CS302 — Logic Programming

Gordon Royle

August 11, 1999

Contents

1 Introduction

This document consists of the lecture notes for the Logic Programming course given at the University of Western Australia. This course involves both theoretical logic programming and its practice as exemplified by the Prolog programming language. The theoretical portions are heavily based on [4], and a portion of this text is essentially a reordering of the material contained therein. The portions on Prolog and propositional logic are based on material from a variety of sources ([1],[6] [2],[5]).

Consider the following two statements:

> If the sun is shining then I wear sunglasses
> The sun is shining

If both of these statements are true, then it is reasonable to conclude that I will be wearing sunglasses. This is an example of natural inference — the *conclusion* "I am wearing sunglasses" follows naturally from the two *premises*. Now consider the following two statements:

> If inflation is increasing then the price of petrol will go up
> Inflation is increasing

Now if both of these statements are true, then we would certainly conclude that the price of petrol will go up. This is another example of natural inference.

It is clear however that these two examples are really just the same thing using different words. The underlying principle is that if we have two statements of the form

> If P is true then Q is true
> P is true

then we can infer that as a consequence Q is true.

More to the point, however is that if a computer is presented with two statements of that form, then it can deduce that Q is a consequence with no understanding of what the statements actually mean. That is, we have specified a purely mechanical rule that mimics our natural reasoning.

For example we could present the computer with the statements

> $Q \leftarrow P$
> P

and the computer would infer Q. We would understand that P and Q referred to particular statements about the real world, but which statements they were is irrelevant to the chain of reasoning performed by the computer.

Logic is the branch of mathematics that deals with expressing statements about the real world in a suitably precise symbolic form, and then processing those symbols using suitably chosen rules. If we believe that the initial statements are true and agree that the mechanical rules are acceptable, then we will accept that the statements produced in this way are true. Natural languages tend to be ambiguous or fuzzy, and it is often difficult to express something very precisely in (say) English. This ambiguity can often make it difficult to decide whether a chain of reasoning (for example, a mathematical proof) is actually correct or not. Using logic removes this ambiguity. In fact it was

one of the great aims of the early 20th century to reduce all of mathematics to this sort of mechanical symbol manipulation. Hilbert issued a challenge to mathematicians to formulate a program whereby once and for all a set of axioms for the foundations of mathematics would be agreed upon, along with a set of inference rules. Then every theorem of mathematics could be unequivocally proved by a suitable mechanical application of the rules and all arguments about what constitutes a proof and differences of opinion would be resolved. This dream was ruined by Gödel who proved that no matter what system was chosen, and what inference rules, then either it would be a contradictory system or there would be some true facts that could not be proved in this fashion.

The subject of logic programming arose out of the realization that logic could be effectively used as a programming language. The basic idea is that the computer solution to a problem consists of two components, the logic of the problem and the control of the problem. The logic describes exactly what the problem is, and the control describes exactly how it is to be solved. These two aspects can be regarded as the *declarative* aspect and the *procedural* aspect of problem solving. In programming in a procedural language such as Pascal or C the programmer must specify precisely what steps are to be taken by the computer. The aim of logic programming is pure declarative programming, in which the programmer merely specifies the problem, and leaves the control completely to the computer. We will be considering the theory of logic programming, and its practice as embodied by the Prolog programming language.

Logic programming in general and Prolog in particular are usually used for non-numerical programming such as expert systems, automatic theorem proving, artificial intelligence and machine learning.

2 Propositional Logic

A logic consists of two things — a *language* and some *inference rules.* The language is simply a set of strings of symbols that will be used in some way to represent statements about the real world, and inference rules are rules that allow one to manipulate the symbols. A logic is useful if the language is expressive enough to allow one to represent reasonably complicated statements, and the inference rules correspond to our natural reasoning processes. The *syntax* of a language determines the legal formulas of the logic. The syntax simply consists of mechanical rules for selecting legal symbols, and rules for constructing legal formulas from those symbols. The *semantics* of a language is concerned with the *meaning* that we attach to the formulas in that language. That is, the semantics is concerned with how a statement about the world can be expressed as a formula, and conversely what a formula in the language is stating about the world.

2.1 Syntax and semantics of propositional logic

In this section we shall introduce propositional logic. First we introduce the legal symbols of propositional logic, and explain how to combine them to form the formulas of the language.

Definition. 2.1 *The symbols of propositional logic consist of the following classes:*

1. *The truth symbols T and F.*
2. *Propositional symbols.*
3. *The connectives $\sim$, $\wedge$ and $\vee$.*
4. *The improper symbols "(", ")", and ",".*

We shall usually use the letters P, Q, R, ... and p, q, r, ... for the propositional symbols, though in Prolog we often use complete lower case words. The improper symbols are merely present to indicate precedence among the connectives.

Definition. 2.2 *A* proposition *is either a truth symbol, a propositional symbol, or a formula formed from propositions P and Q in one of the following ways:*

1. $\sim P$
2. $P \vee Q$
3. $P \wedge Q$

Example. 2.3 The following are propositions: $p \vee q$, $(p \wedge \sim q) \vee (s \wedge t)$ and $p \vee (\sim q \vee \sim s)$. ■

The above definitions give the syntax of propositional logic. Using the above rules we can write down legal propositions. However in order for the logic to be useful, we must relate it to the world we wish to formalize. To do this we need the concept of an interpretation:

Definition. 2.4 *An* interpretation *or* truth assignment $\mathcal{I}$ *is an assignment of the truth values* T *and* F *to each of the propositional symbols.*

We are going to use the sentences of propositional logic to model some part of the real world. Each propositional symbol represents a *proposition* — a definite statement about the world which is either true or false. The formulas of propositional logic are simply formal strings of symbols, and an interpretation provides the connection between them and the real world, by stating which propositional symbols refer to true statements. The same formal symbols can be used to model many different situations; the interpretations of those symbols will generally differ.

The semantics of propositional logic are the rules for determining whether a given formula represents a true or false statement. The truth symbols T and F represent unconditionally true and false statements respectively. The truth value of a propositional symbol is (conditionally) determined by an interpretation. Once an interpretation has been given, the truth values of other propositions are determined according to the truth values of their component parts. The following truth tables give the semantics of the three connectives:

P	$\sim P$
T	F
F	T

P	Q	$P \vee Q$
T	T	T
T	F	T
F	T	T
F	F	F

P	Q	$P \wedge Q$
T	T	T
T	F	F
F	T	F
F	F	F

As each propositional symbol represents some definite true or false statement, so do the compound propositions. Examining the truth tables we see that $\sim P$ coincides with the natural language "not P", $P \wedge Q$ with "P and Q", and $P \vee Q$ with "P or Q". Using truth tables we can work out the truth values of any compound proposition, depending on the truth assignment.

Example. 2.5 The truth table for the proposition $(p \wedge \sim q) \vee (s \wedge t)$ of the previous example. is shown in Figure 1. ■

2.2 Clausal notation

Whilst we can form many legal formulas with the above connectives, for the purposes of logic programming we are particularly interested in a certain type of formula called a *clause*. First we need a couple of definitions.

