The Machine Learning Dictionary for COMP9414

Contact Info Last updated:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

You should use The Machine Learning Dictionary to clarify or revise concepts that you have already met. The Machine Learning Dictionary is *not* a suitable way to *begin* to learn about Machine Learning.

Further information on Machine Learning can be found in the class web page lecture notes section. Other places to find out about machine learning would be the AAAI (American Association for Artificial Intelligence) Machine Learning page and their AI Overview page or AI Reference Shelf for less specific information.

Other related dictionaries:

The Prolog Dictionary - URL: `http://www.cse.unsw.edu.au/~billw/prologdict.html`
The Artificial Intelligence Dictionary - URL: `http://www.cse.unsw.edu.au/~billw/aidict.html`
The NLP Dictionary (Natural Language Processing) - URL: `http://www.cse.unsw.edu.au/~billw/nlpdict.html`

The URL of this Machine Learning Dictionary is `http://www.cse.unsw.edu.au/~billw/mldict.html`

Topics Covered

This dictionary is limited to the ML concepts covered in COMP9414 Artificial Intelligence, at the University of New South Wales, Sydney:

activation level | activation function | Aq | asynchronous | attributes | axon | backed-up error estimate | backpropagation | backward pass in backpropagation | bias | biological neuron | C4.5 | C5 | cell body | clamping | classes in classification tasks | concept learning system (CLS) | conjunctive expressions | connectionism | covering algorithm | decision trees | delta rule | dendrite | entropy | epoch | error backpropagation | error surface | excitatory connection | expected error estimate | feature | feedforward networks | firing | forward pass in backpropagation | function approximation algorithms | generalization

A

activation level

The activation level of a neuron in an artificial neural network is a real number often limited to the range 0 to 1, or -1 to 1. In the case of an input neuron the value is obtained externally to the network. In the case of a hidden neuron or output neuron the value is obtained from the neuron's activation function. The activation of a neuron is sometimes thought of as corresponding to the average firing *rate* of a biological neuron.

activation function

In neural networks, an activation function is the function that describes the output behaviour of a neuron. Most network architectures start by computing the weighted sum of the inputs (that is, the sum of the product of each input with the weight associated with that input. This quantity, the total net input is then usually transformed in some way, using what is sometimes called a squashing function. The simplest squashing function is a step function: if the total net input is less than 0 (or more generally, less than some threshold T) then the output of the neuron is 0, otherwise it is 1. A common squashing function is the logistic function.

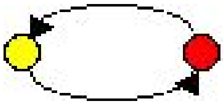
In summary, the activation function is the result of applying a squashing function to the total net input.

Aq

A propositional learning system, developed by Michalski.

asynchronous vs **synchronous**

When a neural network is viewed as a collection of connected computation devices, the question arises whether the nodes/devices share a common clock, so that they all perform their computations ("fire") at the same time, (i.e. **synchronously**) or whether they fire at different times, e.g. they may fire equally often on average, but in a random sequence (i.e. **asynchronously**). In the simplest meaningful case, there are two processing nodes, each connected to the other, as shown below.



In the asynchronous case, if the yellow node fires first, then it uses the then current value of its input from the red node to determine its output in time step 2, and the red node, if it fires next, will use the updated output from the yellow node to compute its new output in time step 3. In summary, the output values of the red and yellow nodes in time step 3 depend on the outputs of the yellow and red nodes in time steps 2 and 1, respectively.

In the synchronous case, each node obtains the current output of the other node at the same time, and uses the value obtained to compute its new output (in time step 2). In summary, the output values of the red and yellow nodes in time step 2 depend on the outputs of the yellow and red nodes in time step 1. This can produce a different result from the asynchronous method.

Some neural network algorithms are firmly tied to synchronous updates, and some can be operated in either mode. Biological neurons normally fire asynchronously.

attributes

An attribute is a property of an instance that may be used to determine its classification. For example, when classifying objects into different types in a robotic vision task, the size and shape of an instance may be appropriate attributes. Determining useful attributes that can be reasonably calculated may be a difficult job - for example, what attributes of an arbitrary chess end-game position would you use to decide who can win the game? This particular attribute selection problem has been solved, but with considerable effort and difficulty.

Attributes are sometimes also called **features**.

axon

The axon is the "output" part of a biological neuron. When a neuron fires, a pulse of electrical activity flows along the axon. Towards its end, or ends, the axon splits into a tree. The ends of the axon come into close contact with the dendrites of other neurons. These junctions are termed synapses. Axons may be short (a couple of millimetres) or long (e.g. the axons of the nerves that run down the legs of a reasonably large animal.)

B

backed-up error estimate

In decision tree pruning one of the issues in deciding whether to prune a branch of the tree is whether the estimated error in classification is greater if the branch is present or pruned. To estimate the error if the branch is present, one takes the estimated errors associated with the children of the branch nodes (which of course must have been previously computed), multiplies them by the estimated frequencies that the current branch will classify data to each child node, and adds up the resulting products. The frequencies are estimated from the numbers of training data instances that are classified as belonging to each child node. This sum is called the backed-up error estimate for the branch node. (The concept of a backed-up error estimate does not make sense for a leaf node.)

See also expected error estimate.

backward pass in backpropagation

The phase of the error backpropagation learning algorithm when the weights are updated, using the delta rule or some modification of it.

The backward pass starts at the output layer of the feedforward network, and updates the incoming weights to units in that layer using the delta rule. Then it works backward, starting with the penultimate layer (last hidden layer), updating the incoming weights to those layers.

Statistics collected during the forward pass are used during the backward pass in updating the weights.

backpropagation or backprop

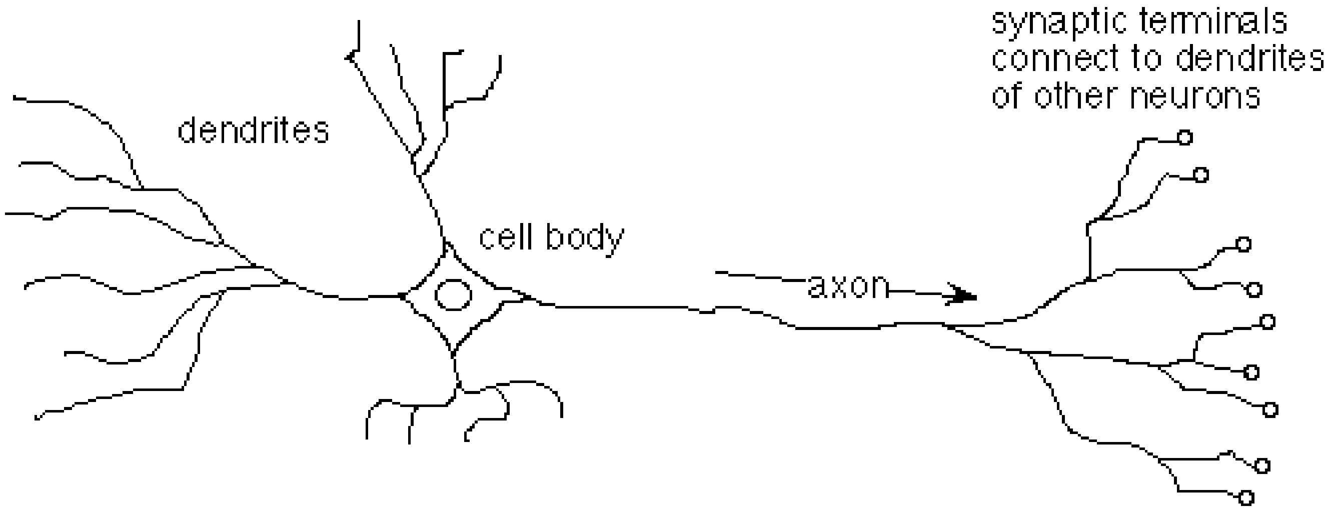
see error backpropagation

bias

In feedforward and some other neural networks, each hidden unit and each output unit is connected via a trainable weight to a unit (the **bias unit**) that always has an activation level of -1.

This has the effect of giving each hidden or output a trainable threshold, equal to the value of the weight from the bias unit to the unit.

biological neuron



C

C4.5

C4.5 is a later version of the ID3 decision tree induction algorithm.

C5

C5 is a later version of the ID3 decision tree induction algorithm.

cell body

The cell body is the large open space in the "middle" of the neuron, between the dendrites and the axon, where the cell nucleus lives. A landmark in biological neuron architecture, but not of significance in relation to artificial neural networks (except for those trying to model biological neurons as distinct from using simplified neuron models to solve diverse problems).

clamping

When a neuron in an neural network has its value forcibly set and fixed to some externally provided value, it is said to be clamped. Such a neuron serves as an input unit for the net.

classes in classification tasks

In a classification task in machine learning, the task is to take each instance and assign it to a particular class. For example, in a machine vision application, the task might involve analysing images of objects on a conveyor belt, and classifying them as *nuts*, *bolts*, or other components of some object being assembled. In an optical character recognition task, the task would involve taking instances representing images of characters, and classifying according to which character they are. Frequently in examples, for the sake of simplicity if nothing else, just two classes, sometimes called *positive* and *negative*, are used.

concept learning system (CLS)

A decision tree induction program - a precursor of ID3.

conjunctive expressions

A conjunctive expression is an expression like:

size=large **and** colour **in** {red, orange}

that is the *conjunction* of two or more simple predicates like `size=large`. The term *conjunction* refers to the presence of the logical operator **and**, which joins or *conjoins* the simple predicates. Occasionally the conjunctive expression might just consist of a single predicate.

See also propositional learning systems and covering algorithm.

connectionism

Connectionism is the neural network approach to solving problems in artificial intelligence - the idea being that connectionists feel that it is appropriate to encode knowledge in the weighted *connections* between nodes in a neural net. The word "connectionist" is sometimes used with all the heat and force associated with political "isms": "He's a connectionist" can be just as damning, coming from the wrong person, as "He's a communist (or capitalist)". It is also sometimes used as a simple adjective, as in "NetTalk is a connectionist system."

covering algorithm

A covering algorithm, in the context of propositional learning systems, is an algorithm that develops a *cover* for the set of positive examples - that is, a set of conjunctive expressions that account for all the examples but none of the non-examples.

The algorithm - given a set of examples:

1. Start with an empty cover.
2. Select an example.
3. Find the set of *all* conjunctive expressions that cover that example.
4. Select the "best" expression *x* from that set, according to some criterion (usually "best" is a compromise between generality and compactness and readability).
5. Add *x* to the cover.
6. Go to step 2, unless there are no examples that are not already covered (in which case, stop).

D

decision trees

A decision tree is a tree in which each non-leaf node is labelled with an attribute or a question of some sort, and in which the branches at that node correspond to the possible values of the attribute, or answers to the question. For example, if the attribute was `shape`, then there would be branches below that node for the possible values of `shape`, say `square`, `round` and `triangular`. Leaf nodes are labelled with a class. Decision trees are used for classifying instances - one starts at the root of the tree, and, taking appropriate branches according to the attribute or question asked about at each branch node, one eventually comes to a leaf node. The label on that leaf node is the class for that instance.

delta rule

The delta rule in error backpropagation learning specifies the update to be made to each weight during backprop learning. Roughly speaking, it states that the change to the weight from node *i* to node *j* should be proportional to output of node *j* and also proportional to the "local gradient" at node *j*.

The local gradient, for an output node, is the product to the derivative of the squashing function evaluated at the total net input to node *j*, and the error signal (i.e. the difference between the target output and the actual output). In the case of a hidden node, the local gradient is the product of the derivative the squashing function (as above) and the weighted sum of the local gradients of the nodes to which node *j* is connected in subsequent layers of the net. Got it?

dendrite

A dendrite is one of the branches on the input end of a biological neuron. It has connections, via synapses to the axons of other neurons.

E

entropy

For our purposes, the entropy measure

$$-\sum_j p_j \log_2 p_j$$

gives us the average amount of information in bits in some attributeof an instance. The rationale for this is as follows: -$\log_2(p)$ is the amount of information in bits associated with an event of probability *p*- for example, with an event of probability ½, like flipping a fair coin, $\log_2((p)$ is -$\log_2(½)$ = 1, so there is one bit of information. This should coincide with our intuition of what a bit means (if we have one). If there is a range of possible outcomes with associated probabilities, then to work out the average number of bits, we need to multiply the number of bits for each outcome (-$\log_2(p)$ by the probability *p*and sum over all the outcomes. This is where the formula comes from.

Entropy is used in the ID3 decision tree induction algorithm.

epoch

In training a neural net, the term epoch is used to describe a complete pass through all of the training patterns. The weights in the neural net may be updated after each pattern is presented to the net, or they may be updated just once at the end of the epoch. Frequently used as a measure of speed of learning - as in "training was complete after *x* epochs".

error backpropagation learning algorithm

The error backpropagation learning algorithm is a form of supervised learning used to train mainly feedforward neural networks to perform some task. In outline, the algorithm is as follows:

1. Initialization: the weights of the network are initialized to small random values.
2. Forward pass: The inputs of each training pattern are presented to the network. The outputs are computed using the inputs and the current weights of the network. Certain statistics are kept from this computation, and used in the next phase. The target outputs from each training pattern are compared with the actual activation levels of the output units - the difference between the two is termed the error. Training may be pattern-by-pattern or epoch-by-epoch. With pattern-by-pattern training, the pattern error is provided directly to the backward pass. With epoch-by-epoch training, the pattern errors are summed across all training patterns, and the total error is provided to the backward pass.

3. Backward pass: In this phase, the weights of the net are updated. See the main article on the backward pass for some more detail.

4. Go back to step 2. Continue doing forward and backward passes until the stopping criterion is satisfied.

See also forward pass, backward pass, delta rule, error surface, local minimum, gradient descent and momentum.

Error backpropagation learning is often familiarly referred to just as **backprop**.

error surface

When total error of a backpropagation-trained neural network is expressed as a function of the weights, and graphed (to the extent that this is possible with a large number of weights), the result is a surface termed the *error surface*. The course of learning can be traced on the error surface: as learning is supposed to reduce error, when the learning algorithm causes the weights to change, the current point on the error surface should descend into a valley of the error surface.

The "point" defined by the current set of weights is termed a point in **weight space**. Thus weight space is the set of all possible values of the weights.

See also local minimum and gradient descent.

excitatory connection

see weight.

expected error estimate

In pruning a decision tree, one needs to be able to estimate the expected error at any node (branch or leaf). This can be done using the **Laplace error estimate**, which is given by the formula

$$E(S) = (N - n + k - 1) / (N + k).$$

where

S	is the set of instances in a node
k	is the number of classes (e.g. 2 if instances are just being classified into 2 classes: say positive and negative)
N	is the is the number of instances in S
C	is the majority class in S
n	out of N examples in S belong to C

F

feature

see attribute.

feedforward net

A kind of neural network in which the nodes can be numbered, in such a way that each node has weighted connections only to nodes with higher numbers. Such nets can be trained using the error backpropagation learning algorithm.

In practice, the nodes of most feedforward nets are partitioned into **layers** - that is, sets of nodes, and the layers may be numbered in such a way that the nodes in each layer are connected to all and only the nodes in the next layer - that is, the layer with the next higher number.

The first layer has no input connections, so consists of input units and is termed the **input layer**.

The last layer has no output connections, so consists of output units and is termed the **output layer**.

The layers in between the input and output layers are termed hidden layers, and consist of hidden units.

When the net is operating, the activations of non-input neurons are computing using each neuron's activation function.

firing

1. In a biological neural network: neurons in a biological neural network fire when and if they receive enough stimulus via their (input) synapses. This means that an electrical impulse is propagated along the neuron's axon and transmitted to other neurons via the output synaptic connections of the neuron. The **firing rate** of a neuron is the frequency with which it fires (cf. activation in an artificial neural network.

2. In an expert system: when a rule in the expert system is used, it is said to fire.

function approximation algorithms

include connectionist and statistical techniques of machine learning. The idea is that machine learning means learning, from a number of examples or instances or training patterns, to compute a function which has as its arguments variables corresponding to the input part of the training pattern(s), and has as its output variables corresponding to the output part of the training patterns, which maps the input part of each training pattern to its output part. The hope is that the function will interpolate / generalize from the training patterns, so that it will produce reasonable outputs when given other inputs.

See also symbolic learning algorithms.

G

generalization in backprop

Learning in backprop seems to operate by first of all getting a rough set of weights which fit the training patterns in a general sort of way, and then working progressively towards a set of weights that fit the training patterns exactly. If learning goes too far down this path, one may reach a set of weights that fits the idiosyncrasies of the particular set of patterns very well, but does not interpolate (i.e. generalize) well.

Moreover, with large complex sets of training patterns, it is likely that some errors may occur, either in the inputs or in the outputs. In that case, and again particularly in the later parts of the learning process, it is likely that backprop will be contorting the weights so as to fit precisely around training patterns that are actually erroneous! This phenomenon is known as **over-fitting**.

This problem can to some extent be avoided by stopping learning early. How does one tell when to stop? One method is to partition the training patterns into two sets (assuming that there are enough of them). The larger part of the training patterns, say 80% of them, chosen at random, form the training set, and the remaining 20% are referred to as the **test set**. Every now and again during training, one measures the performance of the current set of weights on the test set. One normally finds that the error on the training set drops monotonically (that's what a gradient descent algorithm is supposed to do, after all). However, error on the test set (which will be larger, per pattern, than the error on the training set) will fall at first, then start to rise as the algorithm begins to overtrain. Best generalization performance is gained by stopping the algorithm at the point where error on the test set starts to rise.

generalized delta rule

An improvement on the in error backpropagation learning. If the learning rate (often denoted by the Greek letter *eta* is small, the backprop algorithm proceeds slowly, but accurately follows the path of steepest descent on the error surface. If *eta* is too large, the algorithm may "bounce off the canyon walls of the error surface" - i.e. not work well. This can be largely avoided by modifying the delta rule to include a momentum term:

$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta\, \delta_j(n) y_i(n)$$

in the notation of Haykin's text (*Neural networks - a comprehensive foundation*). The constant *alpha*is a termed the momentum constant and can be adjusted to achieve the best effect. The second summand corresponds to the standard delta rule, while the first summand says "add *alpha*times the previous change to this weight.

This new rule is called the generalized delta rule. The effect is that if the basic delta rule would be consistently pushing a weight in the same direction, then it gradually gathers "momentum" in that direction.

gradient descent

Understanding this term depends to some extent on the error surface metaphor.

When an artificial neural network learning algorithm causes the weights of the net to change, it will do so in such a way that the current point on the error surface will descend into a valley of the error surface, in a direction that corresponds to the steepest (downhill) *gradient* or slope at the current point on the error surface. For this reason, backprop is said to be a *gradient descent method*, and to perform *gradient descent in weight space*.

See also local minimum.

H

hidden layer

Neurons or units in a feedforward net are usually structured into two or more layers. The input units constitute the **input layer**. The output units constitute the **output layer**. Layers in between the input and output layers (that is, layers that consist of hidden units) are termed hidden layers.

In layered nets, each neuron in a given layer is connected by trainable weights to each neuron in the next layer.

hidden unit / node

A hidden unit in a neural network is a neuron which is neither an input unit nor an output unit.

hypothesis language

Term used in analysing machine learning methods. The hypothesis language refers to the notation used by the learning method to represent what it has learned so far. For example, in ID3, the hypothesis language would be the notation used to represent the decision tree (including partial descriptions of incomplete decision trees). In backprop, the hypothesis language would be the notation used to represent the current set of weights. In Aq, the hypothesis language would be the notation used to represent the class descriptions (e.g.

class1 <- size=large **and** colour **in** {red, orange}).

See also observation language.

I

ID3

A decision tree induction algorithm, developed by Quinlan. ID3 stands for "Iterative Dichotomizer (version) 3". Later versions include C4.5 and C5.

inhibitory connection

see weight.

input unit

An input unit in a neural network is a neuron with no input connections of its own. Its activation thus comes from outside the neural net. The input unit is said to have its value clamped to the external value.

instance

This term has two, only distantly related, uses:

1. An instance is a term used in machine learning particularly with symbolic learning algorithm, to describe a single training item, usually in the form of a description of the item, along with its intended classification.

2. In general AI parlance, an instance frame is a frame representing a particular individual, as opposed to a generic frame.

J

K

L

Laplace error estimate

this is described in the article on expected error estimates.

layer in a neural network

see article on feedforward networks.

learning program

Normal programs P produce the same output y each time they receive a particular input x. Learning programs are capable of improving their performance so that they may produce different (better) results on second or later times that they receive the same input x.

They achieve this by being able to alter their **internal state**, q. In effect, they are computing a function of two arguments, $P(x \mid q) = y$. When the program is in learning mode, the program computes a new state q' as well as the output y, as it executes. In the case of supervised learning, in order to construct q', one needs a set of inputs xi and corresponding target outputs zi (i.e. you want $P(xi \mid q) = zi$ when learning is complete). The new state function L is computed as:

$$L(P, q, ((x1,z1), ..., (xn, zn))) = q`'$$

See also unsupervised learning, observation language, and hypothesis language.

learning rate

a constant used in error backpropagation learning and other artificial neural network learning algorithms to affect the speed of learning. The mathematics of e.g. backprop are based on *small* changes being made to the weights at each step: if the changes made to weights are too large, the algorithm may "bounce around" the error surface in a counter-productive fashion. In this case, it is necessary to reduce the learning rate. On the other hand, the small the learning rate, the more steps it takes to get to the stopping criterion. See also momentum.

linear threshold unit (LTU)

A linear threshold unit is a simple artificial neuron whose output is its thresholded total net input. That is, an LTU with threshold T calculates the weighted sum of its inputs, and then outputs 0 if this sum is less than T, and 1 if the sum is greater than T. LTU's form the basis of perceptrons.

local minimum

Understanding this term depends to some extent on the error surface metaphor.

When an artificial neural network learning algorithm causes the total error of the net to descend into a valley of the error surface, that valley may or may not lead to the lowest point on the entire error surface. If it does not, the minimum into which the total error will eventually fall is termed a local minimum. The learning algorithm is sometimes referred to in this case as "trapped in a local minimum."

In such cases, it usually helps to restart the algorithm with a new, randomly chosen initial set of weights - i.e. at a new random point in weight space. As this means a new starting point on the error surface, it is likely to lead into a different valley, and hopefully this one will lead to the true (absolute) minimum error, or at least a better minimum error.

logistic function

The function

$$\text{phi}(x) = 1/(1 + \exp(-x))$$

which, when graphed, looks rather like a smoothed version of the step function

$$\text{step}(x) = 0 \text{ if } x < 0, = 1 \text{ if } x >= 0$$

It is used to transform the total net inputof an artificial neuronin some implementations of backprop-trained networks.

M

machine learning

Machine learning is said to occur in a program that can modify some aspect of itself, often referred to as its *state*, so that on a subsequent execution with the same input, a different (hopefully better) output is produced. See unsupervised learning and supervised learning, and also function approximation algorithms and symbolic learning algorithms.

momentum in backprop

See article on generalized delta rule.

multilayer perceptron (MLP)

See also feedforward network. Such a neural network differs from earlier perceptron-based models in two respects:

- most importantly, in their activation function, which consists of transforming the total net input by a sigmoid function, rather than simply thresholding it;
- using a multiple-layer model, with hidden units, rather than just a single input layer and a single output layer.

N

neural network

An artificial neural network is a collection of simple artificial neurons connected by directed weighted connections. When the system is set running, the activation levels of the input units is clamped to desired values. After this the activation is propagated, at each time step, along the directed weighted connections to other units. The activations of non-input neurons are computing using each neuron's activation function. The system might either settle into a stable state after a number of time steps, or in the case of a feedforward network, the activation might flow through to output units.

Learning might or might not occur, depending on the type of neural network and the mode of operation of the network.

neurode

see neuron.

neuron (artificial)

A simple model of a biological neuron used in neural networks to perform a small part of some overall computational problem. It has inputs from other neurons, with each of which is associated a weight - that is, a number which indicates the degree of importance which this neuron attaches to that input. It also has an activation function, and a bias. The bias acts like a threshold in a perceptron.

Also referred to a **neurode**, **node**, or **unit**.

node

see neuron, for a node in a neural network.
See graph for a node in a graph.

noisy data in machine learning

The term "noise" in this context refers to errors in the training data for machine learning algorithms. If a problem is difficult enough and complicated enough to be worth doing with machine learning techniques, then any reasonable training set is going to be large enough that there are likely to be errors in it. This will of course cause problems for the learning algorithm.

See also decision tree pruning and generalization in backprop.

O

observation language

Term used in analysing machine learning methods. The observation language refers to the notation used by the learning method to represent the data it uses for training. For example, in ID3, the observation language would be the notation used to represent the training instances, including attributes and their allowable values, and the way instances are described using attributes. In backprop, the observation language would be the notation used to represent the training patterns. In Aq, the observation language would again be the notation used to represent the instances, much as in ID3.

See also hypothesis language.

output unit

An output unit in a neural network is a neuron with no output connections of its own. Its activation thus serves as one of the output values of the neural net.

over-fitting

see article on generalization in backprop

P

perceptron

A perceptron is a simple artificial neuron whose activation function consists of taking the total net input and ouputting 1 if this is above a threshold *T*, and 0 otherwise.

Perceptrons were originally used as pattern classifiers, where the term pattern is here used not in the sense of training pattern, but just in the sense of an input pattern that is to be put into on of several classes. Perceptual pattern classifiers of this sort (not based on perceptrons!) occur in simple animal visual systems, which can distinguish between prey, predators, and neutral environmental objects.

See also perceptron learning, and the XOR problem.

perceptron learning

The perceptron learning algorithm:

1. All weights are initially set to zero.
2. For each training example:
    - if the perceptron outputs 0 when it should output 1, then add the input vector to the weight vector.
    - if the perceptron outputs 1 when it should output 0, then subtract the input vector to the weight vector.
3. Repeat step 2 until the perceptron yields the correct result for each training example.

propositional learning systems

A propositional learning system represents what it learns about the training instances by expressions equivalent to sentences in some form of logic (e.g.

class1 <- size=large **and** colour **in** {red, orange})

. See Aqand covering algorithm.

pruning decision trees

The data used to generate a decision tree, using an algorithm like the ID3 tree induction algorithm, can include noise - that is, instances that contain errors, either in the form of a wrong classification, or in the form of a wrong attribute value. There could also be instances whose classification is problematical because the attributes available are not sufficient to discriminate between some cases.

If for example, a node of the tree contains, say, 99 items in class C1 and 1 in class C2, it is plausible that the 1 item in class C2 is there because of an error either of classification or of feature value. There can thus be an argument for regarding this node as a leaf node of class C1. This termed *pruning* the decision tree.

The algorithm given in lectures for deciding when to prune is as follows:
At a branch node that is a candidate for pruning:

1. Approximate expected error for the node.
2. Approximate backed-up error from the children assuming that we do not prune.
3. If expected error is less than backed-up error, then prune.

Q

R

recurrent network

A recurrent network is a neural network in which there is at least one cycle of activation flow. To put it another way, underlying any neural network there is a directed graph, obtained by regarding the neurons as nodes in the graph and the weights as directed edges in the graph. If this graph is not acyclic, the network is recurrent.

A **recurrent connection** is one that is part of a directed cycle, although term is sometimes reserved for a connection which is clearly going in the "wrong" direction in an otherwise feedforward network.

Recurrent networks include **fully recurrent networks** in which each neuron is connected to *every* other neuron, and **partly recurrent networks** in which greater or lesser numbers of recurrent connections exist. See also simple recurrent network.

This article is included for general interest - recurrent networks are not part of the syllabus of COMP9414 Artificial Intelligence.

S

sequence prediction tasks

Feedforward networks are fine for classifying objects, but their units (as distinct from their weights) have no memory of previous inputs. Consequently they are unable to cope with sequence prediction tasks - tasks like predicting, given a sequence of sunspot activity counts, what the sunspot activity for the next time period will be, and financial prediction tasks (e.g. given share prices for the last *n* days, and presumably other economic data, etc., predict tomorrow's share price).

Simple recurrent nets *can* tackle tasks like this, because they do have a kind of memory for recording information derived from activation values from previous time steps.

This article is included for general interest - sequence prediction tasks are not part of the syllabus of COMP9414 Artificial Intelligence.

sigmoidal nonlinearity

Another name for the logistic function and certain related functions (such as *tanh*(*x*)). Sigmoidal functions are a type of squashing function. They are called because *sigma* is the Greek letter "s", and the logistic functions look somewhat like a sloping letter "s" when graphed.

simple recurrent network

A simple recurrent network is like a feedforward network with an input layer, and output layer, and a single hidden layer, except that there is a further group of units called **state units** or **context units**. There is one state unit for each hidden unit. The activation function of the state unit is as follows: the activation of a state unit in time step *n* is the same of that of the corresponding hidden unit in time step *n*-1. That is, the state unit activations are copies of the hidden unit activations from the previous time step. Each state unit is also connected to each hidden unit by a sequence prediction tasks.

See also recurrent networks.

This article is included for general interest - simple recurrent networks are not part of the syllabus of COMP9414 Artificial Intelligence.

splitting criterion in ID3

The point of the ID3 algorithm is to decide the best attribute, out of those not already used, on which to split the training instances that are classified to a particular branch node.

The algorithm, in outline, is as follows:

1. if all the instances belong to a single class, there is nothing to do (except create a leaf node labelled with the name of that class).
2. otherwise, for each attribute that has not already been used, calculate the information gain that would be obtained by using that attribute on the particular set of instances classified to this branch node.
3. use the attribute with the greatest information gain.

This leaves the question of how to calculate the information gain associated with using a particular attribute A. Suppose that there are k classes C_1, C_2, ..., C_k, and that of the N instances classified to this node, I_1 belong to class C_1, I_2 belong to class C_2, ..., and I_k belong to class C_k.
Let $p_1 = I_1/N$, $p_2 = I_2/N$, ..., and $p_k = I_k/N$.
The initial entropy E at this node is:

$$-p_1\log_2(p_1) - p_2\log_2(p_2) \ldots -p_k\log_2(p_k).$$

Now split the instances on each value of the chosen attribute A. Suppose that there are r attribute values for A, namely a_1, a_2, ..., a_r.
For a particular value a_j, say, suppose that there are $J_{j,1}$ instances in class C_1, $J_{j,2}$ instances in class C_2, ..., and $J_{j,k}$ instances in class C_k, for a total of J_j instances having attribute value a_j.
Let $q_{j,1} = J_{j,1}/J_j$, $q_{j,2} = J_{j,2}/J_j$, ..., and $q_{j,k} = J_{j,k}/J_j$.
The entropy E_j associated with this attribute value a_j this position is:

$$-q_{j,1}\log_2(q_{j,1}) - q_{j,2}\log_2(q_{j,2}) \ldots -q_{j,k}\log_2(q_{j,k}).$$

Now compute:

$$E - ((J_1/N).E_1 + (J_2/N).E_2 + \ldots + (J_r/N).E_r).$$

This is the **information gain for attribute *A***.
Note that $J_j/N$is the estimated probability that an instance classified to this node will have value $a_j$for attribute A. Thus we are weighting the entropy estimates $E_j$by the estimated probability that an instance has the associated attribute value.

In terms of the example used in the lecture notes, (see also calculations in lecture notes), $k = 2$ since there are just two classes, *positive* and *negative*. $I_1 = 4$ and $I_2 = 3$, and $N =7$, and so $p_1 = 4/7$ and $p_2 = 3/7$, and $E = -p_1\log_2(p_1) - p_2\log_2(p_2) = -(4/7)*\log_2(4/7) - (3/7)*\log_2(3/7)$. In the example, the first attribute A considered is *size*, and the first value of *size* considered, *large*, corresponds to a_1, in the example in the

lecture notes, so $J_{1,1} = 2 = J_{1,2}$, and $J_1 = 4$. Thus $q_{1,1} = J_{1,1}/J_1 = 2/4 = ½$, and $q_{1,2} = J_{1,2}/J_1 = 2/4 = ½$, and $E_1 = -q_{1,1}\log_2(q_{1,1}) -q_{1,2}\log_2(q_{1,2}) = -½*\log_2(½) - ½*\log_2(½) = 1$.
Similarly $E_2 = 1$ and $J_2 = 2$ (*size = small*), and $E_3 = 0$ and $J_3 = 1$ (*size = medium*) so the final information gain,

$$E - ((J_1/N).E_1 + (J_2/N).E_2 + ... + (J_r/N).E_r)$$
$$= E - ((4/7)*E_1 + (2/7)*E_2 + (1/7)*E_3)$$

which turned out to be about 0.13 in the example.

squashing function

One of several functions, including sigmoid and step functions (see threshold used to transform the total net input to a neuron to obtain the final output of the neuron.

stopping criterion in backprop

Possible stopping criteria in error backpropagation learning include:

- total error of the network falling below some predetermined level;
- a certain number of epochs having been completed;

Combinations of the two (e.g. whichever of the two occurs first) and other stopping conditions are possible. See the reference by Haykin (*Neural networks: a comprehensive foundation* p. 153) for more details.

supervised learning

Supervised learning is a kind of machine learning where the learning algorithm is provided with a set of inputs for the algorithm along with the corresponding correct outputs, and learning involves the algorithm comparing its current actual output with the correct or target outputs, so that it knows what its error is, and modify things accordingly.

Contrast unsupervised learning.

symbolic learning algorithms

Symbolic learning algorithms learn concepts by constructing a symbolic expression (such as a decision tree, as in Aq) that describes a class (or classes) of objects. Many such systems work with representations equivalent to first order logic.

Such learning algorithms have the advantage that their internal representations and their final output can be inspected and relatively easily understood by humans.

See also function approximation algorithms.

synapse

A synapse, in a biological neuron, is a connection between the axon of one neuron and the dendrite of another. It corresponds to a weight in an artificial neuron. Synapses have varying strengths and can be changed by learning (like weights). The operation mechanism of synapses is biochemical in nature, with *transmitter substances* crossing the tiny "synaptic gap" between axon and dendrite when the axon fires.

synchronous vs **asynchronous**

See asynchronous

T

target output

The target output of an artificial neuron is the output provided with the training pattern. The difference between this and the actual output of the neuron is the pattern error. This is used in

threshold

One of the parameters of a perceptron . The output of the perceptron is 0 if the total net input of the perceptron is less than its threshold *T*, otherwise it is 1.

Compare bias.

total net input

The total net input to a artificial neuron refers to the weighted sum of the neuron's inputs (that is, the sum of the product of each input with the weight associated with that input. This quantity is then usually transformed in some way (see squashing function) to obtain the neuron's output.

See also activation function.

total sum-squared error (TSS)

A measure of how well a backprop-trained neural network is doing at a particular point during its learning.

Given a training pattern, its squared error is obtained by squaring the difference between the target output of an output neuron and the actual output. The sum-squared error, or **pattern sum-squared error (PSS)**, is obtained by adding up the sum-squared errors for each output neuron. The total sum-squared error is obtained by adding up the PSS for each training pattern.

trainable weight

In artificial neural networks that use weighted connections, some of those connections may have fixed values - i.e. they are specified as not being subject to change when the network is trained, e.g. using the error backpropagation learning algorithm.

The other weights, the ones that *are* subject to change when the net is learning, are referred to as *trainable weights*.

training pattern

This term is used to refer to a set of input values and the corresponding set of desired or target output values for use in training an artificial neural network. Usually, a largish number of training patterns would be used in the training of any genuinely useful neural network.

In toy problems like the XOR problem, only a few training patterns (in the case of XOR, just 4) may be used.

tree induction algorithm

This article describes the basic tree induction algorithm used by ID3 and successors. The basic idea is to pick an attribute *A* with values a_1, a_2, ..., a_r, split the training instances into subsets S_{a_1}, S_{a_2}, ..., S_{a_r} consisting of those instances that have the corresponding attribute value. Then if a subset has only instances in a single class, that part of the tree stops with a leaf node labelled with the single class. If not, then the subset is split again, recursively, using a different attribute.

This leaves the question of how to choose the *best* attribute to split on at any branch node. This issue is handled in the article on splitting criterion in ID3.

U

unit

see neuron.

unsupervised learning

Unsupervised learning signifies a mode of machine learning where the system is not told the "right answer" - for example, it is not trained on pairs consisting of an input and the desired output. Instead the system is given the input patterns and is left to find interesting patterns, regularities, or clusterings among them. To be contrasted to supervised learning, as in error backpropagation and ID3, where the system *is* told the desired or target outputs to associate with each input pattern used for training.

V

W

weight

A weight, in a artificial neural network, is a parameter associated with a connection from one neuron, *M*, to another neuron *N*. It corresponds to a synapse in a biological neuron, and it determines how much notice the neuron *N* pays to the activation it receives from neuron *N*. If the weight is positive, the connection is called **excitatory**, while if the weight is negative, the connection is called **inhibitory**. See also to a neuron.

weight space

See the article on error surfaces in weight space.

windowing in ID3

Windowing, in ID3, and similar algorithms, is a way of dealing with really large sets of training instances.

Without windowing, such an algorithm can be really slow, as it needs to do its information gain calculations (see tree induction algorithms) over huge amounts of data.

With windowing, training is done on a relatively small sample of the data, and then checked against the full set of training data.

Here is the windowing algorithm:

1. Select a sample S of the training instances at random - say 10% of them. The actual proportion chosen would need to be small enough that ID3 could run fairly fast on them, but large enough to be representative of the whole set of examples.
2. Run the ID3 algorithm on the set of training instances to obtain a decision tree.
3. Check the decision tree on the full data set, to obtain a set E of training instances that are misclassified by the tree obtained.
4. If E is empty, stop.
5. Let the new set of training instances S' be the union of S and E.
6. Go to step 2 and run the ID3 calculations with S'.

X

XOR problem

p XOR q is a binary logical operator which can be defined either as **not** (p **equiv** q), or as **not** p **and** q **or** p **and not** q. It rose to fame in the late 60's when Minsky and Papert demonstrated that perceptron-based systems cannot learn to compute the XOR function. Subsequently, when backprogation-trained multi-layer perceptrons were introduced, one of the first tests performed was to demonstrate that they could in fact learn to compute XOR. No one in their right mind would probably ever have cause to compute XOR in this way in practice, but XOR and generalizations of XOR remain popular as tests of backpropagation-type systems.

Y

Z

Bill Wilson's Home Page

Bill Wilson is an Associate Professor in the Artificial Intelligence Group, School of Computer Science and Engineering, University of NSW, and also currently Associate Head of School.

Research

Research Publications	Neural Networks	Natural Language Processing and collection of ill-formed sentences
Research Students past and present	Cognitive Modelling	Combinatorial Search Algorithms

---

Teaching

COMP9414 Artificial Intelligence	COMP9444 Neural Networks	For consulting hours see course home page

On-line Dictionaries developed for COMP9414 Artificial Intelligence			
Prolog	Artificial Intelligence	Machine Learning	Natural Language Programming

---

Administration

Academic Staff FAQ(~100K) New Staff Notification Form	Teaching Committee Home Page	Teaching allocation
Timetables & SE/CE/CS/ BioInf programs	Interpreting UAI cutoffs	Grievance handling
xload and xpref	Yellow form	Casual Academic's Guide
Current Special Consideration Rules	UNSW online handbook	Plagiarism Detection Support
UNSW A-Z Guide	UNSW Assessment Policy and Assessment Gotchas	UNSW Result Codes

---

My official School of CSE Information Page | School of CSE Academics

---

Quick Bio...

Contact Information

Email:	
Phone:	+61 2 9385 6876
Fax:	+61 2 9385 5995
Office:	Room 405, Building K17 (CSE Building), Kensington Campus, UNSW.

School of Computer Science & Engineering
The University of New South Wales
Sydney 2052, AUSTRALIA

UNSW's CRICOS Provider No. is 00098G
Last updated:

Publications of Bill Wilson

In Chronological Order | Sorted by Type

77 78 79 80 82 85 87 88 89 90 91 92 93 94 95 96 97 98 99 00 01 02 03 04 05 06 to appear

My Home Page

Some collaborators' pages: Halford | Phillips | Street

1. Elizabeth J. Morgan, Deborah J. Street and William H. Wilson, *Solution Manual to Combinatorial Theory: an Introduction*, 68 pages, Winnipeg: Charles Babbage Research Centre, 1977.

2. William H. Wilson, When $a^r - b^r$ divides $(a - b)^s$, *Utilitas Mathematica* **13** (1978) 139-141.

3. William H. Wilson, A functorial version of a construction of Hochschild and Mostow for representations of Lie algebras, *Bulletin of the Australian Mathematical Society* **18** (1978) 95-98.

4. William H. Wilson, On coinduced corepresentations, *Bulletin of the Australian Mathematical Society* **18** (1978) 99-103.

5. William H. Wilson, Induced representations of Lie algebras, in *Topics in Algebra*, Lecture Notes in Mathematics No. 697, 197-204, (Springer-Verlag, New York, 1979.)

6. William H. Wilson, On induced representations of Lie algebras, groups, and coalgebras, *Journal of Algebra* **58** (1979) 37-50. html (68K)

7. Graeme S. Halford and William H. Wilson, A category theory approach to cognitive development, *Cognitive Psychology* **12** (1980), 356-411. (PDF - about 135K) *Keywords*: structure mapping, cognitive development, cognitive system level. Here is the abstract.

8. Deborah J. Street and William H. Wilson, On directed balanced incomplete block designs with block size five, *Utilitas Mathematica* **18** (1980) 161-174.

9. William H. Wilson, Christopher F. Burrows and Michael M. Sidhom, Design of an interactive financial planning package, *Australian Computer Journal* **14** (1982) 26-31.

10. Deborah J. Street and William H. Wilson, Balanced designs for two- variety competition experiments, *Utilitas Mathematica* **28** (1985) 113-120.

11. William H. Wilson, Context tracking for natural language processing, *Australian Computer Science Communications* **9** No. 1 (February 1987) 418-428.

12. Luyuan Fang, Tao Li, and William H. Wilson, Implementing semantic networks in stochastic neural nets, *Proceedings of the Australian Joint Conference on Artificial Intelligence* 1987, 323-

332. This paper is reprinted in *Artificial Intelligence Developments and Applications*, edited by J. Gero and R. Stanton, North Holland, 1988, pages 351-361.

13. Luyuan Fang, Tao Li, and William H. Wilson, Storing semantic information in stochastic neural nets, *Australian Computer Science Communications* **10** No. 1 (February 1988), 68-77.

14. Luyuan Fang and William H. Wilson, Character recognition on a computational neural network, in *Proceedings of the Australian Joint Conference on Artificial Intelligence* 1988, 431-438. Also published in *Lecture Notes in Artificial Intelligence (Subseries of Lecture Notes in Computer Science)* **406**, 424-431, Berlin: Springer-Verlag, 1990.

15. Luyuan Fang, William H. Wilson, Tao Li, and Chris Barter, A computational neural network for recognizing character patterns, *Proceedings of the International Computer Science Conference 1988 (Hong Kong)*, 401-405.

16. Tao Li, Luyuan Fang, and William H. Wilson, Running rule-based expert systems on parallel processors, *Knowledge Based Systems*, **2** (1989) 27-36.

17. William H. Wilson, Deborah J. Street and D.A. Maelzer, Hexagonal grids for choice experiments, *Journal of Combinatorial Mathematics and Combinatorial Computing* **5** (1989) 63-68.

18. Luyuan Fang, William H. Wilson, and Tao Li, Experiments on sequence programming with neural networks, *Proceedings of the Third Pan Pacific Computer Conference 1989 (Beijing)*, 164-170.

19. Luyuan Fang and William H. Wilson, A study of sequential pattern processing on neural networks, *Proceedings of the International Joint Conference on Neural Networks 1989 (Washington, D.C.) (IJCNN-89)*, 599.

20. Luyuan Fang, William H. Wilson, and Tao Li, Mean field annealing neural net for quadratic assignment, *INNC-90-PARIS Proceedings*, volume 1, 1990, 282-286.

21. Deborah J. Street, J.A. Eccleston, and William H. Wilson, Tables of small optimal repeated measurements designs, *Australian Journal of Statistics* **32(3)** (1990) 345-359.

22. Tao Li, Luyuan Fang, and William H. Wilson, Parallel approximate solution of 0/1 knapsack optimization on competitive neural networks, pages 281-286 in *Parallel Computing 89*, edited by D.J. Evans, G.R. Joubert, and F.J. Peters, North-Holland, 1990.

23. Luyuan Fang, William H. Wilson, and Tao Li, A neural network for job sequencing, pages 253-256 in *Parallel Processing in Neural Systems and Computers*, edited by Rolf Eckmiler, Georg Hartmann, and Gert Hauske, Amsterdam: North-Holland, 1990.

24. A. Rahardja, A. Sowmya, and W.H. Wilson, Component versus holistic approach to facial expression formation and recognition of facial expressions in images, *Proceedings of the Conference on Neural Networks for Vision and Image Processing, Wang Institute, Boston University May 1991*, 56-57.

25. J. Wiles, G.S. Halford, J.E.M. Stewart, M.S. Humphreys, J.D. Bain, and W.H. Wilson, Tensor Models: a creative basis for memory retrieval and analogical mapping, page 46-48 in *AI, Reasoning, and Creativity, edited T. Dartnall*, Brisbane: Griffith University, 1991.

26. A. Rahardja, A. Sowmya, and W.H. Wilson, A neural network approach to component versus holistic recognition of facial expressions in images, *Proceedings of SPIE Symposium on Intelligent Robots and Computer Vision X: Algorithms and Techniques, (Boston, MA, November 1991)* edited David P. Casasent, SPIE Proceedings Vol. 1607, pages 62- 70.

27. William H. Wilson, Dealing with unknown words: Classifying unknown letter-strings using trigram analysis, *Australian Computer Science Communications* **14** No. 1 (February 1992) 981-988.

28. William H. Wilson, A comparison of architectural alternatives for recurrent networks, *Proceedings of the Fourth Australian Conference on Neural Networks, ACNN'93,* Melbourne, 1-3 February 1993, 189-192. (PDF - 51K)

29. Graeme S. Halford, Janet Wiles, Michael S. Humphreys, and William H. Wilson, Parallel distributed processing approaches to creative reasoning: Tensor models of memory and analogy, pp. 57-60 in *AI and Creativity* ed.T. Dartnall, S. Kim, R. Levinson and D. Subramanian, Artificial Intelligence and Creativity, Papers from the 1993 Spring Symposium, Technical Report SS-93-01. Menlo Park, Ca: AAAI Press, 1993. ISBN 0-92980-38-5.

30. Graeme S. Halford and William H. Wilson, Creativity and capacity for representation: Why are humans so creative, *AISB Quarterly* **85** Autumn 1993, 32-41. [to be republished in a AAAI book edited by Terry Dartnall]

31. Adnan Amin and W.H. Wilson, Hand-printed character recognition system using artificial neural networks, in *Proceedings of the Second International Conference on Document Analysis and Recognition, Japan, October 1993*, (Los Alamitos CA: IEEE Computer Society Press) 943- 946.

32. Graeme S. Halford, William H. Wilson, Jian Guo, Ross W. Gayler, Janet Wiles, and J.E.M. Stewart, Connectionist implications for processing capacity limitations in analogies, pages 363-415 in K.J. Holyoak and J. Barnden (editors) *Advances in Connectionist and Neural Computation Theory: Volume 2: Analogical Connections*, Norwood, NJ: Ablex, 1994. Reprint (HTML format - about 550K including figures)

33. William H. Wilson and Graeme S. Halford, Robustness of tensor product networks using distributed representations, p. 258-261 in *Proceedings of the Fifth Australian Conference on Neural Networks, ACNN'94*, edited by A.C. Tsoi & T. Downs. 31 January-2 February 1994. Brisbane: University of Queensland. (PDF - about 730K). This *is* a different paper from the 1998 paper of the same name.

34. Janet Wiles, Graeme S. Halford, Julie E.M. Stewart, Michael S. Humphreys, John D. Bain, and William H. Wilson, Tensor Models: a creative basis for memory retrieval and analogical mapping, pages 147-161 in *Artificial Intelligence and Creativity: an Interdisciplinary Approach*, edited by T. Dartnall, Kluwer, 1994 .

35. Kyongho Min and William H. Wilson, Chart parser for ill-formed input sentences, *Language Teaching and Research*, **XXIII** (August 1994), Language Research Centre, Chonnam National University, 141-154.

36. William H. Wilson, Stability of learning in classes of recurrent and feedforward networks, in *Proceedings of the Sixth Australian Conference on Neural Networks, ACNN'95*, edited by Margaret Charles & Cyril Latimer, Sydney, 6-8 February 1995, 142-145. (PDF - 95K)

37. Kyongho Min and William H. Wilson, Syntactic recovery and spelling correction of ill-formed sentences, *3rd Conference of the Australasian Cognitive Science Society (CogSci'95)*, Brisbane, April 1995, 81. PDF(45K) or (Microsoft Word RTF format - about 137K)

38. Steven Phillips, Graeme S. Halford, and William H. Wilson, The processing of associations versus the processing of relations and symbols: a systematic comparison, *3rd Conference of the Australasian Cognitive Science Society (CogSci'95)*, Brisbane, April 1995, 96.

39. Graeme S. Halford, William H. Wilson, and Matthew McDonald, Complexity of Structure Mapping in Human Analogical Reasoning: A PDP Model, *Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society*, Pittsburgh, PA, 22-25 July 1995, edited by J.D. Moore & J.F. Lehrman, 597-601, Mahwah, NJ: Lawrence Erlbaum Associates. ISBN 0-8058-2159-7; ISSN 1047-1316. (PDF - about 20K)

40. Steven Phillips, Graeme S. Halford, and William H. Wilson, The processing of associations versus the processing of relations and symbols: a systematic comparison, *Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society*, Pittsburgh, PA, 22-25 July 1995, edited by J.D. Moore & J.F. Lehrman, 688-691. ISBN 0-8058-2159-7; ISSN 1047-1316, Mahwah, NJ: Lawrence Erlbaum Associates. (PDF - about 20K)

41. William H. Wilson, Deborah J. Street and Graeme S. Halford, Solving proportional analogy problems using tensor product networks with random representations, *1995 IEEE International Conference on Neural Networks Proceedings, ICNN'95*, Perth, Australia, 27 November-1

December 1995, edited by Yianni Attikiouzel, 2971-2975, ISBN 0 646 26352 8 (CD-ROM), ISBN 0 780 32769 1 (paper). (PDF - about 172K)

42. Kyongho Min and William H. Wilson, Are Efficient Natural Language Parsers Robust?, *Proceedings of the Eighth Australian Joint Conference on Artifical Intelligence (AI '95)*, Canberra 13-17 November 1995, 283-290. Edited Xin Yao. World Scientific, Singapore, 1995. ISBN 981-02-2484-2

43. Kyongho Min and William H. Wilson, Hierarchical Multiple Error Recovery based on Chart Parsing, *Proceedings of the Third Natural Language Processing Pacific Rim Symposium (NLPRS95)*, Edited by Key-Sun Choi, Seoul, Korea, 4-6 December, 1995, 314-319.

44. Gedeon TD, Wilson, WH, Mann, GA and Bustos, RA Assessment of Basic Competency Acquisition versus Deep Learning in an undergraduate Computing subject, in Hewson, L and Toohey, S The Changing University, *Proceedings UNSW Education '95 Conference*, pp. 171-183, Sydney, 1995.

45. William H. Wilson, Learning Performance of Networks like Elman's Simple Recurrent Networks but having Multiple State Vectors, Cognitive Modelling Workshop, Seventh Australian Conference on Neural Networks, Australian National University Canberra, 9 April 1996. Case Study Number 4. Also a part of Noetica: Open Forum **2**:7 (1996) Memory, Time, Change and Structure in ANNs: Distilling Cognitive Models into their Functional Components. (Papers written for the Cognitive Modelling Workshop of the Seventh Australian Conference on Neural Networks, Australian National University, Canberra, April 1996) Also published in AISB Quarterly **97** April 1997, 39-44.

46. T.D. Gedeon, G.A. Mann, W.H. Wilson, and R.A. Bustos, Separation of Basic Competency Acquisition from Deep Learning in Teaching and Assessment in an undergraduate Computing subject, *Australasian Conference on Computer Science Education*, Sydney, 1996. 287-294.

47. G.S. Halford, W.H. Wilson, and S. Phillips, Human analogical reasoning capacity: Towards a neural net model, *International Journal of Psychology*, **31**(3/4) 1996, 443 ISSN 0020-7594.

48. Adnan Amin, Sankaran Iyer, William H. Wilson, Recognition of hand-printed Latin characters based on a structural approach with a neural network classifier, *Journal of Electronic Imaging* **6**(3) 1997, 303-310. ISSN 1017-9909.

49. Kyongho Min and William H. Wilson, Integrated correction of ill-formed sentences, pp. 369-378 in Advanced Topics in Artificial Intelligence *10th Australian Joint Conference on Artifical Intelligence (AI '97), Lecture Notes in Artificial Intelligence 1342*, edited by Abdul Sattar, Berlin: Springer, 1997. ISBN 3-540-63797-4. PDF (32K)

50. G.S. Halford, W.H. Wilson and S. Phillips, (1997), Abstraction: Nature, costs, and benefits, *International Journal of Educational Research* **27**:1, 21-35. ISSN 0883-0355 0191-765X. PDF

51. W.H. Wilson, & G.S. Halford, Robustness of tensor product networks using distributed representations, *Proceedings of the Ninth Australian Conference on Neural Networks*, Brisbane, Australia, 11-13 February 1998, edited by Tom Downs, Marcus Frean, and Marcus Gallagher. pp. 47-51. ISBN 1-86499-026-0. PDF - about 29KB. This *is* a different paper from the 1994 paper of the same name.

52. Phillips, S., Halford, G. S., and Wilson, W. H. (1998). What changes in children's drawing procedures? Relational complexity as a constraint on representational redescription. *Cognitive Studies: Bulletin of the Japanese Cognitive Science Society* **5**(2) 1998 33-42. ISSN 1341-7924. PDF (216K)

53. Halford, G.S., Wilson, W.H. & Phillips, S. (1998) Relational processing in higher cognition: Implications for analogy, capacity and cognitive development. In K. Holyoak, D. Gentner, & B. Kokinov, (Eds.) *Advances in analogy research: Integration of Theory and Data from the Cognitive, Computational, and Neural Sciences*, pp. 57-73. Sofia, Bulgaria, New Bulgarian University. NBU Series in Cognitive Science. ISBN: 954-535-200-0. PDF

54. Kyongho Min and William H. Wilson, Integrated control of chart items for error repair, *ACL-Coling98 Proceedings* (Proceedings of the Joint 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics, August 10-14, Universite de Montreal), 862-868, 1998. PDF - 41KB

55. G.S. Halford, W.H. Wilson and S. Phillips, Processing capacity defined by relational complexity: Implications for comparative, developmental and cognitive psychology, *Behavioral and Brain Sciences* vol. **21**(6) (1998) 803-831. ISSN 0140-525X. PDF version (376K) or GZipped postscript - 1.09Mb

56. G.S. Halford, W.H. Wilson and S. Phillips, Authors' response: Relational complexity metric is effective when assessments are based on actual cognitive processes, *Behavioral and Brain Sciences* vol. **21**(6) (1998) 848-864. ISSN 0140-525X. PDF (180K).

57. Kyongho Min and William H. Wilson, Syntactic recovery and spelling correction of ill-formed sentences, pp. 293-306 in *Perspectives on Cognitive Science: Theories, Experiments, and Foundations, vol. 2*, edited by Janet Wiles and Terry Dartnall, Stanford, Connecticut: Ablex Publishing Corporation, 1999. ISBN 1-56750-382-9 and 1-56750-383-7.

58. Wilson, W.H., Halford, G.S. & Phillips, S.A., The properties of higher order cognitive processes and how they can be modelled in neural networks, in *Proceedings of the Fourth Conference of the*

Australasian Cognitive Science Society, 1999, ed. R. Heath, B. Hayes, A. Heathcote, C. Hooker. `file:///Cogsc97/wilson.pdf` on Proceedings CD-ROM, available from Richard Heath, Dept of Psychology, University of Newcastle, NSW Australia. (`heath@psychology.newcastle.edu.au`). Note - the conference was held in Sept. 1997, but the proceedings did not become available until a couple of years later. ISBN 0 7259 1059 3. PDF - 20KB

59. Halford, G.S., Wilson, W.H. & Phillips, S.A., A conceptual complexity metric based on representational rank, in *Proceedings of the Fourth Conference of the Australasian Cognitive Science Society*, ed. R. Heath, B. Hayes, A. Heathcote, C. Hooker, 1999. `file:///Cogsc97/halford.pdf` on Proceedings CD-ROM, available from Richard Heath, Dept of Psychology, University of Newcastle, NSW Australia. (`heath@psychology.newcastle.edu.au`). Note - the conference was held in Sept. 1997, but the proceedings did not become available until a couple of years later. ISBN 0 7259 1059 3. (Postscript - about 133K)

60. Gray, B., Halford, G.S., Wilson, W.H. & Phillips, S.A., A neural net model for mapping hierarchically structured analogs, in *Proceedings of the Fourth Conference of the Australasian Cognitive Science Society*, ed. R. Heath, B. Hayes, A. Heathcote, C. Hooker, 1999. `file:///Cogsc97/31half.pdf` on Proceedings CD-ROM, available from Richard Heath, Dept of Psychology, University of Newcastle, NSW Australia. (`heath@psychology.newcastle.edu.au`). Note - the conference was held in Sept. 1997, but the proceedings did not become available until a couple of years later. ISBN 0 7259 1059 3. (Postscript - about 98K)

61. Kyongho Min, William H. Wilson, and Yoo-Jin Moon, Typographical and orthographical spelling error correction, pp. 1781-1785, *Proceedings of Second International Conference on Language Resources and Evaluation (LREC-2000)*, Athens, Greece: European Language Resources Association, 2000. (PDF - about 90K)

62. Wilson, W.H., Halford, G.S., Gray, B., and Phillips, S.A., The STAR-2 Model for Mapping Hierarchically Structured Analogs, pp. 125-159 in Gentner, D., Holyoak, K. J., & Kokinov, B. N. (Eds.) *The analogical mind: Perspectives from cognitive science*. Cambridge, MA: MIT Press, 2001. ISBN 0-262-57139-0 PDF

63. William H. Wilson, Nadine Marcus, and Graeme S. Halford, Access to relational knowledge: a comparison of two models, pp. 1142-1147 in Johanna D. Moore and Keith Stenning (eds) *Proceedings of the 23rd Annual Conference of the Cognitive Science Society*, Edinburgh, Scotland, 1-4 August 2001. Mahwah, NJ: Lawrence Erlbaum Associates, ISBN 0-8058-4152-0, ISSN 1047-1316. PDF (74K)

64. Graeme S. Halford, Steven Phillips, and William H. Wilson, Processing capacity limits are not

explained by storage limits, *Behavioral and Brain Sciences* **24**(1) (2001) 123. This item is a commentary on the following BBS Target article in the same issue: Nelson Cowan: The Magical Number 4 in Short-term Memory: a Reconsideration of Mental Storage Capacity. PDF

65. Deborah J. Street and William H. Wilson, Resolvable designs for resolving disputes, *Journal of Combinatorial Mathematics and Combinatorial Computing* **38** 139-148, 2001. Winnipeg, Ontario: Charles Babbage Research Center. ISSN 0835-3026. Keywords: design dispute resolution PDF(110K)

66. Graeme S. Halford and William H. Wilson, Creativity, Relational Knowledge and Capacity: Why are Humans So Creative? pages 153-180 in Terry Dartnall, ed. *Creativity, Cognition and Knowledge: An Interaction.* Praeger: Westport, Connecticut, 2002. ISBN 0-275-97680-7 0-275-97681-5 (pbk.)

67. Kyongho Min, William H. Wilson, and Yoo-Jin Moon, Preferred Document Classification for a Highly Inflectional/Derivational Language, 12-23 in McKay, Bob and Slaney, John (eds), *AI2002: Advances in Artificial Intelligence, 15th Australian Joint Conference on Artificial Intelligence*, Canberra, Australia, December 2002. *Lecture Notes in Artificial Intelligence* **2557**, Springer, 2002. ISBN 3-540-00197-2.

68. Yuk W. Cheng, Deborah J. Street, and W.H. Wilson, Two stage generalized simulated annealing for the construction of changeover designs, 69-79 in *Designs 2002*, ed. W.D. Wallis. Boston: Kluwer Academic Publishers, 2003. ISBN 1-4020-7599-5.

69. Nadine Marcus, Bill Wilson, Computational model of the evolution of a cognitive architecture, ICCS/ASCS-2003: Joint International Conference on Cognitive Science, Sydney, Australia 13 - 17 July 2003 Abstract [Unrefereed conference]

70. Jane Brennan, Eric Martin and William H. Wilson, A Theory of Proximity Relations, ICCS/ASCS-2003: Joint International Conference on Cognitive Science, Sydney, Australia 13 - 17 July 2003 Abstract [Unrefereed conference]

71. Graeme S Halford, William H Wilson, Steven Phillips, Functionally Structured Cognitive Processes: An Intermediate Level Between Associations and Rules, ICCS/ASCS-2003: Joint International Conference on Cognitive Science, Sydney, Australia 13 - 17 July 2003 Abstract [Unrefereed conference]

72. Kyongho Min, William H. Wilson, Effectiveness of Syntactic Information for Document Classification, pp. 992-1002, in T.D. Gedeon and L.C.C. Fung, (eds), *Proceedings of the 16th Australian Conference of AI, Perth, Australia, December 2003, Lecture Notes on Artificial Intelligence* **2903**, Berlin: Springer, 2003. ISBN 3-540-20646-9.

73. Kyongho Min, William H. Wilson, and Yoo-Jin Moon, Korean Compound Noun Term Analysis Based on a Chart Parsing Technique pp. 186-195, in T.D. Gedeon and L.C.C. Fung, (eds), *Proceedings of the 16th Australian Conference of AI, Perth, Australia, December 2003, Lecture Notes on Artificial Intelligence* **2903**, Berlin: Springer, 2003. ISBN 3-540-20646-9.

74. Kyongho Min, William H. Wilson, and Yoo-Jin Moon, Various Factors Influencing Document Classification, pp. 50-59, in M.Mohammadian, (ed.), *CIMCA 2004 Proceedings, Gold Coast, Australia*, 12-14 July 2004, ISBN 1-740-88188-5.

75. Kyongho Min, William H. Wilson, and Yoo-Jin Moon, Syntactic and Semantic Disambiguation of Numeral Strings Using an N-gram Method, pp. 82-91 in Shichao Zhang and Ray Jarvis (eds) *AI 2005: Advances in Artificial Intelligence Proceedings of 18th Australian Joint Conference on Artificial Intelligence*, Sydney, Australia, December 2005. *Springer Lecture Notes in Artificial Intelligence* **3809**. ISBN 3-540-30462-2; ISSN 0302-9743

To appear

- Graeme S. Halford, Steven Phillips, William H. Wilson, J.E. McCredden, Glenda Andrews, Rosemary Baker, John D. Bain, (in press) The Central Executive: Proposals about its functions and capacity limitations. In Naoyuki Osaka and Robert Logie (eds) *Working memory: Behavioural and neural correlates*. Oxford University Press.

Bill Wilson's contact info

Last updated:
UNSW's CRICOS Provider No. is 00098G

Bill Wilson's Neural Networks Projects

Bill Wilson
Department of Artificial Intelligence
School of Computer Science and Engineering
University of New South Wales

E-mail: billw at cse.unsw.edu.au

This page is still under construction. Aren't they all?

Recurrent Network Architectures

This project began as an attempt to teach to neural networks the graphotactic patterns of English words: e.g. that a word can end but not begin with "nd". It developed into a comparison of different recurrent network archictectures for studying problems of this type, where there the possible value of the next item (to be predicted by the network) is constrained by the values of the last few inputs (letters) but not by inputs a long time in the past. For details, see recurrent network publications.

Tensor Product Networks

Human Analogical Problem Solving

Applications of formal systems and neural networks in cognitive modelling. This project, in collaboration with Graeme Halford, Department of Psychology, University of Queensland, concerns analogical reasoning and memory processes. A current direction in this work is to use tensor product networks to model capacity and analogical reasoning, and relational processing in humans. For details, see tensor product network publications.

For simple code to solve a proportional analogy problem, see C code for original STAR program. STAR stood for Structured Tensor Analogical Reasoning, as I recall.

Other/Past Projects on Neural Networks

These included work with Luyuan Fang and Tao Li on optimization neural networks, with Agus Rahardja and Arcot Sowmya on face recognition using feedforward networks, and on handwritten character recognition using a conventional front-end to preprocess the image, then a NN recognizer. For details, see optimization / face recognition / character recognition publications.

Neural Networks Research in the UNSW AI and IE Department

Neural networks group home page

Maintained by Bill Wilson,
School of Computer Science and Engineering,
University of NSW, Sydney, 2052 Australia

```
Email:  billw at cse.unsw.edu.au
Phone:  +61 2 9385 6876
Fax:    +61 2 9385 4071
```

```
Last modified: 13 January 1997
```

Bill Wilson's Natural Language Processing Research Page

Bill Wilson
School of Computer Science and Engineering
University of New South Wales

E-mail: billw at cse.unsw.edu.au

Well, actually lately I've been side-tracked working on recurrent neural network architectures. This work originally arose from a problem in computational linguistics.

This problem was an attempt to capture the graphotactic patterns of English words: e.g. that a word can end but not begin with "nd". It developed into a comparison of different recurrent network archictectures for studying problems of this type. The graphotactic project in turn developed out of a project that used letter trigrams to try to distinguish, among novel words in text, the ones that were typographical or similar errors, and those that were real words not covered by one's lexicon. (Obviously, some typographical errors result in words with graphotactically legal structure (like the mis-spelling of "architecture" that I just noticed, a couple of sentences back), and some real unknown words would be borrowings from languages with different graphotactic structures, but the idea was to get at least some idea of whether the novel word was OK.

Two of my students, however, are carrying on the NLU research here at UNSW:

- Graham Mann, see particularly BEELINE, his language-based direction-following system.
- Kyongho Min: working on correcting single and multiple errors in NL sentences.
  Here are some publications with Min.

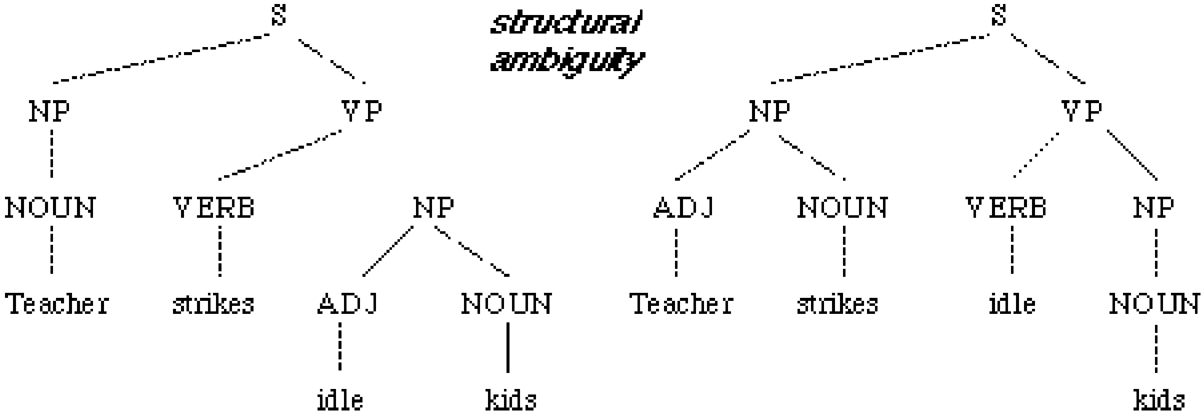
Here's a collection of ill-formed sentences (as used by Min, but collected by me).

Two other people in the School of Computer Science and Engineering do NLP-relevant work:

- Andrew Taylor: "automatic recognition of animal vocalisations" - a kind of speech recognition, I guess; and "using NLP to extract knowledge from biological definitions."
- John Shepherd: retrieving relevant text-based material from collections of textfiles - or in his words: "information filtering: adaptive agents for filtering Internet news";

Microsoft Research Institute has a NLP Resources page on NLP people/places/... in Australia.

Just for fun (supposedly a real headline somewhere, somewhen) ...



Bill Wilson's contact info

Last updated:

Past and Present Research Students with Bill Wilson

Past

Luyuan Fang
PhD 1990, Optimization neural networks

Agus Rahardja
MEngSc 1992, Face recognition via neural networks

Kyongho Min
MEngSc 1992, Parsing ill-formed sentences

Guy Smith
MSc 1993, Dynamic neural nets

Ashley Aitken
PhD 1997, Physiological neuron simulator
(co-supervised with Keith Tait)

Graham Mann
PhD 1996, Control of a rational agent by natural language

Kyongho Min
PhD 1997, Error recovery with a bidirectional chart parser

Brett Gray
PhD 2003, Automatic concept formation
(Dept of Psychology, Uni. of Queensland, jointly supervised with Graeme Halford)

Barry Drake
PhD 2004; Intelligence, Redundancy and Space.
Barry is currently (2005) working at Canon Information Systems Research Australia

Dates may be approximate

Current

Zhalaing Cheung (Zhalaing began at the start of 2002, on automated help desks)

Claire D'Este jointly supervised with Claude Sammut. (Claire began July 2002) - Teaching robots to communicate.

Matt Kameron

Spyridon Revithis

Home Page | Email: billw at cse.unsw.edu.au
Last updated:
CRICOS Provider No. 00098G

Bill Wilson's Projects in Cognitive Modelling

These projects are joint in various combinations with:

- *Graeme Halford*, *Cognition and Human Reasoning Laboratory Department of Psychology, University of Queensland*
- *Steven Phillips*, *Neuroscience Research Institute, AIST*, *Tsukuba, Japan.*
- Nadine Marcus, School of Computer Science and Engineering, Uni of NSW, and
- Brett Gray, School of Psychology, Uni of Queensland: brett at psy.uq.edu.au
- John Sweller, School of Education, University of New South Wales: j.sweller at unsw.edu.au

Graeme, Brett, Bill, and Steve on research retreat.

Recent grants

- 1994-6: Australian Research Council Large Grants Scheme: $190702 with Graeme Halford (Dept of Psychology, Uni. of Qld): "A Parallel Distributed Processing Model of Human Analogical Reasoning". Grant number: AC9332704

- 1996-8: Australian Research Council Large Grants Scheme: $150000 with Graeme Halford (Dept of Psychology, Uni. of Qld): "The acquisition and processing of relational knowledge". Grant number A79600056

- 2000-2: Australian Research Council Large Grants Scheme $56500 with Graeme Halford, Brett Gray (Dept of Psychology, Uni. of Qld), and Steven Phillips (Electrotechnical Laboratory, Japan):"Relational knowledge in categories and similarity: a neural net model." $55000 in 2001, $55000 in 2002. Grant number: A10027091

- 2001-3: Australian Research Council Large Grants Scheme SPIRT (Strategic Partnership with Industry for Research and Training) scheme: $22,292 p.a. to fund a stipend for a Ph.D. student to work on a project titled "Highly-tailored document retrieval systems for help desks". Grant number: C00107560

Publications on cognitive modelling

Modelling Human Problem Solving Using Evolutionary Computation Techniques

This project, with John Sweller and Nadine Marcus, aims to model John's theory about the development of human problem solving solving skills in a particular area (e.g. solving routine algebra problems or routine physics problems) from a searched-based method at the start to a schemata-based method as expertise grows. The model will use evolutionary computation for the search-based version, and we will investigate the development of the schemata in this context.

Models of relational processing

With Nadine Marcus, I am investigating the problem of using feedforward nets and other neural network models to simulate human relational processing, and in particular the multi-dimensional access property of human relational (semantic) memory.

By multi-dimensional access, we refer to the ability of humans who know a fact like "Jane likes pizza" to answer a range of questions, including "Does Jane like pizza?", "What does Jane like?", "Who likes pizza?" and even "What is the relationship between Jane and pizza?" (though I wouldn't try that one on a five-year-old!) and beyond - "Who likes what?", etc.

A PDP model of human analogical reasoning

The aim of this study is to further develop and test a Parallel Distributed Processing (PDP) model of analogical reasoning, called the Structured Tensor Analogical Reasoning (STAR) model (Halford, Wilson, Guo, Gayler, Wiles & Stewart, 1994). The 1994 STAR model can simulate solution of analogical reasoning problems, but the proposed project will extend the model in the following ways:

- Test whether it is a valid model of human analogical reasoning.
- Incorporate distributed representations into the serial processing aspects of the model.
- Incorporate learning functions into the model
- Model the development of representations.

The acquisition and processing of relational knowledge

Relational knowledge lies at the core of most higher cognitive processes. It is the basis of mathematical thinking. Number operations, addition and multiplication, may be thought of as ternary relations, since each consists a set of ordered 3-tuples such as +(2,3,5), +(3,4,7), *(2,3,6), *(3,4,12) etc. Relational knowledge is also important in science generally, and the conceptual sophistication which is essential to expertise depends on relational knowledge (e.g. the relationship between force, mass, and acceleration). Relations are the basis of all structured knowledge, since a structure consists of a set of elements on which one or more relations is defined (i.e. a structure is an ordered pair (S,R), where S is a set of elements and R is a set of relations on subsets of S).

Within the psychological literature, the concepts of transitivity, class inclusion, and configurational concepts (oddity, conditional discrimination, transverse patterning, and negative patterning) are all relational concepts. Analogical reasoning, which has been shown to be of fundamental importance to higher cognition in humans is a map from base to target, both of which are coded in terms of relations.

The concept of relational knowledge is also relevant to certain distinctions of historical and contemporary importance. For example, Gestalt Psychology emphasised relations and structure, in opposition to associationism, which did not use relations as a fundamental explanatory construct. As we will see, symbolic processing generally can be defined as processing of relations, in contrast to imaginal codes, implicit processing, or subsymbolic processing. Norman's (1986) observation that "People do seem to have at least two modes of operation, one rapid, efficient, subconscious, the other slow, serial, and conscious." can be accounted for largely by the distinction between associative and relational processing.

However, as Smith (1989) has noted, "... *despite all the empirical work, and the importance of relational concepts, there is no unified framework for thinking about their structure and about how they develop*".

Though these distinctions have been of considerable importance, the precise properties of each mode of processing have remained somewhat obscure.

Our work has indicated that these modes can be defined mathematically, and their essential properties can be captured in neural net models. The first step to doing this is to provide a precise definition of relational processing, then to derive its properties, and contrast them with alternative modes of processing.

The aims of this project are:

- to define a mathematical basis for relational processing, as distinct from associative processing, and to determine to what extent these two modes can account for traditional and contemporary distinctions including association/gestalt, subsymbolic/symbolic;

- to implement connectionist models of these processes, using tensor product networks, recurrent networks, and hybrids of these for the relational models;

- to derive their properties, theoretically and by testing the implementations; *and*
- to test empirically the predictions from the model.

  The empirical work focuses on induction of relational schemas, assessment of relational concepts, and with the effect of factors such as age, processing capacity, ageing and frontal lobe damage.

Maintained by Bill Wilson,
School of Computer Science and Engineering,
University of NSW, Sydney, 2052 Australia
Email: billw at cse.unsw.edu.au
Phone: +61 2 9385 6876
Fax: +61 2 9385 1812

Last updated:

Bill Wilson's Combinatorial Search Algorithms Research

Bill Wilson
Department of Artificial Intelligence
School of Computer Science and Engineering
University of New South Wales

E-mail: billw at cse.unsw.edu.au

I do this sort of thing jointly with Debbie Street.

Actually, if you want to know more about it, you'd do better to talk to her.
However, here are my publications on combinatorial searches.

Bill Wilson's contact info

Last modified: 02 July 1996

The Prolog Dictionary

Contact Info Last updated:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

This version of the Prolog dictionary assumes the syntax of SWI Prolog.

You should use The Prolog Dictionary to clarify or revise concepts that you have already met. The Prolog Dictionary is *not* a suitable way to *begin* to learn about Prolog. Further information on Prolog can be found in the SWI Prolog documentation linked above. More details on Prolog can be found in the class web page lecture notes section.

Other related dictionaries:
The AI Dictionary - URL: `http://www.cse.unsw.edu.au/~billw/aidict.html`
The Machine Learning Dictionary - URL: `http://www.cse.unsw.edu.au/~billw/mldict.html`
The NLP Dictionary (Natural Language Processing) - URL: `http://www.cse.unsw.edu.au/~billw/nlpdict.html`

Other places to find out about artificial intelligence include the AAAI (American Association for Artificial Intelligence) AI Overview page or AI Reference Shelf

The URL of this Prolog Dictionary is `http://www.cse.unsw.edu.au/~billw/prologdict.html`

Topics Covered

This dictionary is limited to the Prolog concepts covered in COMP9414 Artificial Intelligence at the University of New South Wales, Sydney.

arity | atom | backtracking | binding | body | built-in predicates | clause | comments | cut | debugging | declarative | files | functor | goal | head | indentation | infix vs prefix | `is` | list | mutual recursion | neck | number | output in Prolog | predicate | procedure | query | reading | recursion | relation | relations and functions | rule | see | seeing | seen | structure | succeed | tell | telling | told | term | testing | tracing execution | underscore | white space | writing | variable

A

arity

The arity of a functor is the number of arguments it takes. For example, the arity of `likes`, as in `likes(jane, pizza)`, is 2, as it takes two arguments, `jane` and `pizza`.

Arity	*Example*	*Could Mean ...*
1	`happy(fido).`	Fido is happy.
2	`likes(jane, pizza).`	
3	`gave(mary, john, book).`	Mary gave John a book.

atom

An atom, in Prolog, means a single data item. It may be of one of three types:

- a string atom, like 'This is a string' or
- a symbol, like `likes`, `john`, and `pizza`, in `likes(john, pizza).` Atoms of this type must start with a lower case letter. They can include digits (after the initial lower-case letter) and the underscore character (_).
- strings of special characters, like `<--->`, `...`, `===>`. When using atoms of this type, some care is needed to avoid using strings of special characters with a predefined meaning, like the neck symbol `:-`

  The available special characters for constructing this class of atom are: `+`, `-`, `*`, `/`, `<`, `>`, `=`, `:`, `.`, `&`, `_`, and `~`.

