Artificial Intelligence I

Matthew Huntbach, Dept of Computer Science, Queen Mary and Westfield College, London, UK E1 4NS. Email: mmh@dcs.qmw.ac.uk.

Notes on Logic

This is a set of notes covering the use of predicate logic in Artificial Intelligence. They can be considered an extension of the work you have already done in the Introduction to Logic course, and you will find the Barwise and Etchemendy book you used in that course useful for this part of the Artificial Intelligence course. In particular, the parts of Barwise and Etchemendy that were not covered in Introduction to Logic are covered here. Chapter 2 of the Peter Flach book recommended for the course covers the material in these notes, as does the Appendix of the Ian Pratt book.

This section of the Artificial Intelligence I notes does get quite technical. It is intended to give you some idea of the logical background to the language Prolog. Even so there is a lot of what Formal Methods people call "hand-waving" – moving briskly over parts where if one really wants to be precise over what is meant one has to introduce even more formality and more symbols.

If you get confused, don't worry. The next set of notes will introduce Prolog in a different way, describing it as "just another programming language". The importance of the logical background to Prolog is that when we write programs or perform calculations in it we can be a lot more clear as to exaxtly what we mean than if we were using a more traditional imperative programming language.

The Language of First Order Logic

The notation of first-order logic was introduced in the Introduction to Logic course. You will already be familiar with the symbols of first-order logic, and their meanings in terms of truth tables. You have also seen the idea of *natural deduction*: the use of rules to draw arguments from premises.

We will start by briefly summarising what you should already know.

Firstly, we have atomic sentences which consist of a predicate name followed by a number of arguments, representing a simple fact. For example `Likes(John,Mary)` might be interpreted as representing the fact "John likes Mary". It is assumed that such facts are either true or false.

We then have various connectives which operate on sentences, so if A and B are sentences:

$\neg$A is true when A is false, and is false when A is true.

A$\vee$B is false when both A and B are false, and is true otherwise.

A$\wedge$B is true when both A and B are true, and is false otherwise.

A$\rightarrow$B is false when A is true and B is false, and is true otherwise.

A$\leftrightarrow$B is true when both A and B are true, true when both A and B are false, and false otherwise.

We have quantifier symbols, which enable atomic sentences to have variables as well as names as arguments, with each variable governed by a quantifier. So, if x is a variable and A a sentence containing x:

$\exists$xA means there is at least one value which can be substituted for x in A to make A a true sentence.

$\forall$xA means that A is true for all possible values which could be substituted for x.

One way of thinking of $\forall$xA is as a shorthand for $A_{n_1} \wedge A_{n_2} \wedge A_{n_3} \wedge \ldots$, where A_m is A with m substituted for the variable x (this is more commonly written A{x/m}), and $n_1, n_2, n_3, \ldots$ is the set of all possible values a variable could have, that is all things our logic statements deal with. Similarly, $\exists$xA can be considered a shorthand for $A_{n_1} \vee A_{n_2} \vee A_{n_3} \vee \ldots$,.The name for this set of all things is the *Herbrand universe*. Note that this set is finite, consisting of all the names that occur in all our sentences if we do not have any function symbols, but it becomes infinite once we include function symbols since even

if we have just one name, say `John`, and one function, say `father`, the Herbrand universe is: {`John`,`father(John)`,`father(father(John))`, …}.

The *Herbrand base* is the set of all possible atomic sentences, that is each predicate name combined with all possible combinations of arguments from the Herbrand universe. Once again, if we introduce functions we jump from a finite set to an infinite set. So if we have no functions, the single predicate `Likes` (with arity 2), and the two names `John` and `Mary`, the Herbrand base is:

`{Likes(John,John), Likes(John,Mary), Likes(Mary,John), Likes(Mary,Mary)}`

if we introduced the function `father`, it would become a lot more complicated.

A *Herbrand interpretation* is simply a subset of the Herbrand base. We can use a Herbrand interpretation to tell us whether a sentence is to be reckoned as true or false. For each of the atomic sentences that make it up, we assign the value true if it is a member of the Herbrand interpretation, false otherwise. We then use the truth tables for the connectives to find the value of the whole compound sentence, bearing in mind the expansion of the quantifiers ∃ and ∀ referred to above.

Clearly, this is not an efficient way of determining the truth or falsehood of a sentence. Not only that, but because the Herbrand base is infinite once we have functions we could not possibly prove the truthfulness of all sentences because that would involve a search of an infinite space.

The importance of introducing the above concepts is that they allow us to give a meaning to our statements and deductions in logic, which are, after all, just symbols on a piece of paper or stored in a machine. A Herbrand interpretation may be considered a link between the symbols in a piece of logic and the "real world". For instance, if we have a simple piece of logic with just the names `John` and `Mary` and the 2-ary predicate `Likes`, then the Herbrand interpretation `{Likes(Mary,Mary),Likes(John,Mary)}` represents the situation where Mary likes herself but does not like John, while John does not like himself but likes Mary. We could (but won't) take things further to give a semantics in which the symbols `John` and `Mary` are linked to real life people John and Mary, and the symbol `Likes` is linked to a real-life liking relationship, but since these are just symbols we should be aware that the interpretation we have in mind is nothing to do with real-world people, we just decided, perversely, to use those names. It may be that we meant `John` to represent the integer 2, `Mary` to represent the integer 3, and `Likes` to represent the arithmetic relation ≤, in which case the Herbrand interpretation which correctly describes what we had in mind is:

`{Likes(John,John),Likes(John,Mary),Likes(Mary,Mary)}`

Higher Order Logic

As we have used the term "first-order logic" you may wonder whether there are such things as second, third and so on order logic. There are, but they are rarely used due to the difficulty of reasoning with them. In Functional Programming you may remember the idea of a "higher order function", that is a function which itself took a function as an argument. For example, the function `map` takes a list and a function and returns the list obtained by applying the input function to every item in the list. The definition of `map` would have a variable which could be replaced in execution with different functions.

Higher order logic is a logic in which variables may range over predicates, so that a variable introduced by a quantifier may be used as a predicate name or predicate names may themselves be given as arguments to predicates. For example, a predicate `p` with two arguments is defined as symmetric if for any `x` and `y` whenever `p(x,y)` is true, `p(y,x)` is also true. We could write this formally as:

`∀p Symmetric(p) ↔ (∀x∀y p(x,y)↔p(y,x))`

Here `p` is being used as a variable. In general, an `n`th order logic has variables for `n-1`th order predicates. In the example, `Symmetric` is a second-order predicate since it takes an argument which is the name of a first order predicate. We will not consider higher order logics any further in this course.