Definition. 2.6 *A* literal *is a propositional symbol* P *or its negation* $\sim P$ *(and they are called respectively a* positive literal *and a* negative literal*).*

Definition. 2.7 *A* clause *is a formula of the form*

$$A_1 \vee A_2 \vee \ldots A_k \vee \sim B_1 \vee \sim B_2 \vee \ldots \vee \sim B_m$$

where each A_i *and* B_i *is a positive literal.*

p	q	s	t	$\sim q$	$p \wedge \sim q$	$s \vee t$	$(p \wedge \sim q) \vee (s \wedge t)$
T	T	T	T	F	F	T	T
T	T	T	F	F	F	T	T
T	T	F	T	F	F	T	T
T	T	F	F	F	F	F	F
T	F	T	T	T	T	T	T
T	F	T	F	T	T	T	T
T	F	F	T	T	T	T	T
T	F	F	F	T	T	F	T
F	T	T	T	F	F	T	T
F	T	T	F	F	F	T	T
F	T	F	T	F	F	T	T
F	T	F	F	F	F	F	F
F	F	T	T	T	F	T	T
F	F	T	F	T	F	T	T
F	F	F	T	T	F	T	T
F	F	F	F	T	F	F	F

Figure 1: The truth table for $(p \wedge \sim q) \vee (s \wedge t)$

We use a special clausal notation for clauses as follows

Definition. 2.8 *The clause*

$$A_1 \vee A_2 \vee \ldots A_k \vee \sim B_1 \vee \sim B_2 \vee \ldots \vee \sim B_m$$

is denoted

$$A_1, A_2, \ldots, A_k \leftarrow B_1, B_2, \ldots, B_m$$

This again is merely a formal definition, so let us consider a simple example and try to decide the "meaning" of $\leftarrow$. To do so we consider the truth table for $Q \leftarrow P$ which is the truth table for $Q \vee \sim P$.

P	Q	$\sim P$	$Q \vee \sim P$	$Q \leftarrow P$
T	T	F	T	T
T	F	F	F	F
F	T	T	T	T
F	F	T	T	T

Notice that $Q \leftarrow P$ is true, provided that Q is true whenever P is true. This expresses intuitively what we mean by "is implied by" — Q is implied by P is true provided that Q is true whenever P is true. More simply we can just take the semantics of $\leftarrow$ to be "if": $Q \leftarrow P$ can be read as "Q is true if P is true".

A similar analysis of the most general clause reveals that it can be read as B_1 AND B_2 AND ... AND B_m implies A_1 OR A_2 OR ... OR A_k. Notice that the commas on the left of the $\leftarrow$ are to be read as ORs whilst the commas to the right of the $\leftarrow$ are to be read as ANDs. Restricting our attention to clauses does not result in any loss of generality as far as propositional logic goes, because any proposition can be expressed in clausal form. In fact we go a bit further, and logic programming deals almost exclusively with a special subset of the clauses called Horn clauses.

Definition. 2.9 *A* Horn clause *is a clause with at most one positive literal.*

The two parts of the clause (the positive literals on the left hand side of the $\leftarrow$ and the negative literals on the right hand side of the $\leftarrow$) are called the *head* and *body* of the clause respectively, or more impressively the *consequent* and the *antecedent* of the clause.

Therefore Horn clauses come in 4 varieties:

$A \leftarrow B_1, B_2, \ldots, B_k$ The semantics of this are that A is true if ALL of the B_i's are true.

$\leftarrow B_1, B_2, \ldots, B_k$ Taking this out of the special clausal notation, we see that it says that at least one of the B_is must be false. One can also see this by interpreting the empty OR in the head of the clause to mean F.

$A \leftarrow$ Now, an empty AND is taken as true, so this clause is true precisely when A is true. Thus in a clause of this form the arrow is superfluous — we shall omit the arrow whenever convenient.

$\leftarrow$ Given the above comments on the meaning of empty conjunctions and disjunctions, the above clause can be read as "true implies false" and hence can never be true. This empty clause is to be understood as a contradiction. We use the special notation $\Box$ for the empty clause.

Any proposition P can be expressed in *clausal form*, that is, one can find a set of clauses S such that P is true precisely when all the clauses of P are true. Basically one replaces all propositions of the form $P \wedge Q$ by the set of two propositions $\{P, Q\}$ (it is slightly more complicated than this). Practical logic programming is almost always done only with clauses; for propositional logic there is nothing lost.

2.3 Logical Implication

We start this section with two examples. Consider two natural language propositions as follows "It is raining" and "If it rains then the farmers are happy". Suppose we accepted that those two clauses were true at that particular moment . Then almost all of us would regard it as very natural to *infer* that the proposition "the farmers are happy" is also true. Our natural reasoning tells us that "the farmers are happy" is *implied* by the two propositions.

In propositional logic, let us consider the set of propositions $S = \{P, Q \leftarrow P\}$. Suppose under some interpretation $\mathcal{I}$ all the propositions in S are true (we say that S is *satisfied* by $\mathcal{I}$). Then we can conclude that Q must also be true under $\mathcal{I}$ by considering all possible values of P and Q in a truth table

P	Q	$Q \leftarrow P$
T	T	T
T	F	F
F	T	T
F	F	T

Notice that the only case when P and $Q \leftarrow P$ are both true is the first line in the table, and Q is true in that case. So, we are tempted to follow our natural reasoning and say that we can conclude that P is implied by the propositions S.

Definition. 2.10 *A proposition P is said to be* logically implied *by the set of propositions S, provided that P is true whenever S is true. In this situation, P is called a* logical consequence *of S.*

The idea of logical implication in propositional logic mirrors our own views of the nature of reasoning. Despite a complete lack of knowledge of farmers and rain, propositional logic states that "the farmers are happy" is a conclusion that can be drawn provided that both the propositions "it is raining" and "the farmers are happy $\leftarrow$ it is raining" are true. Propositional logic is a *formalization* of our own reasoning process.

Therefore we can state that our main aim in propositional logic programming is to find the logical consequences of a given set of propositions. If we can state some facts and relationships about a particular situation as a set S of clauses in propositional logic, then propositions that are logical consequences of S will be accepted as true statements about that situation. However the finding of logical consequences does not require any particular knowledge about the specific situation, but is merely the mechanical application of the rules of inference of propositional logic. In particular a computer can be programmed to find logical consequences of a set of clauses.

It is always possible in propositional logic to find logical consequences by the truth table method. However, if there are n literals contained in all the propositions, then the truth table must have 2^n lines, which rapidly escalates beyond feasibility. Therefore we must consider other techniques for proving logical implication.

2.4 Inference

We shall consider a rule of inference called *resolution*. Consider two clauses, one involving a positive literal L, and another involving its negation $\sim L$. Suppose P is the clause $L \vee A_1 \vee A_2 \vee \ldots \vee A_k$ and Q is the clause $\sim L \vee B_1 \vee B_2 \vee \ldots \vee B_m$ (where here A_i, B_i may be either positive or negative literals). The *resolvent* of P and Q is the clause $A_1 \vee A_2 \vee \ldots \vee A_k \vee B_1 \vee B_2 \vee \ldots \vee B_m$. Now, this resolvent is a logical consequence of P and Q. This is easy to see by considering whether L is true or false. If L is true, then $\sim L$ is false, and hence at least one of the B_i is true, and thus the resolvent is true. We simply reverse the argument if L is false. Resolution is the name given to the process of finding resolvents of clauses.

Therefore resolution allows us to find some logical consequences of a given set of clauses. Two questions fundamental to the theory of logic programming are those of *soundness* and *completeness*. We say that the resolution rule is *sound* because all the clauses it produces are logical consequences of the initial set of clauses. An inference rule is called *complete* if every clause that is logically implied by a set of clauses can be found by repeated applications of that rule. We have already indicated that resolution is sound, and we shall discuss completeness later.

So consider the following brute force technique for finding logical consequences of a set of clauses S. Choose any two clauses in S that can be resolved, find their resolvent and add it to the set S. Repeat this process until the resulting set S^* is closed under the operation of resolution. The set S^* is called the resolution closure of S. Now, anything that is in the resolution closure S^* is a logical consequence of S.

This leads to a simple deductive technique called *forward chaining inference*. Forward chaining inference means that one starts with the propositions in S and applies the resolution rule until all logical consequences are found, or one particular desired proposition is found. The sequence of steps taken is considered to be a proof of that proposition.