Numbers, in Prolog, are not considered to be atoms.

B

backtracking

Backtracking is basically a form of searching. In the context of Prolog, suppose that the Prolog interpreter is trying to satisfy a sequence of goals *goal_1*, *goal_2*. When the Prolog interpreter finds a set of variable bindings which allow *goal_1* to be satisfied, it commits itself to those bindings, and then seeks to satisfy *goal_2*. Eventually one of two things happens: (a) *goal_2* is satisfied and finished with; or (b) *goal_2* cannot be satisfied. In either case, Prolog backtracks. That is, it "un-commits" itself to the variable bindings it made in satisfying *goal_1* and goes looking for a *different* set of variable bindings that allow *goal_1* to be satisfied. If it finds a second set of such bindings, it commits to them, and proceeds to try to satisfy *goal_2* again, with the new bindings. In case (a), the Prolog interpreter is looking for *extra* solutions, while in case (b) it is still looking for the first solution. So backtracking may serve to find extra solutions to a problem, or to continue the search for a first solution, when a first set of assumptions (i.e. variable bindings) turns out not to lead to a solution.

Example: here is the definition of the standard Prolog predicate "member":

```
member(X, [X | Rest]). % X is a member if its the first element
member(X, [Y | Rest]) :-
    member(X, Rest).   % otherwise, check if X is in the Rest
```

You may not think of `member` as a backtracking predicate, but backtracking is built into Prolog, so in suitable circumstances, `member` will backtrack:

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

Here `member` backtracks to find every possible solution to the query given to it. Consider also:

```
?- member(X, [a, a, a]).
X = a ;
X = a ;
X = a ;
No
```

Here `member` backtracks even though it keeps on finding the same answer. What about

```
?- member(a, [a, a, a]).
```

In one view, Prolog should respond:

```
Yes ;
Yes ;
Yes ;
No
```

However, this useless behaviour is inhibited, and a single `Yes` is printed.

The term backtracking also applies to seeking several sets of variable bindings to satisfy a single goal.

In some circumstances, it may be desirable to inhibit backtracking, as when only a single solution is required. The Prolog cut goal allows this.

binding

Binding is a word used to describe giving a value to a variable. It normally occurs during application of a Prolog rule, or an attempt to satisfy the goals in a Prolog query.

For example, when attempting to respond to the query:

```
?- lectures(john, Subject), studies(Student, Subject).
```

the Prolog interpreter first finds a fact that indicates a subject that John lectures - perhaps `lectures(john, 9311)`. At this point, the variable `Subject` is bound to `9311` (`Subject = 9311`). In other words, `Subject`, temporarily, has the value `9311`. Then Prolog tries to satisfy the second goal `studies(Student, Subject)` with `Subject = 9311`, i.e. to satisfy `studies(Student, 9311)`. In doing so, if it does find a solution, it will bind `Subject` to one or more values (like `jack`) in turn. Then Prolog will backtrack and undo the binding `Subject = 9311` and look for another value for `Subject` that satisfies `lectures(john, Subject)`, such as `Subject = 9331`, and then proceed for appropriate bindings for `Student` again, with the new binding for `Subject`.

body

the last part of a Prolog rule. It is separated from the head by the neck symbol `:-`. It has the form of a comma-separated list of goals, each of which is a functor, possibly followed by a comma-separated list of arguments, in parentheses. E.g. in the rule

```
sister_of(X,Y) :- female(Y), same_parents(X,Y).
```

the two goals `female(Y), same_parents(X,Y)`form the body.

built-in procedure or predicate

To make Prolog programming more practical, a number of predicates or procedures are built in to Prolog. These include some utility procedures, which *could* have been programmed by the Prolog user, and some which do non-logic programming things, like the input and output routines, debugging routines, and a range of others.

C

clause

A clause in Prolog is a unit of information in a Prolog program ending with a full stop ("`.`"). A clause may be a fact, like:

```
likes(mary, pizza).
```

or a rule, like:

```
eats(Person, Thing) :- likes(Person, Thing), food(Thing).
```

A group of clauses about the same relationis termed a procedure.

comment

in Prolog, a the start of a comment is signalled by the character `%`. The rest of the line after and including the % is ignored by the Prolog interpreter. Examples:

```
% This is a full line comment.
% The next line contains a part-line comment.
member(Item, [Item|Rest]). % member succeeds if Item is 1st object in list.
```

Commenting Rules: As in other programming languages, comments should be used freely to explain the high-level significance of sections of code, to explain tricky sections of code, etc. Comments should *not* echo what the code already clearly says. Comments like that actually get in the way of understanding, for example because they are likely to make the reader ignore the comments. Where it is possible to make code understandable by using a meaningful functor name or variable name, this is preferable to a comment.

It is good practice to begin each Prolog procedure with a comment describing the predicate, e.g.

```
% member(Item, List) - succeeds if the item is a member of the list
```

If the list needs to have some special properties, e.g. if it must be a list of numbers, or must be instantiated (that is, have a value) at the time the predicate is called, then this *header* comment should say so:

```
% member(Item, List) - succeeds if the item is a member of the list;
%   List must be instantiated at the time member is called. Item need not
%   be instantiated.
```

It can be a good idea for the header comments to indicate examples of the kind of data on which the predicate is intended to operate, and what the result should be, if this can be done reasonably briefly. E.g.

```
% Examples of use:
% ?- member(b, [a, b, c]).
% No
%
% ?- member(X, [a, b, c]).
% X = a ;
% X = b ;
% X = c ;
% No
```

Each file of Prolog code should begin with a (file) header comment, indicating who wrote the code, when, and what it is (overall) intended to do. This would be enough in a short Prolog assignment. In a "industrial strength" Prolog system, there would be details on the origins of algorithms used, and a revision history.

A good source of examples of Prolog commenting is example code made available in class, such as this one. Note however that this one is rather more heavily commented than usual, for instructional purposes. Note also that code examples presented on screen may be *under*-commented, because of the difficulty of fitting the comments and the code on the screen, and because the oral presentation accompanying the on-screen material replaces the comments to some extent.

Don't make your lines of comments (or code) too long - long lines can be hard to read, and really long lines may be folded, turning your neat formatting into a dog's breakfast. Stick to a maximum line length of 70 or 80 characters. Cute lines and/or boxes constructed of comment characters have the side effect of preventing the reader from seeing as much of your code at one time. The reader may then have to page up and down to figure out what your code does. They will not like you for this.

See also indentation and white space.

cut, !

The cut, in Prolog, is a goal, written as `!`, which always succeeds, but cannot be backtracked past. It is used to prevent unwanted backtracking, for example, to prevent extra solutions being found by Prolog.
The cut should be used sparingly. There is a temptation to insert cuts experimentally into code that is not working correctly. If you do this, bear in mind that when debugging is complete, you should understand the effect of, and be able to explain the need for, every cut you use. The use of a cut should thus be commented.

D

debugging

means removing errors from program code.

declarative

A declarative programming language is one in which the relationships between the data are stated (or *declared* in a (usually logic-based) language, and then some automatic mechanism, e.g. a theorem-prover, is used to answer queries about the data. Prolog is a declarative programming language. Haskell, Miranda, Lisp, and Scheme are *functional* programming languages, in which all constructs are expressed using functions, function arguments, and function results, and C, C++, Java, Perl, Pascal, Modula-2, and Fortran are examples of (high-level) *procedural* programming languages, in which the program code expresses procedures to follow in manipulating the data.

E

F

files

When one writes a Prolog program, usually the facts and rules of the program are stored in a (text) file, and then loaded into the Prolog interpreter. Files have other uses in Prolog, too.

For example, we may wish to write out a table of results that have computed for us by our Prolog program. One can use built-in predicates like `write, nl, putc, tab` and others to write out the table, but by default it will appear on the computer screen. To direct this output to a file, we use the `tell` built-in predicate. Suppose that we wish to write the table to a file called "mytable.data". By executing the (pseudo-)goal `tell("mytable.data")`, we tell Prolog that the new *current output stream* is to be the file "mytable.data". Subsequent writes will go to this file. When one wishes to stop writing to the file and resume writing on the screen, one uses the built-in predicate `told` (with no arguments). Also, the query `?- telling(X).` binds `X` to the name of the current output file. If the current output stream is not a file, then `X` will be bound to something that indicates that the current output stream is the screen - for example, in Unix, `X` may be bound to the atom `stdout` (standard output, which is normally the screen). Example:

```
?- tell('mytable.data'), write('***** Table of results *****'), nl, told.
% the file mytable.data should now contain a single line of text as above
```

The situation for reading from a file is analogous. One can use built-in predicates like `read, getc` and others to read, by default from the keyboard. By executing the (pseudo-)goal `see('mydata.text')`, we tell Prolog that the new *current input stream* is to be the file `mydata.text`. Subsequent reads will come from this file. When one wishes to stop reading from the file and resume reading from the keyboard, one uses the built-in predicate `seen` (with no arguments). Also, the query `?- seeing(X).` binds `X` to the name of the current input file. If the current input stream is not a file, then `X` will be bound to something that indicates that the current output stream is the screen - for example, in Unix, `X` may be bound to the atom `stdin` (standard input, which is normally the keyboard). Example:

```
?- see('mydata.text'), read(X), seen, write(X), nl.
% the first Prolog term in the file mydata.text should now appear
% on the screen, having been read from the file with read(X), and then
% written to the screen with write(X) and nl.
```

functor

In Prolog, the word functor is used to refer to the atom at the start of a structure. For example, in `likes(mary, pizza)`, `likes` is the functor. In a more complex structure, like

```
persondata(name(smith, john), date(28, feb, 1963))
```

the top-level functor is termed the *principal functor* - in this case `persondata` - There is also a built-in predicate called `functor`, used to extract the functor and arity of a structure.

G

goal

A query to the Prolog interpreter consists of one or more goals. For example, in

```
?- lectures(john, Subject), studies(Student, Subject).
```

there are two goals, `lectures(john, Subject)` and `studies(Student, Subject)`. A goal is something that Prolog tries to *satisfy* by finding values of the variables (in this case `Student` and `Subject`) that make the goal succeed.

H

head

the first part of a Prolog rule. It is separated from the body by the neck symbol `:-`. It normally has the form of a functor (i.e. a relation symbol, followed by a comma-separated list of parameters, in parentheses. E.g. in the rule

```
sister_of(X,Y) :- female(Y), same_parents(X,Y).
```

`sister_of(X,Y)` is the head.

I

indentation

Indenting your Prolog code in a standard way is a technique, along with commenting it, to make your code more easily understood (by humans). The standard way to lay out a Prolog procedure is exemplified below, using a procedure to compute the length of a list. Comments have been omitted, to allow you to focus just on the indentation.

```
listlength([], 0).

listlength([Head | Tail], Length) :-
    listlength(Tail, TailLength),
    Length is TailLength + 1.
```

Sometimes the indentation rules can be bent: for example, it is not unusual to put on a single line, a rule with a body that contains just a single (short) clause.

```
bad(Dog) :- bites(Dog, _).
```

See also comments and white space.

infix and prefix predicates

Most built-in predicates in Prolog, and, by default, the ones you write yourself, are *prefix* predicates - that is, the name of the predicate precedes the arguments (which are surrounded by parentheses and separated by commas). For example, when we use the built-in predicate `member`, we write something like `member(Item, [a, b, c])` - first the predicate name, `member`, then "(", then the first argument, `Item`, then a comma, then the second argument, `[a, b, c]`, then ")".

However, with predicates like `=`, `<` and `>` that are usually written between their arguments, as in `First < Max`, we write, in Prolog as in mathematics, in *infix* notation. Another infix built-in predicate is `is`, which is used in evaluating arithmetic expressions.

It is possible to define your own infix predicates - this is beyond the scope of this discussion, but you can look up the Prolog operator `op` in your

Prolog manual or textbook.

is

The `is` predefined predicate is used in Prolog to force the evaluation of arithmetic expressions. If you just write something like `X = 2 + 4`, the result is to bind `X` to the unevaluated term `2 + 4`, not to `6`. Try it:

```
?- X = 2 + 4.

X = 2+4
```

If instead you write `X is 2 + 4`, Prolog arranges for the second argument, the arithmetic expression `2 + 4`, to be evaluated (giving the result `6`) before binding the result to X.

```
?- X is 2 + 4.

X = 6
```

It is only and always the second argument that is evaluated. This can lead to some strange-looking bits of code, by mathematical standards. For example, `mod` is the remainder-after-division operator, so in Prolog, to test whether a number `N` is even, we write `0 is N mod 2`, rather than the usual mathematical ordering: N mod 2 = 0.

What is the difference between `N = 1` and `N is 1`? In final effect, nothing. However, with `N is 1`, Prolog is being asked to do an extra step to work out the value of 1 (which, not surprisingly, is 1). Arguably, it is better to use `N = 1`, since this does not call for an unnecessary evaluation.

The message is: *use `is` when, and only when, you need to evaluate an arithmetic expression.*

A common mistake, for people used to procedural programming languages like C, is to try to change the binding of a variable by using an goal like `N is N + 1`. This goal will never succeed, as it requires `N` to have the same value as `N + 1`, which is impossible. If you find yourself wanting to do something like this, you could look at the code of `factorial` for inspiration.

J

K

L

lists

A list in Prolog is written as a comma-separated sequence of items, between square brackets. For example, `[1, 2, 3]` is a list. The empty list is written `[]`.

Frequently it is convenient to refer to a list by giving the first item, and a list consisting of the rest of the items. In this case, one writes the list as `[First | Rest]`.

We have expressed this here using variables, but this need not be so, for example, we could write `[1, 2, 3]` as:

- `[1 | [2, 3]]`
- `[1 | Rest]`, where `Rest` is bound to `[2, 3]`
- `[First | [2, 3]]`, where `First` is bound to `1`
- `[First | Rest]`, where `First` is bound to `1`, and `Rest` is bound to `[2, 3]`
- `[1, 2 | [3]]`
- `[1, 2, 3 | []]`

and many more possibilities.

M

mutual recursion

Sometimes a predicate does not explicitly refer to itself (this would be (simple) recursion), but rather refers to a second predicate which in turn refers to the first. This sets up a two-step *mutual recursion*. Three- or more- step mutual recursion situations can also occur. The usual conditions applicable to recursion must occur - there must be a "trivial branch" somewhere in the cycle of mutually recursive predicates, and somewhere in the cycle something must happen to ensure that each time around the cycle, a simpler version of the problem is being solved.

N

neck

the symbol `:-`, used in a Prolog rule to separate the head from the body. Usually read as *if*. Thus

```
a :- b, c.
```

is read as

`a` (is true) *if* `b` *and* `c` (are true).

number

Numbers in Prolog can be whole numbers, like:

1 1313 0 -97 9311

or fractional numbers, like:

3.14 -0.0035 100.2

The range of numbers that can be used is likely to be dependent on the number of bits (amount of computer memory) used to represent the number. The treatment of real numbers in Prolog is likely to vary from implementation to implementation - real numbers are not all that heavily used in Prolog programs, as the emphasis in Prolog is on symbol manipulation.

O

output in Prolog

Output in Prolog is not always needed, as often the variable bindings return all the information that is required. If explicit output is required, a set of extra-logical built-in predicates is available.
These include:

- `write(X)` which writes the term X to the current output stream (which means the window on your workstation unless you have done something fancy).
- `print(X, ...)` which writes a variable number of arguments to the current output stream. If an argument is a string (like `'Hello world!\n'`) then the string is printed without quotes, and any `'\n'` and `'\t'` are interpreted as newline and tab, respectively. A newline is printed at the end of the printing operation.
- `prin(X, ...)` is like `print` except that it does not append a newline to the end of the output.
- `nl` starts a new line.

P

predicate

In this context, used interchangeably with the term procedure.

procedure

A procedure in Prolog is a group of clauses about the same relation, for example:

```
is_a_parent(X) :- father_of(X, _).
is_a_parent(X) :- mother_of(X, _).
```

Here, two clauses together define the condition for someone to be a parent - together they form a procedure.

Q

query

A query is a list of one or more goals typed to the Prolog interpreter's `?-` prompt, separated by commas, and terminated with a full stop (`.`). For example,
`?- lectures(john, Subject), studies(Student, Subject).`
is a query comprising two goals. Prolog tries to satisfy the goals, and if it manages to do this, the query is said to succeed. If not, the query fails.

If the query fails, Prolog types "No". If it succeeds, Prolog either types the list of variable bindings it had to assume in order to make the query succeed, or, if no variable bindings were necessary, it types "Yes".

If several variable bindings allow the query to succeed, then normally (i.e. in the absence of cuts) all the bindings will be typed out, one after the other, with the Prolog user typing a `;` to let the Prolog system know that they are ready to see the next set of bindings.

R

reading

Article on reading coming real soon now.

recursion

A recursive rule is one which refers to itself. That is, its body includes a goal which is an instance of the relation that appears in the head of the rule. A well-known example is the `member` procedure:

```
member(Item, [Item|Rest]).
member(Item, [_|Rest]) :- member(Item, Rest).
```

The second clauseis clearly recursive - `member`occurs both in the head and the body. A recursive procedure can work because of two things:

1. the instance of the relation (like `member`) in the body is in some way simpler than that in the head, so that Prolog is being asked to solve a simpler problem. (In `member`, the list that is the argument to `member` in the body is shorter by one than that in the head.)
2. there must be a "trivial branch" "base case" or "boundary case" to the recursion - that is, a clause that does *not* involve recursion. (In the case of `member`, the first clause is the trivial branch - it says that `member` succeeds if `Item` is the first member of the list that is the second argument.)

A recursive data structure is one where the structures include substructures whose principle functoris the same as that of the whole structure. For example, the tree structure:

```
tree(tree(empty, fred, empty), john, tree(empty, mary, empty))
```

is recursive. Usually, recursive data structures require recursive procedures.

See also mutual recursion.

relation

The word *relation*, in Prolog, has its usual mathematical meaning. See relations and functions if you are not sure about what this is. A *unary* relation is a set of objects all sharing some property. Example:

```
is_dog(fido). is_dog(rex). is_dog(rover).
```

A *binary*relation is a set of pairs all of which are related in the same way. Example:

```
eats(fido, biscuits). eats(rex, chocolate).
eats(rover, cheese).  eats(lassie, bone).
```

Similarly for *ternary*relations (triples) and so on. In the examples above, `is_dog`and `eats`are the functorsfor the relations. Prolog stores information in the form of relations, or more specifically relational instances.

See also the the next entry for more detail.

relations and functions in mathematics and in Prolog

A *binary* relation, in mathematics, is a set of ordered pairs. For example, if we are thinking about the 3 animals: *mouse*, *rabbit*, and *sheep*, and the "smaller" relation, then the set of pairs would be {(*mouse*, *rabbit*), (*mouse*, *sheep*), (*rabbit*, *sheep*)}. In mathematical notation, you would write, e.g., (*mouse*, *rabbit*) ∈ *smaller*, or *mouse smaller rabbit*. Or you might use a symbol instead of *smaller*, perhaps "<" - *mouse* < *rabbit*, or σ - *mouse* σ *rabbit*. This is referred to as an *infix* notation, because the name of the relation is written in-between the two objects (*mouse* and *rabbit*).

In Prolog, instead, by default we use prefix notation: `smaller(mouse, rabbit)`.

This view of a relation is sometimes called the *extensional* view - you can lay out the full "extent" of the relation, at least in the case of a relation over a finite number of items, like our "smaller" example. The alternative view is called *intensional*, where we focus on the meaning of the relation. In Prolog, this corresponds to a rule expressing the relation, such as:

```
smaller(X, Y) :-
        volume(X, XVolume),
        volume(Y, YVolume),
        XVolume < YVolume.
```

This, of course, only defines a meaning or intension for `smaller` relative to the unspecified meaning of the relation `volume(_,_)`, and the meaning of `<`, which is defined by the Prolog implementation.

When we come to non-binary relations, such as unary ones (like `is_a_dog(fido)`) or ternary ones (like `gives(john, mary, book)` or `between(rock, john, hard_place)`) the infix relation doesn't make any sense, so prefix notation is normal. (Postfix notation - `fido is_a_dog` can work for unary relations.)

While we often don't think of it this way, a *function* is a kind of relation. If you are thinking about the function $f(x) = x^2$, then the essential thing is that for each value of *x*, there is a value of $f(x)$ (namely x^2). In fact, there is a *unique* value of $f(x)$. So a function (of a single variable) can be viewed as a binary relation such that for every relevant first component *x*, there is exactly one pair in the relation that has that first component[1]: the pair is $(x, f(x))$. In the case of $f(x) = x^2$, the pairs are (x, x^2) for every applicable value of *x*. We need to specify what the applicable values of *x* are - the *domain* of the function. If the domain is {1, 2, 3}, then the pairs are {(1,1), (2,4), (3,9)}. If the domain is all natural numbers, then we can't write out the extension of the function in full, but we can use a set expression such as $\{(n, n\times n) \mid n \in N\}$ where *N* signifies the set of all natural numbers.

In Prolog, despite the fact that it uses a relational notation (except for arithmetic expressions), functions are common, but expressed using the relational notation that depends on the definition/convention in the previous paragraph. In our Prolog code defining `smaller(X, Y)`, above, the relation called `volume` is in fact a function written relationally. We are saying that for every relevant object `X` there is a value `XVolume` that is the volume of `X`. This is a function-type relationship.

The notion of domain applies to relations as well as functions: a more detailed definition of a binary relation *on a set A* says that such a relation is a subset of the set *A*×*A* of all pairs of elements of *A*. *A* is the domain of the relation. A binary relation between sets *A* and *B* is a subset of *A*×*B*. And so on for ternary relations and beyond. A unary relation on *A* is just a subset of *A*. Thus *is_a_dog* is a subset of the set of all *Animals*. Or a subset of the set of all *Things*, depending on just how broadly you want to consider the concept of being a dog.

However, in Prolog, domains of relations are not explicitly addressed.[2] The notion of domain corresponds exactly to that of *type* in typed programming languages like, C, C++, Modula-2, Pascal, etc. Prolog has no built-in way of confining the definition of a relation (whether extensionally or by a rule) to a particular domain. If you want to, you can build type-checking into your rules, by giving an definition of a type as a unary relation. For example, if you wanted to define the type of all popes, you could enumerate them all, though the list would be long:

```
pope(peter).
...
pope(john_paul_ii).
pope(benedict_xvi).
```

Then the goal `pope(X)` would check if `X` was a pope.

Another example: when defining the relation `sister` in Prolog, you would usually write something like:

```
sister(X, Y) :-
        female(X), female(Y),
        mother(X, Mother), mother(Y, Mother),
        father(X, Father), father(Y, Father),
        not (X = Y).
```

`female(X)` and `female(Y)` are type-checks - without them, you are defining `sibling`, not `sister`. In practice, you can't enumerate all females, unlike all popes, but in many cases you would be able to enumerate all the relevant ones.

Footnotes:

[1]: if there is, not *exactly*, but *at most* one pair in the relation that has *x* as its first member, we say that the relation is a *partial function*. A partial function is like a function that has "holes" in it - *f*(*x*) never has more than one value, but for some *x* in the domain of the function, *f*(*x*) might not have any value. A real-life example is the partial function *eldest_child_of*, defined on the domain of all adult humans.

[2]: there are a couple of areas of Prolog where there is some built-in notion of type - for example, the built-in predicate `<` will generate an error message if you try to use it compare a string like `'mouse'` with a number like `9`.

rule

A rule in Prolog is a clause with one or more variables in it. Frequently, rules have a head, neck and body, as in:

```
eats(Person, Thing) :- likes(Person, Thing), food(Thing).
```

Sometimes, however, the body can be omitted:

```
member(Item, [Item|Rest]).
```

This says that the Item is a member of any list (the second argument) of which it is the *first* member. Here, no special condition needs to be satisfied to make the head true - it is always true.

S

structures

Structures in Prolog are simply objects that have several components, but are treated as a single object. Suppose that we wish to represent a date in Prolog - dates are usually expressed using a day, a month, and a year, but viewed as a single object. To combine the components into a single structure, we choose a functor, say `date`, and use it to group the components together - for the date usually expressed as 21 March 2004, we would probably write

```
date(21, mar, 2004)
```

Note that the order of the components is our choice - we might have instead written `date(2004, mar, 21)`The choice of functor is arbitrary, too.

Structures may be nested, too - the following example groups a name and a date, perhaps the person's date of birth:

```
persondata(name(smith, john), date(28, feb, 1963))
```

See also Bratko, section 2.1.3.

succeed

A Prolog goal succeeds if it is possible to either check it directly against the facts known to Prolog, or find bindings for variables in the goal that makes the goal true.

A query succeeds if each of its goals succeeds.

T

term

Syntactically, all data objects in Prolog are terms. For example, the atom `mar` and `date` and the structure `date(2004, mar, 21)` are terms. So are the numbers `2004` and `21`, for that matter.

testing

Here are some tips on testing your Prolog code.
And here are some tips on writing recursive procedures in Prolog that include some tips on debugging (in example 3).

tracing

Tracing the execution of a Prolog query allows you to see all of the goals that are executed as part of the query, in sequence, along with whether or not they succeed. Tracing also allows you to see what steps occur as Prolog backtracks.

To turn on tracing in Prolog, execute the "goal"

```
?- trace.

Yes
```

When you are finished with tracing, turn it off using the "goal"

```
?- notrace.
```

Here is an example of `trace` in action. First some code, which computes factorial N - the product of the numbers from 1 to N. By convention, factorial 0 is 1. Factorial N is often written N!, where the exclamation mark signifies the "factorial operator". Here is the code:

`factorial(0, 1).`	rule 1	factorial(0) = 1
`factorial(N, NFact) :-`	rule 2	
`    N > 0,`	rule 2, goal 1	if N > 0, then
`    Nminus1 is N - 1,`	rule 2, goal 2	compute N - 1,
`    factorial(Nminus1, Nminus1Fact),`	rule 2, goal 3	recursively work out factorial(N-1),
`    NFact is Nminus1Fact * N.`	rule 2, goal 4	then multiply that by N

The tables above and below may not display correctly if viewed in a narrow window. If the commentary doesn't seem to line up correctly with the text to the left of it, try making your browser window wider.

Prolog dialog/trace output	**Commentary**

```
% prolog -s factorial.pl                   Invoke prolog, loading code for factorial
blah blah blah ...                         Greeting from Prolog
?- trace.                                  Turn on tracing
Yes
[trace]  ?- factorial(3, X).               Call factorial
   Call: (7) factorial(3, _G284) ? creep[1] Trace echoes query, replacing X with a unique variable
^  Call: (8) 3>0 ? creep                   Rule 2, Goal 1 (N > 0) is invoked
^  Exit: (8) 3>0 ? creep                   Goal 1 succeeds immediately
^  Call: (8) _L205 is 3-1 ? creep          Rule 2, Goal 2 invoked to compute 3 - 1
^  Exit: (8) 2 is 3-1 ? creep              and succeeds
   Call: (8) factorial(2, _L206) ? creep   Rule 2, Goal 3 is invoked: level 2 call to factorial(2, ...)
^  Call: (9) 2>0 ? creep                   Goal 1 again for new call to factorial
^  Exit: (9) 2>0 ? creep                   Goal 1 succeeds
^  Call: (9) _L224 is 2-1 ? creep          New version of goal 2
^  Exit: (9) 1 is 2-1 ? creep              succeeds
   Call: (9) factorial(1, _L225) ? creep   Rule 2, Goal 3 is invoked: level 3 call to factorial(1, ...)
^  Call: (10) 1>0 ? creep                  Goal 1 again
^  Exit: (10) 1>0 ? creep                  succeeds
^  Call: (10) _L243 is 1-1 ? creep         Goal 2 again
^  Exit: (10) 0 is 1-1 ? creep             successfully
   Call: (10) factorial(0, _L244) ? creep  Rule 2, Goal 3 is invoked: recursive call to factorial(0, ...)
   Exit: (10) factorial(0, 1) ? creep      This time, Rule 1 succeeds at once.
^  Call: (10) _L225 is 1*1 ? creep         This is Goal 4 of level 3
^  Exit: (10) 1 is 1*1 ? creep             Compute 1 * 1 without trouble
   Exit: (9) factorial(1, 1) ? creep       so the level 3 factorial call succeeds
^  Call: (9) _L206 is 1*2 ? creep          This is goal 4 of level 2
^  Exit: (9) 2 is 1*2 ? creep              Compute 1 * 2 without trouble
   Exit: (8) factorial(2, 2) ? creep       so the level 2 factorial call succeeds
^  Call: (8) _G284 is 2*3 ? creep          This is goal 4 of level 1
^  Exit: (8) 6 is 2*3 ? creep              Compute 2 * 3 without trouble
   Exit: (7) factorial(3, 6) ? creep       so the level 1 factorial call succeeds

X = 6 ;                                    ... with this binding of X - type ";" to find more solutions
   Redo: (10) factorial(0, _L244) ? creep  Prolog backtracks looking for another solution
^  Call: (11) 0>0 ? creep
^  Fail: (11) 0>0 ? creep
   Fail: (9) factorial(1, _L225) ? creep
   Fail: (8) factorial(2, _L206) ? creep
   Fail: (7) factorial(3, _G284) ? creep
                                           without success
                                           Turn off tracing.
```

```
No
[debug]  ?- notrace. % turn off tracing

Yes
[debug]  ?-
```

[1] What is this "creep" business? In SWI Prolog, the implementation of Prolog which this dictionary uses for the syntax of its examples, when you press return at the end of a line of tracing, Prolog prints "creep" on the *same* line, and then prints the next line of trace output on the next line. Pressing return again produces "creep" again and another line of tracing, and so on.

There are further tracing facilities in SWI Prolog. Do

```
?- help(trace).
```

to start to find out about them.

U

underscore

The Prolog variable _ (underscore) is a "don't care" variable, which will match anything. For example, the rule

```
bad(Dog) :- bites(Dog, _).
```

says that something (`Dog`) is bad if `Dog` bites anything. The "anything" is represented by _. Unlike other variables in Prolog, a bare _ can match different things in the same rule. So, for example, if `gives(From, To, Gift)` is a three-place predicate that is true if `From` gives `Gift` to `To`, then

```
giver(X) :- gives(X, _, _).
```

signifies that someone (`X`) is a "giver" if `X` gives something to anybody - the two _ s don't have to match the same thing. So if `gives(fred, john, black_eye).` is stored in the Prolog database, then the rule above allows us to infer that `fred` is a giver.

Prolog systems make internal use of variable names beginning with an underscore, e.g. `_G157`. These are not "don't care" variables, but are chosen to not clash with any variable name a Prolog programmer is likely to use.

Underscores may also be used in the middle of variable or atom names, as in `Intermediate_solution`, `likes_person` - again, these are not "don't care" variables. The idea is to make the variable or atom name more readable.

V

variable

A variable in Prolog is a string of letters, digits, and underscores (_) beginning either with a capital letter or with an underscore. Examples:

`X`, `Sister`, `_`, `_thing`, `_y47`, `First_name`, `Z2`

The variable _ is used as a "don't care" variable, when we don't mind what value the variable has. For example:

```
is_a_parent(X) :- father_of(X, _).
is_a_parent(X) :- mother_of(X, _).
```

That is, X is a parent if they are a father or a mother, but we don't need to know who they are the father or mother of, to establish that they are a parent.

Variables are used in more than one way:

- to express constraints in parts of a rule:
  `likes(A,B) :- dog(B), feeds(A,B).`
  Note that both `A` and `B` appear on both sides of the neck symbol - the appearance of the same variable in two (or more places) in effect says that when the variable is bound to a value, it must be bound to the same value in all the places that it appears in a given rule.
  Note that if the same variable appears in two or more different rules, it might be bound to different values in the different rules. This is achieved by Prolog renaming the variables internally, when it comes to use the rule - usually the names are things like `_1`, `_2`, `_3`, etc. This is why variables with names like these sometimes turn up in error messages when your Prolog program goes wrong.

- to formulate queries, as in the following example:
  `?- studies(X, 9311).` Prolog responds by finding all the values for `X` for which `studies(X, 9311)` is true, and successively listing them, e.g.

```
X = fred ;
X = jane ;
X = abdul ;
X = maria ;
```

W

white space

In writing code in Prolog or any other programming language, white space (blank lines, indentation, perhaps alignment of related data items) can be used to make the code easier to follow. Here are some suggestions:

- Put a blank line before the start of any prolog procedure.
- Group collections of facts using blank lines.
- Sometimes it helps to line up comments, but not at the expense of making lines longer than about 75-80 characters.

- Don't put helper procedures in the middle of other procedures: separate them out before or after the procedure they help. If your helper procedure is used in several other procedures, you might want to put it right at the end of all your code.

Example. In reasonable size browser windows, the bad example will have long lines that are folded by the browser, and the good example will not. In practice, you could make the lines in the good example rather longer - up to 75-80 characters.

<table>
<tr><th>Good</th><th>Bad</th></tr>
<tr><td>% smallest(ListOfNumbers, Min)
% binds Min to the smallest item
% in the non-empty ListOfNumbers.
% ListOfNumbers should be instantiated
% at time of call.
smallest([FirstNum], FirstNum).
smallest([FirstNum | Rest], Min) :-
      smallest(Rest, MinRest),
      smaller(FirstNum, MinRest, Min).

% smaller(First, Second, Bigger)
% helper procedure for smallest.
smaller(A, B, A) :-
      A <= B.
smaller(A, B, B) :-
      B < A.</td><td>% smallest(ListOfNumbers, Min) binds Min to the smallest item in the non-empty ListOfNumbers. ListOfNumbers should be instantiated at time of call.
smaller(A, B, A) :- A <= B.
smaller(A, B, B) :- B < A.
% This is a helper procedure for smallest.
smallest([FirstNum], FirstNum).
smallest([FirstNum | Rest], Min) :- smallest(Rest, MinRest), smaller(FirstNum, MinRest, Min).</td></tr>
</table>

See also indentation and comments.

writing

Article on writing coming real soon now.

X

Y

Z

The AI Dictionary

Contact Info Last updated:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

You should use The AI Dictionary to clarify or revise concepts that you have already met. The AI Dictionary is *not* a suitable way to *begin* to learn about AI. Further information on AI can be found in the class web page lecture notes section.

Other related dictionaries:

The Prolog Dictionary - URL: `http://www.cse.unsw.edu.au/~billw/prologdict.html`
The Machine Learning Dictionary - URL: `http://www.cse.unsw.edu.au/~billw/mldict.html`
The NLP Dictionary (Natural Language Processing) - URL: `http://www.cse.unsw.edu.au/~billw/nlpdict.html`

Other places to find out about artificial intelligence include the AAAI (American Association for Artificial Intelligence) AI Overview page or AI Reference Shelf

The URL of this AI Dictionary is `http://www.cse.unsw.edu.au/~billw/aidict.html`

Topics Covered

This dictionary is limited to the AI concepts covered in COMP9414:

ako | backward chaining | best first search | breadth first search | certainty factor | condition-action rule | conflict resolution | default | demon | if added | if removed | if replaced | if needed | if new | range | help demon | cache | multi_valued | depth first search | edge | expert system | facet | fire | forward chaining | frame | generic frame | goal state | graph | directed acyclic graphs | trees | binary trees | adjacency matrices | graph search algorithms | heuristic | inference engine | inheritance | initial state | instance frame | isa | match-resolve-act cycle | node | operator | state | initial state | goal state | path | procedural attachment | ripple down rules (RDR) | rule-based system | search | semantic network | slot | vertex | working memory

A

ako

"ako" signifies "a kind of". It acts as a relation between two types of things, and specifies that one is a subset of the other. For example, `chair ako furniture` signifies that the concept of `chair` is a subconcept of the concept of `furniture`, or to put it another way, the set of all chairs is a subset of the set of all furniture.

"ako" acts like an operator in the iProlog frame implementation. For example, assuming that a `furniture` frame already existed, `chair ako furniture with ...` would have the effect of creating a new generic frame with all the slots of the `furniture` generic frame, together with whatever extra slots were specified after the `with`.

"ako" should be contrasted with isa.

B

backward chaining

Backward chaining is a means of utilizing a set of condition-action rules. In backward chaining, we work back from possible conclusions of the system to the evidence, using the rules backwards. Thus backward chaining behaves in a goal-driven manner.

One needs to know which possible conclusions of the system one wishes to test for. Suppose, for example, in a medical diagnosis expert system, that one wished to know if the data on the patient supported the conclusion that the patient had some particular disease, D.

In backward-chaining, the goal (initially) is to find evidence for disease D. To achieve this, one would search for all rules whose action-part included a conclusion that the patient had disease D. One would then take each such rule and examine, in turn, the condition part of the rule. To support the disease D hypothesis, one has to show that these conditions are true. Thus these conditions now become the goals of the backward-chaining production system. If the conditions are not supported directly by the contents of working memory, we need to find rules whose action-parts include these conditions as their conclusions. And so on, until either we have established a chain of reasoning demonstrating that the patient has disease D, or until we can find no more rules whose action-parts include conditions that are now among our list of goals.

Backward-chaining is to be contrasted with forward chaining.

best first search

Best-first search, rather than plunging as deep as possible into the tree (as in depth-first search), or traversing each level of the tree in succession (as in breadth-first search), uses a heuristic to decide at each stage which is the best place to continue the search. Here is a more technical discussion of best-first search.

breadth first search

Breadth-first search is best understood with respect to a tree, though it can be applied to graphs in general, provided a *starting node* is nominated.

A complete breadth first search traverses every node of the tree or graph, starting from the root node or starting node, first processing, checking, or inspecting the root/starting node. In future we'll just say it "processes" the node. Next it processes the neighbours of the root/starting node (in some order), and then the neighbours of the neighbours, and so on, until all the nodes have been processed.

In the case of a tree, no node will be visited twice - this is a property of trees. In the case of a graph, whether a directed acyclic graph or a general graph, some nodes *will* be visited twice. On the second and subsequent visits, the node and its neighbours should be ignored. Thus a breadth-first algorithm on such graphs needs to mark each node as *visited* when it first encounters it, and check each node it comes to process to see if it has already been visited.

For the tree shown below, the order of visiting for a breadth first search would be: `A B C D E F G H I J`

```
       A
      / \
     /   \
    B     C
   / \   / \
  D   E F   G
 /       \   \
H         I   J
```

Compare depth first search.

C

certainty factor

A certainty factor is a number, often in the range -1 to +1, which is associated with a condition or an action of a rule. In more detail, each component of a condition may have an certainty factor associated with it - for example if the condition is of the form A **and** B, then there could be a certainty factor for A and a certainty factor for B.

A certainty factor of 1 means that the fact (or proposition) is highly certain. A certainty factor of 0 means no information about whether the proposition is true or not. A certainty factor of -1 means that the proposition is certainly false. A certainty factor of 0.7 means that the proposition is quite likely to be true, and so on.

The certainty factors of conditions are associated with facts held in working memory. Certainty factors for actions are stored as part of the rules.

Rules for manipulating certainty factors are given in the lecture notes on uncertain reasoning.

However, here is a simple example. Suppose that there is a rule

if P **then** Q (0.7)

meaning that if P is true, then, with certainty factor 0.7, Q follows. Suppose also that P is stored in working memory with an associated certainty factor of 0.8. Suppose that the rule above fires (see also match-resolve-act cycle). Then Q will be added to working memory with an associated certainty factor of 0.7 * 0.8 = 0.56.

condition-action rule

A condition-action rule, also called a *production* or *production rule*, is a rule of the form

if *condition* **then** *action*.

The condition may be a compound one using connectives like **and**, **or**, and **not**. The action, too, may be compound. The action can affect the value of working memoryvariables, or take some real world action, or potentially do other things, including stopping the production system.

The knowledge of many expert systems is principally stored in their collections of rules.

See also inference engine.

conflict resolution

Conflict resolution in a forward-chaining inference engine decides which of several rules that *could* be fired (because their condition part matches the contents of working memory should actually be fired. Conflict resolution proceeds by sorting the rules into some order, and then using the rule that is first in that particular ordering. There are quite a number of possible orderings that could be used:

Specificity Ordering

If a rule's condition part is a superset of another, use the first rule since it is more specialised for the current task.

Rule Ordering

Choose the first rule in the text, ordered top-to-bottom.

Data Ordering

Arrange the data in a priority list. Choose the rule that applies to data that have the highest priority.

Size Ordering

Choose the rule that has the largest number of conditions.

Recency Ordering

The most recently used rule has highest priority. The least recently used rule has highest priority. The most recently used datum has highest priority. The least recently used datum has highest priority

Context Limiting

Reduce the likelihood of conflict by separating the rules into groups, only some of which are active at any one time. Have a procedure that activates and deactivates groups.

D

default

In frames, a default (value) is a value to be assigned to a particular slot in an instance frame if a specific value has not be assigned or *added* to that slot. The default is specified as part of the corresponding generic frame, and thus is inherited by the instance frame when the instance frame is created.

demon

A demon is a facet of a slot in a frame which causes some action to be taken when the frame is accessed in certain types of ways. For example, an **if-needed** demon is activated or *triggered* if the value of the slot is required and a value has not yet been stored in the slot, and it should calculate or otherwise obtain a value for the slot, while a **range** demon is triggered if a new value is added to the slot, to check that the value added is permissible for this particular slot.

Here is a list of the demon types supported by the iProlog frame implementation:

if_added

demons are triggered when a new value is put into a slot.

if_removed

demons are triggered when a value is removed from a slot.

if_replaced

is triggered when a slot value is replaced.

if_needed

demons are triggered when there is no value present in an instance frame and a value must be computed from a generic frame.

if_new

is triggered when a new frame is created.

range

is triggered when a new value is added. The value must satisfy the range constraint specified for the slot.

help

is triggered when the range demon is triggered and returns false.

The following are not demons but demon-related slots in a frame.

cache

means that when a value is computed it is stored in the instance frame.

multi_valued

means that the slot may contain more than one value.

For examples, see the lecture notes.

depth first search

Depth-first search is best understood with respect to a tree, though it can be applied to graphs in general, provided a *starting node* is nominated.

A complete depth first search traverses every node of the tree or graph, starting from the root node or starting node, first processing, checking, or inspecting the root/starting node. In future we'll just say it "processes" the node. Next it considers (but does not yet process) the neighbours of the root/starting node. The neighbours should be ordered in some way. Suppose that the first neighbour is called F1. Depth-first search proceeds to search first the subtree or subgraph composed of the neighbours of F1. Suppose that the first of F1's neighbours is F2. Depth-first search proceeds by searching the subtree or subgraph composed of the neighbours of F2. And so on, until the bottom of the tree/graph is reached. Thus the algorithm can be expressed recursively as follows (for a tree):

```
to depthFirstSearch a tree with root R
  if tree is empty
  then % we're finished
  else
     let N1, N2, ..., Nk be the neighbours of R
         depthFirstSearch the subtree with root N1
         depthFirstSearch the subtree with root N2
         ...
         depthFirstSearch the subtree with root Nk
```

In the case of a tree, no node will be visited twice - this is a property of trees. In the case of a graph, whether a directed acyclic graph or a general graph, some nodes *will* be visited twice. On the second and subsequent visits, the node and its neighbours should be ignored. Thus a depth-first algorithm on such graphs needs to mark each node as *visited* when it first encounters it, and check each node it comes to process to see if it has already been visited.

For the tree shown below, the order of first visiting for a depth first search would be: `A B D H E C F I G J`

```
                 A
                / \
               /   \
              B     C
```

```
   / \   / \
  D   E F   G
 /       \   \
H         I   J
```

In the case of binary trees there are 3 common variants of depth-first search called *pre-order, in-order, and post-order traversal*. The variants distinguish between *first visiting* a node, and *processing* that node, i.e. doing something with the data stored in the node (e.g. printing out the name of the node). There are six possible orders for three objects, of which the three orders commonly used are as follows:

- process node, visit left subtree, visit right subtree. This variant is called pre-order traversal. In order traversal of the binary tree shown above processes the nodes in the order `A B D H E C F I G J` as for simple depth-first search.
- visit left subtree, process node, visit right subtree. This variant is called in-order traversal. In order traversal of the binary tree shown above processes the nodes in the order `H D B E A F I C G J`.
- visit left subtree, visit right subtree, process node. This variant is called post-order traversal. In order traversal of the binary tree shown above processes the nodes in the order `H D E B I F J G C A`.

Compare breadth first search.

directed acyclic graph

see graph

directed graph

see graph

E

edge

A component of a graph.

expert system

An expert system is a computer system intended to perform at the level of a human expert in whatever domain it aspires to deal with. Early expert systems included systems aimed at medical diagnosis. All expert systems must confront the issue of knowledge representation - that is, how the program will encode the knowledge of the human expert.

Knowledge representation methods include production rules, frames, ripple-down rules, semantic networks. Many or most expert systems are rule-based systems.

F

facet

Facets are the components of a slot in a frame. Most slots would have a **value** facet and a **default** facet. Many would also have facets corresponding to various demons such as a *range* or *if-needed* demon - each demon is a facet.

fire

A production is said to be ready to fire if its condition part matches the contents of working memory.

forward chaining

Forward chaining is a means of utilizing a set of condition-action rules. In this mode of operation, a rule-based system is data-driven. We use the data to decide which rules can fire, then we fire one of those rules, which may add to the data in working memory and then we repeat this process until we (hopefully) establish a conclusion.

For more details of this method of using rules, see inference engine.

Forward-chaining is to be contrasted with backward chaining.

frames

Frames are a knowledge representation technique. They resemble an extended form of record (as in Pascal and Modula-2) or struct (using C terminology) or class (in Java) in that they have a number of slots which are like fields in a record or struct, or variable in a class. Unlike a record/struct/class, it is possible to add slots to a frame dynamically (i.e. while the program is executing) and the contents of the slot need not be a simple value. If there is no value present in a slot, the frame system may use a default for frames of that type, or there may be a demon present to help compute a value for the slot. Documentation for frames as implemented in iProlog is available at http://www.cse.unsw.edu.au/~claude/teaching/AI/notes/prolog/Frames/Frames.html.

Demons in frames differ from methods in a Java class in that a demon is associated with a particular slot, whereas a Java method is not so linked to a particular variable.

G

generic frame

A frame that serves as a template for building instance frames. For example, a generic frame might describe the "elephant" concept in general, giving defaults for various elephant features (number of legs, ears, presence of trunk and tusks, colour, size, weight, habitat, membership of the class of mammals, etc.), which an instance frame would describe a particular elephant, say "Dumbo", who might have a missing tusk and who would thus have the default for number of tusks overridden by specifically setting number of tusks to 1. Instance frames are said to inherit their slots from the generic frame used to create them. Generic frames may also inherit slots from other generic frames of which they are a subconcept (as with *mammal* and *elephant* - elephant inherits all the properties of mammal that are encoded in the *mammal* generic frame - warm blood, bear young alive, etc.)

goal state

See article on operators and states.

graph

This material should be familiar to you if you have studied discrete mathematics, either in a mathematics department, or, for example, in the subject COMP9020 Foundations of Computer Science.

Technically, a (directed) graph is a set V of **vertices** or **nodes**, together with a set E of **directed edges**, which are ordered pairs of vertices. Graphs are widely used in computer science as a modelling tool. A simple example of a graph would be V = {1, 2, 3} and E = {(1,2), (3,1)}, which could be drawn as:

3 -----> 1 -----> 2

Usually it is convenient to have names or "labels" for the nodes - technically this could be expressed as a mapping from V to a set VLabels of labels for vertices. Sometimes it is useful (as in semantic networks), to have labels for the edges. Again, technically this can be expressed as a function from E to a set ELabels of labels for edges.

It is also possible to have undirected graphs, in which the edges are not ordered but rather unordered pairs. Consider the possibility of edges from a node to itself - sometimes these could be useful, sometimes not.

A *directed cycle* in a directed graph is a sequence of edges (v1, v2), (v2, v3), ..., (vn, v1) such that the second vertex of the final edge is the same as the first vertex of the first edge. Here is a simple example:

```
1 ------> 3
^         |
|         |
|         v
2 <------ 4
```

The *in-degree* of a vertex v in a directed graph is the number of directed edges (u, v) such that v is the second vertex of the edge. In the following example, vertex b has in-degree 2 and the other vertices have in-degree 0. In the example above, all vertices have in-degree 1.

```
            a
            |
            |
            v
            b <------ c
```

The *out-degree* of a vertex u in a directed graph is the number of directed edges (u, v) such that u is the first vertex of the edge. In the example above, vertex b has out-degree 0 and the other vertices have out-degree 1.

Directed acyclic graphs are graphs with no directed cycles.

Trees are a special kind of directed graph, in which there is a special vertex, called the *root*, and which has in-degree 0, and every other vertex has in-degree 1.

Binary trees are a special kind of tree, in which every node has out-degree at most 2.

Graphs can be represented in a variety of ways. One can represent them directly as lists or other collections of (directed) edges:

```
edge(1, 3).
edge(3, 4).
edge(4, 2).
edge(2, 1).
```

or

```
[[1, 3], [3, 4], [4, 2], [2, 1]]
```

It is also possible to use adjacency matrices, a representation using a matrix of 0s and 1s:

```
0 0 1 0
1 0 0 0 .... the 1 in the (2,1)-position tells us (2,1) is an edge
0 0 0 1
0 1 0 0
```

The structures represented using graphs in Artificial Intelligence (and elsewhere) frequently need to be searched. There are a number of graph search algorithms for this purpose.

graph search algorithms

See breadth first search, depth first search, and best first search. There are many other search concepts - see the textbook by Bratko or elsewhere: look for **minimax search**, and **alpha-beta pruning** for a start, if interested.

H

heuristic

A heuristic is a fancy name for a "rule of thumb" - a rule or approach that doesn't always work or doesn't always produce completely optimal results, but which goes some way towards solving a particularly difficult problem for which no optimal or perfect solution is available.

I

inference engine

A rule-based system requires some kind of program to manipulate the rules - for example to decide which ones are ready to fire. (i.e. which ones have conditions that match the contents of working memory). The program that does this is called an inference engine, because in many rule-based systems, the task of the system is to infer something, e.g. a diagnosis, from the data using the rules. See also match-resolve-act cycle.

inheritance

Inheritance is a property of semantic networks and frames. It works as follows.

In the case of a semantic network, if I try to use the network to retrieve, say, the number of legs of a node (with name "Dumbo") the system will first look to see if the "Dumbo" node has an explicit "legs" link. If so, it is followed. If not, inheritance is applied, and, since "Dumbo" is an object rather than a type, the **isa** link is followed. Assuming "Dumbo" *isa* `elephant`, we then check the "elephant" node to see if it has a link labelled "legs". If so, we use it. If not, we look to see if "elephant" has a **ako** link, perhaps to "mammal". If mammal has a link labelled "legs" then we use it. If not, then we look for further "ako" links from either "elephant" or "mammal", and so on, until we either find the "legs" or run out of semantic network to search.

In the case of a frame, the effect is the same, but the details are different - an "elephant" frame would have been constructed using the information in a "mammal" frame, since `elephant ako mammal`. The "Dumbo" instance frame will have been constructed using the "elephant" generic frame since `Dumbo isa elephant`. Thus `elephant` inherits all the properties of `mammal` and `Dumbo` inherits all the properties of `elephant`. At each stage, there is an opportunity to change e.g. the number of legs if Dumbo should happen to an aberrant elephant in the matter of legs.

initial state

See article on operators and states.

instance frame

See generic frames.

isa

"isa" signifies "is a". It acts as a relation between an object and a type, and specifies that the object is a member of the type. For example, `Fido isa dog` signifies that the object `Fido` is a member of the set of all `dog`s.

"isa" acts like an operator in the iProlog frame implementation. For example, `X isa dog with ...` would have the effect of binding X to a new instance frame with all the slots of the `dog` generic frame, together with whatever extra information (such as slot values) was provided after the `with`.

"isa" should be contrasted with ako.

J

K

L

M

match-resolve-act cycle

The match-resolve-act cycle is the algorithm performed by a forward-chaining inference engine. It can be expressed as follows:

```
loop
    1. match all condition parts of condition-action rules
       against working memory and collect all the rules that match;
    2. if more than one match, resolve which to use;
    3. perform the action for the chosen rule
until action is STOP or no conditions match
```

Step 2 is called conflict resolution. There are a number of conflict resolution strategies.

N

node

A component of a graph.

May also be a component of a neural network (see node in a neural network.

O

operators and states

In problem solving, the term **operator** is used to describe one of the procedures that can be used to move from one **state** to another. One starts at an **initial state** and uses the allowable operators to move *towards* the or a **goal state**.

The sequence of states and operators (or sometimes just the states) that lead from the initial state to the goal state is referred to as a **path**. The trick lies in choosing operators that do in fact lie along a path, and preferably a short or cheap path towards the or a goal state. Sometimes it is possible to navigate intelligently through **state space**, but sometimes blind backtracking search through state space is the only possibility.

In most cases, it is necessary to check, at each state, which operators are feasible at this particular point. If, for example, the states were physical positions in some real or simulated terrain, the operators moved one in different physical directions, it would be necessary to check for example that there were no barriers in some directions, or that there were no bad consequences for steps in certain directions. For example, a step to the West might land lead to a state which could not be escaped from with any of the available operators.

P

path

See article on operators and states.

procedural attachment

Procedural attachment refers to the practice of attaching small *procedures* called demons to slots in frames. Demons in frames resemble *methods* in object-oriented languages such as Java, and historically precede them.

production *or* **production rule**

See condition-action rule.

Q

R

ripple-down rules

Not covered in COMP9414. Discussed in COMP9416.

rule-based system

A rule-based system is one based on condition-action rules. See also backward chaining, forward chaining, inference engine, working memory, and match-resolve-act cycle.

S

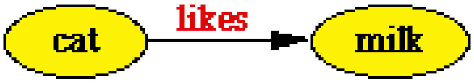
search

Search is a prevalent metaphor in artificial intelligence. Many types of problems that do not immediately present themselves as requiring search can be transformed into search problems. An example is problem solving, which can be viewed in many cases as search a state space, using operators to move from one state to the next.

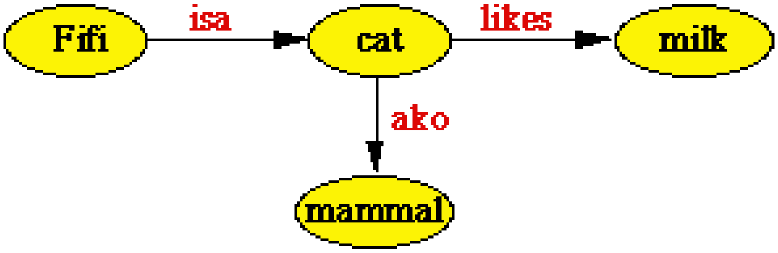
Particular kinds of search are described under the headings breadth-first search, depth-first search, and best-first search

semantic network

Semantic networks are a knowledge representation technique. More specifically, it is a way of recording all the relevant relationships between members of set of objects and types. "Object" means an individual (a particular person, or other particular animal or object, such as a particular cat, tree, chair, brick, etc.). "Type" means a set of related objects - the set of all persons, cats, trees, chairs, bricks, mammals, plants, furniture, etc. Possible relationships include the special set-theoretic relationships **isa** (set membership) and **ako**(the subset relation), and also general relationships like **likes**, **child-of**. Technically a semantic network is a node- and edge-labelled directed graph, and they are frequently depicted that way. Here is a pair of labelled nodes and a single labelled edge (relationship) between them (there could be more than one relationship between a single pair):



Here is a larger fragment of a semantic net, showing 4 labelled nodes (Fifi, cat, mammal, milk) and three labelled edges (isa, ako, likes) between them.



slot

A slot in a frame is like a field in a record or struct in languages like Pascal, Modula-2 and C. However, slots can be added dynamically to frames, and slots contain substructure, called facets.

The facets would normally include a value, perhaps a default, quite likely some demons, and possibly some flags like the iProlog frame system's **cache** and **multi_valued** facets.

state

See the article on operators and states.

T

tree

see graph

U

V

vertex

A component of a graph. The irregular plural of vertex is *vertices*.

W

working memory

The working memory of a rule-based system is a store of information used by the system to decide which of the condition-action rules is able to be fired. The contents of the working memory when the system was started up would normally include the input data - e.g. the patient's symptoms and signs in the case of a medical diagnosis system. Subsequently, the working memory might be used to store intermediate conclusions and any other information inferred by the system from the data (using the condition-action rules.

The term "working memory" is also used in cognitive psychology, where it refers to the limited store of "chunks" (roughly, items in memory) available at the same time for conscious processing.

Y

Z

The Natural Language Processing Dictionary

aka NLP Dictionary

Contact Info Last updated:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Other related dictionaries:

The Prolog Dictionary - URL: `http://www.cse.unsw.edu.au/~billw/prologdict.html`
The Artificial Intelligence Dictionary - URL: `http://www.cse.unsw.edu.au/~billw/aidict.html`
The Machine Learning Dictionary - URL: `http://www.cse.unsw.edu.au/~billw/mldict.html`

The URL of this NLP Dictionary is: `http://www.cse.unsw.edu.au/~billw/nlpdict.html`

You should use The NLP Dictionary to clarify or revise concepts that you have already met. The NLP Dictionary is *not* a suitable way to *begin* to learn about NLP. Further information on NLP can be found in the class web page lecture notes section.
Other places to find out about artificial intelligence include the AAAI (American Association for Artificial Intelligence) AI Overview page or AI Reference Shelf

If you wish to suggest an item or items that should be included, or if you found an item that you felt was unclear, please let me know (E-mail: `billw at cse.unsw.edu.au`).

Topics Covered

This dictionary is mainly limited to the NLP concepts covered or mentioned in COMP9414 Artificial Intelligence at the University of New South Wales, Sydney.

A abstract noun | accepter | active arc | active chart | active voice | ADJ | adjective | adjective phrase | ADJP | ADV | adverb | adverbial phrase | ADVP | agent | agreement | alphabet (in grammar) | ambiguity | anaphor | animate | antecedent | apposition | article | aspect | ATN | augmented grammar | augmented transition networks | AUX | auxiliary verb

B Bayes' rule | BELIEVE | bigram | bitransitive | bottom-up parser | bound morpheme

C cardinal | case | cataphor | CFG | chart | chart parsing | Chomsky hierarchy | CNP | co-agent | co-refer | common noun | common noun phrase | complement | compositional semantics | concrete noun | conditional probability | CONJ | conjunction | constituent | context-free | context-sensitive | corpus | count noun | CSG

D DE | DE list | declarative | demonstrative | derivation | descriptive grammar | determiner | discourse entity | discourse entity list | distinguished non-terminal

E ellipsis, elliptical | embedded sentence | evoke | exists | experiencer

F failure of substitutivity | features in NLP | first person | FOPC | forall | free morpheme | FUT | future perfect

G gender | generalized phrase structure grammar | generate | GPSG | grammar | grammar rule

H HDPSG | head feature | head subconstituent | head-driven phrase structure grammar | hidden Markov model | history list | HMM

I ill-formed text | imperative | independence (statistical) | indicative | infinitive | inflection | instrument | intensifier | INTERJ | interjection | intransitive

K KB | knowledge base | knowledge representation | knowledge representation language | KRL

L lambda reduction | language generated by a grammar | left-to-right parsing | lemma | lexeme | lexical category | lexical functional grammar | lexical generation probability | lexical insertion rule | lexical symbol | lexicon | LFG | local discourse context | logical form | logical operator

M Marcus parsing | mass noun | modal | mood | morpheme | morphology | MOST1

N N | n-gram | nominal | non-terminal | noun | noun modifier | noun phrase | NP | number (grammatical)

O object | ordinal | output probability

P parse tree | parser | part of speech | part-of-speech tagging | participle | particle | passive voice | PAST | past perfect | patient | person | phone | phoneme | phonetics | phonology | phrasal category | phrasal verb | phrase | pluperfect | PLUR | plural noun | possessive | PP | PP attachment | pragmatics | pre-terminal | predicate | predicate operator | predictive parser | PREP | preposition | prepositional phrase | PRES | prescriptive grammar | present perfect | production | progressive | proper noun | proposition

Q qualifier | quantifier | quantifying determiner |

R reference | referential ambiguity | regular | relative clause | rewriting process | right-linear grammar | right-to-left parsing | robustness

S S | second person | semantic grammar | semantics | sentence | sentential form | shift-reduce parser | simple future | simple past | simple present | singular noun | Skolem functions and constants | skolemization | speech act | start symbol | statistical NLP | string | structural ambiguity | SUBCAT | subcategorization | subject | subjunctive | substitutivity | surface speech act | syntax | systemic grammar

T tagged corpus | tense | term | terminal | THE | thematic role | theme | third person | top-down parser | transitive | trigram

U unrestricted grammar

V V | VAR feature | verb | verb complement | verb group | verb phrase | VFORM | victim | Viterbi algorithm | VP

W wh-question | word | word-sense ambiguity

Y y/n question

A

abstract noun

An abstract noun is a noun that does not describe a physical object, for example *philosophy*. Contrast common noun.

accepter

An accepter is a program (or algorithm) that takes as input a grammar and a string of terminal symbols from the alphabet of that grammar, and outputs `yes` (or something equivalent) if the string is a sentence of the grammar, and `no` otherwise. Contrast parser.

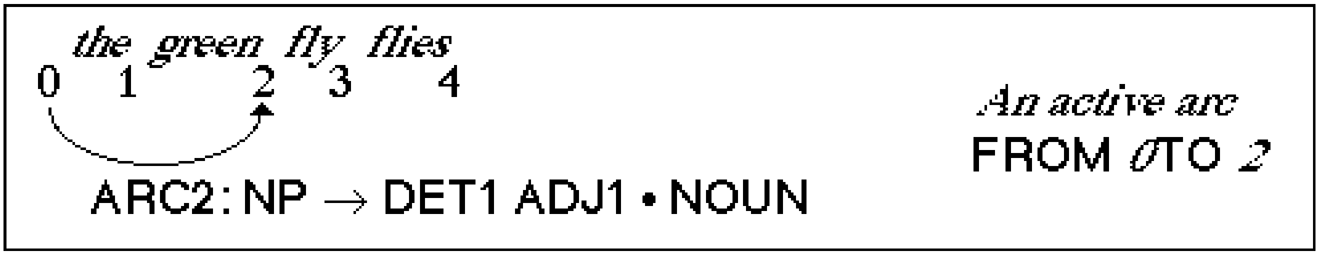
active arc

An active arc is a structure used by a chart parser as it attempts to parse a sentence. It is derived ultimately from a rule of the grammar being used, and consists of:

- a *name* for the arc,
- a *type* - the phrasal category being sought,
- a *list of found constituents*, i.e. constituents required by a grammar rule
- a *list of types of the constituents not yet found*,
- a *from* position, indicating the position in the sentence of the start of the first found constituent, and
- a *to* position, indicating the position in the sentence of the end of the last found constituent.

The symbol `->` is used to separated the type from the list of found constituents, and a dot is used to separate the list of found constituents from the list of types of constituents not yet found.

Example:



Here ARC2 is the name, NP is the type, DET1 ADJ1 is the list of found constituents, NOUN is the (single item) list of constituents not yet found, and the *from* and *to* positions are 0 and 2. This active arc would derive from the grammar rule NP -> DET ADJ NOUN which says that an NP may be an DETerminer followed by an ADJective followed by a NOUN.

active chart

= chart (in a chart parser)

active voice

Sentences in English may be in active or passive form. The active form makes the one who is performing the action in the sentence (termed the agent in semantics) the grammatical subject. This serves to focus attention on the agent. Example: "John ate the pizza". The alternative, the passive voice, makes the thing acted on into the grammatical subject, thus focussing attention on that thing, rather than on the agent. Example: "The pizza was eaten by John." Many writers appear to believe that use of the passive seems more formal and dignified, and consequently it is over-used in technical writing. For example, they might write "The following experiments were performed" when it would be clearer to say "We [i.e. the authors] performed the following experiments."

Contrast mood, tense, and aspect.

ADJ

symbol used in grammar rules for an adjective.

adjective

An adjective is a word that modifies a noun by specifying an attribute of the noun. Examples include adjectives of colour, like *red*, size or shape, like *round* or *large*, along with thousands of less classifiable adjectives like *willing*, *onerous*, etc. In grammar rules, we use the symbol ADJ for the pre-terminal category of adjectives.

Adjectives are also used as the complements of sentences with verbs like "be" and "seem" - "He is happy", "He seems drunk".

ADJ is a lexical grammatical category.

adjective phrase

Adjective Phrase (or adjectival phrase) is a phrasal grammatical category. Adjective phrase is usually abbreviated to ADJP. They range from simple adjectives (like "green" in "the green grass") through short lists of adjectives possibly modified by an adverb or so (like "really large, cream" in "that really large, cream building") to fairly complicated constructs like "angry that he had been ignored" in "Jack was angry that he had been ignored". The longer adjective phrases are frequently take the form of an adjective followed by a complement, which might be a "that"+Sentence complement (as in "angry that he had been ignored"), or a PP complement or a "to"+VP complement.

The longer ADJPs are most often found as complements of verbs such as "be" and "seem". ADJP is a phrasal grammatical category.

ADJP

symbol used in grammar rules for an adjective phrase.

ADV

symbol used in grammar rules for an adverb.

adverb

An adverb is a word that modifies a verb, ("strongly", in "she swam strongly") an adjective, ("very", in "a very strong swimmer") or another adverb ("very", in "she swam very strongly").

Many adverbs end with the morpheme -ly, which converts an adjective X into an adverb meaning something like "in an X manner" - thus "bravely" = "in a brave manner". Other adverbs include intensifiers like "very" and "extremely". There are also adverbs of time (like "today", "tomorrow", "then" - as in "I gave him the book then"), frequency ("never", "often"), and place ("here", "there", and "everywhere").

ADV is a lexical grammatical category.

adverbial phrase

Adverbial phrases are phrases that perform one of the functions of an adverb. They include simple phrases that express some of the same types of concepts that a single adverb might express, such as frequency - "every week", duration - "for three weeks", time - "at lunchtime", and manner - "this way" ("Do it this way"), or "by holding his head under water for one minute".

Adverbial Phrase is a phrasal grammatical category. Adverbial phrase is usually abbreviated to ADVP.

ADVP

symbol used in grammar rules for an adverbial phrase.

agent

A AGENT is a case used in logical forms. It signifies the entity that is *acting* in an *event*. It normally corresponds to the syntactic subject of an active voice declarative sentence. In the logical form for a state description, the term EXPERIENCER is used for the corresponding entity. AGENTs appear in the frame-like structures used to describe logical forms: e.g. the following, representing "John breaks it with the hammer":

```
(BREAK e1 [AGENT (NAME j1 "John"]
          [THEME (PRO i1 IT1)]
          [INSTR <THE h1 HAMMER>])
```

agreement

Agreement is the phenomenon in many languages in which words must take certain inflections depending on the company they keep. A simple case occurs with verbs in the third person singular form and their singular subjects: "Jane likes cheese" is correct, but * "Jane like cheese" and * "My dogs likes cheese" are not, because the subjects and verbs do not agree on the number feature. The name used in the lecture notes for the agreement feature is **AGR**. The possible values of the **AGR** feature are 1s, 2s, 3s, 1p, 2p, 3p, signifying 1st person singular, 2nd person singular, ..., 3rd person plural. Pronouns like "I" and "me" have **AGR**=1s, "you" has **AGR**={2s,2p} as it is not possible to distinguish singular from plural in this case, and so on. Definite noun phrases like "the green ball" have **AGR**=3s.

Allen

This refers to the book by James Allen, *Natural Language Processing*, second edition, Benjamin Cummings, 1995.

alphabet (in grammar)

The "alphabet" of a grammar is the set of symbols that it uses, including the terminal symbols (which are like words) and the non-terminal symbols which include the grammatical categories like N (noun), V (verb), NP (noun phrase), S (sentence), etc.

See also context-free grammar, and context-sensitive grammar.

ambiguity

An ambiguity is a situation where more than one meaning is possible in a sentence. We consider three types of ambiguity:

word-sense ambiguitystructural ambiguityreferential ambiguity

There can be situations where more than one of these is present.

anaphor

An anaphor is an expression that refers back to a previous expression in a natural language discourse. For example: "Mary died. *She* was very old." The word *she* refers to *Mary*, and is described as an anaphoric reference to *Mary*. *Mary* is described as the **antecedent** of *she*. Anaphoric references are frequently pronouns, as in the example, but may also be definite noun phrases, as in: "Ronald Reagan frowned. The President was clearly worried by this issue." Here *The President* is an anaphoric reference to *Ronald Reagan*. Anaphors may in some cases not be *explicitly* mentioned in a previous sentence - as in "John got out his pencil. He found that *the lead* was broken." *The lead* here refers to a subpart of *his pencil*. Anaphors need not be in the immediately preceding sentence, they could be further back, or in the same sentence, as in "John got out his pencil, but found that *the lead* was broken." In all our examples so far the anaphor and the antecedent are noun phrases, but VP and sentence-anaphora is also possible, as in "I have today dismissed the prime minister. It was my duty in the circumstances." Here *It* is an anaphoric reference to the VP *dismissed the prime minister*.

For a fairly complete and quite entertaining treatment of anaphora, see Hirst, G. *Anaphora in Natural Language Understanding: A Survey* Springer Lecture Notes in Computer Science 119, Berlin: Springer, 1981.

animate

A feature of some noun phrases. It indicates that the thing described by the noun phrase is alive, and so capable of acting, i.e. being the agent of some act. This feature could be used to distinguish between *The hammer broke the window* and *The boy broke the window* - in the former, the hammer is not animate, so cannot be the agent of the *break* action, (it is in fact the instrument), while the boy *is* animate, so can be the agent.

antecedent

See anaphor.

apposition

A grammatical relation between a word and a noun phrase that follows. It frequently expresses equality or a set membership relationship. For example, "Rudolph the red-nosed reindeer [had a very shiny nose]" - here Rudolph = the unique red-nosed reindeer. Another example, "Freewheelin' Franklin, an underground comic-strip character, [was into drugs and rock music]", expresses a set membership relation: Freewheeling_Franklin in "underground comic-strip characters".

article

Words like "the", "a", and "an" in English. They are a kind of determiner. See also the **quantifying logical operator** THE.

aspect

The phrase "I am reading" is in the *progressive aspect*, signifying that the action is still *in progress*. Contrast this with "I read" which does not likely refer to an action that is currently in progress. Aspect goes further than this, but we shall not pursue the details of aspect in this subject. If interested, you could try Huddleston, R., "Introduction to the Grammar of English" Cambridge, 1984, pp. 157-158 and elsewhere.

ATN

= augmented transition network

augmented grammar

An augmented grammar is what you get if you take grammar rules (usually from a context-free grammar) and add extra information to them, usually in the form of feature information. For example, the grammar rule s -> np vp can be augmented by adding feature information to indicate that say the AGR feature for the vp and the np must agree:

(s AGR ?agr) -> (np AGR ?agr) (vp AGR ?agr)

This is in an Allen-like notation. In Prolog we would write something like:

```
s(P1, P3, Agr) :- np(P1, P2, Agr), vp(P2, P3, Agr).
```

Actually, this is too tough - the AGR feature of a VP, in particular, is usually fairly ambiguous - for example the verb "love" (and so any VP of which it is the main verb) has AGR=[1s,2s,1p,2p,3p], and we would want it to agree with the NP "we" which has AGR=[1p]. This can be achieved by computing the intersection of the AGRs of the NP and the VP and setting the AGR of the S to be this intersection, provided it is non-empty. If it is empty, then the S goal should not succeed.

```
s(P1, P3, SAgr) :-
        np(P1, P2, NPAgr),
        vp(P1, P2, VPAgr),
        intersection(NPAgr, VPAgr, SAgr),
        nonempty(SAgr).
```

where `intersection` computes the intersection of two lists (regarded as sets) and binds the third argument to this intersection, and `nonempty` succeeds if its argument is not the empty list.

Augmented grammar rules are also used to record SEM and VAR features in computing logical forms, and to express the relationship between the SEM and VAR of the left-hand side and the SEM(s) and VAR(s) of the right-hand side. For example, for the rule vp -> v (i.e. an intransitive verb), the augmented rule with SEM feature could be:

(vp SEM (lambda X (?semv ?varv X)) VAR ?varv) ->
(v SUBCAT _none SEM ?semv VAR ?varv)

where SUBCAT _none indicates that this only works with an intransitive verb.

augmented transition network

A parsing formalism for augmented context free grammars. Not covered in current version of COMP9414, but described in Allen.

AUX

symbol used in grammar rules for an auxiliary verb.

auxiliary verb

A "helper" verb, not the main verb. For example, in "He would have read the book", "would" and "have" are auxiliaries. A reasonably complete list of auxiliary verbs in English is:

Auxiliary	**Example**
do	I did read
have	He has read
be	He is reading
shall/will/should/would	He should read
can, could	She can read
may, might, must	She might read

Complex groupings of auxiliaries can occur, as in "The child *may have been being taken* to the movies".

Some auxiliaries (*do*, *be*, and *have*) can also occur as verbs in their own right.

Auxiliary verb is often abbreviated to AUX.

AUX is a lexical grammatical category.

B

Bayes' rule

This statistical rule relates the conditional probability Pr(A | B) to Pr(B | A) for two events A and B. The rule states that

Pr(A | B) = Pr(B | A) * Pr(A) / Pr(B)

BELIEVE

BELIEVE is a modal operator in the language for representing logical forms. BELIEVE and other operators like it have some unexpected properties such as failure of substitutivity. For more details, read page 237 in Allen. Page 542 ff. provides yet more on belief in NLP (but this material is well beyond the needs of COMP9414).

bigram

A bigram is a pair of things, but usually a pair of lexical categories. Suppose that we are concerned with two lexical categories L1 and L2. The term bigram is used in statistical NLP in connection with the conditional probability that a word will belong to L2 given that the preceding word was in L1. This probability is written Pr(L2 | L1), or more fully Prob(w[i] in L2 | w[i-1] in L1). For example, in the phrase "The flies", given that *The* is tagged with ART, we would be concerned with the conditional probabilities Pr(N | ART) and Pr(V | ART) given that *flies* can be tagged with N and V.

bitransitive

A verb in English that can take two objects, like *give*, as in "He gave his mother a bunch of flowers". Here "his mother" is the **indirect object** and "a bunch of flowers" is the **direct object**. The same sentence can also be expressed as "He gave a bunch of flowers *to* his mother", with the direct and indirect objects in the opposite order, and the indirect object marked by the preposition "to". The preposition in such cases is usually "to", or "for" (as in "He bought his mother a bunch of flowers" = "He bought a bunch of flowers *for* his mother."

Bitransitive verbs can appear with just one or even no syntactic objects ("I gave two dollars", "I gave at the office") - their distinguishing characteristic is that they *can* have two objects, unlike intransitive and transitive verbs.

bottom-up parser

A parsing method that proceeds by assembling words into phrases, and phrases into higher level phrases, until a complete sentence has been found. Contrast top-down.

The chart parser described in lectures is a bottom-up parser, and can parse sentences, using any context-free grammar, in cubic time: i.e., in time proportional to the cube of the number of words in the sentence.

bound morpheme

A bound morpheme is a prefix or suffix, which cannot stand as a word in its own right, but which, can be attached to a free morpheme and modify the meaning of the free morpheme. For example, "happy" is a free morpheme, which becomes "unhappily" when the prefix "un-", and suffix "-ly", both bound morphemes, are attached.

C

cardinal

Number words like one, two, four, twenty, fifty, hundred, million. Contrast ordinal.

case

The term *case* is used in two different (though related) senses in NLP and linguistics. Originally it referred to what is now termed **syntactic case**. Syntactic case essentially depends on the relationship between a noun (or noun phrase) and the verb that governs it. For example, in "Mary ate the pizza", "Mary" is in the nominative or subject case, and "the pizza" is in the accusative or object case. Other languages may have a wider range of cases. English has remnants of a couple more cases - genitive (relating to possession, as with the pronoun "his") and dative (only with bitransitive verbs - the indirect object of the verb is said to be in the dative case).

Notice that in "The pizza was eaten by Mary", "the pizza" becomes the syntactic subject, whereas it was the syntactic object in the equivalent sentence "Mary ate the pizza".

With **semantic case**, which is the primary sense in which we are concerned with the term *case* in COMP9414, the focus is on the meaning-relationship between the verb and the noun or noun phrase. Since this does not change between "Mary ate the pizza" and "The pizza was eaten by Mary", we want to use the same syntactic case for "the pizza" in both sentences. The term used for the semantic case of "the pizza" is theme. Similarly, the semantic case of "Mary" in both versions of the sentence is agent. Other cases frequently used include instrument, coagent, experiencer, **at-loc**, **from-loc**, and **to-loc**, **at-poss**, **from-poss**, and **to-poss**, **at-value**, **from-value**, and **to-value**, **at-time**, **from-time**, and **to-time**, and **beneficiary**.

Semantic cases are also referred to as **thematic roles**.

cataphor

Opposite of anaphor, and much rarer in actual language use. A cataphor is a phrase that is explained by text that comes *after* the phrase. Example: "Although he loved fishing, Paul went skating with his girlfriend." Here *he* is a cataphoric reference to *Paul*.

CFG

= context-free grammar

chart

A chart is a data structure used in parsing. It consists of a collection of active arcs (sometimes also called **edges**), together with a collection of constituents, (sometimes also called **inactive arcs** or

inactive edges.

See also chart parsing.

chart parsing

A chart parser is a variety of parsing algorithm that maintains a table of *well-formed substrings* found so far in the sentence being parsed. While the chart techniques can be incorporated into a range of parsing algorithms, they were studied in lectures in the context of a particular bottom-up parsing algorithm.