Models and Entailment

A *model* of a set of sentences in logic is a Herbrand interpretation which makes all those sentences true. Note that a set of sentences may have more than one model. For example, suppose we want to represent the statements "John and Mary are humans. All humans like themselves". The logic representation of this is:

`Human(John).`

`Human(Mary).`

`∀x Human(x)→Likes(x,x).`

The last sentence can be considered a shorthand for:

`(Human(John)→Likes(John,John))∧(Human(Mary)→Likes(Mary,Mary))`

Our sentences say nothing about whether John likes Mary or Mary likes John. So the following is a model for the sentences (note that having introduced the predicate `Human`, `Human(John)` and `Human(Mary)` are in the Herbrand base):

`{Likes(John,John),Likes(Mary,Mary),Human(John),Human(Mary)}`

but there are three other models:

`{Likes(John,John),Likes(Mary,Mary),Human(John),Human(Mary),Likes(John,Mary)}`

`{Likes(John,John),Likes(Mary,Mary),Human(John),Human(Mary),Likes(Mary,John)}`

`{Likes(John,John),Likes(Mary,Mary),Human(John),Human(Mary),Likes(John,Mary),`
`Likes(Mary,John)}`

Any interpretation which does not contain all the atomic sentences in the first model is not a model, since it would lead to some of the sentences in our original set of sentences being false, so for example:

`{Likes(John,John),Likes(Mary,Mary),Human(Mary),Likes(Mary,John)}`

is not a model because it does not contain `Human(John)`, so `Human(John)` in our original sentences would be false.

In general we would prefer to take the first model as the correct interpretation of the sentences, since we it seems sensible to assume that if we have not been told some fact or given the information to infer it, that fact must be false. It would be difficult if not impossible in most information systems to have to store not only true knowledge but also everything we knew to be false.

We therefore restrict our attentions to *minimal models* where a minimal model is defined as a model such that no subset of it is also a model. In doing this, however, we should recall the comments made in the first set of notes about the problems with micro-worlds: it may be reasonable to assume in a micro-world that everything we don't know is false, but if our logic is trying to connect to the real world we may have to be ready to account for the fact that there are things we don't know, and that therefore a conclusion of false may really mean "false in the absence of further information".

Having defined the concept of minimal model, one thing we should note is that not all sets of sentences have a unique minimal model. Consider the following:

`Human(John).`

`Human(Mary).`

`∀x Human(x)→Man(x)∨Woman(x).`

These sentences have nine models:

① `{Human(John),Human(Mary),Man(John),Woman(Mary)}`

② `{Human(John),Human(Mary),Man(John),Man(Mary)}`

③ `{Human(John),Human(Mary),Man(John),Man(Mary),Woman(Mary)}`

④ `{Human(John),Human(Mary),Woman(John),Woman(Mary)}`

⑤ `{Human(John),Human(Mary),Woman(John),Man(Mary)}`

⑥ `{Human(John),Human(Mary),Woman(John),Man(Mary),Woman(Mary)}`

⑦ {Human(John),Human(Mary),Woman(John),Man(John),Woman(Mary)}

⑧ {Human(John),Human(Mary),Woman(John),Man(John),Man(Mary)}

⑨ {Human(John),Human(Mary),Woman(John),Man(John),Man(Mary),Woman(Mary)}

Four of these are minimal models, the first, second, fourth and fifth. The problem is that the logic tells us that a human is a man or a woman, but there is nothing to enable us to determine which any human is, nor indeed is it stated that someone cannot be both a man and a woman.

Sets of sentences with a unique minimal model are termed *definite*, since it is possible to draw a definite set of conclusions from them: those in the unique model. Sets of sentences without a unique minimal model are termed *indefinite*.

The point of this discussion is to suggest caution over machine reasoning. Human reasoning may bring in all sorts of knowledge from the outside world which we are hardly aware we are doing. It is only when we obtain absurd conclusions from machine reasoning that we realise that the machine is simply manipulating symbols and it cannot possibly make any inferences which depend on links we intend those symbols to have with the outside world. Since we did not state that a person cannot simultaneously be a man and a woman, the machine has no reason to discard those models. We might suppose that the first model above is the natural one because we know that John is a man's name and Mary a woman's name, but there is no reason for the machine to make this link.

In order to escape from these problems, we need to ensure that any system of reasoning does not depend on links with the real world. If we ensure that a system of reasoning is correct for all models, then we can be sure it is correct for the one we intend, as well as for any other. Stated formally, we say that if P is a set of sentences, our premises, and C is a conclusion then P *entails* C if every model for P is also a model for C. This is written $P \models C$.

Deduction, Soundness and Completeness

You will have seen in the Introduction to Logic course that there are various rules that may be used to draw or deduce conclusions from premises. The most well known one is *Modus ponens* or elimination rule for implication. This states that if our premises include F and $F \rightarrow G$, we can conclude G. This is often presented as:

$$\frac{F \quad F \rightarrow G}{G} \quad (\rightarrow \text{elim})$$

Another rule eliminates the universal quantifier: since $\forall x F$ means F is true for any value of x, we can substitute any value t for x in F. F with t substituted for x is written $F\{x/t\}$ so the rule is presented as:

$$\frac{\forall x F}{F\{x/t\}} \quad (\forall \text{ elim})$$

If we took the suggestion that $\forall x A$ is a shorthand for $A_{n_1} \wedge A_{n_2} \wedge A_{n_3} \wedge \ldots$, then the above is a form of and-elimination, since we can certainly conclude any A_{n_i} from $A_{n_1} \wedge A_{n_2} \wedge A_{n_3} \wedge \ldots$.

The example, going back to the Ancient Greeks, traditionally used to demonstrate modus ponens is actually a combination of these two rules "All men are mortal, Socrates is a man, therefore Socrates is mortal" can be broken down into:

Premises: `Man(Socrates)` $\quad \forall x$ `Man(x)`$\rightarrow$`Mortal(x)`

$\forall$ elimination in the second premise gives us `Man(Socrates)`$\rightarrow$`Mortal(Socrates)`, then by $\rightarrow$ elimination we get `Mortal(Socrates)`.

Another reasoning rule, *modus tollens*, uses a sort of backwards reasoning. It states that if we have $F \rightarrow G$, and we know G is false, then we must conclude that F too is false:

$$\frac{\neg G \quad F \rightarrow G}{\neg F}$$

If we can show some conclusion C starting from premises P using a set of reasoning rules R, we say that C is *derivable* from P using R. This is written $P \vdash_R C$. Unless we are comparing different sets of

reasoning rules or have some special reason to want to emphasise the fact that we are using a particular set of rules, we drop the subscript and say just that $P \vdash C$.

Now we have a way, using the idea of models, of saying formally whether something is a logical conclusion from a set of premises, and we have the idea of a set of rules that enable us to determine conclusions from premises. Ideally, we would want these two things, entailment and derivability, to be the same, that is $\vdash \equiv \models$ (this combination of symbols is used as an icon by the Association for Logic Programming, a group of mainly academics which is dedicated to research into and the promotion of logic programming languages). The extent to which $\vdash$ can be identical to $\models$ is beyond the scope of this course, but it can be shown that there are languages, including first-order arithmetic, where it is <u>impossible</u> to find a set of rules R such that $\vdash_R \equiv \models$. This is the Gödel Incompleteness theorem, named after the logician Gödel who first showed it.

In general, there may be reasons why we use a set of rules such that $\vdash$ is not equivalent to $\models$ even for logic systems where such a compromise is not necessary. Often this will be for reasons of efficiency. As we showed with our discussion on common-sense reasoning, there may be occasions where we are willing to accept conclusions which we cannot say are strictly logical conclusions in the sense of logical entailment, for example conclusions which we think are probably correct but rely on some unproved assumptions.

To be more precise about it, we can define two conditions which a mechanism $\vdash$ for drawing conclusions might have: *soundness* and *completeness*.