Consider the set of clauses

p

q

r

$w \leftarrow p, r$

$v \leftarrow w, q, s$

$s \leftarrow w$

Let us consider a proof using forward chaining inference that the clause v is a logical consequence of S. At each step we indicate two clauses to be resolved, and their resolvent. The two clauses are written one above the other, then a vertical line, and then the resolvent of those two clauses.

$$\begin{array}{l|l} p & \\ w \leftarrow p, r & w \leftarrow r \end{array}$$

$$\begin{array}{l|l} r & \\ w \leftarrow r & w \end{array}$$

$$\begin{array}{l|l} w & \\ v \leftarrow w, q, s & v \leftarrow q, s \end{array}$$

$$\begin{array}{l|l} q & \\ v \leftarrow q, s & v \leftarrow s \end{array}$$

$$\begin{array}{l|l} w & \\ s \leftarrow w & s \end{array}$$

$$\begin{array}{l|l} s & \\ v \leftarrow s & v \end{array}$$

So we observe that after a series of resolutions we obtain the clause v. Therefore v is a logical consequence of S. The same result could have been achieved via truth tables, but would have been considerably more tedious.

2.5 Refutation

Refutation is another technique for applying the resolution rule for inference which is particularly useful for determining whether a specific clause is a logical implication of a set of clauses. Forward chaining inference starts with the given clauses, and produces new clauses obtained by repeated applications of the resolution rule. However it is hard to direct this practice to obtain a proof of a specific logical implication, and simply proceeding at random and hoping that it turns up sooner or later is rather inefficent. Therefore, in practice refutation is the most important method of applying resolution. We start with some preliminary results.

Theorem. 2.11 *Let S be a set of propositions. Then the proposition P is logically implied by S if and only if $S \cup \{ \sim P\}$ is unsatisfiable.*

PROOF. Suppose P is logically implied by S. Then whenever all the propositions in S are true, so is P, and hence $\sim P$ is false. Thus $S \cup \{ \sim P\}$ can never be satisfied. Conversely, suppose that $S \cup \{ \sim P\}$ is unsatisfiable. Then whenever a truth assignment satisfies S, it must be the case that $\sim P$ is false, and hence that P is also true. ■

This theorem reduces the problem of finding logical consequences to the problem of deciding whether a set of propositions is unsatisfiable. As explained above, this is equivalent to deciding whether a set of clauses is unsatisfiable. The following theorem shows that this is always sufficient to prove any logical implication.

Theorem. 2.12 (Completeness of Propositional Resolution) *Let S be a set of clauses. Then S is unsatisfiable if and only if $\Box \in S^*$.*

Therefore we now have a procedure for proving that a proposition P is a logical consequence of the set of propositions S.

1. Form the set $S_1 = S \cup \{ \sim P\}$.
2. Put S_1 into clausal form.
3. Use resolution on S_1, and demonstrate that $\Box \in S_1^*$.

Notice however that we have omitted all mention of *control* from the above procedure. The completeness theorem for propositional resolution guarantees that there exists a proof by refutation, but this does not necessarily help us to actually find it. Much of the divergence between theoretical logic programming, and the practical logic programming embodied by languages like Prolog is based on the necessity for controlling the actual execution of the resolution process. In addition we have not specified what happens if the above process fails. Does this mean that P is not a logical consequence of S? Or simply that we have not been able to find a proof? The completeness theorem guarantees that if we manage to search the whole of S_1^* without finding $\square$ then P is not a logical consequence of S.

2.6 Propositional Logic in Prolog

Prolog is a computer language designed to prove logical implication by refutation. We shall see more details later as to exactly how Prolog works. One aim of Prolog is that it be as similar to mathematical logic as possible.

Prolog works by maintaining a database of clauses that are considered to be the known facts. Then interactively, at the prompt the user enters a proposition to be proven. Prolog negates that proposition, adds it to the database and then tries to prove that the resulting set of clauses is unsatisfiable by refutation. If it succeeds in showing that the set of clauses is unsatisfiable, then the response is "Yes" and then the proposition is proven, otherwise the response is "No", and the given proposition does not follow from the database. The database of clauses is usually called the *program* and the proposition to be proven is called the *goal*.

Example. 2.13 Consider the rain/farmers example of earlier. We saw that it could be formalized as the two clauses p and $q \leftarrow p$. In Prolog, we enter these as the program almost exactly as written.

```
p
q :- p.
```

This is loaded into Prolog from file by the command

```
consult('filename')
```

Then in response to the quote from Prolog (which is `| ?-` in CProlog) one types in the proposition to be proved

```
| ?- q.

yes
```

and Prolog returns a `yes` indicating that q is indeed a logical consequence of the clauses in the database. ■

We can do our previous example as well:

```
p.
q.
r.
w :- p,r.
v :- w,q,s.
s :- w.
```

can be entered as the database (most conveniently by saving this in a file and using the `consult` command, and then questioned:

```
| ?- v.

yes
```

We can do more interesting examples by modelling a simple real-life situation. Consider a student who has taken various classes and wishes to see if he [1] has satisfied the mathematics requirements for a degree. The mathematics requirements are satisfied if he has done the calculus sequence, the finite mathematics sequence and the algebra sequence. We could express this in propositional Prolog by

```
mathReq :- calcSeq, discreteSeq, algSeq.
```

where the proposition *calcSeq* is true if the student has passed the calculus sequence and so on. Now, each of these sequences requires certain courses to have been taken.

```
calcSeq :- basicCalc, advCalc.
```

indicates that the calculus sequence requires one to take basic calculus and advanced calculus. Basic calculus can be satisfied in three ways

```
basicCalc :- takenCalc1, takenCalc2.
basicCalc :- takenCalcA, takenCalcB.
basicCalc :- schoolCalc.
```

either by taking the courses *Calc1* and *Calc2* or by getting an exemption based on school calculus, or by taking the engineering sequence *CalcA* and *CalcB*. Notice that an OR statement is simply expressed by writing down several separate Horn clauses. The advanced calculus is just one further course.

```
advCalc :- takenM222.
```

The discrete mathematics requires one to take some graph theory and combinatorics OR one of those two together with numerical analysis.

```
discreteSeq :- takenGraph, takenComb.
discreteSeq :- takenGraph, takenNumA.
discreteSeq :- takenComb, takenNuma.
```

Finally the algebra requirements are linear algebra and group theory.

```
algSeq :- takenLin, takenGroup.
```

Then for any particular student we could enter the courses that they have taken by specifying that certain propositions are true.

```
takenGraph.
takenNuma.
takenLin.
schoolCalc.
takenM222.
```

Finally we can now question the system.

```
| ?- mathreq.

no
```

So we discover that the mathematics requirements are not met. Of course we don't know why. We can check each of the individual sequences.

[1] The use of "he" should not be taken to imply specific gender. It may be read "he or she", the latter phrase being too ugly to repeatedly use

```
| ?- calcSeq.

yes

| ?- discreteSeq.

no
```

and so we discover that at least one problem is that the discrete sequence is not satisfied.

This example is very simple and could easily be done by hand. Furthermore it is not very useful because it simply responds `yes` or `no` rather than giving more helpful information. Nevertheless this kind of rule based system is at the core of modern deductive databases, and many kinds of expert system.

Another reason that this system is so limited is that the set of clauses applies only to one particular student. If we wish to examine the case of another student, we must delete or enter all the clauses that are different. We can only avoid this by introducing variables into our logical system, and for this reason we consider the predicate logic.

3 Predicate Logic

Predicate logic is a much more powerful logic than propositional logic, and it allows one to describe much more complicated situations. Anything that can be described in propositional logic can be described unchanged in predicate logic. As we stated earlier a logic consists of a language and some inference rules. The language of predicate logic is a *first-order language*, and once again we shall use the resolution rule as our only inference rule.

3.1 First order languages

A first order language is set of formulas formed from an alphabet of symbols according to specific rules.

