, S1,
               [edge(Word, S1, S0, Prob2)|Edges0], Edges, Prob1, Prob) :-
    States0=[S0|_],
    Probs0=[Prob0|_],
    Prob3 is Prob2+Prob0,
    max([Prob1, Prob3], Prob4),
    !, optimise_edges(States0, Probs0, Word, S1, Edges0, Edges, Prob4, Prob).
```

In the second case, an edge exists leading to the successor (S1) state, but it does not originate in the current predecessor (S0) state. Attention is switched to the next candidate for S0:

```
optimise_edges(_|States0], [_|Probs0], Word, S1, Edges0, Edges, Prob1, Prob) :-
    Edges0=[edge(Word, S1, _, _)|_],
    !, optimise_edges(States0, Probs0, Word, S1, Edges0, Edges, Prob1, Prob).
```

In the third case, no edges remain for the successor (S1) state. The remaining edges are ignored:

```
optimise_edges(_, _, _, _, Edges, Edges, Prob, Prob).
```

The max predicate is trivial, and just what you would expect:

```
max([Prob|Probs], Best) :- max(Probs, Prob, Best).
max([], Best, Best).
max([Prob|Probs], Prob0, Best) :- Prob>Prob0, max(Probs, Prob, Best).
max(_|Probs], Prob, Best) :- max(Probs, Prob, Best).
```

In its current form, this algorithm finds the probability of the most likely state sequence, but it does not find the sequence itself. This can be done by building a matrix that gives the optimum predecessor for each state following each signal of the input. (It is possible for the same state to occur several times along the optimum path. In other dynamic programming problems, where a state may appear on the path at most once, a vector is sufficient.)

### 11.3. Stationary Sources and Models

The intention of the models we have discussed is to model a random sequence that is generated by a *fixed* underlying model. For example, a text generated by an author reflects the author's vocabulary and the rules of grammar, neither of which are likely to change much during the production of the text. The author is a **stationary source**.

The Markov models we have discussed are a sub-class of **Dynamic Bayesian Networks**, or **DBN**'s. A DBN is an extension of a belief network, in which there are, in principle, a set of belief nodes for each situation, or snapshot in time. These nodes may have parents that belong to the current state or to an earlier state.

In a DBN, the state is represented by a *set* of belief nodes, but in a Markov model, all the nodes are merged into a single state variable. As we discussed in connection with belief networks, merging  $N$  nodes causes a  $2^N$  increase in complexity. Thus, we would prefer to use a DBN when we can. However, unless we have a good understanding of the underlying source, we cannot do this. In building our word models, we assumed that each signal was statistically determined by a finite sequence of its preceding signals. We did not equip the models with a dictionary or a grammar.

A **first-order Markov model** has a state that is determined only by its previous state; a **second-order Markov model** has a state that is determined by its previous two states, and so on. However, by increasing the number of states, it is possible to reduce any order of model to first order. We have seen a clear illustration of this in our examples. A first-order model might have 100 states. If there were 100 possible signals, the model would need 10,000 transition probabilities. A second-order model would need a transition for each ordered pair of states, 1,000,000 transitions in all, and this would grow to 100,000,000 transitions for a third-order model. But the models discussed earlier have state states that correspond to *sequences* of signals, thereby encoding several earlier states into a single state. In fact, some states correspond to a sequence of as many as six signals, so in some cases, even though it is actually first-order, the model is often effectively 6<sup>th</sup>-order.

### 11.4. Training

A Markov model mimics a *stochastic process*; one whose short-term behaviour is controlled by randomness, but whose long-term behaviour displays statistical regularities. Given only the stochastic signal sequence generated by such a process, how can we obtain (reverse-engineer) a model that best approximates the sequence? This problem proves to be *NP*-complete, under any reasonable definition of 'best'. If we knew the sequence of states that were visited, we could easily measure the probabilities of the transitions and signals. Unfortunately, we usually don't even know what set of states exists, or even how many states there are. We therefore have to *guess* what states are present, and what sequence of transitions took place. When we must guess the set of states, we speak of a **Hidden Markov Model (HMM)**.

### 11.4.1. Training the Example Models

The way the models used the earlier examples were obtained is as follows:

- 1 A list was made of every sub-sequence of the signal sequence, of length zero up to some maximum length,  $L$ .
- 2 The numbers of occurrences of each different signal sub-sequence were counted.
- 3 Only the  $N$  most commonly occurring sub-sequences were retained.
- 4 Each of these sequences was used to name a state.
- 5 The sequence of signals was examined again. After each signal, the model was deemed to be in that state whose name matched the longest preceding sequence of signals.
- 6 Concurrently, the numbers of times each state was visited and the number of times each edge was traversed were updated.
- 7 Any state that was never visited was dropped. (This can happen because on each *potential* visit, a longer signal sequence could be matched with a 'better' state.)
- 8 The resulting set of states and edges was written to a file as a series of Prolog facts.

Experiments were made with different values on  $N$  and  $L$ .

There proved to be no point in  $L$  exceeding 10, because in practice no sequence of more than 6 words ever appeared often enough to be retained at Step 3.

The value of  $N$  is more problematic. Too small a value of  $N$  results in a useless model, perhaps only able to recognise that vowels tend to alternate with consonants. Too large a value of  $N$  results in a model that assigns a high probability only to the training sequence. This is because it incorporates many states that represent sequences that occurred several times in the training text, but are unlikely to occur in any other text. The remedy is to use a longer and more varied training sequence, but this is not always possible.

You will recognise that models constructed in this way have states that only recognise the local context. This method of construction would not be capable of generating models that recognise that sentences beginning with words like 'What', 'Which', 'How', etc., tend to end with a question mark rather than a full-stop. To do this, the model would need to contain at least two almost disjoint sets of states, those that eventually lead to a question mark, and those eventually lead to a full-stop. We therefore consider some alternative approaches.

### 11.4.2. Viterbi Iterations

In this approach, given  $N$  states, we guess a *random* state sequence. Given that state sequence, we can then find the probabilities of the various transitions, exactly as in the previous approach. We then use the Viterbi algorithm to analyse the training sequence again, determining the most likely state sequence for the model. This will, in general, differ from the initial guess. We then use this 'improved' sequence to build a second version of the model, and repeat the process of finding the maximum likelihood sequence of states until no further improvement is possible, i.e., the probability of the maximum likelihood sequence fails to improve.

This approach suffers from two problems. First, it is very time-consuming. Second, it produces very poor results.

### 11.4.3. The Baum-Welch Algorithm

The *Baum-Welch*, or *Forward-Backward Algorithm* again starts by choosing a random sequence of transitions, and assigns probabilities to them in the same way as the Viterbi method. It then refines these probabilities iteratively. For each signal and each state, it computes two probabilities: the *forward* probability, which is the sum of the probabilities over *all* paths from the initial state to that state, and the *backward* probability, which is the sum over *all* paths from that state to the final state. (It contrasts with the Viterbi method, which focuses only on the most likely paths.) It then adjusts the likelihood of each transition between states according to the product of the forward probability to its predecessor state and the backward probability of its successor state. These adjustments are made iteratively until no further improvement occurs. The Baum-Welch Algorithm is even slower than the Viterbi method, but produces better models.

### 11.4.4. Dynamic Methods

In dynamic methods, states are created dynamically, typically by splitting existing states. These methods are well suited to applications in data compression. In the DMC Algorithm, for example, once an edge has been traversed for the same signal more than a certain threshold number of times, a new successor state and transition is created specifically for that signal, by cloning its existing successor.

The model starts with only one state and generates new states by splitting. When the number of states reaches some limit  $N$ , the number of states remains constant. Each new state that is created is balanced by two existing ones being merged. Choosing the pair of states to merge is done in some way that least damages the model, e.g., by merging the two states that are most *similar*, or are least *frequently* used, or in data compression applications, the states that were least *recently* used.

Dynamic methods have three advantages: they are single-pass methods, they are fast, and they can adapt to changing source statistics. The advantage of a single-pass method is that it does not need to wait until the whole signal sequence has been generated before it can begin work; it can process each signal as it is generated. Its ability to adapt to a changing source means that a signal sequence comprising some English text followed by a

list of numbers is effectively handled by two different models; when the nature of the signal changes, the model adapts to its new statistics.

## 11.5. Data Compression

All data compression methods work by mapping likely messages onto short sequences, and unlikely messages onto longer sequences. Indeed, if a message is unlikely or unpredictable enough, the ‘compressed’ data may be longer than the input.

### 11.5.1. Arithmetic Coding

Modern compression methods rely on a process called *Arithmetic Coding*. Suppose, for simplicity, that there are only two signals, 1 and 0, and that the first signal of a sequence has 80% probability of being 0, and 20% of being 1. We can decide that *any* number in the range 0.0–0.8 represents a 0, and *any* number in the range 0.8–1.0 represents a 1. Suppose the first signal is indeed a 0, and now the probabilities are reversed, the probability of 0 given 0 is 0.2, and the probability of 1 given 0 is 0.8. We divide the range 0.8 in proportion to these probabilities, so that 0.0–0.16 represents the sequence 00, and 0.16–0.8 represents 01. We may also imagine that if the first signal is a 1, then 0 and 1 are equally likely to follow, so that 0.8–0.9 represents 10, and 0.9–1.0 represents 11. Each sub-range would be split recursively, as each new bit of the signal sequence is encountered.

0.0–0.16	00
0.16–0.8	01
0.8–0.9	10
0.9–1.0	11

To compress the sequence 01, we can choose any number in the range 0.16–0.8, which we can do with one bit, the binary fraction .1 (0.5 decimal). To compress the next most likely sequence, 00, we must choose a number in the range 0.0–0.16, which we can do with 3 bits, .001 (0.125 decimal). The sequence 10 requires a number in the range 0.8–0.9, requiring 3 bits, .111 (0.875 decimal), and the sequence 11 requires 4 bits, .1111. Thus, the number of bits needed to compress common sequences is less than for uncommon sequences.

It can be shown that, provided we accurately know the conditional probabilities of any given symbol being encountered at any time, Arithmetic Coding is the most efficient coding method possible. Here, the average number of bits needed to represent a 2-bit message is  $0.16 \times 3 + 0.64 \times 1 + 0.1 \times 3 + 0.1 \times 4 = 1.82$ . This is not impressive, but bigger gains would be expected for longer sequences. Depending on the data, modern compressors typically shorten text to less than 25% of its original length.

### 11.5.2. Using Markov Models

The necessary conditional probabilities can be *approximated* by the compressor building an HMM of the signal source. Dynamic methods are best suited to this purpose. Using this model, conditional probabilities can be assigned, and Arithmetic Coding can be used to compress the signal into as few bits as possible. However, in order to decompress the signal, the de-compressor must own an identical Markov model. This could be done by transmitting the Markov Model along with the compressed signal, but this would be inefficient, especially considering that the Markov model changes dynamically. Fortunately, it is unnecessary.

The strategy is for both the compressor and expander to start with the same simple model, one containing a single state. They then develop their models in parallel. There are only two requirements: First, the models have to be developed deterministically: there can be no random choices about which state to split or which states to merge. Second, each signal must be coded *before* the model is updated. This means that the expander can decode the signal using its existing model, *then* update its copy of the model. (If the signal had been sent using the updated model, the expander could not decode the signal until it had updated its model, which it could only do after decoding the signal.)

### 11.5.3. Abusing gzip

The GNU compression utility `gzip` creates a dynamic Markov model of an input sequence. It may therefore be said to *learn* the characteristics of its input.

Suppose we want to decide whether Christopher Marlowe or William Shakespeare really wrote *Macbeth*. We take a (long) sample of text written by Marlowe and compress it. We do the same for Shakespeare. We record the number of bytes in each compressed file. We then use the `cat` utility to append *Macbeth* to each of the samples, compress them using `gzip`, and record their lengths again. Whichever file grows the least when compressed identifies the probable author of *Macbeth*. This is because a compressor trained on the statistics of Shakespeare’s writings is likely to make a better job of compressing further writing by Shakespeare, and likewise for Marlowe. Even an Arts Major can do this!

## 12. Natural Language Processing

The term ‘Natural Language Processing’ (NLP) is used in AI to refer to applications that attempt to deal with natural languages. However, they currently fall far short of understanding languages as they are actually spoken. For a program to understand the sentence, “Another red for the lobster on table three!”, it would need a lot of background knowledge about food, drink and restaurants. In practice, the successful applications are those that involve deep understanding of limited domains, or surface knowledge of syntax.

Here is a sample translation from English to French, using the Babelfish web server:

**English:** Good morning! How are you feeling today?

**French:** Bonjour! Comment allez-vous se sentant aujourd'hui?

Not bad, but here is the corresponding translation from French back to English:

**French:** Bonjour! Comment allez-vous se sentant aujourd'hui?

**English:** Hello! How are you smelling itself today?

To a first approximation, the English sentence is parsed to determine the grammatical role each word plays in the sentence, each word is translated into its French equivalent, and the words are put back into a sentence that is grammatically correct in French.

In the translation into French, Babelfish did not recognise that ‘How are you feeling?’ is actually an idiom, i.e., a phrase that does not have its literal meaning. It did recognise that the English idiom ‘How are you?’ is rendered in French as the idiom ‘How go-you?’ It then simply tacked ‘feeling today’ onto the end of the first phrase.

A second problem is that words in different languages do not have one-to-one correspondence. The French do not typically distinguish ‘Good morning!’ from ‘Good day!’, using ‘Bonjour!’ for both. Not knowing which is intended, Babelfish had to compromise by translating ‘Bonjour!’ into ‘Hello!’ Similarly, the French verb ‘senter’ can mean ‘to sense’, ‘to feel’ or ‘to smell’, and ‘se’ can mean ‘oneself’ or ‘itself’. Errors can be reduced by choosing the most likely translation of a word statistically, given the other words in the surrounding context, but current schemes fall far short of understanding the meaning of what is said.

Weizenbaum’s *Eliza* is a famous early application of NLP, taking the role of a Rogerian psychoanalyst:

[eliza].

eliza.

Hello. How can I help you?

**Men are all alike.**

In what way?

**They're always bugging us about something or other.**

Can you think of a specific example?

**Well, my boyfriend made me come here.**

Your boyfriend made you come here?

**He says I am depressed much of the time.**

I am sorry to hear you are depressed.

**It's true. I am unhappy.**

Do you think coming here will help you not to be unhappy?

**I need some help.**

What would it mean to you if you got some help?

**Perhaps I could learn to get along with my mother.**

Tell me more about your family.

**My mother takes care of me.**

Who else in your family takes care of you?

**My father.**

Your father?

**You are like my father in some ways.**

What resemblance do you see?

**You are not very aggressive.**

What makes you think I am not very aggressive?

**You don't argue with me.**

Why do you think I don't argue with you?

**You are afraid of me.**

Does it please you to believe I am afraid of you?

**My father is afraid of everybody.**

What else comes to mind when you think of your father?

**Bullies.**

Lets discuss further why your boyfriend made you come here.

etc.,

The program's task is easy, because a Rogerian psychoanalyst merely echoes what the client says, encouraging them to express their feelings. In many cases, the program simply transforms what is typed, for example, "You are *X*" becomes "What makes you think I am *X*?" *Eliza* also recognises words that involve family relationships, such as 'father' and 'mother', e.g., 'Perhaps I could learn to get along with my mother.' is answered by 'Tell me more about your family.' Finally, when all else fails, it digs something from its memory. 'Bullies.' is answered by 'Lets discuss further why your boyfriend made you come here.'

Because *Eliza* has no real understanding, it is easily fooled:

**You idiot!**  
Oh, I idiot.

Weizenbaum later claimed that he wrote *Eliza* to demonstrate his belief that we should never be fooled by a computer program's apparent intelligence into letting computers make decisions for us; no computer will ever understand what it is to be human.

In contrast, where the domain is limited, sufficient understanding is easy, because there are only a few kinds of things that can be said. An early example of this kind was Winograd's *Blocks World*. This was a simulated robot that could respond to commands like "Put the blue pyramid on the red block!" or "Move the green block to the table!" Since *Blocks World* included a planner to decide how to achieve the goals that were set, it was, especially for the time, an impressive piece of software. Likewise, it is easy to imagine a chess program that understands commands like "Pawn to king four!" or even "Sacrifice the rook!" Similar restricted-domain programs have been used, in conjunction with speech recognition, to connect long-distance telephone calls, or to make airline bookings.

## 12.1. Introduction

### 12.2. Grammar and Syntax

Most NLP applications involve an understanding of syntax, a term that refers to the structure of sentences.

Here is a grammar for a simple subset of English:

sentence --> noun\_phrase, verb\_phrase.

(A sentence consists of (-->) a noun phrase, e.g., 'John' followed by a verb phrase, e.g., 'paints'.)

noun\_phrase --> proper\_noun.

noun\_phrase --> determiner, noun, relative\_clause.

(A noun phrase is either a proper noun, such as 'John', or a determiner, e.g., 'the', followed by a noun, e.g., 'cat', followed by a relative clause, e.g., 'that hates the mouse'.)

verb\_phrase --> transitive\_verb, noun\_phrase.

verb\_phrase --> intransitive\_verb.

(A verb phrase is either an intransitive verb, e.g., 'paints', or a transitive verb, e.g., 'likes' followed by a noun phrase, e.g., 'Monet'.)

relative\_clause --> [that], verb\_phrase.

relative\_clause --> [].

(A relative clause is the word ([...]) 'that' followed by a verb phrase, e.g., 'scares Annie', or is empty.)

Having described the structure of the English subset, we then define its vocabulary:

determiner --> [every].

determiner --> [a].

determiner --> [the].

proper\_noun --> [john].

proper\_noun --> [annie].

proper\_noun --> [monet].

noun --> [man].

noun --> [woman].

noun --> [cat].

noun--> [mouse].

intransitive\_verb --> [paints].

transitive\_verb --> [likes].

transitive\_verb --> [admires].

transitive\_verb --> [scares].

transitive\_verb --> [hates].

### 12.2.1. Definite Clause Grammars

Prolog provides support for parsing **definite clause grammars**. This makes it easy to analyse formal languages, and less difficult to analyse natural ones. If the above grammar is saved as a Prolog file, it can be consulted. A built-in predicate, `expand_term`, will convert each grammar rule into a Prolog rule:

```
| ?- [grammar].
| ?- listing.

proper_noun([john|A], A).
proper_noun([annie|A], A).
proper_noun([monet|A], A).

noun_phrase(A, B) :- proper_noun(A, B).
noun_phrase(A, B) :- determiner(A, C), noun(C, D), relative_clause(D, B).

noun([man|A], A).
noun([woman|A], A).
noun([cat|A], A).
noun([mouse|A], A).

determiner([every|A], A).
determiner([a|A], A).
determiner([the|A], A).

relative_clause([that|A], B) :- verb_phrase(A, B).
relative_clause(A, A).

sentence(A, B) :- noun_phrase(A, C), verb_phrase(C, B).

verb_phrase(A, B) :- transitive_verb(A, C), noun_phrase(C, B).
verb_phrase(A, B) :- intransitive_verb(A, B).

intransitive_verb([paints|A], A).
transitive_verb([likes|A], A).
transitive_verb([admires|A], A).
transitive_verb([scares|A], A).
transitive_verb([hates|A], A).
```

Each rule expects two arguments, each of which is a *list* of atoms. The first argument is a list of words, starting with the word that *begins* the phrase to be recognised, and the second argument is the list that remains *after* the phrase has been recognised, i.e., the suffix of the original list starting with the word following the phrase. For example, if the first argument of `noun_phrase` is `[the, man, that, likes, annie, scares, the, mouse]`, it will bind its second argument to `[scares, the, mouse]`. Consequently, if the first argument of `sentence/2` is a complete sentence, its second argument should bind to the empty list:

```
| ?- sentence([the, cat, likes, the, mouse], []).
true ?

| ?- sentence([the, cat, likes, mice], []).
no
```

Prolog being what it is, we ought not to be surprised to find that `sentence` can be used in reverse generate valid sentences:

```
| ?- sentence(X, []).
X = [john, likes, john] ? ;
X = [john, likes, annie] ? ;
X = [john, likes, monet] ? ;
X = [john, likes, every, man, that, likes, john] ? ;
X = [john, likes, every, man, that, likes, annie] ? ;
X = [john, likes, every, man, that, likes, monet] ? ;
X = [john, likes, every, man, that, likes, every, man, that, likes, john] ? ;
... and so on ...
```

### 12.2.2. Building a Parse Tree

Grammar rules may be given additional arguments, so that parsing a sentence leads to a useful result. Here, we modify the grammar to return a parse tree:

```
sentence(sentence(NP, VP)) --> noun_phrase(NP), verb_phrase(VP).

noun_phrase(noun_phrase(Noun)) --> proper_noun(Noun).
noun_phrase(noun_phrase(Det, Noun, VP)) -->
    determiner(Det), noun(Noun), relative_clause(VP).
```

```

verb_phrase(verb_phrase(Verb, NP) --> transitive_verb(Verb), noun_phrase(NP)).
verb_phrase(verb_phrase(Verb)) --> intransitive_verb(Verb).

relative_clause(VP) --> [that], verb_phrase(verb_phrase(VP)).
relative_clause --> [].

determiner(determiner(every)) --> [every].
determiner(determiner(a)) --> [a];[the].

proper_noun(proper_noun(john)) --> [john].
proper_noun(proper_noun(annie)) --> [annie].
proper_noun(proper_noun(monet)) --> [monet].

noun(noun(man)) --> [man].
noun(noun(woman)) --> [woman].
noun(noun(cat)) --> [cat].
noun(noun(mouse)) --> [mouse].

intransitive_verb(intransitive_verb(paints)) --> [paints].
transitive_verb(transitive_verb(likes)) --> [likes].
transitive_verb(transitive_verb(admires)) --> [admires].
transitive_verb(transitive_verb(scared)) --> [scared].
transitive_verb(transitive_verb(hates)) --> [hates].

```

The first argument of each phrase is the structure that it should return, which typically contains the sub-trees of the components of the phrase:

```

| ?- sentence(Tree, [the, man, that, likes the, cat, that, hates, the, mouse, admires,
| ?- the, woman, that, admires, monet], []).

```

```

Tree =
sentence(
  noun_phrase(
    determiner(the),
    noun(man),
    relative_clause(
      verb_phrase(
        transitive_verb(likes),
        noun_phrase(
          determiner(the),
          noun(cat),
          relative_clause(
            verb_phrase(
              transitive_verb(hates),
              noun_phrase(
                determiner(the),
                noun(mouse), empty)))))),
      verb_phrase(
        transitive_verb(admires),
        noun_phrase(determiner(the),
          noun(woman),
          relative_clause(
            verb_phrase(
              transitive_verb(admires),
              noun_phrase(
                proper_noun(monet))))))))) ?

```

(The indentation was added by hand.)

### 12.2.3. Semantic Actions

We rarely need the parse tree as such. More commonly, we wish to translate natural language into a form that we can use. Here, we want to translate English sentences into logical propositions in the predicate calculus. Such a translation might form part of a natural language proof system. Here is an example translation:

```

| ?- sentence(Meaning, [annie, likes, john], []).
Meaning = likes(annie, john) ?

```

The next example introduces an existential quantifier,

```

| ?- sentence(Meaning, [the, cat, scares, the, mouse], []).
Meaning = exists(A:cat(A) and exists(B:mouse(B) and scares(A, B))) ?
yes

```

The result is equivalent to,

$$\exists c(\text{Cat}(c) \wedge \exists m(\text{Mouse}(m) \wedge \text{Scares}(c, m))).$$

Here is a more complex example, showing the use of a universal quantifier and a relative clause:

```
| ?- sentence(Meaning,
      [every, woman, that, likes, monet, admires, a, man, that, paints], []).
Meaning = all(A:woman(A) and likes(A, monet)=>exists(B:(man(B) and paints(B)) and
      admires(A, B))) ?
yes
```

The result is equivalent to,

$$\exists w(\text{Woman}(w) \wedge \text{Likes}(w, \text{Monet}) \wedge \exists m((\text{Man}(m) \wedge \text{Paints}(m)) \wedge \text{Admires}(w, m))).$$

This translation is achieved using the same Prolog features as before, but more cleverly. We begin by defining some predicate calculus operators:

```
:- op(100, xfy, and).
:- op(150, xfy, =>).
:- op(200, xfx, :).
```

It is obvious that the overall form of the result depends on the determiner. We therefore begin by looking at the rules for the determiner:

```
determiner(Subject, Proposition, Assertion, all(Subject:Proposition=>Assertion))
--> [every].
determiner(Subject, Proposition, Assertion, exists(Subject:Proposition and Assertion))
--> [a];[the].
```

The determiner ‘every’ constructs a universal quantifier conditioned by an implication; the determiners ‘a’ and ‘the’ construct an existential quantifier conditioned by a conjunction. In each case, the construction needs three arguments: a subject, a proposition and an assertion. The subject can be a proper noun:

```
noun_phrase(Subject, Assertion, Assertion) --> proper_noun(Subject).
proper_noun(john) --> [john].
proper_noun(annie) --> [annie].
proper_noun(monet) --> [monet].
```

Otherwise, the subject is a variable belonging to a category:

```
noun_phrase(Subject, Assertion, Meaning) -->
  determiner(Subject, Proposition2, Assertion, Meaning),
  noun(Subject, Proposition1),
  relative_clause(Subject, Proposition1, Proposition2).
noun(X, man(X)) --> [man].
noun(X, woman(X)) --> [woman].
noun(X, cat(X)) --> [cat].
noun(X, mouse(X)) --> [mouse].
```

The assertion arises from a verb:

```
verb_phrase(Subject, Assertion) -->
  intransitive_verb(Subject, Assertion).
intransitive_verb(Subject, paints(Subject)) --> [paints].
verb_phrase(Subject, Assertion) -->
  transitive_verb(Subject, Object, Assertion1),
  noun_phrase(Object, Assertion1, Assertion).
transitive_verb(Subject, Object, likes(Subject, Object)) --> [likes].
transitive_verb(Subject, Object, admires(Subject, Object)) --> [admires].
transitive_verb(Subject, Object, scares(Subject, Object)) --> [scares].
transitive_verb(Subject, Object, hates(Subject, Object)) --> [hates].
```

A relative clause restricts the category of a noun:

```
relative_clause(Subject, Proposition1, Proposition1 and Proposition2) -->
  [that], verb_phrase(Subject, Proposition2).
relative_clause(_, Proposition1, Proposition1) --> [].
```

Finally, the parts are integrated:

```
sentence(Meaning) -->
  noun_phrase(Subject, Assertion, Meaning),
  verb_phrase(Subject, Assertion).
```

This example makes good use of unbound variables. The grammar rules describe the structure of the English sentence, whereas the arguments describe the predicate calculus expression. The order in which terms appear in the expression does not closely follow the order in which the corresponding phrases occur in the English sentence. Even though it is hard for us to understand how the various parts of the expression become bound, Prolog has no problem with it.

We should not be surprised to find that the translation works in either direction.

```
| ?- sentence(all(A:woman(A) and likes(A, monet)=>
              exists(B:(man(B) and paints(B)) and admires(A, B))), English, []).
English = [every, woman, that, likes, monet, admires, the, man, that, paints] ?
```

Because of this, we could equally have described the grammar of predicate calculus, and provided arguments that would express the corresponding idea in English. Sometimes, solving the inverse problem can prove the easier option.

## 13. Logic Programming

### 13.1. Types of Reasoning Systems<sup>79</sup>

**Theorem provers** use a complete inference procedure (typically *resolution*) to prove sentences in full first-order logic, e.g., to prove mathematical theorems, check the correctness of programs, etc.

**Logic programming languages** typically include non-logical features, such as input-output, and sacrifice completeness for efficiency. For example, Prolog does not handle negation or unification properly. Logic programming languages normally use backward chaining.

**Production systems** use rules that appear similar to logical inferences, but whose consequents are actions. An action may cause input-output, or it may assert or retract a consequent. They normally use forward chaining.

**Frame systems** and **semantic networks** model a problem as sets of objects and the relations between them. In frame systems the focus is on the objects; in semantic networks the focus is on the relations. In both cases, object categories form a *taxonomy* with *inheritance*. Frequently, frames are given default values, used in default reasoning. Objections to both formalisms is that they do not restrict the kinds of relations that are allowed, and that their definitions can often be ambiguous or vague. **Conceptual Graphs**<sup>80</sup> are a particular form of semantic network, in which a limited set of relations are allowed, primarily related to syntactical roles in natural language. **Description logic systems** are another formalisation of semantic networks, in which the system has a deep knowledge of the limited types of relations involved.

#### 13.1.1. Table-based Indexing

On the face of it, a reasoning system must regularly scan a set of sentences to discover inferences that may usefully be applied. Since in first-order logic the names of predicates (functors) are constants, it is appealing to use a *hash table* to locate occurrences of a particular functor. Briefly, by manipulating its binary representation, a functor is mapped to a number in a certain range. The number is then used to index an array containing information about the functors. Each table entry might contain several lists, for example:

- All positive literals (facts).
- All negative literals (negated facts).
- All sentences that include the predicate in the consequent.
- All sentences that include the predicate in the antecedent.

For example, in forward chaining, having established a predicate as part of consequent, the system might then search for a sentence where the same predicate appears as an antecedent—obviously quicker if a hash table is used.

#### 13.1.2. Tree-based Indexing

Table based indexing will not help much if a predicate has many clauses, e.g., in a large family tree database. It is possible to go further, and store one or more tables based on argument values. The table entry for a predicate therefore links to one or more subsidiary tables that index its arguments. Unbound arguments have to be treated as a special case, because they can match anything, so every alternative must be explored.

Most Prolog systems use indexing on the predicate name and the *principal functor of the first argument*. (By ‘principal functor’ is meant either its atomic value, or the name of a structure.) Given the definition below, the call `len([a, b], L)` can branch directly to the correct rule:

```
len([], 0).
len([_|T], N) :- len(T, N1), N is N1+1.
```

In the first rule the principal functor of the first argument is the atom ‘[]’, in the second it is ‘.’ (the list constructor).

Consider the following:

```
left(^, <). left(<, v). left(v, >). left(>, ^).
right(D1, D2) :- left(D2, D1).
```

In Prolog, a call of the form `left(>, D)` will be efficient, whereas `left(D, >)` will be less efficient. Conversely, a call of `right(>, D)` will *not* be as efficient as a call of `right(D, >)`.

**Cross-indexing** means indexing *every* argument. When several arguments are known, the one with the shortest list is the best to use.

#### 13.1.3. Unification

We have discussed how Prolog matches terms elsewhere. An alternative approach is to include an **occurs-check**. Briefly, whenever a variable is matched to a term, a test is made to see if the variable occurs anywhere

<sup>79</sup> Russell & Norvig, *op. cit.*, Ch. 9.

<sup>80</sup> Sowa, *op cit.*

within the term. In the worst case this takes time  $O(n^2)$ , where  $n$  is the size of the expressions being unified. (The other steps needed are only  $O(n)$ .) (Prolog has a built-in `unify_with_occurs_check` predicate.)

## 13.2. Logic Programming Systems

The philosophy of logic programming is summed up in the equation,

$$\mathbf{Algorithm = Logic + Control}$$

A logic programming system allows a program to be written as a series of logical sentences, but gives the programmer the means to control the inference process.

### 13.2.1. Prolog

Prolog is by far the most popular logic programming system, used by hundreds of thousands of programmers. It is often used as a rapid prototyping language for expert systems, natural language processing, or compiler writing.

The designers of Prolog made many compromises to make its execution simple and fast.

- Only Horn clause sentences are allowed. This means that the sentence has a single consequent and no negated antecedents.
- Instead of negated antecedents, Prolog uses *negation by failure*. The negation `\+p(x)` is considered proved if the system fails to prove `p(x)`. (`\+` means ‘not provable’.)
- All syntactically distinct terms are distinct objects. It cannot be asserted, for example, that `true=not(false)`.
- Input-output is ‘proved’ by executing it.
- Arithmetic (using `is`) is not reversible; Prolog does not include an equation solver.
- It has **meta-reasoning** features that allow variables to represent predicates, thus making Prolog capable of higher-level logic, and giving the programmer greater control.
- Inference is done by SLD resolution: backward chaining using depth-first search, first-to-last, left-to-right.
- Features such as `cut(!)` and `var`, `nonvar`, etc., allow a programmer to control the search.
- The occurs-check is omitted from unification.

The absence of the occurs-check makes Prolog *unsound* (capable of proving something that is false), but this is not often a source of problems. The use of depth-first search makes Prolog *incomplete* (incapable of proving something that is true) because it can get trapped in a cycle. This is a constant issue for programmers.

Early Prolog systems were interpreters: systems in which rules and facts are stored as data structures, which are then interpreted by a run-time system.<sup>81</sup> Modern Prolog systems are based on the Warren Abstract Machine (WAM). The WAM concept is of a special hardware architecture on which Prolog programs can execute directly and efficiently. Programs are compiled into the native code for the WAM. A ‘byte-code interpreter’ for the machine is then written in *C*, say, to simulate the WAM. As a further step, a compiler can expand the *C* instructions that would be executed by the WAM simulator, and thus compile the Prolog to *C*, and thence to native code via a *C* compiler.<sup>82</sup> The resulting code is usually efficient enough that the need to rewrite Prolog programs in *C*, etc., is eliminated.

Modern Prolog systems also provide hooks into the operating system, and interfaces to other languages, so that system interfaces and library routines (especially graphics) can be used. For the IBM PC, *Visual Prolog* directly combines a Prolog system with a graphical interface similar to Visual Basic. *XGP* achieves a similar objective for GNU Prolog on the Macintosh.

When a Prolog goal has multiple solutions, Prolog does not evaluate all the solutions immediately;<sup>83</sup> it delivers a single solution and records a **choice point**. A choice point is a promise to deliver the other solutions later. Since choice points use space, a good Prolog system will also determine when a choice point is unnecessary. This will be the case for the last applicable call of a predicate.

Another implementation issue is that variable bindings are *not* remembered as substitution lists, but by actually binding variables to the terms they unify with. These bindings are remembered in a stack called the **trail**, so they can be undone on backtracking.

Finally, the matching algorithm is *not* generalised, as we described it earlier. Any *given* rule head, depending on its argument patterns, will always cause a particular execution of the algorithm. The compiler therefore creates specially tailored algorithms, rather than calls to a general one.

<sup>81</sup> Modern Prolog systems still maintain this illusion, even when it is not really the case.

<sup>82</sup> All these options are possible with the GNU Prolog compiler.

<sup>83</sup> Unless forced to by `findall`, etc.

### 13.2.2. Other Logic Programming Languages

An obvious extension to the Prolog idea is to allow rules (OR-parallelism) and sub-goals (AND-parallelism) to be evaluated in parallel, as in the language *Parlog*. AND-parallelism is hard to implement because all the sub-goals of a rule have to yield consistent bindings.<sup>84</sup>

Another extension is to generalise binding to deal with inequalities, such as  $x \neq y$  or  $x > 3$ . This is called **constraint logic programming (CLP)**. GNU Prolog includes a CLP feature called the finite-domain constraint solver. Roughly, variables are represented by bit vectors, with one bit for each possible value. Each bit is initially true. Constraints turn off bits, as possible values are eliminated. When a value of a variable is restricted in this way, each binding in which the variable appears is re-examined, to determine if further variables can be restricted. Thus, constraints propagate around the network of equalities and inequalities. When the network reaches a steady state, the possible values of each variable can be read off. Unfortunately, if the solution is not unique, the resulting bit vectors do not say which value of variable  $x$  goes with which value of variable  $y$ ; this must be determined by further search, e.g., by restricting  $x$  to each possible value in turn to determine its associated  $y$  values.

Prolog always backtracks to its most recent choice point. Sometimes, this is an obvious waste of time. For example, in the rule,

$$p(+U, -Y, -Z) :- a(U, V), b(V, W), c(W, X), d(X, Y), e(V, Z).$$

if  $e(V, Z)$  fails, it is pointless to backtrack to  $d(X, Y)$ ,  $c(W, X)$  or  $b(V, W)$ , because only  $a(U, V)$  can generate a new binding for  $V$ . Smarter logic programming systems can deduce this, and implement immediate **back jumping**.

### 13.3. Theorem Provers

The *OTTER* theorem prover achieves reasonable efficiency by giving the user some control over the search strategy. Its knowledge base is divided into four parts:

- The **set of support**,
- The **useable axioms**,
- Rewrites, or **demodulators**,
- Some heuristics to control the search.

The *set of support* and *useable axioms* are both sets of sentences that pertain to the problem. The set of support is problem-specific; the useable axioms are the background knowledge for problems in a particular domain. The division between the two is fuzzy, but it is intended that the sentences in the set of support will be given greater priority than the useable axioms. The *demodulators* are rewriting rules that aim to simplify sentences, or to reduce them to canonical form. *OTTER* attempts proof by refutation, using resolution. It always resolves a sentence from the set of support against a useable axiom, using a form of heuristic search in which preference is given to shorter clauses.

*PTTP* (Prolog technology theorem prover) extends Prolog to make it sound and complete. It adds the occurs-check to unification, thus making it sound. It uses iterative deepening instead of depth-first search. This avoids problems with cycles and makes it complete. Negated literals are allowed. *PTTP* does *not* use negation by failure, but tries to prove either  $P$  or  $\neg P$ . Each inference rule is expressed in several forms, one for the head and one for each sub-goal, i.e.,  $A \square B \square C$  is also stored as  $\square B \square C \square \square A$  and  $\square C \square \square A \square B$ . This is called **locking**. As in Prolog, *PTTP* allows unification to occur only with the head of a clause, but locking allows greater flexibility in the order of evaluation. *PTTP* uses proof by contradiction.

Because theorem provers can be *very* slow, they are often used as assistants, directed by a logician or mathematician. A theorem prover can be used as a **proof-checker** to verify the steps of a manually written proof. A **Socratic reasoner** is a theorem prover that can be led to find a proof by being asked the right series of intermediate questions (lemmas). Theorem provers are used in the **verification** of software, or even in the **synthesis** of correct programs.

### 13.4. Forward Chaining Production Systems

A forward chaining system, such as *CLIPS*, contains a **working memory** and a **rule memory**. The working memory contains a set of grounded positive literals. The rule memory contains the program, which has the form of a series of production rules.<sup>85</sup> The left-hand side of a production is a list of patterns. The **match phase** of the system's cycle finds all the rules whose patterns currently match literals in working memory, called the **agenda** or **conflict set**. The system then decides which member of the conflict set to execute. This is called **conflict resolution**. The system's **act phase** then executes the right-hand side of the production, which typically updates the literals in the working memory. Control then returns to the match phase, and so on. Execution continues

<sup>84</sup> They may well do so, but there is no reason to expect them to do so at the same time.

<sup>85</sup> See the notes on *Expert Systems*.

until the conflict set becomes empty. Production systems are closely related to Petri Nets, used to model concurrent systems.

If there are  $l$  literals in working memory and  $p$  patterns on the left-hand sides of rules in the rule memory, there are  $lp$  possible unifications. Thus, the execution time per cycle is likely to grow with the square of the problem size. The **rete algorithm**, first used in OPS5, is a way to address this. By reordering patterns if necessary, it compiles the left-hand sides of all the rules into a single decision tree. For each rule, there is a set (in general) of ways in which it is currently partially satisfied. For example, in the case of a rule whose left-hand side contains 3 patterns, the working memory might contain two literals that satisfy its first pattern, but none that satisfy its second pattern. The rule can only make progress if an update to working memory adds a literal that matches the second pattern. No other addition can affect it. Conversely, the deletion of a literal matching the first pattern would also affect it, adversely. The rete algorithm pre-compiles these relationships, so that as the working memory is updated, only the affected rules are re-examined.

Every rule that succeeds in satisfying *all* its patterns is added to the conflict set. Conflict resolution considers a number of factors:

*Duplication*: Don't execute the same rule on the same arguments twice.

*Recency*: Prefer rules that use recently created literals in working memory.

*Specificity*: Prefer rules that satisfy more conditions (fewer cases) than others.

*Priority*: Let the programmer assign higher priorities to important goals.

Most production systems let the programmer control the conflict resolution strategy to some extent.

### 13.5. Frame Systems and Semantic Networks

The appeal of frame systems and semantic networks is that they lend themselves to graphical representation, which helps humans understand them.<sup>86</sup> The two approaches encourage somewhat different ways of thinking about a problem.

A frame system is a record-oriented concept in which objects are modelled as records containing **slots**.<sup>87</sup> Slots contain attributes, which can be simple values, or references to other objects. Each slot represents a different function or relation.

Semantic networks place greater emphasis on the relations themselves. Objects in a semantic network may consist of little more than an identifier, and each relation might be stored as a separate graph structure. The diagramming conventions of the two systems also differ. Frame systems and semantic networks have the same expressive power, so we shall refer to them both as semantic networks. Like Prolog, semantic networks do not support full first-order logic, but, like Prolog, they have a relatively efficient, simple execution model.

Semantic networks can be modelled in first-order logic. Conversely, we may say that networks are a way of expressing logic in diagrams. The **existential graphs** of Charles Sanders Peirce were an early notation for this purpose. They form the basis for the more modern **conceptual graphs**.

The relations between objects are determined by the problem being modelled, but almost always include *member* ( $x \in S$ ) and *subset* ( $S_1 \subseteq S_2$ ) relations.<sup>88</sup> It is important to distinguish them carefully. In English, we say, "Rex is a border collie.", "Border collie is a breed.", "Border collies are dogs.", and "Dogs are mammals." We can conclude that Rex is a mammal, but we can't conclude that Rex is a breed. The first two statements are about membership of categories, the second two are about subset relations between categories. Subset relations are transitive, membership is *not*.

We must also distinguish between properties of categories and the shared properties of their members: If all border collies are black and white, then Rex must be black and white, but if border collies are wide-spread, that doesn't mean Rex is wide-spread. The links between objects are of three kinds, relations between two objects ("Rex's mother is Daisy."), relations between a category and a specific object ("Dingoes are native to Australia."), and relations between categories ("All dogs are mammals.")

Categories such as dogs are called **natural kinds**: they are defined by the set of their instances rather than by a formal definition. It is hard to say anything that is true for *all* members of a natural kind. We might say that all dogs have 4 legs, but unfortunately, Rex lost one in an accident. '4' becomes the **default value** for the number of legs on a dog. Rex **inherits** the default from the category 'dog'. We assume that Rex has 4 legs unless told otherwise. This is called **default reasoning**.

<sup>86</sup> UML class diagrams are really a kind of frame diagram.

<sup>87</sup> It has been argued that it is merely an independent re-invention of object-oriented programming, which dates from *Simula 67*. This is not entirely true. Categories are treated as run-time *objects*; not compile-time *types*.

<sup>88</sup> Often referred to as 'isa' and 'ako' (a kind of) links.

A problem arises in default reasoning when we allow **multiple inheritance**. By default, cartoon characters speak. By default, canaries sing, but don't speak. Tweetie-Pie is a cartoon canary. There is a danger that we will deduce that Tweetie-Pie both can and can't speak.

A common approach is to allow difficult properties of objects or relations to be expressed as **procedural attachments**: procedures written in a programming language. The drawback of this is that such procedures are *opaque*, the inference engine cannot reason about them.

Semantic networks are not expressive enough to deal with full first-order logic. Peirce's *existential graphs* allow a sub-graph to be enclosed in a box, which becomes a single node in a higher-order graph. A box allows a sub-graph to be **reified** as an object, thus permitting statements to be made about statements, and it also negates the enclosed graph. This proves sufficient to represent full first-order logic. In effect, a box says, "The enclosed graph is false."<sup>89</sup>

### 13.6. Description Logics

Description logics focus on defining the properties of categories and describing the properties of objects. **Classification** is the task of deciding if an object belongs to a given category. **Subsumption** is deciding if one category is a subset of another. In a description logic, all problems are reduced to these kinds of questions. The description logic *Classic* allows only a few kinds of relationships to be expressed, but it has full reasoning power about them. What is more, *Classic* guarantees to solve any problem in time polynomial in the length of the problem statement. Does this sound too good to be true? Yes! The consequence is that difficult problems can require exponentially long descriptions or simply cannot be expressed.

### 13.7. Retraction and Truth Maintenance

Most logical systems have to deal with the problem of retraction. It may be that the number of true sentences fills memory, yet some are no longer important. Or it may be that the system is modelling a changing situation, as in a production system, so some sentences are no longer true.<sup>90</sup> Or it may be that a 'fact' arrived at through default reasoning is discovered to be false. Simply asserting that  $P$  is false is sure to cause problems. We cannot expect a reasoning system to cope with believing both  $P$  and  $\neg P$ . Given such a contradiction, it is possible to deduce *anything*. **Retraction** means *removing*  $P$ , so that the system now believes neither  $P$  nor  $\neg P$ .

A complication that arises from this is that a reasoning system may have previously deduced other 'truths', using  $P$  as an antecedent. These sentences need to be re-examined. This is the problem of **truth maintenance**.

A **truth maintenance system** is program that keeps track of dependencies between sentences, so that retraction will be more efficient. The same information can be used to implement **dependency-directed backtracking**, a sophisticated form of *back jumping* that avoids repeating previously fruitless searches. The information also enables explanations to be made of how a proof was obtained.

When a proof is obtained through default reasoning, a truth maintenance system can list the assumptions it has made. A **justification-based truth maintenance system** needs the required set of assumptions to be given explicitly. An **assumption-based truth maintenance system** keeps track of *all* sets of assumptions that have lead to a sentence becoming true.

Both types of system run into computational complexity problems. In practice, it proves important to distinguish assumptions from facts that *cannot* be retracted, and to keep the former set as small as possible.

---

<sup>89</sup> Two enclosing boxes were needed to say that the graph was true.

<sup>90</sup> Prolog attempts to minimise these problems by copying pointers rather than objects, and by garbage collecting all objects that can no longer be accessed by the program.

## 14. Fundamentals of Knowledge Representation

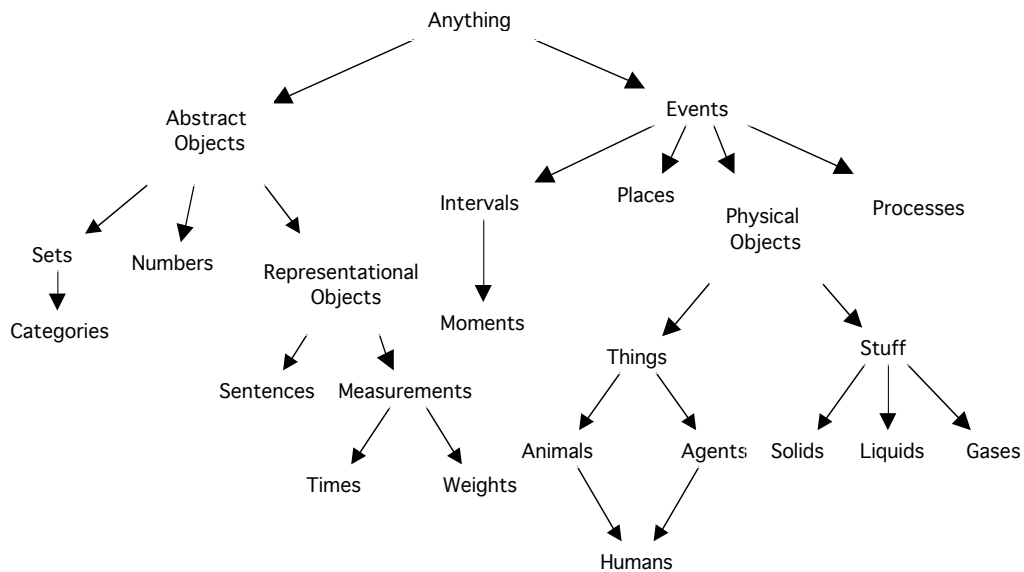
### 14.1. Introduction

**Knowledge Engineering** combines **knowledge acquisition** with devising a formal representation for the knowledge that is acquired. The **knowledge representation** defines a formal language. Like any formal language, experience is required to use it well, and it is easy to make mistakes when expressing a problem. There should be a clear separation between the **knowledge base** and the **inference procedure** used to reason about the knowledge.

An important part of defining a representation is **ontological engineering**: devising the **categories** used to classify things.

### 14.2. Top-Level Ontology

The following ontology is from *Russell & Norvig*, Ch. 8, Fig. 8.2. Notice that it is not a tree; *Humans* are both *Animals* and *Agents*. In general, an ontology forms a lattice. The terminology used here is that used in AI, in so far as there is any agreement.



**Abstract Objects** have no existence in time or space;

**Events** are regions of space-time.

**Sets** are arbitrary collections of objects;

**Categories** are sets of objects that share certain properties.

**Sentences** include statements in English, First-Order Predicate Calculus, Prolog or Java.

**Measurements** are values of properties, e.g., “5cms.”

**Intervals** are regions of time, composed of an infinite number of **moments**;

**Places** are regions of space.

**Things** are discrete objects;

**Stuff** is divisible.

**Processes** are changes in the state of things over time.

#### 14.2.1. Categories

Most reasoning occurs at the level of **categories**, e.g., ‘pits’, or ‘cells’, rather than about individual objects.

**Reification** is the act of regarding a category as an object in its own right, e.g.,  $Count(Cells)=16$ , even though no cell has a property equal to 16.

Subclass-superclass relationships (**taxonomies**) exist between categories. Subclasses **inherit** properties of their superclasses.

Statements about categories are easily stated in first-order logic:

$Pit_{1,2} \sqsubseteq Pits$	Objects are members of categories. <sup>91</sup>
$Pits \sqsubseteq Cells$	Categories are subclasses of bigger categories. <sup>92</sup>
$\Box x x \sqsubseteq Pits \sqsubseteq Bottomless(x) \sqsubseteq Breezy(x)$	Members of a category have certain properties.
$\Box x x \sqsubseteq Cells \sqsubseteq Bottomless(x) \sqsubseteq x \sqsubseteq Pits$	Members of a category can be recognised.
$Pits \sqsubseteq WumpusObjects$	A category is a member of a category of categories.
$\Box x x \sqsubseteq Pits \sqsubseteq x \sqsubseteq Cells \sqsubseteq Bottomless(x)$	Categories may be defined by necessary and sufficient conditions for membership. <sup>93</sup>

Two categories are **disjoint** if they have no members in common, e.g., *Pits* and *WumpusLairs*. A set of categories is an **exhaustive decomposition** of a superclass if every member of the superclass is a member of at least one subclass, e.g.,  $\{Breezy, Smelly, BreezeAndStenchFree, Fatal\}$ . A **partition** is a disjoint exhaustive decomposition, i.e., every member of the superclass is a member of exactly one subclass, e.g.,  $\{Pits, WumpusLairs, Safe\}$ .

### 14.2.2. Measures

Measures are described by a **units function**, such as grams, inches or centimetres. For example:  $Diameter(Pit_{1,2}) = Centimetres(90)$ . Conversion between units may be possible, e.g.,

$$\Box l Inches(l) = Centimetres(2.54 \Box l),$$

$$\Box t Centigrade(t) = Fahrenheit(32 + 1.8 \Box t).$$

Monetary *instruments* are distinct from monetary *amounts*, etc., e.g.,

$$\Box d d \sqsubseteq DollarNotes \sqsubseteq Value(d) = \$ (1.00).$$

In many cases, we cannot assign numerical values to measures, but we can still **order** them, e.g.,  $MusicalQuality(BeethovensFifth) > MusicalQuality(MaryHadALittleLamb)$ . Such orderings are likely to be subjective, and partial (not defined for all pairs.) Such ideas are used in the branch of AI called **qualitative physics**, e.g.,

$$\Box x \Box y Heavy(x) \sqsubseteq Light(y) \sqsubseteq Weight(x) > Weight(y).$$

### 14.2.3. Composite Objects

The relation  $PartOf(x, y)$  expresses the idea that  $x$  is a part of  $y$ .  $PartOf$  is *transitive*:  $PartOf(x, y) \sqsubseteq PartOf(y, z) \sqsubseteq PartOf(x, z)$ . An object with parts is called a **composite object**. The **structure** of an object describes how the parts are related, e.g.,

$$\Box c Car(c) \sqsubseteq \Box w_1, w_2, w_3, w_4, t_1, t_2, t_3, t_4, a_1, a_2, b$$

$$Wheel(w_1) \sqsubseteq Wheel(w_2) \sqsubseteq Wheel(w_3) \sqsubseteq Wheel(w_4)$$

$$\Box Tyre(t_1) \sqsubseteq Tyre(t_2) \sqsubseteq Tyre(t_3) \sqsubseteq Tyre(t_4)$$

$$\Box Axle(a_1) \sqsubseteq Axle(a_2) \sqsubseteq Body(b)$$

$$\Box PartOf(b, c) \sqsubseteq PartOf(a_1, c) \sqsubseteq PartOf(a_2, c)$$

$$\Box PartOf(w_1, c) \sqsubseteq PartOf(w_2, c) \sqsubseteq PartOf(w_3, c) \sqsubseteq PartOf(w_4, c)$$

$$\Box PartOf(t_1, w_1) \sqsubseteq PartOf(t_2, w_2) \sqsubseteq PartOf(t_3, w_3) \sqsubseteq PartOf(t_4, w_4)$$

$$\Box Attached(a_1, b) \sqsubseteq Attached(a_2, b) \sqsubseteq a_1 \neq a_2 \sqsubseteq (\Box a Axle(a) \sqsubseteq PartOf(a, c) \sqsubseteq a = a_1 \ a = a_2)$$

$$\Box Attached(w_1, a_1) \sqsubseteq Attached(w_2, a_1) \sqsubseteq Attached(w_3, a_2) \sqsubseteq Attached(w_4, a_2) \sqsubseteq w_1 \neq w_2 \neq w_3 \neq w_4$$

$$\Box Attached(t_1, w_1) \sqsubseteq Attached(t_2, w_2) \sqsubseteq Attached(t_3, w_3) \sqsubseteq Attached(t_4, w_4) \sqsubseteq t_1 \neq t_2 \neq t_3 \neq t_4 \quad ^{94}$$

When the object is an *event*, we call its structure a **schema** or **script**.

### 14.2.4. Events

One way to model the passage of time is using the **situation calculus**, describing the changes between successive **situations**, or snapshots. It is not a good model for gradual changes or when several things can happen at the same time.<sup>95</sup>

The **event calculus** is meant to remedy these problems. For example, World War II is an event that occurred in different places (e.g., France, Singapore) at different times in the 20<sup>th</sup> Century.

<sup>91</sup> Often referred to as an *ISA* relation.

<sup>92</sup> Often referred to as an *AKO* (*AKindOf*) relation.

<sup>93</sup> It is not so easy to define **natural kinds**. What, for example, do all *games* (e.g., Hide-and-Seek, Soccer, Klondyke, Wumpus) have in common?

<sup>94</sup> This example clearly shows that although First-Order Predicate Calculus *can* describe anything, it is not always convenient or concise.

<sup>95</sup> It is ideal for the *Wumpus* game.

Events have parts called **sub-events**, e.g.,

*Subevent(BattleOfTheBulge, WorldWarII),*  
*Subevent(WorldWarII, TwentiethCentury).*

Events also belong to categories, e.g., *WorldWarII*  $\sqsubseteq$  *Wars*. In the event calculus, we use the notation  $E(Wars, TwentiethCentury)$  to say that at least one war occurred in the 20<sup>th</sup> Century, i.e.,

$\exists w w \sqsubseteq Wars \sqsubseteq SubEvent(w, TwentiethCentury).$

Events composed of distinct parts are called **discrete events**.

An event that is bounded only in the temporal dimension (e.g., the 20<sup>th</sup> Century) is called an **interval**. An interval includes *all* sub-events that happen within its time period. The notation  $T(Wars, 1942AD)$  would mean that at least one war occurred throughout 1942.

An event that is bounded in space, but not in time, is called a **place**.

**Processes** are to discrete events what stuff is to objects. They are continuous events such that any part of them is of the same kind. Thus a *Studying* event is a **process** (or **liquid event**) whose parts consist of *Studying* processes.

The infinitesimal elements of time intervals are called **moments**. A moment may be considered to be an interval of zero duration.

$Duration(i)$  is the difference between the finish,  $Time(End(i))$ , and start,  $Time(Start(j))$ , of  $i$ .

$Meet(i, j)$  is true if  $End(i)$  occurs at the same time as  $Start(j)$ . ( $Time(End(i))=Time(Start(j))$ ).

$Before(i, j)$  and  $After(j, i)$  are true if  $End(i)$  occurs before  $Start(j)$ . ( $Time(End(i))<Time(Start(j))$ ).

$During(i, j)$  is true if  $i$  is any sub-interval of  $j$ .

$Overlap(i, j)$  is true if  $i$  and  $j$  have any common sub-interval.

These relations can describe actions, e.g., “The result of going from a to b is to be at b.”

$\exists i \exists j T(Go(a, b), i) \sqsubseteq T(At(b), j) \sqsubseteq Meet(i, j)$

**Temporal logic** is often used in reasoning about real-time computer systems. It uses the notation  $\Box P(x)$  to mean that  $P(x)$  remains true for **all** future times, and  $\Diamond P(x)$  that  $P(x)$  will be true at **some** future time. The following versions of DeMorgan’s laws are then useful,

$\Box \Box P(x) \sqsubseteq \Box \Diamond P(x)$

$\Box \Box P(x) \sqsubseteq \Diamond \Box P(x)$

#### 14.2.5. Fluents

An object that changes its properties over time is called a **fluent** (that which flows), e.g., *ANR*: the network of track owned by Australian National Railways. We could define a property,  $TrackLength(r)$ , to represent the total track length of a rail network.  $TrackLength(ANR)$  is therefore a function of time. Without such a concept, we would find it hard to say that the Adelaide-Darwin rail link increased the length of the ANR tracks.

A fluent may even be used to represent an object that changes its *identity*, such as “The Prime Minister of Australia”. Again, a fluent makes it easier to say that the Australian Prime Minister was male throughout the 20<sup>th</sup> Century.

#### 14.2.6. Stuff

In English, we can say “a cow”, but we don’t say “a meat”. If you cut a piece of meat in half, you have two pieces of meat. If you cut a cow in half, you have a dead cow. A herd of cows consists of individuals; but meat is a substance or **stuff**, and doesn’t **individuate**. Another way to put this is that ‘cow’ is a **count noun**, and ‘meat’ is a **mass noun**. Properties of stuff that are retained under subdivision, such as colour or taste, are called **intrinsic**; properties that are not retained, such as volume, are called **extrinsic**.

#### 14.2.7. Mental Constructs

Suppose Lois Lane believes Superman can fly. We can write,  $Believes(Lois, Flies(Superman))$ . But Lois does not believe Clark Kent can fly,  $\Box Believes(Lois, Flies(Clark))$ . But since Clark Kent and Superman are one and the same, the principle of **referential transparency**, the idea that we can substitute one thing for another thing it is equal to, does *not* apply to the second argument of *Believes*. Of course, if Lois knew that Clark Kent and Superman were the same, then that would be different. We can therefore only reason about what Lois Lane believes by using what she knows or believes, not what is actually true.

*Believes* is an unusual kind of relation in another way, because relations link *terms*, but the second argument of *Believes* is a *logical sentence*—its truth is independent of whether Lois believes it. The process of treating a sentence as an object is called **reification**.

A **syntactic theory** of belief deals with reified sentences by treating them as strings. (Deductions about these strings are then restricted by a limited set of transformations.) **Modal logic** is similar in purpose. It uses the

functions  $K$  (knows) and  $B$  (believes), and includes special inference rules. The semantics of modal logic may be defined in terms of **possible worlds**. An agent ought not believe in a possible world that is inconsistent with what it knows. We use the term **epistemology** to talk about the study of states of belief.

Actually, there are several kinds of modal logic. ‘Modal’ refers to modal verbs, such as ‘should’, ‘may’, ‘can’, ‘might’, and so on. So a modal logic can discuss what ought to be, or what can be, and so on. An important modal logic concerns logical possibility and necessity, where  $\Diamond P(x)$  means that  $P(x)$  is possibly true, and  $\Box P(x)$  means that  $P(x)$  is necessarily true.  $\Box \Diamond P(x) \wedge \Box \Box P(x)$  expresses the idea that what is not possible is necessarily false. Temporal logic, discussed above, is a logic of this kind.

In first-order logic, no new inference can invalidate a known true sentence. First-order logic is said to be **monotonic**, because the list of true sentences can only increase over time. Because an agent acquires knowledge over time, its beliefs may change non-monotonically. For example, Lois believes that Clark Kent can’t fly. But if she learns that Clark and Superman are the same, she will have to revise this belief.<sup>96</sup> **Non-monotonic logics** raise the problem of **belief maintenance**: efficiently updating an agent’s set of beliefs as new knowledge is acquired.

#### 14.2.8. Action

An agent needs knowledge so it can choose its **actions** more wisely than a simple reflex agent. One way of understanding actions is to look at their **effects**. Effects can be divided into positive effects: predicates that an action causes to become true, and negative effects: predicates that an action forces to become false. In addition, actions must satisfy certain **preconditions** to be applicable. By considering a desired goal, an agent can **plan** a sequence of actions that lead to that goal, taking care that each action will be given the correct preconditions to achieve its desired effects.

### 14.3. Frames and Semantic Nets

The two main paradigms for representing knowledge in Artificial Intelligence are **frames** and **semantic nets** (networks).

Frames correspond closely to objects in Java, but differ in that the rules for inheritance are usually chosen by the programmer, rather than being part of a predefined scheme. Properties of objects are given by the values of **slots**. A slot may contain a value, a reference to another object, or a procedure (similar to a Java method).

Semantic nets focus on the *relationships* between objects. The objects themselves are represented by identifiers. They are related by graphs or mappings. There are some useful properties that many relations share, e.g., transitivity, homogeneity, symmetry, etc. This enables a few general polymorphic data structures and operations to replace many specific ones.

We have already seen a semantic net representation of family relationships as the set of predicates, `male`, `female` and `parent`. From these can be derived many other less direct relationships. However, a predicate representation is inconvenient if a program changes its knowledge base dynamically, for although Prolog allows new facts to be asserted using `asserta` and `assertz`, or existing ones to be retracted using `retract`, the effects of these predicates are *not* undone on backtracking.

Instead, the same information may be stored as sets:

```
Female={'Mary of Teck', 'Mary', 'Elizabeth', 'Elizabeth II', 'Margaret', 'Anne',
      'Sarah'},
Male=  {'George V', 'Edward VIII', 'George VI', 'Henry', 'John', 'Philip', 'Charles',
      'Andrew', 'Edward', 'David'},
Parent={'Mary'-{'George V', 'Mary of Teck'},
      'Edward VIII'-{'George V', 'Mary of Teck'},
      'George VI'-{'George V', 'Mary of Teck'},
      'Henry'-{'George V', 'Mary of Teck'},
      'John'-{'George V', 'Mary of Teck'},
      'Elizabeth II'-{'George VI', 'Elizabeth'},
      'Margaret'-{'George VI', 'Elizabeth'},
      'Charles'-{'Elizabeth II', 'Philip'},
      'Anne'-{'Elizabeth II', 'Philip'},
      'Andrew'-{'Elizabeth II', 'Philip'},
      'Edward'-{'Elizabeth II', 'Philip'},
      'Sarah'-{'Margaret', 'Anthony'},
      'David'-{'Margaret', 'Anthony'}}.
```

Here, the parent relation is represented as a set of A-B pairs. Forming pairs using ‘-’ is common practice in Prolog, and is simply a handy way of forming a small data structure (‘-’ is only interpreted as ‘minus’ in the

<sup>96</sup> This is an example of default reasoning: organisms of human appearance don’t fly.

context of `is`, `==`, `<`, etc.). In this case, the first of each pair is the child, and the second of the pair is a list of the child's parents.

We may choose many other structures to represent the same information. For example, we could use the `child` relation, which is the inverse of the `parent` relation. The most *compact* way to represent a relation is by its

**Galois rectangles:**