An inference procedure $\vdash$ is called *sound* if whenever $P \vdash Q$, it is also the case that $P \models Q$. That is, any conclusions it draws are entailed by the premises, but we cannot say that every conclusion that is entailed by the premises can be drawn by $\vdash$.

An inference procedure $\vdash$ is called *complete* if whenever $P \models Q$, it is also the case that $P \vdash Q$. That is, every conclusions that is entailed by the premises can be drawn by $\vdash$, but it may also draw some extra ones which are not entailed by the premises (and we cannot tell which they are).

Obviously, if $\vdash \equiv \models$ then $\vdash$ is both sound and complete.

Conjunctive Normal Form

You will know from Introduction to Logic that it is possible to transform sentences in logic without changing their meaning. For example, $(C \vee \neg(A \wedge B)) \rightarrow D$ is identical in meaning to $(A \wedge B \wedge \neg C) \vee D$ – a truth table for both would be the same. One way of simplifying deduction is to convert sentences to a standard or *canonical* form. This is similar to the way the first step in solving a polynomial equation in algebra is to convert it to its canonical form of terms and coefficients, so, for example, if we have $3x(x^2+4)-7x^3=(x+2)(x^2-3)$ we convert it to $-5x^3-2x^2+15x+6=0$.

The canonical form we shall consider is termed *Conjunctive Normal Form* or CNF. In this form a sentence always takes the shape of a conjunction of terms, each of which is a disjunction of literals. A literal is either an atom, or a negated atom, A or $\neg A$. So a disjunction of literals is a set of these linked together with $\vee$s, i.e. $A_1 \vee A_2 \vee \ldots \vee A_n$. A conjunction of these sets links them together with $\wedge$s, so it has the general form: $(A_{11} \vee A_{12} \vee \ldots \vee A_{1n_1}) \wedge (A_{21} \vee A_{22} \vee \ldots \vee A_{2n_2}) \wedge \ldots (A_{m1} \vee A_{m2} \vee \ldots \vee A_{mn_m})$.

Converting sentences in logic to CNF involves making use of a number of equivalence rules. Converting sentences using these equivalence rules is different from inferring facts using deduction rules because equivalence rules can be applied both ways. We know, for example, that de Morgan's rule states that $\neg(A \wedge B)$ is equivalent to $(\neg A \vee \neg B)$, where A and B are <u>any sentence</u> (so they may themselves be compound sentences made up of atomic sentences joined by connectives). Conversion can go either way to give a sentence which is identical in meaning, so will evaluate the same under any interpretation. The rules of natural deduction, however, are one way. So by *modus ponens* we can say that with A and $A \rightarrow B$ we can conclude B, but we cannot go the other way and say that with B and $A \rightarrow B$ we can conclude A. Any interpretation which makes A and $A \rightarrow B$ true will make B true, but it is not the case that any interpretation which makes B will also make A and $A \rightarrow B$ true.

The first step in the conversion of a sentence in predicate logic to CNF is to note that the symbols $\rightarrow$ and $\leftrightarrow$ are unnecessary. It is always possible to convert any compound of the form $A \rightarrow B$ to $\neg A \vee B$, and

any compound of the form $A \leftrightarrow B$ to the form $(A \wedge B) \vee (\neg A \wedge \neg B)$, remembering that A and B here can be any sentence.

The second step involves making negation apply only to individual atoms, and not to compound sentences, or *maximally distributing* negation. This is done using de Morgan's rules, so anything of the form $\neg(A \wedge B)$ becomes $(\neg A \vee \neg B)$ and anything of the form $\neg(A \vee B)$ becomes $(\neg A \wedge \neg B)$ until the negations are pushed down inside all the brackets. Also, $\neg\neg A$ always becomes A for any A. A sentence which has been converted to this form is described as being in *negation normal form* . To deal with sentences involving quantifiers and variables we make use of the equivalent $\neg \forall x A \equiv \exists x(\neg A)$ and $\neg \exists x A \equiv \forall x(\neg A)$. The maximal distribution means these rules are applied until there is no further scope for applying them.

The third step involves maximally distributing occurrences of $\vee$. This means converting any occurrences of $A \vee (B \wedge C)$ to the equivalent $(A \vee B) \wedge (A \vee C)$. The additional rules for the quantifiers push them out so they operate over whole disjunctions. This involves noting that $A \vee (\exists x B)$ is equivalent to $\exists x(A \vee B)$ and $A \vee (\forall x B)$ is equivalent to $\forall x(A \vee B)$ providing that the variable name x does not also occur in A bound by a quantifier outside the whole expression (a similar situation to the scope rules you saw in functional programming). A simple way to overcome this problem is to ensure that with every $\forall$ and $\exists$ you use a variable name which is not re-used with any other $\forall$ or $\exists$. Again, the rules are applied repeatedly until they cannot be applied further.

The fourth step maximally distributes $\forall$. This uses the equivalence $\forall x(A \wedge B) \equiv (\forall x A) \wedge (\forall x B)$. This leads to universal quantifiers applying only to individual disjunctions.

The final step, which removes the existential quantifier $\exists$, requires a little more discussion. It involves a process called *Skolemisation* after the mathematician Thoralf Skolem. Skolemisation involves an alternative way of looking at $\exists$ than considering $\exists x A$ to be a shorthand for $A_{n_1} \vee A_{n_2} \vee A_{n_3} \vee \ldots$, as we did previously. The idea is that we invent a name for something which we have stated exists and fills the given properties. So, for example, if we have the sentence "There exists someone who is a professor and teaches logic", the predicate logic sentence is $\exists x(\texttt{Professor(x)} \wedge \texttt{TeachesLogic(x)})$. If we Skolemise this, we name the individual, let us say with the name `LogProf`. Then the sentence becomes $\texttt{Professor(LogProf)} \wedge \texttt{TeachesLogic(LogProf)}$. The new name introduced is referred to as a *Skolem constant*, and must not occur elsewhere in the sentence we are Skolemising. The process is a little more complex when the Skolemisation takes place within a universally quantified expression. Suppose our sentence is "Every college has a professor who teaches logic there", or in predicate logic

$$\forall \texttt{y(College(y)} \rightarrow \exists \texttt{x(Professor(x)} \wedge \texttt{TeachesLogicAt(x,y)))}.$$

Skolemisation as above would give

$$\forall \texttt{y(College(y)} \rightarrow \texttt{(Professor(LogProf)} \wedge \texttt{TeachesLogicAt(LogProf,y)))}$$

which suggests that there is a single individual who teaches logic at every college. The correct way of dealing with this is to introduce a *Skolem function* which is a function which takes as arguments every univesally quantified variable in scope. In our example, this would involve replacing `x` by `logProf(y)`, where `logProf(y)` has the meaning "the logic professor at college `y`". This makes the sentence:

$$\forall \texttt{y(College(y)} \rightarrow \texttt{Professor(logProf(y))} \wedge \texttt{TeachesLogicAt(logProf(y),y))}.$$

Skolem constants are frequently used to represent what in natural language would use what grammarians call the "genitive" case, or the "possessive", in English using "' s"or "of". Note that the ambiguity over the scope of $\exists$ and $\forall$ can exist in natural language. For example, the English sentence "There is a professor who teaches logic at every college" could be taken to mean the above statement in logic, or that there is indeed a single individual who teaches at every college (in English conversation the ambiguity is often resolved by differing the stress on the words). This indicates why logic notation is preferred, as providing we make full use of brackets or have an agreed "binding strength" to our logic operators, there is no such ambiguity. In the case where there is a single peripatetic logic professor, the logic sentence is:

$$\exists \texttt{x(Professor(x)} \wedge \forall \texttt{y(College(y)} \rightarrow \texttt{TeachesLogicAt(x,y)))}$$

In general, as indicated in the above sentences, the quantifier $\exists$ tends to be used alongside the connective $\wedge$, while $\forall$ tends to be used with the connective $\rightarrow$ when translating natural language sentences to logic.