Definition. 3.1 *An alphabet consists of symbols in the following classes.*

1. *Variable symbols*
2. *Constant symbols*
3. *Function symbols*
4. *Predicate symbols*
5. *The connectives* $\sim$, $\wedge$, $\vee$, $\rightarrow$ *and* $\leftrightarrow$
6. *The quantifiers* $\forall$ *and* $\exists$.
7. *The improper symbols "(", ")", and ",".*

Conventionally, we shall use u, v, w, x, y, z or subscripted versions of these for variables, f, g, h, ... for functions, and p, q, r, ..., for predicates. Further along, particularly when we discuss Prolog we will often use complete lower-case words for function symbols and predicate symbols, and words starting with a capital letter for variables.

The improper symbols are used in constructing well-formed formulas (the legal formulas of predicate logic) and also to force precedence amongst the connectives and quantifiers. Without parentheses the following is the default precedence.

$$\sim, \forall, \exists$$
$$\vee$$
$$\wedge$$
$$\rightarrow, \leftrightarrow$$

Now, the syntax of predicate logic is the rules for constructing a first order language.

Definition. 3.2 *A* term *is defined inductively as follows*

1. *A variable is a term*
2. *A constant is a term*
3. *If f is a function of arity n and $t_1, t_2, \ldots, t_n$ are terms then $f(t_1, t_2, \ldots, t_n)$ is a term*

Definition. 3.3 *A term is called* ground *if it contains no variables.*

A first order language is defined to be the set of all well-formed formulas constructed from a specific alphabet according to the following rules.

Definition. 3.4 *A* well-formed formula *is defined inductively as follows:*

1. *If p is a predicate of arity n and t_1, t_2, ... , t_n are terms then $p(t_1, t_2, \ldots, t_n)$ is an* atomic formula *(or* atom*)*
2. *If P and Q are formulas then so are $(\sim P)$, $(P \vee Q)$, and $(P \wedge Q)$.*
3. *If P is a formula and x a variable then $(\forall x P)$ and $(\exists x P)$ are formulas*

The brackets are not necessary, and when there is no confusion as to precedence they will be freely dropped.

Example. 3.5 Suppose that the language has constants *bill*, *mary*, *jennifer*, *jack* and a binary predicate called *married*. Then the following expressions are well formed formulas

married(jennifer,bill)
married(mary,jack)

However, the following expressions are also equally valid well-formed formulas:

married(jack,bill)
married(jennifer,jennifer)
$\forall x$ *married(jennifer,x)*

■

Notice that at this stage we have simply discussed the mechanical rules for constructing well-formed formulas — the syntax of predicate logic. In some sense the syntax of a language is merely its "grammar". We have simply given the formal rules for constructing "grammatical" sentences in the language but have as yet made no attempt to attach any meaning to these sentences.

In order to do this, we must consider the semantics of predicate logic, which will be done formally in the next section. Informally though, the symbols $\sim$, $\wedge$ and $\vee$ will mean "not", "and" and "or" respectively, and $\forall x$ and $\exists x$ will mean "for all x" and "there exists an x" respectively. To formally discuss the semantics of these symbols we will require an extended concept of interpretation. First we complete our discussion of syntax with some additional terminology.

As in propositional logic, we consider special types of well-formed formula.

Definition. 3.6 *A* literal *is an atom A or its negation $\sim A$ (and they are called respectively a* positive literal *and a* negative literal*).*

Definition. 3.7 *A* clause *is a formula of the form*

$$\forall x_1 \forall x_2 \ldots \forall x_n A_1 \vee A_2 \vee \ldots A_k \vee \sim B_1 \vee \sim B_2 \vee \ldots \vee \sim B_m$$

where $\{x_1, x_2, \ldots, x_n\}$ *are all the variables appearing in the positive literals* A_1, A_2, ..., A_k, B_1, B_2, ..., B_m.

We use a special clausal notation for clauses as follows

Definition. 3.8 *The clause*

$$\forall x_1 \forall x_2 \ldots \forall x_n A_1 \vee A_2 \vee \ldots A_k \vee \sim B_1 \vee \sim B_2 \vee \ldots \vee \sim B_m$$

is denoted

$$A_1, A_2, \ldots, A_k \leftarrow B_1, B_2, \ldots B_m$$

Exactly as in propositional logic, a *Horn clause* is one with at most one positive literal. Now we are in the position to define a logic program.

Definition. 3.9 *A* program clause *is a clause with precisely one positive literal, that is, a clause of the form* $A \leftarrow B_1, B_2, \ldots, B_k$.

Definition. 3.10 *A* goal clause *is a clause with no positive literals, that is, a clause of the form* $\leftarrow B_1, B_2, \ldots, B_k$.

Definition. 3.11 *A* logic program *is a finite set of program clauses.*

3.2 Semantics of Predicate Logic

As with propositional logic, the formulas in a first order language are merely formal strings of symbols. In order to be able to model real world situations we must attach meanings to the symbols. In propositional logic this was easy because each propositional symbol merely represented a simple statement of fact and an interpretation merely consisted of assigning the value *true* or *false* to each propositional symbol and the truth values of the compound propositions were completely determined. However predicate logic allows the introduction of variables whose meaning must be given, functions whose action must be described, and finally predicates, which are functions of various arities with codomain $\{T, F\}$. Therefore we need a more sophisticated interpretation than the simple truth assignment of propositional logic.

Definition. 3.12 *Let* D *be a non-empty set, called the* domain. *Then an* interpretation *of a first order language consists of the following*

1. *For each constant symbol* a *the assignment of an element* a' *of* D
2. *For each* $n-$*ary function symbol* f *the assignment of a mapping* $f' : D^n \to D$.
3. *For each* $n-$*ary predicate symbol* p *the assignment of a mapping* $p' : D^n \to \{T, F\}$.

Therefore an interpretation gives some meaning to most of the symbols in a first order language. The set D is the set of objects to which everything in the language refers—the domain of discourse. Under a specific interpretation, constant symbols refer to specified elements of the domain D and the function symbols refer to actual functions on the set D. Each predicate symbol now represents an actual real-world relationship about the objects in the set D.

Now the definitions of term and atomic formula make more sense. A term is composed of variables, constants and functions applied to terms and hence refers to a specic element of the domain D. That is, a term refers to one of the things we are trying to discuss. An atomic formula is then a statement about a relationship between terms; in fact it is a simple factual statement about some of the objects in D. Therefore each well-formed formulas is actually an assertion about relationships held by the elements of D and as such has a truth value of *true* or *false* according to whether the elements of D actually obey the stated relationship or do not.

Example. 3.13 Now let us return to the example we used before and see how we could construct a sensible interpretation. Recall that our first-order language had only the constant symbols $\{jack, bill, jennifer, mary\}$ and the predicate symbol *married*. Now for an interpretation we need a domain: let us choose the four people Jack, Bill, Jennifer and Mary to be the domain. Each constant symbol must represent an element of the domain. Let the symbol *jack* refer to the person Jack (in our earlier notation *jack'* = Jack), *bill* refer to the person Bill, *jennifer* refer to the person Jennifer and *mary* refer to the person Mary. Then we must find an interpretation for the predicate symbol *married* — we say that *married(x,y)* will be true provided that the people referred to by the symbols x and y are actually married. Furthermore assume that in real life, Jack is married to Mary, and Jennifer is married to Bill. Then via this interpretation we can actually ascribe a meaning to particular well-formed formulas.

married(jennifer,bill) Jennifer is married to Bill

married(mary,jack) Mary is married to Jack

married(jack,bill) Jack is married to Bill

married(jennifer,jennifer) Jennifer is married to Jennifer

$\forall x$ *married(jennifer,x)* For all x, Jennifer is married to x.