That algorithm will now be summarized:

to parse a sentence S using a grammar G and lexicon L:

1. Initially there are no constituents or active arcs
2. Scan the next word *w* of the sentence, which lies between positions *i* and *i+1* in the sentence.
3. Look up the word *w* in the lexicon L. For each lexical category *C* to which *w* belongs, create a new constituent of type *C*, from *i* to *i+1*.
4. Look up the grammar G. For each category *C* found in the step just performed, and each grammar rule *R* whose right-hand side begins with *C*, create a new active arc whose rule is *R*, with the dot in the rule immediately after the first category on the right-hand side, and from *i* to *i+1*.
5. If any of the active arcs can have their dots advanced (this is only possible if the arc was created in a previous cycle of this algorithm) then advance them.
6. If any active arcs are now completed (that is, the dot is now after the last category on the right-hand side of the active arc's rule), then convert that active-arc to a constituent (or *inactive* arc), and go to step 4.
7. If there are any more words in the sentence, go to step 2.

to check if an active arc can have its dot advanced

1. Let the active arc be ARC*x*: *C* -> *C*[1] ... *C*[*j*] . *C*[*j*+1] ... *C*[*n*] from *m* to *n*.
2. If there is a constituent of type *C*[*j*] from *n* to *p*, then the dot can be advanced.
   The resulting new active arc will be:
   ARC*y*: *C* -> *C*[1] ... *C*[*j*+1] . *C*[*j*+2] ... *C*[*n*] from *m* to *n*
   where *y* is a natural number that has not yet been used in an arc-name.

Example: For the active arc ARC2: NP -> ART1 . ADJ N from 2 to 3 if there is a constituent ADJ2: ADJ -> "green" from 3 to 4 (so that the **to position**, 3, and the **type**, ADJ, of the constituent of the active arc immediately after the dot, match the **from position**, 3, and the **type**, ADJ, of the constituent ADJ2) then the active arc ARC2 can be extended, i.e. have its dot advanced, creating a new active arc, say ARC3: NP -> ART1 ADJ2 . N from 2 to 4.

Chomsky hierarchy

The Chomsky hierarchy is an ordering of types of grammar according to generality. The classification in fact only depends on the type of **grammar rule** or **production** used. The grammar types described in COMP9414 included:

- unrestricted grammars (rules of the form α -> β with no restrictions on the strings α and β)
- context sensitive grammars (rules of the form α -> β with the restriction length(α) <= length(β))
- context free grammars (rules of the form X -> β where X is a single non-terminal symbol)
- regular grammars (rules of the form X -> a and X -> aN where X and N are non-terminal symbols, and a is a terminal symbol.)

Named after the linguist Noam Chomsky.

CNP

symbol used in grammar rules for an common noun phrase.

co-agent

You really need to know what an agent is before proceeding. A co-agent is someone who acts *with* the agent in a sentence. A sentence with a prepositional phrase introduced by the preposition *with* and whose object is an animate is likely to be a coagent. "Jane ate the pizza *with her mother*" - *her mother* is the coagent.

co-refer

Two items (an anaphor and its antecedent) that describe the same thing are said to *co-refer*.

common noun

A common noun is a noun that describes a type, for example *woman*, or *philosophy* rather than an individual, such as *Amelia Earhart*. Contrast proper noun.

common noun phrase

A common noun phrase is a phrasal grammatical category of chiefly technical significance. Examples include "man" "big man" "man with the pizza", but not these same phrases with "the" or "a" in front - that is, "the man with the pizza", etc., are NPs, not a CNP. The need for the category CNP as a separate named object arises from the way articles like "the" act on a CNP. The word "the", regarded as a natural language quantifier, acts on the whole of the CNP that it precedes: it's "the[man with the pizza]", not "the[man] with the pizza". For this reason, it makes sense to make phrases like "man with the pizza" into syntactic objects in their own right, so that the semantic interpretation phase does not need to reorganize the structural description of the sentence in order to be able to interpret it.

complement

A complement is a grammatical structure required in a sentence, typically to complete the meaning of a verb or adjective. For example, the verb "believe" can take a **sentential complement**, that is, be followed by a sentence, as in "I believe you are standing on my foot."

There is a wide variety of complement structures. Some are illustrated in the entry for subcategorization.

An example of an adjective with a complement is "thirsty for blood", as in "The football crowd was thirsty for blood after the home team was defeated." This is a PP-complement. Another would be "keen to get out of the stadium", a TO-INF complement, as in "The away-team supporters were keen to get out of the stadium."

compositional semantics

Compositional semantics signifies a system of constructing logical forms for sentences or parts of sentences in such a way that the meanings of the components of the sentence (or phrase) are used to construct the meanings of the whole sentence (or whole phrase). For example, in "three brown dogs", the meaning of the phrase is constructed in an obvious way from the meanings of *three*, *brown* and*dogs*. By way of contrast, a phrase like "kick the bucket" (when read as meaning "die") does not have compositional semantics, as the meaning of the whole ("die") is unrelated to the meanings of the component words.

The semantic system described in COMP9414 assumes compositional semantics.

concrete noun

A concrete noun is a noun that describes a physical object, for example *apple*. Contrast abstract noun.

conditional probability

The conditional probability of event B *given* event A is the probability that B will occur given that we know that A has occurred. The example used in lecture notes was that of a horse Harry that won 20 races out of 100 starts, but of the 30 of these races that were run in the rain, Harry won 15. So while the probability that Harry would win a race (in general) would be estimated as 20/100, the conditional probability Pr(Win | Rain) would be estimated as 15/30 = 0.5. The formal definition of Pr(B | A) is Pr(B & A) / Pr(A). In the case of B = Win and A = Rain, Pr(B & A) is the probability that it will be raining and Harry will win (which on the data given above is 15/100), while Pr(A) is the probability that it will be raining, or 30/100. So again Pr(B | A) = 0.15/0.30 = 0.5

CONJ

symbol used in grammar rules for a conjunction.

conjunction

A conjunction is a word used to join two sentences together to make a larger sentence. Conjunctions include **coordinate conjunctions**, like "and", "or" and "but": "Jim is happy and Mary is proud", "India will win the test match or I'm a monkey's uncle".

There are also **subordinate conjunctions**, like "if" and "when", as in "I will play with you **if** you will lend me your marbles" and "I will lend you this book **when** you return the last one you borrowed".

Conjunctions may also be used to join nouns, adjectives, adverbs, verbs, phrases ...
Examples:

nouns	Boys and girls [come out to play].
adjectives	[The team colours are] black and yellow.
adverbs	[He was] well and truly [beaten].
verbs	[Mary] played and won [her match].
phrases	across the river and into the trees [She] fell down and hit her head.

Conjunction is often abbreviated to CONJ.

CONJ is a lexical grammatical category.

constituent

A constituent, in parsing, is a lexical or phrasal category that has been found in a sentence being parsed, or alternatively one that is being sought for but has not yet been found.

See active arc. When an active arc is completed (when all its sub-constituents are found), the active arc becomes a constituent.

Constituents are used to create new active arcs - when there is a constituent X1 of type X, and a grammar rule whose right hand side starts with the grammar symbol X, then a new active arc of type X may be created, with the constituent X1 listed as a found constituent for the active arc (the only one, so far).

The components of a constituent, as recorded in the chart parsing algorithm described in lectures, are:

component	*example*	NP1: NP -> ART1 ADJ1 N1 from 0 to 3

name	NP1	usually formed from the type + a number
type	NP	a phrasal or lexical category of the grammar
decomposition	ART1 ADJ1 N1	(ART1, ADJ1 and N1 would be the names of other constituents already found)
from	0	sentence position of the left end of this NP
to	3	sentence position of the right end of this NP

.

context-free

See context-free grammar and Chomsky hierarchy and contrast with context-sensitive grammar.

context-free grammar

A context-free grammar is defined to be a 5-tuple (P, A, N, T, S) with components as follows:

P	A set of **grammar rules** or **productions**, that is, items of the form X -> α, where X is a member of the set N, that is, a non-terminal symbol, and α is a string over the alphabet A. An example would be the rule NP -> ART ADJ N which signifies that a Noun Phrase can be an ARTicle followed by an ADJective followed by a Noun, or N -> *horse*, which signifies that *horse* is a Noun. NP, ART, ADJ, and N are all non-terminal symbols, and *horse* is a terminal symbol.
A	the alphabet of the grammar, equal to the disjoint union of N and T
N	the set of non-terminal symbols (i.e. grammatical or phrasal categories)
T	the set of terminal symbols (i.e. words of the language that the grammar defines)
S	a *distinguished* non-terminal, normally interpreted as representing a full sentence (or program, in the case of a programming language grammar)

context-sensitive

See context-sensitive grammar and Chomsky hierarchy and contrast with context-free grammar.

context-sensitive grammar

A context-sensitive grammar is a grammar with context-sensitive rules. There are two equivalent formulations of the definition of a context-sensitive grammar rule (cf. Chomsky hierarchy):

- rules of the form α -> β where α and β are strings of alphabet symbols, with the restriction that length(α) <= length(β)
- rules of the form λ X ρ -> λ β ρ where λ, ρ, and β are (possibly empty) strings of alphabet symbols, and X is a non-terminal. λ and ρ are referred to as the left and right context for X -> β in the context-sensitive rule.

Context-sensitive grammars are more powerful than context-free grammars, but they are much harder to work with.

corpus

A corpus is a large body of natural language text used for accumulating statistics on natural language text. The plural is *corpora*. Corpora often include extra information such as a tag for each word indicating its part-of-speech, and perhaps the parse tree for each sentence.

See also statistical NLP.

count noun

A noun of a type that can be counted. Thus *horse* is a count noun, but *water* is not. Contrast mass noun.

CSG

= context-sensitive grammar

D

DE

= discourse entity

DE list

See history list.

declarative

= indicative.

demonstrative

A kind of determiner, that is, an ingredient of noun phrases. This class of words includes "this", "that", "these", and "those". They are part of the reference system of English. That is, they is used to tell which of a number of possibilities for the interpretation of the rest of the noun phrase is in fact intended. Demonstratives are most useful in spoken language, and are often accompanied by a pointing gesture.

derivation

A derivation of a sentence of a grammar is, in effect, a proof that the sentence can be derived from the start symbol of the grammar using the grammar rules and a rewriting process. For example, given the grammar 1.S -> NP VP, 2. NP -> ART N, 3. VP -> V, and lexical rules 4. ART -> "the", 5. N -> "cat", and 6. V -> "miaowed", we can derive the sentence "the cat miaowed" as follows:

S => NP VP	rule 1
=> ART N VP	rule 2

=> the N VP	rule 4
=> the cat VP	rule 5
=> the cat V	rule 3
=> the cat miaowed	rule 6

One can then write S =>* "the cat miaowed": i.e. =>* is the symbol for the **derivation relation**. The symbol => is referred to as **direct derivation**. A **sentential form**is any string that can be derived (in the sense defined above) from the start symbol S.

descriptive grammar

The sense in which the term grammar is primarily used in Natural Language Processing. A grammar is a formalism for describing the syntax of a language. Contrast prescriptive grammar.

determiner

Determiners are one of the ingredients of noun phrases. Along with cardinals and ordinals, they make up the set of specifiers, which assist in reference - that is, determining exactly which of several possible alternative objects in the world is *referred* to by a noun phrase. They come in several varieties - articles, demonstratives, possessives, and quantifying determiners.

discourse entity

A discourse entity (DE) is a something mentioned in a sentence that could act as a possible antecedent for an anaphoric reference, e.g. noun phrases, verb phrases and sentences. For example, with the sentence "Jack lost his wallet in his car", the DEs would include representations of "Jack" "his wallet", "his car", "lost his wallet in his car" and the whole sentence. The whole sentence could serve as the antecedent for "it" in a follow-up sentence like "He couldn't understand it" (while "Jack" would be the antecedent of "He").

Sometimes discourse entities have a more complex relation to the text. For example, in "Three boys each bought a pizza", clearly "Three boys" gives rise to a DE that is a set of three objects of type boy (B1: |B1| = 3 **and** B1 subset_of {x|Boy(x)}), but "a pizza", in this context, gives rise to a representation of a set P1 of three pizzas (whereas in the usual case "a pizza" would give rise to a DE representing a single pizza.)
P1 = {p | pizza(p) **and** exists(b) : Boy(b) **and** y = pizza_bought_by(b)}.
The function "pizza_bought_by" is the Skolem function referred to in lectures as "sk4".

discourse entity list

See history list.

distinguished non-terminal

See context-free grammar.

E

ellipsis, elliptical

Ellipsis refers to situations in which sentences are abbreviated by leaving out parts of them that are to be understood from the context. For example, if someone asks "What is your name?" and the reply is "John Smith" then this can be viewed as an elliptical form of the full sentence "My name is John Smith".

Ellipsis causes problems for NLP since it is necessary to infer the rest of the sentence from the context.

"ellipsis" is also the name of the symbol "..." used when something is omitted from a piece of text, as in "Parts of speech include nouns, verbs, adjectives, adverbs, determiners, ... - the list goes on and on."

"elliptical" is the adjectival form of "ellipsis".

embedded sentence

An embedded sentence is a sentence that is contained inside another sentence. Some examples, with the embedded sentence in italics:

- John believes that *Mary likes pizza*
- If *Mary likes pizza* then she may come to our pizza party.
- If *Joan liked pizza*then she would come to our pizza party.

evoke

A noun phrase is said to *evoke* a discourse entity if the noun phrase refers to something related to a previously mentioned discourse entity (but not to an already-mentioned DE). For example, in "Jack lost his wallet in his car. Later he found it under the front seat.", the phrase "the front seat" evokes a discourse entity that has not actually been mentioned, but which is in a sense already present as *part of* the the DE created by the phrase "his car".

See also anaphor.

exists

"exists" is a textual way of writing the *existential quantifier*, which is otherwise written as an back-to-front capital E. It corresponds fairly closely to the English word "some". Thus,

exists(X, likes(X, spinach))

would be read as "for some entity X, X likes spinach" or just "something likes spinach". This might be too broad a statement, as it could be satisfied, for example, by a snail X that liked spinach. It is common therefore to restrict the proposition to something like:

exists(X, is_person(X) **and** likes(X, spinach))

i.e. "Some person likes icecream." That is, we are restricting the type of X to persons. In some cases, it is more reasonable to abbreviate the type restriction as follows:

exists(X : person, likes(X, spinach))

See also forall, Skolem functions and this riddle.

experiencer

Experiencer is a case that usually fills a similar syntactic role to the agent but where the entity involved cannot be said to *act*. It is thus associated with the use of particular verbs like "remember", as in "Jim remembered his homework when he got to school". Here "Jim" is the *experiencer* of the "remember" situation.

F

failure of substitutivity

In some situations, things that are equal cannot be substituted for each other in logical forms. Consider (BELIEVE SUE1 (HAPPY JACK1)) - JACK1 may = JOHN22 (i.e. the individual known as Jack may also be called John, e.g. by other people, but *Sue believes John is happy* may not be true, e.g. because Sue may not know that JACK1 = JOHN22. Thus JOHN22 cannot be substituted for JACK1, even though they are equal in some sense. See also Allen pp. 237-238.

features in NLP

Features can be thought of as slots in a lexicon entry or in structures used to build a logical form. They record syntactic or semantic information about the word or phrase. Examples include the **AGR** agreement feature, the **SEM** feature that records the logical form of a word or phrase, and the **VAR** feature that records the variable used to name the referent of a phrase in a logical form.

first person

One of the choices for the person feature. A sentence is "in the first person" if the subject of the sentence is the speaker, or the speaker and some other individual(s), as in "I like pizza" and "We like pizza".

"I" and "we" are first-person pronouns, as are "me", "us". Other words with the first-person feature include "mine", "my", "myself", "ours", "our", and "ourselves".

FOPC

This stands for First Order Predicate Calculus, a standard formulation of logic that has logical operators like **and**, **or**, and **not**, predicate symbols and constants and functions, and terms built from these, together with the quantifiers "forall and exists. It is common for semantic representation systems in NLP to be expressed in languages that resemble or are based on FOPC, though sometimes they add significant features of more elaborate logical systems.

forall

"forall" is a textual way of writing the *universal quantifier*, which is otherwise written as an upside-down capital A. It corresponds fairly closely to the English words "each" and "every". Thus,

forall(X, likes(X, icecream))

would be read as "for every entity X, X likes icecream" or just "everything likes icecream". This would be too broad a statement, as it would allege that, for example, rocks like icecream. It is usual therefore to restrict the proposition to something like:

forall(X, is_person(X) => likes(X, icecream))

i.e. "Every person likes icecream." That is, we are restricting the type of X to persons. In some cases, it is more reasonable to abbreviate the type restriction as follows:

forall(X : person, likes(X, icecream))

See also exists.

free morpheme

A free morpheme is a basic or root form of a word, to which can be attached bound morphemes that modify the meaning. For example, "happy" is a free morpheme, which becomes "unhappy" when the prefix "un-", a bound morpheme, is attached.

FUT

See modal operators - tense and tense - future.

future perfect

See tense.

G

gender

One of the features of a noun phrase. In English, gender is only marked in third-person singular pronouns and associated words. The possible values of the gender feature are *masculine*, *feminine*, and *neuter*.

type	*masculine*	*feminine*	*neuter*	*example*
pronoun (nominative)	he	she	it	he hit the ball.
pronoun (accusative)	him	her	it	Frank hit him.
pronoun(possessive adjective)	his	her	its	Frank hit his arm.
pronoun(possessive)	his	hers	its	The ball is his.
pronoun(reflexive)	himself	herself	itself	Frank hurt himself.

generalized phrase structure grammar (GPSG)

An alternative grammatical formalism, in which, among other things, the non-terminal symbols of context-free grammars are replaced by sets of features, and the grammar rules show the relationships between these objects much as context-free rules show the relationships between grammar symbols in a CFG.

generate

If there is a derivation of a sentence from a grammar, then the grammar is said to *generate* the sentence.

GPSG

= generalized phrase structure grammar

grammar

1. A system for describing a language, the rules of a language.
2. A formal system for describing the syntax of a language. In COMP9414, we are principally concerned with context-free grammars, sometimes augmented.

   See also Chomsky hierarchy.

grammar rule

See Chomsky hierarchy and context-free grammars.

H

HDPSG

= head-driven phrase structure grammar

head feature

A head feature is one for which the feature value on a parent category must be the same as the value on the head subconstituent. Each phrasal category has associated with it a **head**

subconstituent - N, NAME or PRO or CNP for NPs, VP for S, V for VP, P (= PREP) for PP. For example, VAR is a head feature for a range of phrasal categories, including S. This means that an S gets its VAR feature by copying the VAR feature of its head subconstituent, namely its VP. Head features are discussed on page 94-96 of Allen.

head subconstituent

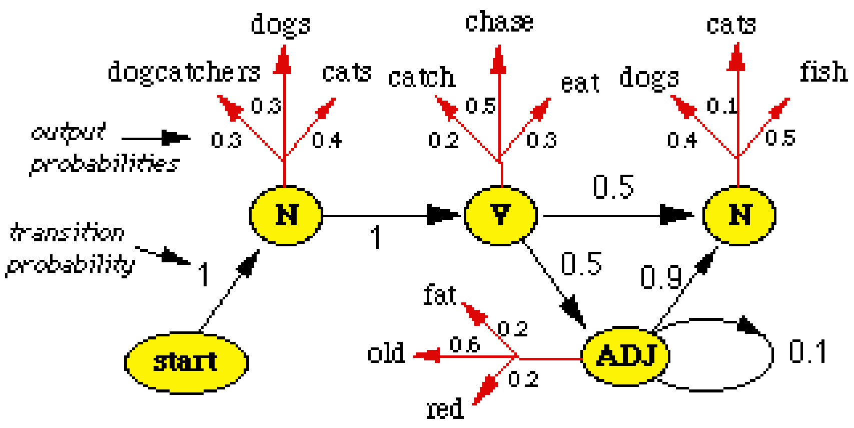
See head feature.

head-driven phrase structure grammar

A grammatical formalism. See Allen p. 154.

hidden Markov model

A Hidden Markov Model, for our purposes in COMP9414, is a set of states (lexical categories in our case) with directed edges (cf. directed graphs) labelled with **transition probabilities** that indicate the probability of moving to the state at the end of the directed edge, given that one is now in the state at the start of the edge. The states are also labelled with a function which indicates the probabilities of outputting different symbols if in that state (while in a state, one outputs a single symbol before moving to the next state). In our case, the symbol output from a state/lexical category is a word belonging to that lexical category. Here is an example:



Using this model, the probability of generating "dogcatchers catch old red fish" can be calculated as follows: first work out the probability of the lexical category sequence -> N -> V -> ADJ -> ADJ -> N, which is 1 * 1 * 0.5 * 0.1 * 0.9 = 0.045, and then multiply this by the product of the output probabilities of the words, i.e. by 0.3 * 0.2 * 0.6 * 0.2 * 0.5 = 0.0036, for a final probability of 0.000162.

history list

This is the list of discourse entities mentioned in recent sentences, ordered from most recent to least recent. Some versions also include the syntactic and semantic analyses of the previous sentence (or previous clause of a compound sentence). Some versions keep only the last few sentences worth of discourse entities, others keep all the discourse entities since the start of the discourse.

HMM

= Hidden Markov model

I

ill-formed text

Much "naturally occurring" text contains some or many typographical errors or other errors. Industrial-strength parsers have to be able to deal with these, just as people can deal with typos and ungrammaticality. Such a parser is called a **robust** parser.

Here is a list of 300+ ill-formed sentences.

imperative

An imperative sentence is one that expresses a command, as opposed to a question or a statement. See also WH-question, Y/N-question, indicative, subjunctive, and mood.

independence (statistical)

Two events A and B are said to be statistically independent if Pr(B | A) = Pr(B) - i.e. whether or not A is a fact has no effect on the probability that B will occur. Using Bayes' rule, this can be reformulated as Pr(A **and** B) = Pr(A) * Pr(B).

indicative

An indicative sentence is one that makes a statement, as opposed to a question or a command. See also WH-question, Y/N-question, imperative, subjunctive, and mood.

inflection

An inflection is a type of bound morpheme, with a *grammatical* function. For example, the suffix "-ing" is an inflection which, when attached to a participle form of the verb. Other inflections in English form the other parts of verbs (such as the past tense and past participle forms), and the plural of nouns.

Some words inflect regularly, and some inflect irregularly, like the plural form "children" of "child", and the past tense and past participle forms "broke" and "broken" of the verb "break".

instrument

A semantic case, frequently appearing as a prepositional phrase introduced by the preposition "with". For example, in "Mary ate the pizza with her fingers", the prepositional phrase "with her fingers" indicates the instrument used in the action described by the sentence.

intensifier

A kind of adverb, used to indicate the level or intensity of an adjective or another adverb. Examples include "very", "slightly", "rather", "somewhat" and "extremely". An example of use with an adjective: "Steve was somewhat tired". An example of use with an adverb: "Mary ran very quickly".

INTERJ

symbol used in grammar rules for an interjection.

interjection

Interjection is often abbreviated to INTERJ.

INTERJ is a lexical grammatical category. It usually appears as a single word utterance, indicating some strong emotion or reaction to something. Examples include: "Oh!", "Ouch!", "No!", "Hurray!" and a range of blasphemies and obscenities, starting with "Damn!".

intransitive

A verb that can take no syntactic object, like *laugh*, as in "He laughed loudly", or "She laughed at his remark". Contrast bitransitive and transitive. See also subcategorization.

J

K

KB

= knowledge base

knowledge base

See knowledge representation language.

knowledge representation

See knowledge representation language.

knowledge representation language

The term knowledge representation language (KRL) is used to refer to the language used by a particular system to encode the knowledge. The collection of knowledge used by the system is referred to as a knowledge base (KB).

KRL

= knowledge representation language

L

lambda reduction

The process of applying a lambda-expression to it's argument (in general, arguments, but the examples we've seen in COMP9414 have all been single argument lambda-expressions). A lambda expression is a formula of the form (lambda ?x P(?x)), in an Allen- like notation, or `lambda(X, p(X))` in a Prolog-ish notation. P(?x) (or p(X)) signifies a formula involving the variable ?x (or X). The lambda-expression can be viewed as a function to be applied to an argument. The result of applying (lambda ?x P(?x)) to an argument A is P(A) - that is, the formula P(?x) with all the instances of the variable ?x replaced by A. (In the Prolog version, applying `lambda(X, p(X))` to `a` results in `p(a)`.)
Using a more real example, if we apply (lambda ?x (EAT1 l1 ?x PIZZA1)) to MARY1 we get (EAT1 l1 MARY1 PIZZA1)).

Prolog code for lambda-reduction is:

```
lambda_reduce(lambda(X, Predicate), Argument, Predicate) :-
     X = Argument.
```

Applying this to an actual example:

```
: lambda_reduce(
         lambda(X, eats(e1, X, the1(p1, pizza1))),
         name(m1, "Mary"),
         Result) ?

X = name(m1, "Mary")
Result = eats(e1, name(m1, "Mary"), the1(p1, pizza1))
```

language generated by a grammar

The language generated by a grammar is the set of all sentences that can be derived from the start symbol S of the grammar using the grammar rules. Less formally, it is the set of all sentences that "follow from" or are consistent with the grammar rules.

left-to-right parsing

Parsing that processes the words of the sentence from left to right (i.e. from beginning to end), as opposed to right-to-left (or end-to-beginning) parsing. Logically it may not matter which direction parsing proceeds in, and the parser will work, eventually, in either direction. However, right-to-left parsing is likely to be less intuitive than left-to-right. If the sentence is damaged (e.g. by the presence of a mis-spelled word) it may help to use a parsing algorithm that incorporates both left-to-right and right-to-left strategies, to allow one to parse material to the right of the error.

lemma

A set of lexemes with the same stem, the same major part-of-speech, and the same word-sense. E.g. {cat, cats}.

lexeme

Fancy name for a word, including any suffix or prefix. Contrast free and bound morphemes.

lexical functional grammar

A grammatical formalism, not covered in COMP9414.

lexical generation probability

The probability that a particular lexical category (in context or out of context) will give rise to a particular word. For example, suppose, in a system with a very small lexicon, there might be only two nouns, say *cat* and *dog*. Given a corpus of sentences using this lexicon, one could count the number of times that the two words *cat* and*dog* occurred (as a noun), say `ncats` and `ndogs`. Then the lexical generation probability for *cats* as a noun would be `ncats/(ncats+ndogs)`, written symbolically as Pr(*cat* | N).

lexical insertion rule

A rule of a grammar (particularly a context-free grammar) of the form X -> ω, where ω is a single word. In most lexicons, all the lexical insertion rules for a particular word are "collapsed" into a single lexical entry, like ("pig" N V ADJ). ("pig" is familiar as a N, but also occurs as a verb ("Jane pigged herself on pizza") and an adjective, in the phrase "pig iron", for example.

lexical symbol, lexical category

Also called a **pre-terminal** symbol. A kind of non-terminal symbol of a grammar - a non-terminal is a lexical symbol if it can appear in a lexical insertion rule. Examples are N, V, ADJ, PREP, INTERJ, ADV. Non-examples include NP, VP, PP and S (these are non-terminals). The term **lexical category** signifies the collection of all words that belong to a particular lexical symbol, for example, the collection of all Nouns or the collection of all ADJectives.

lexicon

A lexicon is a collection of information about the words of a language about the lexical categories to which they belong. A lexicon is usually structured as a collection of **lexical entries**, like ("pig" N V ADJ). ("pig" is familiar as a N, but also occurs as a verb ("Jane pigged herself on pizza") and an adjective, in the phrase "pig iron", for example. In practice, a lexical entry will include further information about the roles the word plays, such as feature information - for example, whether a verb is transitive, intransitive, bitransitive, etc., what form the verb takes (e.g. present participle, or past tense, etc.)

LFG

= lexical functional grammar

local discourse context

The local discourse context, or just local context includes the syntactic and semantic analysis of the preceding sentence, together with a list of objects mentioned in the sentence that could be antecedents for later pronouns and definite noun phrases. Thus the local context is used for the reference stage of NLP. See also history list.

logical form

Logical forms are expressions in a special language, resembling FOPC (first order predicate calculus) and used to encode the meanings (out of context) of NLP sentences. The logical form language includes:

terms

constants or expressions that describe objects: FIDO1, JACK1

predicates

constants or expressions that describe relations or properties, like BITES1. Each predicate has an associated number of arguments - BITES1 is binary.

propositions

a predicate followed by the appropriate number of arguments: (BITES1 FIDO1 JACK1), (DOG1ÊFIDO1) - Fido is a dog. More complex propositions can be constructed using logical operators (NOT (LOVES1 SUE1 JACK1)), (& (BITES1 FIDO1 JACK1) (DOG1 FIDO1)).

quantifiers

English has some precise quantifier-like words: *some, all, each, every, the, a*as well as vague ones: *most, many, a few*. The logical form language has quantifiers to encode the meanings of each quantifier-like word.

variables

are needed because of the quantifiers, and because while the words in a sentence in many cases give us the types of the objects, states and events being discussed, but it is not until a later stage of processing (reference) that we know to what instances of those types the words refer.

Variables in logical form language, unlike in FOPC, persist beyond the "scope" of the quantifier. E.g. *A man came in. He went to the table.* The first sentence introduces a new object of type MAN. The *He*, in the second sentence refers to this object.

NL quantifiers are typically restricted in the range of objects that the variable ranges over. In *Most dogs bark* the variable in the MOST1 quantifier is restricted to DOG1 objects: (MOST1 d1 : (DOG1 d1) (BARKS1 d1)).

predicate operators

A predicate operator takes a predicate as an argument and produces a new predicate. For example, we can take a predicate like CAT1 (a unary predicate true of a single object of type CAT1) and apply the predicate operator PLUR that converts singular predicates into the corresponding plural predicate (PLUR CAT1), which is true of any set of cats with more than one member.

modal operators

Modal operators are used to represent certain verbs like believe, know, want, that express **attitudes** to other propositions, and for tense, and other purposes. *Sue believes Jack is happy* becomes

(BELIEVE SUE1 (HAPPY JACK1))

With tenses, we use the modal operators PRES, PAST, FUT, as in:

(PRES (SEES1 JOHN1 FIDO1))
(PAST (SEES1 JOHN1 FIDO1))
(FUT (SEES1 JOHN1 FIDO1))

logical operator

The operators **and**, **or**, **not**, => (implies), and <=> (equivalent to). **and** is sometimes written as &. They are used to connect propositions to make larger propositions: e.g.
(IS-BLUE SKY1) **and** (IS-GREEN GRASS1) **or** (CAN-FLY PIG1)

M

Marcus parsing

A parsing technique, not covered in COMP9414.

mass noun

A noun that cannot be counted. *Water* is a mass noun, as is *sand* (if you want to count sand, you refer to *grains*). Contrast count noun.

modal

A modal auxiliary is distinguished syntactically by the fact that it forces the main verb that follows it to take the infinitive form. For example, "can", "do", "will" are modal ("she can eat the pizza", "she does eat pizza", "she will eat pizza") but "be" and "have" are not ("she is eating pizza", "she has eaten pizza").

modal operator

As far as we are concerned in COMP9414, modal operators are a feature of the logical form language used to represent certain **epistemic** verbs like "believe", "know" and other verbs like "want", and the tense operators, which convert an untensed logical form into a tensed one.

Thus if (LIKES1 JACK1 SUE1) is a formula in the logical form language, then we can construct logical forms like (KNOW MARY1 (LIKES1 JACK1 SUE1)) meaning that Mary knows that Jack likes Sue. Similarly for (BELIEVE MARY1 (LIKES1 JACK1 SUE1)) and (WANT MARG1 (OWN MARG1 (?obj : &((PORSCHE1 ?obj) (FIRE-ENGINE-RED ?obj))))) - that's *Marg wants to own a fire-engine red Porsche*.
The **tense** operators include FUT, PRES, and PAST, representing future, present and past. For example, (FUT (LIKES1 JACK1 SUE1)) would represent *Jack will like Sue*.

See also failure of substitutivity.

mood

See also articles on individual moods.

Mood	*Description*	*Example*
indicative	A plain statement	John eats the pizza
imperative	A command	Eat the pizza!
WH-question	A question with a phrasal answer, often starting with a question-word beginning with "wh"	Who is eating the pizza? What is John eating? What is John doing to the pizza?
Y/N-question	A question with yes/no answer	Did John eat the pizza?
subjunctive	An embedded sentence that is **counter-factual** but must be expressed to, e.g. explain a possible consequence.	If *John were to eat more pizza* he would be sick.

morpheme

A unit of language immediately below the word level. See free morpheme and bound morpheme, and morphology.

morphology

The study of the analysis of words into morphemes, and conversely of the synthesis of words from morphemes.

MOST1

A rather vague natural language quantifier, corresponding to the word "most" in English. "Many", "a few", and "several" are other quantifier-type expressions that are similarly problematical in their interpretation.

N

N

symbol used in grammar rules for a noun.

n-gram

A n-gram is an n-tuple of things, but usually of lexical categories. Suppose that we are concerned with *n* lexical categories L1, L2, ..., Ln. The term n-gram is used in statistical NLP in connection with the conditional probability that a word will belong to Ln given that the preceding words were in L1, L2, ..., L[n-1]. This probability is written Pr(Ln | L[n-1]...L2 L1), or more fully Prob(w[i] in Ln | w[i-1] in L[n-1] & ... & w[i-n-1] in L1). See also bigram and trigram, and p. 197 in Allen.

nominal

Word for a noun functioning as an adjective, as with the word "wood" in "wood fire". Longer expressions constructed from nominals are possible. It can be difficult to infer the meaning of the nominal compound (like "wood fire") from the meanings of the individual words.- for instance, while "wood fire" presumably means a fire made with wood, "brain damage" means damage to a brain, rather than damage made with a brain. Another example: "noun modifier" could on the face of it either mean a noun that acts as a modifier (i.e. a nominal as just defined) or a modifier of a noun.

In fact, **noun modifier** is a synonym for nominal.

non-terminal

A non-terminal symbol of a grammar is a symbol that represents a lexical or phrasal category in a language. Examples in English would include N, V, ADJ, ADV (lexical categories) and NP, VP, ADJP, ADVP and S (phrasal categories). See also terminal symbol and context-free grammar.

noun

A noun is a word describing a (real or abstract) object. See also mass noun, count noun, common noun, abstract noun, proper noun, and concrete noun.
Constrast verb, adjective, adverb, preposition, conjunction, and interjection.

Noun is often abbreviated to N.

N is a lexical grammatical category.

noun modifier

= nominal.

noun phrase

Noun Phrase is a phrasal grammatical category. Noun phrase is usually abbreviated to NP. NPs have a noun as their head, together with (optionally) some of the following:
adjectives, nominal modifiers (i.e. other nouns, acting as though they were adjectives), certain

kinds of adverbs that modify the adjectives, as with "very" in "very bright lights", participles functioning as adjectives (as in "hired man" and "firing squad"), cardinals, ordinals, determiners, and quantifiers. There are constraints on the way these ingredients can be put together. Here are some examples of noun phrases: *Ships* (as in *Ships are expensive to build*, *three ships* (cardinal + noun), *all three ships* (quantifier + cardinal + noun), *the ships* (determiner + noun), *enemy ships* (nominal + noun), *large, grey ships* (adjective + adjective + noun), *the first three ships* (determiner + ordinal + cardinal + noun), *my ships* (possessive + noun).

NP

symbol used in grammar rules for a noun phrase.

number (grammatical)

The term grammatical number refers to whether the concept described consists of a single unit (singular number), like "this pen", or to more than one unit (plural number), like "these pens", or "three pens".

In some languages other than English, there may be different distinctions drawn - some languages distinguish between one, two, and many, rather than just one and many as in English.

Nouns in English are mostly marked for number - see plural.

Pronouns and certain determiners may also be marked for number. For example, "this" is singular, but "these" is plural, and "he" is singular, while "they" is plural.

O

object

The object of a sentence is the noun phrase that appears after the verb in a declarative English sentence. For example, in *The cat ate the pizza*, *the pizza* is the object. In *The pizza was eaten by the cat*, there is no object. Object noun phrases can be arbitrarily long and complex. For example, in *He ate a pizza with lots of pepperoni, pineapple, capsicum, mushrooms, anchovies, olives, and vegemite*, the object is *a pizza with lots of pepperoni, pineapple, capsicum, mushrooms, anchovies, olives, and vegemite*. [No, I do not have shares in a pizza company.]

See also bitransitive, transitive, and intransitive.

ordinal

A form of number word that indicates rank rather than value. Thus "one, two, three, four, five, six, seven" are cardinal numbers, whose corresponding ordinal numbers are "first, second, third, fourth, fifth, sixth, seventh".

output probability

= lexical generation probability, but used in the context of a Hidden Markov Model.

P

parse tree

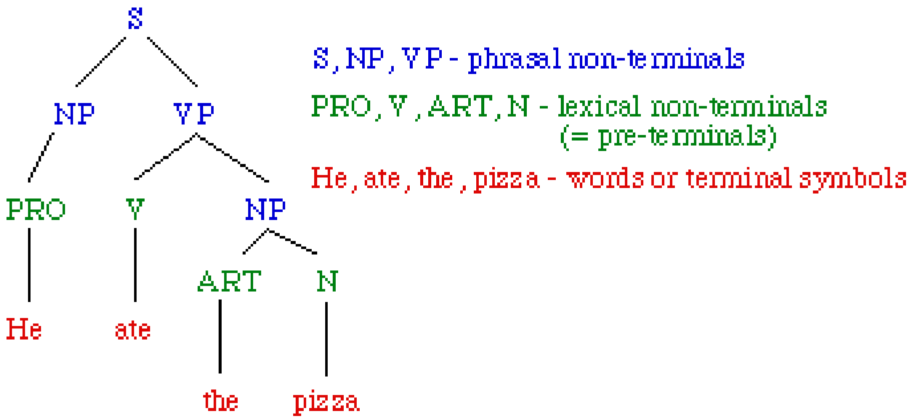
A parse tree is a way of representing the output of a parser, particularly with a context-free grammar. Each phrasal constituent found during parsing becomes a branch node of the parse tree. The words of the sentence become the leaves of the parse tree. As there can be more than one parse for a single sentence, so there can be more than one parse. Example, for the sentence "He ate the pizza", with the respect to the grammar with rules

S -> NP VP, NP -> PRO, NP -> ART N, VP -> V NP,

and lexicon

("ate" V) ("he" PRO) ("pizza" N) ("the" ART)

the parse tree is


S
NP
VP
PRO
V
NP
ART
N
He
ate
the
pizza
S, NP, VP - phrasal non-terminals
PRO, V, ART, N - lexical non-terminals
(= pre-terminals)
He, ate, the, pizza - words or terminal symbols

Note that this graphical representation of the parse tree is unsuitable for further computer processing, so the parse tree is normally represented in some other way internally in NLP systems. For example, in a Prolog-like notation, the tree above could be represented as:

```
s(np(pro("He")),
  vp(v("ate"),
     np(art("the"), n("pizza")))).
```

parser

A parser is an algorithm (or a program that implements that algorithm) that takes a grammar, a lexicon, and a string of words, decides whether the string of words can be derived from the grammar and lexicon (i.e. is a sentence with respect to the grammar and lexicon.

If so, it produces as output some kind of representation of the way (or ways) in which the sentence can be derived from the grammar and lexicon. A common way of doing this is to output (a) parse tree(s).

part-of-speech tagging

The process of going through a corpus of sentences and labelling each word in each sentence with its part of speech. A **tagged corpus** is a corpus that has been so labelled. A **tag** is one of the labels. Large-scale corpora might use *tag-sets* with around 35-40 different tags (for English). See Allen Fig. 7.3 p. 196 for an example of a tag-set.

part of speech

The role, like noun, verb, adjective, adverb, pronoun, preposition, etc. that a word is either playing in a particular sentence (e.g. *like* is acting as a preposition in *I like pizza*) or that it can play in some sentence: e.g. *like* can act as a verb, noun, adjective, adverb, preposition, and conjunction. (It can also act as a "filled pause", as do *um*, *er*, and *uh* - e.g. *>He's, like, a pizza chef in this, like, fast food joint downtown.*

participle

Participles come in two varieties (in English) - **present participles** and **past participles**. (Often abbreviated to PRESPART and PASTPART or something equivalent, like ING and EN). Present participles are variants on verbs; they end in "-ing", as in "setting", "being", "eating", "hiring". Past participles end in "-ed", "-en", or a few other possibilities, as in "set" (past participle the same as the infinitive form of the verb), "been", "eaten", "hired", "flown" (from "fly").

Participles are used in constructing tensed forms of verbs, as in "he is eating", "you are hired", and also as though they were adjectives in phrases like "a flying horse" and "a hired man".

In some cases, present participles have become accepted as nouns representing an instance of the action that the underlying verb describes, as with "meeting".

PRESPART and PASTPART are lexical grammatical categories.

particle

A particle is usually a word that "normally" functions as a preposition, but can also modify the sense of a noun. Not all prepositions can be particles. An example of a word functioning as a particle is "up", in "The mugger beat up his victim". Here "beat up" functions as a unit that determines the action being described. A telltale sign of a particle is that it can often be separated from the verb, as in "The mugger beat the victim up". Sometimes it can be non-trivial for an NLP system to tell whether a word is being used as a particle or as a preposition. For example, in "Eat up your dinner", "up" is definitely a particle, but in "He eats up the street", "up" is a preposition, but it takes real-world knowledge to be sure of this, as the alternative possibility, that the person being referred to is eating the street, is syntactically reasonable (though not pragmatically reasonable, unless perhaps "he" refers to a bug-eyed asphalt-eating alien.)

See also phrasal verb.

passive voice

Both active and passive voice are described in the article on active voice.

PAST

See modal operators - tense and tense.

PASTPART

An abbreviation for Past Participle, particularly in grammar rules.

past perfect

See tense.

patient

= object.

person

Person is a feature of English noun phrases that is principally of significance with pronouns and related forms. The possible values of person are first person signifying the speaker (possibly with his/her companions), second person signifying the person addressed (possibly with his/her companions), and third person signifying anybody else, i.e. not speaker or person addressed or companion of either.

Below is a table of the forms of pronouns, etc. in English, classified by person and syntactic case:

case	*first person*	*second person*	*third person*
nominative	I/we	thou/you/ye	he/she/it/they
accusative	me/us	thee/you/ye	him/her/it/them
possessive adjective	mine/ours	thine/yours	his/hers/its/theirs
possessive	my/our	thy/your	his/her/its/their
reflexive	myself/ourselves	thyself/yourself yourselves	himself/herself itself/themselves

phone

A low-level classification of linguistic sounds - phones are the acoustic patterns that are significant and distinguishable in some human language. Particular languages may group together several phones and regard them as equivalent. For example, in English, the L-sounds at the beginning and end of the word "loyal", termed "light L" and "dark L" by linguists, are distinguished in some languages. Light L and dark L are termed **allophones** of L in English. Similarly, the L and R sounds in English are regarded as equivalent in some other languages.

phoneme

Start by reading about phones. Phonemes are the groups of phones (i.e. allophones) regarded as linguistically equivalent by speakers of a particular language. Thus native English speakers hear light L and dark L as the same sound, namely the phoneme L, unless trained to do otherwise. One or more phonemes make up a morpheme.

phonetics

The study of acoustic signals from a linguistic viewpoint, that is, how acoustic signals are classified into phones.

phonology

The study of phones, and how they are grouped together in particular human languages to form phonemes.

phrasal category

A kind of non-terminal symbol of a grammar - a non-terminal determines a phrasal category if it cannot appear in a lexical insertion rule, that is, a rule of the form X -> ω, where ω is a word. Examples include NP, VP, PP, ADJP, ADVP and S . Non-examples include N, V, ADJ, PREP, INTERJ, ADV (see lexical category).

phrasal verb

A phrasal verb is one whose meaning is completed by the use of a particle. Different particles can give rise to different meanings. The verb "take" participates in a number of phrasal verb constructs - for example:

take in	deceive	He was taken in by the swindler
take in	help, esp. with housing	The homeless refugees were taken in by the Sisters of Mercy
take up	accept	They took up the offer of help.
take off	remove	She took off her hat.

phrase

A unit of language larger than a word but smaller than a sentence. Examples include noun phrases, verb phrases, adjectival phrases, and adverbial phrases.

See also phrasal categories.

pluperfect

See tense.

PLUR

A predicate operator that handle plurals. PLUR transforms a predicate like BOOK1 into a predicate (PLUR BOOK1). If BOOK1 is true of any book, then (PLUR BOOK1) is true of any set of books with more than one member. Thus "the books fell" could be represented by (THE x : ((PLUR BOOK1) x) (PAST (FALL1 x))).

plural noun

A noun in a form that signifies more than one of whatever the base form of the noun refers to. For example, the plural of "pizza" is "pizzas". While most plurals in English are formed by adding "s" or "es", or occasionally doubling the last letter and adding "es", there are a number of exceptions. Some derive from words borrowed from other languages, like "criterion"/"criteria", "minimum"/"minima", "cherub"/"cherubim", and "vertex"/"vertices". Others derive from Old English words that formed plurals in nonstandard ways, like "man"/"men", "mouse"/"mice", and "child"/"children".

possessive

This is a name applied to two English pronoun forms that indicate possession. There are possessive adjectives and possessive pronouns. They are tabulated below:

person & number	*possessive adjective*	*possessive pronoun*
first person singular	my	mine
first person plural	our	ours
second person singular (archaic)	thy	thine
second person (modern)	your	yours
third person singular	his/her/its	his/hers/its
third person plural	their	theirs

PP

Abbreviation for prepositional phrase.

PP attachment

The problem of deciding what component of a sentence should be modified by a prepositional phrase appearing in the sentence. In the classic example "The boy saw the man on the hill with the telescope", "with the telescope" could modify "hill" (so the man is on the "hill with the telescope") or it could modify "saw" (so the boy "saw with the telescope"). The first attachment corresponds to a grammar rule like np -> np pp, while the second corresponds to a grammar rule like vp -> v np pp. Both rules should be present to capture both readings, but this inevitably leads to a multiplicity of parses. The problem of choosing between the parses is normally deferred to the semantic and pragmatic phases of processing.

pragmatics

Pragmatics can be described as the study of meaning in context, to be contrasted with semantics, which covers meaning out of context. For example, if someone says "the door is open", there is a single logical form for this. However, there is much to be done beyond producing the logical form, in really understanding the sentence. To begin with, it is necessary to know which door "the door" refers to. Beyond that, we need to know what the intention of the speaker (say, or the writer) is in making this utterance. It could be a pure statement of fact, it could be an explanation of how the cat got in, or it could be a tacit request to the person addressed to close the door.

It is also possible for a sentence to be well-formed at the lexical, syntactic, and semantic levels, but ill-formed at the pragmatic level because it is inappropriate or inexplicable in context. For example, "Try to hit the person next to you as hard as you can" would be pragmatically ill-formed in almost every conceivable situation in a lecture on natural language processing, except in quotes as an example like this. (It might however, be quite appropriate in some settings at a martial arts lesson.)

pre-terminal

See lexical category.

predicate

This term is used in (at least) three senses:

1. In NLP, equivalent to verb phrase, used in the analysis of a sentence into a subject and predicate.
2. In logic, a predicate is a logical formula involving predicate symbols, variables, terms, quantifiers and logical connectives.
3. In Prolog - see here.

predicate operator

Predicate operators form a part of the logical form language. They transform one predicate into another predicate. For example, the predicate operator PLUR transforms a singular predicate like (DOG x) which is true if x is a dog, into a plural equivalent (PLUR DOG) such that ((PLUR DOG) x) is true if x is a set of more than one dog.

predictive parser

A predictive parser is a parsing algorithm that operates top-down, starting with the start symbol, and *predicting* or guessing which grammar rule to used to rewrite the current sentential form Alternative grammar rules are stacked so that they can be explored (using backtracking) if the current sequences of guesses turns out to be wrong.

On general context-free grammars, a vanilla predictive parser takes exponential parsing time (i.e. it can be very very slow). See also bottom-up parsers.

PREP

symbol used in grammar rules for a preposition.

preposition

A preposition is a part of speech that is used to indicate the role that the noun phrase that follows it plays in the sentence. For example, the preposition "with" often signals that NP that follows is the instrument of the action described in the sentence. For example "She ate the pizza with her fingers". It can also indicate the co-agent, especially if the NP describes something that is animate. For example, "She ate the pizza with her friends". As well as signalling a relationship between a noun phrase and the main verb of the sentence, a preposition can indicate a relationship between a noun phrase and another noun phrase. This is particularly the case with "of", as in "The length of the ruler is 40 cm".

Prepositions are the head items of prepositional phrases.

Preposition is often abbreviated to PREP.

PREP is a lexical grammatical category.

prepositional phrase

Prepositional phrase is a phrasal grammatical category. Prepositional phrase is usually abbreviated to PP. PPs serve to modify a noun phrase or a verb or verb phrase. For example, in "The house on the hill is green", the prepositional phrase "on the hill" modifies "the house", while in "He shot the deer with a rifle", "with a rifle" is a prepositional phrase that modifies "shot ..." (except in the extremely rare case that the deer has a rifle :->).

Prepositional phrases normally consist of a preposition followed by a noun phrase.

The main exception is the **possessive clitic** `'s`, as in "my uncle's car", where the `'s` functions as a preposition ("my uncle's car" = the car *of* my uncle") but follows the noun phrase that the preposition normally precedes.

Occasionally other structures are seen, such as "these errors notwithstanding", an allowable variant on "notwithstanding these errors" ("notwithstanding" is a preposition).

PP is a phrasal grammatical category.

See also PP attachment.

PRES

See modal operators - tense and tense - present.

prescriptive grammar

When we think of grammar, we often think of the rules of good grammar that we may have been taught when younger. In English, these may have included things like "never split an infinitive" i.e. do not put an adverb between the word "to" and the verb, as in "I want to really enjoy this subject." (The origin of this rule is said to be the fact that infinitive-splitting is grammatically impossible in Latin: some early grammarians sought to transfer the rule to English for some twisted reason.) Grammar in this sense is called "prescriptive grammar" and has nothing to do with descriptive grammar", which is what we are concerned with in NLP.

PRESPART

An abbreviation for Present Participle, particularly in grammar rules.

production

= grammar rule - see Chomsky hierarchy and context free grammar.

progressive

See aspect.

proper noun

A proper noun is a noun that names an individual, such as *Amelia Earhart*, rather than a type, for example *woman*, or *philosophy*. Proper nouns are often compound. *Amelia* and *Earhart* would each rank as proper nouns in their own right.
Contrast common noun.

proposition

A proposition is a statement of which it is possible to decide whether it is true or false. There are atomic propositions, like "Mary likes pizza", and compound ones involving

Q

qualifier

Umbrella term for adjectives and nominals or noun modifiers.

quantifier

1. (in semantic interpretation) - objects in the logical form language that correspond to the various words and groups of words that act in language in the way that quantifiers do in formal logic systems. Obvious examples of such words in English include *all each every*

some most many several. Less obvious examples include *the*, which is similar in effect to "there exists a unique" - thus when we refer to, say, "the green box", we are indicating that there exists, in the current discourse context, a unique green box that is being referred to. This phrase would be represented in the logical form language by an expression like (THE **b1** : &((BOX1 **b1**) (GREEN1 **b1**))).

NL quantifiers are typically restricted in the range of objects that the variable ranges over. In *Most dogs bark* the variable in the MOST1 quantifier is restricted to DOG1 objects: (MOST1 d1 : (DOG1 d1) (BARKS1 d1))

2. In logic, this term refers to the logical operators "forall" and "exists".
3. This term is also sometimes used for quantifying determiners.

quantifying determiner

A quantifying determiner, in English, is one of a fairly small class of words like "all", "both", "some", "most", "few", "more" that behave in a similar way to quantifiers in logic.

See also determiners.

R

reference

Reference, in NLP, is the problem/methods of deciding to what real-world objects various natural language expressions *refer*. Described in Chapter 14 of Allen. See also anaphor, cataphor, co-refer, discourse entity, history list, and local discourse context

referential ambiguity

A type of ambiguity where what is uncertain is what is being referred to by a particular natural language expression. For example, in *John hit Paul. He was angry with him.* it is not entirely clear to whom the pronouns "he" and "him" refer - it could be that the second sentence explains the first, in which case the "he" is John, or it could be that the second sentence gives a consequence of the first, in which case the "he" is Paul.

regular

1. Regular grammar = right-linear grammar - see Chomsky hierarchy.
2. A regular verb, noun, etc. is one that inflects in a regular way. "save" is a regular verb and "house" is a regular noun. On the other hand, "break" (with past tense "broke" and past participle "broken") is an **irregular** verb, and "mouse" (with plural form "mice") is an irregular noun.

relative clause

Relative clauses involve sentence forms used as modifiers in noun phrases. These clauses are often introduced by relative pronouns such as *who*, *which* and *that*. For example, "The man *who gave Barry the money*". See Allen p.34

rewriting process

The rewriting process is what is used in derivation to get from one sentential form to the next.

The process is as follows with context free grammars: pick a non-terminal X in the current string (or sentential form) and a grammar rule whose left-hand side is that non-terminal X. Replace X in the current string by the right-hand side of the grammar rule, to obtain a new current string. This definition also works for regular grammars. A single step in the rewriting process is called a **direct derivation**. For an example, see derivation.

The process is similar with context-sensitive grammarsand unrestricted grammars, except that instead of picking a non-terminal X in the current string, we find a *substring* of the current string that matches the left-hand side of some context-sensitive or unrestricted grammar rule, and replace it with the right-hand side of that grammar rule.

right-linear grammar

See Chomsky hierarchy.

right-to-left parsing

See the article on left-to-right parsing.

robustness

A parser or other NLP algorithm is robust if it can recover from or otherwise handle ill-formed or otherwise deviant natural language expressions. For example, we would not want a syntactic parser to give up as soon as it encounters a word that is not in its lexicon - preferably it should try to infer the lexical category and continue parsing (as humans do when they first encounter a new word).

S

S

symbol used in grammar rules for a sentence.

second person

One of the choices for the person feature. A sentence is "in the second person" if the subject of the sentence is the person(s) addressed as in "you like pizza" and the archaic "Ye like pizza" and "Thou likest pizza".

"you", "thou" and "ye" are second-person pronouns, as is "thee". Other words with the second-person feature include "yours", "thine", "your", "thy", "yourself", "yourselves", and "thyself".

semantic grammar

A variant on a context free grammar, in which the non-terminals correspond to semantic rather than syntactic concepts. A system of this type encodes semantic knowledge about the types of sentences likely to appear in the input in the grammar rules. For example, a system to handle text about ships and ports might encode in its grammar rules information about the subject of a sentence about "docking" must be a ship:

docksentence -> shipnp dockvp

The problem with semantic grammar is that for coverage of a significant portion of a language, a huge number of rules would be required, and a massive analysis of the meanings that those rules could encode would be included in their development.

semantics

Semantics is the study of meaning (as opposed to form/syntax, for example) in language. Normally semantics is restricted to "meaning out of context" - that is, too meaning so far as it can be determined without taking context into account. See Allen page 10 for the different levels of language analysis, and chapters 8-12 for detailed treatment (we covered parts of chapters 8 and 9 only in COMP9414).

See also logical form.

sentence

Sentence is the level of language above phrase. Above sentence are pragmatic-level structures that interconnect sentences into paragraphs, etc., using concepts such as cohesion. They are beyond the scope of this subject.

Sentences are sometimes classified into simple, compound, complex, and compound-complex, according to the absence or presence of conjunctions, relative clauses (phrases with verb groups that are introduced by "which", "that", etc.) or both. They may also be analysed into subject and predicate.

Sentence is often abbreviated to S (see also start symbol).

S is a phrasal grammatical category.

sentential form

See article on derivation.

shift-reduce parser

A type of parsing algorithm, not discussed in COMP9414.

simple future

See tense.

simple past

See tense.

simple present

See tense.

singular noun

A noun in a form that signifies one of whatever type of object the noun refers to. For example, *dog* is singular, whereas *dogs* is plural.

Skolem functions and constants

Universally quantified variables can be handled (and are handled in Prolog) simply by assuming that any variable is universally quantified. Existentially quantified variables must thus be removed in some way. This is handled by a technique called **skolemization**.

In its simplest form, skolemization replaces the variable with a new constant, called a **Skolem constant**. For example, the formula:

```
exists(y, forall(x, loves(x, y))
```

would be encoded as an expression such as

```
loves(X, sk1),
```

where `sk1` is a new constant that stands for the object that is asserted to exist, i.e. the person (or whatever) that is loved by every `X`.

Quantifier scoping dependencies are shown using new *functions* called **Skolem functions**. For example, the formula:

```
forall(y, exists(x, loves(x,y))
```

would be encoded as an expression such as

```
loves(sk2(Y), Y),
```

where sk2 is a new function that produces a potentially new object for each value of Y.

skolemization

See Skolem functions

speech act

A term from the pragmatic end of language use: when we say (or write) something, each utterance has a purpose and, if effective, accomplishes an act of some type. Examples of different types of speech act include:

ask request inform deny congratulate confirm promise

. Not covered in COMP9414, except to point out that it is ultimately vital to understanding language. For some discussion, see Allen p. 542 ff., and compare surface speech act.

start symbol

The start symbol of a grammar is a another name for the "distinguished non-terminal" of the grammar. Details at context-free grammar. The start symbol of most NLP grammars is S (for sentence).

statistical NLP

A group of techniques relying on mathematical statistics and used in NLP to, for example, find the most likely lexical categories or parses for a sentence. Often the techniques are based on frequency information collected by analysing very large corpora of sentences in a single language, to find out, for example, how many times a particular word (*dog*, perhaps) has been used with a particular part of speech. The sentences in the corpus have usually been tagged in some way (sometimes manually) so that the information about the part of speech each time each word is used, is known. Sometimes the sentences are hand-parsed, as well (a **treebank**).

See chapter 7 in Allen, and also Bayes' rule, bigram, trigram, n-gram, conditional probability, statistical independence, Hidden Markov Model, and Viterbi algorithm

stem

= bound morpheme.

string

A "string over an alphabet A" means a sequence of symbols taken from the alphabet A, where by alphabet we mean just a set of symbols that we are using in a similar way to the way that we use, say, the Latin alphabet to make up words. Thus a word (in English) is a string over the alphabet {a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z} (plus arguably a few other items like hyphen and apostrophe). A construct like "ART ADJ NOUN" is a string over an alphabet that includes the symbols ART, ADJ, and NOUN. Similarly "NP of NP" is a string (over some alphabet that includes the symbols "NP" and "of") that has two non-terminal symbols and one terminal symbol (namely "of").

structural ambiguity

A form of ambiguity in which what is in doubt is the syntactic structure of the sentence or fragment of language in question. An example of pure structural ambiguity is "old men and women" which is ambiguous in that it is not clear whether the adjective old applies to the women or just to the men. Frequently structural ambiguity occurs in conjunction with word-sense ambiguity, as in "the red eyes water" which could signify "the communist looks at water":
(S (NP (ART the) (N red)) (VP (V eyes) (NP (N water))))
or alternatively "the reddened eyes drip tear fluid"
(S (NP (ART the) (ADJ red) (N eyes)) (VP (V water)))

See also referential ambiguity.

SUBCAT

The name for the feature used to record the subcategorization of a verb or adjective.

subcategorization

Verbs and some adjectives admit complement structures. They are said to *subcategorize* the structures that they can be followed by. For example, some verbs can be followed by two noun phrases (like *Jack gave Mary food*), some by at most one (like *Jack kicked the dog*), and some by none (like *Jack laughed*). We would record this by saying that the verbs have SUBCAT _np_np, or SUBCAT _np, or SUBCAT _none. Further examples are shown below (taken from Figures 4.2 and 4.4 in Allen:

Value	**Example Verb**	**Example of Use**
_none	laugh	Jack laughed
_np	find	Jack found a key
_np_np	give	Jack gave Sue the paper
_vp:inf	want	Jack wants to fly
_np_vp:inf	tell	Jack told the man to go
_vp:ing	keep	Jack keeps hoping for the best
_np_vp:ing	catch	Jack caught Sam looking at his desk
_np_vp:base	watch	Jack watched Sam look at his desk
_np_pp:to	give	Jack gave the key to the man

_pp:loc	be	Jack is at the store
_np_pp:loc	put	Jack put the box in the corner
_pp_mot	go	Jack went to the store
_np_pp:mot	take	Jack took the hat to the party
_adjp	be, seem	Jack is happy
_np_adjp	keep	Jack kept the dinner hot
_s:that	believe	Jack believed that sharks wear wedding rings
_s:for	hope	Jack hoped for Mary to eat the pizza.

Notice that several verbs (*give be keep*) among the examples have more than one SUBCAT. This is not unusual. As an example of subcategorization by adjectives, notice that "Freddo was happy to be a frog" is OK, so *happy* subcategorizes _vp:inf, but "Freddo was green to ..." cannot be completed in any way, so *green* does not subcategorize _vp:inf.

subject

The subject of a sentence is the noun phrase that appears before the verb in a declarative English sentence. For example, in *The cat sat on the mat*, *The cat* is the subject. In *The mat was sat on*, *The mat* is the subject. Subject noun phrases can be arbitrarily long and complex, and may not look like "typical" noun phrases. For example, in *Surfing the net caused him to fail his course*, the subject is *surfing the net*. [Please excuse the subliminal message.]

subjunctive

A subjunctive sentence is an embedded sentence that expresses a proposition that is counterfactual (not true), such as "If *John were to eat more pizza*, he would make himself sick", as opposed to a Y/N-question, a WH-question, a command, or a statement. As can be seen from the example, the subjunctive form of a verb resembles the past form in modern English, even though it frequently refers to a possible future action or state (as in our example). The past forms of some modals (e.g. "should", "would" which were originally the past forms of verbs "shall" and "will") are used for little else in modern English.

See also Y/N-question, WH-question, imperative, indicative, and mood.

substitutivity

See failure of substitutivity.

surface speech act

This term refers to analysing the type of sentence into standard syntactic categories - assertion, command, and the two kinds of question: yes/no-questions and wh-questions. See Allen p. 250.

To be contrasted with (pragmatic) speech acts.

syllepsis

Not a part of COMP9414 Artificial Intelligence.
A figure of speech in which a single word appears to be in the same relationship to two others, but must be understood in a different sense with each of the two other words (the "pair"). See also zeugma. *I'm leaving for greener pastures and ten days*.

One of a couple of dozen little-used terms for figures of speech.

syntax

Syntax means the rules of language that primarily concern the *form* of phrases and sentences, as distinct from the substructure of words (see morphology) or the meaning of phrases and sentences in or out of context (see pragmatics and semantics).

systemic grammar

An alternative approach to linguistic grammar, driven from the functional (rather than the structural) end of language. Not covered in COMP9414. See p.95 (Box 4.3) in Allen.

T

tag

1. See part of speech tagging.

2. As **TAG**, it is also an acronym for Tree-Adjoining Grammar. TAGs were not mentioned elsewhere in COMP9414, but are mentioned here just in case you run into them in a book somewhere.

tagged corpus

See part of speech tagging.

tense

The tense of a verb or a sentence relates to the time when the action or state described by the verb or sentence occurred/occurs/will occur. The main contrast with tense is between past, present and future. Most verbs in English indicate the past/present distinction by inflection (a few, like "set", are invariant in this respect). Thus "break" and "breaks" are the present tense forms of the verb "break", and "broke" is the past tense form. The future tense is constructed, in English, by using the auxiliaries "shall" and "will" with the verb - "it will break", for example. Here is a list of the six major tense forms:

Form	*Example*	*Meaning*

present	He drives a Ford	The "drive" action occurs in the present, though it suggests that this is habitual - it may have occurred in the past and may continue in the future.
simple past	He drove a Ford	The action occurred at some time in the past.
simple future	He will drive a Ford	The action will occur at some time in the future.
past perfect or pluperfect	He had driven a Ford	At some time in the past, it was true to say "He drove a Ford".
future perfect	He will have driven a Ford	At some point in the future, it will be true to say "He drove a Ford".

See also participle.

Contrast tense, mood and aspect.

term

1. Used in the logical form language to describe constants and expressions that describe objects.
2. Used in FOPC to refer to a class of objects that may be defined, recursively, as follows:
   - a constant is a term;
   - a variable is a term;
   - a function f applied to a suitable number of terms t1, t2, ..., tn is a term: f(t1, t2, ..., tn).
3. Used to refer to certain types of Prolog language constructs

terminal symbol

A terminal symbol of a grammar is a symbol that can appear in a sentence of the grammar. In effect, a terminal symbol is a word of the language described by the grammar.

See also non-terminal symbol and context-free grammar.

THE

The word *the* gives rise to an important NL quantifier written as THE in Allen or as `the` or `the1` in the Prolog notation used in the assignment. Thus *the dog barks* has logical form:

(THE **d1** : (DOG1 **d1**) (BARKS1 **d1**)).

Here **d1** is the variable over which THE quantifies.
This is also written as (BARKS1 <THE **d1** DOG1>) in Allen's notation, and as `barks1(the(d1, dog1))` in the Prolog notation.

thematic role

= semantic case.

theme

Term used for the noun phrase that follows the verb in an active voice, indicative sentence in English. Also referred to as the object or sometimes the victim.

third person

One of the choices for the person feature. A sentence is "in the third person" if the subject of the sentence is neither the speaker nor the person(s) addressed, as in "she likes pizza" and "they like pizza".

"she", "he", "it", and "they" are third-person pronouns, as are "her", "him", and "them". Other words with the third-person feature include "hers", "his", "theirs", "their", "herself", "himself", "itself" and "themselves".

top-down parser

A parser that starts by hypothesizing an S (see start symbol) and proceeds to refine its hypothesis by expanding S using a grammar rule which has S as its left-hand side (see rewriting process, and successively refining the non-terminals so produced, and so on until there are no non-terminals left (only terminals).

See also predictive parser.

transitive

A verb that can take a single syntactic object, like *eat*, as in "He ate the pizza". Sometimes transitive verbs appear without their object, as in "He ate slowly" - the distinguishing characteristic of transitive verbs is that they *can* take an object (unlike intransitive verbs, and they cannot take two objects, as bitransitive verbs can. See also subcategorization.

trigram

A trigram is a triple of things, but usually a triple of lexical categories. Suppose that we are concerned with three lexical categories L1, L2 and L3. The term trigram is used in statistical NLP in connection with the conditional probability that a word will belong to L3 given that the preceding words were in L1 and L2. This probability is written Pr(L3 | L2 L1), or more fully Prob(w[i] in L3 | w[i-1] in L2 & w[i-2] in L1). For example, in the phrase "The green flies", given that *The* is tagged with ART, and *green* with ADJ, we would be concerned with the conditional probabilities Pr(N | ADJ ART) and Pr(V | ADJ ART) given that *flies* can be tagged with N and V. See also bigram and n-gram.

U

unrestricted grammar

See Chomsky hierarchy.

V

V

symbol used in grammar rules for a verb.

VAR feature

A feature used as part of the logical form generation system described in lectures. It is used to provide a "discourse variable" that corresponds to the constituent it belongs to. It is useful in handling certain types of modifiers - for example, if we have a ball **b1** and it turns out to be red, then we can assert (in the logical form for "red ball") that the object is both red and a ball, by including &((BALL1 **b1**) (RED1 **b1**)). Grammar rules, once augmented to handle logical forms, usually give explicit instructions on how to incorporate which VAR feature. For example, the rule for intransitive VPs:

(VP VAR ?v SEM (lambda a2 (?semv ?v a2))) ->
(V SUBCAT _none VAR ?v SEM ?semv)

indicates that the VPs VAR feature is derived from that of its V subconstituent and shows how the feature (?v) is also incorporated into the SEM of the VP.

VAR features are described in Allen on page 268 ff.

verb

A word describing an action or state or attitude. Examples of each of these would be "ate" in "Jane ate the pizza", "is" in "Jane is happy", and "believed" in "Jane believed Paul at the pizza".

Verbs are one of the major sources of inflection in English, with most verbs having five distinct forms (like "eat" with "eat"/"eats"/"eating"/"ate"/"eaten". The verb "be" is the most irregular, with forms "be", "am", "is", "are", "being", "was", "were", "been", plus some archaic forms, like "art" as in "thou art".

Verb is often abbreviated to V.

V is a lexical grammatical category.

verb complement

The structure which follows the verb or verb group in a sentence.

Example	*Type of complement*

Jane laughed.	empty
Jane ate the pizza.	NP
Jane believed Paul ate the pizza.	S
Jane wanted to eat the pizza.	to+VP
Jane gave Paul the pizza.	NP+NP
Jane was happy to eat the pizza.	ADJP

See also verb phrase.

verb group

This term is used for a sequence of words headed by a verb together with auxiliaries, and possibly adverbs and the **negative particle** "not".

For example, in "Jane may not have eaten all the pizza", the verb group is "may not have eaten".

verb phrase

Verb Phrase is a phrasal grammatical category. Verb phrase is usually abbreviated to VP. A verb phrase normally consists of a verb or verb group and a complement, together possibly with adverbial modifiers and PP modifiers. The simplest complements are noun phrases, but sentential complements and similar structures are also possible.

See also predicate.

VFORM

A feature of verbs that signifies what form of the verb is present - particularly useful with verbs are irregular in some of their forms, or where a particular form of the verb is required by a particular syntactic rule (for example, modal auxiliaries force the infinitive form of the verb - VFORM inf).

VFORM	Example	Comment
base	break, be, set, decide	base form
pres	break, breaks, am, is, are, set, sets, decide, decides	simple present tense
past	broke, was, were, set, decided	simple past tense
fin	-	finite = tensed = pres or past
ing	breaking, being, setting, deciding	present participle
pastprt	broken, been, set, decided	past participle
inf	-	used for infinitive forms with *to*

victim

= object.

Viterbi algorithm

The Viterbi algorithm is an algorithm applicable in a range of situations that allows a space that apparently has an exponential number of points in it to be searched in polynomial time.

The Viterbi algorithm was not actually described in detail in COMP9414, but was referred to in the section on statistical NLP in connection with a method for finding the most likely sequence of tags for a sequence of words. Reference: Allen p. 201 ff., especially from p. 202.

VP

symbol used in grammar rules for a verb phrase.

W

wh-question

A WH-question sentence is one that expresses a question whose answer is not merely yes or no, as opposed to a Y/N-question, a command, or a statement. See also Y/N-question, imperative, indicative, subjunctive, and mood.

word

Words are units of language. They are built of morphemes and are used to build phrases (which are in turn used to build sentences.

- See also lexeme
- See alsoterminal symbol

word-sense

One of several possible meanings for a word, particularly one of several with the same part of speech. For example, *dog* as a noun has at least the following senses: canine animal, a type of fastening, a low person, a constellation - *the dog barked, dog the hatches, You filthy dog! Canis major is also called the great dog.*

word-sense ambiguity

A kind of ambiguity where what is in doubt is what sense of a word is intended. One classic example is in the sentence "John shot some bucks". Here there are (at least) two readings - one corresponding to interpreting "bucks" as meaning male deer, and "shot" meaning to kill, wound or damage with a projectile weapon (gun or arrow), and the other corresponding to interpreting "shot" as meaning "waste", and "bucks" as meaning dollars. Other readings (such as damaging some dollars) are possible but semantically implausible. Notice that all readings mentioned have the same syntactic structure, as in each case, "shot" is a verb and "bucks" is a noun.

See also structural ambiguity and referential ambiguity.

X

Y

y/n question

A Y/N-question sentence is one that expresses a question whose answer is either yes or no, as opposed to a WH-question, a command, or a statement. See also WH-question, imperative, indicative, subjunctive, and mood.

Z

zeugma

Not a part of COMP9414 Artificial Intelligence, but it allows us to avoid having an empty list of Z-concepts in the NLP Dictionary. `:-)`

A zeugma is a syllepsis in which the single word fails to give meaning to one of its pair. *She greeted him with arms and expectations wide*.

One of a couple of dozen little-used terms for figures of speech.

CSE Programs

- BE Computer Engineering
- BE Software Engineering
- BSc Computer Science
- BE Bioinformatics

Proposed BE(Software Engineering) MEngSc

- BE(Software Engineering) MEngSc

Proposed BE/MCom structures

- BE(Computer Engineering) MCom - approved
- BE(Software Engineering) MCom

Joint program structures (if known)

- BE(CE) MBiomedE

Bill Wilson
School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, AUSTRALIA
Email: billw at cse.unsw.edu.au
Phone: +61 2 9385 3986
Fax: +61 2 9385 5995

School of Computer Science and Engineering Grievance Handling Home Page

What is a Grievance?

A **grievance** is defined as a "cause for complaint, especially of unjust treatment". Common sense should be used before seeing a grievance officer. The fact of failing a course, exam, quiz, or assignment is **not** by itself a grievance. There has to be an element of unfairness relating to the particular failure.

Who Should You See?

For a course-related grievance, the possibilities include: your tutor or demonstrator, your lecturer, your program director, or a grievance officer. For a harassment-related grievance, the people mentioned above may be able to help if you are a student, or, if you are a CSE employee, you can approach your supervisor or supervisor's supervisor (if the supervisor is harassing you), or a grievance officer.

You should also consider whether you should instead be contacting:

- University Security on extension 56666, or phone 9385 6666, e.g. because you are facing an immediate threat to your safety;
- the University Counselling Service e.g. because the problem you face is not specific to a particular course. Counselling may be able to help if you are failing courses despite working hard, or because you cannot get motivated to work, or cannot concentrate, or feel depressed, or ...
  One CSE grievance officer was recently approached by a student who was having trouble concentrating because a person in the flat next door to him was playing loud music late at night. Problems like that are real enough, but CSE can't help you with them.
- the Equity and Diversity Unit, e.g. because you have a disability and your lecturer will not take what you believe are adequate steps to allow for your disability.

Who is the School Grievance Officer?

If you have a grievance that has to do with the School of Computer Science and Engineering, you can appeal to a School Grievance Officer, currently

Who	e-mail	Remark
Nadine Marcus	□	
Bill Wilson	□	

Putting it in Writing

If the grievance has to do with a course that you are enrolled in, then you should in the first instance **either** discuss the problem with the lecturer or your tutor **or** better, express the problem in writing and send the written description to your lecturer. Reasons that a written description of the problem might be better are:

- You are more likely to remember and put down in writing all the relevant facts - in an oral discussion of the problem you may not make your case in the best way;
- In an oral discussion, you lecturer may feel under pressure to make a decision in your favour, and in an effort to make a considered decision may put you off - you will find this frustrating;
- Because you feel strongly about your case, you may say something that you later regret. Your lecturer will, of course, try not to be influenced by any inappropriate behaviour, but lecturers are only human.

On the other hand, if the matter is a minor one - e.g. a mistake in adding up marks on a test paper, then usually a calm personal approach to the tutor or lecturer is sensible, at least as a first attempt to resolve the problem.

Should you see your Program Director?

In some cases, it might be appropriate to talk to your Program Coordinator next (see table below).

If you contact a Grievance Officer or your Program Coordinator in writing, be sure to include your full name and student number. This is particularly important if you are sending a paper letter, or an e-mail from a non-CSE account. If you are complaining about a particular person, we will of course be contacting that person to get their version of events. If you do not wish for your identity to be revealed to the person about whom you are complaining, please indicate this.

Program	*Coordinator*
BE Computer Engineering	Sri Parameswaran and Ashesh Mahidadia
BE Software Engineering	Ken Robinson

BE Bioinformatics	Bruno Gaeta
BSc Computer Science	Tim Lambert and Achim Hoffmann
Undergraduate Thesis	Albert Nymeyer or your program director
MEngSc	Eric Martin
MIT & GradCertAdvComp	Eric Martin
MInfSc & GradDip(InfSc) (all plans)	Eric Martin
MCompIT & GradDipCompIT & GradCertComputing	Eric Martin
MCompSc & GradDip(CompSc)	Eric Martin
MSc, ME, PhD	Albert Nymeyer

Applying for a Re-Mark

If your grievance is that you don't feel you were awarded a fair mark for a course, be aware that you can apply to the University, through the Student Centre in the Chancellery, to either have the mark checked (make sure all assignment marks were recorded, all parts of exam marked, and final grade calculated correctly), or to have the mark checked and the whole of your exam re-marked. [Following a re-mark, your final mark could go either up or down, or remain unchanged.]

If the problem cannot be resolved by other means, you have the option of appealing to a School Grievance Officer.

If your grievance should happen to relate to both School Grievance Officers, then you could contact the Head of School, or your Faculty's Grievance Officer.

How to Approach a Grievance Officer

In most cases, you should begin by sending an e-mail to the the grievance officer *describing the problem in reasonable detail* and requesting an interview. Please also indicate what steps you have already taken to resolve the problem. This will allow the grievance officer to do some groundwork on your case - sometimes grievances can be resolved immediately in this way.

Abuse of the Grievance Procedure

A tiny proportion of students attempt to use the University's Grievance procedures for a range of improper purposes, including unfairly blackening the reputation of another person. UNSW has penalties for this practice.

Anonymous letters

If you send to a Grievance Officer an unsigned letter (or something similar, like an e-mail from a non-University e-mail address which we cannot confirm comes from a particular student) then we will of course read the contents. However, it is unlikely that the Grievance Officer will be able to do anything about the complaint in the letter. To be able to act, we have to know who is complaining, we may have to check that they are enrolled in a CSE course/program if that is what you are complaining about, and if it is not something quickly checked and fixed, we may need to interview you about the problem.

If you so request, the Grievance Officer will of course keep your identity confidential to the full extent possible, and will not reveal your identity to the person about whom you are complaining without your permission. However, in many cases, to get anywhere, we would have to tell the person about whom you are complaining what the complaint is about, so that they can (a) give their viewpoint; and hopefully (b) fix the problem.