At this point, let us consider an example of conversion to CNF. Our initial sentence is:

$\forall x([a(x) \wedge b(x)] \rightarrow [c(x,m) \wedge \exists y([\exists z(c(y,z))] \rightarrow d(x,y))]) \vee \forall x(e(x))$

For readability two types of brackets are used, `()` and `[]`. The `m` is a constant, `x`, `y` and `z` are variables.

The step of removing the $\rightarrow$ is done in two stages, first the outer one:

$\forall x(\neg[a(x) \wedge b(x)] \vee [c(x,m) \wedge \exists y([\exists z(c(y,z))] \rightarrow d(x,y))]) \vee \forall x(e(x))$

then the inner one:

$\forall x(\neg[a(x) \wedge b(x)] \vee [c(x,m) \wedge \exists y(\neg[\exists z(c(y,z))] \vee d(x,y))]) \vee \forall x(e(x))$

The second step removes the negation which applies to a compound sentence, and the negation which applies to a quantifier:

$\forall x([\neg a(x) \vee \neg b(x)] \vee [c(x,m) \wedge \exists y(\forall z[\neg c(y,z)] \vee d(x,y))]) \vee \forall x(e(x))$

We can apply Skolemisation at this point, replacing `y` with `f(x)` in order to remove the $\exists$ quantifier:

$\forall x([\neg a(x) \vee \neg b(x)] \vee [c(x,m) \wedge (\forall z[\neg c(f(x),z)] \vee d(x,f(x)))]) \vee \forall x(e(x))$

In order to allow for the third step, we replace one of the occurrences of `x` with the new variable `u`, and rearrange the $\forall$ quantifiers, giving:

$\forall x \forall u \forall z(\neg a(x) \vee \neg b(x) \vee [c(x,m) \wedge (\neg c(f(x),z) \vee d(x,f(x)))] \vee e(u))$

Now distributing the occurrences of $\vee$ gives

$\forall x \forall u \forall z\ ([\neg a(x) \vee \neg b(x) \vee c(x,m) \vee e(u)] \wedge$
$[\neg a(x) \vee \neg b(x) \vee (\neg c(f(x),z) \vee d(x,f(x))) \vee e(u)])$

Distributing $\forall$, and also noting the associativity of $\vee$, gives

$\forall x \forall u \forall z\ [\neg a(x) \vee \neg b(x) \vee c(x,m) \vee e(u)] \wedge$
$\forall x \forall u \forall z\ [\neg a(x) \vee \neg b(x) \vee \neg c(f(x),z) \vee d(x,f(x)) \vee e(u)]$

We can drop the quantifiers and assume all variables are universally quantified across disjunctions, giving:

$(\neg a(x) \vee \neg b(x) \vee c(x,m) \vee e(u)) \wedge (\neg a(x) \vee \neg b(x) \vee \neg c(f(x),z) \vee d(x,f(x)) \vee e(u))$

Clausal Form

The clausal form of a sentence is simply another way of writing the conjunctive normal form. Recall that the process of converting to CNF reduces a sentence in logic to the form:

$(A_{11} \vee A_{12} \vee \ldots \vee A_{1n_1}) \wedge (A_{21} \vee A_{22} \vee \ldots \vee A_{2n_2}) \wedge \ldots (A_{m1} \vee A_{m2} \vee \ldots \vee A_{mn_m})$.

where each A_{ij} is a literal, either a positive literal if it is an atomic sentence, that is a predicate name followed by its argument, or a negative literal if it is a negated atomic sentence A.

We can convert this into a set of disjunctions by deleting the $\wedge$s, thus:

$$\{(A_{11} \vee A_{12} \vee \ldots \vee A_{1n_1}), (A_{21} \vee A_{22} \vee \ldots \vee A_{2n_2}), \ldots (A_{m1} \vee A_{m2} \vee \ldots \vee A_{mn_m})\}$$

The process of Skolemisation will have removed all the existential quantifiers, and the process of maximally distributing universal quantifiers means that no universal quantifier reaches across several disjunctions. The universal quanitifiers can be ommitted since all remaining variables are universally quantified.

Each disjunction consists of positive and negative literals, so it can be written:

$$(A_1 \vee A_2 \vee \ldots \vee A_n) \vee (\neg B_1 \vee \neg B_2 \vee \ldots \vee \neg B_m)$$

But another way of writing this makes use of the fact that $A \vee \neg B \equiv \neg B \rightarrow A$. So the above can be written:

$$\neg(\neg B_1 \vee \neg B_2 \vee \ldots \vee \neg B_m) \rightarrow (A_1 \vee A_2 \vee \ldots \vee A_n)$$

which, by de Morgan's rule, is:

$$B_1 \wedge B_2 \wedge \ldots \wedge B_m \rightarrow A_1 \vee A_2 \vee \ldots \vee A_n$$

If a sentence can be converted to a set of clauses of this form where the number of positive literals, n, in each conjunction is always 1, it is called *Horn* and the individual clauses are called *Horn clauses* after the logician Alfred Horn. It is usual in Horn clauses to write the implication the other way round, so a Horn clause takes the form:

$$A \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_m$$

with $m \geq 0$. The Horn property means that the set of clauses are definite, as described earlier, meaning they have a unique minimal model.

A Horn clause may be interpreted as expressing the rule "A is true if B_1 and B_2 and … and B_m are true". Note that the "if" here is not an "if and only if", so the rule might be better interpreted as "one reason for A being true could be B_1 and B_2 and … and B_m being true". If we are keeping to the closed world assumption (see the previous set of notes) that there is no unknown information, then we can assume that the set of clauses with A on the left-hand side of the $\leftarrow$ provide the complete range of possibilities for explaining why A could be true. Note that if $m=0$, the $\leftarrow$ is omitted and the clause is taken as a straight assertion that A is true. Such a clause without conditions is called as a *fact*.

The clausal form of the example we gave above is:

$c(x,m) \vee e(u) \leftarrow a(x) \wedge b(x)$

$d(x,f(x)) \vee e(u) \leftarrow a(x) \wedge b(x) \wedge c(f(x),z)$

which does not have the Horn property, as both clauses have more than one literal to the left-hand side of the $\leftarrow$.

Resolution

If we try to draw some desired conclusion merely by randomly applying rules of deduction until we reach it, we are soon going to find our process of deduction hopelessly inefficient. The process of *resolution* seeks to rationalise natural deduction by working on clausal forms and using just one basic rule.

You will know from Introduction to Logic that if it is desired to prove some conclusion, it is possible either to start from the premises and try and derive the conclusion (known as *forward chaining*), or to start from the conclusion and try and work backwards to the premises (known as *backward chaining*).