In natural language some of those statements are true and others are false. In order for our logic to be useful, the rules for assigning truth values to formulas should be consistent with our natural reasoning. ■

There is one specific problem to do with assigning truth values to well-formed formulas and that is that some formulas have *free variables*. For example, consider the formula $p(x)$. Is this true under an interpretation $\mathcal{I}$ or not? We cannot assign a truth value to this formula because the variable x does not at this stage refer to any specific element of D. In fact it makes some sense to regard this as not having any truth value at all. However for technical reasons it is formally better to introduce an artificial construct called a *variable assignment* to deal with such formulas.

Definition. 3.14 *A* variable assignment *with respect to an interpretation is an assignment of an element of D to each variable of the language.*

Once a variable assignment has been given, then any term t represents an element t' of D. This is because each variable now represents some specific element of D. For every function f involved in the term, there is a corresponding function f' defined under the interpretation. Applying f' to the arguments given by the variable assignment yields the element of D that corresponds to t'.

Therefore given an interpretation and a variable assignment, we can now give truth values to any formula in the language.

Definition. 3.15 *Let $\mathcal{I}$ be an interpretation of the first order language L with domain D and let V be a variable assignment. Then any formula in L can be given a truth value as follows:*

1. *If the formula is an atom $p(t_1, t_2, \ldots, t_n)$ then the truth value is $p'(t'_1, t'_2, \ldots, t'_n)$*
2. *If the formula is of the form $\sim P$, $P \vee Q$, $P \wedge Q$, $P \rightarrow Q$ or $P \leftrightarrow Q$ then the truth value depends on the truth values of P and Q according to the following table*

P	Q	$\sim P$	$P \vee Q$	$P \wedge Q$	$P \rightarrow Q$	$P \leftrightarrow Q$
T	T	F	T	T	T	T
T	F	F	T	F	F	F
F	T	T	T	F	T	F
F	F	T	F	F	T	T

3. *If the formula is of the form $\exists xP$, then the truth value of the formula is true if there exists some element $d \in D$ such that P is true when all occurrences of the variable x are assigned the value d (all other variables retain their assignments under V)*

4. *If the formula is of the form $\forall xP$, then the truth value of the formula is true, if for every $d \in D$, P is true when all occurrences of the variable x are assigned the value d (all other variables retain their assignments under V)*

In general the truth or otherwise of a formula depends on the variable assignment, but in certain circumstances it does not.

Definition. 3.16 *In a formula of the form $\exists xP$ or $\forall xP$, any occurrences of the variable x in P are said to be* bound. *A* closed formula *is one in which every variable is bound.*

The truth value of a closed formula does not depend on the variable assignment, merely the interpretation. In all the following we will only be interested in closed formulas and therefore the artificial device of a variable assignment will not reappear.

Example. 3.17 Let us return to our example. We see that the truth values of the formulas *married(mary,jack)* and *married(jennifer,bill)* are both true, simply by the interpretation of *married*. Similarly the formula *married(jack,bill)* is false simply by the interpretation. Now the formula $\forall x$ *married(jennifer,x)* is a closed formula, which is true provided it is true for every replacement of the variable x. However considering $x =$ *jack* shows that the formula is false. Observe that the formula *married(y,bill)* is not a closed formula and its truth value depends on a particular variable assignment. ■

Notice however that any formula can be converted into a closed formula by quantifying all the variables it contains.

Example. 3.18 The formula $\forall xp(x, y, z)$ is not a closed formula, because y and z are unbound. However quantifying the variables y and z (using either the universal or existential quantifier) yields a closed formula, say $\forall y \exists z \forall x p(x, y, z)$. ■

Let us consider another example of interpretations of a first-order language, this time one involving function symbols.

Example. 3.19 Consider the first order language with constant symbols a and b, function symbol *s/1* and predicate symbols *p/2*, *q/1* and *r/2*. Consider the following interpretation of this language.

> The domain D is the set of integers
> The function symbol s represents the successor function $s : x \to x + 1$
> The constants a and b are assigned to 0 and 1 respectively
> The predicate symbol p represents the binary predicate $>$ (that is, *p(x,y)* is true if $x > y$)
> The predicate symbol q represents the unary predicate "is greater than zero" (so, $q(x)$ is true if $x > 0$)
> The predicate symbol r represents that binary predicate "divides" (so $r(x, y)$ is true if x divides y).

Then consider each of the following formulas

$\forall x \exists y$ *p(x,y)* For any integer x, there is an integer y such that y is less than x. This is a true statement about integers, and a few moments thought shows that its truth value in predicate logic is true.

$\exists x \forall y$ *p(x,y)* There is an integer x such that x is greater than every integer y — this is saying that there is a greatest integer, which is false.

p(s(a),b) a represents the integer 0, so $s(a)$ represents the integer 1. Thus this statement is asserting that 1 is greater than 1, which is false.

$\forall x$ *(p(x,a)* $\rightarrow$*q(x))* The integer x is greater than the integer 0 if $x > 0$. This is true because both sides are true at precisely the same times.

■

3.3 Models and Validity

It is important to realise that a given language has many different interpretations. Some interpretations are relatively natural, like the arithmetical example in the previous section where the predicates could be described very simply ("less than" etc). However recall that a predicate simply asserts that some relationship holds between elements of D and there is no reason why such a predicate should not be very complicated, very artificial and arbitrary.

Given any set of closed formulas S the truth value of the formulas depends only on the interpretation. It may be the case that all the formulas in S are true under a given interpretation $\mathcal{I}$. In this case the interpretation $\mathcal{I}$ is called a *model* for S.

Definition. 3.20 *Let S be a set of closed formulas of a first order language. Then an interpretation $\mathcal{I}$ is called a* model *for S if all the formulas in S are true under $\mathcal{I}$.*

We can reverse the situation somewhat and supose that we are given a set S of closed formulas and asked to *find* an interpretation that makes them true (that is, to find a model for S). Sometimes it may occur that every possible interpretation is a model for S and sometimes every possible interpretation does not satisfy S. There is special terminology for these two situations.

Definition. 3.21 *Let S be a set of closed formulas of a first order language. Then S is called* satisfiable *if there is some interpretation that is a model for S, and* unsatisfiable *if there is no interpretation that is a model for S. If every interpretation is a model for S, then S is called* valid.

Sometimes it is quite easy to see whether or not a formula is satisfiable or valid, but on other occasions it can be quite challenging.

Consider the formula $\exists x\ p(x)$. The status of this formula is very easy to determine, because it is very easy to find an interpretation that makes it true and a different interpretation that makes it false. For instance an interpretation that is a model for the formula is given by $D = \{0\}$, $p'(0) = T$ and an interpretation that is not a model is given by $D = \{0\}$, $p'(0) = F$.

In general, any atomic formula can be satisfied or otherwise by a simple choice of predicate. More interesting formulas are those that involve implications of some sort. Consider the following formula:

$$\forall x\ p(x) \leftrightarrow \exists x\ p(x)$$

This is again very simple to analyse, in that it is clear that the left hand side is asserting that $p(x)$ is always true, whereas the right hand side is simply asserting that $p(x)$ is true for some value of x. Therefore it is easy to construct a model for this formula (simply make p always true), but it is also easy to find an interpretation that is not a model, by choosing p to be a predicate that is sometimes true but not always (for example, $D = \{0, 1\}$, $p'(0) = T$, $p'(1) = F$.)

A more complicated example is given by the following question: Is this formula valid?

$$\forall x\ \forall y\ \exists z\ (p(x, z) \wedge p(z, y)) \rightarrow \exists x\ p(x, x)$$

In order to show that the formula is *not* valid, it is sufficient to give a single interpretation that makes the formula false. However in order to show that the formula *is* valid, it is necessary to give some sort of proof (because implicitly all possible interpretations must be considered.

To solve problems of this kind, there are no particular step-by-step methods that are guaranteed to succeed; to a large extent it depends on how cunningly the predicate can be designed.