```
Child={
  {'George V', 'Mary of Teck'}-{'Mary', 'Edward VIII', 'George VI', 'Henry', 'John'},
  {'George VI', 'Elizabeth'}-{'Elizabeth II', 'Margaret'},
  {'Elizabeth II', 'Philip'}-{'Charles', 'Anne', 'Andrew', 'Edward'},
  {'Margaret', 'Anthony'}-{'Sarah', 'David'}}.
```

However, in a **frame**-oriented approach, we concentrate on the *objects*, in this case, persons. We might declare each person as a fact in the following form:

```
person(Name, Sex, Mother, Father, Children)
```

Which leads to the following predicate:

```
person('Mary of Teck', female, _, _,
  {'Mary', 'Edward VIII', 'George VI', 'Henry', 'John'}).
person('Mary', female, 'Mary of Teck', 'George V', _).
person('Elizabeth', female, _, _, {'Elizabeth II', 'Margaret'}).
person('Elizabeth II', female, 'Elizabeth', 'George VI',
  {'Charles', 'Anne', 'Andrew', 'Edward'}).
person('Margaret', female, 'Elizabeth', 'George VI', {'Sarah', 'David'}).
person('Anne', female, 'Elizabeth II', 'Philip', _).
person('Sarah', female, 'Margaret', 'Anthony', _).
person('George V', male, _, _, {'Mary', 'Edward VIII', 'George VI', 'Henry', 'John'}).
person('Edward VIII', male, 'Mary of Teck', 'George V', _).
person('George VI', male, 'Mary of Teck', 'George V', {'Elizabeth II', 'Margaret'}).
person('Henry', male, 'Mary of Teck', 'George V', _).
person('John', male, 'Mary of Teck', 'George V', _).
person('Philip', male, _, _, {'Charles', 'Anne', 'Andrew', 'Edward'}).
person('Charles', male, 'Elizabeth II', 'Philip', _).
person('Andrew', male, 'Elizabeth II', 'Philip', _).
person('Edward', male, 'Elizabeth II', 'Philip', _).
person('David', male, 'Margaret', 'Anthony', _).
```

where we have adopted a common Prolog convention: if some information is unknown, it is represented by an unbound variable.

Again, if this information needs to be updated by the program, it can be expressed as a set:

```
Persons={person('Mary of Teck', female, _, _,
  {'Mary', 'Edward VIII', 'George VI', 'Henry', 'John'}),
  person('Mary', female, 'Mary of Teck', 'George V', _),
  person('Elizabeth', female, _, _, {'Elizabeth II', 'Margaret'}),
  person('Elizabeth II', female, 'Elizabeth', 'George VI',
  {'Charles', 'Anne', 'Andrew', 'Edward'}),
  person('Margaret', female, 'Elizabeth', 'George VI', {'Sarah', 'David'}),
  person('Anne', female, 'Elizabeth II', 'Philip', _),
  person('Sarah', female, 'Margaret', 'Anthony', _),
  person('George V', male, _, _,
  {'Mary', 'Edward VIII', 'George VI', 'Henry', 'John'}),
  person('Edward VIII', male, 'Mary of Teck', 'George V', _),
  person('George VI', male, 'Mary of Teck', 'George V',
  {'Elizabeth II', 'Margaret'}),
  person('Henry', male, 'Mary of Teck', 'George V', _),
  person('John', male, 'Mary of Teck', 'George V', _),
  person('Philip', male, _, _, {'Charles', 'Anne', 'Andrew', 'Edward'}),
  person('Charles', male, 'Elizabeth II', 'Philip', _),
  person('Andrew', male, 'Elizabeth II', 'Philip', _),
  person('Edward', male, 'Elizabeth II', 'Philip', _),
  person('David', male, 'Margaret', 'Anthony', _)}.
```

It is not hard to retrieve information from such a set. For example, to retrieve the sex of Charles, we may write the following query,

```
| ?- in(person('Charles', Sex, _, _, _), Persons).
Sex=male
```

Each of the positions in a frame is called a **slot**. Sometimes we need to write programs in which the slot structure needs to be flexible, perhaps allowing for new slot types to be added during program execution. It is then common to use an **association-list** (or **attribute-list**). Each frame is represented as a list of slots. If the value of a slot is unknown, the slot is often omitted:

```
Persons={name='Mary of Teck', sex=female,
         children={'Mary', 'Edward VIII', 'George VI', 'Henry', 'John'},
         {name='Mary', sex=female, mother='Mary of Teck', father='George V'},
         {name='Elizabeth', sex=female, children={'Elizabeth II', 'Margaret'}},
         {name='Elizabeth II', sex=female, mother='Elizabeth', father='George VI',
          children={'Charles', 'Anne', 'Andrew', 'Edward'}},
         {name='Margaret', sex=female, mother='Elizabeth', father='George VI',
          children={'Sarah', 'David'}},
         {name='Anne', sex=female, mother='Elizabeth II', father='Philip'},
         {name='Sarah', sex=female, mother='Margaret', father='Anthony'},
         {name='George V', sex=male,
          children={'Mary', 'Edward VIII', 'George VI', 'Henry', 'John'}},
         {name='Edward VIII', sex=male, mother='Mary of Teck', father='George V'},
         {name='George VI', sex=male, mother='Mary of Teck', father='George V',
          children={'Elizabeth II', 'Margaret'}},
         {name='Henry', sex=male, mother='Mary of Teck', father='George V'},
         {name='John', sex=male, mother='Mary of Teck', father='George V'},
         {name='Philip', sex=male, children={'Charles', 'Anne', 'Andrew', 'Edward'}},
         {name='Charles', sex=male, mother='Elizabeth II', father='Philip'},
         {name='Andrew', sex=male, mother='Elizabeth II', father='Philip'},
         {name='Edward', sex=male, mother='Elizabeth II', father='Philip'},
         {name='David', sex=male, mother='Margaret', father='Anthony'}}.
```

Finding Charles's sex is now a little harder:

```
| ?- all(Who, Persons), in(name='Charles', Who), in(sex=Sex, Who).
Who={name='Charles', sex=male, mother='Elizabeth II', father='Philip'}
Sex=male
```

( '=' has been used here (like ' - ') merely as a convenient way of forming pairs.)

To improve the level of abstraction, it is common practice to access slots through the predicates `get_slot` or `set_slot`. This makes the remainder of the program independent of the way the data is represented. An association list could be accessed like this,

```
get_slot(Name, Frame, Value) :- in(Name=Value, Frame).
set_slot(Name, Value, Frame0, Frame) :-
    in(Name=Value1, Frame0), !,
    difference(Frame0, {Name=Value1}, Frame1),
    union({Name=Value}, Frame1, Frame).
set_slot(Name, Value, Frame0, Frame) :-
    union({Name=Value}, Frame0, Frame).
```

whereas the record-based scheme could be accessed like this:

```
get_slot(name, person(Name, _, _, _), Name).
get_slot(sex, person(_, Sex, _, _), Sex).
get_slot(mother, person(_, _, Mother, _, _), Mother).
get_slot(father, person(_, _, _, Father, _), Father).
get_slot(children, person(_, _, _, _, Children), Children).
set_slot(name, Name, person(_, Sex, Mum, Dad, Kids),
         person(Name, Sex, Mum, Dad, Kids)).
set_slot(sex, Sex, person(Name, _, Mum, Dad, Kids),
         person(Name, Sex, Mum, Dad, Kids)).
set_slot(mother, Mum, person(Name, Sex, _, Dad, Kids),
         person(Name, Sex, Mum, Dad, Kids)).
set_slot(father, Dad, person(Name, Sex, Mum, _, Kids),
         person(Name, Sex, Mum, Dad, Kids)).
set_slot(children, Kids, person(Name, Sex, Mum, Dad, _),
         person(Name, Sex, Mum, Dad, Kids)).
```

It is also possible to provide synthesised slots, such as parent:

```
get_slot(parent, Person, Mother) :- get_slot(mother, Person, Mother).
get_slot(parent, Person, Father) :- get_slot(father, Person, Mother).
```

To provide slots such as `uncle`, however, so that *other* person records could be accessed, `get_slot` needs a 4th argument—typically the entire knowledge base.

#### 14.4. Data Normalisation

In providing a person frame with `mother`, `father` and `children` attributes, we have provided the same information as the `parent` and `child` relations discussed earlier. We also know that these are inverses of each other, and that one can be derived from the other. Therefore, we have **redundancy**. If we needed to record a particular father-child relationship, we would need to set the `father` slot of the child, then also modify the `children` slot of the father. Whether redundant information should be stored depends on the ratio of inspections to updates. If updates are rare and inspections are frequent, it may be worth storing redundant information in forms that quickly satisfy the most common queries. If updates are the more common, redundancy should be avoided, because it causes extra work during updating and wastes storage.

Apart from this, the case against redundancy is that it offers the opportunity for the programmer to introduce bugs: *inconsistencies*, for example, where a person has a father, but is not one of the father's children. The aim of **normalisation** is to remove redundancy.

The usual approach is to ensure that each frame has a unique identifier, or **key**, and that every other attribute depends on the key, the whole key, and nothing but the key.

There is a hierarchy of normal forms that is well-known from database theory:

**First Normal Form (1NF)** demands that each frame (or record) should have a unique key, and that every attribute is atomic, i.e., either a number or an atom. This means that attributes such as `children` are *not* allowed, because a list is a structure, not atomic. Where we would like to use a list, we are instead encouraged to store the inverse relationship, e.g., each child should have `mother` and `father` attributes, both of which *are* atomic. The key of a record can be a single attribute, or it can be a combination of several attributes. For example, we might use a frame to represent a marriage, and identify it by the *composite key* (`wife`, `husband`, `date`). (Given the possibility that people can remarry, neither the `wife` nor the `husband` alone is a key, and given that the same two people can divorce and remarry one other, even the two together may not be enough.) In the absence of a natural choice for a key, a serial number can be used, sometimes referred to as an *object identifier*.

**Second Normal Form (2NF)** requires that there should be no *partial dependencies*. A partial dependency exists when the value of an attribute does not depend on the whole of the key. For example, in the case of the hypothetical marriage frame above, `birth_date_of_husband` would depend on only part (`husband`) of the key. Instead, we are encouraged here to store information about the husband in a separate husband frame, otherwise his date-of-birth would be replicated each time he re-married.

**Third Normal Form (3NF)** requires that there be no *transitive dependencies*. An example of a transitive dependency would be for a person frame to contain attributes for the birth dates of a person's mother and father. Such attributes are more properly stored in the `mother` and `father` frames themselves. Transitive dependencies can exist when one frame has an attribute that is the key of another frame. If the first frame also stores an attribute that is really a property of the second frame, it is a transitive dependency. Redundancy arises because several child frames can reference the same parent frame. For example, *each* of the children of a given mother might store their mother's birth date, but it only needs to be stored *once* in the mother frame.

Strictly, we ought to modify these definitions to take account of the case when the attribute concerned is part of the key itself, or, if we originally had a choice of keys, *might* have chosen as part of a key. For example, the `date` associated with a marriage is part of its key, but even though it is an attribute of the marriage, we can't say that it is the cause of a partial dependency. Therefore, we have to exempt keys and potential keys from the rules. Formalising this commonsense idea leads to a number of technical problems that need not concern us. Difficulties rarely arise in practice, and when they do, learning more about normal forms will not help us much. Ideally, each non-key attribute is *functionally dependent* on the key attributes, i.e., is a function of the key.

An extreme form of normalisation is to place each key-attribute pair in a separate frame. If we do this, we reach a semantic net representation in which each attribute is separately represented as a function of a key. Consequently, the distinction between semantic nets and frames is a spectrum rather than a dichotomy.

In order to express a many-to-many relationship, such as `married` in normal form, it is necessary to *reify* it, i.e., to create frames that represent the pairs in the relation, e.g., `marriage` frames.

##### 14.4.1. Abstract Data Structures

It is important to distinguish between a data structure and the abstract mathematical object it represents. For example, a program can represent a *set* in several different ways. For example, Prolog can represent a set by a unary predicate, `female`, for example. This is a suitable representation when the set is fixed, or can be read from a file, but is inconvenient to use if the set is a run-time variable. In that case, a better alternative would be to use a list. The list might be in standard total order, or in arrival order, it might contain duplicates, or it might not. In other words, `[a, b, c]`, `[c, a, b]`, `[a, c, a, b]` are all potential representations of the set  $\{b, a, c\}$ .

Among other alternatives, it is possible to represent a set using a binary search tree. We could use the convention that `bst(Low, Value, High)` is a structure whose second argument is a value, first argument is a tree containing smaller values, and third argument is tree containing larger values. If the empty tree is represented by the atom `empty`, either

```
bst(bst(empty, a, empty), b, bst(empty, c, empty))
```

or

```
bst(empty, a, bst(empty, b, bst(empty, c, empty)))
```

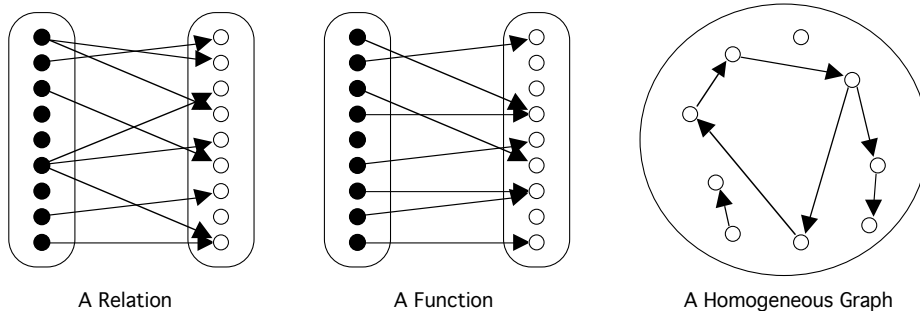
could represent the set  $\{a, b, c\}$ . But we could equally use the *list* `[Low, Value, High]` instead of the structure `bst(Low, Value, High)` and use `[]` in place of `empty`, giving a tree such as `[[[], a[]], b, [[], c[]]]` or `[[[], a[], b, [[], c, []]]`. The message here is that we ought to concentrate on the abstraction, *set*, rather than the means used to represent it.

*Conversely*, a list such as `[a, c, a, b]` might represent the set  $\{a, b, c\}$ , or it might represent the sequence  $\langle a, c, a, b \rangle$ . The main difference between a set and a sequence is that the order of terms in a sequence matters. A sequence has a first term, second term, and so on. A set does *not* have a first member, second member, etc., although it does sometimes make sense to talk about its smallest member, second smallest member, etc.

Some data structures we are familiar with are just degenerate cases of others. For example, a *stack* is a special case of a list, in which we only have access to its head.

How many abstract structures do we need? Surprisingly, just two: *sets*, and *graphs*. A graph is a structure consisting of two sets of elements, called the *domain* and *codomain*, and a set of *edges*, or pairs of elements, the first from the domain, and the second from the codomain. Often, the domain and codomain are the same, when the graph is said to be *homogeneous*. If each element of the domain maps to at most one element of the codomain, the graph is a *function*, otherwise it is a *relation*.

A *sequence* is a special kind of function: one that maps the integers 1, 2, 3, and so on, onto the values of its terms. For example, the sequence  $\langle a, c, a, b \rangle$  has the set of edges  $\{1 \mapsto a, 2 \mapsto c, 3 \mapsto a, 4 \mapsto b\}$ . Likewise, a stack is just a sequence with a restricted set of operators. Everything we ever want to represent can be reduced to two ideas: sets, and pairs. (A group of three things  $(a, b, c)$ , can be reduced to pairs:  $((a, b), c)$ . Prolog cannot even distinguish the two structures.)



#### 14.4.1.1. Operations on Relations and Graphs

Relations and graphs are often regarded as sets of pairs. This means that we may apply any of the set operations to relations. In addition, relations have a powerful algebra of their own, and often, thinking in such terms is helpful.

$R_1 ; R_2$  denotes the *composition* of  $R_1$  and  $R_2$ , i.e., the effect of applying  $R_1$  to an argument, then applying  $R_2$  to the result. Thus, we may write,

$$\text{Grandparent} \sqsubseteq \text{Parent} ; \text{Parent}$$

However, since the *Parent* relation is composed with itself, we may simplify this:

$$\text{Grandparent} \sqsubseteq \text{Parent}^2$$

In a similar way, we may define *Greatgrandparent* and *Ancestor* as follows,

$$\text{Greatgrandparent} \sqsubseteq \text{Parent}^3$$

$$\text{Ancestor} \sqsubseteq \text{Parent}^+$$

$\text{Parent}^+$  denotes the *transitive closure* of the *Parent* relation. Thinking of the *Parent* relation as a directed graph,  $\text{Parent}^+$  is the set of all paths of length  $\geq 1$ . If we wanted to consider persons to be their own ancestors, we would use the *reflexive transitive closure*,  $\text{Parent}^*$ .

In a similar way, we may use negative powers;  $R^{-1}$  denotes the *inverse* of  $R$ , so that we can define *Child*, *Grandchild* and *Descendant* as follows,

$$\text{Child} \sqsubseteq \text{Parent}^{-1}$$

$$\text{Grandchild} \sqsubseteq \text{Parent}^{-2}$$

$Descendant \sqsubseteq (Parent^{-1})^+$

We may still use *set operators* to define sets:

$Person \sqsubseteq Female \sqcup Male$

Since relations are a kind of set, we may also use set operators to combine relations. For example, *Sibling* relates two *different* individuals who share at least one parent. We may define *Sibling* as the *asymmetric difference* ( $\setminus$ ) between the relation of sharing a parent, and an identity relation:

$Sibling \sqsubseteq (Parent ; Parent^{-1}) \setminus I_{Person}$

### 14.5. An Alternative Ontology

Sowa<sup>97</sup> stresses that an ontology is not tree-like, but is actually a complete lattice—although not all of the lattice may be useful in a particular domain. He gives 12 central categories based on two dichotomies: **Physical/Abstract** and **Continuant/Occurrent**, and a trichotomy: **Independent/Relative/ Mediating**. The physical/abstract distinction can easily be answered by asking if the thing has mass or energy. The continuant/occurrent distinction depends on the point of view. From the viewpoint of an historian the Great War is an occurrent, but from the viewpoint of a soldier in the trenches, it was a continuant. The trichotomy relates to the previous section. Independent entities correspond to sets, relative entities correspond to relations, and mediating entities correspond to reified relations. For example, the unary predicate *Husband*(*x*) defines an independent category, the binary predicate *Married*(*x*, *y*) defines a relative category, and *Marriage*(*x*, *y*) defines a mediating category. (Usually, when we reify a relation, it is found to have further properties; a marriage has a date, and so on.)

This leads to the following matrix of possibilities:

	Physical		Abstract	
	Continuant	Occurrent	Continuant	Occurrent
Independent	Object	Process	Schema	Script
Relative	Juncture	Participation	Description	History
Mediating	Structure	Situation	Reason	Purpose

In Sowa’s ontology, all 12 combinations of properties appear. However, although Sowa has named 6 super-categories, he has not given names to the super-categories IO, IC, RI, RC, MO, MC, PO, PC, AO, and AC. It is not hard to see what these categories represent. IO, for example is the union of IPO (Process) and IAO (Script), not something for which there is a common word.

Actuality (IP)	A physical object.
Prehension (RP)	A physical relationship.
Nexus (MP)	An instance of a physical relationship, i.e., the relationship considered in its own right.
Form (IA)	An abstract object.
Proposition (RA)	A statement about an object or objects.
Intention (MA)	A statement about a proposition, e.g., What is it for? How do I feel about it? Do I believe it?
Object (IPC)	An actuality (IP) considered as a continuant (C), which retains its identity over time.
Process (IPO)	An actuality (IP) considered as an occurrent (O). A dynamic process.
Schema (IAC)	A form (IA) that does not specify time-like relationships, e.g., a database schema.
Script (IAO)	A form (IA) that represents time-like sequences, e.g., a computer program.
Juncture (RPC)	A prehension (RP) considered as a continuant over time. A stable relationship between objects, e.g., a joint between two wires.
Participation (RPO)	A prehension (RP) considered as an occurrent. A relationship between a process and a continuant, e.g., a program statement being executed.
Description (RAC)	A proposition (RA) that characterises a continuant by a schema (IAC), e.g., a proposition describing an object.
History (RAO)	A proposition (RA) that characterises an occurrent by a script (IAO), e.g., a trace of a program.
Structure (MPC)	A nexus (MP) considered as a continuant (C) that explains how the junctures (RPC) of some components are organised, e.g., a car (as opposed to a collection of car parts).
Situation (MPO)	A nexus (MP) considered as an occurrent, e.g., a planning meeting. Situations may have other situations as components.

<sup>97</sup> *Op. cit.*

Reason (MAC)		An intention (MA) concerning a continuant. A description might describe a computer program, a reason would explain what it is useful for.
Purpose (MAO)		An intention (MA) concerning an occurrent, e.g., why a program was executed.

Wille's *Formal Concept Theory* is a similar approach to categorisation. The idea is that if we have any collection of objects, and any list of properties, we can put all the objects with identical sets of properties into the same category. If we take any two categories formed in this way, we can form a super-category that contains both, and contains all objects that have the intersection of both sets of properties, and also form a sub-category of both that contains objects with the union of the properties. In principle, any combination of properties defines a category, and the categories form a complete lattice. If we apply this approach to the categories 'cat' and 'dog', their intersection roughly corresponds to 'carnivorous mammals that make good pets', but their union corresponds to no actual animal.

## 15. Default Reasoning

### 15.1. Implementing Frames

Frames are a way of capturing information about **inheritance**, **categories**, and **instances**.<sup>98</sup>

A category has a name, and a set of slots, which may have default values e.g.,

```
disaster has {killed:_, injured:_, damage:'$1,000,000'}.
event    has {place:'Adelaide', day:yesterday, time:afternoon}.
```

A category may be a kind of (ako) a super-category, e.g.,

```
disaster ako event.
```

A frame may be an instance of (isa) a category, e.g.,

```
Frame isa disaster.
```

Here is a file (`frames.pro`) that implements frames. The `ako` operator is used to declare that a category is a sub-category of a parent category. The `has` operator is used to declare the list of slots (attributes) associated with a category. The `isa` operator creates a frame instance of a given category:

```
:- op(500, xfx, [has, ako, isa, :]).
:- include(sets).
(Category:Slots) isa Category:-
    findset(Slot:_, collect_slot(Category, Slot), Slots1),
    eliminate_duplicates(Slots1, {}, Slots).
```

When an instance is created, it includes a slot for each attribute of the category itself and all its ancestors. All its slot names are initially paired to different (unbound) anonymous variables:

```
collect_slot(Category, Slot) :-
    Category has Slots,
    all(Slot:_, Slots).
collect_slot(Category, Slot) :-
    Category ako Parent,
    collect_slot(Parent, Slot).
```

Unfortunately, if the child category shares a slot name with one of its ancestors (which is quite possible, since it may over-ride the ancestor's default value), `findset` (which compares terms in standard order) considers the two anonymous variables to be different, and more than slot with the same name results.

`eliminate_duplicates` gets rid of the duplicates. (`in/2` works by matching, so that two unbound variables are treated as equal):

```
eliminate_duplicates(Slots0, Slots, Slots) :- empty(Slots0).
eliminate_duplicates(Slots0, Slots1, Slots) :-
    least(Slots0, Slot, Slots2),
    (in(Slot, Slots1)->Slots3=Slots1; union({Slot}, Slots1, Slots3)),
    !, eliminate_duplicates(Slots2, Slots3, Slots).
```

Setting the value of an unbound slot in a frame is trivial:

```
set_slot(_:Slots, Slot, Value) :- in(Slot=Value, Slots).
```

Getting the value of a slot is more complex, because if the slot is unbound, the frame's category and its ancestors must be searched to find the appropriate default:

```
get_slot(_:Slots, Slot=Value) :-
    in(Slot=Value, Slots),
    atomic(Value), !.
get_slot(Event:_, Slot=Value) :-
    Event has Slots,
    in(Slot=Value, Slots),
    atomic(Value), !.
get_slot(Event:_, Slot=Value) :-
    Event ako Parent,
    Parent has Slots,
    get_slot(Parent:Slots, Slot=Value), !.
get_slot(_:Slots, Slot=Value) :- in(Slot=Value, Slots).
```

Frames do not have to consist solely of data; they can have methods associated with them. For example, a frame representing a point on the X-Y plane could have slots for its *X* and *Y* values. But points can also be

<sup>98</sup> P.H. Winston, *Op. cit.*, Ch. 9: Frames and Inheritance.

represented by polar,  $r, \phi$  coordinates. The polar coordinates could be derived from the  $X, Y$  coordinates by **methods**. A good implementation of `get_slot` would make it unnecessary to know whether points were stored in  $X, Y$  or  $r, \phi$  form.

Although frames have much in common with objects in Java, etc., there are big differences. First, a class hierarchy can be defined at run time rather than compile time. Second, the inheritance rules can be chosen by the programmer, so that Java-like rules are only one possibility that a frame-based system could use. Third, general-purpose predicates (e.g., `find_slots`) can be written to work with *any* class hierarchy that adheres to certain conventions (i.e., `ako`, `isa`, etc.).

## 15.2. Using Frames

The following program shows how information about earthquakes and other events can be extracted from news stories. First, we consider its category definitions:

```
:- include(frames).
event has {place:'Adelaide', day:yesterday, time:afternoon}.
disaster has {killed:_, injured:_, damage:'$1,000,000'}.
political_event has {who:'John Howard', what:_}.
sports_event has {sport:'AFL', winner:'Port Power', score:_}.
earthquake has {magnitude:'about 7', place:'San Francisco', fault:'San Andreas'}.

disaster ako event.
political_event ako event.
sports_event ako event.
earthquake ako disaster.
```

The most generic category is an event. An earthquake is a kind of (`ako`) disaster, which is a kind of (`ako`) event. As such, an earthquake frame will have a total of eight properties, or **slots**: `place`, `day`, `time`, `killed`, `injured`, `damage`, `magnitude` and `fault`. The default place, San Francisco over-rides Adelaide. There are no defaults for `killed` or `injured`.

The purpose of the program is to extract data from news stories and produce a summary. Here is an example of its use:

```
| ?- report(Frame), print(Frame), nl.
Today, an extremely serious earthquake of magnitude 8.5 hit Lower Slabovia, killing 25 people and causing $500 million in damage.
Earthquake Summary
damage: $500 million
day:today
fault:San Andreas
killed:25
magnitude:8.5
place:Lower Slabovia
time:afternoon

earthquake:{damage: $500 million, day:today, fault:_668, injured:_652, killed:25,
magnitude:8.500, place:Lower Slabovia, time:_636}
```

The goal `report(Frame)` reads words from the keyboard, analysing them for information connected with the events it can recognise, and summarising its results. It does *not* understand what is typed. It looks for trigger words, like 'magnitude' or 'fault'.

The program reformats the facts it has extracted into a standard summary. A slot is reported if a value was found in the text, or if it has a default value. For example, `time:afternoon` is inherited from the event category; `fault:San Andreas` is inherited from the earthquake category. The default `fault` is inappropriate here; Lower Slabovia is not in California. Unbound slots without defaults (e.g., `injured`) are not listed.

Because of the defaults used in frames, `report` can virtually invent a summary on its own:

```
| ?- report(Frame), print(Frame), nl.
Earthquake!
Earthquake Summary
damage: $1,000,000
day:yesterday
fault:San Andreas
magnitude:about 7
place:San Francisco
time:afternoon
```

```
earthquake:{damage:_138, day:_126, fault:_150, injured:_134, killed:_130,
magnitude:_146, place:_122, time:_118}
```

Here is the report predicate:

```
report(Frame) :- read_news(Frame), write_news(Frame), !.
```

`read_news/1` reads input from the keyboard, analysing it into words. It then searches the list of words (using `find`, discussed shortly) for a word that indicates the category of event, creates a frame of that category, and having done that, it attempts to fill the slots in the frame, again using `find` to search for words or phrases that can supply the wanted values:

```
read_news(Frame) :-
    read_words(Words),
    find(Words, category:Category),
    Frame isa Category,
    Frame = _:Slots,
    fill_slots(Words, Slots).

fill_slots(Words, Slots) :-
    findset(Slot:Value,
            (all(Slot:_, Slots), find(Words, Slot:Value)),
            Slots).
```

`write_news` prints a heading appropriate to the category of event, then prints the value of every slot that has one. Recall that `get_slot` will either return the value of the slot itself, or a default inherited from the category or its parents.

```
write_news(earthquake:Slots) :-
    nl, print('Earthquake Summary'), nl, nl,
    write_slots(earthquake:Slots, Slots).
write_news(disaster:Slots) :-
    nl, print('Disaster Summary'), nl, nl,
    write_slots(disaster:Slots, Slots).
write_news(political_event:Slots) :-
    nl, print('Political Event Summary'), nl, nl,
    write_slots(political_event:Slots, Slots).
write_news(sports_event:Slots) :-
    nl, print('Sports Event Summary'), nl, nl,
    write_slots(sports_event:Slots, Slots).

write_slots(Frame, Slots) :-
    all(Slot:_, Slots),
    get_slot(Frame, Slot:Value),
    (nonvar(Value)->print((Slot:Value)), nl; true),
    fail.
write_slots(_, _).
```

## 15.3. Analysing the Input

### 15.3.1. Low-level Input in Prolog

`read_words(Words)` returns the list of words (`Words`) typed at the keyboard. This is a task that is common to several example programs, so we shall examine it in detail here. In this case, the predicate reads a single line, i.e., any string of characters terminated by RETURN. Prolog provides a `get0/1` predicate, which binds its argument to the ASCII code of the next input character:

```
read_words(Words) :- read_line(Chars), words(Chars, Words).

read_line(Chars) :- get0(LookAhead), read_line(LookAhead, Chars).

read_line(LookAhead, []) :- end_of_line(LookAhead), !.
read_line(LookAhead, [LookAhead|Chars]) :- get0(Char1), read_line(Char1, Chars).

end_of_line(Char) :- Char < 32.
```

There is a trick to using `get0`. Since `end_of_line/1` usually fails, it must *not* call `get0` to test if the *next* character is the end-of-line, because `get0` causes the *side-effect* of reading the next character, and the side-effect would not be undone on back-tracking. So each test for end-of-line would throw away a character from the input stream. Instead, `read_line/1` calls `read_line/2`, passing it a *look-ahead character*. `end_of_line` examines the look ahead, but does *not* call `get0`. (Since the end-of-line character is system-dependent, `end_of_line` regards *any* control character as a line terminator.)

Once `read_line/1` has accepted a line of input, the ASCII codes are grouped into ‘words’. For our purposes, a ‘word’ is one of four things: a letter followed by other letters, hyphens or apostrophes; a digit followed by other

digits, decimal points or commas; a sequence of spaces; or any single character not falling into the other three classes. A sequence of spaces is always returned as a single space.

```

words([], []).
words([Char|Chars], [Word|Words]) :- letter(Char), !,
    restword(Chars, Chars1, Rest),
    atom_codes(Word, [Char|Rest]),
    words(Chars1, Words).
words([Char|Chars], [Word|Words]) :- digit(Char), !,
    restnum(Chars, Chars1, Rest),
    atom_codes(Word, [Char|Rest]),
    words(Chars1, Words).
words([Char|Chars], [' '|Words]) :- Char== " ", !,
    spaces(Chars, Chars1), words(Chars1, Words).
words([Char|Chars], [Punct|Words]) :-
    atom_codes(Punct, [Char]), words(Chars, Words).

letter(Char) :- Char >= "A", Char <= "Z".
letter(Char) :- Char >= "a", Char <= "z".
digit(Char) :- Char >= "0", Char <= "9".

restword([Char|Chars], Chars1, [Char|Rest]) :-
    (letter(Char);Char== "-" ;Char== "-"), !,
    restword(Chars, Chars1, Rest).
restword(Chars, Chars, []).

restnum([Char|Chars], Chars1, [Char|Rest]) :-
    (digit(Char);Char== "." ;Char== "-"), !,
    restnum(Chars, Chars1, Rest).
restnum(Chars, Chars, []).

spaces([Char|Chars], Chars) :- Char== " ", !.
spaces(Chars, Chars).

```

As a convenience, Prolog allows a list of one number to be treated as a number. The comparisons within `letter/1` (for example) are valid, even though `Char` is a number and "A" is really the list `[65]`.

Each word scanned is converted to an atom, using `atom_codes/2`. Atoms use less space than strings, and are easier to deal with. Thus, `read_words` binds its argument to a list of atoms, e.g., `['Today', ',', ' ', 'an', ' ', 'extremely', ' ', 'serious', ' ', 'earthquake', ' ', 'of', ' ', 'magnitude', ' ', '8.5', ' ', 'hit', ' ', 'Lower', ' ', 'Slabovia', ' ', 'killing', ' ', '25', ' ', 'people', ' ', 'and', ' ', 'causing', ' ', '$', '500', ' ', 'million', ' ', 'in', ' ', 'damage', '.']`.

### 15.3.2. Filling the Slots

`find/2` scans the list of words, looking for trigger phrases that might supply the value of the given slot. It has many clauses, which constitute, in effect, *expert knowledge* about how English is used in newspaper articles. In practice, such rules might include a statistical measure of goodness, and only the best matches would be chosen. Here is one of them:

```

find(Words, time:T) :-
    subsequence([X, ':', Y, Z], Words),
    number(X), number(Y),
    (Z=am;Z=pm), !,
    flatten([X, ':', Y, Z], T).

```

`find/2` uses simple *ad hoc* pattern matching. In this case, if the list of words contains a number `X`, a colon, a number `Y` and either `am` or `pm`, in that order, they are concatenated to give the value of the time slot. There are several other rules for finding a time, arranged in order of precedence. There are many more `find` rules for the remaining slots. The last `find` rule of all is,

```
find(_, _).
```

which prevents `find_slots` failing when the word list contains no data for a slot.

The `subsequence` predicate uses the simplest possible approach, trying to match its first argument against each position in its second argument in turn:

```

subsequence([], _).
subsequence([W|T1], [W|T2]) :- match(T1, T2).
subsequence(W, [_|T]) :- subsequence(W, T).

match([], _).
match([W|T1], [W|T2]) :- match(T1, T2).

```

The `flatten` predicate concatenates a list of 'words' into a single word. It uses Prolog's built-in `atom_codes/2` predicate to convert atoms to lists and back again:

```
flatten(Words, Atom) :- flatten2(Words, Str), atom_codes(Atom, Str).  
flatten2([], []).  
flatten2([Word|Words], Str) :-  
    atom_codes(Word, Str1),  
    append(Str1, Str2, Str), !,  
    flatten2(Words, Str2).
```