One method of proof often used by mathematicians is *reductio ad absurdum*. This starts by assuming the thing that one is trying to prove is false. It is then shown that adding this false statement to the premises leads to an absurdity: it is possible to deduce a statement that cannot be true under any circumstances. Formalising this, it means that if we have premises P, we can say that some sentence Q is true if $P \cup \{\neg Q\}$ is *unsatisfiable*, that is it has no model. We use the symbol $\square$ to mean the absurd sentence that can never be true in any circumstances. In standard predicate logic for any A, $A \wedge \neg A$ is equivalent to $\square$.

Resolution involves noting the following rule of inference: if we have a conjunction of two disjunctions, one of which contains some positive literal A and the other contains the negative literal A, then we can eliminate A altogether to obtain a single disjunction of the remaining literals. Formalising it, we can say that:

$$\frac{(B_1 \vee B_2 \vee \ldots \vee B_n \vee A) \wedge (\neg A \vee C_1 \vee C_2 \vee \ldots \vee C_m)}{B_1 \vee B_2 \vee \ldots \vee B_n \vee C_1 \vee C_2 \vee \ldots \vee C_m} \text{(resolution)}$$

Informally, this can be explained as saying that if we have a conjunction of two disjunctions then at least one literal from each disjunction must be true. If one disjunction contains A and the other $\neg A$, then it cannot be the case that both A and $\neg A$ are true. So if A is true, one of the literals apart from $\neg A$ in the other disjunction must be true, or if $\neg A$ is false, one of the literals apart from A in the other must be true. So the whole can be reduced to a disjunction of the remaining literals. Since we are using CNF, it is possible to write our disjunctions as sets of literals, and in this alternative form the resolution rule looks like:

$$\frac{\{B_1,B_2,\ldots B_n,A\} \quad \{\neg A,C_1,C_2,\ldots C_m\}}{\{B_1,B_2,\ldots B_n,C_1,C_2,\ldots C_m\}} \qquad \text{(resolution)}$$

It can then be seen that various other rules are just special cases of resolution. Given the equivalence $P \rightarrow Q \equiv \neg P \vee Q$, or in CNF form $\{\neg P,Q\}$, modus ponens is just the specialisation of resolution above with $A=P$, $n=0$, $m=1$:

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

and, similarly, modus tollens is the case where $A=Q$, $n=1$ and $m=0$:

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

Reaching the absurd statement occurs when both m and n are 0:

$$\frac{\{A\} \quad \{\neg A\}}{\{\}}$$

the empty set being the equivalent of the absurd statement $\Box$.

In general if we are applying a series of resolutions to prove some fact, at any one point we need only pick one literal out to be resolved against, there is no need to try each possible resolution on each possible literal. The rule used to decide which literal is picked out is called the *selection rule*, and the whole process is known as *SLD resolution* (S for the selection rule, L for linear resolution referring to the construction of a proof which goes one step at a time, and D for definite clauses, i.e. with only one positive literal). Note that clauses may be re-used in resolution as often as needed, that is they are not consumed during resolution, as we know $A \wedge A \wedge B$ is equivalent to $A \wedge B$, so if C is obtained by combining A with B in resolution, we can still write the results as $A \wedge C$.

As an example of resolution, let us consider a simple set of clauses, or on the right hand side the equivalent in CNF. The clauses are numbered for reference in the text:

①	$A \leftarrow B \wedge C \wedge D$	$\{A,\neg B,\neg C,\neg D\}$
②	$A \leftarrow E \wedge F$	$\{A,\neg E,\neg F\}$
③	$B \leftarrow F$	$\{B,\neg F\}$
④	E	$\{E\}$
⑤	F	$\{F\}$
⑥	$A \leftarrow F$	$\{A,\neg F\}$
⑦	$B \leftarrow G$	$\{B,\neg G\}$

To prove that A holds we need to add an assumption that A does not hold, and then use resolution to reach the absurd or empty clause. So we start with $\{\neg A\}$ which can be considered added to the set of clauses or conjunctions ① to ⑦, and then apply resolution transformations (as we have seen each step results in a logically equivalent set of clauses). It can be seen that this is a form of backwards chaining, as it starts with what we want to prove. The process (numbering the steps in the proof with reversed numbers) of proof proceeds:

❶ By resolution with clause ② we obtain $\{\neg E,\neg F\}$

❷ Then by resolution with clause ④ we obtain $\{\neg F\}$

❸ Then by resolution with clause ⑤ we obtain $\{\}$, the empty set so we conclude A must be a consequence of our original set of clauses.

An objection might be raised at this point. At the start we decided to resolve on clause ②, but we could have resolved on clause ①. If we had an automatic system doing resolution we could not rely on it using foresight to predict exactly which clause is the best to resolve on. Our automatic system would probably resolve on the first clause it finds which is capable of resolution. The resolution would then start with {¬A} and go:

❹ By resolution on clause ① we get {¬B,¬C,¬D}

❺ By resolution on clause ③ we get {¬F,¬C,¬D}

❻ By resolution on clause ⑤ we get {¬C,¬D}

But now we are stuck, since we can't resolve on any of our clauses since there are none with a positive literal C or D. If we go back to point ❹ we could decide to resolve on clause ⑦ instead of clause ③, giving:

❼ By resolution on clause ⑦ we get {¬G,¬C,¬D}

Now again we are stuck, since there are no clauses with positive literal G, C or D. The only way out is to go back to the initial position, which we shall call point ❿, where we had a choice of clauses with positive literal A, and take the route originally suggested through points ❶, ❷ and ❸.

What we are doing is *search with backtracking*. We keep on resolving using the simple rule that we try the first clause that matches with the leftmost negative literal in our set of negative literals. If we reach a point where we can't go any further, we go back to the last position we were in where we had a choice of clauses to resolve on and try the next matching clause against the leftmost literal of that position. Only if there are no untried choices left anywhere do we conclude that the goal we were trying to prove must be false.

Note that the fact that we always try to resolve on the first negative literal in our set can also slow us down. Suppose that at point ❹ instead of trying to resolve on the ¬B we tried to resolve on the ¬C or ¬D. We would then have immediately discovered that any attempts to proceed from point ❹ are doomed to failure because of the lack of a match for ¬C or ¬D. This is another case where we cannot expect an automatic system to have foresight, so it will keep to the simple rule of trying to resolve from the set of negative literals in left-to-right order.

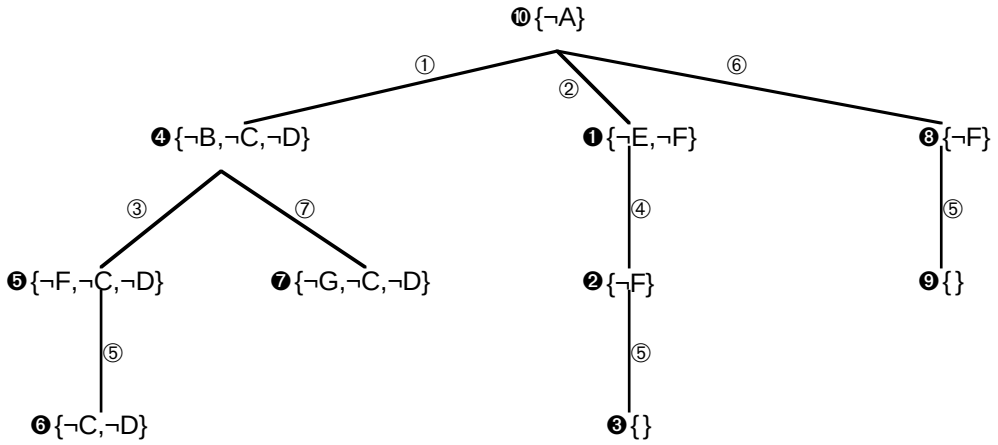
As a final note, there are in fact two ways to prove A. At point ❿ we could have gone:

❽ By resolution with clause ⑥ we get {¬F}

❾ By resolution with clause ⑤ we get {}.