As it is far easier to show that the formula is not valid, by finding a specific interpretation, this should be attempted first. If all attempts at finding an interpretation that is not a model fail, then there must be a reason, and finding that reason can often be converted into a proof that all interpretations must be models.

In this case, we see that in order to make the formula false we must find some interpretation that makes the left hand side $\forall x\ \forall y\ \exists z\ (p(x,z) \wedge p(z,y))$ true, while making the right hand side false. As a first guess we can see that defining $p(x,y)$ to be true always will make the left-hand side true, but unfortunately it also makes the right hand side true. Considering the right-hand side we see that however we choose p it needs to be the case that $p(x,x)$ is never true. However $p(x,y)$ must be true a lot of the time because the left-hand side shows us that for any x and y we can find a z with the property that both $p(x,z)$ and $p(z,y)$ are true. Taking this to the extreme we consider defining p to be a predicate which is always true *except* when its values are the same (in other words, defining p to be "not equals"). This clearly works, and so we finish up by giving the specific solution: $D = \mathbf{Z}$, and $p(x,y)$ is true if $x \neq y$. This interpretation makes the formula false and hence the formula is not valid.

Solving such problems often requires one to think of how to define a predicate; the question remains as to how to think of an undefined predicate. There are a few standard tricks which may prove useful. A unary predicate (that is, a predicate $p(x)$ with one argument) can often be usefully thought of best as simply a subset of D. Namely, $p(x)$ is true for some elements of D but false for some other elements of D. If we separate out those elements x for which $p(x)$ is true, then we have subset of D which is easy to conceptualize. The realisation that a unary predicate p can actually be *defined* merely by giving a subset of D frees one from trying to give a "formula" or "recipe" for p. A binary predicate $p(x,y)$ can most usefully be thought of as a graph. Given any binary predicate, we can define a graph G where the vertices of G are the domain D and there is an edge from the vertex x to y if $p(x,y)$ is true. Then any binary predicate defines a graph and any graph defines a binary predicate. This allows us to get a more specific handle on what a predicate actually "looks like".

In our previous example we were looking for a predicate $p/2$ that satisfied the condition

$$\forall x\ \forall y\ \exists z\ (p(x,z) \wedge p(z,y))$$

but not the condition

$$\exists x\ p(x,x)$$

Expressing these as graph properties we see that the first condition states that there must be a path of length 2 between any two vertices, and the second condition states that there is a loop on some vertex. It is then easy to find a graph that satisfies the first condition but not the second. Once a solution has been found, then it can be expressed in the most convenient form — if necessary, the graph itself can be used as the definition of the predicate.

Ternary and higher predicates are very much more difficult to think about because the corresponding analogue of a graph (which is called a k-uniform hypergraph) is less easy to work with.

3.4 Inference and Resolution

As with propositional logic, our aim is to start with a set of formulas that we accept as true statements, and then to deduce new formulas that are logical consequences of the original set of propositions. If we can manage to give purely mechanical rules for the deduction procedure, then an automated system will be able to produce new facts from our original database of facts.

Definition. 3.22 *Let S be a set of closed formulas. Then the closed formula F is called a* logical consequence *of S if every model for S is also a model for F.*

It is instructive to compare this definition with the same one for propositional logic. In particular notice that F is a logical consequence of S only if *every* model for S is also a model for F. Logical implication does not depend on what particular situation is being modelled, simply on the form of

the relationships and facts. This seems to imply that to determine whether something is a logical consequence of a set of formulas requires one to consider infinitely many models. Looking carefully at the definition, we see that if S is itself unsatisfiable, then any formula is a logical consequence of S.

Example. 3.23 Consider the set S of clauses

p(x) ←q(x)
q(a)

Now in any model for S, the first clause must be true for all x. Therefore, in particular the clause *p(a) ←q(a)* must be true — this is called a *ground instance* of the clause. Then because *q(a)* is true, it must also be the case that *p(a)* is true. Therefore the formula *p(a)* is a *logical consequence* of S. ■

The two important points of note about this example are how it parallels our own reasoning process, and the fact that the "meanings" of p and q played no role in the chain of reasoning.

Theorem. 3.24 *Let S be a set of closed formulas. Then F is a logical consequence of S if and only if $S \cup \{ \sim F\}$ is unsatisfiable.*

PROOF. First suppose that $S \cup \{ \sim F\}$ is not satisfiable, and let $\mathcal{I}$ be a model for S. Then it must be the case that $\sim F$ is false, and hence that F is true. Therefore $\mathcal{I}$ is a model for F as well. Conversely, suppose that F is a logical consequence of S. Then consider a model for S. As all models for S are models for F we see that F is true, so $\sim F$ is false. Thus $S \cup \{ \sim F\}$ is not satisfiable. ■

As in propositional logic, the main inference rule we will use is resolution. For the moment we shall assume that the clauses are ground clauses (that is, have no variables); later we give a more general definition.

Definition. 3.25 (Resolution for ground clauses) *Let F_1 be the ground clause*

$$A_1, A_2, \ldots, A_m \leftarrow B_1, B_2, \ldots, L, \ldots, B_n$$

and let F_2 be the ground clause

$$C_1, C_2, \ldots, L, \ldots, C_r \leftarrow D_1, D_2, \ldots, D_s$$

The resolvent *of F_1 and F_2 is the clause*

$$A_1, A_2, \ldots, A_m, C_1, C_2, \ldots, C_r \leftarrow B_1, B_2, \ldots, B_n, D_1, D_2, \ldots, D_s$$

obtained by deleting the literal L that occurred negatively in F_1 and positively in F_2 and merging all the other literals into a single clause.

An identical proof to that for propositional logic shows that the resolvent of two clauses is a logical consequence of those two clauses, and hence resolution is a *sound* inference rule.

As in propositional logic, proof by refutation is the main technique for applying the resolution rule to prove logical implication. Recall that refutation involves negating the formula to be proved, adding it to the set of clauses and then showing that this augmented set of clauses is unsatisfiable by repeatedly finding resolvents until the empty clause (contradiction) is obtained. Let us return to our example and demonstrate how refutation may be used.

Example. 3.26 Given the set of clauses S

p(x) ←q(x)
q(a)

we shall try to show that *p(a)* is a logical consequence of S. First we negate *p(a)* and add it to the set of clauses and then use resolution. We take a ground instance of the clause *p(x)* $\leftarrow$*q(x)*, namely *p(a)* $\leftarrow$*q(a)* and resolve it with $\sim$*p(a)* to obtain the clause $\leftarrow$*q(a)* and then resolve that clause with *q(a)* to obtain $\square$. ■

Notice that there are several differences between this application of refutation and the ones in propositional logic. One important feature is that when we used the clause $p(x) \leftarrow q(x)$, which we recall is shorthand notation for $\forall x\ (p(x) \vee \sim q(x))$ we actually used the clause with a specific value substituted for x. That is, we used a *ground instance* of the clause. As there are in general infinitely many ground instances of a clause, this raises the question of exactly which ground instance should be used.

In the example above, we have demonstrated how logic programming can prove that a given clause $p(a)$ is a logical consequence of the program. However this is only useful if we already know what the answer is and want the computer to verify it for us. It is more likely that we actually want to find out the value of a variable x such that *p(x)* is a logical consequence of the program. We would like to be able to specify the query with a variable and have the computer respond that $x = a$ is a suitable choice. In other words we require the computer to actually *find* the logical consequence, rather than just verify it.