Appeals

If you are unsatisfied with the decision of the grievance officer whom you approach, there is a chain of appeal possibilities, including the Head of School, Faculty Grievance Officer, and Registrar. Sometimes there is more than one School Grievance Officer - in this case, if you take a case to one School grievance officer, you may not seek to have your complaint considered by another School grievance officer. [Your case might, however, be referred by one grievance officer to another, e.g. because the other grievance officer has special expertise, or because the first grievance officer has too much work at the time.]

Last updated:

UNSW's CRICOS Provider Number: 00098G

Casual Academic Employment in the School of Computer Science & Engineering

1. How do I apply for work as a casual tutor or demonstrator?
2. Getting hired
3. Getting inducted
4. Getting paid
5. What if I can't make it to my tute or lab?
6. How come lab demonstrators get paid $24 for a 1-hour lab while tutors get $72 for a 1-hour tute?
7. How do I claim for extra work beyond what's in my contract if I am asked to do some - e.g. extra consultations at peak times.

If your casual tutoring/demonstrating question isn't answered here, please tell me.

School of Computer Science & Engineering
The University of New South Wales
Sydney 2052, AUSTRALIA
Email: billw at cse.unsw.edu.au
Phone: +61 2 9385 6876
Fax: +61 2 9385 4071

Last modified: 20 April 2001

REVISED UNSW SPECIAL CONSIDERATION PROCEDURES

(Approved at the Academic Board, 7 August 2001, (Item 8.1.1(a), RESOLVED AB01/52)

Items in red are annotations added by Bill Wilson and do not form part of the Academic Board resolution.

Review of Special Consideration

A Working Party of Chris Daly, Ian Sharpe, Helen Swarbrick and Judith Tonkin was established at a meeting of Presiding Members of Faculty to produce recommendations on the reform of the special consideration policy and procedures. Diane Dwyer, Helen Milfull, Brett O'Halloran, Grant Walter and Geoff Whale were invited to join the Working Party.

The policy and procedures for students with a disability or longterm health problem were not part of this review and will continue to be the responsibility of the Equity Officer (Disabilities).

The committee considered the following information:

- The number of special consideration applications received at UNSW in the last three sessions.
- Current UNSW policy and procedures.
- Summaries of the equivalent policy at the Universities of Melbourne, Queensland, Sydney and Wollongong.

It was agreed that the review was needed because

- The number of applications was increasing.
- Some students submitted special consideration for almost each course each session.
- Many applications for special consideration had no relevance to the student's performance
- NSS had different capabilities. Geoff Whale explained that NSS would not be able to carry comments over to assessment schedules but would be able to hold comments which could alert course and program authorities that there was information about a student which needed to be considered.

It was agreed that the review should aim to produce the following outcomes:

- To reduce if not eliminate applications, particularly medical certificates, for trivial conditions submitted as insurance.
- To ensure that serious cases of illness or misadventure were given careful consideration.
- To achieve equity and consistency in the assessment of students.
- To provide more information to course authorities to promote better decision-making and earlier assessment of applications.
- To eliminate the possibility of students playing faculties and departments off against each other.

The following procedure was discussed:

1. Student to take original documentation (eg medical certificate) together with the Special Consideration application form to NewSouth Q where staff would ensure that it complied with the University's guidelines (see below), stamp the original, take a copy (to be kept for 12 months) and return the stamped original to the student.
2. NewSouth Q would enter a summary of the application into a database, including dates, severity, brief description. This summary would be distributed to the course authorities of the courses mentioned in the application and attached to the student's record on NSS to alert program and course authorities that there was information potentially relevant to assessment decisions.

Applications would be accepted only in the following circumstances:

1. Where academic work had been hampered to a substantial degree by illness or other cause. Except in unusual circumstances a problem involving only three consecutive days or a total of five days within the teaching period of a semester would not be considered sufficient grounds for an application.
2. The circumstances would have to be unexpected and beyond the student's control. Students would be told they are expected to give priority to their University study commitments and any absence must clearly be for circumstances beyond their control. Work commitments would not normally be considered a justification. In cases where students are prevented from completing assessment, discontinuation without failure may be a more appropriate outcome.
3. An absence from an examination would have to be supported by a medical certificate or other document which clearly indicated that the student was unable to be present.
4. A student absent from an examination or who attended an examination and wanted to request special consideration would normally be required to provide a medical certificate dated the same day as the examination.
5. An application for special consideration would have to be provided within three working days of the assessment to which it referred. In exceptional circumstances an application may be accepted outside the three-day limit.

What is satisfactory documentation?

Documentation should be accompanied by the UNSW Request for Special Consideration form. Medical certificates and other documentation submitted by students in support of applications for consideration should comply with the following conditions, as relevant:

Medical certificates

- The certificate is signed by the student’s own medical practitioner or a practitioner from the University Health Service. The practitioner must have seen the student during the illness or immediately afterwards, when it was first possible to seek help.
- Certificates signed by family members are not acceptable.
- The certificate should be on the UNSW proforma which can be found at *watch this space* Most doctors in the UNSW area should have supplies of this form. After the UNSW Web "Student Gateway" is redesigned - by end of October - the form will also be downloadable from the web.

- Within the limits of confidentiality, the certificate should describe the nature and seriousness of the student’s problem, so that an assessment of the possible effects of the illness or accident on performance can be made.
- The certificate indicates the degree of incapacity of the student (serious, moderate, mild) and its duration or probable duration.

Other documentation

- Documentation will depend on the nature of the circumstances but it must support the student’s account of the effect on his/her performance.

- Documentation relating to a non-medical absence should indicate the importance of the event and why it prevented the student from attending for assessment.

Only documentation which meets the requirements listed above will be accepted. No consideration will be given when the condition or event is unrelated to performance in assessment or is considered not to be serious.

The course authority is responsible for considering the special consideration application. The Assessment Review group is responsible for ensuring that all relevant information has been taken into account in deciding a final mark for a student in a course. One or more of the following may be taken into account.

1. The student’s performance in other items of assessment in the course.
2. The severity of the event.
3. Academic standing in other courses and in the program.
4. History of previous applications for special consideration.

If an application for illness or misadventure is accepted, the following action may ensue

1. No action.
2. Additional assessment or a supplementary examination. Additional assessment may take a different form from the original assessment. If a student is granted additional assessment, the original assessment may be ignored at the discretion of the course authority. Consequently, a revised mark based on additional assessment may be greater or less than the original mark.
3. Marks obtained for completed assessment tasks aggregated or averaged to achieve a percentage.
4. Deadline for assessment extended.
5. Discontinuation from the course. This is unlikely to occur after an examination or final assessment has taken place.

Judith Tonkin
30 July 2001
(amended)

Plagiarism Detection Support

If run from a class account `/home/teachadmin/bin/moss_assignment` and `/home/teachadmin/bin/yap_assignment` should do plagiarism checking using moss or yap on the specified assignment, e.g.:

```
/home/teachadmin/bin/moss_assignment ass1
```

It unpacks the submissions and guesses the assignment language. It should work OK for simpler assignments.

For more complex assignment, you may need to supply extra arguments or do some pre-processing and invoke moss/yap directly.

No GUI, but this code would useful in building one (but I'm definitely not building a GUI).

I've shifted moss & yap into `/home/teachadmin/bin` too.

Andrew

Last modified: 12 Feb 2003

Assessment Goof-Ups

A guide for academic staff in the School of Computer Science & Engineering, UNSW

- **The Assessment Plan Must Not Be Changed after Week 1**

Who Says?	UNSW Assessment Rules § 7.4 part 5
Why Not?	The Course Outline, including the assessment plan, is a kind of contract with the students. Students might have switched to another course if the assessment had been different.
Is there a workaround?	Assessment may be changed with the consent of **all** students.
What to do if you're in breach	Contact the Associate Head of School

- **Classes Must Not Be Held, nor Assignments etc. Due after Week 14**

Who Says?	UNSW Assessment Rules § 2.5
Why Not?	Week 15 is revision week for students. It does **not** make a difference if there is no exam in *your* course, as the students will normally be doing other courses, too.
Is there a workaround?	No. The only grey area is if your assignment due in week 14 has to be extended, for *serious* reasons, to week 15. Contact the Associate Head of School if you are thinking about doing this.
What to do if you're in breach	Contact the Associate Head of School

- **Assessments Worth 20% or More of the Final Mark Must Not Be Held in Week 14**

Who Says?	UNSW Assessment Rules § 2.5
Why Not?	Week 14, too, is partly for revision for the final exams. It does **not** make a difference if there is no exam in *your* course, as the students will normally be doing other courses, too.

Is there a workaround?	No. But it's worth noting that a large assignment due in week 14 is OK if the students have a long time to work on it. For example, a 40% assignment due in week 14 but which they (should have) begun in week 8 is fine, because they would have had 7 weeks to do the assignment, so 40%/7 or about 6% is allocated to week 14.
What to do if you're in breach	Contact the Associate Head of School

- **Students Must Receive Feedback on their Progress By Week 8**

Who Says?	UNSW Assessment Rules § 2.1
Why?	Students may withdraw without financial penalty up to the end of week 2, and withdraw without academic penalty up to the end 8. They need feedback by week 8 on their progress so that they can make this call.
Is there a workaround?	No. The week 8 deadline is absolute for students (unless they are, e.g., run over by a bus after week 8), so it is absolute for lecturers, too.
What to do if you're in breach	Contact the Associate Head of School

- **Students Must Receive Complete Information about Assessment in Week 1**

Who Says?	UNSW Assessment Rules § 2.8
What Details?	(Quoted from the URL above): ○ the weight of each task in contributing to the overall mark; ○ the formulas or rules used to determine the overall mark; ○ minimum standards that are applied to specific assessment tasks, and the consequences if such standards are not met (including failure to submit particular tasks); ○ rules regarding penalties applied to late submissions; and ○ precise details of what is expected in terms of presentation of work for assessment. Emphasis should be placed on appropriate referencing conventions and requirements, on the degree of cooperation permitted between students, and on what constitutes plagiarism and the consequences of committing it.
What to do if you're in breach	Contact the Associate Head of School

Bill Wilson's contact info

Last updated:
UNSW's CRICOS Provider No. is 00098G

Quick Biography of Bill Wilson

- Born 1948, Sydney, Australia
- Schooling:
  1. West Ryde Public School, Sydney, NSW Australia 1955-1958 class photo (76K)
  2. Eastwood Public School, Sydney, NSW Australia 1959-1960 class photo (64K)
  3. Marsden High School, Sydney, NSW Australia 1961-1965 class photo (31K) ... More class photos

- National Undergraduate Scholar → B.Sc.(First Class Honours) at Australian National University, Canberra, 1970. Resident at Burton Hall.
- M.Sc. at Australian National University, Canberra, 1972
- Ph.D. at University of Sydney, 1977
- Dip.Comp.Sc. at University of Queensland, 1979

- Tutor, Mathematics, University of Queensland, 1976-1979
- Applications Programmer, University of Sydney Computing Centre, 1979-1980
- Lecturer, Computer Science, University of Queensland, 1981-1984
- Lecturer, Computer Science, Flinders University of South Australia, 1985-1989
- Visiting Fellow, Cognitive Studies Centre, University of Essex, 1986
- Senior Lecturer, Computer Science & Engineering, University of NSW, 1989-2000
- Visiting Fellow, Psychology, University of Queensland, 1991 and 1995
- Associate Head, School of Computer Science & Engineering, University of NSW, 1999-2001, 2002s2-2005s1, & 2006s1-?
- Associate Professor, Computer Science & Engineering, University of NSW, 2001-now

---

Erdös number: ≤ 5

Non-academic activities.

Bill Wilson

Contact Info | Photo Links Page | Bio Page

Last updated: undefined

COMP9414: Artificial Intelligence

Lecture Notes

January 2002: These lecture notes were developed and used over the period 1996-2001 during which I lectured the first two thirds of COMP9414. The notes for the material on Prolog Programming, Knowledge Representation and Machine Learning are based on earlier notes by Prof. Claude Sammut. The material on Computer Vision, that is, the final third of the material, was developed by A/Prof. Arcot Sowmya. Her links may become unavailable as time goes on.

Prolog Programming, Knowledge Representation and Machine Learning (5 weeks)

- Dictionary of Machine Learning Concepts (68 K)

Natural Language Processing (4 weeks)

- Linguistics
- Grammars and Parsing
- Features and Augmented Grammars
- Semantics and Logical Form
- Linking Syntax and Semantics
- Knowledge Representation for Natural Language
- Local Discourse and Reference
- Ambiguity Resolution - Statistical Methods

- Dictionary of Natural Language Processing Concepts (144K)

Lectures notes by A/Prof. Arcot Sowmya on...

Computer Vision (4 weeks)

- Introduction to Computer Vision
- Digital Images and Properties
- Pixel Relationships
- Binary Vision
- Preprocessing and Edge Detection
- Segmentation and Line Finding
- Region-Based Segmentation and Representation (covered in same document as previous section).
- 3D Segmentation and Description
- Model-Based Vision (covered in same document as previous section).
- Images Files

Bill Wilson | billw at cse.unsw.edu.au | Home Page
Last modified: 16 September 2002
CRICOS Provider No. 00098G

Artificial Intelligence

Aim:

To introduce basic issues in AI.

Plan:

- Brief history of modern AI
- Some applications of AI
- Intelligent agents
- Symbolic and non-symbolic representations

What is Artificial Intelligence?

"AI is the boundary between what people can do and what computers can't [yet]."

History

Ancient Times

- Aristotle (Logic as an instrument for studying thought)

Early Efforts by Philsopher/Mathemticians

- Descartes (The mind body problem)
- Hume (Cognition is computation)
- Euler (1735 - representation of structure; search)

Formal Notation and Calculating Machines

- Leibnitz (1887 - first system of formal logic; calculating machines)
- Babbage (Programmable computing machines)
- Boole (1847, 1859)
- Frege (1879, 1884 - first-order predicate calculus)

Modern Founders of AI

- Alan Turing ("Computing Machinery and Intelligence"; Turing test)
- McCulloch & Pitts (neural nets)
- Norbert Wiener (cybernetics)
- John von Neumann (game theory)
- Claude Shannon (information theory)
- Newell & Simon (The Logic Theorist)
- John McCarthy (LISP, common sense reasoning)
- Marvin Minsky (Frames)
- Donald Michie (Freddy)

Achievements of AI

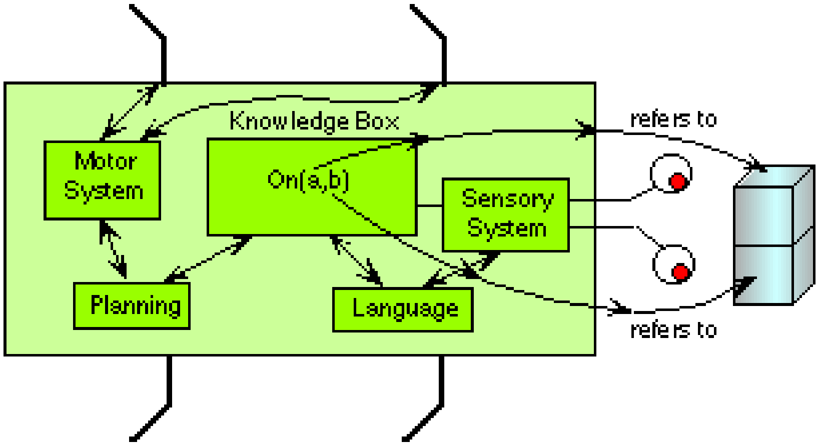
- Deep Thought is an international grand master chess player.
- Sphinx can recognise continuous speech without training for each speaker. It operates in near real time using a vocabulary of 1000 words and has 94% word accuracy.
- Navlab is a truck that can drive along a road at 55mph in normal traffic.
- Carlton and United Breweries use an AI planning system to plan production of their beer.
- Robots are used regularly in manufacturing.
- Natural language interfaces to databases can be obtained on a PC.
- Machine Learning methods have been used to build expert systems.
- Expert systems are used regularly in finance, medicine, manufacturing, and agriculture

How do we build an intelligent agent?

- Must be able to perceive its environment.
- Must be able to affect its environment.
- Must be able to reason about observations and actions
- Must be able to learn from observations and actions.
- Must have goals.

Symbolic Representations

To construct intelligent systems it is necessary to employ internal representations of a symbolic nature, with cognitive activity corresponding to computational manipulation of these symbolic representations. The symbolic representations must refer to the external world.

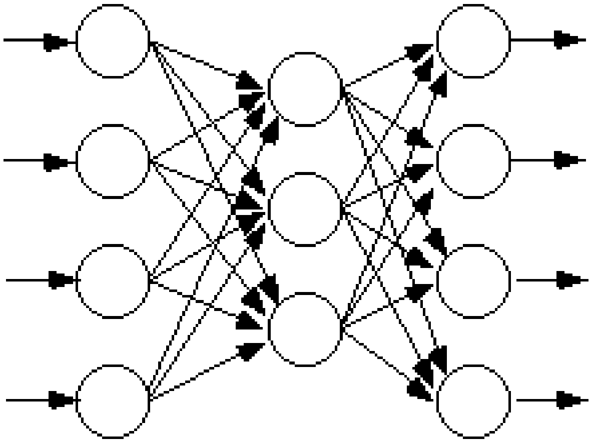


Advantages of symbolic representation:

- The system builder can read what the system knows.
- Knowledge is represented by sentences in a formal language.
- It is possible to read the representation and understand the meaning of the knowledge.

Non-symbolic Representations

Knowledge is represented by weights on connections in a network



Advantages of non-symbolic representation

- Can deal with combinations of attributes such as an image.
- Noise tolerant

CRICOS Provider Code No. 00098G

INTRODUCTION TO PROLOG

Reference: Bratko chapters 1-5

Aim:

To introduce enough of Prolog to allow students to do the assignment work in this course, thereby gaining some experience of AI programming.

Plan:

- What is Prolog? Programming with relations.
- Facts
- Questions
- Variables
- Conjunctions of goals, backtracking
- Rules
- Structures
- Recursive programs
- Lists
- Controlling execution - the cut

What is Prolog?

- invented early seventies by Alain Colmerauer in France.
- Programmation en Logique (Programming in Logic).
- differs from the most common programming languages
- is a declarative language
  - programmer specifies a goal to be achieved
  - Prolog system works out how to achieve it
- traditional programming languages are said to be procedural
- procedural programmer must specify in detail how to solve a problem:

> mix ingredients;
> beat until smooth;
> bake for 20 minutes in a moderate oven;
> remove tin from oven;
> put on bench;
> close oven;

turn off oven;

- in purely declarative languages, the programmer only states what the problem is and leaves the rest to the language system

Applications of Prolog

Some applications of Prolog are:

- intelligent data base retrieval
- natural language understanding
- expert systems
- specification language
- machine learning
- robot planning
- automated reasoning
- problem solving

The Prolog Language

Reference:	Bratko Chapter 1

Relations

- Prolog programs specify *relationships* among objects and properties of objects.
- When we say, "John owns the book", we are declaring the ownership relationship between two objects: John and the book.
- When we ask, "Does John own the book?" we are trying to find out about a relationship.
- Relationships can also rules such as:

  Two people are sisters **if**

  they are both female **and** they have the same parents.

- A rule allows us to find out about a relationship even if the relationship isn't explicitly stated as a fact.

Programming in Prolog

- declare facts describing explicit relationships between objects and properties objects might have (e.g. Mary likes pizza, grass has_colour green)
- define rules defining implicit relationships between objects (e.g. the sister rule above) and/or rules defining implicit object properties (e.g. X is a parent if there is a Y such that Y is a child of X).

One then uses the system by:

- asking questions above relationships between objects, and/or about object properties (e.g. does Mary like pizza? is Joe a parent?)

Facts

- Properties of objects, *or* relationships between objects;
- "Dr Turing lectures in course 9020", is written in Prolog as:

```
lectures(turing, 9020).
```

- *Notice:*
  - names of properties/relationships begin with lower case letters.
  - the relationship name appears as the first term
  - objects appear as comma-separated arguments within parentheses.
  - A period "." must end a fact.
  - objects also begin with lower case letters. They also can begin with digits (like 9020), and can be strings of characters enclosed in quotes (as in `reads(fred, "War and Peace")`).
- `lectures(turing, 9020).` is also called a *predicate*

Facts about a hypothetical computer science department:

```
% lectures(X, Y): person X lectures in course Y
lectures(turing, 9020).
lectures(codd, 9311).
lectures(backus, 9021).
lectures(ritchie, 9201).
lectures(minsky, 9414).
lectures(codd, 9314).

% studies(X, Y): person X studies in course Y
studies(fred, 9020).
studies(jack, 9311).
studies(jill, 9314).
studies(jill, 9414).
studies(henry, 9414).
studies(henry, 9314).

%year(X, Y): person X is in year Y
year(fred, 1).
year(jack, 2).
year(jill, 2).
year(henry, 4).
```

Together, these facts form Prolog's *database*. Should you wish to experiment with them using Prolog, they are available at http://www.cse.unsw.edu.au/~billw/cs9414/notes/prolog/facts03.

Queries

- Once we have a database of facts (and, soon, rules) we can ask questions about the stored information.

- Suppose we want to know if Turing lectures in course 9020. We can ask:

% **prolog -s facts03** (multi-line welcome message) ?- *lectures(turing, 9020).* Yes ?- *<control-D>* %	facts03 loaded into Prolog "?-" is Prolog's prompt output from Prolog hold down control & press D to leave Prolog

- *Notice:*
  - In SWI Prolog, queries are terminated by a full stop.
  - To answer this query, Prolog consults its database to see if this is a known fact.
  - In example dialogues with Prolog, the text in *italics* is what the user types.

Another example query

```
?- lectures(codd, 9020).
No
```

- if answer is `Yes`, the query *succeeded*
- if answer is `No`, the query *failed*
- The use of lower case for `codd` is critical.
- Prolog is not being intelligent about this - it would not see a difference between this query and `lectures(fred, 9020).` or `lectures(xyzzy, 9020).`
  though a person inspecting the database can see that `fred` is a student, not a lecturer, and that `xyzzy` is neither student nor lecturer.

Variables

- Suppose we want to ask, "What course does Turing teach"?
- This could be written as:

  ```
  Is there a course, X, that Turing teaches?
  ```

- The variable X stands for an object which the questioner does not know about yet.
- To answer the question, Prolog has to find out the value of X, if it exists.
- As long as we do not know the value of a variable it is said to be *unbound*.
- When a value is found, the variable is said to *bound* to that value.
- The name of a variable must begin with a capital letter or an underscore character, "_".

Variables 2

- To ask Prolog to find the course which Turing teaches, the following query is entered:

  ```
  ?- lectures(turing, Course).
  Course = 9020 ← output from Prolog
  ```

- To ask which course(s) Prof. Codd teaches, we may ask,

```
?- lectures(codd , Course).
Course = 9311 ; ← type ";" to get next solution
Course = 9314 ;
No
```

- Prolog can find all possible ways to answer a query, unless you explicitly tell it not to (see *cut*, later).

Conjunctions of Goals in Queries

- How do we ask, "Does Turing teach Fred"?

- This means finding out if Turing lectures in a course that Fred studies.

```
?- lectures(turing, Course), studies(fred, Course).
```

- i.e. "Turing lectures in course, `Course` and Fred studies (the same) `Course`".

- The question consists of two *goals*.

- To answer this question, Prolog must find a single value for Course, that satisfies both goals.

Backtracking in Prolog

- Who does Codd teach?

```
?- lectures(codd, Course), studies(Student, Course).
Course = 9311
Student = jack ;

Course = 9314
Student = jill ;

Course = 9314
Student = henry ;
```

- Prolog solves this problem by proceeding left to right and then *backtracking*.

- When given the initial query, Prolog starts by trying to solve

  ```
  lectures(codd, Course)
  ```

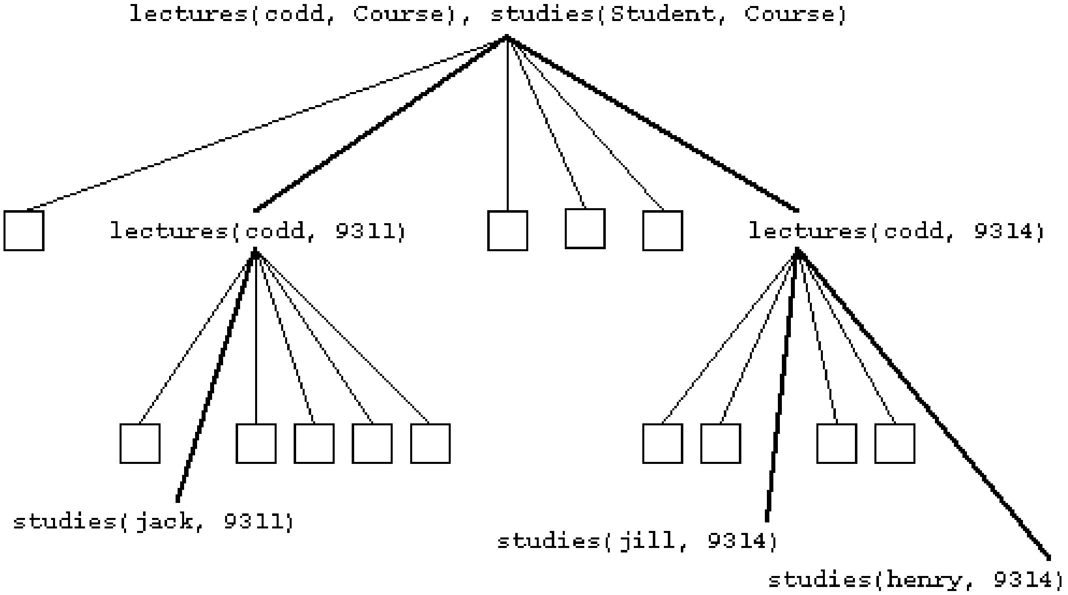
- There are six `lectures` clauses, but only two have `codd` as their first argument.
- Prolog uses the first clause that refers to `codd`: `lectures(codd, 9311).`
- With `Course = 9311`, it tries to satisfy the next goal, `studies(Student, 9311)`.
- It finds the fact `studies(jack, 9311).` and hence the first solution: (`Course = 9311, Student = jack`)

Backtracking in Prolog 2

- After the first solution is found, Prolog retraces its steps up the tree and looks for alternative solutions.
- First it looks for other students studying `9311` (but finds none).
- Then it
  - backs up
  - rebinds `Course` to `9314`,
  - goes down the `lectures(codd, 9314)` branch
  - tries `studies(Student, 9314)`,
  - finds the other two solutions:
    (`Course = 9314, Student = jill`)
    and (`Course = 9314, Student = henry`).

Backtracking in Prolog 3

To picture what happens when Prolog tries to find a solution and backtracks, we draw a "proof tree":



Rules

- The previous question can be restated as a general rule:

```
One person, Teacher, teaches another person, Student if
        Teacher lectures in a course, Course and
        Student studies Course.
```

- In Prolog this is written as:

```
teaches(Teacher, Student) :-
        lectures(Teacher, Course),
        studies(Student, Course).

?- teaches(codd, Student).
```

- Facts are *unit clauses* and rules are *non-unit clauses*.

Clause Syntax

- ":-" means "if" or "is implied by". Also called the *neck* symbol.
- The left hand side of the neck is called the *head*.
- The right hand side of the neck is called the *body*.

- The comma, ",", separating the goals is stands for *and*.
- Another rule, using of the *predefined predicate* ">".

```
        more_advanced(S1, S2) :-
                year(S1, Year1),
                year(S2, Year2),
                Year1 > Year2.
```

Tracing Execution

```
        more_advanced(S1, S2) :-
                year(S1, Year1),
                year(S2, Year2),
                Year1 > Year2.
```

```
?- trace.
Yes
[trace] ?- more_advanced(henry, fred).
  Call: more_advanced(henry, fred) ? *   bind S1 to henry, S2 to fred
  Call: year(henry, _L205) ?             test 1st goal in body of rule
  Exit: year(henry, 4) ?                 succeeds, binds Year1 to 4
  Call: year(fred, _L206) ?              test 2nd goal in body of rule
  Exit: year(fred, 1) ?                  succeeds, binds Year2 to 1
^ Call: 4>1 ?                            test 3rd goal: Year1 > Year2
^ Exit: 4>1 ?                            succeeds
  Exit: more_advanced(henry, fred) ?     succeeds
Yes
[debug]  ?- notrace.
```

* The `?` is a prompt. Press the return key at end of each line of tracing. Prolog will echo the <return> as `creep`, and then print the next line of tracing.

Structures

Reference:	Bratko chapter 2

- Functional terms can be used to construct complex data structures.
- If we want to say that John owns the novel Tehanu, we can write: owns(john, 'Tehanu').

- Often objects have a number of attributes: owns(john, book('Tehanu', leguin)).
- The author LeGuin has attributes too: owns(john, book('Tehanu', author (leguin, ursula))).
- The arity of a term is the number of arguments it takes.
- all versions of `owns` have arity 2, but the detailed structure of the arguments changes.
- `gives(john, book, mary).` is a term with arity 3.

Asking Questions with Structures

- How do we ask, "What books does John own which were written by someone called LeGuin"?

```
?- owns(john, book(Title, author(leguin, GivenName))).
Title = 'Tehanu'
GivenName = ursula
```

- What books does John own?

```
?- owns(john, Book).
Book = book('Tehanu', author(leguin, ursula))
```

- What books does John own?

```
?- owns(john, book(Title, Author)).
Title = 'Tehanu'
Author = author(leguin, ursula))
```

- Prolog performs a complex matching operation between the structures in the query and those in the clause head.

Library Database Example

- A database of books in a library contains facts of the form

```
book(CatalogNo, Title, author(Family, Given)).
member(MemberNo, name(Family, Given), Address).
loan(CatalogNo, MemberNo, BorrowDate, DueDate).
```

- A member of the library may borrow a book.
- A "loan" records:
    - the catalogue number of the book
    - the number of the member
    - the date on which the book was borrowed
    - the due date

Library Database Example 2

- Dates are stored as structures:
  `date(Year, Month, Day)`
- e.g. `date(2007, 6, 16)` represents 16 June 2007.
- which books has a member borrowed?

```
borrowed(MemFamily, Title, CatalogNo) :-
   member(MemberNo, name(MemFamily, _), _),
   loan(CatalogNo, MemberNo, _, _),
   book(CatalogNo, Title, _).
```

- The underscore or "don't care" variables (_) are used because for the purpose of this query we don't care about the values in some parts of these structures.

Comparing Two Terms

- we would like to know which books are overdue; how do we compare dates?

```
%later(Date1, Date2) if Date1 is after Date2:

later(date(Y, M, Day1), date(Y, M, Day2)) :-
                Day1 > Day2.
later(date(Y, Month1, _), date(Y, Month2, _)) :-
                Month1 > Month2.
later(date(Year1, _, _), date(Year2, _, _)) :-
                Year1 > Year2.
```

- This rule has three clauses: in any given case, only one clause is appropriate. They are tried in the given order.

- Again we are using the comparison operator ">"
- More complex arithmetic expressions can be arguments of comparison operators - e.g. `X + Y >= Z * W * 2`.

Overdue Books

```
% overdue(Today, Title, CatalogNo, MemFamily):
%   given the date Today, produces the Title, CatalagNo,
%   and MemFamily of all overdue books.

overdue(Today, Title, CatalogNo, MemFamily) :-
  loan(CatalogNo, MemberNo, _, DueDate),
  later(Today, DueDate),
  book(CatalogNo, Title, _),
  libmember(MemberNo, name(MemFamily, _), _).
```

Due Date

- Assume the loan period is one month:

```
due_date(date(Y, Month1, D),
         date(Y, Month2, D)) :-
           Month1 < 12,
                   Month2 is Month1 + 1.

due_date(date(Year1, 12, D),
         date(Year2, 1, D)) :-
           Year2 is Year1 + 1.
```

The `is` operator

- The right hand argument of `is` must be an arithmetic expression that can be evaluated right now (no unbound variables).
- This expression is evaluated and bound to the left hand argument.

- `is` is not a C-style assignment statement:
  - `X is X + 1` won't work!
  - except via backtracking, variables can only be bound once, using `is` or any other way

- `=` does not cause evaluation of its arguments:

```
?- X = 2, Y = X + 1.     ?- X = 2, Y is X + 1.
X = 2                    X = 2
Y = 2+1                  Y = 3
```

- Use `is` if and only if you need to evaluate something:

  `X is 1` BAD! - nothing to evaluate

  `X = 1` GOOD!

Recursive Programs

Reference:	Bratko section 1.3 (doesn't cover trees, though)

Binary Trees

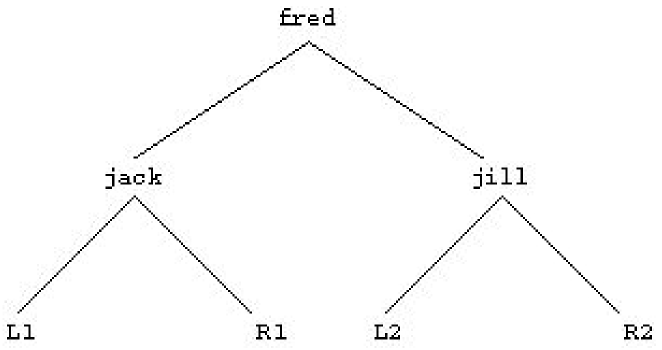
- In the library database example, some complex terms contained other terms, for example, `book` contained `name`.

- The following term also contains another term, this time one similar to itself:

```
tree(tree(L1, jack, R1), fred, tree(L2, jill, R2))
```

- The variables L1, L2, R1, and R2 should be bound to sub-trees (this will be clarified shortly).

- A structure like this could be used to represent a "binary tree" that looks like:

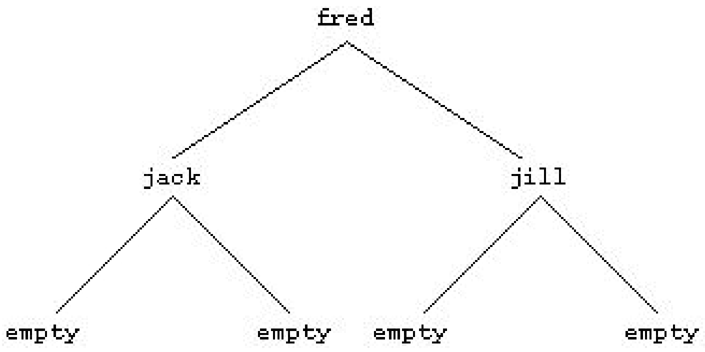


- Binary because each "node" has two branches (our backtrack tree before had many branches at some nodes)

- A term that contains another term that has the same principal functor (in this case `tree`) is said to be recursive.

- Regular trees have leaves.

- In our case, a leaf is a node with two empty branches:



- `empty` is an arbitrary symbol to represent the empty tree

Recursive Programs for Recursive Structures

- A binary tree is either empty or contains some data and a left and right subtree which are also binary trees.

- In Prolog we express this as:

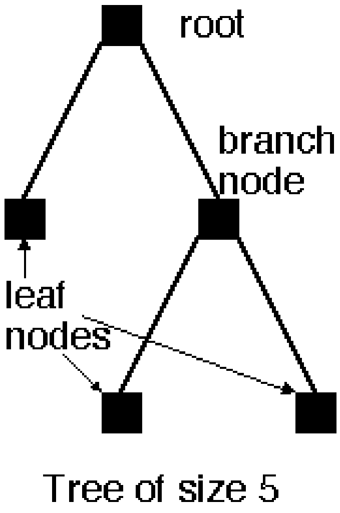
```
is_tree(empty).                            trivial branch
is_tree(tree(Left, Data, Right)) :-        recursive branch
      is_tree(Left),
      some_data(Data),
      is_tree(Right).
```

- A non-empty tree is represented by a 3-arity term.

Recursive Programs for Recursive Structures 2

- Let us define (or measure) the size of tree (i.e. number of nodes):

```
tree_size(empty, 0).
tree_size(tree(L, _, R), Total_Size) :-
        tree_size(L, Left_Size),
        tree_size(R, Right_Size),
        Total_Size is
        Left_Size + Right_Size + 1.
```



Tree of size 5

- The size of an empty tree is zero.
- The size of a non-empty tree is the size of the left sub-tree plus the size of the right sub-tree plus one for the current tree node.
- The data does not contribute to the total size of the tree.
- Recursive data structures need recursive programs. A recursive program is one which refers to itself, thus, `tree_size` contains goals that call for the `tree_size` of smaller trees.

Lists

Reference:	Bratko chapter 3

- A list may be nil (i.e. empty) or it may be a term which has a head and a tail
- The head may be any term or atom.
- The tail is another list.
- We could define lists as follows:

```
is_list(nil).
is_list(list(Head, Tail)) :-
        is_list(Tail).
```

- A list of numbers [1, 2, 3] would look like:

```
list(1, list(2, list(3, nil)))
```

- Since lists are used so often, Prolog has a special notation:

```
[1, 2, 3] = list(1, list(2, list(3, nil)))
```

Examples of Lists

```
?- [X, Y, Z] = [1, 2, 3]. Match the terms on either side of =
X = 1
Y = 2
Z = 3
```

```
?- [X | Y] = [1, 2, 3].    | separates head from tail of list.
X = 1
Y = [2, 3]
```

```
?- [X | Y] = [1]?
X = 1
Y = []                    The empty list is written as []
                          Lists "end" in an empty list!
```

The first several elements of the list can be selected before matching the tail:

```
?- [X, Y | Z] = [fred, jim, jill, mary].
X = fred                               Must be at least two elements
Y = jim                                in the list on the right.
Z = [jill, mary]
```

More Complex List Matching

```
?- [X | Y] = [[a, f(e)], [n, m, [2]]].

X = [a, f(e)]
Y = [[n, m, [2]]]
```

Notice that Y is shown with an extra pair of brackets: Y is the tail of the entire list: `[n, m, [2]]` is just one element.

List Membership

- A term is a member of a list if

    - the term is the same as the head of the list, or
    - the term is a member of the tail of the list.

- In Prolog:

```
member(X, [X | _]).
```

trivial branch:
a rule with a head but no body

```
member(X, [_ | Y]) :-
        member(X, Y).
```

recursive branch

The first rule has the same effect as: `member(X, [Y|_]) :- X = Y.`
The form `member(X, [X|_]).` is preferred, as it avoids the extra calculation.

- `member` is actually predefined in Prolog.

Programming Principles for Recursive Structures

- Only deal with one element at a time.
- Believe that the recursive program you are writing has already been written.

  In the definition of `member`, we are already assuming that we know how to find a member in the tail.

- Write definitions, not programs!
    - If you are used to writing programs for conventional languages, then you are used to giving instructions on how to perform certain operations.
    - In Prolog, you define relationships between objects and let the system do its best to construct objects that satisfy the given relationship.

Concatenating Two Lists

- Suppose we want to take two lists, like `[1, 3]` and `[5, 2]` and concatenate them to make `[1, 3, 5, 2]`
- First, the trivial branch: `concat([], L, L).`

- Next, the recursive branch:

```
        concat([Item | Tail1], L, [Item | Tail2]) :-
              concat(Tail1, L, Tail2).
```

- For example, consider `concat([1], [2], [1, 2])`: By the recursive branch:

```
        concat([1 | []], [2], [1 | [2]]) :-
              concat([], [2], [2]).
```

  and `concat([], [2], [2])`
  holds because of the trivial branch.

- Entire program is:

```
        concat([], L, L).
        concat([Item | Tail1], L, [Item | Tail2]) :-
              concat(Tail1, L, Tail2).
```

An Application of Lists

- Find the total cost of a list of items:

```
        % data:
        cost(cornflakes, 230).
        cost(cocacola, 210).
        cost(chocolate, 250).
        cost(crisps, 190).
```

-
```
  % code:
        total_cost([], 0).
        total_cost([Item|Rest], Cost) :-
            total_cost(Rest, CostOfRest),
            cost(Item, ItemCost),
            Cost is ItemCost + CostOfRest.

        % sample query:
        : total_cost([cornflakes, crisps], X)?

        X = 420
```

Controlling Execution

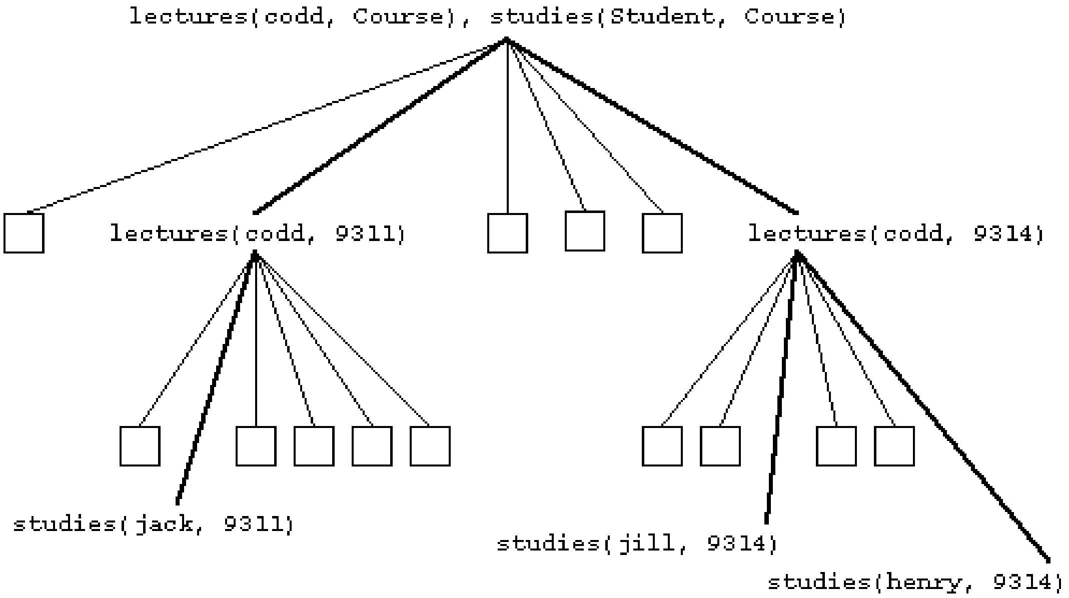
Reference:	Bratko chapter 5

The Cut Operator (!)

- Sometimes we need a way to prevent Prolog finding all solutions, i.e. a way to stop backtracking.
- The cut operator, written `!`, is a built-in goal that prevents backtracking.
- It turns Prolog from a nice declarative language into a hybrid monster.
- Use cuts sparingly and with a sense of having sinned.

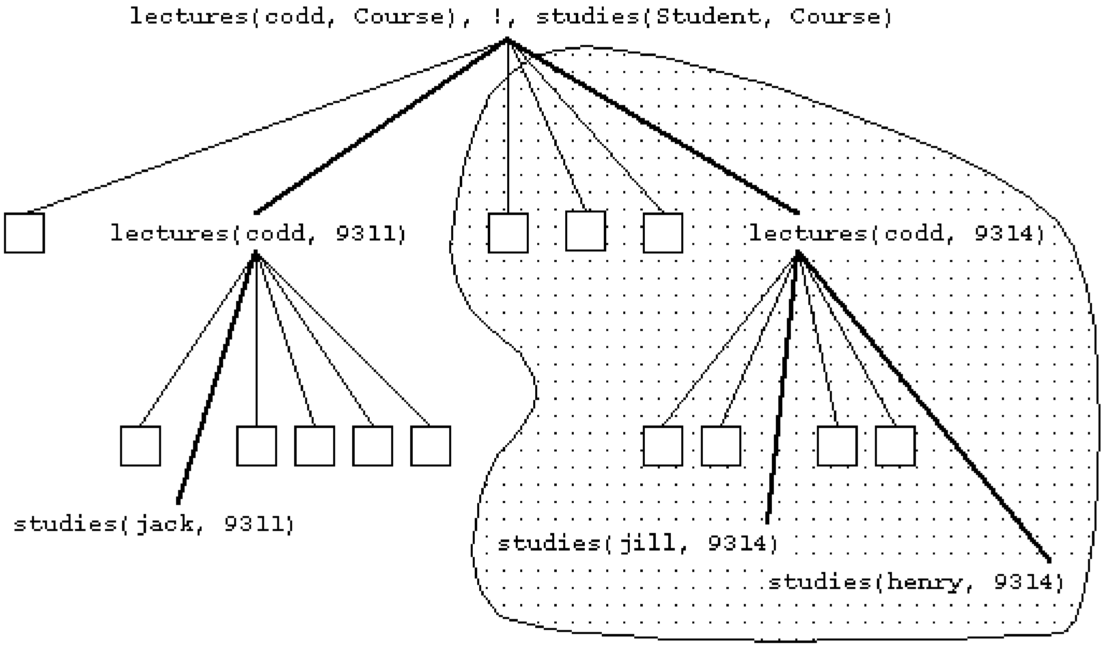
Cut Operator 2

Recall this example:



Cut Prunes the Search Tree

- If the goal(s) to the right of the cut fail then the entire clause fails and the the goal which caused this clause to be invoked fails.



- In particular, alternatives for Course are not explored.

Cut Prunes the Search Tree 2

- Another example: using the `facts03` database, try

```
?- lectures(codd, X).
X = 9311 ;
X = 9314 ;
No

?- lectures(codd, X), ! .
X = 9311 ;
No
```

- The cut in the second version of the query prevents Prolog from backtracking to find the second solution.

Summary: Introduction to Prolog
We have introduced facts, queries, variables, conjunctions of goals, rules, structures, recursion, trees and lists, and controlling execution by means of the "cut".

---

due to Maurice Pagnucco.

Logic and Rules

Expressions in a formal language conform to unambiguous rules of construction. Inferences are drawn by following strict laws for manipulating expressions in a formal language The language we use most often is clausal form logic.

Propositional Calculus

A *propositional constant* is a symbol (like p, q, r, ...) that stands for some like ``Sydney is a city''. Propositions are *atomic* formulae. A *well-formed formula*(wff) is defined as:

if X is atomic then X is a wff

if X is a wff then so is **not** X

if X and Y are wffs then so is X **or** Y

X **and** Y is defined as **not**(**not** X **or not** Y)

(X => Y) is defined as (**not** X **or** Y)

(X **equiv** Y) is defined as (X => Y) **and** (Y => X)

The letters X and Y are propositional variables that denote any proposition symbol.

Conjunctive Normal Form

In *conjunctive normal form*(CNF) we eliminate conditional and bi-conditional expressions by converting them to disjunctions. Negation is moved in so that it only applies to atoms:

not(X **and** Y) becomes (**not** X **or not** Y)

not(X **or** Y) becomes (**not** X **and not** Y)

not not X becomes X

Disjunction is distributed over conjunction using:

(Z **or** (X **and** Y)) becomes
((Z **or** X) **and** (Z **or** Y))

Clausal Form

Usually, we reduce arbitrary expressions to clausal form. For example,

not(p **or** q) => (**not** p **and not** q)	
not not(p **or** q) **or** (**not** p **and not** q)	eliminating =>
(p **or** q) **or** (**not** p **and not** q)	driving **not** in
not p **or** (p **or** q)) **and** (**not** q **or** (p **or** q))	distributing **and** over **or**
{{**not** p, p, q}, {**not** q, p, q}}	dropping **and** and **or**

The expressions in the braces are either atoms or negated atoms and are called *literals*. In clausal form, positive literals are placed to the right of an arrow symbol and negative atoms to the left. E.g.

p, q <- p
p, q <- q

In general, a clause is an expression of the form:

p_1, ..., p_m <- q_1, ..., q_n

The literals on the left are disjoined *conclusions*. The literals on the right are conjoined *conditions*.

Predicate Calculus

Propositional calculus cannot deal with statements of generality like,

`All men are mortal'.

To do this, we need predicates, arguments, variables and quantifiers. eg.

(forall x)(man(x) => mortal(x))

The Syntax of Predicate Calculus

- Any constant symbol or variable is a term.
- If F_k is a k-place function symbol and a_1, ..., a_k are terms then F_k(a_1, ..., a_k) is a term.
- If P_k is a k-place predicate symbol and a_1, ..., a_k are terms then P_k(a_1, ..., a_k) is a wff.

- Negation and disjunction rules are as for propositional calculus.
- If W is a wff and X is a variable then (forall X) W is a wff.
- (exists X) W is defined to be **not** ((forall X) **not** W).

First-Order Clauses

Like propositional calculus, there is a procedure for converting arbitrary expressions into clauses. If a clause has the form p_1, ..., p_m <- q_1, ..., q_n and contains variables x_1, ..., x_k, the interpretation is:

- (forall x_1, ..., x_k), p_1 **or ... or** p_m is true if q_1 **and ... and** q_n are true.
- if n=0 (i.e. no conditions are specified) then p_1 **or ... or** p_m is true.
- if m= 0 (i.e. no conclusins are specified) then q_1 **and ... and** q_n is false.

Horn Clauses

A *Horn clause*only has a single positive literal. For example,

p <- q_1, ..., q_n

A program in Prolog consists of Horn clause definitions. For example,

```
on(a, b).
on(b, c).

above(X, Y) :- on(X, Y).
above(X, Y), on(Z, Y), above(X, Z).
```

The Resolution Proof Procedure

To prove p follows from some theory, T, assume **not** p and then try to derive a contradiction from its conjunction with T. Resolution requires a pattern matching operation, called *unification*. When matching literals, we look for variable substititions that will make the two expressions identical. For example,

```
runs_faster_than(X, zeno)
runs_faster_than(tortoise, Y)
```

are identical under the substitution {X/tortoise, Y/zeno}. A clause that contains no variables is called a ground clause. To resolve two non-ground clauses, you must find a unifier for complimentary literals. For example, {beats_in_race(X, zeno), **not** younger_than(X, zeno)}

and

{**not** beats_in_race(tortoise, Y), **not** philosopher(Y)}

have unifier U = {X/tortoise, Y/zeno} and generate the resolvent

{**not** philosopher(zeno), **not** younger_than(tortoise, zeno)}

We can *prove*a formula, p, if we can derive it from a theory, T by a sequence of resolution steps. We write this as T |- p. Resolution uses a proof by *refutation*. That is, add the negation of the goal to the theory and show that the new theory is inconsistent, ie. implies *false*. The empty clause, {}, is interpreted as false. So if theory derives false, we have an inconsistent theory.

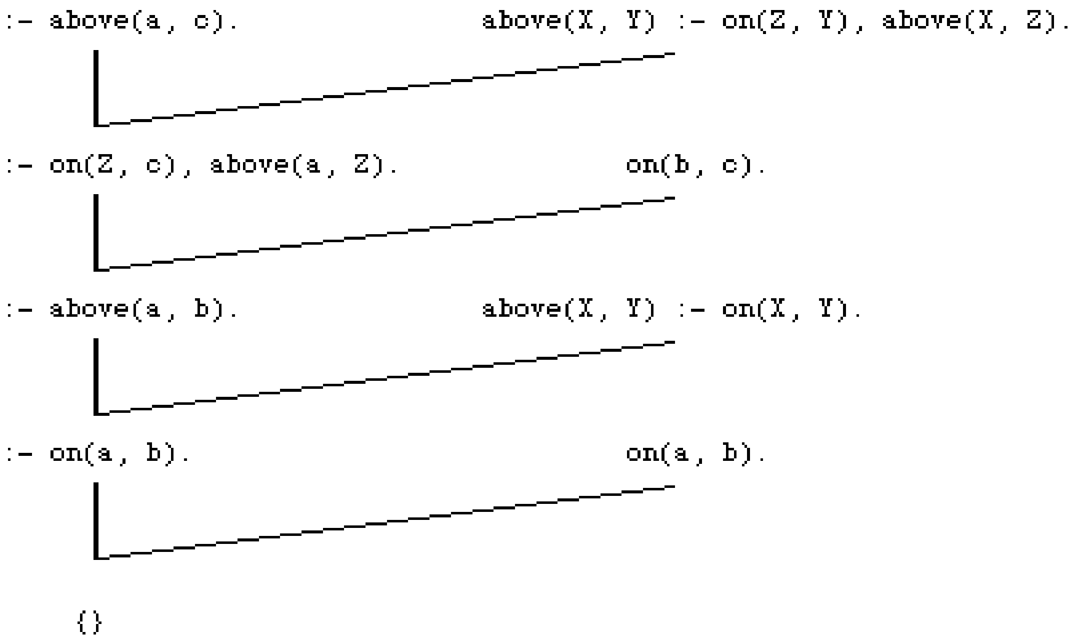


Figure 1: Resolution Proof

Resolution uses *backward chaining*to focus the search for clauses to resolve. There are many refinements to this search. Prolog resolves clauses and their literals in *input order*, i.e., top-to-bottom, left-to-right.

Soundness and Completness

- A proof procedure is *sound*if every formula it derives is true. I.e. it cannot prove something it shouldn't.
- A proof procedure is *complete*if it can derive every thing that is possible to derive from a theory. That is, here is no true statement that it cannot prove.
- *Decidability*means that we can always show if a proposition follows from a theory.

- Prolog's proof procedure is sound and complete for Horn clauses.
- Unrestricted first-order logic is undecidable.

PRODUCTION RULES

Reference: Bratko ed. 3, chapter 15, page 347-

Aim:

To describe the systems that represent knowledge in the form of rules. Rule-based systems normally use a *working memory* that initially contains the input data for a particular run, and an *inference engine* to find applicable rules and apply them.

Keywords: backward chaining, condition-action rule, conflict resolution, expert system, fire, forward chaining, inference engine, match-resolve-act cycle, ripple-down rules, rule-based system, working memory

Plan:

- condition-action rules can represent knowledge
- backward and forward chaining
- rules, facts, working memory, inference engine
- match-resolve-act cycle: conflict resolution strategies
- BAGGER example system

Rules

The components of a rule-based system have the form:

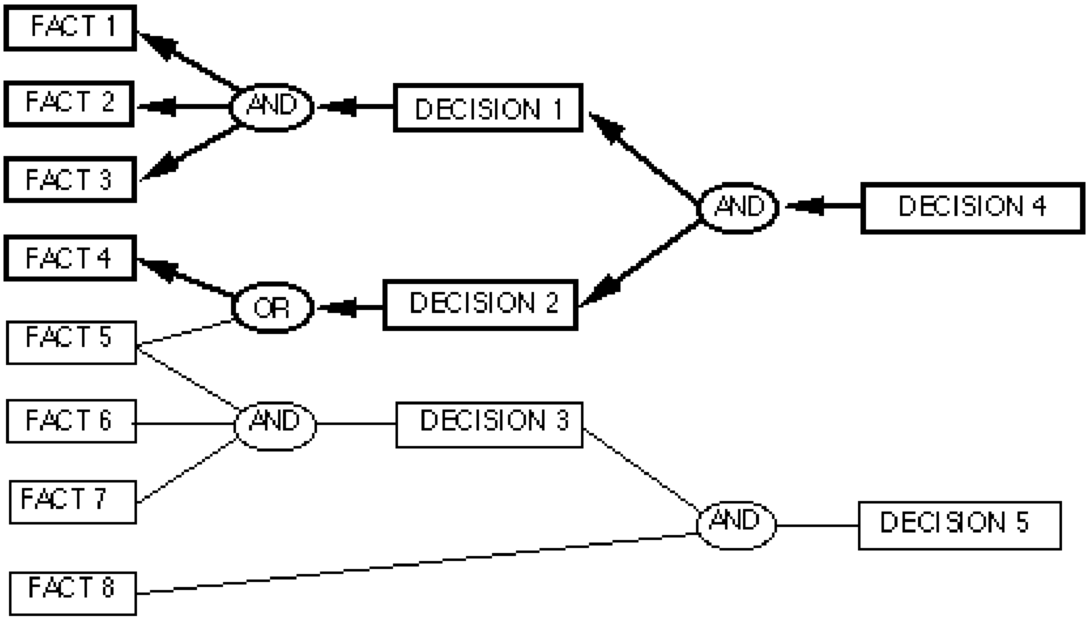
```
if <condition> then <conclusion>
```

Rules can be evaluated by:

- backward chaining
- forward chaining

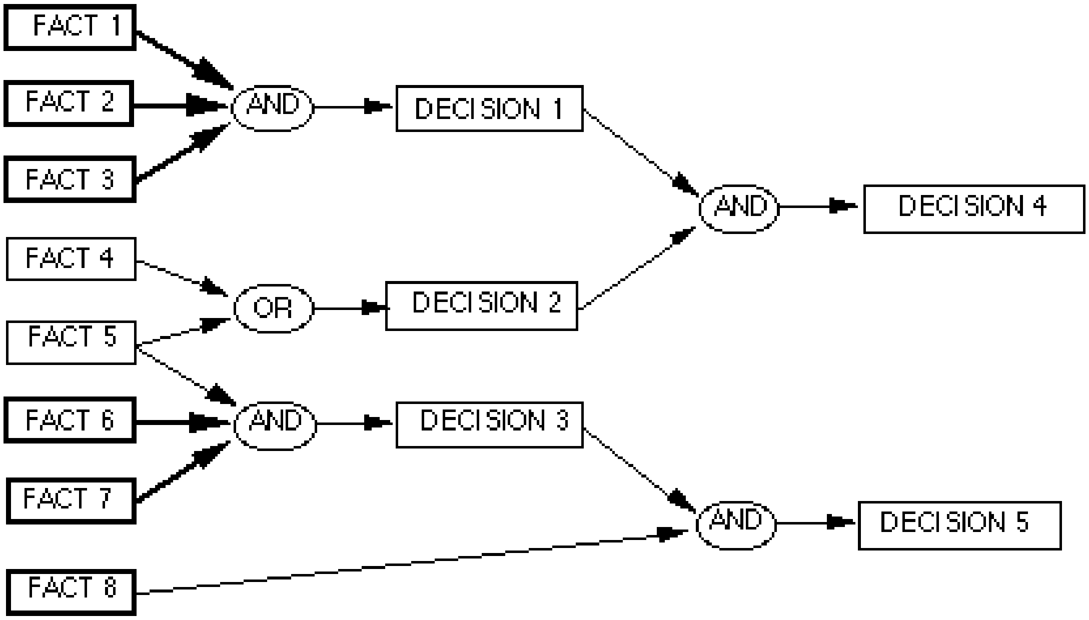
Backward Chaining

- To determine if a decision should be made, work backwards looking for justifications for the decision.
- Eventually, a decision must be justified by facts.



Forward Chaining

- Given some facts, work forward through inference net.
- Discovers what conclusions can be derived from data.



Forward Chaining 2

Until a problem is solved or no rule's 'if' part is satisfied by the current situation:

1. Collect rules whose 'if' parts are satisfied.
2. If more than one rule's 'if' part is satisfied, use a conflict resolution strategy to eliminate all but one.
3. Do what the rule's 'then' part says to do.

Production Rules

A production rule system consists of

- a set of rules
- working memory that stores temporary data
- a forward chaining inference engine

Match-Resolve-Act Cycle

The match-resolve-act cycle is what the inference engine does.

```
loop
    match conditions of rules with contents of working memory
    if no rule matches then stop
    resolve conflicts
    act (i.e. perform conclusion part of rule)
end loop
```

BAGGER

- Bagger is a simple rule-based system that describes how to pack items at a supermarket check-out.
- While explaining Bagger, we shall describe a number of potential strategies for conflict resolution.
- Bagger's working memory has an associated table of attributes of the objects (stock items) at the supermarket.
- There are 4 steps in Bagger, and Bagger uses a Working Memory item called "Step" to keep track of where it is up to.
- Each rule checks the value of "Step" as part of its **if** part, and will be applicable only to one of the four steps.
- This makes it easier to be sure that the rules will not interact in unexpected ways (a pitfall in creating rule-based systems).

Steps in Bagger

1. **Check order**: Check what the customer has selected; look to see if something

is missing, suggest additions.

2. **Pack large items**:Put the large items in the bag; put big bottles first.
3. **Pack medium items**:Put in the medium sized items; put frozen food in plastic bags.
4. **Pack small items**:Put in the small items wherever there is room.

Working Memory

```
Step:     Check order
Bag1:     <empty>
Unpacked: Bread
          Glop
          Granola (2)
          Ice cream
          Chips
```

Attributes of Objects

ITEM	CONTAINER TYPE	SIZE	FROZEN?
Bread	Plastic bag	Medium	No
Glop	Jar	Small	No
Granola	Cardboard box	Large	No
Ice cream	Cardboard carton	Medium	Yes
Pepsi	Bottle	Large	No
Chips	Plastic bag	Medium	No

Rules for Step 1

```
B1:
if      the step is check-order
and     there is a bag of chips
and     there is no soft-drink bottle
then    add one bottle of soft drink to the order

B2:
if      the step is check-order
then    discontinue the check-order step
and     start the pack-large-items step
```

Which of these rules should be chosen when in the check order step?

Conflict Resolution

Specificity Ordering

If a rule's condition part is a superset of another, use the first rule since it is more specialised for the current task.

Rule Ordering

Choose the first rule in the text, ordered top-to-bottom.

Data Ordering

Arrange the data in a priority list. Choose the rule that applies to data that have the highest priority.

Size Ordering

Choose the rule that has the largest number of conditions.

Conflict Resolution continued

Recency Ordering

The most recently used rule has highest priority *or*
the least recently used rule has highest priority *or*
the most recently used datum has highest priority *or*
the least recently used datum has highest priority.

More details

Context Limiting

> Reduce the likelihood of conflict by separating the rules into groups, only some of which are active at any one time. Have a procedure that activates and deactivates groups.

Rules for Step 2

```
B3:
if      the step is pack-large-items
and     there is a large item to be packed
and     there is a large bottle to be packed
and     there is a bag with < 6 large items
then    put the bottle into the bag

B4:
if      the step is pack-large-items
and     there is a large item to be packed
and     there is a bag with < 6 large items
then    put the large item into the bag

B5:
if      the step is pack-large-items
and     there is a large item to be packed
then    get a new bag
```

Working Memory So Far

```
Step:           pack-medium-items
Bag1:           Pepsi
                Granola (2)
Unpacked:       Bread
                Glop
                Ice cream
                Chips
```

Rules for Step 3

```
B7:
if      the step is pack-medium-items
and     there is a medium item to be packed
and     there is an empty bag or a bag with medium items
and     the bag is not yet full
and     the medium item is frozen
and     the medium item is not in a freezer bag
then    put the medium item in a freezer bag

B8:
if      the step is pack-medium-items
and     there is a medium item to be packed
and     there is an empty bag or a bag with medium items
and     the bag is not yet full
then    put the medium item in the bag

B9:
if      the step is pack-medium-items
and     there is a medium item to be packed
then    get a new bag

B10:
if      the step is pack-medium-items
then    discontinue the pack-medium-items step
and     start the pack-small-items step
```

Working Memory So Far

```
Step:           pack-small-items
```

```
Bag1:           Pepsi
                Granola (2)
Bag2:           Bread
                Ice cream (in freezer bag)
                Chips
Unpacked:       Glop
```

Rules for Step 4

```
B11:
if      the step is pack-small-items
and     there is a small item to be packed
and     the bag is not yet full
and     the bag does not contain bottles
then    put the small item in the bag

B12:
if      the step is pack-small-items
and     there is a small item to be packed
and     the bag is not yet full
then    put the small item in the bag

B13:
if      the step is pack-small-items
and     there is a small item to be packed
then    get a new bag

B14:
if      the step is pack-small-items
then    discontinue the pack-small-items step
and     stop
```

Implementing Rules in Prolog

Click here to see executable Prolog code for a simple production rule system.

To use this code, copy it to your own directory, e.g. by

```
% cd
% cp ~cs9414/public_html/Examples/rules-swi.pl ~
```

then start Prolog and do the following dialogue:

Dialogue with rules-swi.pl

```
% prolog rules-swi.pl
?- wm(X).
```

[Prolog will tell you which facts it knows (a, b, and c). Don't forget to type ";" after each solution is produced by Prolog.]

```
?- run.
Yes
?- wm(X).
```

[The answer tells you that Prolog still knows that a, b, and c are true, and also which other "facts" it now knows. Don't forget the ";"s.]

```
?- already_fired(X, Y).
```

[This time the answer tells you that two rules have fired, and gives their names and their conditions. A rule with the name `null` is also mentioned - this is a workaround in the code to avoid Prolog complaining that `already_fired` is undefined, in cases where no rules have yet been fired.]

```
?= ^D Exit from Prolog
%
```

You can also play with the code - e.g. by writing your own rules and facts, and running the system with them.

Summary: Rule-Based Systems

Rule-based systems consist of a set of rules, a working memory and an inference engine. The rules encode domain knowledge as simple condition-action pairs. The working memory initially represents the input to the system, but the actions that occur when rules are fired can cause the state of working memory to change. The inference engine must have a conflict resolution strategy to handle cases where more than one rule is eligible to fire.

Much of the material on this page is based on an earlier version by Claude Sammut.

CGIWrap Error: Execution of this script not permitted

Execution of (cs9414/notes/kr/rules/forward.pl) is not permitted for the following reason:

Script is not executable. Issue 'chmod 755 filename'

Server Data:

Server Administrator/Contact: ss@cse.unsw.edu.au
Server Name: cgi.cse.unsw.edu.au
Server Port: 80
Server Protocol: HTTP/1.0
Virtual Host: cgi.cse.unsw.edu.au

Request Data:

User Agent/Browser: Mozilla/3.0 (compatible; WebCapture 2.0; Auto; Windows)
Request Method: GET
Remote Address: 210.212.244.133
Remote Port: 12022
Referring Page: http://www.cse.unsw.edu.au/~billw/cs9414/notes.html

Uncertain Reasoning

Reference: Bratko ed. 3, p. 360-

Aim:

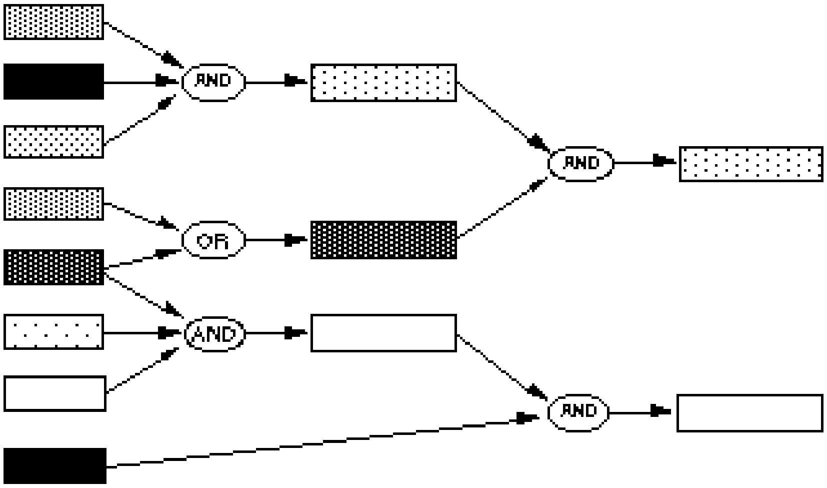
Sometimes the knowledge in rules is not certain. Rules then may be enhanced by adding information about how certain the conclusions drawn from the rules may be. Our aim in this secion is to describe **certainty factors** and their manipulation.

Keywords: certainty factor

Plan:

- why we might be uncertain
- measuring uncertainty
- using certainty factors in rules
- combining uncertain evidence from more than one source

- Often, experts can't give definite answers.
- May require an inference mechanism that derives conclusions by combining uncertainties.



Certainty Factors

- Logic and rules provide all or nothing answers
- An expert might want to say that something provides evidence for a conclusion, but it is not definite.
- For example, the MYCIN system, an early expert system that diagnosed bacterial blood infections, used rules of this form:

```
if   the infection is primary-bacteremia
```

```
and  the site of the culture is one of the sterile sites
and  the suspected portal of entry is the gastrointestinal tract
then there is suggestive evidence (0.7) that the infection is bacteroid
```

- 0.7 is a *certainty factor*

Certainty factors have been quantified using various different systems, including linguistics ones (certain, fairly certain, likely, unlikely, highly unlikely, definitely not) and various numeric scales, such as 0-10, 0-1, and -1 to 1. We shall concentrate on the -1 to 1 version.

Certainty factors may apply both to facts and to rules, or rather to the conclusion(s) of rules.

A "Theory" of Certainty

- Certainty factors range from -1 to +1
- As the certainty factor (CF) approaches 1 the evidence is stronger *for* a hypothesis.
- As the CF approaches -1 the confidence *against* the hypothesis gets stronger.
- A CF around 0 indicates that there is little evidence either for or against the hypothesis.

Certainty Factors and Rules

- Premises for rules are formed by the *and* and *or* of a number of facts.
- The certainty factors associated with each condition are combined to produce a certainty factor for the whole premise.
- For two conditions P1 and P2:
  CF(P1 and P2) = min(CF(P1), CF(P2))
  CF(P1 or P2) = max(CF(P1), CF(P2))
- The combined CF of the premises is then multiplied by the CF of the rule to get the CF of the conclusion

Example

if (P1 **and** P2) **or** P3 **then** C1 (0.7) **and** C2 (0.3)

Assume CF(P1) = 0.6, CF(P2) = 0.4, CF(P3) = 0.2

CF(P1 and P2) = min(0.6, 0.4) = 0.4

CF(0.4, P3) = max(0.4, 0.2) = 0.4

CF(C1) = 0.7 * 0.4 = 0.28

CF(C2) = 0.3 * 0.4 = 0.12

Combining Multiple CF's

- Suppose two rules make conclusions about C.
- How do we combine evidence from two rules?
- Let $CF_{R1}(C)$ be the current CF for C.
- Let $CF_{R2}(C)$ be the CF for C resulting from a new rule.
- The new CF is calculated as follows:

$CF_{R1}(C) + CF_{R2}(C) - CF_{R1}(C) * CF_{R2}(C)$	when $CF_{R1}(C)$ and $CF_{R2}(C)$ are both positive
$CF_{R1}(C) + CF_{R2}(C) + CF_{R1}(C) * CF_{R2}(C)$	when $CF_{R1}(C)$ and $CF_{R2}(C)$ are both negative
$[CF_{R1}(C) + CF_{R2}(C)]/[1 - \min(\lvert CF_{R1}(C)\rvert, \lvert CF_{R2}(C)\rvert)]$	when $CF_{R1}(C)$ and $CF_{R2}(C)$ are of opposite sign

---

What do certainty factors mean?

- They are guesses by an expert about the relevance of evidence.
- They are *ad hoc*.
- CF's are tuned by trial and error.
- CF's hide more knowledge.

Summary: Uncertain Reasoning
Certainty factors quantify the confidence that an expert might have in a conclusion that s/he has arrived at. We have given rules for combining certainty factors to obtain estimates of the certainty to be associated with conclusions obtained by using uncertain rules and uncertain evidence.
Exercises on uncertain reasoning [- try them before clicking ...] Solutions to exercises

CRICOS Provider Code No. 00098G

Assumption-Based Truth Maintenance

In problem solving, it is often useful to project what might happen in hypothetical circumstances. When information is received over a period of time (i.e. not known all at once) it may be necessary to make assumptions, allowing them to be contradicted later. An ATMS maintains independent chains of reasoning based on assumptions and adjusts current beliefs as new information is received. The basic ATMS architecture is shown in Figure 1.

The ATMS does not take part in any problem solving. It is required to indicate what can be believed in the current environment i.e. under a given set of assumptions so whenever the problem solver draws a conclusion, it passes that conclusion as well as the justification to the ATMS. The ATMS then incorporates the new information into its set of beliefs and indicates if any contradictions have arisen.

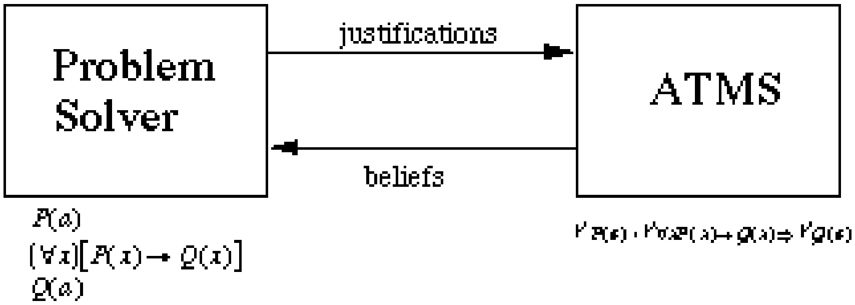

Problem Solver
justifications
ATMS
beliefs
$P(a)$
$(\forall x)[P(x) \rightarrow Q(x)]$
$Q(a)$
$\vdash P(a), \vdash \forall x P(x) \rightarrow Q(x) \Rightarrow \vdash Q(a)$

Figure 1. ATMS Architecture

The input to an ATMS consists of a sequence of propositional horn clauses presented over a period of time. Figure 1 shows three logical expressions used to make an inference. When these are transmitted to the ATMS, each expression is converted to an atom because the ATMS does not care about the inference itself hence all of the clauses that the ATMS sees are propositional. A further example of input to the ATMS is the sequence of clauses:

```
assume a.       p :- a, b.
assume b.       p :- b, c, d.
assume c.
assume d.       q :- a, c.
assume e.       q :- d, e.

:- a, b, e.     r :- p, q.
```

In this case, we have entered all of the assumptions first, followed by a *nogood*. A nogood set is a set of propositions that cannot be true simultaneously, so *a*, *b*, and *e* cannot all be true at the same time. The rest of the horn clauses indicate conclusions drawn by the problem solver. For example, it was concluded that *p* was true because a and *b* where assumed to be true and *r* was found to be true because *p* and *q* are hypotheses supported by the current set of assumptions.

Whenever assumptions are made and each time a clause is entered, we update a graph in which each node represents a belief. The information above would result in the graph shown in Figure 2, drawn as and "and-or" tree where the arcs indicate those propositions that are to conjoined.

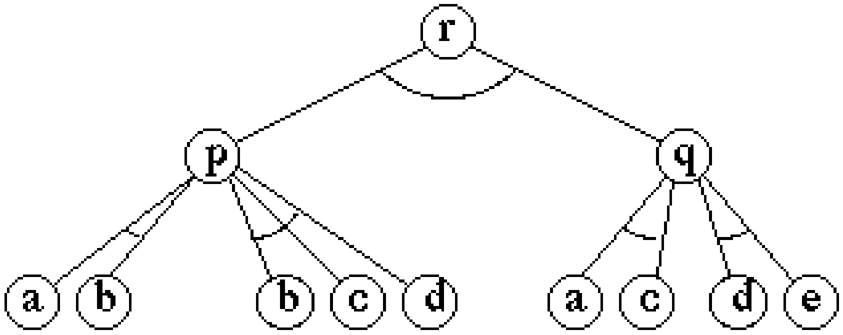


Figure 2: ATMS graph

The information stored in each node is:

- The node description (e.g. a, b, c, *etc*.)
- Consequents of the node (the parent nodes in the graph)
- Justifications for the node (the child nodes, grouped by the clause of origin)
- The node's label (indicating the assumptions upon which this belief is based)

The label is a set of *environments* and an environment is a set of assumptions under which the node is believed true. For example we can describe each of the nodes in the graph using Prolog notation as follows:

```
node(p, [r], [(p :- a, b), (p :- b, c, d)], {{a, b}, {b, c, d}}).
node(q, [r], [(q :- a, c), (q :- d, e)],    {{a, c}, {d, e}}).
node(r, [],  [(r :- p, q)],                 {{a, b, c}, {b, c, d, e}}).
```

Thus, *r*, can be believed if we assume that *a*, *b* and *c* are true at the same time or *b*, *c*, *d* and *e* are all true. Note that the assumptions for *r* are obtained by propagating the assumptions from its descendants. The reason for keeping the label is so that we can easily determine whether a proposition is believable or not.

P, *q* and *r* are nodes that have been derived by some deduction on the part of the problem solver. There are three types of nodes that are not derived. They are:

Assumptions:

these nodes are that are justified by themselves e.g.

```
node(a, [], [], {{a}}).
```

Premises:

these are nodes representing facts that do not need justification, e.g.

```
node(g, [], [], {{}}).
```

Note that the label contains an empty environment meaning that the proposition can be believed without having to assume anything.

False:

false is the node implied by all nogood sets:

```
node(false, [], [], {}).
```

If, after updating the graph, a node has an empty label then it is not believed to be true any more. This is because there is no set of assumptions that can justify the conclusion. An empty label results when a set of assumptions (i.e. an environment) contains a nogood set and therefore must be removed since the assumptions are contradictory. If all of the environments are removed, the label becomes empty indicating that the proposition cannot be believed. For example, r could have been derived by the following rules:

```
p :- a, b.
q :- d, e.
_________
r :- p, q.
```

However, this would require r to depend on the assumptions {a, b, d, e} which would appear as an environment in *r*'s label. Since this contains a nogood set, this environment must be removed from the label.

In addition to removing nogood environments, the ATMS algorithm also ensures that a label does not contain environments that are supersets of others since the label would then have redundant information. For example, *r*'s label could contain two environments {a, b, c} and {a, b, c, d}. Since {a, b, c} subsumes {a, b, c, d}, the second environment is redundant and can be removed.

We refer the reader to de Kleer for a detailed description of the ATMS update algorithm. For our purposes, the main operations of the ATMS are best seen in an example. Suppose that the ATMS has been given all but the final horn clause `r:- p, q`. When this clause is presented, it is up to the ATMS to adjust its beliefs. This proceeds as follows:

1. Find the labels for the antecedents in the clause
   p: {{a, b}, {b, c, d}}
   q: {{a, c}, {d, e}}

2. Find all combinations of assumptions for *p and q*.
   {{a, b, c}, {a, b, d, e}, {a, b, c, d}, {b, c, d, e}}
3. Eliminate all environments that are no good (i.e. are supersets of the nogoods).
   {{a, b, c}, {a, b, c, d}, {b, c, d, e}}
4. Eliminate all environments that are supersets of others in the label.
   {{a, b, c}, {b, c, d, e}}
5. The new label is now attached to the node *r*. If *r* had already existed and had its own consequents then *r*'s label would have to be propagated recursively through to the consequents.