In fact, a good way of showing the process of resolution is through a tree diagram, known as the SLD tree. The root node of the tree stores the initial query. The branches are the trees whose roots are produced by each possible resolution using the various clauses (with the selection rule assumed). Below we show an SLD tree for the problem discussed above. The tree diagram assumes that resolution is always in a left-to-right order, it would be possible to show a more complex form of the tree diagram which shows the possibilities of trying different orders, but we shall not go further into that at this point.

The black circled numbers in the diagram represent the states similarly numbered in the explanation above, the white circled numbers represent the clause used for resolution to obtain the child state. It can be seen that the method of resolution we have described: left-to-right on the literals, using the clauses in the order they are given and adding the new literals obtained to the front of the remaining ones, amounts to a depth-first left-to-right search of this tree for nodes storing the state {}.



SLD resolution using the selection rule of always picking the leftmost goal is preferred because it is efficient and can be shown to be both sound and complete, except for one problem. This can be illustrated using the example we have been covering by supposing there is an extra clause, ⑧ B ← A or in CNF {B,¬A}. Then, at point ❹ we would have an extra branch of the tree labelled with leading to the state {¬A,¬C,¬D}. But ¬A is a re-occurrence of the literal at the root of the tree. So the result would be that the whole tree would be repeated, and inside it repeated again, forever. Since the tree is infinite, the search would stay inside it and never get out to its right to find the way to show that A holds. This is similar to getting caught in an infinite loop in an imperative program, or an infinite recursion in a functional program. In general, any solution to the right of an infinite branch in the search tree will never be found.

Unification

In our explanation of resolution we have ignored the possibility of atoms containing variables. Resolution depends on two literals being identical, except for one being positive and the other being negative. To show how resolution works with variables, let us consider our previous example expanded so that the predicates now take arguments. We will assume that arguments a, b, c, or d are constants while those beginning with x or y are variables. The change in type font is meant to indicate that the names are now predicate names, not symbols standing for literals. Here is our new program, with the equivalent CNF form to the right:

①	A(a,x) ← B(y)∧C(y,x)∧D(x)	{A(a,x),¬B(y),¬C(y,x),¬D(x)}
②	A(b,x) ← E(x)∧F(x)	{A(b,x),¬E(x),¬F(x)}
③	B(x) ← F(x)	{B(x),¬F(x)}
④	E(c)	{E(c)}
⑤	F(x)	{F(x)}
⑥	A(d,x) ← F(x)	{A(d,x),¬F(x)}
⑦	B(x) ← G(x)	{B(x),¬G(x)}

It should be recalled that the process of maximally distributing ∀ means that the scope of any variable in CNF applies only to the disjunction, so the logic equivalent of the above is:

∀x∀y(A(a,x)∨¬B(y)∨¬C(y,x)∨¬D(x))∧
∀x(A(b,x)∨¬E(x)∨¬F(x))∧
∀x(B(x)∨¬F(x))∧
(E(c))∧
∀x(F(x))∧
∀x(A(d,x)∨¬F(x))∧
∀x(B(x)∨¬G(x))

For ease of explanation, it may be easier to suppose that all variable names are unique. We can then safely moves the quantifiers out, thus:

```
∀x1∀y1∀x2∀x3∀x5∀x6∀x7(
        (A(a,x1)∨¬B(y1)∨¬C(y1,x1)∨¬D(x1))∧
        (A(b,x2)∨¬E(x2)∨¬F(x2))∧
        (B(x3)∨¬F(x3))∧
        (E(c))∧
        (F(x5))∧
        (A(d,x6)∨¬F(x6))∧
        (B(x7)∨¬G(x7)))
```

Now, suppose we are trying to prove `A(x,y)` for some `x` and `y`. The way this is done is by stating that there are no values of `x` and `y` such that `A(x,y)` is true, in other words `∀x∀y(¬A(x,y))` is added to the set of disjunctions. Again, we shall call this point ❿. At this stage, it can be noted that there are no exact matches. However, if we were a little more precise in our requirements, and stated that `a` was substituted for the variable `x` and `x1` for the variable `y`, we would have `¬A(a,x1)` which does match with the positive literal in clause ①. It is always possible to substitute any variable for any value we like, since the variables are universally quantified.

This time we shall accompany each step in the resolution not only with the clause number used for the resolution, but also with the extra variable bindings needed to make the match. So we have:

❶ By resolution with clause ①, using substitution `{x/a,y/x1}` we get `{¬B(y1),¬C(y1,x1),¬D(x1)}`

Note that we can abandon the set notation for the literals, since we are now treating them as a list, left-to-right. Substitutions of variable names for variable names can be omitted from the set of substitutions if we apply the substitution throughout, and it does not matter which name we use, so we can describe the state more concisely as:

`B(y1),C(y1,y),D(y)` substitution `{x/a}`

The next step uses the substitution `{y1/x3}` so we get:

❷ By resolution with clause ③, `F(y1),C(y1,x1),D(x1)` substitution `{x/a}`

❸ Then by resolution with ⑤, `C(y1,x1),D(x1)` substitution `{x/a}`

At this point, no further resolution can take place, so it is necessary to backtrack to point ❶ and try an alternative resolution for `B(y1)` using clause ⑦:

❹ By resolution with clause ⑦, `G(y1),C(y1,x1),D(x1)` substitution `{x/a}`

Again, there is no further resolution possible, so it is necessary to go right back to point ❿ and consider an alternative resolution for `A(x,y)`. Note that we go back to that point completely, so the substitutions tried in points ❶, ❷, ❸ and ❹ are forgotten.

❺ By resolution with clause ②, `E(y),F(y)` substitution `{x/b}`

Now if we add the substitution `y/c`, we can resolve on clause ④. Note the substitution carries on to the literal which takes `y` as an argument:

❻ By resolution with clause ④, `F(c)` substitution `{x/b,y/c}`

At this point, we can resolve on clause ⑤ providing we substitute `c` for `x5`:

❼ By resolution with clause ⑤, □ substitution `{x/b,y/c}`

This gives us the empty clause, □, so we can conclude that `¬A(x,y)` is contradicted when we make the substitution `{x/b,y/c}`. In other words, `A(b,c)` is a logical conclusion from the original clauses.

Note, however, that we have not fully explored the possibilities. We have found one possible substitution in `¬A(x,y)` but there may be others.