Let us summarise the overall procedure of logic programming. We express the facts about some situation as a set of Horn clauses. These form the logic program P. We wish to find logical consequences of the program P. The formulas to be proved to be logical consequences of P have the form

$$\exists y_1 \ldots \exists y_r (B_1 \wedge B_2 \wedge \ \ldots \ \wedge B_n)$$

where we wish to actually find the values of y_1, y_2, ..., y_r that make this formula true. Negating this existential formula results in all the existential quantifiers (the $\exists$ s) being converted into universal quantifiers ($\forall$ s) and therefore results in a goal clause G. The goal clause G has the form

$$\leftarrow B_1, B_2, \ldots, B_n$$

The system then takes the program P and the goal clause G and proceeds to demonstrate that $P \cup \{G\}$ is unsatisfiable, by finding particular values for y_1, y_2, ..., y_n that make $P \cup \{G\}$ unsatisfiable. Then the theorem on logical consequences shows that $\sim G$, which is the original formula, is a logical consequence of P, and that the particular values of y_1, y_2, ..., y_n that have been found is one of the solutions we are seeking.

Example. 3.27 We demonstrate these ideas with the simple example above. Remember that we wish to find a specific x such that $p(x)$ is a logical consequence of the program P. Therefore we are asking the question "Is $\exists x\ p(x)$ a logical consequence of P". This statement is then negated to become the goal clause $G = \forall x\ \sim p(x)$, which can be written $\leftarrow p(x)$. After some computation we find that the value $x = a$ makes $P \cup \{G\}$ unsatisfiable, which proves that $p(a)$ is a logical consequence of P. ■

3.5 An example in Prolog

Consider now the difference when we ask Prolog to verify a logical consequence and actually find a logical consequence. Continuing with the simple example above, the logic program becomes the following Prolog program:

```
p(X) :- q(X).
q(a).
```

If we simply wish to verify that $p(a)$ is a logical consequence of this program we do so in the natural fashion.

```
| ?- p(a).

yes
```

However, if we actually want to *find* the value of x that works then we can phrase the same question with a variable as the argument (remembering of course that all variables in Prolog start with a capital letter).

```
| ?- p(X).

X = a
```

Let us examine what happened in that calculation: The program P consisted of the two original clauses. We wished to know whether the statement

$$\exists X p(X)$$

was true or not. Therefore we negated it to the goal clause G

$$\forall X \sim p(X)$$

and asked the system to process that. The system responded that the program together with the statement

$$\forall X \sim p(X)$$

was unsatisfiable, because choosing the value $X = a$ made $P \cup \{ \sim p(a)\}$ unsatisfiable. Therefore $X = a$ is a solution to the original problem. We can think of this as a kind of double negative situation. The user adds the negation of what he wants to the database, and then Prolog demonstrates that this new database is unsatisfiable. This is equivalent to the mathematical procedure of proof by contradiction.

4 Prolog

In this section we leave the theory for a short while and consider the basics of the Prolog programming language. Some of the concepts involved have not yet been formally explained, but nevertheless it is useful to actually start programming in Prolog.

4.1 The standard beginning example

The standard example for a beginning Prolog program is that of relationships between people. We consider predicates such as *parent*, *male*, *female* etc. Let us start with some simple facts.

```
parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(joe,jim).
parent(pat,jim).

female(pam).
male(tom).
male(bob).
female(liz).
female(ann).
female(pat).
male(joe).
male(jim).
```

When we write these predicates down we are thinking of some "intended meaning" — we are using *parent(pam,bob)* to indicate that Pam is the parent of Bob. Prolog of course understands none of this. To Prolog there is a binary predicate symbol *parent* which is true when its arguments are *pam* and *bob*.

We can now ask some simple questions, simply based on these facts.

```
| ?- male(tom).

yes

| ?- male(X).

X = tom
```

Notice that there is more than one solution to *male(X)*, namely $X = tom$, $X = bob$ and $X = jim$. Some versions of Prolog will wait for a prompt after finding one solution and others will find all solutions. In CProlog, the system waits for a prompt — if the user responds with a semi-colon, then another solution is found, and if the user responds with a full-stop (period) the system exits. In Prolog2 the system actually prompts the user to enter a 'y' or 'n' to indicate whether more solutions are wanted or not.

Now we may wish to express some other relationships in terms of the ones we already have. This is done in a very natural way. For example A is a child of B provided that B is a parent of A. This can be written in Prolog directly:

```
child(X,Y) :- parent(Y,X).

| ?- child(bob,A).

A = pam
```

We can distinguish between mothers and fathers.

```
mother(X,Y) :- parent(X,Y), female(X).
father(X,Y) :- parent(X,Y), male(X).
```

and go to more distant relations

```
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
```

Let us trace through a calculation involving this program using our (incomplete) knowledge so far of how Prolog is going to perform the calculation using resolution. Suppose we ask the question "Who is Jim's mother?" We actually negate the statement $\exists X\, mother(X, jim)$ to produce the goal clause $\leftarrow mother(X,jim)$, and then feed that to Prolog.

```
| ?- mother(X,jim)
```

Now Prolog works by taking the clause that has just been given—in this case

$$\leftarrow mother(X, jim)$$

and repeatedly tries all possible ways of resolving it in order to produce the empty clause. We shall discuss later the consequences of the way Prolog tries to perform the resolution — for now we shall simply describe what it does. Prolog takes first literal in the goal, in this case the only one is

$$mother(X, jim)$$

and tries to resolve that with the head of one of the program clauses. It looks down the database and finds the clause

$$mother(X, Y) \leftarrow parent(X, Y), female(X)$$

Now it tries to find an instance of the program clause so that the literals in the goal clause and program clause match — this process is called *unification* and it is absolutely central to the operation of Prolog. We shall deal with this extensively in the next chapter. Intuitively we can see that

$$mother(X, jim) \leftarrow parent(X, jim), female(X).$$

is an instance that matches the goal. Therefore we can resolve with this clause and obtain the resolvent

$$\leftarrow parent(X, jim), female(X)$$

Now Prolog proceeds with this as the new goal. Once again the first literal is chosen

$$parent(X, jim)$$

and Prolog tries to match that with the head of one of the program clauses. In this case the first 5 clauses do not match, but finally we obtain a match and we resolve with the clause

$$parent(joe, jim)$$

obtaining the new goal clause

$$\leftarrow female(joe)$$

Prolog then searches through the database, but cannot resolve this goal clause with any of the program clauses. Therefore Prolog has not yet succeeded in showing that this set of clauses is unsatisfiable. However Prolog made certain choices previously about which clause to resolve with, and which particular instance of that clause to use when resolving. As there is no reason to believe that the first choices will necessarily lead to success, Prolog performs *backtracking* whereby it returns to the previous goal and tries to resolve it in a different way. So, having failed, Prolog returns to the goal

$$\leftarrow parent(X, jim), female(X)$$

and tries to resolve it with a different program clause. In this case there is the alternative choice of

$$parent(pat, jim)$$

and the resolution gives a new goal of

$$\leftarrow female(pat)$$

which then resolves directly with the program clause

$$female(pat)$$

to produce the empty clause. Thus Prolog has proved that *mother(pat,jim)* is a logical consequence of the program, and therefore that $X = pat$ is a solution to the original goal.

To finish this example we examine an example of recursive programming in Prolog. Consider the predicate *ancestor(X,Y)* which is to be true if X is an ancestor of Y, that is, if X is a parent of Y, or a grandparent, or a great-grandparent and so on. Now for any one specific relationship we can devise a predicate

```
greatgreatgrandparent(X,Y) :- parent(X,X1), parent(X1,X2),
      parent(X2,X3), parent(X3,Y).
```

but it seems difficult at first to see how to devise one predicate that will cope with any level of ancestry. The solution to this is found in the concept of *recursion*. A recursive definition of a concept involves defining something in terms of itself. In our example we could use

```
ancestor(X,Z) :- parent(X,Z).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

Here we are saying that X is an ancestor of Z if either X is a parent of Z, or if X is the parent of an ancestor of Z. Notice that the recursive definition consists of two parts — a *boundary condition* and a *recursive step*. The recursive step is a circular condition in which *ancestor* is defined in terms of *ancestor* which is defined in terms of *ancestor* and so on. However the arguments change at each step and the terms get closer to the boundary condition. Eventually the potentially infinite cycle is broken when the boundary case is reached.

We can try our definition on the program so far — in CProlog we respond with a semicolon to each answer to get Prolog to search for any further answers.