From this example, it can be seen that the ATMS algorithm requires frequent set union operations to calculate a new label and also many subset tests to find nogoods and environments that are subsumed by other environments.

References

de Kleer, J. (1986). An Assumption based TMS, *Artificial Intelligence*, **28**(1).

de Kleer, J. (1986). Extending the ATMS, *Artificial Intelligence*, **28**(1).

de Kleer, J. (1986). Problem Solving with ATMS, *Artificial Intelligence*, **28**(1).

Semantic Networks and Frames

Reference: Bratko ed. 3, section 15.7 / page 372-

Aim:

To describe semantic networks and frames. Semantic nets are a simple way of representing the relationships between entitities and concepts. Frames can do the things that semantics networks do, but take a more object-oriented type approach. Frames allow procedures called demons to be attached to their slots greatly increasing the power of this knowledge representation method.

Keywords: ako, default facet in a frame, demon, if_added demon, if_removed demon, if_replaced demon, if_needed demon, if_new demon, range demon, help demon, cache facet, multi-valued facet, facet, frame, generic frame, inheritance, instance frame, isa, procedural attachment, semantic network, slot

Plan:

- commonsense reasoning using defaults and inheritance
- semantic network examples
- quantification and semantic networks
- frames, slots, procedural attachment, demons
- `cylinder` example
- `person`, `measure`, `ph_frame` example

Logic and Rules

So far we have been dealing with rule-based systems. These have the following attributes:

- Symbolic representation
- Inference mechanism allows conclusions to be drawn from facts and rules.
- Certainty factors allow rules to deal with uncertainty. [These will be covered later.]

Common Sense Reasoning

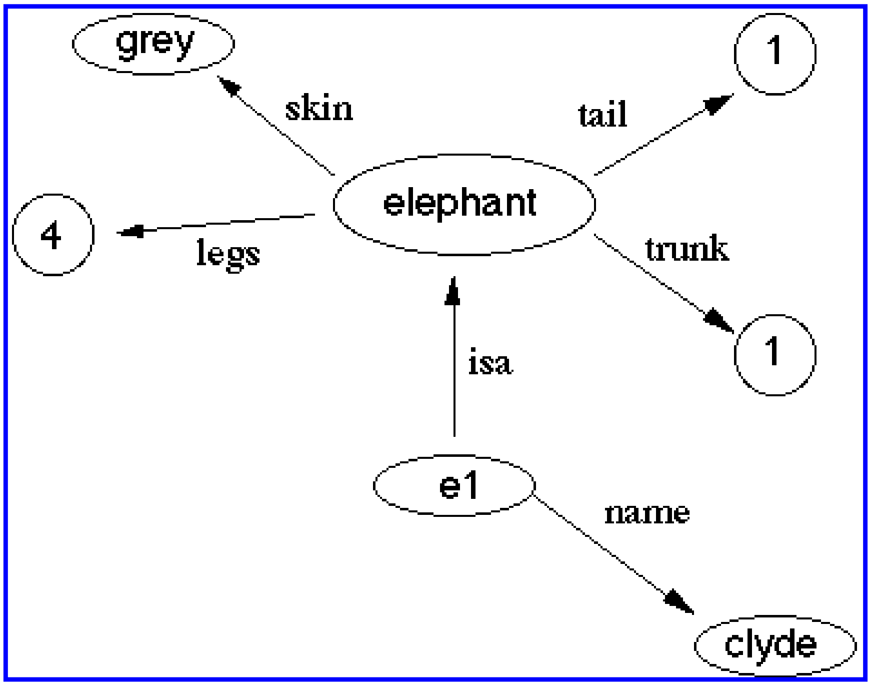
However, rules are cumbersome as a way of encoding relational knowledge and knowledge about objects.

- People have vast background knowledge to cope with everyday situations.

- We don't have to be told everything explicitly because we can call on the background knowledge.
- We use `default' knowledge to handle situations where knowledge is incomplete.
- This is called common sense reasoning.

Defaults and Inheritance

- Defaults and inheritance are ways of achieving some commonsense:
- Inheritance is a way of reasoning by default, that is, when information is missing, fall back to defaults.
- Semantic networks represent inheritance.



Using Inheritance

- To find the value of a property of *e1*, first look at *e1*.

- If the property is not attached to that node, "climb" the *isa* link to the node's parent and search there.

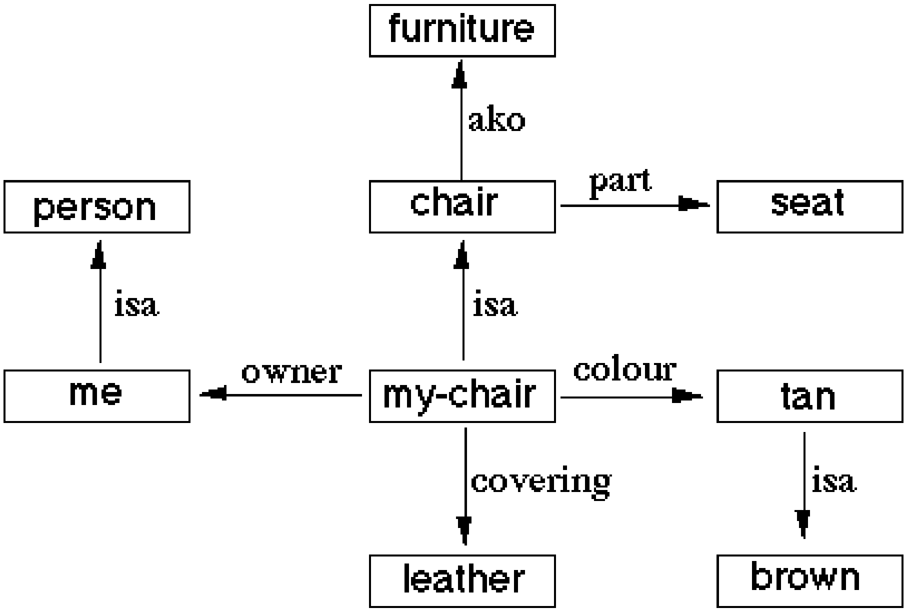
  - *isa* signifies set membership: ∈

  - *ako* signifies the subset relation: ⊆

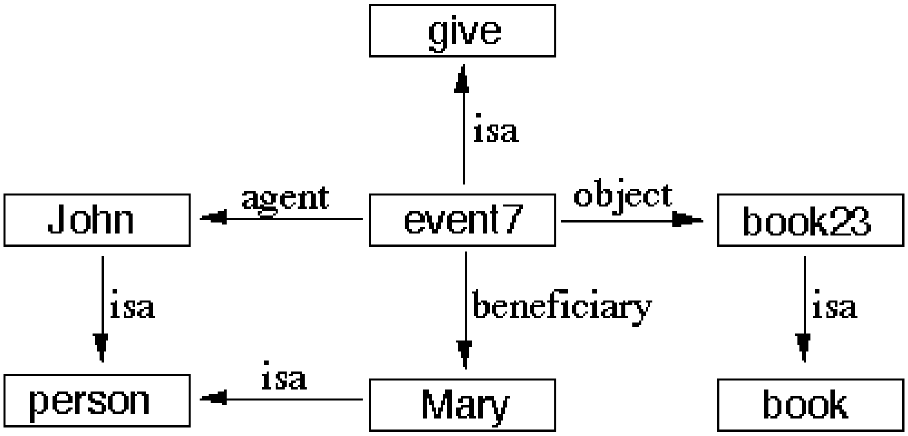
- Repeat, using *isa/ako* links, until the property is found or the inheritance hierarchy is exhausted.
- Sets of things in a semantic network are termed **types**.
- Individual objects in a semantic network are termed **instances**.
- Here is Prolog code for doing inheritance with semantic nets.

Examples of Semantic Networks

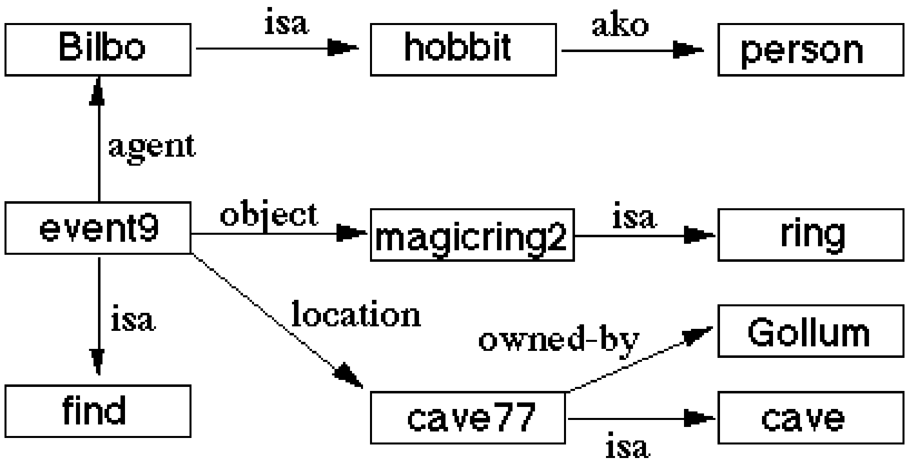
State: I own a tan leather chair.



Event: John gives the book to Mary.



Complex event:Bilbo finds the magic ring in Gollum's cave.



Frames

- Frames implement semantic networks.
- They add **procedural attachment**.
- A frame has **slots** and slots have **values**.
- A frame may be **generic**, i.e. it describes a class of objects.
- A frame may be an **instance**, i.e. it describes a particular object.
- Frames can inherit properties from generic frames.

Demons

Demons are attached to slots to cause side effects when the slot is accessed.

if_added
demons are triggered when a new value is put into a slot.

if_removed
demons are triggered when a value is removed from a slot.

if_replaced
is triggered when a slot value is replaced.

if_needed
demons are triggered when there is no value present in an instance frame and a value must be computed from a generic frame.

More Demons

if_new

is triggered when a new frame is created.

range

is triggered when a new value is added. The value must satisfy the range

help

is triggered when the range demon is triggered and returns false.

Demon-related facets

cache

means that when a value is computed it is stored in the instance frame. condition.

multi_valued

means that the slot may contain more than one value.

A Simple Frame Example

Documentation for the frame language used in this example is available.

```
cylinder ako object with
  height:
    range      number(new value) and new value > 0
    help       print("Height must be a positive number")
    if_needed  ask
    if_removed remove volume from this cylinder
    cache      yes;
  radius:
    range      number(new value) and new value > 0
    help       print("Radius must be a positive number")
    if_needed  ask
    if_removed remove cross_section from this cylinder
    cache      yes;
  cross_section:
    if_needed  pi * radius of this cylinder ^ 2
    if_removed remove volume from this cylinder
    cache      yes;
  volume:
    if_needed  cross_section of this cylinder *
```

```
                 height of this cylinder
    cache      yes!
```

A More Complicated Example

```
person ako object with
  name:
    range      atom(new value)
    help       print("The name should be a string.")
    if_new     ask
    cache      yes;
  sex:
    range      new value in [male, female]
    help       print("Sex can only be male or female, not ", new
value)
    if_needed  ask
    if_replaced print("Are you sure you want a sex change?")
    if_removed print("Are you sure you want the sex removed?")
    cache      yes;
  year_of_birth:
    range      year of current_date - 120 .. year of current_date
    help       print("Invalid year of birth.")
    if_needed  ask("Year of birth")
    cache      yes;
  age:
    cache      yes
    if_needed  year of current_date - year_of_birth of this
person;
  parents:
    multivalued yes
    range      new value must_be_a person
    help       print("The value in a parents slot must be a
person.");
  height:
    range      10..220
    help       if new value < 10 then
                  print(new value, "cm is too short."),
               if new value > 220 then
                  print(new value, "cm is too tall."),
               print("The height should be between 10 and 220cm.")
    if_needed  ask("What is the height of ", name of this person);
  weight:
    range      1..150
    if_needed  ask("What is the weight of ", name of this person)
    cache      yes
    if_added   if new value > 100 then
                  print("Your ", this slot, " is too high!");
  occupation:
    range      atom(new value)
    help       print("The occupation should be a string.")
    if_needed  ask("What is the occupation")
    cache      yes
    if_removed print("I used to be a ", old value, ".")!

measure ako object with
  current_value:
    range      allowable_low of this measure .. allowable_high of
this measure
    help       print("The patient is dead!")
    if_new     ask(prompt of this measure)
    cache      yes
    if_added
        if      new value < expected_low of this measure then
                replace interpretation of this measure by low
        else if new value > expected_high of this measure then
                replace interpretation of this measure by high
        else    replace interpretation of this measure by normal
    if_replaced replace last_value of this measure by old value!
```

Comment: The first reference to interpretation of this measure has the effect of dynamically adding a slot to the measure frame. Similarly, references to low , normal , and high dynamically add these as possible values.
end of comment

```
ph_frame ako measure with
    prompt:            default "ph level";
    allowable_low:     default 6;
    allowable_high:    default 8;
    expected_high:     default 7.6;
    expected_low:      default 6.5!
```

```
'HCO3_frame' ako measure with
    prompt:             default "HCO3 level";
    allowable_low:      default 6;
    allowable_high:     default 8;
    expected_high:      default 7.6;
    expected_low:       default 6.5!

paCO2_frame ako measure with
    prompt:             default "paCO2 level";
    allowable_low:      default 6;
    allowable_high:     default 8;
    expected_high:      default 7.6;
    expected_low:       default 6.5!

patient ako person with
  ph:
    cache               yes
    if_needed   make [ph_frame];
  'HCO3':
    cache               yes
    if_needed   make ['HCO3_frame'];
  paCO2:
    cache               yes
    if_needed   make [paCO2_frame];
  diagnosis:
    multivalued yes
    default     [];
  investigation:
    if_new
        if      interpretation of ph of this patient = low
        then    add acidosis to diagnosis of this patient,

        if      interpretation of ph of this patient = high
        then    add alkalosis to diagnosis of this patient,

        if      interpretation of paCO2 of this patient = low
        then    add hypocarbic to diagnosis of this patient,

        if      interpretation of paCO2 of this patient = high
        then    add hypercarbic to diagnosis of this patient,
        if      acidosis in diagnosis of this patient
        and     interpretation of 'HCO3' of this patient = low
        then    add primary_metabolic_acidosis
                        to diagnosis of this patient!

current_date isa object with
    year:       2005;
    month:      4;
    day:        1!
```

Summary: Semantic Networks and Frames
Semantic networks start out by encoding relationships like isa and ako but can represent quite complicated information on this simple basis. Frames organize knowledge around concepts considered to be of interest (like person and patient in the example code). The also allow procedural attachment - that is, demons can be attached to slots so that the mere fact of creating a frame or accessing a slot can cause significant computation to be performed.

Problem Solving in Prolog

Reference: Bratko chapter 12

Aim:

To illustrate search in AI using a fairly well-known example problem. We also briefly introduce a number of different methods for exploring a state space (or any other graph to be searched).

Keywords: breadth first search, depth first search, edge in a graph, goal state, graph, directed acyclic graphs, trees, binary trees, adjacency matrices, graph search algorithms, initial state, node, operator, state, initial state, goal state, path, search, vertex

Plan:

- states, operators and searching
- representing state spaces using graphs
- finding a path from A to B in a graph
- missionaries & cannibals: representing states & operators
- methods of search: depth-first, breadth-first

Problem solving has traditionally been one of the key areas of concern for Artificial Intelligence. Below, we present a common problem and demonstrate a simple solution.

Missionaries and Cannibals

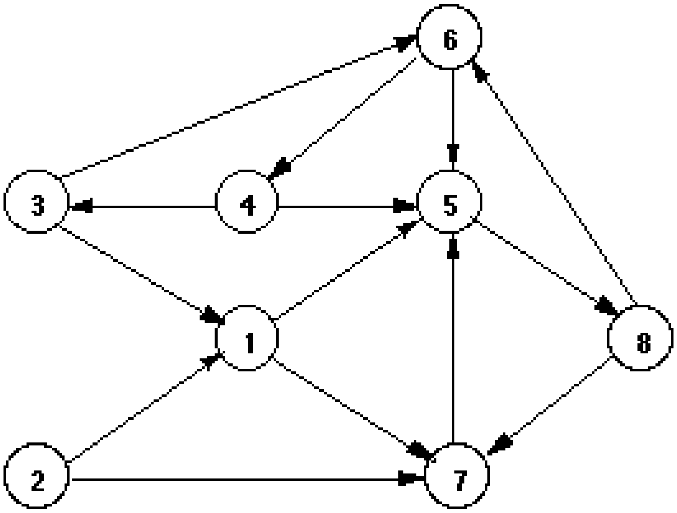
- There are three missionaries and three cannibals on the left bank of a river.
- They wish to cross over to the right bank using a boat that can only carry two at a time.
- The number of cannibals on either bank must never exceed the number of missionaries on the same bank, otherwise the missionaries will become the cannibals' dinner!
- Plan a sequence of crossings that will take everyone safely accross.

This kind of problem is often solved by a graph search method. We represent the problem as a set of

states
    which are snapshots of the world and
operators
    which transform one state into another

States can be mapped to nodes of a graph and operators are the edges of the graph. Before studying the missionaries and cannibals problem, we look at a simple graph search algorithm in Prolog. the missionaries and cannibals program will have the same basic structure.

Graph Representation



A graph may be represented by a set of edge predicates and a list of vertices.

```
edge(1, 5).     edge(1, 7).
edge(2, 1).     edge(2, 7).
edge(3, 1).     edge(3, 6).
edge(4, 3).     edge(4, 5).
edge(5, 8).
edge(6, 4).     edge(6, 5).
edge(7, 5).
edge(8, 6).     edge(8, 7).

vertices([1, 2, 3, 4, 5, 6, 7, 8]).
```

Finding a path

- Write a program to find path from one node to another.
- Must avoid cycles (i.e. going around in circle).
- A template for the clause is:

```
path(Start, Finish, Visited, Path).
```

Start
: is the name of the starting node

Finish
: is the name of the finishing node

Visited
: is the list of nodes already visited.

Path

is the list of nodes on the path, including **Start** and **Finish**.

The Path Program

- The search for a path terminates when we have nowhere to go.

```
path(Node, Node, _, [Node]).
```

- A path from Start to Finish starts with a node, X, connected to Start followed by a path from X to Finish.

```
path(Start, Finish, Visited, [Start | Path]) :-
        edge(Start, X),
        not(member(X, Visited)),
        path(X, Finish, [X | Visited], Path).
```

Here is an example of the `path` algorithm in action.

Representing the state

Now we return to the problem of representing the missionaries anc cannibals problem:

- A state is one "snapshot" in time.
- For this problem, the only information we need to fully characterise the state is:
  - the number of missionaries on the left bank,
  - the number of cannibals on the left bank,
  - the side the boat is on.

  All other information can be deduced from these three items.
- In Prolog, the state can be represented by a 3-arity term,

```
state(Missionaries, Cannibals, Side)
```

Representing the Solution

- The solution consists of a list of moves, e.g.

```
[move(1, 1, right), move(2, 0, left)]
```

- We take this to mean that 1 missionary and 1 cannibal moved to the right bank, then 2 missinaries moved to the left bank.
- Like the graph search problem, we must avoid returning to a state we have visited before.

- The visited list will have the form:

```
    [MostRecent_State | ListOfPreviousStates]
```

Overview of Solution

- We follow a simple graph search procedure:
    - Start from an initial state
    - Find a neighbouring state
    - Check that the new state has not been visited before
    - Find a path from the neighbour to the goal.

  The search terminates whe we have found the state:

```
    state(0, 0, right).
```

Top-level Prolog Code

```
% mandc(CurrentState, Visited, Path)

mandc(state(0, 0, right), _, []).
mandc(CurrentState, Visited, [Move | RestOfMoves]) :-
        newstate(CurrentState, NextState),
        not(member(NextState, Visited)),
        make_move(CurrentState, NextState, Move),
        mandc(NextState, [NextState | Visited], RestOfMoves]).

make_move(state(M1, C1, left), state(M2, C2, right), move(M, C, right)) :-
        M is M1 - M2,
        C is C1 - C2.
make_move(state(M1, C1, right), state(M2, C2, left), move(M, C, left)) :-
        M is M2 - M1,
        C is C2 - C1.
```

Possible Moves

- A move is characterised by the number of missionaries and the number of cannibals taken in the boat at one time.
- Since the boat can carry no more than two people at once, the only possible combinations are:

```
    carry(2, 0).
    carry(1, 0).
    carry(1, 1).
```

```
        carry(0, 1).
        carry(0, 2).
```

- where `carry(M, C)` means the boat will carry `M` missionaries and `C` cannibals on one trip.

Feasible Moves

- Once we have found a possible move, we have to confirm that it is feasible.
- I.e. it is not feasible to move more missionaries or more cannibals than are present on one bank.
- When the state is `state(M1, C1, left)` and we try `carry(M, C)` then

```
        M <= M1 and C <= C1
```

- must be true.
- When the state is `state(M1, C1, right)` and we try `carry(M, C)` then

```
        M + M1 <= 3 and C + C1 <= 3
```

- must be true.

Legal Moves

- Once we have found a feasible move, we must check that is legal.
- I.e. no missionaries must be eaten.

```
        legal(X, X).
        legal(3, X).
        legal(0, X).
```

- The only safe combinations are when there are equal numbers of missionaries and cannibals or all the missionaries are on one side.

Generating the next state

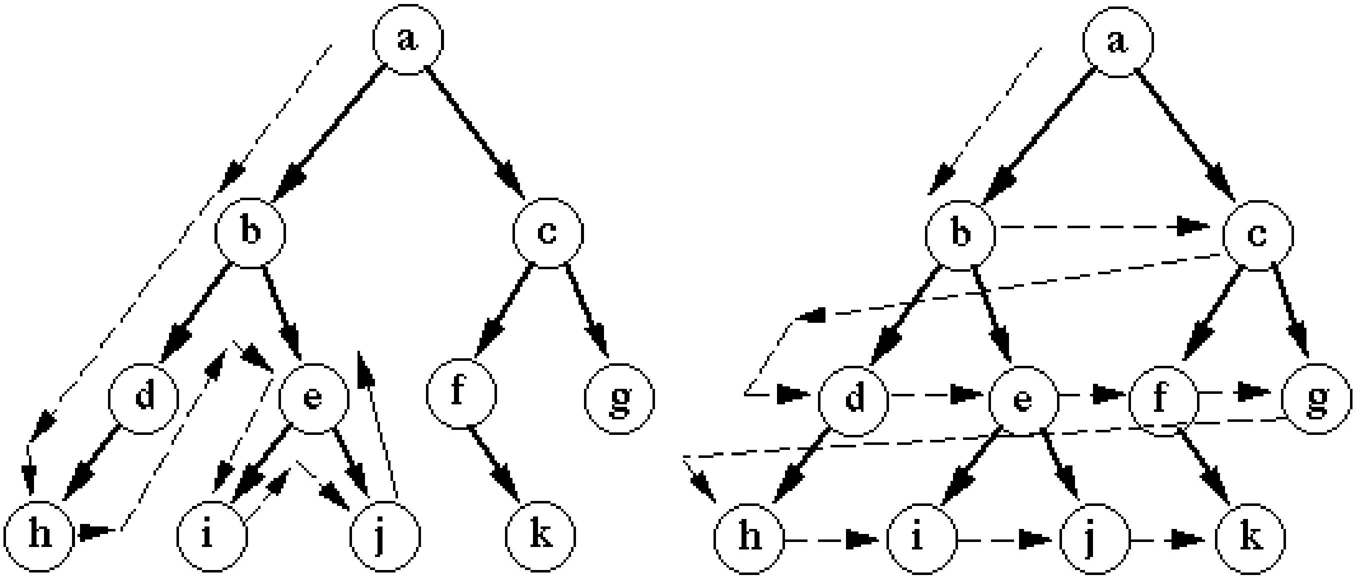
```
newstate(state(M1, C1, left), state(M2, C2, right)) :-
        carry(M, C),
        M <= M1,
        C <= C1,
        M2 is M1 - M,
        C2 is C1 - C,
        legal(M2, C2).
```

```
newstate(state(M1, C1, right), state(M2, C2, left)) :-
        carry(M, C),
        M2 is M1 + M,
        C2 is C1 + C,
        M2 <= 3,
        C2 <= 3,
        legal(M2, C2).
```

The complete code, with instructions for use, is available at
http://www.cse.unsw.edu.au/~billw/cs9414/notes/mandc/mandc.pro

Methods of Search

In the preceding example, the state space is explored in an order determined by Prolog. In some situations, it might be necessary to alter that order of search in order to make search more efficient. To see what this might mean, here are two alternative methods of searching a tree.



Depth-first search

Breadth-first search

Depth first search begins by diving down as quickly as possible to the leaf nodes of the tree. Traversal can be done by:

- visiting the node first, then its children (pre-order traversal):
  a b d h e i j c f k g
- visiting the children first, then the node (post-order traversal):
  h d i j e b k f g c a
- visiting some of the children, then the node, then the other children (in-order traversal)
  h d b i e j a f k c g

There are many other search methods and variants on search methods. We do not have time to cover these in

COMP9414, but you can find out about some of them in the text by Bratko. For example, chapter 12 deals with **best-first search**.

Summary: Problem Solving and Search in AI

We introduced the concepts of states and operators and gave a graph traversal algorithm that can be used as a problem solving tool. We applied this to solve the "missionaries and cannibals" problem.
We also outlined depth-first search, breadth-first search, and alluded to the existence of a range of other search methods.

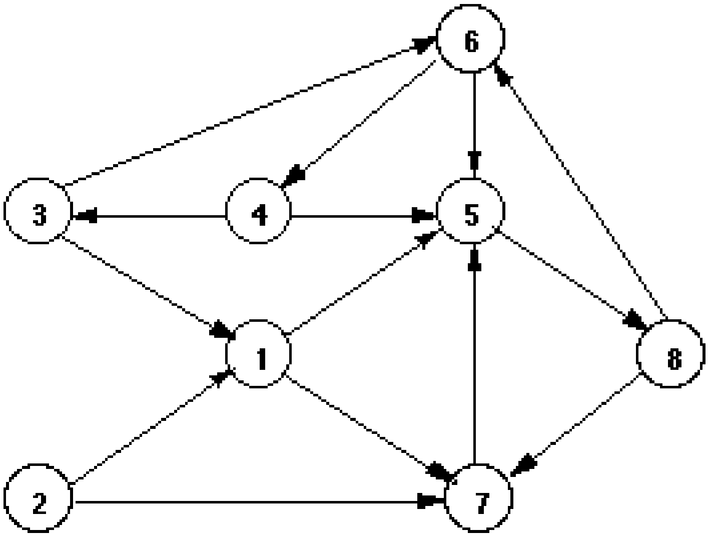
Example of execution of graph search algorithm

Below is a trace of the execution of the query

```
?- path(1, 6, [1], Path).
```

using the Prolog code for `path` given in lectures, and the `edge`s given in lectures, that is:

```
path(Node, Node, _, [Node]).                      % rule 1
path(Start, Finish, Visited, [Start | Path]) :-   % rule 2
     edge(Start, X),
     not(member(X, Visited)),
     path(X, Finish, [X | Visited], Path).

edge(1, 5).     edge(1, 7).
edge(2, 1).     edge(2, 7).
edge(3, 1).     edge(3, 6).
edge(4, 3).     edge(4, 5).
edge(5, 8).
edge(6, 4).     edge(6, 5).
edge(7, 5).
edge(8, 6).     edge(8, 7).
```



The trace was produced using the SWI Prolog trace facility. Comments on what is happening are shown at right.

```
?- trace.
```

```
?- path(1, 6, [1], Path).
```

`  Call: (7)  path(1, 6, [1], _G289)`	original query
`  Call: (8)  edge(1, _L208)`	try to match first goal in body of rule 2
`  Exit: (8)  edge(1, 5)`	... and succeed, with _L208 = 5
`^ Call: (8)  not(member(5, [1]))`	second goal of body of rule 1
`  Call: (9)  lists:member(5, [1])`	
`  Call: (10) lists:member(5, [])`	
`  Fail: (10) lists:member(5, [])`	
`  Fail: (9)  lists:member(5, [1])`	member fails
`^ Exit: (8)  not(member(5, [1]))`	so not(member...) succeeds
`  Call: (8)  path(5, 6, [5, 1], _G344)`	third goal in body of rule 2 - recursively look for a path from 5 to 6
`  Call: (9)  edge(5, _L246)`	look for an edge leaving 5
`  Exit: (9)  edge(5, 8)`	and succeed: _L246 = 8
`^ Call: (9)  not(member(8, [5, 1]))`	second goal as before
`  Call: (10) lists:member(8, [5, 1])`	
`  Call: (11) lists:member(8, [1])`	
`  Call: (12) lists:member(8, [])`	
`  Fail: (12) lists:member(8, [])`	

```
  Fail: (11) lists:member(8, [1])

  Fail: (10) lists:member(8, [5, 1])

^ Exit: (9)  not(member(8, [5, 1]))

  Call: (9)  path(8, 6, [8, 5, 1], _G353)        again recursively call path to look
                                                  for a path from 8 to 6.
  Call: (10) edge(8, _L267)
                                                  first goal of third invocation of path

  Exit: (10) edge(8, 6)
                                                  first goal succeeds

  Call: (10) not(member(6, [8, 5, 1]))

  ...

^ Exit: (10) not(member(6, [8, 5, 1]))
                                                  second goal succeeds

  Call: (10) path(6, 6, [6, 8, 5, 1], _G362)     fourth call to path, looking for
                                                  a path from 6 to 6...
  Exit: (10) path(6, 6, [6, 8, 5, 1], [6])       this time rule 1 applies, & succeeds at
                                                  once
  Call: (9)  path(8, 6, [8, 5, 1], [8, 6])       which means that the third invocation
                                                  of path succeeds...
  Call: (8)  path(8, 6, [5, 1], [5, 8, 6])       which means that the second
                                                  invocation
                                                  of path succeeds...
  Call: (7)  path(8, 6, [1], [1, 5, 8, 6])       and so the first invocation
                                                  of path succeeds...

Path = [1, 5, 8, 6]

Yes
```

To turn off tracing, type `notrace!` at the prompt (?-).

Machine Learning

In 1956, Bruner, Goodnow and Austin published their book *A Study of Thinking*, which became a landmark in psychology and would later have a major impact on machine learning. The experiments reported by Bruner, Goodnow and Austin were directed towards understanding a human's ability to categorise and how categories are learned.

> We begin with what seems a paradox. The world of experience of any normal man is composed of a tremendous array of discriminably different objects, events, people, impressions...But were we to utilize fully our capacity for registering the differences in things and to respond to each event encountered as unique, we would soon be overwhelmed by the complexity of our environment... The resolution of this seeming paradox ... is achieved by man's capacity to categorize. To categorise is to render discriminably different things equivalent, to group objects and events and people around us into classes... The process of categorizing involves ... an act of invention... If we have learned the class "house" as a concept, new exemplars can be readily recognised. The category becomes a tool for further use. The learning and utilization of categories represents one of the most elementary and general forms of cognition by which man adjusts to his environment.

The first question that they had to deal with was that of representation: what is a concept? They assumed that objects and events could be described by a set of attributes and were concerned with how inferences could be drawn from attributes to class membership. Categories were considered to be of three types: conjunctive, disjunctive and relational.

> ...when one learns to categorise a subset of events in a certain way, one is doing more than simply learning to recognise instances encountered. One is also learning a rule that may be applied to new instances. The concept or category is basically, this "rule of grouping" and it is such rules that one constructs in forming and attaining concepts.

The notion of a rule as an abstract representation of a concept in the human mind came to be questioned by psychologists and there is still no good theory to explain how we store concepts. However, the same questions about the nature of representation arise in machine learning, for the choice of representation heavily determines the nature of a learning algorithm. Thus, one critical point of comparison among machine learning algorithms is the method of knowledge representation employed.

In this section we will do a quick tour of a variety of learning algorithms and seen how the method of representing knowledge is crucial in the following ways:

- Knowledge representation determines the concepts that an algorithm can and cannot learn.
- Knowledge representation affects the speed of learning. Some representations lend themselves to

more efficient implementation than others. Also, the more expressive the language, the larger is the search space.

- Knowledge representation determines the readability of the concept description. A representation that is opaque to the user may allow the program to learn, but a representation that is transparent also allows the user to learn.

BIBLIOGRAPHY

Aha, D. W., Kibler, D., & Albert, M. K. (1991). Instance-Based Learning Algorithms. *Machine Learning*, **6**(1), 37-66.

Banerji, R. B. (1980). *Artificial Intelligence: A Theoretical Approach*. New York: North Holland.

Bruner, J. S., Goodnow, J. J., & Austin, G. A. (1956). *A Study of Thinking*. New York: Wiley.

Buntine, W. (1988). Generalized Subsumption and its Applications to Induction and Redundancy. *Artificial Intelligence*, **36**, 149-176.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. Ann Arbor, Michigan: University of Michigan Press.

King, R. D., Lewis, R. A., Muggleton, S., & Sternberg, M. J. E. (1992). Drug design by machine learning: the use of inductive logic programming to model the structure-activity relationship of trimethoprim analogues binding to dihydrofolate reductase. *Proceedings of the National Academy Science*, **89**.

Michalski, R. S. (1973). Discovering Classification Rules Using Variable Valued Logic System VL1. In *Third International Joint Conference on Artificial Intelligence*. (pp. 162-172).

Michalski, R. S. (1983). A Theory and Methodology of Inductive Learning. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach*. Palo Alto: Tioga.

Michie, D., & Chambers, R. A. (1968). Boxes: An Experiment in Adaptive Control. In E. Dale & D. Michie (Eds.), *Machine Intelligence 2*. Edinburgh: Oliver and Boyd.

Muggleton, S., & Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. In R. S. Michalski, T. M. Mitchell, & J. G. Carbonell (Eds.), *Proceedings of the Fifth International Machine Learning Conference*. (pp. 339-352). Ann Arbor, Michigan: Morgan Kaufmann.

Muggleton, S., & Feng, C. (1990). Efficient induction of logic programs. In *First Conference on*

Algorithmic Learning Theory, Tokyo: Omsha.

Odetayo, M. (1988) Genetic Algorithms for Control a Dynamic Physical System. M.Sc. Thesis, Strathclyde University.

Plotkin, G. D. (1970). A Note on Inductive Generalization. In B. Meltzer & D. Michie (Eds.), *Machine Intelligence 5*. (pp. 153-163). Edinburgh University Press.

Quinlan, J. R. (1979). Discovering rules by induction from large collections of examples. In D. Michie (Eds.), *Expert Systems in the Micro-Electronic Age*. Edinburgh: Edinburgh University Press.

Quinlan, J. R. (1987). Generating production rules from decision trees. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*. (pp. 304-307). San Mateo, CA: Morgan Kaufmann.

Quinlan, J. R. (1990). Learning Logical Definitions from Relations. *Machine Learning*, **5**, 239-266.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. San Mateo, CA: Morgan Kaufmann.

Reynolds, J. C. (1970). Transformational Systems and the Algebraic Structure of Atomic Formulas. In B. Meltzer & D. Michie (Eds.), *Machine Intelligence 5*. (pp. 153-163).

Robinson, J. A. (1965). A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, **12**(1), 23-41.

Sammut, C. A., & Banerji, R. B. (1986). Learning Concepts by Asking Questions. In R. S. Michalski Carbonell, J.G. and Mitchell, T.M. (Eds.), *Machine Learning: An Artificial Intelligence Approach, Vol 2*. (pp. 167-192). Los Altos, California: Morgan Kaufmann.

Shapiro, E. Y. (1981). *Inductive Inference of Theories From Facts* (Technical Report No. 192). Yale University.

Learning, Measurement and Representation

Reference: Bratko chapter 18

Aim:

To provide a framework and some general terminology for discussing some of the issues in machine learning.

Keywords: connectionism, function approximation algorithms, hypothesis language, learning program, machine learning, noisy data in machine learning, observation language, supervised learning, symbolic learning algorithms, unsupervised learning, classes in classification tasks

Plan:

- programs that learn
- observation and hypothesis languages
- symbolic learning algorithms
- function approximation algorithms

Learning

A learning program is one that is capable of improving its performance through experience. Given a program, *P*, and some input, *x*, a normal program would yield the same result

$$P(x) = y$$

after every application. However, a *learning* program can alter its *initial state q* so that its performance is modified with each application. Thus, we can say

$$P(x \mid q) = y$$

That is, *y* is the result of applying program *P* to input, *x*, given the initial state, *q*.

The goal of learning is to construct a new "initial" state, *q'*, so that the program alters its behaviour to give a more accurate or quicker result. In the case of **supervised** learning, to construct *q'*, one needs a set of inputs *xi* and corresponding target outputs *zi* (i.e. you want P(*xi*) = *zi* when learning is complete.)

$$L(P, q, ((x1,z1),\ldots ,(xn,zn))) = q'$$

Thus, one way of thinking about what a learning program does is that it builds an increasingly accurate approximation to a mapping from input to output: *P(xi* | q') will be closer to *zi* than *P(xi* | q) was.

With **unsupervised** learning the program tries to find regularities and interesting patterns in the data without ever being told what the "right" answer is.

Categorisation problem

The most common learning task is that of acquiring a function which maps objects, that share common properties, to the same class value. This is the *categorisation problem*.

Observation & Hypothesis Languages

The program must be able to represent its observations of the world, and it must also be able to represent hypotheses about the patterns it may find in those observations. Thus, we may sometimes refer to the *observation language* and *hypothesis language*. The observation language describes the inputs and outputs of the program and the hypothesis language describes the internal state of the learning program, which corresponds to its theory of the concepts or patterns that exist in the data.

The input to a learning program consists of descriptions of objects from the universe and, in the case of supervised learning, an output value associated with the example. The universe can be an abstract one, such as the set of all natural numbers, or the universe may be a subset of the real world.

Measurement

No matter which method of representation we choose, descriptions of objects in the real world must ultimately rely on measurements of some properties or attributes of those objects. These may be physical properties such as size, weight, colour, etc or they may be defined for objects, e.g. the length of time a person has been employed, for the purpose of approving a loan. The accuracy and reliability of a learned concept depends heavily on the accuracy and reliability of the measurements.

Representation

A program is limited in the concepts that it can learn by the representational capabilities of both observation and hypothesis languages. For example, if an attribute/value list is used to represent examples for an induction program, the measurement of certain attributes and not others clearly places bounds on the kinds of patterns that the learner can find. The learner is said to be *biased* by its observation language. The hypothesis language also constrains what may be learned.

****

We will divide our attention between two different classes of machine learning algorithms that use distinctly different approaches to the problem of representation:

Symbolic learning algorithms

learn concepts by constructing a symbolic expression that describes a class of objects. We consider algorithms that work with representations equivalent to propositional / first-order logic (starting with **ID3**, an inductive decision tree system).

Function approximation algorithms

include connectionist and statistical methods. These algorithms are most closely related to traditional mathematical notions of approximation and interpolation and represent concepts as mathematical formulae. Our main focus here will be on the **backpropagation** learning algorithm for feedforward neural networks.

Evaluation

Learning algorithms can be assessed in terms of:

* speed of learning

* speed of classification of the resulting function

* degree of generalization ability (do they get the right answer for cases that were not in their training sets)

Summary: Learning, Measurement and Representation

We have discussed the characteristics of a program that learns, distinguishing between *supervised* and *unsupervised* learning. We introduced the concepts of *observation language* and *hypothesis language*, discussed measurement and representation, and made the distinction between *symbolic learning algorithms* and *function approximation algorithms*.
Learning algorithms may be evaluated in terms of speed of learning, speed of computation once learning has occurred, and degree of generalization ability.

Prototypes

The simplest form of learning is memorisation. When an object is observed or the solution to a problem is found, it is stored in memory for future use. Memory can be thought of as a look up table. When a new problem is encountered, memory is searched to find if the same problem has been solved before. If an exact match for the search is required, learning is slow and consumes very large amounts of memory. However, approximate matching allows a degree of generalisation that both speeds learning and saves memory.

For example, if we are shown an object and we want to know if it is a chair, then we compare the description of this new object with descriptions of `typical' chairs that we have encountered before. If the description of the new object is `close' to the description of one of the stored instances then we may call it a chair. Obviously, we must defined what we mean by `typical' and `close'.

To better understand the issues involved in learning prototypes, we will briefly describe three experiments in *Instance-based learning* (IBL) by Aha, Kibler and Albert (1991). IBL learns to classify objects by being shown examples of objects, described by an attribute/value list, along with the class to which each example belongs.

EXPERIMENT 1

In the first experiment (IB1), to learn a concept simply required the program to store every example. When an unclassified object was presented for classification by the program, it used a simple Euclidean distance measure to determine the nearest neighbour of the object and the class given to it was the class of the neighbour.

This simple scheme works well, and is tolerant to some noise in the data. Its major disadvantage is that it requires a large amount of storage capacity.

EXPERIMENT 2

The second experiment (IB2) attempted to improve the space performance of IB1. In this case, when new instances of classes were presented to the program, the program attempted to classify them. Instances that were correctly classified were ignored and only incorrectly classified instances were stored to become part of the concept.

While this scheme reduced storage dramatically, it was less noise-tolerant than the first.

EXPERIMENT 3

The third experiment (IB3) used a more sophisticated method for evaluating instances to decide if they should be kept or not. IB3 is similar to IB2 with the following additions. IB3 maintains a record of the number of correct and incorrect classification attempts for each saved instance. This record summarised an instance's classification performance. IB3 uses a significance test to determine which instances are good classifiers and which ones are believed to be noisy. The latter are discarded from the concept description. This method strengthens noise tolerance, while keeping storage requirements down.

DISCUSSION

Figure 1 shows the boundaries of an imaginary concept in a two dimensions space. The dashed lines represent the boundaries of the target concept. The learning procedure attempts to approximate these boundaries by nearest neighbour matches. Note that the boundaries defined by the matching procedure are quite irregular. This can have its advantages when the target concept does not have a regular shape.

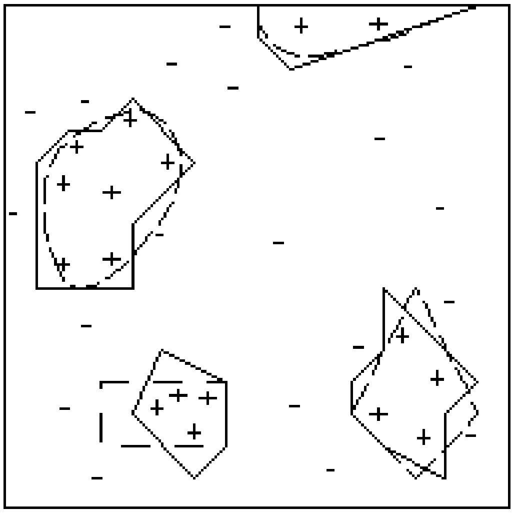


Figure 1. The extension of an IBL concept is shown in solid lines. The dashed lines represent the target concept. A sample of positive and negative examples is shown. Adapted from Aha, Kibler and Albert (1991).

Learning by remembering typical examples of a concept has several other advantages. If an efficient indexing mechanism can be devised to find near matches, this representation can be very fast as a classifier since it reduces to a table look up. It does not require any sophisticated reasoning system and is very flexible. As we shall see later, representations that rely on abstractions of concepts can run into trouble with what appear to be simple concepts. For example, an abstract representation of a chair may consist of a description of the number legs, the height, etc. However, exceptions abound since anything that can be sat on can be thought of as a chair. Thus, abstractions must often be augmented by lists of exceptions. Instance-based representation does not suffer from this problem since it only consists exceptions and is designed to handle them efficiently.

One of the major disadvantages of this style of representation is that it is necessary to define a similarity metric for objects in the universe. This can often be difficult to do when the objects are quite complex.

Another disadvantage is that the representation is not human readable. What does a collection of typical instances tell us about the concept that has been learned?

Function approximation

Statistical and connectionist approaches to machine learning are related to function approximation methods in mathematics. For the purposes of illustration let us assume that the learning task is one of classification. That is, we wish to find ways of grouping objects in a universe. In Figure 2 we have a universe of objects that belong to either of two classes `+' or `-'.

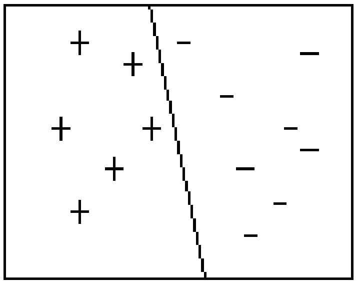


Figure 2: A linear discrimination between two classes

By function approximation, we describe a surface that separates the objects into different regions. The simplest function is that of a line and linear regression methods and perceptrons are used to find linear discriminant functions.

A perceptron is a simple pattern classifier. Given a binary input vector, **x**, a weight vector, **w**, and a threshold value, *T*, if,

$$\sum_i w_i \times x_i > T$$

then the output is 1, indicating membership of a class, otherwise it is 0, indicating exclusion from the class. Clearly, $\mathbf{w} \cdot \mathbf{x} - T$ describes a hyperplane and the goal of perceptron learning is to find a weight vector, **w**, that results in correct classification for all training examples. The perceptron learning algorithm is quite straight forward. All the elements of the weight vector are initially set to 0. For each training example, if the perceptron outputs 0 when it should output 1 then add the input vector to the weight vector; if the perceptron outputs 1 when it should output 0 then subtract the input vector to the weight vector; otherwise, do nothing. This is repeated until the perceptron yields the correct result for each training example. The algorithm has the effect of reducing the error between the actual and desired output.

The perceptron is an example of a *linear threshold unit* (LTU). A single LTU can only recognise one kind of pattern, provided that the input space is linearly separable. If we wish to recognise more than one pattern, several LTU's can be combined. In this case, instead of having a vector of weights, we have an array. The output will now be a vector:

$$\mathbf{u} = \mathbf{W}\mathbf{x}$$

where each element of **u** indicates membership of a class and each row in **W** is the set of weights for one LTU. This architecture is called a *pattern associator*.

LTU's can only produce linear discriminant functions and consequently, they are limited in the kinds of classes that can be learned. However, it was found that by cascading pattern associators, it is possible to approximate decision surfaces that are of a higher order than simple hyperplanes. In cascaded system, the outputs of one pattern associator are fed into the inputs of another, thus:

$$\mathbf{u} = \mathbf{W}(\mathbf{V}\mathbf{x})$$

This is the scheme that is followed by multi-layer neural nets (Figure 3).

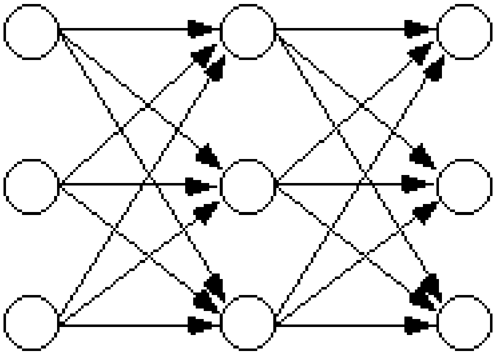


Figure 3. A multi-layer network.

To facilitate learning, a further modification must be made. Rather than using a simple threshold, as in the perceptron, multi-layer networks usually use a non-linear threshold such a sigmoid function, such as

$$\frac{1}{1+e^{-x}}$$

Like perceptron learning, back-propagation attempts to reduce the errors between the output of the network and the desired result. However, assigning blame for errors to hidden units (ie. nodes in the intermediate layers), is not so straightforward. The error of the output units must be propagated back through the hidden units. The contribution that a hidden unit makes to an output unit is related strength of the weight on the link between the two units and the level of activation of the hidden unit when the output unit was given the wrong level of activation. This can be used to estimate the error value for a hidden unit in the penultimate layer, and that can, in turn, be used in make error estimates for earlier layers.

Despite the non-linear threshold, multi-layer networks can still be thought of as describing a complex collection of hyperplanes that approximate the required decision surface.

DISCUSSION

Function approximation methods, such as the ones discussed above, can often produce quite accurate classifiers because they are capable of construction complex decision surfaces. However, knowledge is stored as weights in a matrix. Thus, the results of learning are not easily available for inspection by a human reader. Moreover, the design of a network usually requires informed guesswork on the part of the user in order to obtain satisfactory results. Although some effort has been devoted to extracting meaning from networks, the still communicate little about the data.

Connectionist learning algorithms are still computationally expensive. A critical factor in their speed is the encoding of the inputs to the network. This is also critical to genetic algorithms and we will illustrate that problem in the next section.

CRICOS Provider Code No. 00098G

Induction of Decision Trees

Reference: Bratko sections 18.5, 18.6

Aim:

To describe an algorithm whose input is a collection of instances and their correct classification and whose output is a decision tree that can be used to classify each instance.

Keywords: attributes, backed-up error estimate, C4.5, C5, concept learning system (CLS), decision trees, entropy, expected error estimate, feature, ID3, instances, Laplace error estimate, pruning decision trees, splitting criterion in ID3, tree induction algorithm, windowing in ID3

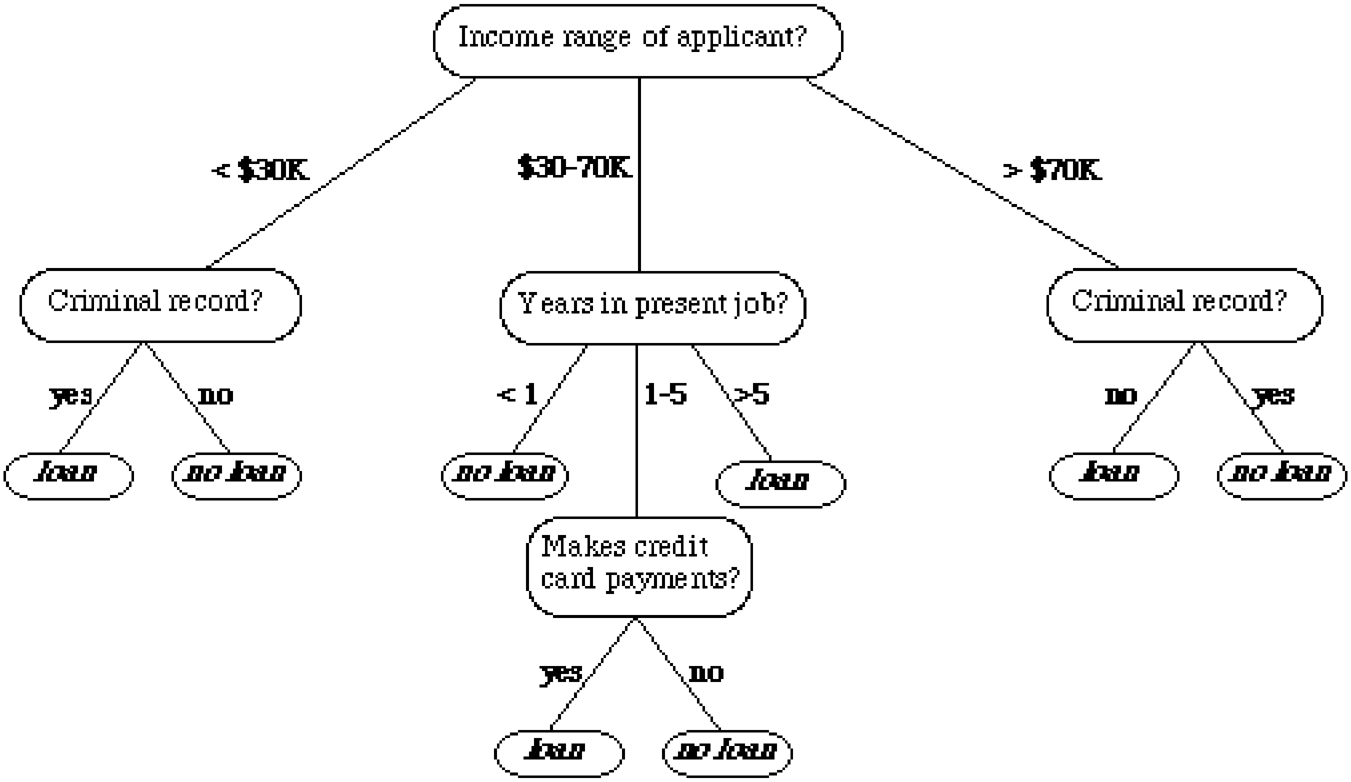
Plan:

- What is a decision tree?
- Building a decision tree
- Which attribute should we split on?
- Information theory and the splitting criterion
- ID3 example & ID3 symbolic version
- ID3 in iProlog
- ID3 enhancements: windowing
- Dealing with noisy data - expected error pruning

Decision Trees

- A decision tree is a tree in which each branch node represents a choice between a number of alternatives, and each leaf node represents a classification or *decision*.

- For example, we might have a decision tree to help a financial institution decide whether a person should be offered a loan:



- We wish to be able to induce a decision tree from a set of data about instances together with the decisions or classifications for those instances.

Example Instance Data

```
size:   small medium large
colour: red blue green
shape:  brick wedge sphere pillar

%% yes

medium  blue    brick
small   red     sphere
large   green   pillar
large   green   sphere

%% no

small   red     wedge
large   red     wedge
large   red     pillar
```

- In this example, there are 7 *instances*, described in terms of three *features* or *attributes* (size, colour, and shape), and the instances are classified into two *classes* `%% yes` and `%% no`.

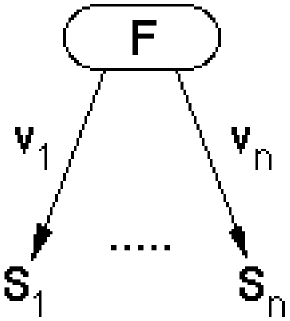
- We shall now describe an algorithm for inducing a decision tree from such a collection of classified instances.

- Originally termed CLS (Concept Learning System) it has been successively enhanced.

- At the highest level of enhancement that we shall describe in these notes, the system is known as **ID3 -** later versions include C4, C4.5 and See5/C5.0 (latest version release 1.20, May 2004).

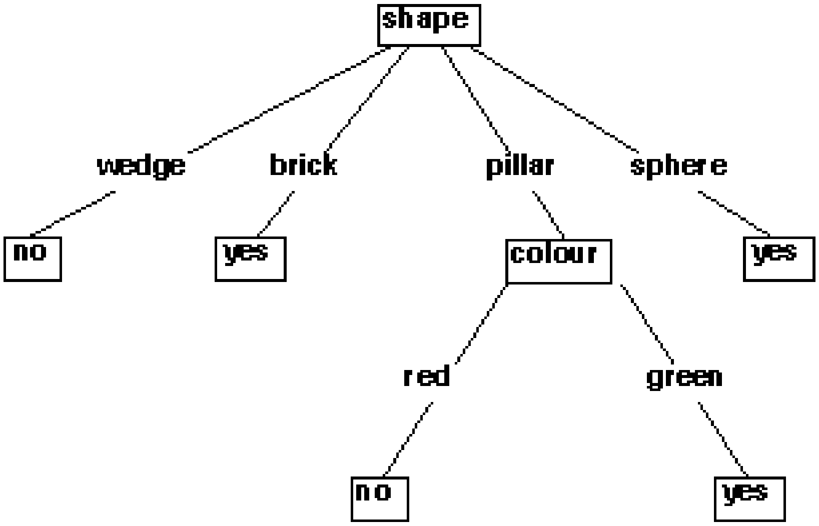
- ID3 and its successors have been developed by Ross Quinlan.

Tree Induction Algorithm



- The algorithm operates over a set of training instances, *C*.
- If all instances in *C* are in class *P*, create a node *P* and stop, otherwise select a *feature* or *attribute* **F** and create a decision node.
- Partition the training instances in *C* into subsets according to the values of *V*.
- Apply the algorithm recursively to each of the subsets *C*.

Output of Tree Induction Algorithm



This can easily be expressed as a nested if-statement

```
if (shape == wedge)
        return no;
if (shape == brick)
        return yes;
if (shape == pillar)
{
        if (colour == red)
                return no;
        if (colour == green)
                return yes;
}
if (shape == sphere)
        return yes;
```

Choosing Attributes and ID3

- The order in which attributes are chosen determines how complicated the tree is.
- ID3 uses information theory to determine the most informative attribute.
- A measure of the information content of a message is the inverse of the probability of receiving the message:

$$information1(M) = 1/probability(M)$$

- Taking logs (base 2) makes information correspond to the number of bits required to encode a message:

$$information(M) = -log_2(probability(M))$$

Information

- The information content of a message should be related to the degree of surprise in receiving the message.
- Messages with a high probability of arrival are not as informative as messages with low probability.
- Learning aims to predict accurately, i.e. reduce surprise.
- Probabilities are multiplied to get the probability of two or more things both/all happening. Taking logarithms of the probabilities allows information to be added instead of multiplied.

Entropy

- Different messages have different probabilities of arrival.
- Overall level of uncertainty (termed entropy) is:

$$-\Sigma\, P\, log_2 P$$

- Frequency can be used as a probability estimate.
- E.g. if there are 5 +ve examples and 3 -ve examples in a node the estimated probability of +ve is 5/8 = 0.625.

Information and Learning

- We can think of learning as building many-to-one mappings between input and output.

- Learning tries to reduce the information content of the inputs by mapping them to fewer outputs.
- Hence we try to minimise entropy.
- The simplest mapping is to map everything to one output.
- We seek a trade-off between accuracy and simplicity.

Splitting Criterion

- Work out entropy based on distribution of classes.
- Trying splitting on each attribute.
- Work out expected information gain for each attribute.
- Choose best attribute.

Example

- Initial decision tree is one node with all examples.
- There are 4 +ve examples and 3 -ve examples
- i.e. probability of +ve is 4/7 = 0.57; probability of -ve is 3/7 = 0.43
- Entropy is: - (0.57 * log 0.57) - (0.43 * log 0.43) = 0.99
- Evaluate possible ways of splitting.
- Try split on *size* which has three values: *large, medium* and *small.*
- There are four instances with size = large.
- There are two large positives examples and two large negative examples.
- The probability of +ve is 0.5
- The entropy is: - (0.5 * log 0.5) - (0.5 * log 0.5) = 1

- There is one small +ve and one small -ve
- Entropy is: - (0.5 * log 0.5) - (0.5 * log 0.5) = 1
- There is only one medium +ve and no medium -ves, so entropy is 0.
- Expected information for a split on size is:

$$\left(1 \times \frac{4}{7}\right) + \left(1 \times \frac{2}{7}\right) + \left(0 \times \frac{1}{7}\right) = 0.86$$

- The expected information gain is: 0.99 - 0.86 = 0.13
- Now try splitting on colour and shape.
- Colour has an information gain of 0.52
- Shape has an information gain of 0.7
- Therefore split on shape.
- Repeat for all subtree

Summary of Splitting Criterion

Some people learn best from an example; others like to see the most general formulation of an algorithm. If you are an "examples" person, don't let the following subscript-studded presentation panic you.

Assume there are *k* classes C_1, ... C_k (*k* = 2 in our example).

to decide which attribute to split on:

- **for** each attribute that has not already been used
    - Calculate the information gain that results from splitting on that attribute
    - Split on the attribute that gives the greatest information gain.

to calculate the information gain from splitting *N* instances on attribute *A*:

- Calculate the entropy E of the current set of instances.
- **for** each value a_j of the attribute A(j = 1, ..., r)
    - Suppose that there are
      $J_{j,1}$ instances in class C_1,
      ...,
      $J_{j,k}$ instances in class C_k,
      for a total of $J_j$instances with $A = a_j$.
    - Let $q_{j,1} = J_{j,1}/J_j$, ..., $q_{j,k} = J_{j,k}/J_j$;
    - The entropy E_j associated with $A = a_j$ is
      $-q_{j,1} \log_2(q_{j,1}) \ldots -q_{j,k} \log_2(q_{j,k})$
- Now compute $E - (J_1/N)E_1 \ldots -(J_r/N)E_r$ - this is the information gain associated with a split on attribute A.

to calculate the entropy E of the current set of instances

- Suppose that of the N instances classified to this node, I_1 belong to classC_1, ..., I_k belong to classC_k,
- Let $p_1 = I_1/N$, ..., $p_k = I_k/N$,
- Then the initial entropy E is $-p_1 \log_2(p_1) -p_2 \log_2(p_2) \ldots -p_k \log_2(p_k)$.

Using the iProlog Implementation of ID3

% **cat example.data**

```
table object(
                texture(smooth, wavy, rough),
                temperature(cold, cool, warm, hot),
                size(small, medium, large),
                class(yes, no)
)!

object(smooth, cold, large, yes).
object(smooth, cold, small, no).
object(smooth, cool, large, yes).
object(smooth, cool, small, yes).
object(smooth, hot, small, yes).
object(wavy, cold, medium, no).
object(wavy, hot, large, yes).
object(rough, cold, large, no).
object(rough, cool, large, yes).
object(rough, hot, small, no).
object(rough, warm, medium, yes).
```

% **prolog example.data**

```
iProlog ML (21 March 2003)
: id(object)?
id0
: pp id0!

object(Texture, Temperature, Size, Class) :-
   (Temperature = cold ->
       (Texture = smooth ->
           (Size = small -> Class = no
           | Size = large -> Class = yes)
       | Texture = wavy -> Class = no
       | Texture = rough -> Class = no)
   | Temperature = cool -> Class = yes
   | Temperature = warm -> Class = yes
   | Temperature = hot ->
       (Texture = smooth -> Class = yes
       | Texture = wavy -> Class = yes
       | Texture = rough -> Class = no)).

:
```

Windowing

- ID3 can deal with very large data sets by performing induction on subsets or *windows* onto the data.

    1. Select a random subset of the whole set of training instances.

    2. Use the induction algorithm to form a rule to explain the current window.

    3. Scan through all of the training instances looking for exceptions to the rule.

    4. Add the exceptions to the window

- Repeat steps 2 to 4 until there are no exceptions left.

Noisy Data

- Frequently, training data contains "noise" - i.e. examples which are misclassified, or where one or more of the attribute values is wrong.
- In such cases, one is like to end up with a part of the decision tree which considers say 100 examples, of which 99 are in class C_1 and the other is apparently in class C_2 (because it is misclassified).
- If there are any unused attributes, we *might* be able to use them to elaborate the tree to take care of this one case, but the subtree we would be building would in fact be wrong, and would likely misclassify real data.
- Thus, particularly if we know there is noise in the training data, it may be wise to "prune" the decision tree to remove nodes which, statistically speaking, seem likely to arise from noise in the training data.
- A question to consider: *How fiercely should we prune?*

Expected Error Pruning

- Approximate expected error assuming that we prune at a particular node.
- Approximate backed-up error from children assuming we did not prune.
- If expected error is less than backed-up error, prune.

(Static) Expected Error

- If we prune a node, it becomes a leaf labelled, C.
- What will be the expected classification error at this leaf?

$$E(S) = \frac{N - n + k - 1}{N + k}$$

(This is called the *Laplace* error estimate - it is based on the assumption that the distribution of probabilities that examples will belong to different classes is uniform.)

S is the set of examples in a node
k is the number of classes
N examples in *S*
C is the majority class in *S*
n out of *N* examples in *S* belong to *C*

Backed-up Error

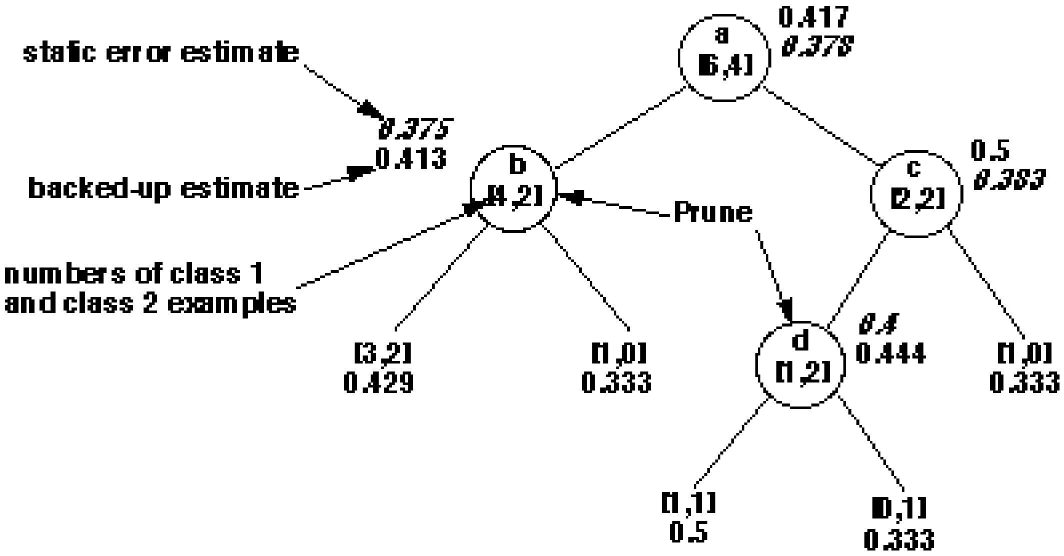
- For a non-leaf node
- Let children of *Node* be *Node1*, *Node2*, etc

$$BackedUpError(Node) = \sum_i P_i \times Error(Node_i)$$

- Probabilities can be estimated by relative frequencies of attribute values in sets of examples that fall into child nodes.

$$Error(Node) = \min(E(Node), BackedUpError(Node))$$

Pruning Example



Error Calculation

- Left child of *b* has class frequencies [3, 2]

$$E = \frac{N - n + k + 1}{N + k} = \frac{5 - 3 + 2 - 1}{5 + 2} = 0.429$$

- Right child has error of 0.333, calculated in the same way

- Static error estimate *E(b)* is 0.375, again calculated using the Laplace error estimate formula, with N=6, n=4, and k=2.

- Backed-up error is:

$$BackedUpError(b) = \frac{5}{6} \times 0.429 + \frac{1}{6} \times 0.333 = 0.413$$

(5/6 and 1/6 because there are 4+2=6 examples handled by node *b*, of which 3+2=5 go to the left subtree and 1 to the right subtree.

- Since backed-up estimate of 0.413 is greater than static estimate of 0.375, we prune the tree and use the static error of 0.375

Summary: Induction of Decision Trees

The ID3 family of decision tree induction algorithms use information theory to decide which attribute shared by a collection of instances to split the data on next. Attributes are chosen repeatedly in this way until a complete decision tree that classifies every input is obtained. If the data is noisy, some of the original instances may be misclassified. It may be possible to prune the decision tree in order to reduce classification errors in the presence of noisy data.
The speed of this learning algorithm is reasonably high, as is the speed of the resulting decision tree classification system. Generalisation ability can be reasonable too.

Neural Networks and Error Backpropagation Learning

Reference: Haykin, chapter 4

Aim:

To introduce some basic concepts of neural networks and then describe the backpropagation learning algorithm for feedforward neural networks.

Keywords: activation level, activation function, axon, backpropagation, backward pass in backpropagation, bias, biological neuron, cell body, clamping, connectionism, delta rule, dendrite, epoch, error backpropagation, error surface, excitatory connection, feedforward networks, firing, forward pass in backpropagation, generalization in backprop, generalized delta rule, gradient descent, hidden layer, hidden unit / node, inhibitory connection, input unit, layer in a neural network, learning rate, linear threshold unit, local minimum, logistic function, momentum in backprop, multilayer perceptron (MLP), neural network, neurode, neuron (artificial), node, output unit, over-fitting, perceptron, perceptron learning, recurrent network, sequence prediction tasks, sigmoidal nonlinearity, simple recurrent network, squashing function, stopping criterion in backprop, synapse, target output, threshold, training pattern, total net input, total sum-squared error, trainable weight, training pattern, unit, weight, weight space, XOR problem

Plan:

- linear threshold units, perceptrons
- outline of biological neural processing
- artificial neurons and the sigmoid function
- error backpropagation learning
    - delta rule
    - forward and backward passes
    - generalized delta rule
    - initialization
    - example: XOR with bp in iProlog
    - generalization and over-fitting
    - applications of backprop

There may be problems in this document with the display of Greek letters - the Greek letters will come out OK on recent versions of web browsers but may appear as raw html on earlier browser versons - for example, if you see a &Sigma;, interpret it as a Greek capital sigma - the symbol used in mathematical notation to express a summation. Other Greek symbols used include η = eta, δ = delta, Δ = capital Delta, α = alpha, and ϕ = phi. If the left-hand side of these equations all came out as Greek symbols, then you should be fine. (Really early web browsers could have trouble with the subscripts, too.)

Classification Tasks

- Statistical and connectionist approaches to machine learning are related to function approximation methods in mathematics.

- For the purposes of illustration let us assume that the learning task is one of classification.

- That is, we wish to find ways of grouping objects in a universe.

- In Figure 1 we have a universe of objects that belong to either of two classes `+' or `-'.

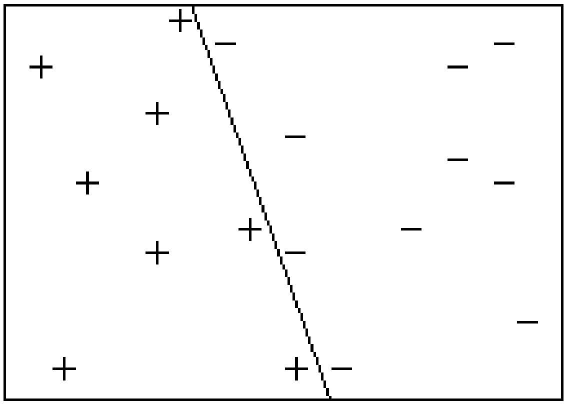


Figure 1: A linear discrimination between two classes

- Using function approximation techniques, we find a surface that separates the space, and the objects in it, into two different regions.

- In the case shown in Figure 1, the "surface" is just a line, and the associated function is called a *linear discriminant function*.

- Linear regression methods, or perceptron learning (see below) can be used to find linear discriminant functions.

History: Perceptrons

- A **perceptron** is a simple pattern classifier.

- Given a binary input vector, **x**, a *weight vector*, **w**, and a threshold value, *T*, if

$$\Sigma_i \, w_i x_i > T$$

then the output is 1, indicating membership of a class, otherwise it is 0, indicating exclusion from the class.

- **w**.**x** = *T* describes a *hyperplane* and the goal of perceptron learning is to find a weight vector, **w**, that results in correct classification for all training examples.

Perceptron Learning (Outline)

- All the elements of the weight vector are initially set to 0.
- For each training example, if the perceptron outputs 0 when it should output 1 then add the input vector to the weight vector; if the perceptron outputs 1 when it should output 0 then subtract the input vector from the weight vector; otherwise, do nothing.
- This is repeated until the perceptron yields the correct result for each training example.
- The algorithm has the effect of reducing the error between the actual and desired output.
- The perceptron is an example of a *linear threshold unit* (LTU).
- A single LTU can only recognise one kind of pattern, provided that the input space is *linearly separable*.
- If we wish to recognise more than one pattern, several LTU's can be combined. In this case, instead of having a vector of weights, we have an array. The output will now be a vector:

  u = W **x**

  where each element of **u** indicates membership of a class and each row in **W** is the set of weights for one LTU.
- This architecture is called a *pattern associator*.
- LTU's can only produce linear discriminant functions and consequently, they are limited in the kinds of classes that they can learn.
- In particular, they cannot be trained to compute the exclusive-or (XOR) function. [Footnote: (XOR(p,q) is true (1) if p is true or q is true but not both.]
- This seemed in the 1960s to kill off perceptron-based learning.

- However, subsequently a generalization of perceptrons was found that solved this problem.

Neural Models of Computation

- Biological neurons provide a model for computation (after all, brains are built from them).
- They have inputs (*dendrites*), outputs (*axon*) and a response to the inputs is generated by a process that gives rise to an electrical pulse propagated along the axon.
- The place where an axon of one neuron connects to a dendrite of another neuron is termed a *synapse*.
- Synapses differ in the effect that they have on the output of the neuron.
- Some are strongly excitatory, some weakly excitatory, some weakly inhibitory, and so on.

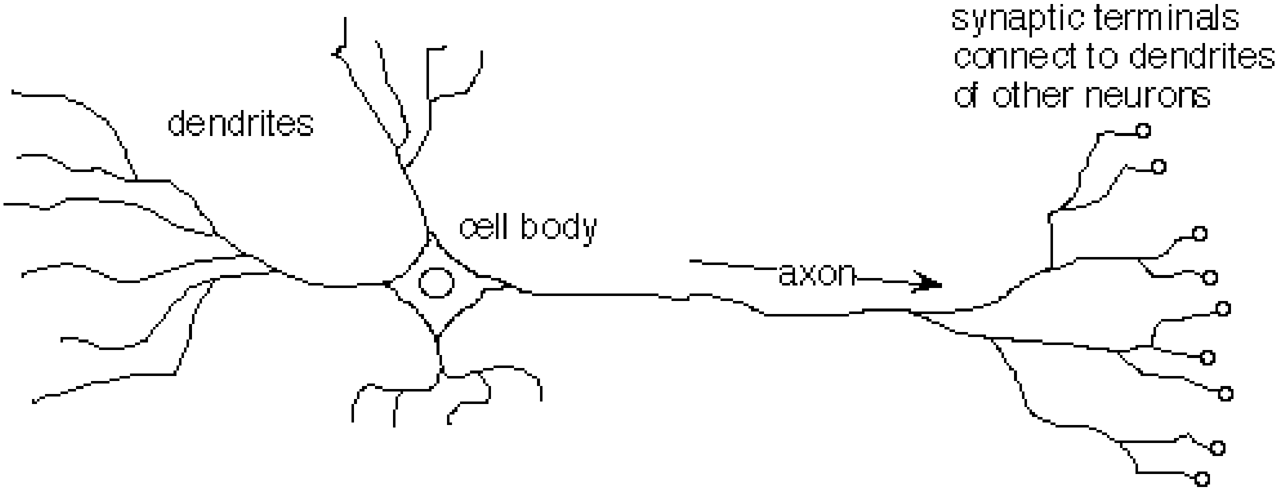


Figure 2: Neuron architecture

Biological Neurons and Artificial Neurons

- There is an analogy between a biological neuron and the simple model neurons (also termed *nodes* or *units*) used in neural networks.
- The incoming signals from the axons of other neurons correspond to the inputs y_i to the node.

- The strength of the synapse corresponds to weight w_i associated with a node.

- The signal propagated along the axon corresponds to the output $\phi(\Sigma\ i\ w_i y_i)$ of the node. (The operator ϕ will be explained below.)

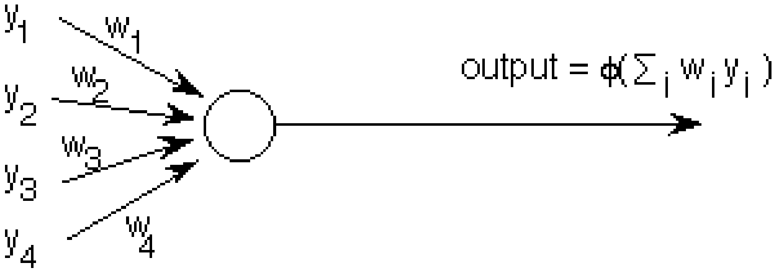


Figure 3: Artificial neuron (node) architecture

Multilayer Perceptrons

- In a multi-layer perceptron (MLP), there is a layer of input nodes, a layer of output nodes, and one or more intermediate layers of nodes that are referred to as *hidden nodes (hidden layers)*.

- In addition to this, each node in an MLP includes a *nonlinearity* at its output end.

- That is, the output function of the node is of the form:

  $$\phi(\Sigma_i\ w_i x_i)$$

  where $\phi(x)$ is a differentiable (smooth) function, frequently the *logistic* function:

  $$\phi(x) = 1/(1 + e^{-x})$$

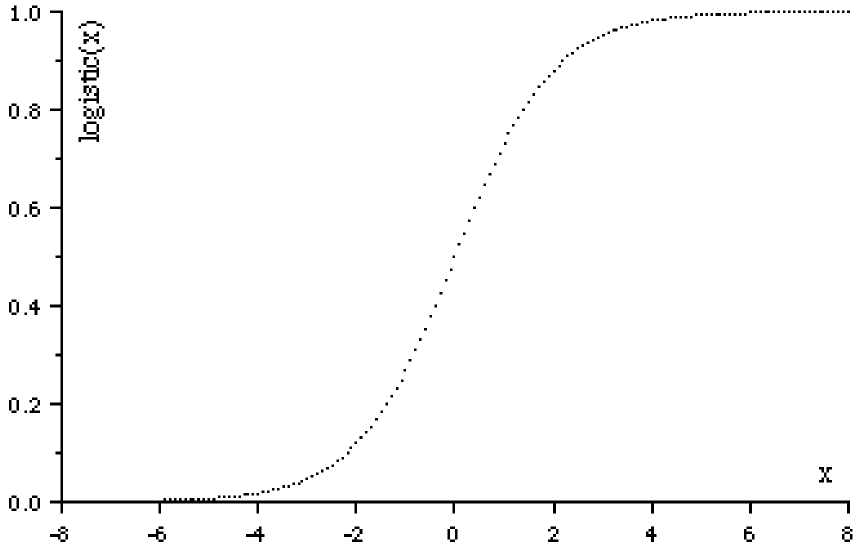


Figure 4: Graph of logistic function ϕ(x)

Overall Layout of MLP

- Typically also, each node in a layer (other than the output layer) is connected to every node in the next layer by a trainable weight.
- The overall layout is illustrated in Figure 5.