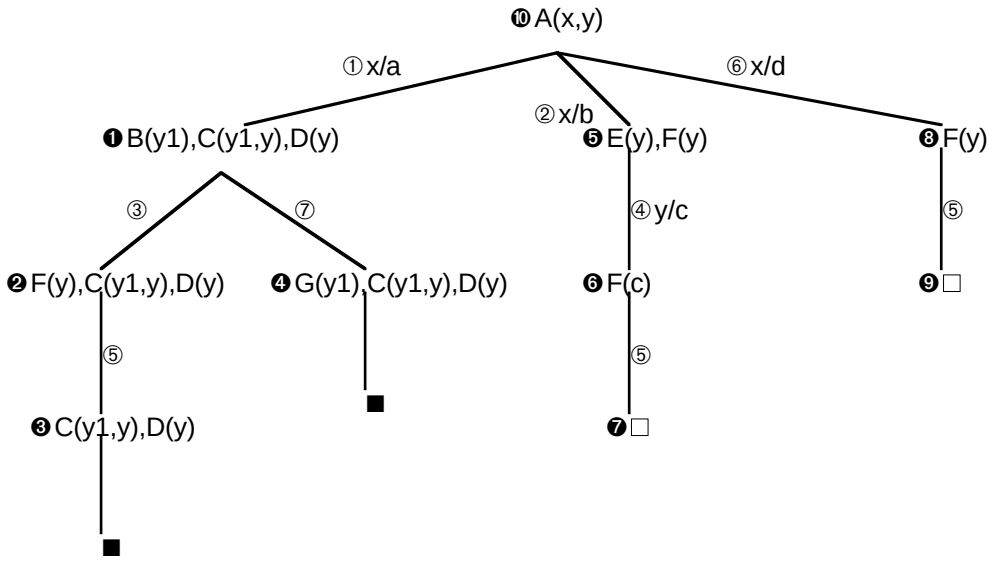
At point ❿ it is also possible to resolve on clause ⑥ providing we make the substitution `{x/d,y/x5}`:

❽ By resolution using clause ⑥, `F(y)` substitution `{x/d}`

By the variable for variable substitution `y/x5` and resolution with clause ⑤ we obtain □.

So we can conclude that ¬A(x,y) is contradicted when we make the substitution {x/d} with y remaining a variable. This means we can conclude ∀yA(d,y).

All this is probably better explained with the SLD tree:



In this case, the arcs of the tree are labelled with both the clause used and the substitution necessary for resolution. States which cannot be resolved further are shown leading to a leaf marked ■,representing failure to find a solution. The variable substitutions which give possible solutions may be found by tracing from the root down to leaves marked □ and applying the substitutions on the arcs to the atom at the root.

The process of substituting values for variables to make two terms identical for use in resolution is called *unification*. To formally define it, we say that two terms t_1 and t_2 are unifiable if there is a substitution for some or all of the variables in t_1 and t_2 such that the terms that result from the substitutions are syntactically identical terms. A substitution consists of a set of substitutions for individual variables. The *most general unifier* of two terms is that unifier which does not have any subsets which are also unifiers. The most general unifier is used to ensure that all possible solutions are returned.

The algorithm for constructing the most general unifier of two terms involves matching up the arguments of the terms. A variable `x1` matched against a variable `x2` unifies and adds the substitution `x1/x2` to the set of substitutions (initially empty) that will form the most general unifier. A variable `x` matched to a constant a or vice versa causes the substitution `x/a` to be added. If two constants are matched, no substitution is added, but the unification fails unless the constants are identical. Although we didn't show any examples above, unification encompasses functions. A variable `x` matched with a function with some arguments `f(…)` or vice versa causes `x/f(…)` to be added to the substitution. If two compound terms are matched, unification fails unless they both have the same function name and the same number of arguments, if they do then unification continues with the individual arguments of the compound term. A compound term cannot be matched with a constant, but a variable matches with a compound term provided that the compound term does not contain any occurrences of the same variable.

For example, if we have `P(x,f(y,g(b))` and `P(h(a),f(z,w))` where `a` and `b` are constants and `w`, `x`, `y` and `z` are variables, the most general unifier is `{x/h(a),y/z,w/g(b)}`, giving the unified term `P(h(a),f(y,g(w))`. If we tried to unify `P(x,f(y,g(w))` with `P(h(a),f(z,w))` however, unification would fail since it would require the substitution `w/g(w)`. The check that no variable is being substituted for a term containing that same variable is known as the *occur check*.

As previously, a clause may be used as many times as necessary in a proof of absurdity from an initial goal, but it is necessary to assume that each time it is used it has a fresh set of variables. This is because if $x_1,\ldots,x_n$ are the variables in some clause C, we can write $\forall(x_1,\ldots,x_n)C\wedge S$ as $\forall(y_1,\ldots,y_n)C\wedge\forall(x_1,\ldots,x_n)C\wedge S$, and then any bindings of $x_1,\ldots,x_n$ caused in resolving with S mean there is still a copy of C with fresh variable $y_1,\ldots,y_n$ avialable.

The Prolog Language

We have now covered enough logic to be able to relate the programming language Prolog to logic. A program in Prolog is essentially a set of Horn clauses. Programs are executed by issuing an initial query, and then resolving using SLD resolution with the left-to-right selection rule. The conventional notation for Prolog is slightly different from that we have used in this explanation using logic. Prolog distinguishes between variables and constants by starting variables with upper case letters and constants with lower case letters. Lower case letters are also used for predicate names. The symbol :– is used for logic ←, and commas are used in the place of the ∧ symbols to the right-hand side of the ←. Prolog clauses are ended with a full stop. So the Prolog equivalent of the example we discussed above would be:

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

In general Prolog does not use the occur check in unifications. It also has some additional "extra-logical" constructs. Apart from that, it can be seen that executing a program in Prolog is just a form of proving the initial call true as a fact in logic (with the condition, if necessary, that variables in it are bound to values determined by unification), with the Prolog program being just a logical description of relationships we hold to be true.

The ability to re-use clauses means that recursion is available in Prolog, and in fact recursion gives it much of its power, and takes the place of the iterative loops in imperative languages. In this way it is similar to the functional languages like Miranda. Both the functional languages and the logic languages are known as *declarative languages*. This is because programming in them is supposed to consist of declaring logical or functional relationships from the problem, rather than thinking in terms of what the machine is actually doing and issuing it "orders". In fact the goal of pure declarative programming is freeing the programmer from having to think in low-level machine terms at all. It is often let down, however, by the need to have non-declarative primitives for such things as input-output, or by having to consider various problems in the order of computation in declarative languages, such as the problem in Prolog of getting caught in an infinite search tree.

In the functional programming paradigm there are many programming languages built around the abstract functional model. You have seen Miranda, ML is another example, but the most famous (and oldest) functional language is Lisp, which is used in a lot of AI work in the USA. Lisp has some disadvantages since it was first developed before functional languages were properly understood, but as is often the case in computing, it has had the benefit of being the first established and thus difficult to displace. The same sort of comment could be made about Prolog. There have been many suggestions for languages which like Prolog are built on the model of being based on logical deduction using resolution, but which improve on Prolog in some way. None of them however has achieved any widespread use. We will leave further discussion of Prolog to another set of notes. Many of the things discussed here may seem clearer when seen in the context of practical experience with the Prolog programming language.

Is Logic Adequate?