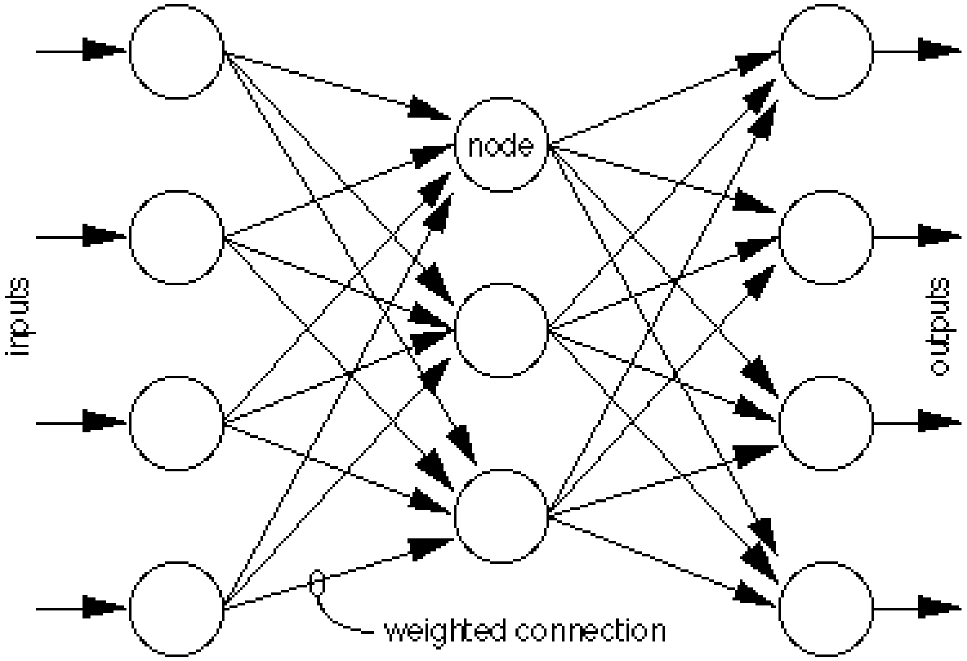


Figure 5. A multi-layer network

Node Internals

- Figure 6 shows the internals of a node.
- The weight w_{j_0} acts like a threshold.
- The y_i are the outputs of other nodes (or perhaps inputs to the network).
- The first step is forming the weighted sum $v_j = \Sigma_i w_{ji} y_i$.
- The second step is applying the nonlinearity function ϕ to v_j to produce the output y_j.

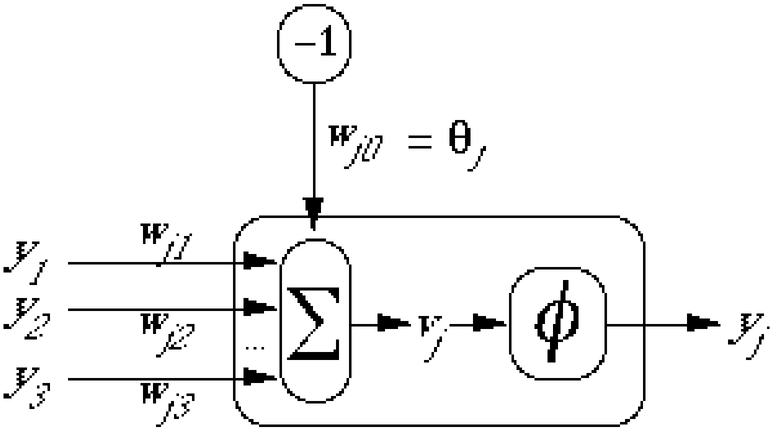


Figure 6. Internal functioning of node *j*

The Error Back-Propagation Learning Algorithm

- This algorithm was discovered and rediscovered a number of times - for details, see, e.g. chapter 4 of Haykin, S. *Neural Networks - a comprehensive foundation,*, 2nd ed., p.156. This reference also contains the mathematical details of the derivation of the backpropagation equations, which we shall omit.
- Like perceptron learning, back-propagation attempts to reduce the errors between the output of the network and the desired result.
- However, assigning blame for errors to hidden nodes (i.e. nodes in the intermediate layers), is not so straightforward. The error of the output nodes must be propagated back through the hidden nodes.

- The contribution that a hidden node makes to an output node is related to the strength of the weight on the link between the two nodes and the level of activation of the hidden node when the output node was given the wrong level of activation.

- This can be used to estimate the error value for a hidden node in the penultimate layer, and that can, in turn, be used in make error estimates for earlier layers.

Weight Change Equation

- The basic algorithm can be summed up in the following equation (the *delta rule*) for the change to the weight w_{ji} from node i to node j:

weight change	learning rate	local gradient	input signal to node j
Δw_{ji}	$= \eta$ *	δ_j *	y_i

where the local gradient δ_j is defined as follows:

1. If node j is an output node, then δ_j is the product of $\phi'(v_j)$ and the error signal e_j, where $\phi(*)$ is the logistic function and v_j is the total input to node j (i.e. $\Sigma_i\ w_{ji}y_i$), and e_j is the error signal for node j (i.e. the difference between the desired output and the actual output);

2. If node j is a hidden node, then δ_j is the product of $\phi'(v_j)$ and the weighted sum of the δ's computed for the nodes in the next hidden or output layer that are connected to node j.
[The actual formula is $\delta_j = \phi'(v_j)$ &Sigma$_k$ $\delta_k w_{kj}$ where k ranges over those nodes for which w_{kj} is non-zero (i.e. nodes k that actually have connections from node j. The δ_k values have already been computed as they are in the output layer (or a layer closer to the output layer than node j).]

Two Passes of Computation

FORWARD PASS: weights fixed, input signals propagated through network and outputs calculated. Outputs o_j are compared with desired outputs $\mathrm{d_j}$; the error signal $\mathrm{e_j = d_j - o_j}$ is computed.

BACKWARD PASS: starts with output layer and recursively computes the local gradient δ_j for each node. Then the weights are updated using the equation above for $\Delta \mathrm{w_{ji}}$, and back to another forward pass.

Sigmoidal Nonlinearity

With the sigmoidal function $\phi(x)$ defined above, it is the case that $\phi'(v_j) = y_j(1 - y_j)$, a fact that simplifies the computations.

Rate of Learning

- If the learning rate η is very small, then the algorithm proceeds slowly, but accurately follows the path of steepest descent in weight space.

- If η is largish, the algorithm may oscillate ("bounce off the canyon walls").

  A simple method of effectively increasing the rate of learning is to modify the delta rule by including a *momentum* term:

  $$\Delta w_{ji}(n) = \alpha\ \Delta w_{ji}(n-1) + \eta\ \delta_j(n) y_i(n)$$

  where α is a positive constant termed the *momentum constant*. This is called the *generalized delta rule*.

- The effect is that if the basic delta rule is consistently pushing a weight in the same direction, then it gradually gathers "momentum" in that direction.

Stopping Criterion

Two commonly used stopping criteria are:

- stop after a certain number of runs through all the training data (each run through all the training data is called an *epoch*);

- stop when the total sum-squared error reaches some low level. By total sum-squared error we mean $\Sigma_p \Sigma_i e_i^2$ where *p* ranges over all of the training patterns and *i* ranges over all of the output units.

Initialization

- The weights of a network to be trained by backprop must be initialized to some non-zero values.

- The usual thing to do is to initialize the weights to small random values.

The XOR Problem

- Because the XOR problem killed perceptrons in the 1960s, it is common to demonstrate the ability of backprop-trained MLPs using the XOR problem.

  We'll look at an example expressed in terms of iProlog's backprop facility.

  The architecture of the desired network is shown below:

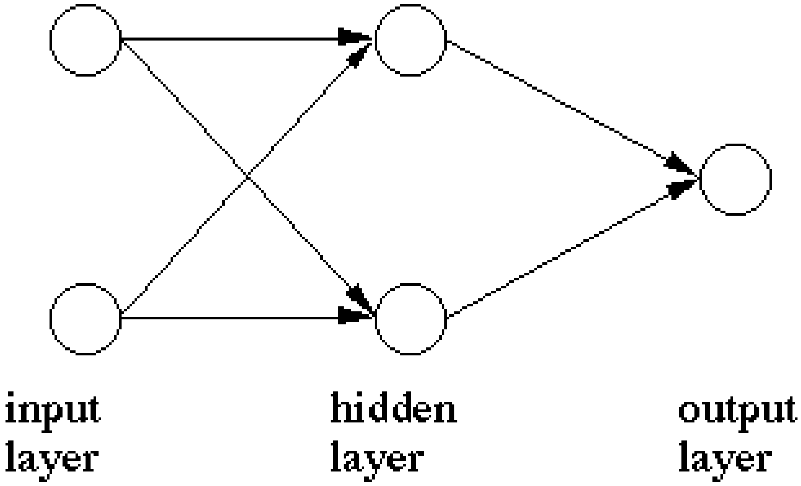


Figure 7: Network architecture for XOR problem

Backprop Specification in iProlog

- We must specify the structure of the network, and the training patterns. This is done, for XOR, using the following syntax:

```
table xor(+input1, +input2, -output)!

xor(0, 0, 0).
xor(1, 0, 1).
xor(1, 1, 0).
xor(0, 1, 1).     % training patterns

go is bp(xor, [2], [learning_rate = 0.25]).
```

```
test(F) :-
        xor(X, Y, Z),
        F::xor(X, Y, Z1),
        print('Expected = ', Z, '; Actual = ', Z1).
```

- This splits into four sections in the obvious way.
- The second section (`xor(0, 0, 0).` etc.) specifies the training patterns.
- The first section tells `bp` which are the inputs and which are the outputs of the training patterns. From this `bp` can figure out the number of input and output nodes.
- The third section (`go is` ...) tells us that `bp` is a function (a non-standard feature of iProlog) and tells `bp` that the `xor` relation has arity 3, that there is a single layer of hidden nodes with 2 nodes in it (it is possible to have multiple hidden layers), and specifies the value of a *parameter* (`learning_rate`).
- The available parameters and their default values are `learning_rate=1.0`, `momentum=0.9`, `error=0.01`, i.e. the total sum-squared error at which bp stops, and `epochs=1000`.]

Backprop in iProlog 2

- When this call to `bp` is performed, a *frame* is created, and the name of the frame - something like `bp0` - is printed assuming the call to `bp` succeeds.
- `bp` might not succeed in some circumstances - e.g. if some maximum number of epochs of training occur without the stopping criterion, expressed in terms of total sum-squared error, being reached.
- The frame contains, in its `rule` slot, information about the activation function of the hidden and output layer nodes - effectively the weights of the trained network.
- The fourth section defines a predicate `test(F)` for testing the trained network. The argument `F` should be a frame containing the information about the trained network.
- The test predicate effectively says - get the `rule` from the frame, then retrieve a pattern (`xor(X, Y, Z)`), evaluate the output for those inputs (`X` and `Y`) using the rule (`F::xor(X, Y, Z1)`), print out the target output `Z` and the actual output `Z1`, then backtrack and get another pattern, etc.
- The "predicate" `pp` allows you to prettyprint Prolog objects. This allows you to inspect the rule in the frame resulting from the call to `bp`.

Backprop in iProlog 3

To see it all happen, start iProlog as shown below, assuming the material shown above is in a file called `bp.xor`:

```
% prolog bp.xor

iProlog (14 August 1997)
: go?
bp0

: pp(bp0)!

xor(Input1, Input2, Output) :-
        X10 is logistic(1.79776 + -5.58764 * Input1 + -5.49273 * Input2),
        X11 is logistic(-5.04447 + 3.27915 * Input1 + 3.2671 * Input2),
        Output is logistic(3.27758 + -6.65149 * X10 + -6.56958 * X11).

: epochs of bp0?
239

: test(xor)?
Expected = 0; Actual = 0
Expected = 1; Actual = 1
Expected = 0; Actual = 0
Expected = 1; Actual = 1
** yes
:
```

Backprop as a Black Art

The tricky things about backprop networks include designing the network architecture:

- number of inputs
- number of hidden nodes
- number of layers,
- number of output nodes),

and then setting the adjustable parameters:

- learning rate
- momentum.

It also turns out to be advisable to stop training early, for the sake of better *generalization* performance.

Generalization

- Generalization means performance on unseen input patterns, i.e. input patterns which were not among the patterns on which the network was trained.
- If you train for too long, you can often get the total sum-squared error very low, by *over-fitting* the training data - you get a network which performs very well on the training data, but not as well as it could on unseen data.
- By stopping training earlier, one hopes that the network will have learned the broad rules of the problem, but not bent itself into the shape of some of the more idiosyncratic (perhaps even noisy) training patterns.
- The next section describes a strategy for picking the right stopping point.

Testing

- Assuming that training patterns are relatively plentiful, one can divide them into two sections (perhaps 80% / 20% of the original collection of training patterns).

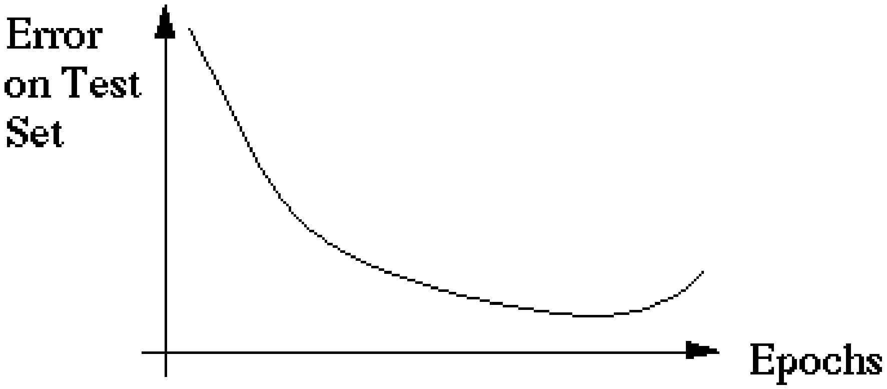


Figure 8: Illustration of over-fitting

- Train on the first section of the training data (80% of them in the example above).

- Test on the second section, the test set.
- Train for different numbers of epochs.
- Typically what is found is that, while error on the training set falls monotonically with the number of epochs, error on the test set falls and then rises.
- Estimate the point at which test-set error begins to rise again. Train for this number of epochs.

Successful Applications of Backprop

- Backprop tends to work well in some situations where human experts are unable to articulate a rule for what they are doing - e.g. in areas depending on raw perception, and where it is difficult to determine the *attributes* (in the ID3 sense) that are relevant to the problem at hand.
- For example, there is a proprietary system, which includes a backprop component, for assisting in classifying Pap smears.
  - The system picks out from the image the most suspicious-looking cells.
  - A human expert then inspects these cells.
  - This reduces the problem from looking at maybe 10,000 cells to looking at maybe 100 cells - this reduces the boredom-induced error rate.
- Other successful systems have been built for tasks like reading handwritten postcodes.

Discussion

- Function approximation methods, such as the ones discussed above, can often produce quite accurate classifiers because they are capable of constructing complex decision surfaces.
- However, knowledge is stored as weights in a matrix, so the results of learning are not easily available for inspection by a human reader.
- Moreover, the design of a network usually requires informed guesswork on the part of the user in order to obtain satisfactory results.
- Although some effort has been devoted to extracting meaning from networks, they still communicate little about the "rules" they implicitly encode.

- Connectionist learning algorithms are still computationally expensive.
- A critical factor in their speed is the encoding of the inputs to the network.
- When learning is complete, the speed of computation of the resulting network is very high.

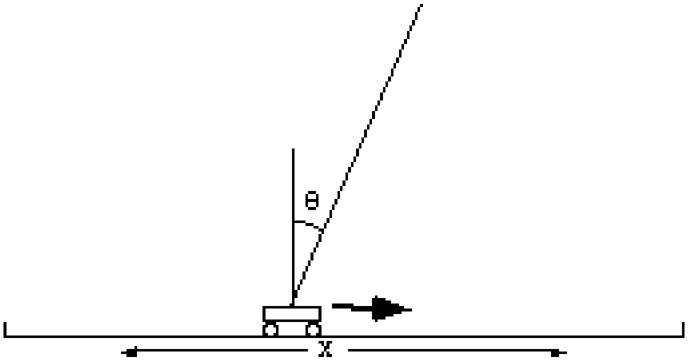
Summary: Error Backpropagation Learning

- After briefly describing linear threshold units, neural network computation paradigm in general, and the use of the logistic function (or similar functions) to transform weighted sums of inputs to a neuron, we outlined the error backpropagation learning algorithm.
- bp's performance on the XOR problem was demonstrated using the iProlog backprop interface.
- A number of refinements to backprop were looked at briefly, including momentum and a technique to obtain the best generalization ability.
- Backprop nets learn slowly but compute quickly once they have learned.
- They can be trained so as to generalize reasonably well.

Learning to Control Dynamic Physical Systems

- Conventional control theory requires a mathematical model to predict the behaviour of a process so that appropriate control decisions can be made.
- Many processes are too complicated to model accurately.
- Often, not enough information is available about the process' environment.
- When the system is too complicated or the environment is not well understood, an adaptive controller may work.
- An adaptive controller learns how to use the control actions available to meet the system's objective.
- The process is treated as a 'black box' and the program interacts with it by conditioned response.

The Pole and Cart



$$\ddot{\theta}_t = \frac{g \sin \theta_t + \cos \theta_t \left[\frac{-F_t - m_p l \dot{\theta}_t^2 \sin \theta_t}{m_c + m_p} \right]}{l \left[\frac{4}{3} - \frac{m_p \cos^2 \theta_t}{m_c + m_p} \right]}$$

$$\ddot{x}_t = \frac{F_t + m_p l \left[\dot{\theta}^2 \sin \theta_t - \ddot{\theta}_t \cos \theta_t \right]}{m_c + m_p}$$

$$x_{t+1} = x_t + \tau \dot{x}_t$$

$$\dot{x}_{t+1} = x_t + \tau \ddot{x}_t$$

$$\theta_{t+1} = \theta_t + \tau \dot{\theta}_t$$

$$\dot{\theta}_{t+1} = \theta_t + \tau \ddot{\theta}_t$$

m_c
: mass of cart

m_p
: mass of pole

l
: distance of centre of mass of pole from the pivot

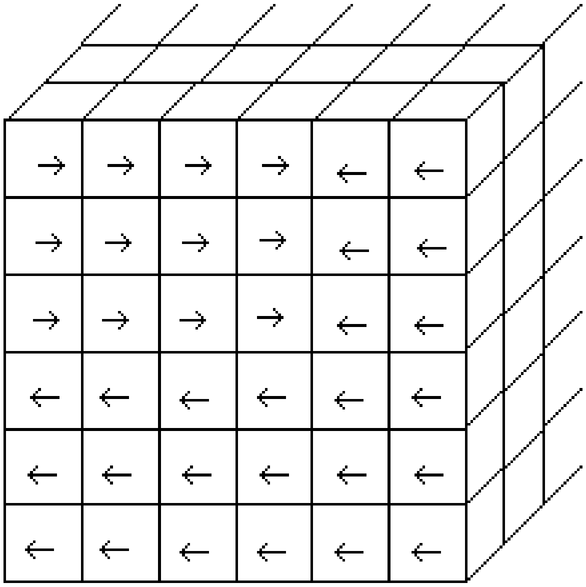
g
: acceleration due to gravity

F
: force applied to cart

t
: time interval of simulation

BOXES

The BOXES learning algorithm partitions the state space into regions according to how each dimension of the space is discretised. Each box represents a region of the problem space. In the pole and cart problem, there are four dimensions, one for each state variable (position and velocity of the cart, angle and angular velocity of the pole).



A box contains

- an action setting (left or right)
- the weighted sum of lifetimes after a left decision (left life)
- the weighted number of times a left decision has been made (left usage)

- the weighted sum of lifetimes after a right decision (right life)
- the weighted number of times a right decision has been made (right usage)

These numbers decay after each trial. That is, before a new value is added to the old value, the old value is multiplied by a factor between 0 and 1 (usually around 0.99). Thus, old experiences have less effect on box settings.

The BOXES Algorithm

```
boxes
        loop
                randomly set starting position
                put trial into t
                if t > 10,000 then exit
                for each box, b
                        if number of entries into b != 0
                                check_box(b)

trial
        put 0 into t
        find the current box
        if pole has fallen then return t
        add one to t
        if t > 10,000 then return t
        add one to number of entries into current box
        add t to time sum of current box
        make move according to setting of box

check_box(b)
        multiply left life by decay
        multiply left usage by decay
        multiply right life by decay
        multiply left usage by decay

        if action setting is LEFT
                add no. of entries - time sum to left life
                add no. of entries to left usage
        if action setting is RIGHT
                add no. of entries - time sum to right life
                add no. of entries to right usage

        put 0 into the no. of entries
        put zero into time sum
```

$$LeftValue \leftarrow \frac{LeftLife}{LeftUsage^k}$$

$$RightValue \leftarrow \frac{RightLife}{RightUsage^k}$$

```
if LeftValue > RightValue
        set action to LEFT
else if LeftValue < RightValue
        set action to RIGHT
else
        make random choice
```

CRICOS Provider Code No. 00098G

Genetic Algorithms

Genetic algorithms (Holland, 1975) perform a search for the solution to a problem by generating candidate solutions from the space of all solutions and testing the performance of the candidates. The search method is based on ideas from genetics and the size of the search space is determined by the representation of the domain. An understanding of genetic algorithms will be aided by an example.

A very common problem in adaptive control is learning to balance a pole that is hinged on a cart that can move in one dimension along a track of fixed length, as show in Figure 4. The control must use `bang-bang' control, that is, a force of fixed magnitude can be applied to push the cart to the left or right.

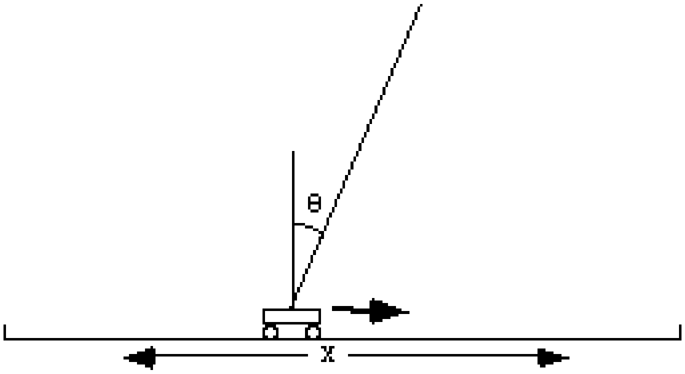


Figure 4. A Pole Balancer

Before we can begin to learn how to control this system, it is necessary to represent it somehow. We will use the BOXES method that was devised by Michie and Chambers (1968). The measurements taken of the physical system are the angle of the pole, θ , and its angular velocity and the position of the cart, *x*, and its velocity. Rather than treat the four variables as continuous values, Michie and Chambers chose to discretise each dimension of the state space. One possible discretisation is shown in Figure 5.

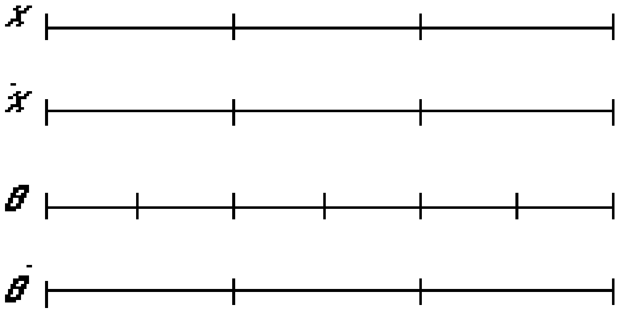


Figure 5. Discretisation of pole balancer state space.

This discretisation results in 3 x 3 x 6 x 3 = 162 `boxes' that partition the state space. Each box has associated with it an action setting which tells the controller that when the system is in that part of the state space, the controller should apply that action, which is a push to the left or a push to the right. Since there is a simple binary choice and there are 162 boxes, there are 2^162 possible control strategies for the pole balancer.

The simplest kind of learning in this case, is to exhaustively search for the right combination. However, this is clearly impractical given the size of the search space. Instead, we can invoke a genetic search strategy that will reduce the amount of search considerably.

In genetic learning, we assume that there is a population of individuals, each one of which, represents a candidate problem solver for a given task. Like evolution, genetic algorithms test each individual from the population and only the fittest survive to reproduce for the next generation. The algorithm creates new generations until at least one individual is found that can solve the problem adequately.

Each problem solver is a *chromosome*. A position, or set of positions in a chromosome is called a *gene*. The possible values (from a fixed set of symbols) of a gene are known as *alleles*. In most genetic algorithm implementations the set of symbols is {0, 1} and chromosome lengths are fixed. Most implementations also use fixed population sizes.

The most critical problem in applying a genetic algorithm is in finding a suitable encoding of the examples in the problem domain to a chromosome. A good choice of representation will make the search easy by limiting the search space, a poor choice will result in a large search space. For our pole balancing example, we will use a very simple encoding. A chromosome is a string of 162 boxes. Each box, or gene, can take values: 0 (meaning push left) or 1 (meaning push right). Choosing the size of the population can be tricky since a small population size provides an insufficient sample size over the space of solutions for a problem and large population requires a lot of evaluation and will be slow. In this example, 50 is a suitable population size.

Each iteration in a genetic algorithm is called a *generation*. Each chromosome in a population is used to solve a problem. Its performance is evaluated and the chromosome is given some rating of fitness. The population is also given an overall fitness rating based on the performance of its members. The fitness value indicates how close a chromosome or population is to the required solution. For pole balancing, the fitness value of a chromosome may be the number of time steps that the chromosome is able to keep the pole balanced for.

New sets of chromosomes are produced from one generation to the next. Reproduction takes place when selected chromosomes from one generation are recombined with others to form chromosomes for the next generation. The new ones are called *offspring*. Selection of chromosomes for reproduction is based on their fitness values. The average fitness of population may also be calculated at end of each generation. For pole balancing, individuals whose fitness is below average are replaced by reproduction of above average chromosomes. The strategy must be modified if two few or two many chromosomes survive. For example, at least 10% and at most 60% must survive.

Operators that recombine the selected chromosomes are called genetic operators. Two common operators are *crossover* and *mutation*. Crossover exchanges portions of a pair of chromosomes at a randomly chosen point called the crossover point. Some Implementations have more than one crossover point. For example, if there are two chromosomes, *X* and *Y*:

```
X = 1001 01011  Y = 1110 10010
```

and the crossover point is 4, the resulting offspring are:

```
O1 = 100110010  O2 = 1110 01011
```

Offspring produced by crossover cannot contain information that is not already in the population, so an additional operator, mutation, is required. Mutation generates an offspring by randomly changing the values of genes at one or more gene positions of a selected chromosome. For example, if the following chromosome,

```
Z = 100101011
```

is mutated at positions 2, 4 and 9, then the resulting offspring is:

```
O = 110001010
```

The number of offspring produced for each new generation depends on how members are introduced so as to maintain a fixed population size. In a *pure* replacement strategy, the whole population is replaced by a new one. In an *elitist* strategy, a proportion of the population survives to the next generation.

In pole balancing, all offspring are created by crossover (except when more the 60% will survive for more than three generations when the rate is reduced to only 0.75 being produced by crossover). Mutation is a background operator which helps to sustain exploration. Each offspring produced by crossover has a probability of 0.01 of being mutated before it enters the population. If more then 60% will survive, the mutation rate is increased to 0.25.

The number of offspring an individual can produce by crossover is proportional to its fitness:

$$\frac{\text{fitness value}}{\text{population fitness}} \times \text{No. of children}$$

where the number of children is the total number of individuals to be replaced. Mates are chosen at random among the survivors.

The pole balancing experiments described above, were conducted by Odetayo (1988) and required an average of 8165 trials before balancing pole. Of course, this may not be the only way of encoding the problem for a genetic algorithm and so a faster solution may be possible. However, this requires effort on the part of the user to devise a clever encoding.

Propositional Learning Systems

Reference: Bratko section 18.5

Aim:

To describe an algorithm whose input is a collection of instances and their correct classification and whose output is one or more sentences in some form of logic, describing each of the possible output classes in terms of the input attributes.

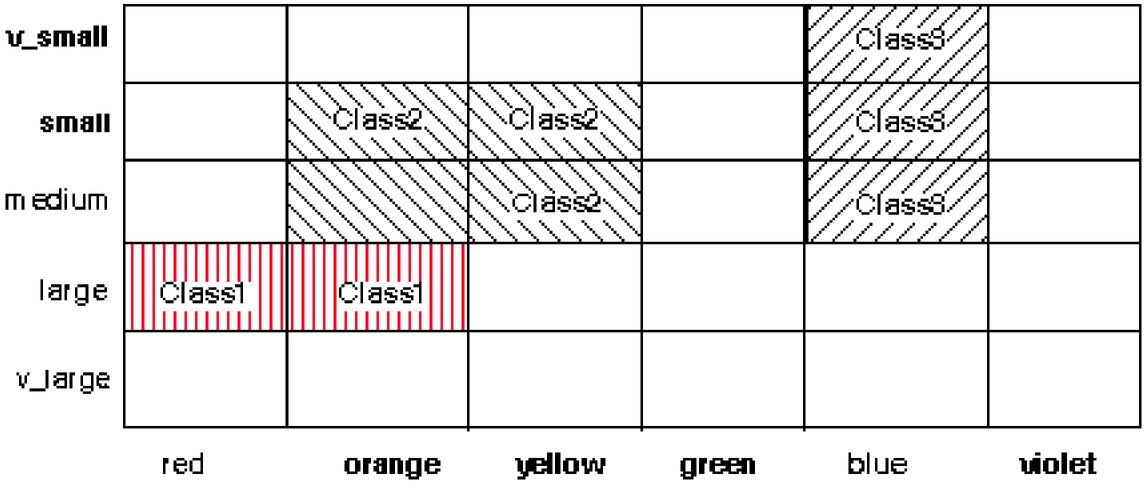
Keywords: Aq, conjunctive expressions, covering algorithm, instances, propositional learning systems, classes in classification tasks

Rather than searching for discriminant functions, symbolic learning systems find expressions equivalent to sentences in some form of logic. For example, we may distinguish objects according to two attributes: size and colour. We may say that an object belongs to class 3 if its colour is red and its size is very small to medium. Following the notation of Michalski (1983), the classes in Figure 1 may be written as:

$$
\begin{aligned}
&class1 \leftarrow size = large \wedge colour \in \{red, orange\}\\
&class2 \leftarrow size \in \{small, medium\} \wedge colour \in \{orange, yellow\}\\
&class3 \leftarrow size \in \{v\_small \ldots medium\} \wedge colour = blue
\end{aligned}
$$

Note that this kind of description partitions the universe into blocks, unlike the function approximation methods that find smooth surfaces to discriminate classes.

Interestingly, one of the popular early machine learning algorithms, Aq (Michalski, 1973), had its origins in switching theory. One of the concerns of switching theory is to find ways of minimising logic circuits, that is, simplifying the truth table description of the function of a circuit to a simple expression in Boolean logic. Many of the algorithms in switching theory take tables like Figure 1 and search for the best way of covering all of the entries in the table.



	red	orange	yellow	green	blue	violet
v_small					Class3	
small		Class2	Class2		Class3	
medium		▨	Class2		Class3	
large	Class1	Class1				
v_large						

Figure 1: Discrimination on attributes and values

Aq, uses a *covering algorithm*, to build its concept description:

```
cover := {};
repeat
        select one positive example, e;
        construct the set of all conjunctive expressions
               that cover e and no negative example in E-;
        choose the `best' expression, x, from this set;
        add x as a new disjunctof the concept;
        remove all positive examples covered by x
until there are no positive examples left;
```

The `best' expression is usually some compromise between the desire to cover as many positive examples as possible and the desire to have as compact and readable a representation as possible. In designing Aq, Michalski was particularly concerned with the expressiveness of the concept description language.

A drawback of the Aq learning algorithm is that, unlike ID3, it does not use statistical information, present in the training sample, to guide induction.

Summary:
Aq iteratively constructs a logic-based description of concepts on the basis of instances described by features.

APPENDIX

Michalski has always argued in favour of rule-based representations over tree structured representations, on the grounds of readability. When the domain is complex, decision trees can become very `bushy' and difficult to understand, whereas rules tend to be modular and can be read in isolation from the rest of the knowledge-base constructed by induction. On the other hand, decision trees induction programs are usually very fast. A compromise is to use decision tree induction to build an initial tree and then derive rules from the tree thus transforming an efficient but opaque representation into a transparent one (Quinlan, 1987).

It is instructive to compare the shapes that are produced by various learning systems when they partition the universe. Figure 1 demonstrates one weakness of decision tree and other symbolic classification. Since they approximate partitions with rectangles (if the universe is 2-dimensional) there is an inherent inaccuracy when dealing with domains with continuous attributes. Function approximation methods and IBL may be able to attain higher accuracy, but at the expense of transparency of the resulting theory. It is more difficult to make general comments about genetic algorithms since the encoding method will affect both accuracy and readability.

As we have seen, useful insights into induction can be gained by visualising it as searching for a cover of objects in the universe. Unfortunately, there are limits to this geometric interpretation of learning. If we wish to learn concepts that describe complex objects and relationships between the objects, it becomes very difficult to visualise the universe. For this reason, it is often useful to rely on reasoning about the concept description language.

As we saw, the cover in Figure 1 can be expressed as clauses in propositional logic. We can establish a correspondence between sentences in the concept description language (the hypothesis language) and a diagrammatic representation of the concept. More importantly, we can create a correspondence between generalisation and specialisation operations on the sets of objects and generalisation and specialisation operations on the sentences of the language.

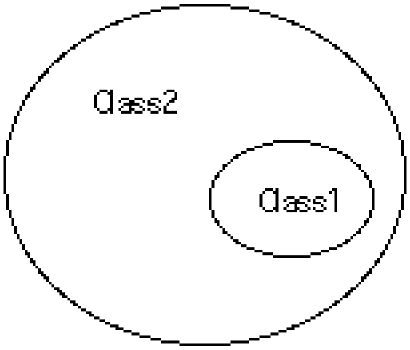


Figure 2: Generalisation as set covering

For example, Figure 2 shows two sets, labelled class 1 and class 2. It is clear that class 1 is a generalisation of class 2 since it includes a larger number of objects in the universe. We also call class 2 a specialisation of class 1. By convention, we say the *description* of class 1 is a generalisation of the *description* of class 2. Thus,

$$class1 \leftarrow size = large \qquad (1)$$

is a generalisation of

$$class2 \leftarrow size = large \wedge colour = red \qquad (2)$$

Once we have established the correspondence between sets of objects and their descriptions, it is often convenient to forget about the objects and only consider that we are working with expressions in a language. The reason is simple. Beyond a certain point of complexity, it is not possible to visualise sets, but it is relatively easy to apply simple transformations on sentences in a formal language. For example, clause (2) can be generalised very easily to clause (1) by dropping one of the conditions.

CRICOS Provider Code No. 00098G

Relations and Background Knowledge

Inductions systems, as we have seen so far, might be described as `what you see is what you get'. That is, the output class descriptions use the same vocabulary as the input examples. However, we will see in this section, that it is often useful to incorporate background knowledge into learning.

We use a simple example from Banerji (1980) to the use of background knowledge. There is a language for describing instances of a concept and another for describing concepts. Suppose we wish to represent the binary number, 10, by a left-recursive binary tree of digits `0' and `1':

```
        [head: [head: 1; tail: nil]; tail: 0]
```

`head' and `tail' are the names of attributes. Their values follow the colon. The concepts of binary digit and binary number are defined as:

$$\begin{aligned} x \in digit &\equiv x = 0 \vee x = 1 \\ x \in num &\equiv (tail(x) \in digit \wedge head(x) = nil) \\ &\quad \vee (tail(x) \in digit \wedge head(x) \in num) \end{aligned}$$

Thus, an object belongs to a particular class or concept if it satisfies the logical expression in the body of the description. Predicates in the expression may test the membership of an object in a previously learned concept.

Banerji always emphasised the importance of a description language that could `grow'. That is, its descriptive power should increase as new concepts are learned. This can clearly be seen in the example above. Having learned to describe binary digits, the concept of digit becomes available for use in the description of more complex concepts such as binary number.

Extensibility is a natural and easily implemented feature of horn-clause logic. In addition, a description in horn-clause logic is a logic program and can be executed. For example, to recognise an object, a horn clause can be interpreted in a forward chaining manner. Suppose we have a set of clauses:

(3) $C_1 \leftarrow P_{11} \wedge P_{12}$

(4) $C_2 \leftarrow P_{21} \wedge P_{22} \wedge C_1$

and an instance:

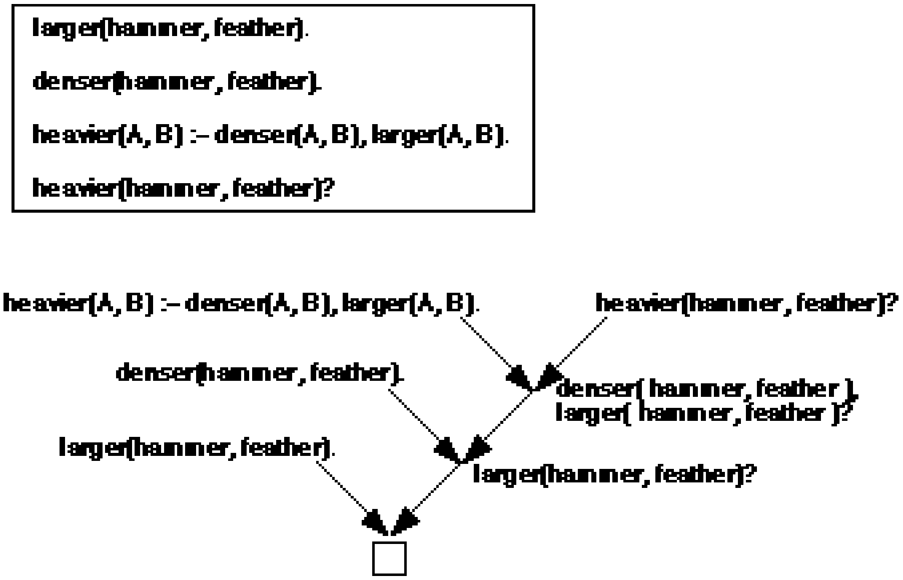
(5) $P_{11} \wedge P_{12} \wedge P_{21} \wedge P_{22}$

Clause (3) recognises the first two terms in expression (5) reducing it to

$$P_{21} \wedge P_{22} \wedge C_1$$

Clause (4) reduces this to C2. That is, clauses (3) and (4) recognise expression (5) as the description of an instance of concept C2.

When clauses are executed in a backward chaining manner, they can either verify that the input object belongs to a concept or produce instances of concepts. In other words, we attempt to prove an assertion is true with respect to a background theory. Resolution (Robinson, 1965) provides an efficient means of deriving a solution to a problem, giving a set of axioms which define the task environment. The algorithm takes two terms and resolves them into a most general unifier, as illustrated in Figure 10 by the execution of a simple Prolog program.



```
larger(hammer, feather).

denser(hammer, feather).

heavier(A, B) :– denser(A, B), larger(A, B).

heavier(hammer, feather)?
```

heavier(A, B) :– denser(A, B), larger(A, B).
heavier(hammer, feather)?
denser(hammer, feather).
denser(hammer, feather),
larger(hammer, feather)?
larger(hammer, feather).
larger(hammer, feather)?
□

Figure 10: A resolution proof tree from Muggleton and Feng (1990).

The box in the figure contains clauses that make up the theory, or knowledge base, and the question to be answered, namely, "is it true that a hammer is heavier than a feather"? A resolution proof is a proof by refutation. That is, answer the question, we assume that it is false and then see if the addition, to the theory, of this negative statement results in a contradiction.

The literals on the left hand side of a Prolog clause are positive. Those on the left hand side are negative. The proof procedure looks for *complimentary* literals in two clauses, i.e. literals of opposite sign that unify. In the example in Figure 10,

$$heavier\ (A, B)$$

and

$$heavier(hammer, feather)$$

unify to create the first resolvent,

$$:-\ denser(hammer, feather), heavier(hammer, feather) \quad .$$

A side effect of unification is to create variable substitutions $\{A / hammer, B / feather\}$. By continued application of resolution, we can eventually derive the empty clause, which indicates a contradiction.

Plotkin's (1970) work "originated with a suggestion of R.J. Popplestone that since unification is useful in automatic deduction by the resolution method, its dual might prove helpful for induction. The dual of the most general unifier of two literals is called the least general generalisation". At about the same time that Plotkin took up this idea, J.C. Reynolds was also developing the use of least general generalisations. Reynolds (1970) also recognised the connection between deductive theorem proving and inductive learning:

> Robinson's Unification Algorithm allows the computation of the greatest common instance of any finite set of unifiable atomic formulas. This suggests the existence of a dual operation of `least common generalization'. It turns out that such an operation exists and can be computed by a simple algorithm.

The method of least general generalisations is based on *subsumption*. A clause C_1 subsumes, or is more general than, another clause C_2 if there is a substitution σ such that $C_2 \supseteq C_1\sigma$.

The least general generalisation of

```
(6)             p(g(a), a)
(7)             and     p(g(b), b)
(8)             is      p(g(X), X).
```

Under the substitution {a/X}, (8) is equivalent to (6). and under the substitution {b/X}, (8) is equivalent to (7). Therefore, the least general generalisation of *p(g(a), a)* and *p(g(b), b)* is *p(g(X), X)* and results in the inverse substitution {X/{a, b}}.

Buntine (1988) pointed out that simple subsumption is unable to take advantage of background information which may assist generalisation. Suppose we are given two instances of a concept cuddly_pet,

(9) $$cuddly\_pet(X) \leftarrow fluffy(X) \wedge dog(X)$$

(10) $cuddly\_pet(X) \leftarrow fluffy(X) \wedge cat(X)$

Suppose we also know the following:

(11) $pet(X) \leftarrow dog(X)$
(12) $pet(X) \leftarrow cat(X)$

According to subsumption, the least general generalisation of (4) and (5) is:

(13) $cuddly\_pet(X) \leftarrow fluffy(X)$

since unmatched literals are dropped from the clause. However, given the background knowledge, we can see that this is an over-generalisation. A better one is:

(14) $cuddly\_pet(X) \leftarrow fluffy(X) \wedge pet(X)$

The moral being that a generalisation should only be done when the relevant background knowledge suggests it. So, observing (9), use clause (11) as a rewrite rule to produce a generalisation which is clause (14). which also subsumes clause (10).

Buntine drew on earlier work by Sammut (Sammut and Banerji, 1986) in constructing his generalised subsumption. Muggleton and Buntine (1998) took this approach a step further and realised that through the application of a few simple rules, they could invert resolution as Plotkin and Reynolds had wished. Here are two of the rewrite rules in propositional form:

Given a set of clauses, the body of one of which is completely contained in the bodies of the others, such as:

$$X \leftarrow A \wedge B \wedge C \wedge D \wedge E$$
$$Y \leftarrow A \wedge B \wedge C$$

the *absorption* operation results in:

$$X \leftarrow Y \wedge D \wedge E$$
$$Y \leftarrow A \wedge B \wedge C$$

Intra-construction takes a group of rules all having the same head, such as:

$$X \leftarrow B \wedge C \wedge D \wedge E$$
$$X \leftarrow A \wedge B \wedge D \wedge F$$

and replaces them with:

$$X \leftarrow B \wedge D \wedge Z$$
$$Z \leftarrow C \wedge E$$
$$Z \leftarrow A \wedge F$$

These two operations can be interpreted in terms of the proof tree shown in Figure 7. Resolution accepts two clauses and applies unification to find the maximal common unifier. In the diagram, two clauses at the top of a "V" are resolved to produce the resolvent at the apex of the "V". Absorption accepts the resolvent and one of the other two clauses to produce the third. Thus, it inverts the resolution step.

Intra-construction automatically creates a new term in its attempt to simplify descriptions. This is an essential feature of inverse resolution since there may be terms in a theory that are not explicitly shown in an example and may have to be invented by the learning program.

DISCUSSION

These methods and others (Muggleton and Feng, 1990; Quinlan, 1990) have made relational learning quite efficient. Because the language of Horn-clause logic is more expressive than the other concept description languages we have seen, it is now possible to learn far more complex concepts than was previously possible. A particularly important application of this style of learning is knowledge discovery. There are now vast databases accumulating information on the genetic structure of human beings, aircraft accidents, company inventories, pharmaceuticals and countless more. Powerful induction programs that use expressive languages may be a vital aid in discovering useful patterns in all these data.

For example, the realities of drug design require descriptive powers that encompass stereo-spatial and other long-range relations between different parts of a molecule, and can generate, in effect, new theories. The pharmaceutical industry spends over $250 million for each new drug released onto the market. The greater part of this expenditure reflects today's unavoidably "scatter-gun" synthesis of compounds which *might*} possess biological activity. Even a limited capability to construct predictive theories from data promises high returns.

The relational program Golem was applied to the drug design problem of modelling structure-activity relations (King *et al*, 1992). Training data for the program was 44 trimethoprim analogues and their observed inhibition of E. coli dihydrofolate reductase. A further 11 compounds were used as unseen test data. Golem obtained rules that were statistically more accurate on the training data and also better on the test data than a Hansch linear regression model. Importantly, relational learning yields understandable rules that characterise the stereochemistry of the interaction of trimethoprim with dihydrofolate reductase observed crystallographically. In this domain, relational learning thus offers a new approach which complements other methods, directing the time-consuming process of the design of potent pharmacological agents from a lead compound, variants of which need to be characterised for likely biological activity before committing resources to their synthesis.

Introduction to Natural Language Processing

Reference: Allen, chapter 2

Aim:

To review the grammar of English, introducing some terms for describing differen t types of English phrases, and the concept of a grammar rule. We also have a qu ick look at how the different levels of linguistic knowledge interact.

Keywords: abstract noun, active voice, ADJ, adjective, adjective phrase, ADJP, ADV, adverb, adverbial phrase, ADVP, agreement, apposition, article, aspect, AUX, auxiliary verb, BELIEVE, bitransitive, bound morpheme, cardinal, case, common noun, concrete noun, CONJ, conjunction, count noun, declarative, demonstrative, descriptive grammar, determiner, ellipsis, embedded sentence, features in NLP, first person, free morpheme, FUT, future perfect, gender, grammar, imperative, indicative, infinitive, inflection, intensifier, INTERJ, interjection, intransitive, lexeme, mass noun, morpheme, morphology, N, nominal, noun, noun modifier, noun phrase, NP, number (grammatical), object, ordinal, participle, particle, passive voice, PAST, past perfect, person, phone, phoneme, phonetics, phonology, phrasal verb, phrase, pluperfect, plural noun, possessive, PP, PP attachment, pragmatics, predicate, PREP, preposition, prepositional phrase, PRES, prescriptive grammar, present perfect, progressive, proper noun, proposition, qualifier, quantifier, quantifying determiner, relative clause, S, second person, sentence, simple future, simple past, simple present, singular noun, speech act, string, subject, subjunctive, syntax, tense, third person, transitive, V, verb, verb complement, verb group, verb phrase, VP, wh-question, word, y/n question

Plan:

- overview of linguistics. Our focus: lexicon, morphology, syntax, semantics, reference
- parts of speech and refinements (e.g. mass and count nouns)
- phrase types (e.g. noun phrase and verb phrase)
- inflection: -ing, -ed, -est
- grammar rules: NP -> ART ADJ N
- tense, aspect, active/passive, transitivity
- complement structure for verbs and adjectives

Introduction: Using a natural language interface is one reasonable way to approach certain kinds of interface design. It has the advantage of requiring no skill other than a rudimentary ability to use a keyboard to type a sentence or two. On the other hand, (Natural Language) NL interfaces have the disadvantage of not being particularly concise. Conceptually, there are two kinds of NL interface: those which restrict themselves to a subset of English (or of some other language) and those which try to provide (more or less) complete coverage of the language (note that even humans do not have this - nobody knows every word/meaning of their native language). Restricted interfaces face the *habitability* problem: can a user easily learn to stay within the language subset covered by the interface?

Other applications of Natural Language Processing (NLP) include Machine Translation (MT) and NL CAI (Computer Aided Instruction) systems. The budget for written translation services runs to tens of billions of dollars per year worldwide. Most of this is translation of commercial and technical documents - contracts, manuals, etc.

The aims of the segment are to familiarise you with some of the basic concepts of NLP (viz. syntactic and semantic processing, and reference resolution). You will be expected to be able to simulate the algorithms discussed (by hand).

The structure, especially the syntactic structure, of a natural language like English is usually expressed by grammar rules which decompose a sentence or other unit of language into small units. For example, we might have a rule that says that a *sentence* can consist (in English) of a *noun phrase* followed by a *verb phrase*. This rule would usually be expressed briefly as S -> NP VP, with S standing for sentence, NP for Noun Phrase, and VP for Verb Phrase. This should explain the significance of the codes (like NP) sprinkled throughout this introduction.

Reading: The NLP lectures will follow the book by James Allen: *Natural Language Understanding* (Benjamin Cummings, second edition, 1995). Chapter 2 of this book covers the same material as the Outline of English, below. The lecture notes are now intended to be self-contained, but be aware that they were originally developed for a class whose members all had copies of the book, hence reading the book as well as the notes may well improve your understanding.

Assessment: One programming assignment + one question on final exam.

Homework: Homework problems and subsequently solutions will be provided to help you learn. They are not assessable and should not be handed in.

Outline Of English

Words

* words are our atomic units of language (morphology later)

* words can be classified as nouns, verbs, articles, prepositions, etc. Some of these categories can be subclassified (e.g. proper nouns, abstract nouns, mass nouns, count nouns)

Grammatical Analysis

If you have studied grammar of English, e.g. in secondary school, then you might have seen sentences analysed into *subject* and *predicate* - examples:
(a) *Men lie* - subject = men; predicate = lie
(a') *Men lie freely* - subject = men; predicate = lie freely
(b) *A frictionless ball strikes a semi-infinite horizontal plane* - subject = A frictionless ball; predicate = strikes a ... plane
(c) *Many people who have fat bank accounts believe they are able to get anything by offering enough money* - subject = Many people ... accounts; predicate = believe they ... money
(d) *John gave Mary a book for her help* - subject = John; predicate = gave Mary ... help
Notice that both subject and predicate can be arbitrarily long. The subject can have a complicated structure, containing verbs, but it basically describes a concrete or abstract entity of some sort. The predicate consists of a verb followed by either a complement (which might be a sentence, like "they are ... money" in example (c) or a description of an entity, like "a semi- ... plane" in example (b), or may be absent, as in (a). There can be modifiers present, like "freely", as in example (a'). There can be further entities mentioned, like "a book" and "her help" in example (d).

Phrasal Categories

The term **noun phrase (NP)** is used for "descriptions of entities". In NLP, it is conventional to refer to predicates as **verb phrases (VPs)**. NPs thus occur both in the subject and (often) in the predicate.

There is a special term for a *preposition* followed by a noun phrase: it is called a **prepositional phrase (PP)**. By a preposition (or PREP) we mean a word like *to, for, by, with, from, in, into, under, over, around,* etc., which help to indicate the role played by the following NP in the action or state that the sentence describes. For example, in *Mary cut the apple with a sharp knife*, *with* marks *a sharp knife* as the **instrument** of the *cut* action. Some prepositions, particularly *of*, instead indicate relationships between objects in complex noun phrases, like *the beliefs of John Howard*. It is convenient to regard many instances of *'s* in English as being a variant on the preposition "of": *John Howard's beliefs* means the same thing as *the beliefs of John Howard*.

Sentences

* simple sentences consist of a verb and a collection of *noun phrases* filling various "roles"

The boy	hit	the window	with the hammer
NP1	V	NP2	with NP3
SUBJECT	V	OBJECT	INSTRUMENT

```
assert (action (hit (ActionName = hit81)
                    (Agent = boy15)
                    (Victim = window9987)
                    (Instrument = hammer234))))
```

Noun Phrases (NPs) and Nouns

* noun phrases can be syntactically simple (e.g. a single pronoun) or arbitrarily long and complicated, containing embedded noun phrases and even embedded sentences (e.g. *The lecturer in applied bovine psychology with the long flowing hair, green teeth, pimples, and bad breath, who stated that all students have a regrettable but definite and indisputable instinct to procrastinate indefinitely, a property which, he claims, they share with certain administrators who shall remain nameless to protect the guilty, and who has remained unpromoted for more years than I care to think about, for reasons which take little imagination to discover, [howled at the moon].*)

- the class of nouns itself has important substructure:

    - abstract vs concrete nouns *chemistry / vinegar*
    - common vs proper nouns *horse / Randwick*
    - singular vs plural nouns *horse/horses, man/men, criterion/criteria, maximum/maxima*
    - count vs mass nouns *programmer / code*

- other components of noun phrases include *specifiers* and *qualifiers*.

  Specifiers
  include **cardinals**(*one, thirty*), **ordinals** (*first, thirtieth*), and **determiners**, which include:

    - articles (ART) - *the, a, an*
    - demonstratives - *this, that, these, those ...*
    - possessives - *my, your, our, their, John's, the cat's, ...*
    - quantifying determiners (QUANT) - *each, every, some, no, few, both, ...*

  There are syntactic constraints on the way specifiers can be arranged. *All three bombs* is OK, but not *three all bombs*. Usually at most one quantifier occurs.

  Qualifiers:

    - adjectives (ADJ) - *angry, green, colourless, ...*
    - noun modifiers (nominals) - as in *brain damage, honours lab, disaster area, honours lab brain damage recovery party hangover, ...*

  Some simple noun phrases (NPs) have the following structure: ART ADJ N - that is, they consist of a ARTicle (like *a* or *the*) followed by an ADJective (like *green* or *big*) followed by a N (like *tree* or *fence*).

Grammar Rules

We can then write a **grammar rule** like NP -> ART ADJ N (and other rules like NP -> ART N and NP -> QUANT CARD N which give alternative NP structures). With NP defined, we can describe the structure of a prepositional phrase (PP) (defined above) by the rule PP -> PREP NP.

Sometimes such grammar rules *generate* strings of words which are not valid, like NP -> QUANT CARD N =>* *both seven horse*. Beyond noting this problem, and saying that we can always fix it by writing more specific rules, we won't worry about it further.

Syntactic Features of Nouns and Noun Phrases

* English is only moderately *inflected*; the main inflection for nouns is **number**, i.e. SINGULAR or PLURAL (some languages have other values for number, such as DUAL (two individuals)). The plural is usually formed from the basic singular form by adding -*s* (*dog* -> *dogs*), adding -*es* (*dress* -> *dresses*), or changing -*y* to -*ies* (*baby* -> *babies*). Irregular plurals include a small number of native English words like *man, woman, child, brother, ox, mouse, goose, tooth* -> *men, women, children, brothers/brethren, oxen, mice, geese, teeth*, and a fairly large number of Latin and some Greek words which follow the rules for those languages (*minimum, maximum, stratum, datum, medium* -> *minima, maxima, strata, data, media* ; *stimulus, focus, nucleus, alumnus* -> *stimuli, foci, nuclei, alumni* ; *larva, vertebra, formula* -> *larvae, verterbrae, formulae* ; *corpus, genus* -> *corpora, genera* ; *criterion, ganglion* -> *criteria, ganglia*). A number of words are or may be invariant between singular and plural: *sheep, fish* -> *sheep, fish/fishes*). Other animal names can have singular form but plural sense when the animal is regarded as game (i.e. suitable for hunting) or food: *the woods abounded with grouse and deer*.

The meanings of the terms 1s, 2s, 3s, 1p, 2p, and 3p, which are used later in the lectures, are as follows:

1s: first person singular (like the pronouns "I", "me")
2s: second person singular (like the pronouns "you", archaic "thou", "thee")
3s: third person singular (like "he", "him", "she", "her", "it")
1p: first person plural (like the pronouns "we", "us")
2p: second person plural (like the pronouns "you", archaic "ye")
3p: third person plural (like the pronouns "they", "them")

Most noun phrases which do not use these pronouns are 3s or 3p, depending on whether they are singular or plural. A set of these terms, such as {3s, 3p} signifies uncertainty between 3s and 3p - for example the word "the" is regarded as having the feature {3s, 3p}, because while it is always used in third person noun phrases, they may be either singular or plural. The word "an", on the other hand, is unambiguously 3s.

* some words, especially pronouns (PRO), are "marked" for *gender, person, number,* and/or *case*

- gender: *he / she / it*
- person: *we / you / they*
- number: *I / we*
- case: *I / me / my*

* there are other complications: *myself, mine*

Elements of Simple Sentences

* sentences may be in one of several *moods*

declarative (indicative) *Bert is listening*
y/n question (interrogative) *Is Bert listening?*
wh-question (interrogative) *When is Bert listening?*
imperative *Listen, Bert!*
subjunctive (looks like a past tense form, but isn't) *If Bert were listening, he might hear something to his advantage*

Verb Groups (VG, VP [-> VG NP PP*])

In syntactic patterns, "X*" means 0 or more Xs, so the rule

VP -> VG NP PP* means that a VP can end in optional PPs (Prepositional Phrases)

* simplest sentences: Subject + Verb Group

[*Ari*] + [*understands*]
[*Every single full-time student*] + [*will have been enslaved*]

* a verb group consists of *head verb* and (possibly) *auxiliary verbs* (which precede the head verb);

* the head verb may be *inflected:*

INFINITIVE	PRESENT	PRESENT	PAST	PAST	
	PARTICIPLE	TENSE	TENSE	PARTICIPLE	
eat	eating	eats	ate	eaten	IRREGULAR (STRONG) much change
set	setting	sets	set	set	IRREGULAR (STRONG) little change

-	-	can	could	-	IRREGULAR (MISSING FORMS)
be	being	am/is/are	was/were	been	IRREGULAR (VERB "to be")
kill	killing	kills	killed	killed	REGULAR (WEAK)

* Here is a fairly complete list of English irregular verbs.

* auxiliary verbs may be forms of *be*, *do*, & *have*, or *modal auxiliaries* like *can* , *could*, *will*, *shall*, *may*, *might*: use of the modal auxiliaries or *do* in a verb group forces the main verb to its infinitive form:

* the -*ing* inflection controls the *progressive aspect*
* the modal *do* (and its variants) control *emphasis*
* a combination of auxiliaries and inflections determine tense

- simple present ...*He eats the pizza*
- simple future ... *He will eat the pizza*
- simple past ... *She ate the pizza*
- present perfect ... *She has eaten the pizza*
- future perfect ... *He will have eaten the pizza*
- pluperfect ... *She had eaten the pizza*

* progressive aspect is "orthogonal" to tense

- present progressive *Graham is eating the pizza*
- future progressive *Graham will be eating the pizza*
- past progressive *Graham was eating the pizza*
- past perfect progressive *Graham has been eating the pizza*
- future perfect progressive *Graham will have been eating the pizza*
- pluperfect progressive *Graham had been eating the pizza*

* verbs may be *intransitive*, *transitive*, or *bitransitive* (some can do two or all of these)

intransitive	Bill glares, Jim eats, Dan gives
transitive	Jim eats buns, Dan gives flowers
bitransitive	Dan gives his mother flowers, Dan gives flowers to his mother

The first noun phrase is called the *subject*. The *object* is the noun phrase after the verb group, in the transitive case. In the bitransitive case, the first noun phrase after the verb group is the *indirect object*, and the second noun phrase after the verb group is the *direct object*.

* transitive and bitransitive verbs can be in *passive* form:

Buns are eaten by Jim
Buns are eaten
The student is given help by Dan
The student is given help
Help is given to the student by Dan
Help is given to the student
Help is given by Dans
* *Is glared by Bill*

(the * indicates that the phrase is *not* correct English)

* some verbs have forms involving *particles* (kind of adverb)

Peter threw up
Trevor ate up the pastie
Jim ate up the street
I looked up his number (but *I looked it up*)

* some verbs take more complex variants: *phrasal verbs*

Jill [gave] *her boyfriend* [up] *for the sake of music*

* verb groups can include *not,* the negative particle

Agreement

* the verb group in a sentence or clause must agree in number with the subject noun phrase: *The dog bites* but *The dogs bite*. NB: the agreement is with the *subject* noun phrase, not the *preceding* noun phrase: *The dog with fleas bites*. This distinction only exists in the simple present tense and when the first auxiliary inflects: *The dog(s) bit/will bite/will have bitten/did bite* etc., but *The dog has bitten/is biting/does bite* vs *The dogs have bitten/are biting/do bite.*

Prepositional Phrases (PREP, PP)

* noun phrases fill *roles* in sentences - if other roles than subject and object are needed, then these are usually marked by *prepositions*: *Blake stole the bear from Amelia* - here *from* marks the *source* of the bear.

* prepositional phrase (classically) = preposition + noun phrase. But there are other possibilities -

out of the oven
We saw [the VDU]*that Jim was electrocuted* [by]

Embedded Sentences (S)

[John's giving up the game] *was cowardly*
The man [who gave Paul the money]*was crazy*
The money [that was given to Paul] *was lost*
The money [given to Paul] *was lost* (reduced form)

Complements

Bert believes [he is Napoleon]
Margaret wants [to own a fire-engine red Porsche]
Blake decided [that he would never steal a bear again]

Adjectives (ADJ) & Adjective Phrases (ADJP)

Adjectives can be inflected to form comparatives: ... and superlatives:

good / better / best
brave / braver / bravest
gracious / more gracious / most gracious

Complexes of words qualifying a noun are termed ADJPs:

*the **brave** tutor*
*the **noble senior** tutor*
*the **most estimable senior** tutor*
*the **completely incomprehensible** problem sheet*
*[He was] **hungry for knowledge***
proud to be a frog
as slow as a linguistics lecture on an autumn evening

Conjunctions (CONJ)

* words like *and, but, if, so, or, ...* can connect various kinds of structures together to make more complicated ones:

*He is rich **and** keeps a secretary to type his HCI assignments*
*She was poor **but** she was honest*
*I'll have a blue **or** green one*

***If** that's a labor government, I think I'll vote Anarchist next time*
*He dislikes football, **so** we put him in gaol*
*This is totally **and** outrageously unjustified*
*Granny rocked back **and** forth in her chair by the fire*

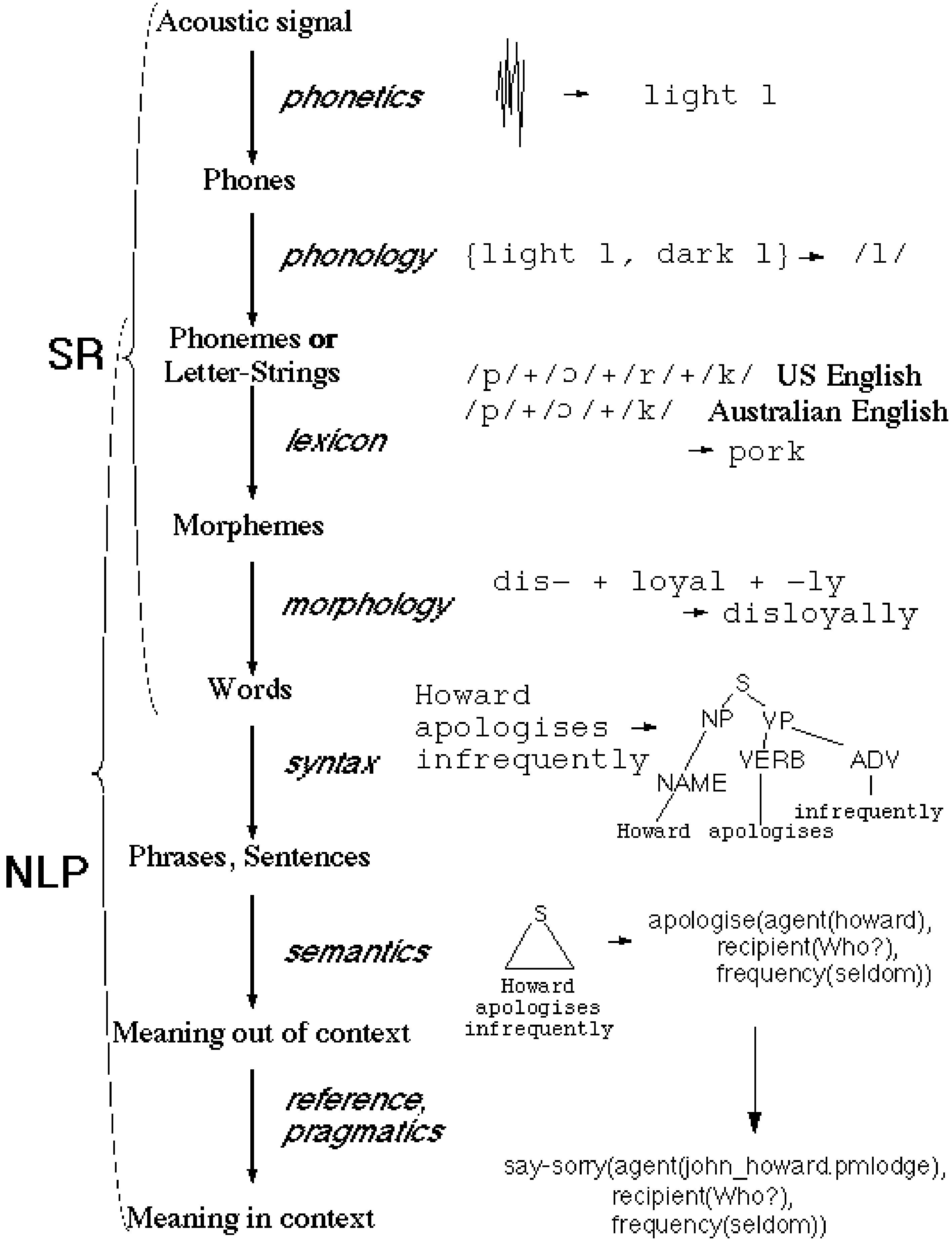
Lexical and Phrasal Categories

Lexical (or preterminal) categories are classes of words. The ones that we have seen in this outline are N, V, ADJ, ADV, PREP, ART, QUANT, POSSADJ, CARD, ORD, PRO, CONJ. Some words may be members of two or more categories, like *damn*, which can be a noun, a verb, and adjective, and adverb, or an interjection (INTERJ).

Phrasal categories are the remaining categories, and (for English) include NP, VP, PP, ADJP, S, and any other categories that it may be convenient to define (using grammar rules).

Overall Picture of Linguistics and Language Processing

SR signifies Speech Recognition; NLP = Natural Language Processing


SR
NLP
Acoustic signal
phonetics
→ light l
Phones
phonology
{light l, dark l} → /l/
Phonemes or
Letter-Strings
/p/+/ɔ/+/r/+/k/ US English
/p/+/ɔ/+/k/ Australian English
→ pork
lexicon
Morphemes
morphology
dis- + loyal + -ly
→ disloyally
Words
syntax
Howard
apologises →
infrequently
S
NP
VP
NAME
VERB
ADV
Howard
apologises
infrequently
Phrases, Sentences
semantics
S
Howard
apologises
infrequently
→ apologise(agent(howard),
recipient(Who?),
frequency(seldom))
Meaning out of context
reference,
pragmatics
say-sorry(agent(john_howard.pmlodge),
recipient(Who?),
frequency(seldom))
Meaning in context

Summary: Outline of English Syntax

While reviewing English syntax, we have introduced a number of terms and symbols for describing types of words and phrases in English, including the *lexical categories* N, V, ADJ, ADV, CONJ, INTERJ, and PREP, and *phrasal categories* NP, VP, PP, ADVP, ADJP, VG, and S. In passing, we also introduced the concept of grammar rules such as

NP -> ART NOUN

Grammars and Parsing

Reference: Chapter 3 of Allen

Aim:

To describe several types of formal grammars for natural language processing, parse trees, and a number of parsing methods, including a bottom-up chart parser in some detail.
We show the use of Prolog for syntactic parsing of natural language text. Other issues in parsing, including PP attachment, are briefly discussed.

Keywords: accepter, active arc, active chart, alphabet (in grammar), ATN, augmented grammar, augmented transition networks, bottom-up parser, CFG, chart, chart parsing, Chomsky hierarchy, constituent, context-free, context-sensitive, CSG, derivation, distinguished non-terminal, generalized phrase structure grammar, generate, GPSG, grammar rule, HDPSG, head-driven phrase structure grammar, language generated by a grammar, left-to-right parsing, lexical category, lexical functional grammar, lexical insertion rule, lexical symbol, lexicon, LFG, Marcus parsing, non-terminal, parse tree, parser, phrasal category, pre-terminal, predictive parser, production, regular, rewriting process, right-linear grammar, right-to-left parsing, robustness, semantic grammar, sentential form, shift-reduce parser, start symbol, systemic grammar, terminal, top-down parser, unrestricted grammar

Plan:

- parsers, parse trees
- context-free grammars, context-free rules
- lexical and phrasal categories
- Chomsky hierarchy: unrestricted > context-sensitive > context-free > regular
- derivation, sentential forms
- top-down parsing: predictive parser
- bottom-up parsing
- garden-path sentences, well-formed substring table
- chart parsing, bottom-up chart parser
- example of chart parser use
- parsing in Prolog
- problems and limitations in syntactic parsing

Grammar

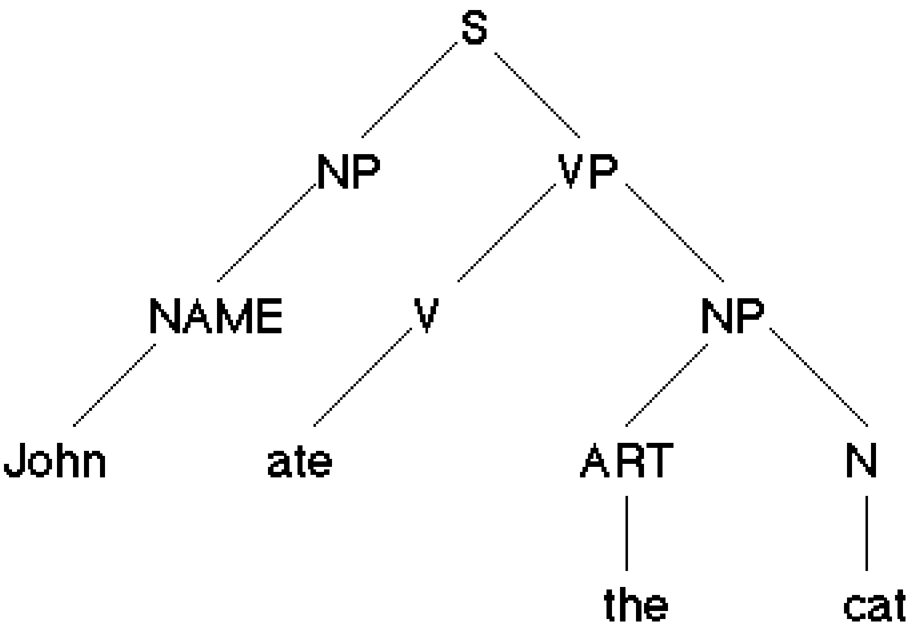
- formal description of the structure of a language (**prescriptive** grammar is something else: rules for a high-status variant of a language: e.g. *don't split infinitives*)

Parser

- an algorithm for analysing sentences given a grammar:

  - may just give yes/no answer to the question *Does this sentence conform to the given grammar*: such a parser is termed an *accepter*

  - may also produce a structure description ("parse tree") for correct sentences:



Here is the same tree in list notation ...

```
(S (NP (NAME John))
   (VP (V ate)
       (NP (ART the))
           (N cat))))
```

and in a Prolog notation ...

```
s(np(name(john)),
  vp(v(ate),
     np(art(the),
        n(cat))))
```

Describing Syntax: Context Free Grammars (CFGs)

1. S -> NP VP 2. VP-> V NP 3. NP-> NAME 4. NP-> ART N	grammar rules
.	.
5. NAME -> John 6. V -> ate 7. ART -> the 8. N -> cat	lexical entries

Definition: A CFG is a 5-tuple (P, A, N, T, S) where:

- P is a set of context-free productions, i.e. objects of the form X -> beta, where X is a member of N, and beta is a string over the alphabet A
- A is an alphabet of grammar symbols;
- N is a set of **non-terminal** symbols;
- T is a set of **terminal** symbols (and N U T = A);
- S is a distinguished non-terminal called the start symbol (think of it as "sentence" in NLP applications).

E.g.

T = { ate, cat, John, the}

N = { S, NP, VP, N, V, NAME, ART}

P = { S -> NP VP, VP-> V NP, NP-> NAME, NP-> ART N, NAME-> John V-> ate, ART-> the, N-> cat} .

Notice how the productions were split into grammar rules and lexicon above. N, V, NAME and ART are called **pre-terminal** or **lexical symbols**.

Aside: In programming language grammars, there would be context-free rules like:

WhileStatement -> **while** Condition **do** StatementList **end**
StatementList -> Statement
StatementList -> Statement **;** StatementList

Types of grammars:

- **unrestricted grammars**

- **context-sensitive grammars**
- **context-free grammars**
- **regular grammars**.

With context-free grammars, these form the **Chomsky hierarchy** of grammars. The four types of grammar differ in the type of rewriting rule alpha -> beta that is allowed:

- *unrestricted grammar*. No restrictions on the form that the Rules can take. Unrestricted grammars are not widely used: their extreme power makes them difficult to use;
- *context sensitive grammar*, or *transformational grammar*. The length of the string alpha on the left-hand side of any rule must be less than or equal to the length of the string beta]] on the right-hand side of the rule. It is equivalent to require that all the productions be of the form lambda A rho -> lambda alpha rho, where lambda and rho are arbitrary (possibly null) strings. lambda and rho are thought of as the *left and right context* in which the non-terminal symbol A can be rewritten as the non-null symbol-string alpha. Hence the term *context-sensitive* grammar. Context-sensitive production rules can be used for transforming an active sentence into the corresponding passive sentence.
- *context-free grammar*, or *phrase structure grammar*. All rules must be of the form A -> alpha, where A is a nonterminal symbol and alpha is an arbitrary string of symbols.
- *regular grammar*, or a *right linear grammar*. All rules take one of two forms: either A -> t, or A -> tN, where A and N are non-terminal symbols and t is a member of the vocabulary (a terminal symbol). Regular grammar rules are not powerful enough to conveniently describe natural languages (or even programming languages). They can sometimes be used to describe portions of languages, and have the advantage that they lead to fast parsing.

Since the restrictions which define the grammar types apply to the *rules*, it makes sense to talk of unrestricted, context-sensitive, context-free, and regular rules.

Deriving sentences from a grammar

To **derive** a sentence from a grammar, start with the start symbol S, and refer to it as the *current string*. Repeatedly perform the following *rewriting process*:

- choose any rule whose LHS occurs in the *current string* (in a CFG, the LHS must be a non-terminal symbol);
- replace the LHS of that rule with the RHS of the rule, in the *current string*, producing a *new current string.*

Repeat until there are no non-terminals remaining in the current string. The current string is then a **sentence** in the **language generated by the grammar**. (Before this, it is termed a **sentential form**.) E.g.:

	Current string	Rewriting
S	=> NP VP	S
	=> **NAME** VP	NP
	=> **John** VP	NAME
	=> John **V NP**	VP
	=> John **ate** NP	V
	=> John ate **ART N**	NP
	=> John ate **the** N	ART
	=> John ate the **cat**	N

Parsing might be the reverse of this process (doing the steps shown above in reverse would constitute a **bottom-up right-to-left parse** of *John ate the cat*.)

Top-Down Parsing ... with CFGs

Basic "predictive" parser

In this parsing method, we guess ("predict") what the next production to be applied should be, and, in case we guess wrong, we stack any alternatives so that we can come back to them if necessary. We "cross off" the leading item of our current sentential form if it can be matched against the next word of the sentence. This algorithm can take exponential time on bad sentences for bad CFGs. With *well-behaved* grammars, it can be a linear time algorithm. NL grammars are not usually all that *well-behaved*.

Use this grammar:

S -> NP VP
NP -> ART N | NAME
PP -> PREP NP
VP -> V | V NP | V NP PP | V PP

Sentence: 1 The 2 dogs 3 cried. 4

```
                    Backup states   Position
S -> NP VP                             1
  -> ART N VP                          1
                   NAME VP             1
  -> (The) N VP                        2
                   NAME VP             1
  -> (dogs) VP                         3
```

```
                   NAME VP            1
  -> V                                3
                   V NP               3
                   V NP PP            3
                   V PP               3
                   NAME VP            1
  -> (cried.)                         4
```

Bottom-up parsing

The dogs cried -> ART N V

-> NP V

-> NP VP

-> S

Using bottom-up parsing methods, all CFGs can be parsed in n*n*n steps, where n is the length of the sentence. The reason predictive parsing may take exponential time is that it may re-parse pieces of the sentence, particularly confusing sentences (like *the horse [that was] raced past the barn fell*):
>

Indication of parsing state	**Comment**
The horse raced past the barn ...	"past the barn" parsed as PP, "raced" parsed as [main] verb
The horse raced past the barn fell	oops!
The horse ...	backtrack to this point and start over
The horse raced past the barn fell	"past the barn" parsed as PP again: "raced" parsed as ADJ [from past participle] starting the ADJP ["raced past the barn"]

Chart Parsing

(Section 3.4 of Allen)

The chart is a record of all the substructures (like *past the barn*) that have ever been built during the parse. A chart is sometimes also called a *well-formed substring table*. Actual charts get complex rapidly.

Charts help with "elliptical" sentences:

1: Q. How much are apples?
2: A. Thirty cents each.
3: Q. Plums?

An attempt to parse 3 as a sentence fails, but all is not lost, as the analysis of *plums* as an NP is on the chart. Successful parsing of the entire utterance as *any* kind of structure can be useful.

A Bottom-Up Chart-Based Parsing Algorithm

This algorithm expresses an efficient bottom-up parsing process. It is guaranteed to parse a sentence of length N within N*N*N parsing steps, and it will perform better than this (N*N steps or N steps) with well-behaved grammars.

The algorithm constructs (phrasal or lexical) constituents of a sentence. We shall use the sentence *the green fly flies* as an example in describing a NL-oriented parser similar to Earley's algorithm. The sentence is notionally annotated with *positions*: our sentence becomes 0*the*1*green*2*fly*3*flies*4.

In terms of this notation, the parsing process succeeds if an S (sentence) constituent is found covering positions 0 to 4.

Points (1) to (8) below do not completely specify the order in which parsing steps are carried out: one reasonable order is to scan a word (as in (2)) and then perform all possible parsing steps as specified in (3) - (6) before scanning another word. Parsing is completed when the last word has been read and all possible subsequent parsing steps have been performed.

Parser inputs: *sentence, lexicon, grammar*.

Parser operations:

(0) The algorithm operates on two data structures: the **active chart** - a collection of active arcs (see (3) below) and the **constituents** (see (2) and (5)). Both are initially empty.

(1) The grammar is considered to include *lexical insertion* rules: for example, if *fly* is a word in the lexicon/vocabulary being used, and if its lexical entry includes the fact that *fly* may be a N or a V, then rules of the form N -> *fly* and V -> *fly* are considered to be part of the grammar.

(2) As a word (like *fly*) is scanned, constituents corresponding to its lexical categories are created:

N1: N -> *fly* FROM 2 TO 3, and

V1: V -> *fly* FROM 2 TO 3

(3) If the grammar contains a rule like NP -> ART ADJ N, and a constituent like ART1: ART -> *the* FROM m TO n has been found, then an *active arc* ARC1: NP -> ART1 * ADJ N FROM m TO n

is added to the *active chart*. (In our example sentence, m would be 0 and n would be 1.) The "*" in an active arc marks the boundary between found constituents and constituents not (yet) found.

(4) *Advancing the "*"*: If the active chart has an active arc like:

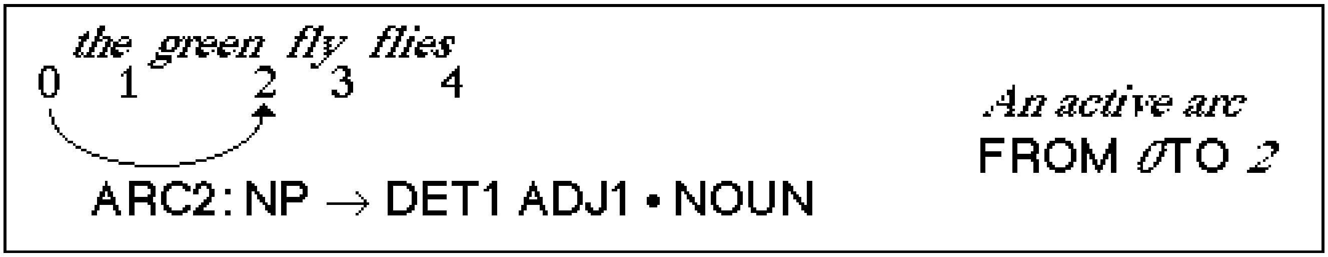
ARC1: NP -> ART1 * ADJ N FROM m TO n

and there is a constituent in the chart of type ADJ (i.e. the first item after the *), say

ADJ1: ADJ -> *green* FROM n TO p

such that the FROM position in the constituent matches the TO position in the active arc, then the "*" can be advanced, creating a new active arc:

ARC2: NP -> ART1 ADJ1 * N FROM m TO p.



(5) If the process of advancing the "*" creates an active arc whose "*" is at the far right hand side of the rule: e.g.

ARC3: NP -> ART1 ADJ1 N1 * FROM 0 TO 3

then this arc is converted to a constituent.

NP1: NP -> ART1 ADJ1 N1 FROM 0 TO 3.

Not all active arcs are ever *completed* in this sense.

(6) Both lexical and phrasal constituents can be used in steps 3 and 4: e.g. if the grammar contains a rule S -> NP VP, then as soon as the constituent NP1 discussed in step 5 is created, it will be possible to make a new active arc

ARC4: S -> NP1 * VP FROM 0 TO 3

(7) When subsequent constituents are created, they would have names like NP2, NP3, ..., ADJ2, ADJ3, ... and so on.

(8) The aim of parsing is to get phrasal constituents (normally of type S) whose FROM is 0 and whose TO is the length of the sentence. There may be several such constituents.

Chart Parsing Example

Grammar:

1. S -> NP VP
2. NP -> ART ADJ N
3. NP -> ART N
4. NP -> ADJ N
5. VP -> AUX V NP
6. VP -> V NP

Sentence: 0 The 1 large 2 can 3 can 4 hold 5 the 6 water 7

Lexicon:

the... ART
large... ADJ
can... AUX, N, V
hold... N, V
water... N, V

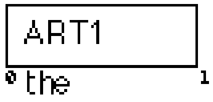
Steps in Parsing:

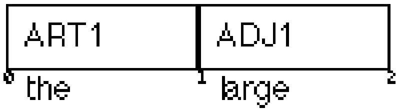
```
*sentence*: the                       *position*: 1
*constituents*:
ART1: ART -> "the" from 0 to 1
*active-arcs*:
ARC1: NP -> ART1 * ADJ N from 0 to 1      [rule 2]
ARC2: NP -> ART1 * N from 0 to 1      [rule 3]

*sentence*: the large                 *position*: 2
*constituents*: add
ADJ1: ADJ -> "large" from 1 to 2
*active-arcs*: add
ARC3: NP -> ART1 ADJ1 * N from 0 to 2     [arc1*->]
ARC4: NP -> ADJ1 * N from 1 to 2          [rule 4]

*sentence*: the large can             *position*: 3
```

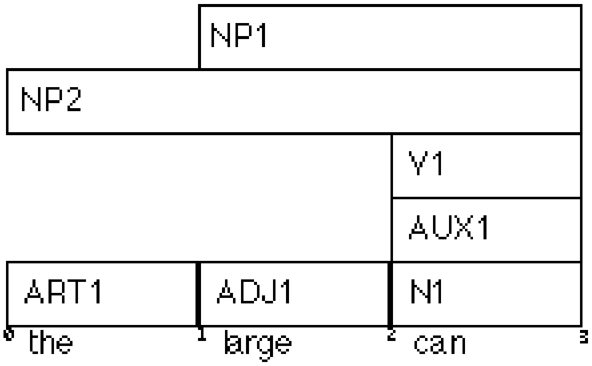
```
*constituents*: add
NP2: NP -> ART1 ADJ1 N1 from 0 to 3       [arc3*->]
NP1: NP -> ADJ1 N1 from 1 to 3            [arc4*->]
N1: N -> "can" from 2 to 3
AUX1: AUX -> "can" from 2 to 3
V1: V -> "can" from 2 to 3
*active-arcs*: add
ARC5: VP -> V1 * NP from 2 to 3           [rule 6]
ARC6: VP -> AUX1 * V NP from 2 to 3       [rule 5]
ARC7: S -> NP1 * VP from 1 to 3           [rule 1]
ARC8: S -> NP2 * VP from 0 to 3           [rule 1]

*sentence*: the large can can         *position*: 4
*constituents*: add
N2: N -> "can" from 3 to 4
AUX2: AUX -> "can" from 3 to 4
V2: V -> "can" from 3 to 4
*active-arcs*: add
ARC9: VP -> AUX1 V2 * NP from 2 to 4      [arc6*->]
ARC10: VP -> V2 * NP from 3 to 4          [rule 6]
ARC11: VP -> AUX2 * V NP from 3 to 4      [rule 5]

*sentence*: the large can can hold
*position*: 5
*constituents*: add
N3: N -> "hold" from 4 to 5
V3: V -> "hold" from 4 to 5
*active-arcs*: add
ARC12: VP -> AUX2 V3 * NP from 3 to 5     [arc11*->]
ARC13: VP -> V3 * NP from 4 to 5          [rule 6]

*sentence*: the large can can hold the
*position*: 6
*constituents*: add
ART2: ART -> "the" from 5 to 6
*active-arcs*: add
ARC14: NP -> ART2 * ADJ N from 5 to 6     [rule 2]
ARC15: NP -> ART2 * N from 5 to 6         [rule 3]

*sentence*: the large can can hold the water
*position*: 7
*constituents*: add
S2: S -> NP1 VP2 from 1 to 7              [arc7*->]
S1: S -> NP2 VP2 from 0 to 7              [arc8*->]
```

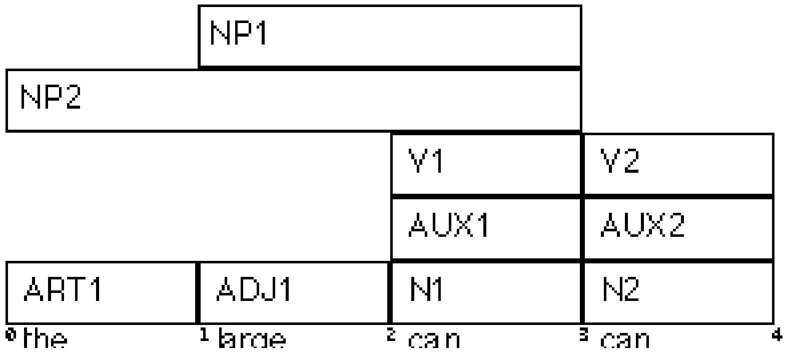
```
VP2: VP -> AUX2 V3 NP3 from 3 to 7         [arc12*->]
VP1: VP -> V3 NP3 from 4 to 7              [arc13*->]
NP3: NP -> ART2 N4 from 5 to 7             [arc15*->]
N4: N -> "water" from 6 to 7
V4: V -> "water" from 6 to 7
*active-arcs*: add
ARC16: VP -> V4 * NP from 6 to 7           [rule 6]
ARC17: S -> NP3 * VP from 5 to 7           [rule 1]
```

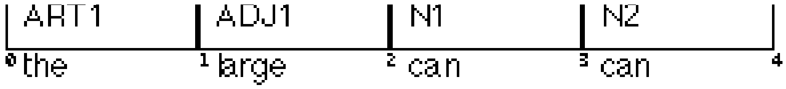
Doing the same steps diagrammatically:

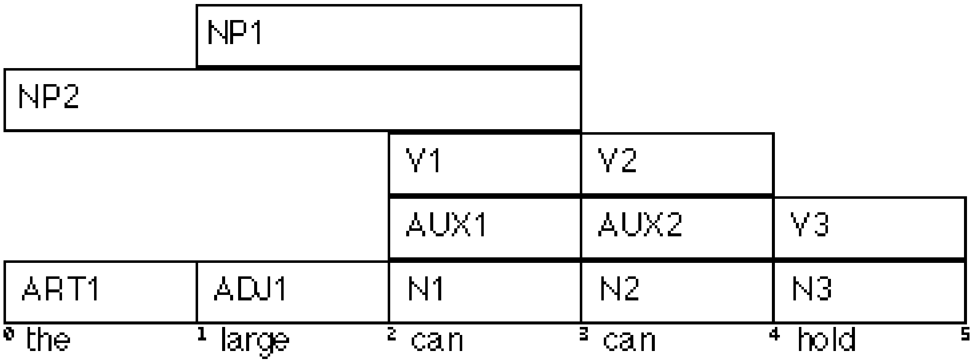


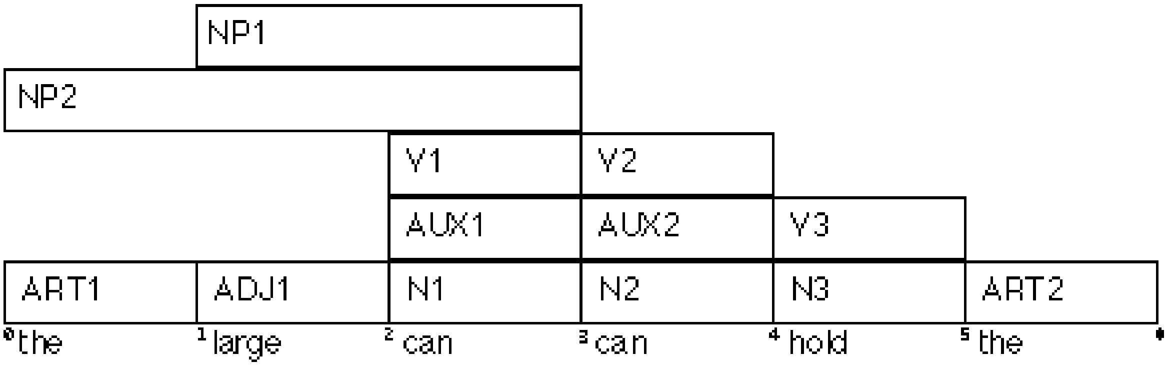


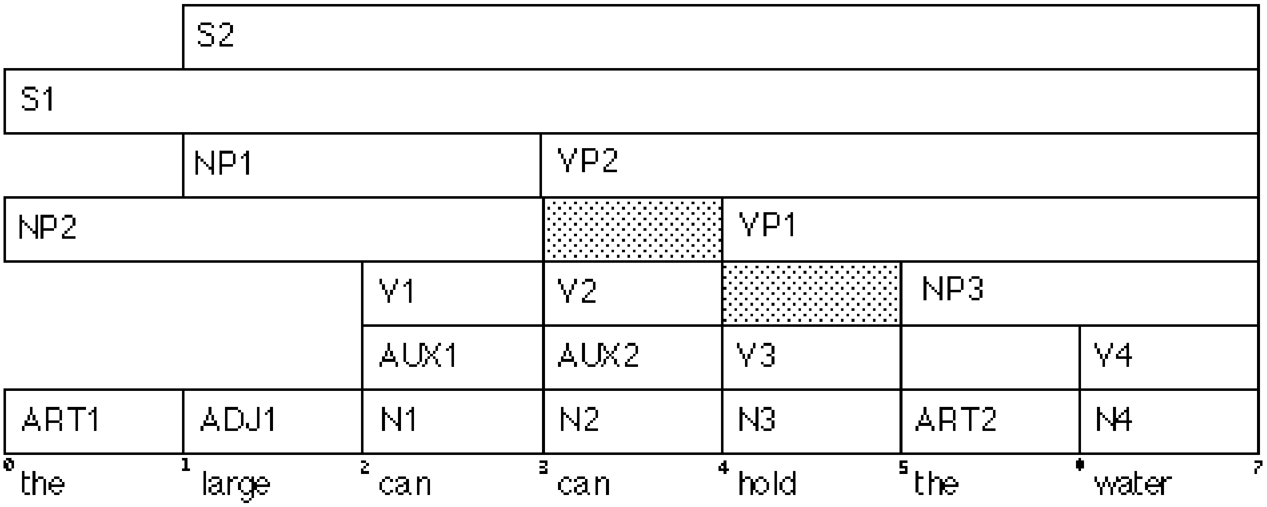





ART1
ADJ1
N1
N2
0 the
1 large
2 can
3 can
4


NP1
NP2
V1
V2
AUX1
AUX2
V3
ART1
ADJ1
N1
N2
N3
0 the
1 large
2 can
3 can
4 hold
5


NP1
NP2
V1
V2
AUX1
AUX2
V3
ART1
ADJ1
N1
N2
N3
ART2
0 the
1 large
2 can
3 can
4 hold
5 the
6


S2
S1
NP1
VP2
NP2
VP1
V1
V2
NP3
AUX1
AUX2
V3
V4
ART1
ADJ1
N1
N2
N3
ART2
N4
0 the
1 large
2 can
3 can
4 hold
5 the
6 water
7

-

```
.
.
.
.
.
.
```

- disadvantage of bottom-up method: will find irrelevant constituents like the VP *hold the water* which would not be noticed by a top-down parser, because it wouldn't be looking for (the start of) a VP at that point;
- a top-down CFG parser can have a chart (but you have to keep track of which constituents are hypothesized, and which are substantiated by the text being parsed);
- mixed-mode parsers have best aspects of both methods (disadvantage - more complicated)

Recording Sentence Structure

A frame-like, or slot/filler representation is suitable: *Jack found a dollar*

```
(S SUBJ (NP NAME "Jack")
   MAIN-V find
   TENSE past
   OBJ (NP ART a HEAD dollar))

s(subj(np(name(jack))),
  mainv(find),
  tense(past),
  obj(np(art(a), head(dollar))))
```

Grammars and Logic Programming

(Section 3.8 in Allen)

Grammar rules can be encoded directly in Prolog:

S -> NP VP ..... `s(P1,P3) :- np(P1,P2), vp(P2,P3).`

That is, there is an S from position P1 to position P3 if there is an NP from P1 to P2 and a VP from P2 to P3. Similarly:

VP-> V NP ..... `vp(P1,P3) :- v(P1,P2), np(P2,P3).`

NP-> NAME ..... `np(P1,P2) :- propernoun(P1,P2).`
NP-> ART N ..... `np(P1,P3) :- art(P1,P2), noun(P2,P3).`

The lexicon can be defined by predicates like:

NAME -> John ..... `isname(john).`
V -> ate ..... `isverb(ate).`
ART -> the ..... `isart(the).`
N -> cat ..... `isnoun(cat).`

One also needs predicates to link the lexicon to the grammar. For each lexical category, like ART, you define a predicate, like `art(From, To) :- word(Word, From, To), isart(Word).` that is true only if the word between the specified position is of that category. Here `word(Word, From, To)` signifies that `Word` is there in the input sentence between positions `From` and `To`. Similarly:

```
noun(From, To) :- word(Word, From, To), isnoun(Word).
v(From, To) :- word(Word, From, To), isverb(Word).
propernoun(From, To) :- word(Word, From, To), isname(Word).
```

To use the system as so far described, one can assert the words in their sentence positions:

```
word(john, 1, 2).
word(ate, 2, 3).
word(the, 3, 4).
word(cat, 4, 5).
```

and then use the Prolog query `s(1,5)?` This will result in `** yes`.

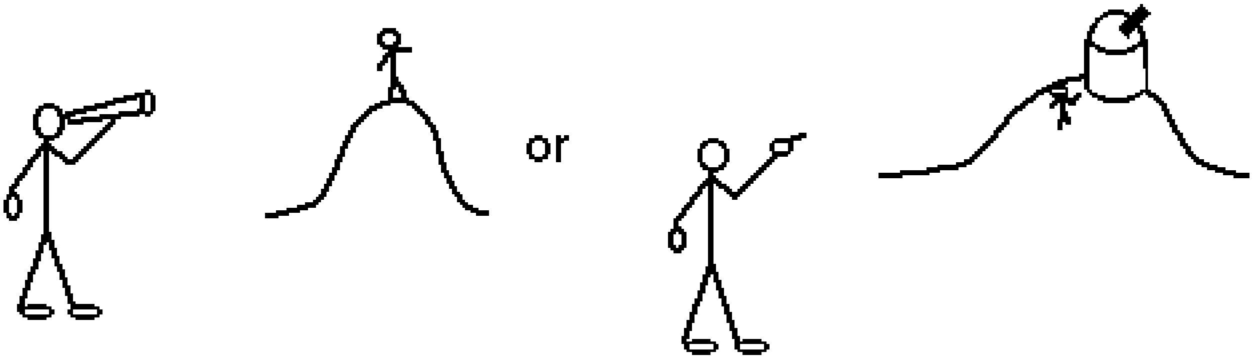
To get at the *structure* of the sentence, add extra arguments to pass the structure across necks as it is built:

```
s(P1,P3,s(NP,VP)) :- np(P1,P2,NP), vp(P2,P3,VP).
vp(P1,P3,vp(v(Verb),NP)) :- v(P1,P2,Verb), np(P2,P3,NP).
np(P1,P2,np(name(Name))) :- proper(P1,P2,Name).
np(P1,P3,np(art(Art),noun(Noun))) :-
art(P1,P2,Art),
noun(P2,P3,Noun).

art(From, To, Word) :- word(Word, From, To), isart(Word).
noun(From, To, Word) :- word(Word, From, To), isnoun(Word).
v(From, To, Word) :- word(Word, From, To), isverb(Word).
proper(From, To, Word) :- word(Word, From, To), isname(Word).
```

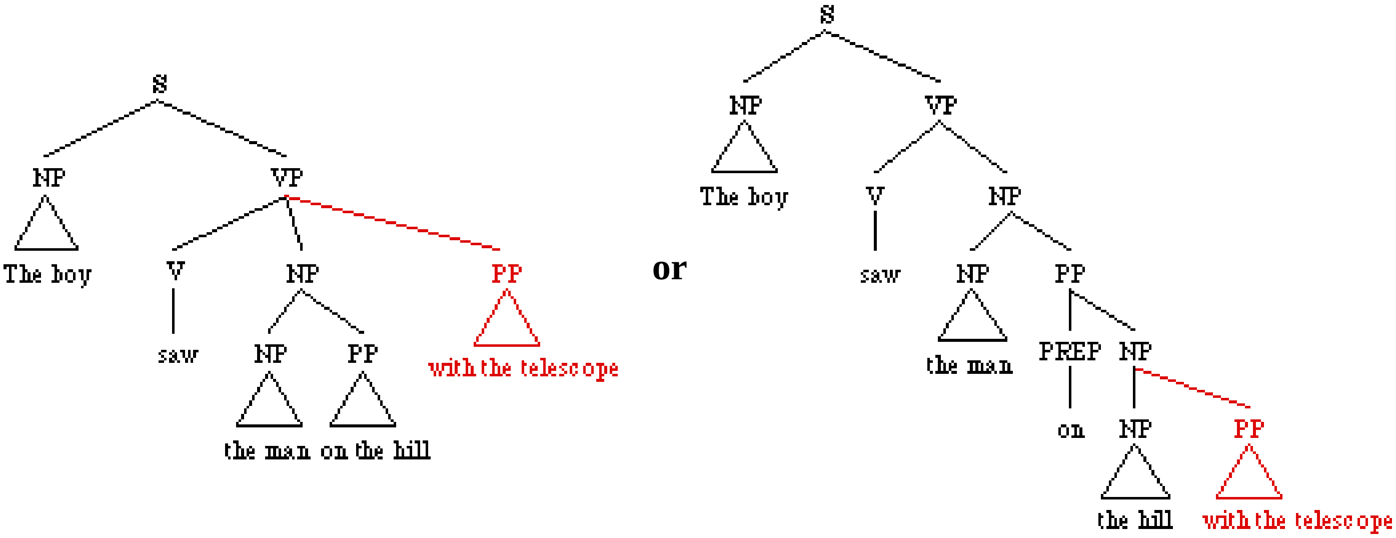
See `http://www.cse.unsw.edu.au/~cs9414/notes/nlp/dcg2.pl` for a full version of this program.

PP attachment

The boy saw the man on the hill with the telescope



The two visual interpretations correspond to two different parses (coming from different grammar rules (VP -> V NP PP and NP -> NP PP):



Other Syntax Representation and Parsing Methods

- augmented transition networks (ATN)
- generalised phrase structure grammar (GPSG)
- head driven phrase structure grammar (HDPSG)
- unification-based parsing methods
- lexical functional grammar (LFG)
- shift-reduce parsers (modified to handle ambiguous grammars)

- Marcus parsing (buffers and lookahead)
- systemic grammar (semiotics-driven)
- semantic grammars (specialise lexical categories to a range of semantic subcategories: grammar potentially becomes huge)

Limitations of Syntax in NLP

- it is reasonable to ask for syntactically correct programs, but unrealistic to ask for syntactically correct NL. Written NL material is sometimes correct, but spoken utterances are rarely grammatical. NL systems must be *syntactically and semantically robust*.

- some approaches have sought to be *semantics-driven*, to avoid the problem of how to deal with syntactically ill-formed text. However, some syntax is essential - else how do we distinguish between *Cyril loves Audrey* and *Audrey loves Cyril*?

Summary: Grammars and Parsing
There are many approaches to parsing and many grammatical formalisms. Some problems in deciding the structure of a sentence turn out to be undecidable at the syntactic level. We have concentrated on a bottom-up chart parser based on a context-free grammar. We will subsequently extend this parser to augmented grammars.

Features and Augmented Grammars

Reference: Chapter 4 of Allen

Aim:

To describe feature systems, principally syntactic ones, and how grammars may be augmented using featural restrictions. We also enhance a basic chart parser to handle features.

Keywords: agreement, augmented grammar, complement, feature, head feature, head subconstituent, SUBCAT, subcategorization, VFORM

Plan:

- agreement and features, notation, feature examples (VFORM, SUBCAT, AGR)
- augmenting grammars with features, notation, head features
- parsing with features - using augmented rules and a chart parser

This area overlaps syntax and semantics. A feature is a piece of information associated with a word.

Feature systems serve a number of goals. One simple one is **agreement**: *a man* is OK but *a men* is not. This can be enforced by attaching to nouns and articles a **feature** that indicates whether the word is singular or plural, and then reading the usual grammar rule NP -> ART N as requiring that the features of ART and N agree.

Features have a name and a value. In this case the name is AGR and the value might be s or p. (In practice, 3s or 3p for *third-person* singular/plural is more usual.)

Binary and Multiple-valued Features

Two-valued features ("binary features") are very common, and the values are often expressed as + or -. For example, the INV feature is a binary feature that indicates whether or not an S structure has an inverted subject (as in a yes/no question.)

The S structure for *Jack laughed* would have INV -, while that for *Did Jack laugh?* would have INV +.

Often the value appears as a prefix: +INV, -INV.

Sometimes features are multiple valued (e.g. "you" - AGR {2s, 2p}).

Most nouns (NPs) have AGR 3s - *the dog*, AGR 3p - *the dogs* or AGR {3s,3p} - *the sheep*. Examples for verbs: *loves* has {3s} , *love* has {1s,2s,1p,2p,3p} .

Testing for Agreement

When checking agreement with multiple-valued features, two features unify through intersection, and agree if the intersection is non-empty

For example, *the sheep love the grass* is OK because {3s,3p} intersection {1s,2s,1p,2p,3p} = {3p} is non-empty , and *the dog love the grass* is not OK because {3s} intersection {1s,2s,1p,2p,3p} is empty .

The VFORM Feature

Verb form (VFORM): this feature indicates the form of the verb, best explained by describing them all:

Form	**Description**	**Examples**
base	base form	go, be, say, decide
pres	simple present tense	go, goes, am, is, are, say, says
past	simple past tense	went, was, were, said, decided
fin	finite, i.e. pres or past	
ing	present participle	going, being, saying, deciding
pastprt	past participle	gone, been, said, decided
inf	used with infinitive forms with the word *to*	

The SUBCAT Feature

Verb complements: complement structure of a verb (i.e. what can/must follow that verb by way of NPs, etc.) can be encoded using features, too;

Complement structure is often referred to as the *subcategorisation* of the verb. Attach a SUBCAT feature to each verb to record this. Some verbs have several allowable complements. Some possibilities:

Value	**Example Verb**	**Example**
_none	laugh	Jack laughed
_np	find	Jack found a key
_np_np	give	Jack gave Sue the paper
_vp:inf	want	Jack wants to fly
_np_vp:inf	tell	Jack told the man to go

_vp:ing	keep	Jack keeps hoping for the best
_np_vp:ing	catch	Jack caught Sam cheating
_np_vp:base	watch	Jack watched Sam buy the drinks

See Figures 4.2 and 4.4 in Allen for more examples of complement structures.

* In *be*-as-copula sentences (*Adrian was sad to be rejected by the walrus*) the complement is determined by the adjective (*sad*). Thus adjectives can have SUBCAT features too.

4.4 A Simple Grammar Using Features

In this grammar, SUBCAT and VFORM features for verbs are used extensively: (VP **VFORM** inf) will be abbreviated VP[inf], and (V **SUBCAT** _np_vp:inf) as V[_np_vp:inf].

	Rule	**Example**
1	S[-inv] -> (NP AGR ?a) (VP[{pres past}] AGR ?a)	
2	NP -> (ART AGR ?a) (N AGR ?a)	a man
3	NP -> PRO	he
4	VP -> V[_none]	[he] cries
5	VP -> V[_np] NP	[he] sees her
6	VP -> V[_vp:inf] VP[inf]	[he] wants to see her
7	VP -> V[_np_vp:inf] NP VP[inf]	[he] wants her to see him
8	VP -> V[_adjp] ADJP	[he] is happy to help
9	VP[inf] -> TO VP[base]	to help
10	ADJP -> ADJ	happy
11	ADJP -> ADJ[_vp:inf] VP[inf]	happy to help

Head features for S, VP: VFORM, AGR
Head features for NP: AGR

Grammar 4.7 of Allen

Head Features

A **head feature** is one for which the feature value on a parent category must be the same as the value on the head subconstituent. Each phrasal category has associated with it a **head** subconstituent - namely N, NAME, or PRO for NP, VP for S, V for VP, and PREP for PP.

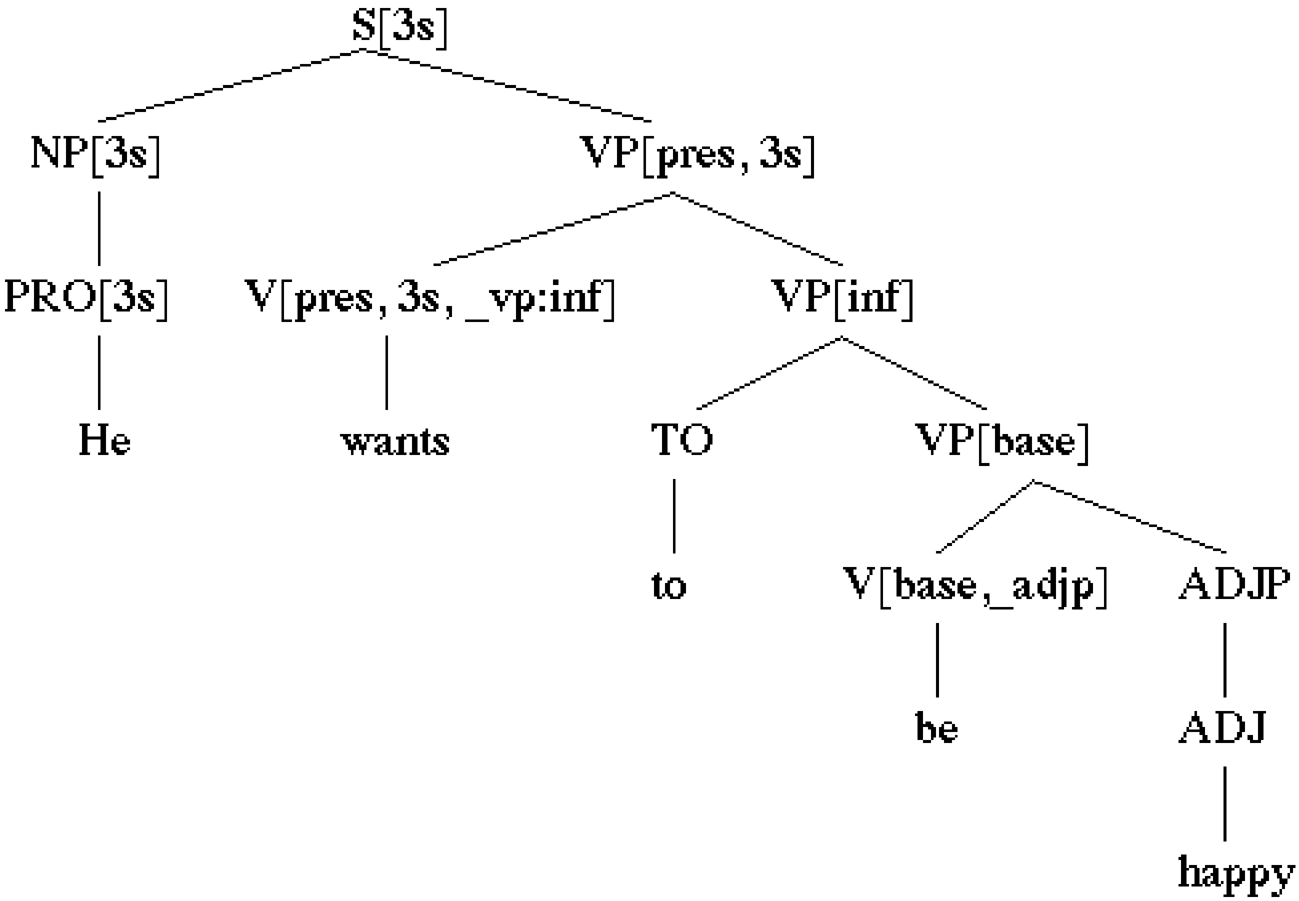
Thus the fact that AGR is listed as a head feature for VP at the foot of grammar 4.7, above, means that any rule for VP in Grammar 4.7 (i.e. rules 4-9) will implicitly copy the AGR feature for V on its right side to the AGR feature for the VP. (You can see this happening below in the expanded versin of rule 7. Head features are discussed on pages 94-96 of Allen.)

This grammar could be expanded to show all the feature values. For example, rules 11 and 7 would become

11. ADJP -> (ADJ SUBCAT _vp:inf) (VP VFORM inf)
7. (VP AGR ?a VFORM ?v) -> (V SUBCAT _np_vp:inf AGR ?a VFORM ?v) NP (VP VFORM inf)

Grammar 4.8 in Allen has the details for all the other rules.

Here is a parse tree with (abbreviated) features shown (cf. Figure 4.9 of Allen).



4.5 Parsing with Features

The chart parsing algorithm can be extended to handle context free grammars augmented with features. Consider the rule

(NP **AGR** ?a) -> (ART **AGR** ?a) (N **AGR** ?a) with the phrases *a man* and *a men*. In both cases, there is a constituent

ART1: (ART AGR 3s) -> "a" from 0 to 1

so we can form active arcs on encountering *a*, with the variable ?a being instantiated to 3s:

ARC99: (NP**AGR** 3s) -> (ART1 **AGR** 3s) * (N **AGR** 3s) from 0 to 1

In the case of *a man* since there is a constituent:

N1: (N AGR 3s) -> "man" from 1 to 2

we can extend the active arc to

ARC100: (NP**AGR** 3s) -> (ART1 **AGR** 3s) (N1 **AGR** 3s) * from 0 to 2

and then convert this to a constituent in the usual way:

NP1: (NP**AGR** 3s) -> (ART1 **AGR** 3s) (N1 **AGR** 3s) from 0 to 2

(Any extra features that the words and phrases might have have been omitted for the sake of simplicity.)
With *a men*, since *men* gives rise to:

N2: (N AGR 3p) -> "men" from 1 to 2

ARC99 cannot be extended in this case.

There is an example of a complete parse using features in Allen (page 100, Figure 4.10 and text nearby).

Summary:

Features allow us to encode extra information into grammar rules. Head features are convenient way of compacting augmented rules. The VFORM and SUBCAT features are important in handling verb phrases. Our bottom-up chart parser can be extended to handle the feature information.

Semantics and Logical Form

Reference: Chapter 8 of Allen

Aim:

To describe a language for representing logical forms - that is, intermediate representations on the way to transforming a parse tree into the final meaning representation. Logical forms must be able to encode possible ambiguities of meaning of a particular parse of a sentence.

Keywords: co-agent, compositional semantics, exists, experiencer, failure of substitutivity, FOPC, forall, instrument, logical form, logical operator, modal, MOST1, patient, PLUR, predicate operator, semantics, substitutivity, term, THE, thematic role, theme, victim

Plan:

- Definition of compositional semantics
- Word senses and ambiguity
- Logical form language - terms, predicates, propositions, logical operators, quantifiers, predicate operators, modal operators.
- Ambiguity in logical forms
- Verbs and states in logical forms - thematics roles
- Logical forms for speech acts and for embedded sentences

Syntax concerns structure; semantics concerns "meaning".

Semantics is often assumed to be *compositional*: the meaning of a phrase like *three green boxes* is constructed from the meanings of the words: *three*, *green*, and *boxes*. Similarly, the meaning of a sentence comes from the meanings of the phrases.

In practice constraints between words or phrases frequently affects the meaning:

```
The pig grunted
                        at the feminist.
                        at the farmer.
                        at the demonstrator.
```

pig = { PIG-ANIMAL, MALE-CHAUVINIST-PIG, POLICEPERSON}

Meaning involves many facets:

- relationships between objects, entities, etc.

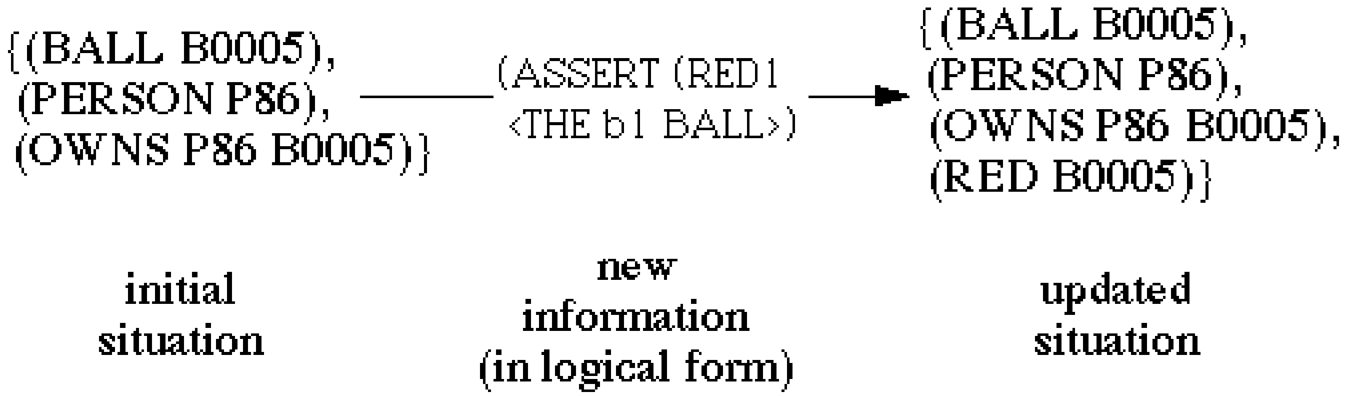
- referents for phrases like *you, him, the red gate* - i.e. objects that correspond to the words in the phrases;
- common sense inferences from basic facts conveyed
- speaker's intent: *the door is open* could be a bare statement, a complaint, a tacit request to close it, ...

Logical Form

It is possible to consider meaning in the absence of context. The representation of meaning in the absence of context is termed *logical form.*

In programming language semantics, meaning is expressed with respect to a model of a computing device. The meaning of a particular PL statement (i.e. command) is expressed by giving the effect of that statement on the model.

Modelling is used in NL semantics, too. NL semantic models are often related to those developed for predicate logic (or modal, sorted, temporal, epistemic, and higher-order logics).



Word Senses and Ambiguity

Words may have more than one **sense**. For example, *ball* could mean a social event (SOCIAL-BALL), or a spherical toy (TOY-BALL): Some words, like *take*, have many different senses (some say 57 senses for *take*).

If the intended sense is not clear, we have "word sense ambiguity". At the logical form level, words are quite often ambiguous, as it may take context to decide the intended sense. *Jack ran the race:* is Jack in charge of the race or did he participate in it?

There are other types of ambiguity:

referential ambiguity: *Jon is angry with Jim. He bites him*.

structural ambiguity: *Happy cats and dogs live on the farm.*

Every child loves a pet.

there can be mixes: *The major exhibits age.*

Co-occurrence constraints on meaning: sometimes ambiguity can be resolved by the company the word keeps, or the syntax of the sentence: *Jack ran in the park* vs *Jack ran in the election.*

The Basic Logical Form Language

This section defines a formal language of logical forms, resembling FOPC (first order predicate calculus):

terms

constants or expressions that describe objects: FIDO1, JACK1

predicates

constants or expressions that describe relations or properties: BITES1. Each predicate has an associated number of arguments - BITES1 is binary (unary = 1 argument; ternary = 3 arguments; n-ary = n arguments).

propositions

a predicate followed by the appropriate number of arguments: (BITES1 FIDO1 JACK1), (DOG1 FIDO1) - *Fido is a dog*.

More complex propositions can be constructed using **logical operators** (NOT (LOVES1 SUE1 JACK1)), (& (BITES1 FIDO1 JACK1) (DOG1 FIDO1)). Note that *and* does not always "translate" as logical & - e.g. it may suggest temporal sequence: *I went home and had a drink.*

quantifiers

in FOPC, only **forall** and **exists**. English has vaguer quantifiers, too: *most, many, a few, a, the, ...*

Variables are introduced here, as in FOPC. However variables in logical form language persist beyond the "scope" of the quantifier. *A man came in. He went to the table.* The first sentence introduces a new object of type MAN. The *He* in the second sentence refers to this object.

NL quantifiers are typically restricted in the range of objects that the variable ranges over. In *Most dogs bark* the variable in the MOST1 quantifier is restricted to DOG1 objects: (MOST1 d1 : (DOG1 d1) (BARKS1 d1))

the and *a* give rise to important NL quantifiers - *the dog barks* has logical form

(THE x : (DOG1 x) (BARKS1 x))

which would be true only if we can determine a unique dog in context, and that dog barks. How we find the unique dog is discussed in the section on *Reference*.

predicate operator

We also need a way to handle plurals as in *the dogs bark*. A new type of thing called a **predicate operator** is introduced that takes a predicate as an argument and produces a new predicate. For plurals, PLUR: if DOG1 is true of any dog, then (PLUR DOG1) is true of any set of dogs with more than one member: (THE x : ((PLUR DOG1) x) (BARKS1 x)).

modal operator

used for verbs like *believe, know, want*, for tense, and other purposes. *Sue believes Jack is happy* becomes

(BELIEVE SUE1 (HAPPY JACK1)).

Modal operators may exhibit **failure of substitutivity** - JACK1 may = JOHN22 (i.e. the individual known as Jack may also be called John, e.g. by other people), but *Sue believes John is happy* may not be true, e.g. because Sue may not know that JACK1 = JOHN22.

Tenses: use modals PRES, PAST, FUT:

(PRES (SEES1 JOHN1 FIDO1))
(PAST (SEES1 JOHN1 FIDO1))
(FUT (SEES1 JOHN1 FIDO1))

Again, substitutivity may fail. JOHN1 may = MINISTER1, and (PAST (OWNS1 JOHN1 FIDO1)) may be true, but (PAST (OWNS1 MINISTER1 FIDO1)) can still be false because John was not minister when he owned Fido.

Encoding Ambiguity in the Logical Form

Because ambiguity often cannot be resolved at the logical form level, it must be possible to represent ambiguities in the logical form language.

Sue watched the ball becomes

(THE b1: ({BALL1 BALL2} b1) (PAST (WATCH1 SUE1 b1)))

where BALL1 and BALL2 are two senses of *ball*.

Every child loves a pet becomes

(LOVES <EVERY c1 (CHILD1 c1)> <A p1 (PET1 p1)>)

making the quantifiers look like terms. This represents the two possibilities

(EVERY c1 : (CHILD1 c1) (A p1 : (PET1 p1) (LOVES1 c1 p1)))
(A p1 : (PET1 p1) (EVERY c1 : (CHILD1 c1) (LOVES1 c1 p1)))

Abbreviation: <EVERY c1 CHILD1>= (EVERY c1 : (CHILD1 c1))

Every child didn't run becomes

(<NOT RUN1> <EVERY c1 CHILD>), encompassing

(NOT (EVERY c1 : (CHILD c1) (RUN1 c1))) and
(EVERY c1 : (CHILD c1) (NOT (RUN1 c1))).

Proper Names and Pronouns

Proper names need to be interpreted in context - *John* could refer to many different Johns. This is handled in logical forms via

(NAME <variable> <name>): for example *John ran* becomes

(<PAST RUN1> (NAME j1 "John")).

Similarly for pronouns and related phenomena like *here* and *yesterday*, we use (PRO <variable> <proposition>). *Every man liked him* becomes

(<PAST LIKE1> <EVERY m1 MAN1> (PRO m2 (HE1 m2)))

where HE1 is the sense for the words *he* and *him*. As with <EVERY c1 CHILD>, (PRO m2 (HE1 m2)) can be shortened to (PRO m2 HE1) as HE1 is a unary predicate.

Verbs and States in Logical Form

Consider

1. *John broke it*

2. *John broke it with the hammer*

This seems to say that there are two predicates for *broke*, one binary and one ternary. Messy. A suitable solution is to convert the "breaking" into an event and assert suitable properties of that event.

```
1. (exists e1: (BREAK e1 (NAME j1 "John") (PRO i1 IT1))
2. (exists e1: (& (BREAK e1 (NAME j1 "John") (PRO i1 IT1))
                  (INSTR e1 <THE h1 HAMMER>)))
```

Extra modifiers can be added as needed, e.g. for *in the hallway*

Part of the intuition here is that a breaking has three fundamental roles associated with it - an **agent**, a **theme** or **victim** or **object**, and an **instrument.** It is common to label the objects in the logical form with these roles or **cases**:

```
2'. (exists e1: (& (BREAK e1)
                   (AGENT e1 (NAME j1 "John"))
                   (THEME e1 (PRO i1 IT1))
                   (INSTR e1 <THE h1 HAMMER>)))
```

Because the pattern in 1 and 2 above is so common, it is usual to leave out the "(exists e1 :" and the "& (" and write 2', for example, as

```
2''.(BREAK e1 [AGENT (NAME j1 "John"]
              [THEME (PRO i1 IT1)]
              [INSTR <THE h1 HAMMER>])
```

Strictly the BREAK should be <PAST BREAK>.

Similarly, *Mary was unhappy* could initially be

```
(<PAST UNHAPPY> (NAME m1 "Mary"))
```

but how to handle modifiers like *in the meeting*? Introduce states **s** and use

```
(<PAST UNHAPPY> s [EXPERIENCER (NAME m1 "Mary")]
                  [IN-LOC <THE m2 MEETING>])
```

Allen tends to switch between various notations as convenient.

Thematic Roles

These include AGENT, THEME, and INSTR, already met, and AT-LOC, FROM-LOC (e.g. *from here*), TO-LOC (*to the ground*), and PATH (*along the gorge*). In the domain of ownership, there are TO-POSS (*gave a book to John*), FROM-POSS, AT-POSS(*John owns a book*). In the domain of time, there are AT-TIME, FROM-TIME and TO-TIME. In the domain of values, there are AT-VALUE and FROM-VALUE and TO-VALUE (*The temperature reached 43 degrees*). There are also EXPERIENCER, for when the subject is not acting (*John believed it was raining*), BENEFICIARY (*Mary bought a present for her mother*) and CO-AGENT (*Mary lifted the table with her sister*).

Speech Acts and Embedded Sentences

Each major sentence type - assertion, yes/no-question, wh-question, command - corresponds to a **surface speech act**. We extend the logical form language by wrapping the surface speech act operator around the rest of the logical form: e.g. for *the man ate a peach*:

```
(ASSERT (<PAST EAT> e1 [AGENT <THE m1 MAN1>]
                       [THEME <A p1 PEACH 1>]))
```

Similarly for *Did the man eat a peach?*

```
(Y/N-QUERY (<PAST EAT> e1 [AGENT <THE m1 MAN1>]
                          [THEME <A p1 PEACH 1>]))
```

and *Eat the peach:*

```
(COMMAND (EAT e1 [THEME <THE p1 PEACH1>]))
```

Wh-questions are more complicated, and require a new generalized quantifier WH, with variants for *who* <WH p1 PERSON>, *what* <WH o1 ANYTHING>, *which dog* <WH d1 DOG1>, and extra quantifiers HOW-MANY and HOW-MUCH for *how many* and *how much*.

E.g. *What did the man eat?*

```
(WH-QUERY (<PAST EAT> e1 [AGENT <THE m1 MAN1>]
                         [THEME <WH w1 PHYSOBJ>]))
```

Finally for embedded sentences like relative clauses:

e.g. *The man who ate a peach left.*

```
(ASSERT
    (<PAST LEAVE> l1
         [AGENT <THE m1 (& (MAN1 m1)
         (<PAST EAT> e2
              [AGENT m1]
              [THEME <A p1 PEACH1>])))])))
```

Summary: Semantics and Logical Form

We have described a logical form language that includes terms, predicates, propositions, logical operators, quantifiers (including special NL quantifiers such as THE), and shown how this language can be used to represent ambiguous sentences.

Linking Syntax and Semantics

Reference: Chapter 9 of Allen

Aim:

To describe an interpretation algorithm that uses the syntactic analysis of a sentence together with grammar rules augmented by feature information describing how the semantic analysis of a phrase is derived from the semantic analyses of its constituents. We shall show how this works for a simple grammar and then investigate a few more complicated examples, including the handling of auxiliary verbs, and prepositional phrases.
The concepts of lambda-expression and lambda-reduction are central to the handling of certain grammar rules.

Keywords: CNP, common noun phrase, lambda reduction, VAR feature

Plan:

- Every constituent must have an interpretation - lambda-expressions, lambda-reduction
- Example: grammar rules with semantic interpretation via features
- The **VAR** feature
- Lexicon entries with **SEM** features
- Handling PPs and VPs with lambda-expressions
- Handling different types of PPs

Interpretation Algorithm

The interpretation algorithm is compositional (meaning of the whole is derived from the meanings of the parts). This means that we must be able to write down a meaning expression for each syntactic part of a sentence: S, NP, VP, PP, N, V, ...

What is the meaning of a verb phrase like the VP *kissed Sue* in *Jack kissed Sue* ? The logical form for *Jack kissed Sue* would be:

(KISS1 k1 (NAME j1 "Jack") (NAME s1 "Sue"))

Clearly there is something "missing" in *kissed Sue*, and we need to signal this. The **lambda calculus** provides a formalism for this. The expression

(lambda X (KISS1 k1 X (NAME s1 "Sue"))

is a predicate that takes one argument. The argument symbol X marks the "gap" in the VP.

Since it is a predicate, you can apply it to an argument (such as (NAME j1 "Jack")) as follows:

(lambda X (KISS1 k1 X (NAME s1 "Sue")) (NAME j1 "Jack") *

using a step called **lambda reduction** which says that (lambda *x* P(*x*))(*a*) = P(*x* | *a*) - i.e. that to apply (lambda *x* P(*x*)) to *a* you replace every occurrence of *x* in P(*x*) by *a*.

Thus the expression marked * above reduces to:

(KISS1 k1 (NAME j1 "Jack") (NAME s1 "Sue"))

PP modifiers can be handled similarly: *in the store* as in *The man in the store* or *The man is in the store* has as meaning

(lambda o (IN-LOC o <THE s1 STORE1>)).

This can be applied to <THE m1 MAN1> to produce

(IN-LOC <THE m1 MAN1> <THE s1 STORE1>).

9.2 A Simple Grammar and Lexicon with Semantic Interpretation

It is necessary to add a SEM (semantic) feature to each lexical entry and grammatical rule. For instance,

(S SEM (?semvp ?semnp)) -> (NP SEM ?semnp) (VP SEM ?semvp)

What does this do when applied to an NP with SEM (NAME m1 "Mary") and a VP with SEM

(lambda a (SEES1 e8 a (NAME j1 "Jack")))

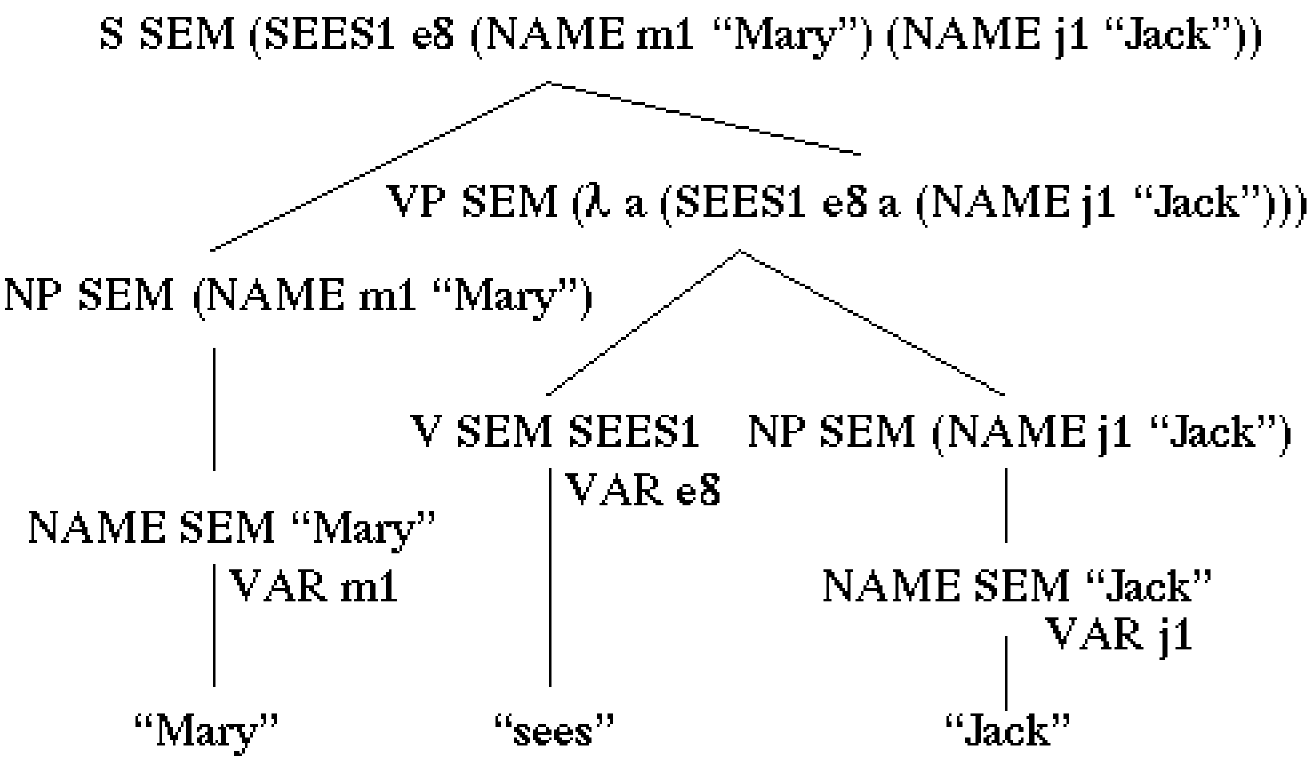
The SEM feature of the S is thus

(lambda a (SEES1 e8 a (NAME j1 "Jack"))) (NAME m1 "Mary")

which simplifies to

(SEES1 e8 (NAME m1 "Mary") (NAME j1 "Jack"))

The whole of the parse/semantic tree for this sentence (*Mary sees Jack*) is shown below (= Fig. 9.1 of Allen):



Here are some more grammatical rules with SEM features:

1	(S SEM (?semvp ?semnp) -> (NP SEM ?semnp) (VP SEM ?semvp)
2	(VP VAR ?v SEM (lambda a2 (?semv ?v a2)))-> (V[_none] SEM ?semv)
3	(VP VAR ?v SEM (lambda a3 (?semv ?v a3 ?semnp))) -> (V[_np] SEM ?semv) (NP SEM ?semnp)
4	(NP WH - VAR ?v SEM (PRO ?v ?sempro)) -> (PRO SEM ?sempro)
5	(NP VAR ?v SEM (NAME ?v ?semname)) -> (NAME SEM ?semname)
6	(NP VAR ?v SEM (?semart ?v : (?semcnp ?v)>) (ART SEM ?semart) (CNP SEM ?semcnp)
7	(CNP SEM ?semn) -> (N SEM ?semn)

Head feature for S, VP, NP, CNP: VAR

(Like Table 9.3 of Allen)

The VAR feature

The VAR feature is new and stores the "discourse variable" that corresponds to the constituent. It will be useful for handling certain forms of modifiers in the development that follows. It is automatically generated by the parser when a lexical constituent is constructed from a word, and then it is passed up the tree by treating VAR as a "head feature".

Here are some **lexical entries with semantic information**:

a	(art AGR 3s SEM A)
can	(aux SUBCAT base SEM CAN1)
car	(n SEM CAR1 AGR 3s)
cry	(v SEM CRY1 VFORM base SUBCAT _none)
decide	(v SEM DECIDES1 VFORM base SUBCAT _none)
decide	(v SEM DECIDES-ON1 VFORM base SUBCAT _pp:on)
dog	(n SEM DOG1 AGR 3s)
fish	(n SEM FISH1 AGR 3s)
fish	(n SEM (PLUR FISH1) AGR 3p)
house	(n SEM HOUSE1 AGR 3s)
has	(aux VFORM pres AGR 3s SUBCAT pastprt SEM perf)
he	(pro SEM HE1 AGR 3s)
in	(p PFORM { LOC MOT} SEM IN-LOC1)
Jill	(name AGR 3s SEM "Jill")
man	(n SEM MAN1 AGR 3s)
men	(n SEM (PLUR MAN1) AGR 3p)
on	(p PFORM {LOC, on} SEM ON-LOC1)
saw	(v SEM SEES1 VFORM past SUBCAT _np)
see	(v SEM SEES1 VFORM base SUBCAT _np IRREG-PAST + EN-PASTPRT +)
she	(pro SEM SHE1 AGR 3s)
the	(ART SEM THE AGR { 3s 3p})
to	(to AGR - VFORM inf)

(Table 9.2 of Allen)

Handling Semantic Interpretation

The chart parser can be modified so that it handles semantic interpretation as follows:

- When a lexical rule is instantiated for use, the VAR feature is set to a new discourse variable;

- whenever a constituent is built, the SEM is simplified by performing any lambda reductions that are possible.

Interpreting an example sentence:

Jill saw the dog

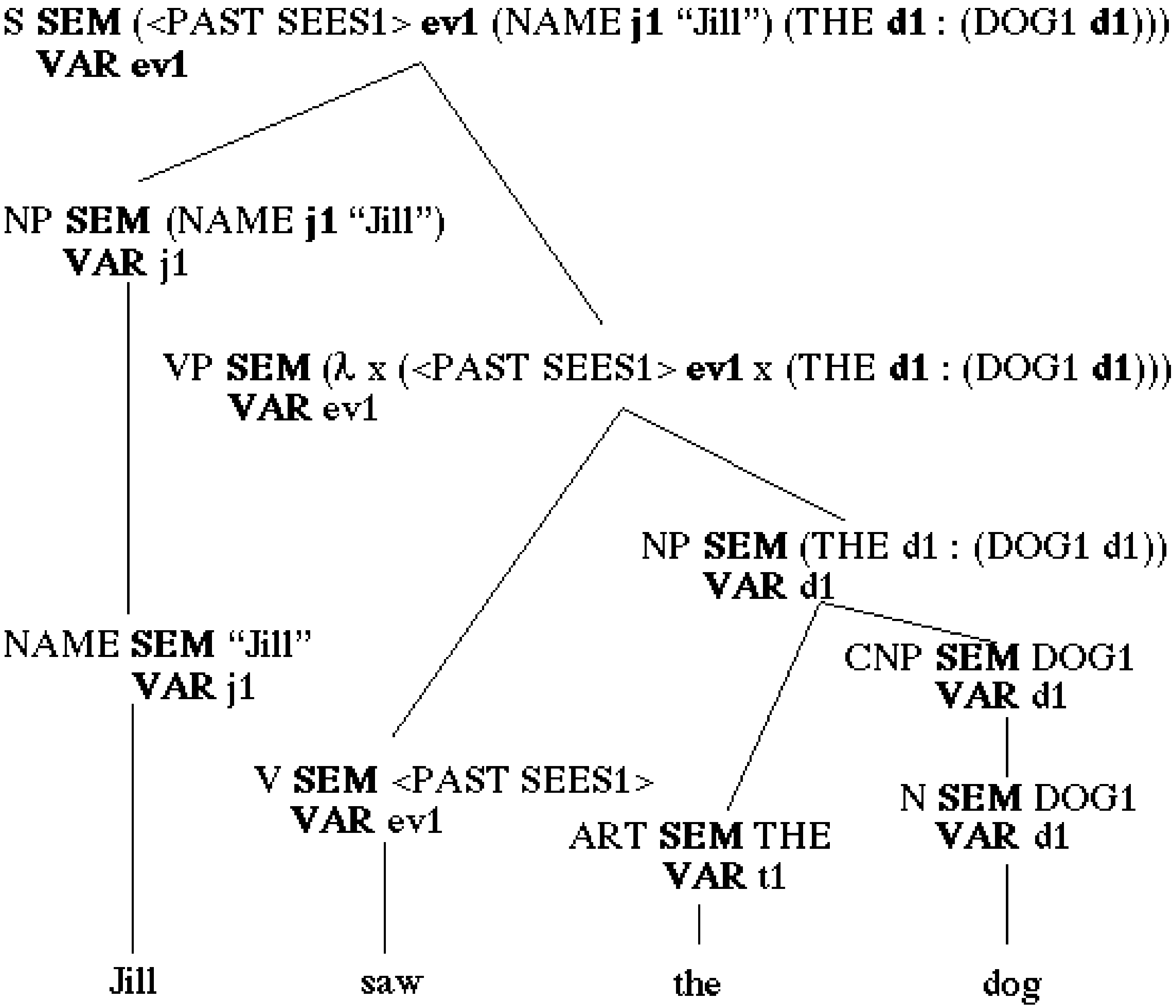
1. The word *Jill* is parsed as a name. A new discourse variable, j1, is generated, and set as the VAR feature of the NAME.
2. This constituent is used with rule 5 to build an NP. Since VAR is a head feature, VAR j1 is passed up to the NP, and the SEM is built using rule 5 to give SEM (NAME j1 "Jill")
3. The lexical entry for the word *saw* generates a V constituent with the SEM <PAST SEES1> and a new VAR ev1.
4. The lexical entry for *the* produces SEM THE.
5. The lexical entry for *dog* produces a N constituent with SEM DOG1 and VAR d1. This in turn gives rise to a CNP constituent with the same SEM and VAR, via rule 7 .
6. Rule 6 combines the SEMs THE and DOG1 with the VAR d1 to produce an NP with the SEM (THE d1 : (DOG1 d1)) and VAR d1.
7. (THE d1 : (DOG1 d1)) is combined with the SEM of the verb and its VAR by rule 3 to form a VP with VAR ev1 and SEM

   (lambda x (<PAST SEES1> ev1 x (THE d1 (DOG1 d1)))
8. This is then combined with the subject NP (NAME j1 "Jill") to form the SEM

   (<PAST SEES1> ev1 (NAME j1 "Jill") <THE d1 (DOG1 d1)>)

   and VAR ev1.

Figure 9.5 in Allen (reproduced below) shows the completed parse tree with SEM and VAR features.



9.3 Prepositional Phrases and Verb Phrases

- *auxiliary verbs*:

(VP SEM (lambda a1 (?semaux (?semvp a1)))) ->
(AUX SUBCAT ?v SEM ?semaux) (VP VFORM ?v SEM ?semvp)

If the ?semaux is a modal operator like CAN1, and ?semvp is a lambda expression such as (lambda x (LAUGHS1 e3 x)), then according to the rule above, the SEM of the VP *can laugh* is

(lambda a1 (CAN1 ((lambda x (LAUGHS1 e3 x) a1)))

which simplifies to

(lambda a1 (CAN1 (LAUGHS1 e3 a1)))

In effect the variable for the subject is lifted through the CAN1 operator. If there is more than one auxiliary, a similar process analyses the other auxiliary(ies).

- *prepositional phrases*: PPs can modify an NP or a VP, or may be subcategorized for by a head word, in which case the preposition acts more as a flag for an argument than as an independent predicate.

  Examples:

The man	**in the corner** ate lunch. PP modifies NP "the man"
The dog barked	**in the alley.** PP modifies VP "barked"
She is ready to take	**up** the challenge. "up" flags object of "take-up"

  a) *PP modifying an NP*: here the SEM of the PP is a unary predicate to be applied to the SEM of the NP:

  (PP SEM (lambda y (?semp y ?semnp))) -> (P SEM ?semp) (NP SEM ?semnp)

  Given *in the corner* if the SEM of the *in* is IN-LOC1, and the SEM of the NP is <THE c1 CORNER1>, then the SEM of the PP would be the unary predicate

  (lambda y (IN-LOC1 y <THE c1 CORNER1>))

  In the context *the man in the corner*, we need a rule to attach the PP to the CNP *man*:

  (CNP SEM (lambda n1 (& (?semcnp n1) (?sempp n1)))) ->
  (CNP SEM ?semcnp) (PP SEM ?sempp)

  In iProlog, this rule would be:

```
cnp(P1, P3, cnp(SynCNP, SynPP),
            lambda(N1, &(SemCNP(N1), LR_Result)),
            VarCNP) :-
    cnp(P1, P2, SynCNP, SemCNP, VarCNP),
    pP(P2, P3, SynPP, SemPP, VarPP),
    lambda_reduce(SemPP,N1,LR_Result).
```

The SEM of *man* is the unary predicate MAN1, so the new SEM of *man in the corner* is

(lambda n1 (& (MAN1 n1) ((lambda y (IN-LOC1 y <THE c1 CORNER1>)) n1)))

((lambda y (IN-LOC1 y <THE c1 CORNER1>)) n1))) simplifies to (IN-LOC1 n1 <THE c1 CORNER1>), so the whole expression becomes

(lambda n1 (& (MAN1 n1) (IN-LOC1 n1 <THE c1 CORNER1>)))

Combining this with *the* using rule 6, we get the SEM

(THE m2 : ((lambda n1 (& MAN1 n1) (IN-LOC1 n1 <THE c1 CORNER1>))) m2))

which simplifies to

(THE m2 : (& (MAN1 m2) (IN-LOC1 m2 <THE c1 CORNER1>)))

b) *PP modifying a VP*: e.g. *cry in the corner*, as in *Jill can cry in the corner*. The relevant syntactic rule would be VP -> VP PP. *cry* has logical form (lambda x (CRIES1 e1 x)). The desired logical form for *cry in the corner* is

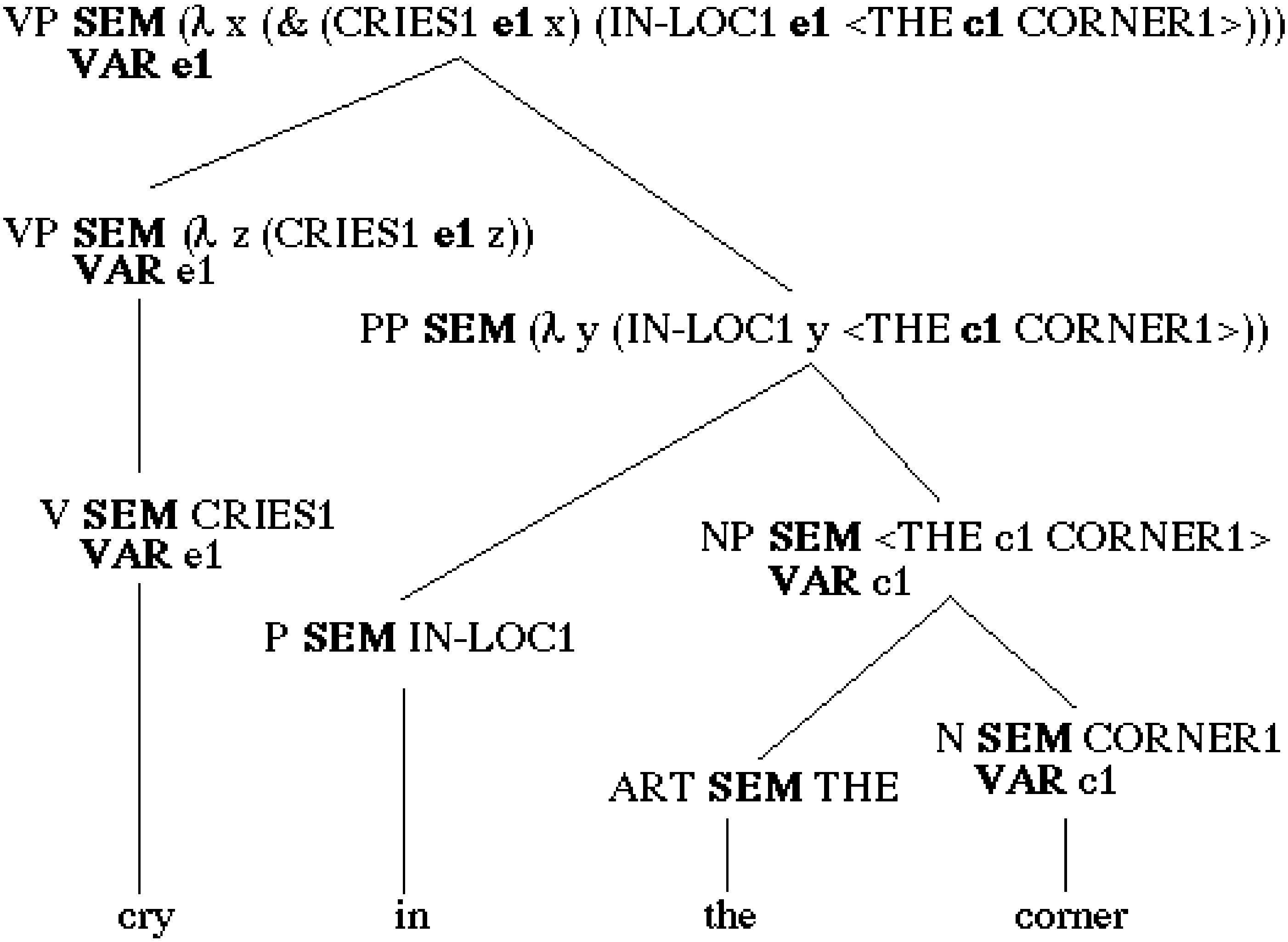
(lambda a (& CRIES1 e1 a) (IN-LOC1 e1 <THE c1 CORNER1>)))

The appropriate semantically augmented rule is:

(VP VAR ?v SEM (lambda x (& (?semvp x) (?sempp ?v)))) ->
(VP VAR ?v SEM ?semvp) (PP SEM ?sempp)

[NB: This VP rule will be superseded by a more specific one soon.]

The parse tree for the VP *cry in the corner* using this rule is shown below (Figure 9.6 of Allen, corrected):


VP SEM (λ x (& (CRIES1 e1 x) (IN-LOC1 e1 <THE c1 CORNER1>)))
VAR e1
VP SEM (λ z (CRIES1 e1 z))
VAR e1
PP SEM (λ y (IN-LOC1 y <THE c1 CORNER1>))
V SEM CRIES1
VAR e1
NP SEM <THE c1 CORNER1>
VAR c1
P SEM IN-LOC1
ART SEM THE
N SEM CORNER1
VAR c1
cry
in
the
corner

c) *PP as a subcategorized constituent in a VP*: e.g. *decide on a couch* meaning, say, (a) decide to e.g. buy a couch. There is an ambiguity here - this phrase could also mean (b) *to make a decision while sitting on a couch*. The appropriate syntactic rule for sense (a) is

VP -> V[_pp:on] PP[on]

and the desired final logical form for the VP is

(lambda s (DECIDES-ON1 d1 s <A c1 COUCH1>))

with <A c1 COUCH1> appearing as an argument. Subcategorized PPs can be handled by distinguishing between "predicate" PPs and "argument" PPs using a binary feature PRED (+ for predicate, - for argument).

A PP is a predicate in cases like "in the corner" in the previous example, where the preposition is interpreted using the predicate IN-LOC1. A PP is an argument when its interpretation, or rather the interpretation of its NP, appears as one of the arguments of the verb, as does in the lambda-expression above.

With the PRED feature available, we can have two rules for PPs, one for each case:

(PP PRED + SEM (lambda x (?semp x ?semnp))) -> (P SEM ?semp) (NP SEM ?semnp)
- interprets preposition as predicate

(PP PRED - PFORM ?pf SEM ?semnp) -> (P ROOT ?pf) (NP SEM ?semnp)
- interprets the NP in the PP as an argument of the verb

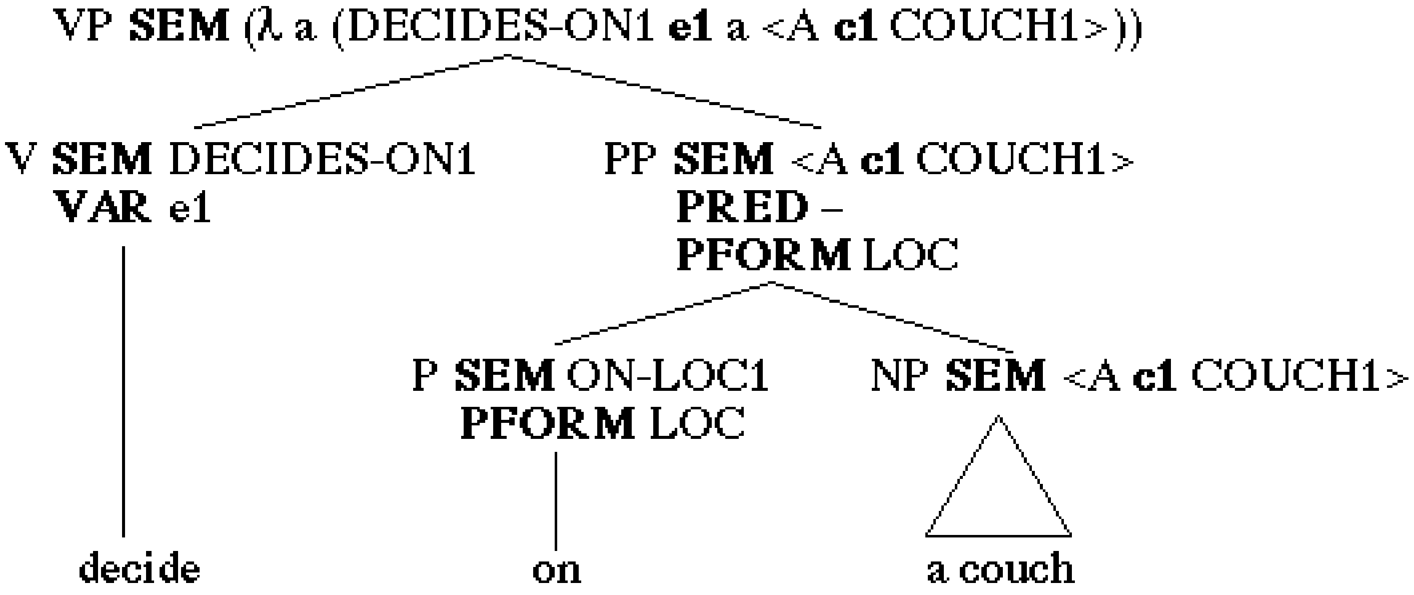
Head feature for PP: PFORM

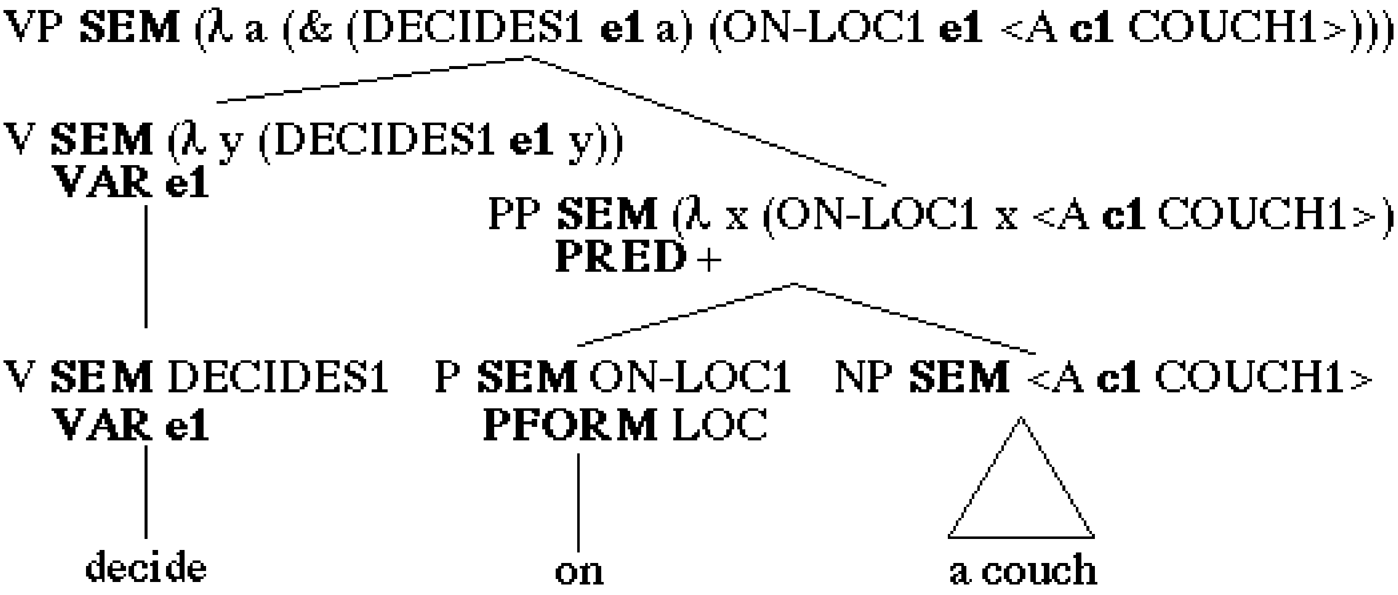
The two analyses are shown below (cf. Figure 9.8 of Allen). The analyses use VP rules found in Grammar 9.7 in Allen - these rules supersede the VP rule that we used previously:

(VP VAR ?v SEM (lambda x1 (& (?semvp x1) (?sempp ?v)))) ->
(VP SEM ?semvp) (PP PRED + SEM ?sempp)

(VP VAR ?v SEM (lambda x2 (?semv ?v x2 ?sempp))) ->
(V[_np_pp:on] SEM ?semvp) (PP PRED - PFORM on SEM ?sempp)

The upper tree is for sense (a) in which *couch* is an argument (PRED -), and the lower tree is for sense (b) in which *couch* is the location modifier for *decides*.




VP **SEM** (λ a (& (DECIDES1 **e1** a) (ON-LOC1 **e1** <A **c1** COUCH1>)))
V **SEM** (λ y (DECIDES1 **e1** y))
VAR e1
PP **SEM** (λ x (ON-LOC1 x <A **c1** COUCH1>)
PRED +
V **SEM** DECIDES1
VAR e1
P **SEM** ON-LOC1
PFORM LOC
NP **SEM** <A **c1** COUCH1>
decide
on
a couch

Summary: Semantic Interpretation

The semantic interpretation algorithm is feature-driven: reading the augmented grammar rules right to left provides a description of how to build the SEM feature of the phrase described by the grammar rule. When the semantic description has a gap (as with VP), the SEM feature is a lambda-expression.