We have discussed logic at length because it has the advantage of precision. The use of models enables us to define precisely what is meant by a statement in logic. Resolution enables us to draw precise inferences from logic. But the question remains is predicate logic adequate to store all forms of knowledge and to carry out all forms of reasoning that we might require an artificial intelligence system to do?

The answer is no, but logicians are working on many ways of extending and varying predicate logic to enable it to capture more aspects of the "real world". There is not space here to go into details of

various proposed varieties of logic, but we shall just discuss briefly some of the issues that have led to them being proposed. It should be noted that whereas logic, as covered in Introduction to Logic and in the notes above is well-defined and accepted, and for this reason may be termed *classical logic*, other forms of logic are the subject of a great deal of debate.

We have already mentioned the microworld problem of having to assume that we know all that there is to know, and therefore anything that can't be proved true is false. Predicate logic as described above has the property of *monotonicity*, which means that anything that can be proved true from the premises will still remain possible to be proved true however many extra premises are added. I deliberately avoided building on the suggestion that something can be proved false by failing to prove it true, because if we accept this our logic loses this property and becomes *non-monotonic*. For example, if we say that ¬B is true if B can't be shown to be false then with:

$A \leftarrow \neg B \wedge C$

C

since we don't have B as a fact or any way of proving B we could say that A is true, but as soon as we add the fact B, without any further rules for A, A becomes false. The issue of non-monotonic reasoning is a major subject in AI research.

One way that has been proposed to get round the problem that we may not always know or be able to prove that something is true or false is the idea of *three-valued logic*. In this variant of logic, an atom can be either true, false or unknown. The same connectives as in classical logic, but truth tables must have extra rows and spaces to allow for the third truth value. We know that we can only definitely say $A \wedge B$ is true if both are true, and if either is false $A \wedge B$ is false even if we don't know the value of the other, so the truth value for $\wedge$ in three-value logic is:

$A \wedge B$	t	f	u
t	t	f	u
f	f	f	f
u	u	f	u

Similarly, the truth table for $\vee$ is:

$A \vee B$	t	f	u
t	t	t	t
f	t	f	u
u	t	u	u

Such a system might have to deal with the possibility that the situation may change as we get to know things that were unknown. Another sort of reasoning system that enables us to deal with the fact that we may not know everything enables us to make deductions based on assuming something we don't know is true or false. We might assume that it has the value we think most likely, but in a safety critical system it could be better to assume the worst unless we have proof otherwise. We need to keep a track of our assumptions so that if we learn some new knowledge we may change the deductions we make from them. We might also want to change our assumptions if we find they lead to contradictions, and we will probably want to be able to say what assumptions we have made when we give an answer. We will cover such reasoning systems (termed "Truth Maintenance Systems") later in this course.

One of the most glaring difference between classical first-order logic and natural language is that tense is a vital part of most sentences in natural language, whereas statements in predicate logic are timeless. Our semantics for classical logic assumed that some atom could be considered either always true, or always false. If we want to reason about situations that change over time, or to be able to provide logic representations which distinguish between English language sentences like "John had eaten an apple", "John ate an apple", "John was eating an apple", "John has eaten an apple", "John eats an apple", "John is eating an apple", "John will eat an apple", "John will have eaten an apple" and so on, we need a form of logic which can deal with time. The range of sentences above show some of the complexity of the situation (English has a particularly complex tense system, note how many variations are introduced by words like "is" and "had", which are referred

to as *auxiliary verbs*); the precise meaning of different statements involving time is a matter of debate for linguists, but a precisely defined logical notation for time would help codify the situation. Forms of logic which deal with time are known as *temporal logics*.

Related to this is the idea of *modal logics* which deal with what is possible and what is necessary. Simple modal logic adds two operators to first order classical logic: $\Box A$ where A is any sentence in modal logic has the meaning "A is necessarily true", while $\Diamond A$ has the meaning "A is possibly true". This is often explained in terms of *possible world semantics*, so $\Box A$ is interpreted as A is true in every possible world, while $\Diamond A$ is interpreted as A is true in some possible world. Rules for manipulating sentences involving $\Box$ and $\Diamond$ are then developed.

If we are interested in modelling human behaviour, we have to take into account knowledge and belief. We need to be aware that it is possible to hold contradictory beliefs, and the more philosophical side of AI can give many examples of the difficulties of representing beliefs in a formal system. A common one is to note that someone may not realise that two names he uses refer to the same person, and thus may hold contradictory beliefs about that person. For example, I may believe that Professor Jones whom I know is a nice man, while the anonymous reviewer of my paper who criticised it unfairly is not a nice man, without realising that the anonymour reviewer was actually Professor Jones, so my beliefs `Nice(Jones)` and `¬Nice(reviewer(MyPaper))` are contradictory, but need to be represented if my beliefs about niceness are to be properly represented. Another form of logic useful for modelling human behaviour, *deontic logic*, is concerned with what people are obliged to do or not do, so is necessary for representing English sentences involving words like "should" and "ought".

A common extension of logic introduces certainty factors. This stems from two considerations. Sometimes we may not be certain whether a particular atom is true or false, or whether a particular rule can be applied, but we may be able to assign a numerical factor to our belief. So, for example, if in some situation we are 90% certain that an observation A holds, and 80% certain that we can make the deduction $A \rightarrow B$, we could say that we multiply the two certainty factors to give the conclusion that we are 72% certain that B holds. It should be noted that this is not the only way of dealing with certainty factors.

In the above case, we are not saying that B holds partially, only that we can't be certain that it holds. Another form of logic, *fuzzy logic*, does hold that predicates can be partially true. The idea is that a statement like "John is tall" is not either true or false, depending on some exact cut-off point of height, but something which is true or false to some degree. This also enables us to deal with sentences involving degrees to which some predicate applies such as "John is quite tall" or "John is very tall".

We will consider logic with certainty factors and degrees of truth in further detail later on in this course.

Further Reading

In the further reading section of the previous set of notes, the "Sussex tradition" of emphasising the cognitive and huamn sides of Artificial Intelligence was emphasised. We can also talk about the "Imperial tradition" of emphasising the logic side of Artificial Intelligence, as much research work on logic programming has taken place at Imperial College. Two useful books for further reading and explanation of the material covered written by authors at Imperial College are:

R.A.Kowalski *Logic for Problem Solving* Elsevier-North Holland, 1979. This is the book which originally showed how predicate logic could be thought of as a programming language.

C.J.Hogger *Essentials of Logic Programming* Clarendon Press, 1990. This book is probably the closest to the material in this chapter, but is obviously more detailed and thorough.

An introduction to the implementation of Prolog, and also to another form of logic language involving parallel processing is given in:

J.D.Newmarch. *Logic Programming: Prolog and Stream Parallel Languages* Prentice Hall 1990.

More detail on the use of ideas from logic programming in parallel processing are given in:

A.Takeuchi. *Parallel Logic Programming* Wiley, 1992.

Two other books which cover logic with the specific intention of showing its use as a programming language are:

J.W.Lloyd *Foundations of Logic Programming* Springer-Verlag, 1984.

U.Nilsson and J,Małuszyński *Logic, Programming and Prolog* Wiley, 1990.

A simple introduction to non-classical logics is:

R.Turner *Logic for Artificial Intelligence* Ellis Horwood 1984.