Knowledge Representation

Reference: Chapter 13 of Allen

Aim:

To outline a representation of the facts conveyed by NL text, based on first-order predicate calculus, and to describe how quantifiers can be avoided in this notation by replacing existentially quantified variables by Skolem constants and Skolem functions, and then assuming all remaining variables are universally quantified.

Keywords: KB, knowledge base, knowledge representation, knowledge representation language, KRL, Skolem functions, skolemization

Plan:

- Knowledge Representation Languages
- Representing quantification in a KRL
- Free variables for "forall"
- Skolem constants and Skolem functions for "exists"

Knowledge Representation Languages

The knowledge representation issues of concern to NLP systems include those relating to general knowledge about the world needed for language understanding, and typically, specific knowledge relating to the domain handled by the particularly NLP system. The term **knowledge representation language** (KRL) is used to refer to the language used by a particular system to encode the knowledge. The collection of knowledge used by the system is referred to as a **knowledge base** (KB).

13.2 A Representation based on FOPC

The language used has much in common with the logical form language. The terms of the language include constants, like *John1*, function applications, like *father(John1)*, and variables, like *x* and *y*. Note that the logical form language did not use constants - everything was expressed in terms of discourse variables to keep the representation context-independent. For example, the logical form term (NAME **j1** "John"), in a specific context, might be represented by the constant *John1* in the KB.

Restricted quantification makes sense in the KRL, as it did in the logical form language. Restrictions follow the quantified variable, separated from it by a semicolon:

∃ x : Person(x) Happy(X) ..... There is a happy person

$\forall$ x : Person(x) Happy(X) ..... All people are happy

The second of these is equivalent to $\forall$ x Person(x) => Happy(X).

Free Variables Can Handle "forall"

Many KR systems do not explicitly use quantifiers. Their variables are all tacitly universally quantified, as in Prolog. Thus `eats(john1, X) :- fried(X).` means *John eats anything if it's fried*. Literally anything, of course - if `fried(cartyres1)` is stored in the KB, then `eats(john1, cartyres1)` is true.

Skolem Constants and Functions Can Handle "exists"

Existentially quantified variables are handled by a technique called **skolemization**, which replaces the variable with a new constant. For example, the formula $\exists$ *y* $\forall$ *x* . `loves`(*x*, *y*) would be encoded as a formula such as `loves(X, sk1)`, where `sk1` is a new constant that stands for the object that is asserted to exist. Quantifier scoping dependencies are shown using new functions called **Skolem functions**. For example, the formula $\forall$ *y* $\exists$ *x* . `loves`(*x*,*y*) would be encoded as a formula such as `loves(sk2(Y), Y)`, where `sk2` is a new function that produces a potentially new object for each value of `Y`.

Summary: Knowledge Representation
A taste of knowledge representation has been presented, with emphasis on the use of Skolem functions and constants

CRICOS Provider Code No. 00098G

Local Discourse and Reference

Reference: Chapter 14 of Allen

Aim:

This section concerns the problem of deciding what phrases (especially noun phrases) refer to. It introduces a simple model of global discourse structure called the history list, and presents an algorithm for referent determination in simple cases.

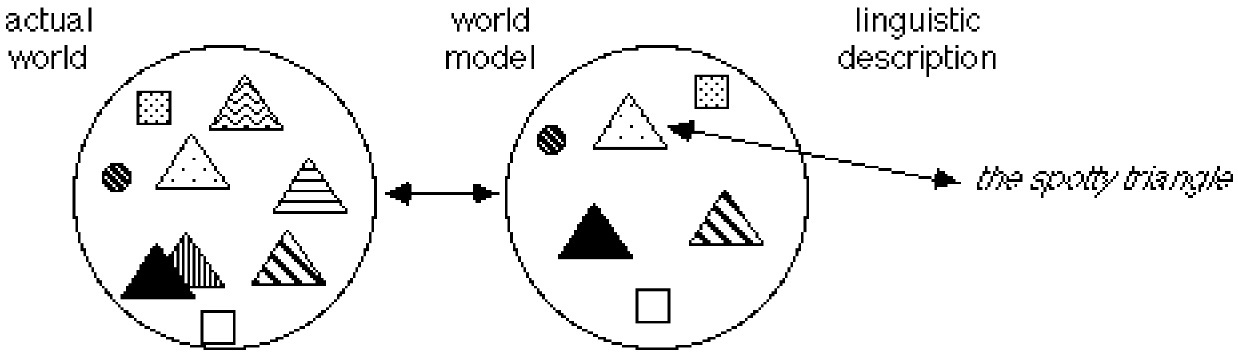
Keywords: anaphor, antecedent, cataphor, co-refer, DE, DE list, discourse entity, discourse entity list, evoke, history list, local discourse context, reference, referential ambiguity

Plan:

- anaphors
- world models
- discourse entities, DE list, evoked DEs
- creation of DEs
- quantifier problems
- history list, simple reference algorithm

This chapter concerns the issues of anaphoric reference, ellipsis, VP anaphora, and one-anaphora. It introduces a simple model of global discourse structure called the *history list*.

anaphora: (Macquarie Dict.) *Linguistics:* The explication of a word by words appearing previous to it in the text: *Mary died. She was very old*. (*She* is explained by *Mary*.)



14.1 Defining Local Discourse Context and Discourse Entities

To begin with, think of local context as including the syntactic and semantic analysis of the preceding sentence, together with a list of objects mentioned in the sentence that could be antecedents for later pronouns and definite noun phrases.

1a Jack[i] lost his wallet[j] in his car[k].
1b He[i] looked for it[j] for several hours.
1c Later he[i] found the wallet[j] under the front seat[k?].

Jack in 1a is said to be the **antecedent** for *he* in 1b: that is, the pronoun and antecedent **co-refer**.

2a Jack *forgot his wallet.*
2b Sam *did* too.

Reference is sometimes intra-sentential:

5 Jack forgot his wallet, and Sam did too.

An important part of the local context is the list of possible antecedents for pronouns, etc., which we will call the **discourse entity (DE) list**.

Sometimes a DE is not explicitly mentioned previously, but rather introduced or **evoked** (like *the front seat* in 1c).

It may also be the case that the item referred to is not precisely identified:

8a John bought a car[i] yesterday.
8b It[i] was very expensive.

Neither speaker nor hearer may be able to identify the car (we can't say that it is CAR56 in our knowledge base (KB).) How can we refer to such an object - in effect a constant whose identity is unknown to us. In effect we have to introduce a new constant into our (logical) language whenever we meet an expression like <A **c1** CAR1> - perhaps C1. If we also introduce a constant I1 for the expression <PRO i1 IT1>, then we can capture co-reference by asserting I1 = C1.

Generating Discourse Entities

Discourse entities are generated for each noun phrase. In many cases, there will be constraints on the entities:

indefinite noun phrases:
    creates new DE - details below
proper names:
    describe some object often already in KB
definite noun phrases and pronouns:

refer to objects previously mentioned or known, often DE in local context

complex NP like *John and Mary*:

evokes a set consisting of all the conjuncts

Indefinite Noun Phrases within the Scope of Universals

10a Three boys[i] each bought a pizza[j].
10b They[i] ate them[j] in the park.

Initial translation of 10a:

$$\begin{array}{l} \exists\, B : |B| = 3 \;\&\; \{x|Boy(x)\} \supset B \\ \quad \forall\, b : b \in B\ . \\ \qquad \exists\, p : Pizza(p)\ .\ Buy(b, p) \end{array}$$

Discourse entities from 10a:

$$\begin{array}{l} B1 : |B1| = 3 \;\&\; \{x|Boy(x)\} \supset B1 \\ P1 : P1 = \{x \mid Pizza(x) \;\&\; \exists\, y \in B1\ .\ x = sk4(y)\} \end{array}$$

Semantic content:

$$\forall\, b : b \in B1.\ sk4(b) \in P1 \;\&\; Buy1(b, sk4(b))$$

14.2 A Simple Model of Anaphora Based on History Lists

The **history list** is a list of discourse entities generated by the preceding sentences, with the most recent listed first. Some systems retain just the last two or three sentences, while some let the list grow indefinitely. The DEs evoked by each sentence are referred to as a *local context*.

The algorithm for finding an antecedent for a pronoun proceeds as follows:

1. check the most recent local context for an antecedent that matches all the constraints related to the pronoun. Constraints include number, gender, and more complex grammatical phenomena such as the rules for the use of reflexive pronouns. Semantic constraints imposing selectional restrictions may occur too: e.g. in *There was cheese on the table. Jack ate it.* the word *it* cannot refer to *the table*.

2. if no referent is found in the current local context, then move down the history list to the next most recent local context and search there.

Definite Descriptions

Similar techniques will often work with definite descriptions.

18a Helen bought a car[i] and a boat yesterday.
18b She paid too much for the car[i].

The phrase *the car* behaves somewhat like an *it* with the constraint of being of type *car*.

Indirect Reference

Sometimes the antecedent is not something mentioned directly in a preceding context, but nevertheless introduced in one.

20a My club held a raffle.
20b The winner won a car.
22a Jack brought a pencil to class.
22b However, he found that the lead was broken.

In 20a/b, the *winner* is introduced by the *raffle*, and in 22a/b the *lead* is a subpart of the *pencil*. With 20a/b, one can interpret *the winner* as $\exists$! w : Win(w, *PRO*) where *PRO* is a "pro-form", which can then be looked for in the local context.

In 22a/b, it is rather more awkward: interpret *the lead* as $\exists$! l : Lead(l) & *R*(l, *PRO*), where this time the relation *R* and the pro-form *PRO* must be identified (the answer should be *R* = SubPartOf, and *PRO* = Pencil1, where Pencil1 is the DE generated from *a pencil* in 22a).

Summary: Local Discourse and Reference

Concentrating mostly on anaphoric reference to noun phrases, we have described a simple algorithm for deciding what object in the history list a phrase in the text refers to. The key ideas presented include searching the history list in most-recent-first order, using relationships between objects to determine indirect references. We also looked briefly at the effects of the use of quantifiers like "each" on the referent of an object within the scope of the quantifier.

Ambiguity Resolution - Statistical Methods

Reference: Allen, Chapter 7

Aim:

To develop enough theory to describe algorithms for disambiguating the syntactic categories of the words, choosing the most likely parse, and enhancing parser efficiency, using statistical techniques, and frequency data on the number of times that:

(a) a word occurs in different lexical categories;
(b) a lexical category occurs following another lexical category (or sequence of categories).
(c) a particular grammar rule is used

Keywords: Bayes' rule, bigram, conditional probability, corpus, hidden Markov model, HMM, independence (statistical), lexical generation probability, n-gram, output probability, part-of-speech tagging, statistical NLP, structural ambiguity, tagged corpus, trigram, Viterbi algorithm

Plan:

- basic probability: conditional probability, Bayes' rule, independence
- part-of-speech identification, corpus, tagged corpus
- estimating probabilities from word frequencies in a corpus
- part-of-speech tagging
- bigrams, trigrams, n-grams
- estimating bigram probabilities
- experiments with a synthetic corpus
- graphical representation of bigram probabilities
- Markov assumption, hidden Markov models
- Viterbi algorithm
- forward probability, backward probability
- probabilistic context-free grammars
- inside probability
- best-first parsing
- context-dependent best-first parser

7.1 Basic Probability Theory

Probabilities are expressed as numbers between 0 and 1. 0 means impossible, and 1 means certain. A **random variable**, basically means a set of mutually exclusive possible outcomes of some event, such as tossing a coin. In the case of tossing a coin, the outcomes are H (heads) and T (tails). Each outcome has a probability: since H and T are equally likely, and the outcome { H **or** T} is certain (probability 1) Pr(H) = Pr(T) = 0.5. In general, if there are n possible outcomes e_1, ..., e_n, then Pr(e_1) + ... Pr(e_n) = 1.

Conditional Probability

Suppose a horse Harry ran 100 races and won 20. One would say Pr(Win) = 0.2. Suppose also that 30 of these races were in wet conditions, and of these, Harry won 15. It looks like Harry does better in the rain: the probability that Harry would win given that it was raining would be 0.5. *Conditional probability* captures this idea: we write Pr(Win|Rain) = 0.5. Formally, the conditional probability of and event e given an event e' is defined by the formula

$$Pr(e \mid e') = Pr(e \,\&\, e') / Pr(e')$$

Bayes' Rule

By the definition, Pr(A | B) = Pr(A & B) / Pr(B) and Pr(B | A) = Pr(B & A) / Pr(A), so Pr(A | B) / Pr(B | A) = Pr(A) / Pr(B), or

$$Pr(A \mid B) = Pr(B \mid A) * Pr(A) / Pr(B)$$

This is called Bayes' rule, and it is sometimes useful in estimating probabilities in the absence of some of the information.

Independence

Two events are said to be **independent** of each other if the occurrence of one does not effect the probability of the occurrence of the other: Pr(e | e') = Pr(e). Using the definition of conditional probability, this is equivalent to saying Pr(e & e') = Pr(e) * Pr(e').

Let's try to relate this to NLP.

Problem: Given a sentence with ambiguous words, determine the *most likely lexical category* for each word. (This is called *part-of-speech identification*.)

Say you need to identify the part-of-speech of words that can be either nouns or verbs. This can be formalized using two random variables: one, C, ranges over the parts of speech (N and V) and the other, W, ranges over all the possible words. Consider an example where W = *flies*. The problem can be stated as determining whether Pr(C = N | W = *flies*) or Pr(C = V | W = *flies*) is greater. Often we shall write things like this as Pr(N | *flies*) for short.

Using the definition of conditional probability, we have

Pr(N | *flies*) = Pr(*flies* & N) / Pr(*flies*)
Pr(V | *flies*) = Pr(*flies* & V) / Pr(*flies*)

So what we really want is Pr(*flies* & N) vs Pr(*flies* & V).

These probabilities can be estimated from a **tagged corpus**. A *corpus*, in this context, means a collection of natural language sentences. The collection might be made from newspaper articles, for example. "Tagged" means that each word in the corpus is labelled ("tagged") with its part of speech.

Suppose we have a corpus of simple sentences containing 1,273,000 words, including say 1000 uses of the word *flies*, 400 of them as an N and 600 of them as a V.

Pr(*flies*) = 1000 / 1,273,000 = 0.00078554595
Pr(*flies* & N) = 400 / 1,273,000 = 0.00031421838
Pr(*flies* & V) = 600 / 1,273,000 = 0.00047132757

Our best guess in this case will thus be that V is the most likely part-of-speech. (And we should be right about 60% of the time.) Of course, we could have seen that directly from the counts, but in more complicated instances, we shall need the actual probabilities.

Incidentally, we can now estimate Pr(V | *flies*):

Pr(V | *flies*) = Pr(*flies* & V) / Pr(*flies*)
= 0.00047132757 / 0.00078554595
= 0.6, which also = count(*flies* & V) / count(*flies*)

7.2 Estimating Probabilities

Probability-based methods rely on our ability to estimate the probabilities. Accurate estimates assume vast amounts of data. This is OK for common words.

However, consider the Brown corpus, a fairly typical corpus of about a million words. It has about 49,000 distinct words. So, on average, each word occurs about 20 times. But in fact, over 40,000 words occur 5 times or less (with *any* part of speech). For these words, the probability estimates will be very crude.

Some low-frequency words may occur zero times with one of the parts-of-speech that they can in fact take. The probability estimate is then 0 for that part of speech.

One technique for dealing with this is to add a small number, like 0.5, to the count of each part of speech for a word before computing the probabilities.

7.3 Automatic Part-of-Speech Tagging

Automatic POS-tagging can bd done by selecting the most likely sequence of syntactic categories for the words in a sentence, and using the syntactic categories in that sequence as the tags for the words in the sentence.

Our method in section 7.1 (always select the most frequent category) works 90% of the time on a per word basis (mainly because lots of words are unambiguous). So to be worth considering, a method needs to improve on 90%.

The standard method is to use some of the local context for the word. With our *flies* example, if the word was preceded by *the*, then the probability of N increases dramatically.

Let w_1, ..., w_T be a sequence of words. We want to find the sequence of lexical categories C_1, ..., C_T that maximizes

1. $Pr(C_1, ..., C_T \mid w_1, ..., w_T)$

Unfortunately, it would take far too much data to generate reasonable estimates for such sequences directly from a corpus, so direct methods cannot be applied.

However, there are approximation methods. To develop them, we must restate the problem using Bayes' rule, which says that this conditional probability equals

2. $Pr(C_1, ..., C_T) * Pr(w_1, ..., w_T \mid C_1, ..., C_T) / Pr(w_1, ..., w_T)$

The common denominator $Pr(w_1, ..., w_T)$ does not affect the answer, so in effect we want to find a sequence C_1, ..., C_T that maximizes

3. $Pr(C_1, ..., C_T) * Pr(w_1, ..., w_T \mid C_1, ..., C_T)$

These probabilities can be approximated by probabilities that are easier to collect by making some independence assumptions. These independence assumptions are not valid, but the estimates work reasonably well in practice.

Bigrams and Trigrams

The first factor in formula 3, Pr(C_1, ..., C_T) , can be approximated by a sequence of probabilities based on a limited number of previous categories, say 1 or 2 previous categories. The **bigram** model uses Pr(C_i | C_{i-1}). The **trigram** model uses Pr(C_i | C_{i-2} C_{i-1}). Such models are called **n-gram** models.

Using bigrams, the following approximation can be used:

Pr(C_1, ..., C_T) approx. = Product(i=1,T) Pr(C_i | C_{i-1})

To account for the beginning of a sentence, we invent a pseudocategory <start> at position 0 as the value of C_0. Then for the sequence ART N V N using bigrams:

Pr(ART N V N) approx. = Pr(ART | <start>) Pr(N | ART) Pr(V | N) Pr(N | V)

The second factor in formula 3,

Pr(w_1, ..., w_T | C_1, ..., C_T)

can be approximated by assuming that a word appears in a category independent of the words in the preceding or succeeding categories - it is approximated by:

Product(i=1,T) Pr(w_i | C_i).

With these approximations, we are down to finding the sequence C_1, ..., C_T maximizing

Product(i=1,T) Pr(C_i | C_{i-1}) Pr(w_i | C_i)

These probabilities can be estimated from a tagged corpus. For example, the probability that a V follows an N can be estimated as follows:

Pr(C_i = V | C_{i-1} = N) approx. = Count(N at position i-1 and V at i) / Count(N at position i-1)

Bigram Frequencies with a Toy Corpus

Table 7.4 in Allen gives some bigram frequencies for an artificially generated corpus of simple sentences.

The corpus consisted of 300 sentences and had words in only four categories: N, V, ART, and P, including 833 nouns, 300 verbs, 558 articles, and 307 prepositions for a total of 1998 words. To deal with the problem of sparse data, any bigram not listed in Figure 7.4 was assumed to have a token probability of 0.0001.

Category	**Count at i**	**Pair**	**Count at i,i+1**	**Bigram**	**Estimate**
<start>	300	<start>,ART	213	Pr(Art|<start>)	.71
<start>	300	<start>,N	87	Pr(N|<start>)	.29
ART	558	ART,N	558	Pr(N|ART)	1
N	833	N,V	358	Pr(V|N)	.43
N	833	N,N	108	Pr(N|N)	.13
N	833	N,P	366	Pr(P|N)	.44
V	300	V,N	75	Pr(N|V)	.35
V	300	V,ART	194	Pr(ART|V)	.65
P	307	P,ART	226	Pr(ART|P)	.74
P	307	P,N	81	Pr(N|P)	.26

Table 7.4 Bigram probabilities from the generated corpus

The lexical generation probabilities Pr(W_i | C_i) can be estimated simply by counting the number of occurrences of each word by category. Figure 7.5 in Allen gives some counts for individual words, from which the probability estimates in Table 7.6 in Allen are generated.

	N	**V**	**ART**	**P**	**TOTAL**
flies	21	23	0	0	44
fruit	49	5	1	0	55
like	10	30	0	31	61
a	1	0	201	0	202
the	1	0	300	2	303
flower	53	15	0	0	68
flowers	42	16	0	0	58
birds	64	1	0	0	65
others	592	210	56	284	1142
TOTAL	833	300	558	307	1998

Table 7.5 A summary of some of the word counts in the corpus

Pr(the | ART)	0.54 .	Pr(a | ART)	0.360
Pr(flies | N)	0.025	Pr(a | N)	0.001
Pr(flies | V)	0.076	Pr(flower | N)	0.063
Pr(like | V)	0.1	Pr(flower | V)	0.05
Pr(like | P)	0.068	Pr(birds | N)	0.076
Pr(like | N)	0.012		

Figure 7.6 The lexical generation probabilities

Given N categories and T words, there are N^T possible tag-sequences. Luckily, it is not necessary to consider all of these because of the independence assumptions that were made about the data.

Finite-state Machine Model

Since we are only dealing with bigram probabilities, the prob ability that the i-th word is in category C_i depends only on the category of the i-1-th word, C_{i-1}. Thus the process can be modelled by a special type of finite state machine, as shown in Fig. 7.7. Each node represents a possible lexical category & the transition probabilities (i.e. the bigram probabilities from Table 7.4) show the probability of one category following another.

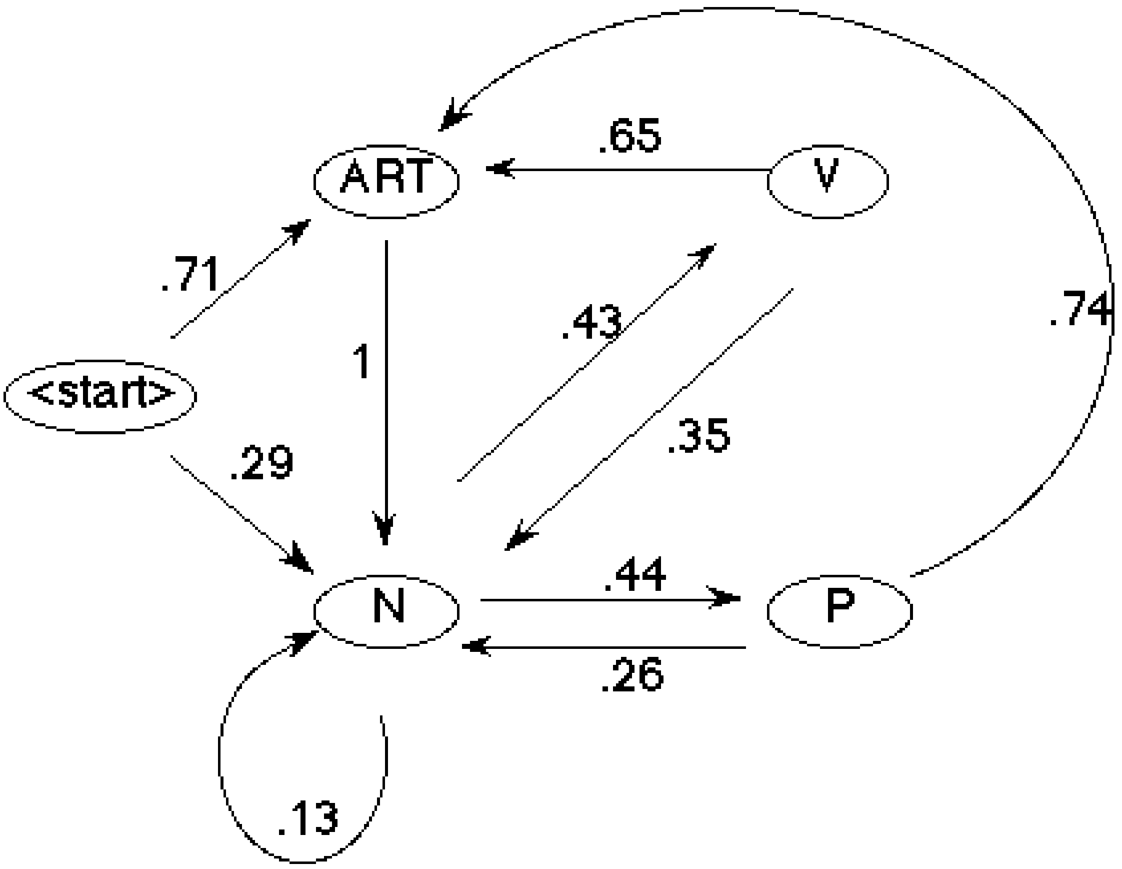


Figure 7.7: A network capturing the bigram probabilities

With such a network, you can compute the probability of any sequence of categories simply by finding the path through the network indicated by the sequence and multiplying the transition probabilities together. Thus, the sequence ART N V N would have probability 0.71 * 1 * 0.43 *0.35 = 0.107. The validity of this depends on the assumption that the probability of a category occurring depends only on the immediately preceding category. This assumption is referred to as the **Markov assumption**, and networks like Fig. 7.7 are called **Markov chains**.

The network representation can now be extended to include the lexical generation probabilities as well: we allow each node to have an **output probability** which gives a probability to each possible output that could correspond to the node. For instance the node N in Fig. 7.7 would be associated with a probability table that gives, for each word, how likely that word is to be selected if we randomly select a noun. The output probabilities are exactly the lexical generation probabilities from Fig. 7.6. A network like that in Figure 7.7 with output probabilities associated with each node is called a **Hidden Markov Model** (HMM).

The probability that the sequence N V ART N generates the output *Flies like a flower* is computed as follows:

The probability of the path N V ART N given Fig. 7.7, is .29 * .43 * .65 * 1 = 0.081. The probability of the output being *Flies like a flower* is computed from the output probabilities in Fig. 7.6:

Pr(flies | N) * Pr (like | V) * Pr(a | ART) * Pr(flower | N)
= 0.025 * .1 * .36 * .063
= 5.4 E-5

Multiplying these together gives us the likelihood that the HMM would generate the sequence: 4.37 E-6. More generally, the probability of a sentence w_1, ..., w_T given a sequence C_1, ..., C_T is

Product(i=1,T) $Pr(C_i \mid C_{i-1}) Pr(w_i \mid C_i)$

Back to Most-likely Tag Sequences

Now we can resume the discussion of how to find the most likely sequence of tags for a sequence of words. The key insight is that because of the Markov assumption, you do not have to enumerate all the possible sequences. Sequences that end in the same category can be collapsed together since the next category only depends on the previous one in the sequence. So if you just keep track of the most likely sequence found so far for each possible ending category, you can ignore all the other less likely sequences.

Fig. 7.8 does *Flies like a flower* in this way.

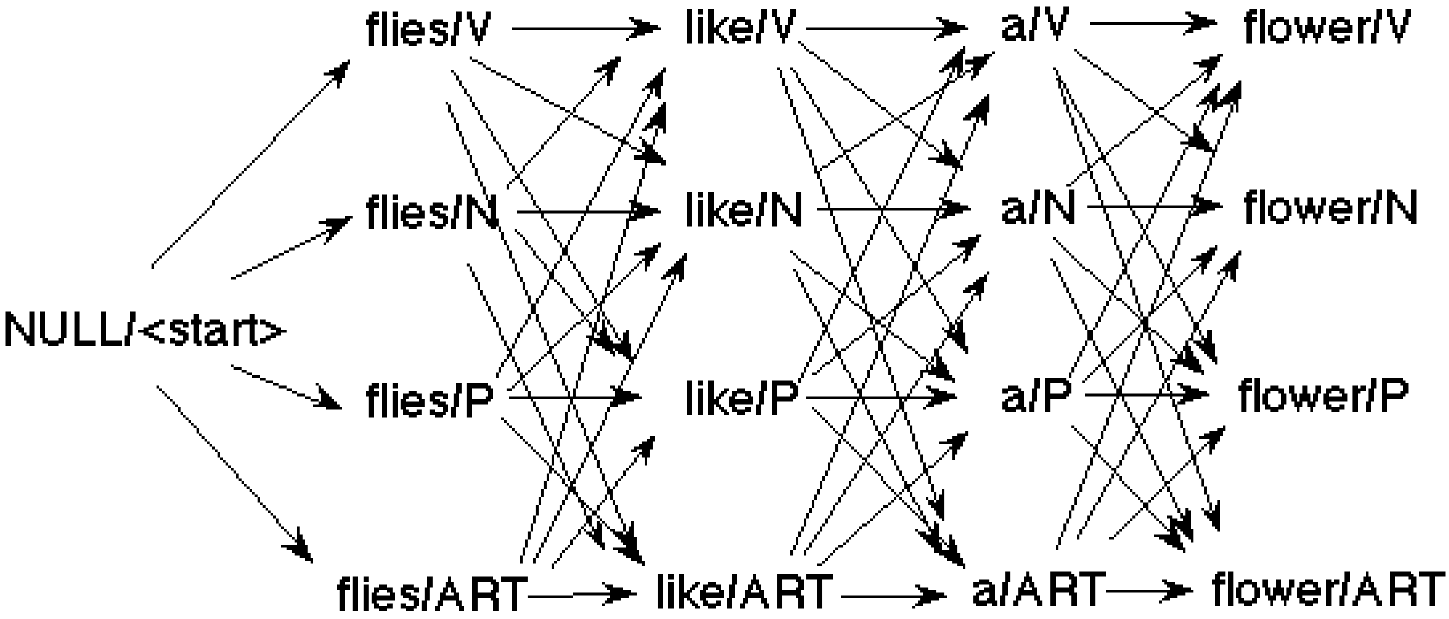


Figure 7.8: Encoding the 256 possible sequences, exploiting the Markov assumption

To find the most likely sequence, sweep forward through the words one at a time finding the most likely sequence for each ending category. In other words, you find the four best sequences for the two words *Flies like*: the best ending with like as a V, the best as an N, the best as a P and the best as an ART. You then use this information to find the four best sequences for the the words *flies like a*, each one ending in a different category. This process is repeated until all the words are accounted for. This algorithm is usually called the **Viterbi algorithm**. For a problem involving T words, and N lexical categories, the Viterbi algorithm is guaranteed to find the most likely sequence using k*T*N*N steps for some constant k.

We will not describe the Viterbi algorithm any further (see p. 202-204 in Allen for the details).

7.4 Obtaining Lexical Probabilities

We have already seen the simple method of estimating lexical probabilities by counting the number of times that each word w appears in a corpus in each category and dividing the counts by the total number of times that the word w appears:

Pr(L_j | w) approx = count(L_j & w) / Sum i=1,N count(L_i & w)

where L_1, ..., L_N are the possible categories.

A better estimate: compute how likely it is that category L_i occurred at position t over all sequences of categories for the input w_1, w_2, ..., w_t. For example, consider the probability that *flies* is a noun in *The flies like flowers*: sum the probabilities of all sequences that end with *flies* as a noun. Using the data in Figs. 7.4 & 7.6, the possibilities are:

The/ART	flies/N	$9.58*10^{-3}$ (= 0.71*0.54*1*0.025) = Pr(ART|emptyset)*Pr(the|ART)*Pr(N|ART)*Pr(flies|N)
The/N	flies/N $4.55*10^{-9}$	
The/P	flies/N	$1.13*10^{-6}$

which adds up to $9.58*10^{-3}$. Similarly, 3 sequences with non-zero probability end with *flies* as a V, with a total probability of $1.13*10^{-5}$. No other non-zero probability sequences ending in *flies* exist, so the sum $9.58*10^{-3} + 1.13*10^{-5} = 9.591*10^{-3}$ is the probability of the sequence *the flies*. Then the probability that *flies* is a noun in this case is

Pr(*flies*/N | *the flies*) = Pr(*flies*/N & *the flies*) / Pr(*the flies*) = 9.58 $*10^{-3}$ / $9.591*10^{-3}$ = 0.9988

and Pr(*flies*/V | *the flies*) = 0.0012.

To develop this more precisely, we define the **forward probability** $\alpha_i(t)$ which is the probability of producing the words w_1, ..., w_t and ending in state w_t/L_i.

$\alpha_i(t) = Pr(w_t/L_i, w_1, ..., w_t)$.

For example, Pr(*flies*/N , *the flies*) would be $\alpha_1(2)$ assuming that N is L_1.

We can derive that:

$\Pr(w_t/L_i \mid w_1, ..., w_t) = \alpha_i(t)$ / Sum j=1,N $\alpha_j(t)$

and a variant of the Viterbi algorithm can be used to compute the forward probabilities (see Fig. 7.14 in Allen if interested).

This can be followed through to obtain the following figures for the final word *flowers* in *the flies like flowers:*

Pr(*flowers*/N | *the flies like flowers*) = 0.967

Pr(*flowers*/V | *the flies like flowers*) = 0.033

You could also consider the **backward probability** $\beta_i(t)$ - the probability of producing $w_t, ..., w_T$ beginning from state w_t/L_i. In the end, a better method of estimating the lexical probabilities is to consider the whole sentence, using the estimate

Pr(wt/Li) = $(\alpha_i(t)*\beta_i(t))$ / Sum j=1,N $(\alpha_j(t)*\beta_j(t))$

7.5 Probabilistic Context-Free Grammars

Just as we can accumulate statistics on the use of lexical grammatical categories (e.g. how many times N → "flies" and N → "flowers" etc. are used) so it is possible to get statistics on phrasal category use (e.g. how many times NP → ART N

and NP → N N are used). One can obtain the statistics by counting the number of times each rule is used in a corpus containing parsed sentences.

Suppose that there are *m* rules $R_1, ..., R_m$ with left-hand side C. You can estimate the probability of using rule R_j to derive C by

$\Pr(R_j \mid C)$ = count(#times R_j used) / Sum i=1,m (#times R_i used)

Grammar 7.17 shows a probabilistic CFG based on a parsed version of our example corpus:

Rule	Count for LHS	Count for Rule	Probability

1.	S → NP VP	300	300	1
2.	VP → V	300	116	.386
3.	VP → V NP	300	118	.393
4.	VP → V NP PP	300	66	.22
5.	NP → NP PP	1023	241	.24
6.	NP → N N	1023	92	.09
7.	NP → N	1023	141	.14
8.	NP → ART N	1023	558	.55
9.	PP → P NP	307	307	1

Grammar 7.17

You can then develop algorithms similar in functioning to the Viterbi algorithm that will find the most likely parse tree for a sentence. Certain invalid independence assumptions are involved. In particular, you must assume that the probability of a constituent being derived by a rule R_j is independent of how the constituent is used as a subconstituent. For example, this assumption would imply that the probabilities of NP rules are the same whether the NP is the subject, the object of a verb, or the object of a preposition, whereas, for example, subject NPs are much more likely to be pronouns than other NPs.

The formalism can be based on the probability that a constituent C generates a sequence of words w_i, w_{i+1}, ... , w_j, written as $w_{i,j}$. This probability is called the **inside probability** and written $\Pr(w_{i,j} \mid C)$.

The probabilities of specific parse trees for a sentence can be found using a standard chart parsing algorithm, where the probability of each constituent is computed from the probability of its subconstituents and the probability of the rule used. When entering an item E of category C on the chart, using rule *i* with *n* subconstituents corresponding to chart entries E_1, ..., E_n:

$$\Pr(E) = \Pr(\text{Rule } i \mid C) * \Pr(E_1) * \ldots * \Pr(E_n)$$

Ultimately a parser built using these techniques doesn't work as well as you might hope, though it does help. Typically it has been found that these techniques identify the correct parse 50% of the time. One critical issue is that the model assumes that the probability of a a particular verb being used in a VP rule is independent of which rule is being considered.

For example, Grammar 7.17 says that rule 3 is used 39% of the time, rule 4 22%, and rule 5 24%. These between them mean that irrespective of what words are used, an input sequence of the form V NP PP will always be parsed with the PP attached to the verb, since the V NP PP interpretation has a probability of .22, whereas the version that attaches the PP to the NP has a probability of .39 * .24 = 0.093. This is true even with verbs that rarely take a _np_pp complement.

7.6 Best-First Parsing

So far, probabilistic CFGs have done nothing for parser efficiency. Algorithms developed to explore high-probability constituents *first* are called **best-first parsing** algorithms. The hope is that the best parse will be found first and quickly.

Our basic chart parsing algorithm can be modified to consider most likely constituents first. The central idea is to make the agenda a **priority queue -** the highest rated elements are always first in the queue, and the parser always removes the highest-ranked constituent from the agenda before adding it to the chart.

Some slight further modifications are necessary to keep the parser working. With the modified agenda system, if the last word in the sentence has the highest score, it will be added to the chart first. Previous versions of the chart parsing algorithm have assumed that we could safely work left-to-right. This no longer holds. Thus, whenever an active arc is added to the chart (e.g. by "moving the dot") it may be that the next constituent needed to extend the arc is already in the chart, and we must check for this.

The complete new algorithm is shown in Fig. 7.22.

Adopting a best-first strategy makes a significant improvement in parser efficiency. Using grammar 7.17 and the lexicon information from the corpus, the sentence *The man put a bird in the house* is parsed correctly after generating 65 constituents. The standard bottom-up chart parser generates 158 constituents from the same sentence.

The best-first parser is guaranteed to find the highest probability parse (see Allen p. 214 for proof, if interested).

---

to add a constituent C from position p_1 to p_2:

1. Insert C into the chart from position p_1 to p_2.
2. For any active arc of the form $X \rightarrow X_1 \ldots * C \ldots X_n$ from position p_0 to p_1, add a new active arc $X \rightarrow X_1 \ldots C * \ldots X_n$ from position p_0 to p_2.

to add an active arc $X \rightarrow X_1 \ldots C * C' \ldots X_n$ to the chart from position p_0 to p_2:

1. If C is the last constituent (that is, the arc is completed), add a new constituent of type X to the agenda.
2. Otherwise, if there is a constituent Y of category C' in the chart from p_2 to p_3, then *recursively* add

an active arc X → X_1 ... C C' * ... X_n from p_0 to p_3 (which may of course add further arcs or create further constituents).

Figure 7.22 The new arc extension algorithm

7.7 A Simple Context-Dependent Best-First Parser

The best-first algorithm improves efficiency but not accuracy. This section explores a simple alternative method of computing rule probabilities that uses more context-dependent lexical information. The idea exploits the fact that the first word in a constituent is often the *head word* and thus has a big effect on the probabilities of rules that account for the constituent.

This suggests a new probability measure for rules that takes into account the first word: Pr(R | C, w):

$$\mathbf{Pr(R\ \ |\ C,\ w) =}\ \frac{\text{count(\# times rule R used for cat C starting with w)}}{\text{count(\# times cat C starts with w)}}$$

How this helps: for instance, in the corpus, singular nouns rare occur alone as a noun phrase (i.e. NP → N), and plural nouns are rarely used as a noun modifier (i.e. starting rule NP → N N) - this can be seen from the statistics in Fig. 7.23:

Rule	**the**	**house**	**peaches**	**flowers**
NP → N	0	0	.65	.76
NP → N N	0	.82	0	0
NP → NP PP	.23	.18	.35	.24
NP → ART N	.76	0	0	0
Rule	**ate**	**bloom**	**like**	**put**
VP → V	.28	.84	0	.03
VP → V NP	.57	.10	.9	.03
VP → V NP PP	.14	.05	.1	.93

Figure 7.23 Some rule estimates based on the first word

Also, these context-sensitive rules encode verb preferences for different subcategorizations - see the bottom half of Fig. 7.23. A parser based on these probabilities does substantially better at getting PP attachment right (but still gets 34% wrong). It also turns out to be more efficient (36 constituents, compared with 65 (best-first) and 158 (vanilla parser) for *The man put the bird in the house*).

Further details in Allen, pp. 217-218.

Summary: Ambiguity Resolution - Statistical Methods
The concepts of independence and conditional probability, together with frequency statistics on parts-of-speech and grammar rules obtained from an NL corpus, allow us to work out the most likely lexical categories for words in a sentence, to order the parsing agenda for best-first parsing. We found in all the cases we examined that context-dependent methods work best.

```
% Welcome to rules.pl (~cs9414/public_html/notes/kr/rules/rules.pl, in full).
% To run the system, save a copy in your home directory, then (on a host that
% runs iProlog), do:
%     % prolog rules.pl
%     : wm(X)? % should reply that the only working memory facts are a, b, c.
%     : trace! % optional - this makes Prolog show the goals it's executing
%     : run!   % runs rule-based system. See note below.
%     : notrace! % turn off tracing
%     : wm(X)? % this time it should say that a, b, c, d, & e are all facts.
%     : ^D
% Note: as there there are two rules (rule1 and rule2), and they are both
%       eligible to fire given that a, b, and c are all true, both rules should
%       fire, and the conclusion d of rule1 and the conclusion e of rule2
%       should both be facts when execution finishes.
%
% Original code by Claude Sammut, date lost in the mists of time.
% Modified for iProlog by Bill Wilson, October 2001.
%    iF is so spelled because "if" is predefined in iProlog.
% One day I'll get around to checking compatibility with a more
% widely used prolog interpreter ... Bill W

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% To express rules in a natural way, we define some operators
% First argument to op is the precedence
% Second argument is the associativity (left, right, nonassoc)
% Third argument is operator name (or list of names)
% This is basically a cosmetic trick to avoid writing the rules as
% iF(rule1, then(and(a, and(b, c)), d).
% iF(rule2, then(and(a, b), e).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

op(900, xfx ,iF)!
op(800, xfx, then)!
op(700, xfy, and)!

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Example Rules
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

rule1
iF a and b and c then d.

rule2
iF a and b then e.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initial Working memory
% We use "wm" to distinguish working memory elements from
% other entries in Prolog's data base
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

wm(a).
wm(b).
wm(c).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Reset
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 1. Assert a dummy already_fired(_,_) fact so retract will not fail if there
%    are no real such facts to retract.
% 2. Next retract all already_fired(_,_) assertions, in case we are rerunning
%    the system and there are some old assertions lying around from last time.
% 3. Now re-assert the dummy already_fired(_,_) fact so that "can_fire" will
%    be confused by the fact that there is no such predicate as already_fired
%    the first time around the match-resolve-act cycle.
% 4. Finally, the cut prevents this predicate endlessly cycling, asserting and
%    retracting.

init :-
        assert(already_fired(null, false)),
        retract(already_fired(_, _)),
        assert(already_fired(null, false)), !.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Start execution
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

run :-
        init,
        exec.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Select a rule, fire it and repeat until no rules are satisfied
% f "fire" succeeds, cut will prevent backtracking
% If "fire" fails, the cycle will repeat
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

exec :-
    repeat,                % Always succeeds on backtracking
    select_rule(R),        % Select a single rule for firing
    fire(R), !.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% "findall" collects all solutions to "can_fire"
% resolve selects one of those rules
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

select_rule(SelectedRule) :-
    findall(Rule, can_fire(Rule), Candidates),
    resolve(Candidates, SelectedRule).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Find a rule that hasn't fired before and has its conditions satisfied
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

can_fire(RuleName iF Condition then Conclusion) :-
    RuleName iF Condition then Conclusion,    % Look up rule in data base
    not(already_fired(RuleName, Condition)),  % Has it already fired?
    satisfied(Condition).                     % Are all conditions satisfied?


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% If pattern is "A and ..." then look for A in working memory
% then check rest recursively.
%
%    (A and B) = (x and y and z)
%    A = x
%    B = y and Z
%
% If pattern is a single predicate. Look it up.
% Note that "!" prevents a conjunction reaching the second clause
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

satisfied(A and B) :- !,
    wm(A),
    satisfied(B).
satisfied(A) :-
    wm(A).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Very simple conflict resolution strategy: pick the first rule
% Also check in case no rules were found
%
% Exercise: rewrite this to choose the rule which has the most conditions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

resolve([], []).
resolve([X|_], X).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Add "already_fired" clause to data base so that this instance of the rule
%    is never fired again.
% Add all terms in conclusion to database, if not already there
% Fail to force backtracking so that a new execution cycle begins
%
% If there is no rule to fire, succeed. This terminates execution cycle
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

fire(RuleName iF Condition then Conclusion) :- !,
    assert(already_fired(RuleName, Condition)),
    add_to_wm(Conclusion),
    fail.
fire(_).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% For each term in condition
%    add it to workking memory if it is not already there.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

add_to_wm(A and B) :- !,
    assert_if_not_present(A),
    add_to_wm(B).
add_to_wm(A) :-
    assert_if_not_present(A).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% If term is in working memory, don't do anything
% Otherwise, add new term to working memory.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

assert_if_not_present(A) :-
    wm(A), !.
assert_if_not_present(A) :-
    assert(wm(A)).
```

Program Testing

These notes are formulated for Prolog programming, but many of the ideas can be applied to programming in functional or procedural languages.

Testing your program is a necessary part of program creation.

Except for trivial programs, testing can never prove that a program is correct. What testing does is to try to find errors in your code. No matter how many errors you find and fix, there may be more. Still, tested code is better than untested code.

Designing Tests

If you write a Prolog procedure to help some other procedure, test the helper procedure in isolation before you try it out inside the procedure it is helping. For example, suppose you want to be able to determine the maximum of two numbers. Suitable Prolog code is:

```
min(A, B, B) :- A > B.
min(A, B, A) :- A <= B.
```

To ensure that your procedure works, you should try it out on examples of all the likely scenarios. For example, with `min`, the three likely categories of data are where A > B, where A = B, and where A < B. So you should do at least one test case for each of these:

```
min(1, 2, X)?
min(3, 3, X)?
min(5, 4, X)?
```

and make sure it gives the correct answer for each.

When you believe that `min` is working correctly, you can start testing the procedure that calls `min`.

When you are dealing with lists, replacing an item in a list, say, then again you should look for a range of critical situations - for example:

- What happens if the item to be replaced is not in the list?
- What happens if the item to be replaced is the first item in the list?
- What happens if the item to be replaced is the last item in the list?
- What happens if the item to be replaced is in middle of the list?
- What happens if the item to be replaced occurs more than once in the list?
- What happens if the list is empty? [If the empty list is an allowable argument.]
- What happens if the list is long? You may have an algorithm that works, in the sense that given enough time, it will compute the correct thing, and when you run it on a list of 3 or 4 items it seems to be fine, but if you run it on a list of 30 items, it might take 372 years to complete the computation, when a better algorithm would complete the computation (correctly :-) in less than a microsecond. Don't laugh - somebody handed an assignment like that in 2004. Solution: try it on longer lists!

If your procedure has more than one rule in it, then you should make sure that your testing covers each rule. In particular, if you write a recursive procedure, then you should make sure that you cover the base case and the recursive case.

Tracing

In response to any `iProlog` prompt (": "), you can type `trace!`. This has the effect of turning on tracing of Prolog execution. This can sometimes be helpful in figuring out what your program is doing that it shouldn't be doing. Traces can however produce more output than one can easily work through, so it's usefulness is limited.

There is an example of tracing in iProlog here.

To turn tracing off, type `notrace!` in response to a iProlog prompt.

Tips

As you develop a program, you are likely to have to test it several times, as you find each bug and try to fix it. It can be quite time-consuming to repeatedly type in the test queries over and over again. You can automate this to some extent by putting your queries into a file - call it `mytests` say - and then, if your program is in the file `myprog.pl`, you can do the Unix command

```
prolog myprog.pl <mytests
```

Then Prolog will load the code in `mycode.pl` and then treat the contents of `mytests` as if you typed the contents of `mytests` into Prolog interactively. All that will be printed, however, will be the output of the queries. If you can figure out which query caused which output, then that's OK. If not, then you can add extra code to `mytests` so that Prolog prints a message from time to time so you can tell where it is up to. For example,

```
write('Testing min now'), nl,
write('First test, min(1,2,X)?'), nl,
min(1,2,X)?
```

`nl` means print a "newline" - that is terminate the current line of output and move to the next line. [If you're wondering how `write` and `nl` fit into a logic-language like Prolog, it's like this: `write` and `nl` are built-in Prolog predicates that can be thought of as always succeeding, and which have the *side-effect* of printing something in the window from which you are running Prolog.]

You can also insert `write`s and `nl`s into your rules in order to keep track of what's going on, though to a large extent the `trace` does this sort of thing for you when turned on.

```
% Version of factorial(N, FactorialN) with a bug in it.
factorial(0, 1).
factorial(N, Result) :-
        write("Entering recursive rule for factorial with N = "), write(N), nl,
        Nminus1 is N - 1,
        factorial(Nminus1, Nminus1Factorial),
        Result is N * Nminus1Factorial.
```

Try copying and pasting this code into Prolog, and then typing the goal `factorial(5, Result)?` and see what happens. Can you figure out the bug? Solution - but have a go yourself, first!

Some people prefer to copy and paste individual tests into the prolog interpreter rather than repeatedly running all of the tests.

Don't forget to retest your code when/if you move it from your home computer to one of the School of Computer Science and Engineering's computer systems. "It worked at home" is a not uncommon cry of anguish: we test your program on the machines in CSE.

School of Computer Science & Engineering
The University of New South Wales
Sydney 2052, AUSTRALIA
Email: billw at cse.unsw.edu.au
Phone: +61 2 9385 6876
Fax: +61 2 9385 5995

Tips on Writing Recursive Procedures in Prolog

The Basics - Base Case and Recursive Case(s)

When writing a recursive procedure in Prolog, there will always be at least two rules to code: at least one rule for the *base case*, or non-recursive case, and at least one rule for the *recursive case*.

The Base Case

Typically the base-case rule(s) will deal with the smallest possible example(s) of the problem that you are trying to solve - a list with no members, or just one member, for example. If you are working with a tree structure, the base case might deal with an `empty` tree, or a tree with just one node in it (like `tree(empty, a, empty)`).

The Recursive Case

To write the recursive case rule(s), you need to think of how the current case of the problem could be solved assuming you had already solved it for all smaller cases. For example, if you were adding a list of 10 numbers, and you had a way of summing the last 9 numbers in the list, then you would do so, then add on the first number on the list. (It might seem more natural to add up the *first* 9 numbers and then add the last number to the subtotal, but in Prolog it is easy to access the first item in the list, but not the last.)

Example 1: adding up a list of numbers

Here's an example of how to apply this to adding up a list of numbers. The comments beginning %% would not normally appear - they are there this time to help you match the scheme described in the previous paragraphs to the code.

```
% addup(List, Sum)
% Binds Sum to the sum of the numbers in the list.
% Assumes that each member of the list really is a number.
% The List must be instantiated at the time addup is called.
% Example of use:
% addup([1,2,3,4], X)?
% X = 10

%% base case
addup([], 0). % sum of the empty list of numbers is zero

%% recursive case: if the base-case rule does not match, this one must:
addup([FirstNumber | RestOfList], Total) :-
        addup(RestOfList, TotalOfRest),   % add up the numbers in RestOfList
        Total is FirstNumber + TotalOfRest.
```

Example 2: finding the last item of a list

This example shows how to find the last item in a list:

```
% lastitem(List, Last)
% Binds Last to the item at the end of List.
% List must be instantiated at the time of call, and must not be empty.
% Example of use:
% lastitem([a,b,c,d], X)?
% X = d

%% base case: list with just one item.
lastitem([OnlyOne], OnlyOne).

%% recursive case: ignore first item, seek last item of rest of list
lastitem([First | Rest], Last) :-
        lastitem(Rest, Last).
```

Example 3: Squaring each item in a list of numbers

This example shows how to build a result that is a list, i.e. we are transforming a list to produce a new list.

This time we'll start with a version with bugs (= errors) in it and also a stylistic error, and then we'll correct the errors. With each version of the code, you should look carefully at it before reading on, and try to spot the bug (or stylistic error). Spotting the bugs and stylistic errors is something you will have to do for yourself when you write your own programs, so start on it now!

NB: `square_1`, `square_2`, `square_3`, and `square_4` **are all wrong**. Only `square_5` is correct.

For the sake of brevity in this exposition, I've left out the comments in all but the final version. In practice you would develop the comments as you wrote the code. However, remember that just because you say something is true about the code, in your comments, doesn't mean it is. And don't forget to review and if necessary correct your comments when you find and fix a bug.

```
square_1([First | Rest], [First * First | SquareRest]) :-
        square_1(Rest, SquareRest).
```

Try it out:

```
: square_1([1, 3, 5], Squares)?
** no
```

Oops - we left out the base case. If you turn on tracing before you execute the query above, you will be able to see 1 * 1, 3 * 3, and 5 * 5 being produced, but because there is no base case, Prolog cannot complete the query.

```
square_2([], []).
square_2([First | Rest], [First * First | SquareRest]) :-
```

```
        square_2(Rest, SquareRest).
```

Try it out again:

```
: square_2([1, 3, 5], Squares)?
Squares = [1 * 1, 3 * 3, 5 * 5]
```

That's better, but why didn't it work out that 1 * 1 = 1, 3 * 3 = 9, etc.?
Answer: because we didn't ask it to. In Prolog, in most cases, evaluation of an arithmetic expression must be explicitly called for using the built-in predicate `is`.

```
square_3([], []).
square_3([First | Rest], [Square | SquareRest]) :-
        FirstSquared is First * First,
        square_3(Rest, SquareRest),
        Square = FirstSquared.
```

Try it:

```
: square_3([1, 3, 5], Squares)?
Squares = [1, 9, 25]
```

Whee! It works. But it contains a stylistic problem. The final goal `Square = FirstSquared` is redundant - it can be done better by simply writing `FirstSquared` instead of `Square` in the head of the rule:

```
square_4([], []).
square_4([First | Rest], [Firstsquared | SquareRest]) :-
        FirstSquared is First * First,
        square_4(Rest, SquareRest).
```

Always try it out after any change:

```
: square_4([1, 3, 5], Squares)?
Squares = [_R36, _R55, _R74]
```

Say what? Why doesn't it work any more? Oh. We mistyped `FirstSquared` as `Firstsquared` in the head of the rule. Prolog cares desperately about upper and lower case letters, and the "s" of Squared is lower case in the head of the rule, but upper case in the first goal of the body. This can lead to a range of mystifying error messages - the type above, with one or more unresolved `_R`... variables, is one possibility.

```
%% base case
% nothing to square; result is empty list.
square_5([], []).
%% recursive case
% square the First item, recursively square the Rest.
```

```
square_5([First | Rest], [FirstSquared | SquareRest]) :-
        FirstSquared is First * First,
        square_5(Rest, SquareRest).
```

Always try it out - even if you just added comments, in case you forgot to put a % sign at the front of a comment line.

```
: sqare_5([1, 3, 5], Squares)?

ERROR: Undefined predicate.
In:      sqare_5([1, 3, 5], _R15) ?
```

What NOW!!!??? Oh. We mis-typed `square_5` as `sqare_5` in the *query*.

```
: square_5([1, 3, 5], Squares)?
Squares = [1, 9, 25]
```

Whew!

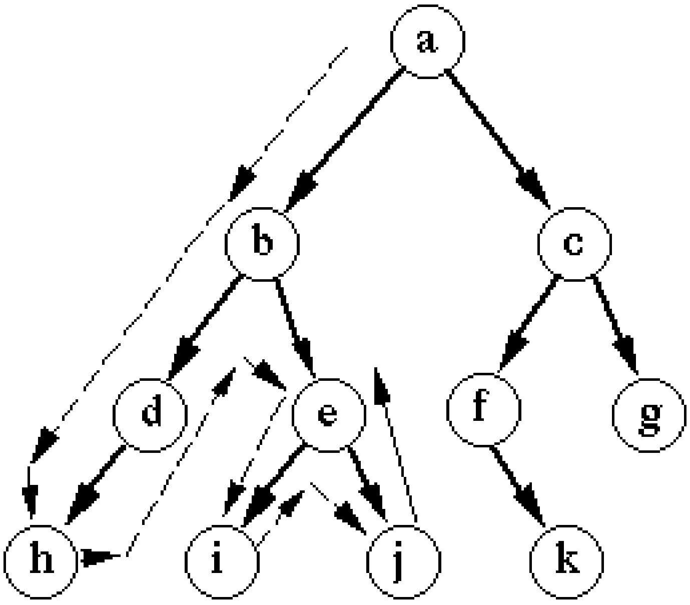
Bill Wilson's contact info

Last updated:

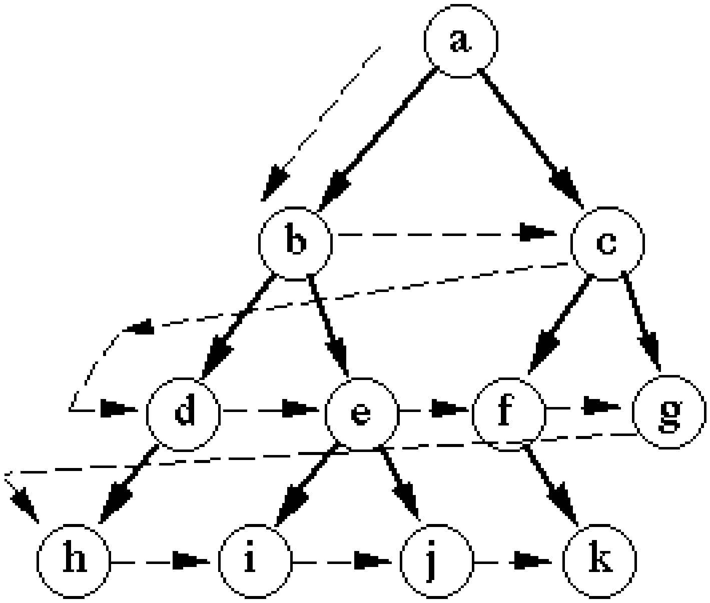
Methods of Search

References:	Depth-first search	Bratko section 11.2
	Breadth-first search	Bratko section 11.3
	Best-first search	Bratko chapter 12

In the missionaries and cannibals example, the state space is explored in an order determined by Prolog. In some situations, it might be necessary to alter that order of search in order to make search more efficient. To see what this might mean, here are two alternative methods of searching a tree.



Depth-first search



Breadth-first search

Depth first search begins by diving down as quickly as possible to the leaf nodes of the tree (or graph). Traversal can be done by:

- visiting the node first, then its children (*pre-order* traversal): `a b d h e i j c f k g`
- visiting the children first, then the node (*post-order* traversal): `h d i j e b k f g c a`
- visiting some of the children, then the node, then the other children (*in-order* traversal) `h d b i e j a f k c g`

Breadth-first search traverses the tree or graph level-by-level, visiting the nodes of the tree above in the order `a b c d e f g h i j k`.

Implementing Depth-First Search

Hereafter, assume that we have a graph, represented using the `edge` predicate, and that a node N is a goal node if `goal(N)` is true.

```
solve(Node, Solution) :-
  depthfirst([], Node, Solution).

% depthfirst(Path, Node, Solution).
depthfirst(Path, Node, [Node|Path]) :-
  goal(Node).
depthfirst(Path, Node, Sol) :-
  edge(Node, Node1),
  not(member(Node1, Path)),
  depthfirst([Node|Path], Node1, Sol).

: edge(1, 2).         %        1
: edge(1, 3).         %       / \
: edge(2, 4).         %      2   3
: edge(3,5).          %      |   |
: edge(5,6).          %      4   5
: goal(6).            %          |
: solve(1, Soln)?     %          6 <- goal
Soln = [6, 5, 3, 1]
```

Implementing Breadth-First Search

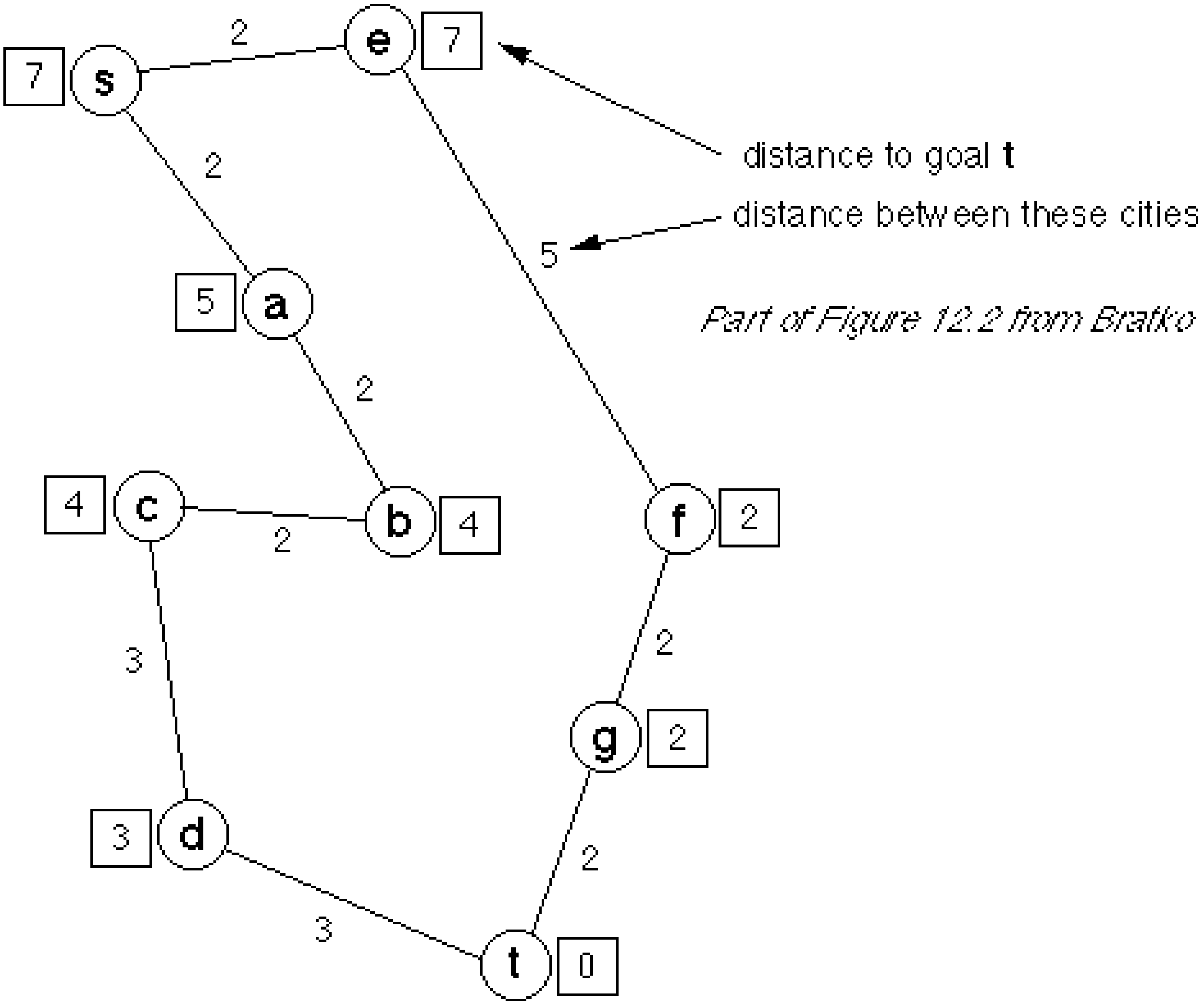
```
% solve(Start, Solution).
%  Solution is a path (in reverse order)
%  from Start to a goal.

solve(Start, Solution) :-
  breadthfirst([[Start]], Solution).

% breadthfirst([Path1, Path2, ...], Solution).
%  Solution is an extension to a goal of
%  one of the paths.

breadthfirst([[Node|Path]|_], [Node|Path]) :-
  goal(Node).
  % Always first check if goal reached

% If not, then extend this path by all
% possible edges, put these new paths on the
% end of the queue (Paths1) to check, and do
% breadthfirst on this new collection of
% paths, Paths1:
breadthfirst([Path|Paths], Solution) :-
  extend(Path, NewPaths),
  conc(Paths, NewPaths, Paths1),
  breadthfirst(Paths1, Solution).

% extend([N|Rest], NewPaths).
% Extend the path [N|Rest] using all edges
% through N that are not already on the path.
% This produces a list NewPaths.
extend([Node|Path], NewPaths) :-
  findall([NewNode, Node|Path],
          (edge(Node, NewNode),
           not(member(NewNode,[Node|Path]))),
          NewPaths),
  !.
extend(Path,[]). %findall failed: no edge

edge(1, 2). edge(1, 3). edge(2, 4).
edge(3, 5). edge(5, 6).
goal(3).
```

Best-first search (A* algorithm).

In some situations, we have partial knowledge of the structure of the search space that can be applied to guide search.

We can inspect all the currently-available transitions, and rank them on the basis of our partial knowledge. Here high rank means that the transition looks promising in relation to the goal.

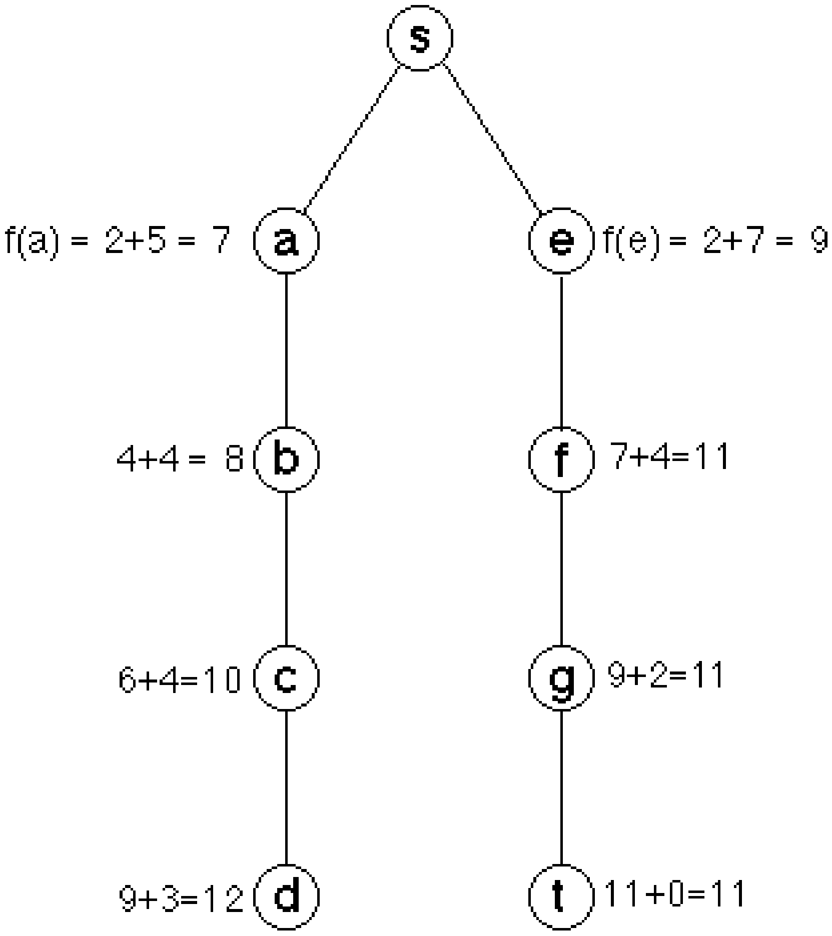
We'll describe the best-first algorithm in terms of a specific example involving distances by straight line and by road from a start point s to a goal point t:


s
7
e
7
2
2
5
a
5
2
b
4
c
4
2
f
2
g
2
2
2
3
d
3
3
t
0
distance to goal **t**
distance between these cities
Part of Figure 12.2 from Bratko

Let us define, for any node N, g(N) to be the distance travelled from the start node s to reach N. Note that this is a known quantity by the time you reach N, but that in general it could vary depending on the route taken through state space from s to N.

In our example scenario, we don't know the distance by road from N to t, but we do know the straightline distance. Let us call this distance h(N). As our heuristic to guide best-first search, we use f(N) = g(N) + h(N). That is, we will search first from the node that we have found so far that has the lowest f(N).

Let's see how this will work in hand-simulation:



Part 2 of Figure 12.2 from Bratko, ed. 3

To understand the code for `bestfirst`, we need to know about some representations used by the program:

- edge(N1, N2, Cost) signifies an edge between vertices N1 and N2 with cost (or distance) Cost.
- h(N, H) signifies that the h value of node N is H (i.e. h(N) = H).
- lf(N, F/G) represents a single node tree - a leaf - N is a node in the state space, G is g(N), and F is f(N) = G + h(N).

- tr(N, F/G, Subs) represents a tree with non-empty subtrees - Subs is a list of the subtrees, and F is the *updated* *f*-value of N, that is the *f*-value of the most promising successor of N. Subs is ordered according to increasing *f*-values of the subtrees.

Thus at the moment that the node s has been expanded, the search tree looks like this: tr(s, 7/0, [lf(a,7/2), lf(e,9/2)]).

The key procedure in bestfirst search is `expand`:

expand(P, Tree, Bound, Tree1, Solved, Solution).

This expands the current (sub)tree as long as the *f*-value <= Bound.

P	Path between start node and Tree.
Tree	Current search (sub)tree.
Bound	*f*-limit for expansion of Tree.
Tree1	Tree expanded within Bound; consequently, the *f*-value of Tree1 is > Bound (unless a goal node has been found during the expansion).
Solved	Indicator whose value is `yes`, `no` or `never`.
Solution	A solution path from the start node through Tree1 to a goal node within Bound (if such a goal node exists).

```
bestfirst(Start, Solution) :-
  expand([], l(Start, 0/0), 9999, _, yes, Solution).
   % Assume 9999 is > any f-value.
% Case 1: goal leaf-node, construct a solution path:
expand(P, lf(N,_),_,_,yes,[N|P]) :- goal(N).

% Case 2: leaf-node, f-value less than Bound
% Generate successors and expand them within Bound
expand(P, lf(N, F/G), Bound, Tree1, Solved, Sol) :-
  F =< Bound,
  (findall(M/C, (edge(N,M,C), not(member(M,P))), Succ),
   !,                        % Node N has successors
   succlist(G, Succ, Ts),  % Make subtrees Ts
   bestf(Ts, F1),            % f-value of best successor
   expand(P, tr(N,F1/G, Ts), Bound, Tree1, Solved, Sol)
   ;
   Solved = never            % N has no successors - dead end
  ).

% Case 3: non-leaf, f-value < Bound
% Expand the most promising subtree; depending on results,
% procedure continue will decide how to proceed
expand(P, tr(N,F/G,[T|Ts]), Bound, Tree1, Solved, Sol) :-
  F =< Bound,
  bestf(Ts, BF), min(Bound, BF, Bound1),
  expand([N|P], T, Bound1, T1, Solved1, Sol),
  continue(P, tr(N,F/G,[T1|Ts]), Bound, Tree1, Solved1, Solved,
Sol).

% Case 4: non-leaf with empty subtrees
% This is a dead end that will never be solved
```

```
expand(_,t(_,_,[]),_,_,never,_) :- !.

% Case 5: value > Bound. Tree may not grow.
expand(_, Tree, Bound, Tree, no, _) :-
  f(Tree, F), F > Bound. % f extracts F value from Tree
% Helper predicates:

% continue(Path, Tree, Bound, NewTree, SubtreeSolved,
%     TreeSolved, Solution).
continue(_,_,_,_,yes,yes,Sol).
continue(P,tr(N,F/G,[T1|Ts]),Bound,Tree1,no,Solved,Sol) :-
  insert(T1,Ts,NTs),
  bestf(NTs,F1),
  expand(P,tr(N,F1/G,NTs),Bound,Tree1,Solved,Sol).
continue(P,tr(N,F/G,Ts,Bound,Tree1,Solved,Sol) :-
  bestf(Ts,F1),
  expand(P,tr(N,F1/G,Ts),Bound,Tree1,Solved,Sol).

% succlist(G0,[Node/Cost1,...],[lf(BestNode,BestF/G,...]).
%     Make list of search leaves ordered by their f-values.
succlist(_,[],[]).
succlist(Go,[N/C|NCs],Ts) :-
  G is G-+C,
  h(N,H),
  F is G+H,
  succlist(G0,NCs,Ts1),
  insert(lf(N,F/G),Ts1,Ts).

% insert T into list of trees Ts preserving order with
% respect to f-values
insert(T,Ts,[T|Ts]) :-
  f(T,F), bestf(Ts,F1),
  F =< F1, !.
insert(T,[T1|Ts],[T1,Ts1]) :-
  insert(T,Ts,Ts1).

% extract f-value
f(lf(_,F/_),F).
f(tr(_,F/_,_),F).

bestf([T|_],F) :- f(T,F).
bestf([],9999).
```

Admissibility of A* search

A search algorithm is *admissible* if it always produces an optimal solution. In our case, this would mean finding an optimal solution as the first solution. For each node n, let h*(n) denote the cost of an optimal path from n to a goal node.

Theorem: An A* algorithm that uses a heuristic function h that satisfies h(n) <= h*(n) for all n in the state space, is admissible.

Summary

We introduced the concepts of states and operators and gave a graph traversal algorithm that can be used as a problem solving tool.

We applied this to solve the "missionaries and cannibals" problem.

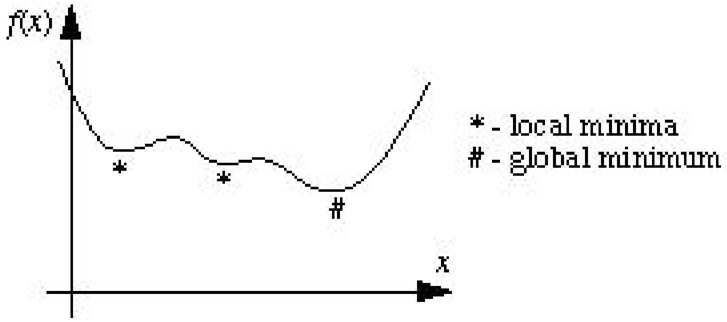
We described depth-first search, breadth-first search, and best-first search.

Minimum or Minima, and Other Irregular Plurals in English

There seems to be widespread confusion about the use of the words *minimum* and *minima* (along with *maximum / maxima*) in a mathematical context.

These and a number of other words, all borrowed from Latin or Greek, have potentially confusing singular and plural forms.

In fact, *minimum* is the singular form, just as in normal English usage, and *minima* refers to two or more. Thus we can talk about $x = a$ being *a local minimum* of a function $f(x)$, meaning that for $x = a$, $f(x)$ has a value that is lower than for nearby values of x.



A function will often have several local *minima*, however - that is, several dips in its graph. (A local *minimum* is actually the *bottom* of a dip.)

One of these dips/minima will be lower than all the rest. In the scenario in the diagram, this lowest local minimum is referred to as the *global minimum*. Note that *global* means "everywhere"; there can only be one global minimum for $f(x)$. In particular, it doesn't make sense to talk about ??"a global minima" (because "a" implies singular, and "minima" implies plural) or to talk about "global minima" unless you are talking about minima of two or more different functions.

"Local minima" of a function makes sense (assuming the function has more than one local minimum), but ??"a local minima" and ??"a minima" still are not right, as once again, "a" implies singular, and "minima" implies plural.

Similar rules apply to *maximum* and *maxima*, *extremum* and *extrema*, and *optimum* and the less often used *optima*.

Other unusual singular/plural forms in mathematical English

Singular	Singular example	Plural	Plural example
criterion	satisfy a criterion	criteria	several criteria
automaton	this automaton	automata	several deterministic automata
matrix	this matrix has eigenvalues ...	matrices	these symmetric matrices ...
vertex	find a vertex ...	vertices	a triangle has 3 vertices
datum	this single datum	data	this collection of data
schema	here is a schema for for-loops	schemata	some schemata for for-loops and if-statements

A longer list of mostly technical terms with Greek/Latin style plurals

Plural	Singular
abscissae	abscissa
addenda	addendum
algae	alga
alumnae	alumna
alumni	alumnus
alveoli	alveolus
analyses	analysis
annuli	annulus
antennae	antenna
appendices	appendix
aquaria	aquarium
atria	atrium
automata	automaton
axes	axis
bacilli	bacillus
bronchi	bronchus
cacti	cactus
caesurae	caesura
calculi	calculus
catharses	catharsis

cicatrices	cicatrix
colloquia	colloquium
colossi	colossus
compendia	compendium
consortia	consortium
continua	continuum
corpora	corpus
corrigenda	corrigendum
crania	cranium
crises	crisis
criteria	criterion
curricula	curriculum
data	datum
desiderata	desideratum
diaereses	diaeresis
diagnoses	diagnosis
dicta	dictum
directrices	directrix
dodecahedra	dodecahedron
effluvia	effluvium
ellipses	ellipsis
emphases	emphasis
encomia	encomium
ephemerides	ephemeris
equilibria	equilibrium
errata	erratum
esophagi	esophagus
extrema	extremum
foci	focus
fungi	fungus
genera	genus
genii	genius

honoraria	honorarium
icosahedra	icosahedron
incubi	incubus
infima	infimum
lacunae	lacuna
loci	locus
magi	magus
matrices	matrix
maxima	maximum
media	medium
memoranda	memorandum
millenia	millenium
minima	minimum
minutiae	minutia
moduli	modulus
mycobacteria	mycobacterium
nebulae	nebula
neuroses	neurosis
nuclei	nucleus
nucleoli	nucleolus
octahedra	octahedron
opera	opus
optima	optimum
ova	ovum
periphrases	periphrasis
phenomena	phenomenon
phyla	phylum
placentae	placenta
pleurae	pleura
podia	podium
polyhedra	polyhedron
psychoses	psychosis

pudenda	pudendum
pupae	pupa
quanta	quantum
radices	radix
radii	radius
referenda	referendum
rhombi	rhombus
rostra	rostrum
sacra	sacrum
sanataria	sanatarium
sanatoria	sanatorium
sarcophagi	sarcophagus
scapulae	scapula
schemata	schema
septa	septum
sera	serum
simplices	simplex
simulacra	simulacrum
solaria	solarium
spectra	spectrum
spermatozoa	spermatozoon
stadia	stadium
sterna	sternum
stigmata	stigma
stimuli	stimulus
strata	stratum
substrata	substratum
succubi	succubus
syllabi	syllabus
symposia	symposium
synopses	synopsis
syntheses	synthesis

termini	terminus
testes	testis
tetrahedra	tetrahedron
theses	thesis
thromboses	thrombosis
tori	torus
tumuli	tumulus
ultimata	ultimatum
umbilici	umbilicus
verrucae	verruca
vertebrae	vertebra
vertices	vertex

Bill Wilson's contact info

Australian Riddle

Q. What is the difference between Jean-Paul Sartre and a koala?

A. Answer is here.