```
| ?- ancestor(X,pat)

X = bob ;

X = pam ;

X = tom ;

no
```

We see that Prolog correctly found all the three ancestors of Pat — well all that the database knew about anyway. Briefly examine how Prolog found those answers. Starting with the goal clause

$$\leftarrow ancestor(X, pat)$$

it tried to match that with the first clause involving ancestor in the database, and resolve. A match was found with

$$ancestor(X, pat) \leftarrow parent(X, pat)$$

so after resolution the goal was replaced with

$$\leftarrow parent(X, pat)$$

This goal succeeded with solution $X = bob$ so that was the first answer produced. Then we asked Prolog to try and resatisfy the goal so it continued the search. There were no more solutions to *parent(X,pat)* so backtracking occurred and Prolog tried to resatisfy the original goal. This time it matched with the second clause in the definition of ancestor, the particular instance being

$$ancestor(X, pat) \leftarrow parent(X, Y), ancestor(Y, pat)$$

so after resolution the goal becomes

$$\leftarrow parent(X, Y), ancestor(Y, pat)$$

Prolog chooses the first literal, in this case *parent(X,Y)* and tries to satisfy it, and it succeeds with $X = pam$, $Y = bob$. Then after resolution the new goal is

$$\leftarrow ancestor(bob, pat)$$

which succeeds on the first clause of ancestor. Therefore we have found $X = pam$.

We shall need some more terminology. Consider the goal $\leftarrow$*parent(X,Y)*. This contains two variables which have not yet been given values — such variables are called *uninstantiated*. On the other hand consider the goal $\leftarrow$*parent(X,Y), ancestor(Y,pat)*. This goal was resolved with the ground clause *parent(pam,bob)*. In order to resolve the goal with that clause the variable X had to be *instantiated* to the value *pam* and the variable Y had to be instantiated to the value *bob*. In addition, the variable Y was instantiated *throughout the goal clause* so that the resolvent of those two clauses was *ancestor(bob,pat)*. However on backtracking, when Prolog tries to *resatisfy* the goal $\leftarrow$*parent(X,Y)* the variables become uninstantiated again, until a new match is found. We can regard an instantiated variable as having a value assigned to it until backtracking occurs, when the value may be discarded.

4.2 Prolog is not logical

Prolog is an attempt to use logic as a programming logic. However it only approximates the ideal, and we have already seen an example of where the theory of logic programming and the practice diverge. Consider our ancestor example of the previous section. Suppose we had written it

```
ancestor(X,Z) :- ancestor(Y,Z), parent(X,Y).
ancestor(X,Z) :- parent(X,Z).
```

Logically, this is exactly equivalent to the original statement, because all we have done is to re-order some of the clauses. Nevertheless if we try to use this in Prolog we get

```
| ?- ancestor(X,pat).

! Out of local stack during execution

[ execution aborted ]
```

The problem occurs because the first match for *ancestor* is another call to *ancestor*. Therefore the recursive definition never meets the boundary case, and cycles for ever (or until the stack of the computer is exhausted). This is a consequence of the fact that Prolog always chooses the left-hand literal first, and always chooses the first program clause in the database. Therefore it is important in Prolog to know what rule the system uses to make your programs work — this is already a significant divergence from the ideal of programming in logic.

4.3 Arithmetic in pure Prolog

Arithmetic in Prolog is one of the first places where theoretical logic programming diverges from practical programming. The main problem is that predicate logic does not directly have any concept representing a number. We cannot use constants to represent different numbers because then we would require an infinite number of clauses to represent the simplest properties of addition, and multiplication and so on.

Therefore to represent integers we shall use the same idea as used in the foundations of mathematics. We shall represent numbers by certain terms, those being *zero, s(zero), s(s(zero)), s(s(s(zero)))* .... Obviously we will interpret *zero* to mean 0, and *s* as the *successor function*, thereby giving us that the term *s(s(s(zero)))* represents the number 3 and so on.

In order to define propositions relating to arithmetic it is necessary for us to be able to identify numbers. Consider the following predicate:

```
isnumber(zero).
isnumber(s(X)) :- isnumber(X).
```

This predicate is a precise declarative description of what it means to be a number. The two clauses represent the following facts

0 is a number
$x + 1$ is a number if x is a number

Now the aim of logic programming is that such a declarative description is sufficient to actually be a program. By that we mean that simply the logical definition of number as given above should contain enough information for the system to decide how to actually perform the calculation. Therefore we should simply be able to write suitably coded versions of these statements directly, and the computer should do the rest. Let us test our program to see if it achieves this aim.

```
| ?- isnumber(zero).

yes
```

```
| ?- isnumber(s(s(s(zero)))).

yes
| ?- isnumber(s(s(2))).

no
```

Notice that this program therefore identifies anything that is a term with s applied repeatedly to *zero*. It achieves this by immediately succeeding if the argument is actually *zero*, otherwise if the argument is $s(X)$, then it replaces the goal by *isnumber(X)*. Therefore we can imagine each step stripping one s from the argument until either *zero* is reached and the argument found to be a genuine number, or neither of the clauses match and the proof fails.

One of the features of Prolog, compared to a procedural programming language is that a logical description can often be used in many different ways. So for example we can try the following command

```
| ?- isnumber(X).

X = zero ;

X = s(zero) ;

X = s(s(zero)) ;

X = s(s(s(zero)))

yes
```

In this case we are asking for an X which is a number. Prolog returns first with $X = zero$, and then on being prompted for more answers with the semicolon, successively returns $X = s(zero)$, $X = s(s(zero))$ and so on. When enough answers have been obtained simply hitting the return key stops the search.

Let us now try and define a few more arithmetical operations. We shall start with the operation of $\leq$. Thinking of how to specify this logically we come up with the following ideas

0 is less than or equal to any other number
$x + 1 \leq y + 1$ if $x \leq y$

Simply translating these simple logical facts into Prolog gives us the desired program

```
lessthanequal(zero,X) :- isnumber(X).
lessthanequal(s(X),s(Y)) :- lessthanequal(X,Y).

| ?- lessthanequal(s(s(zero)),s(s(s(zero)))).

yes
| ?- lessthanequal(s(s(zero)),s(zero)).

no
```

Just as with the predicate for identifying numbers this predicate can be used in several different ways.

```
| ?- lessthanequal(s(zero),X).

X = s(zero) ;
```

```
X = s(s(zero))

yes
```

We can consider addition of two numbers. Now normally we regard addition as a procedural operation, that is, the instruction $z = x+y$ is normally taken as an instruction to take the arguments x and y, perform the operation + on them and assign the result to the variable z. But in logic we are only concerned with predicates, and therefore we must define a ternary predicate *plus(X,Y,Z)* which is to be true whenever Z represents the number obtained by adding X and Y.

```
plus(zero,X,X) :- isnumber(X).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
```

Now this one predicate can be used as a complete addition system with many different uses. It can confirm correct additions such as

```
| ?- plus(s(s(zero)),s(s(s(zero))),s(s(s(s(s(zero)))))).

yes
```

or perform additions such as

```
| ?- plus(s(s(zero)),s(s(s(zero))),X).

X = s(s(s(s(s(zero)))))
```

or subtractions

```
| ?- plus(s(s(zero)),X,s(s(s(s(s(zero)))))).

X = s(s(s(zero)))
```

or even finding all the pairs of numbers that add up to a certain number.

```
| ?-  plus(X,Y,s(s(s(zero)))).

X = zero
Y = s(s(s(zero))) ;

X = s(zero)
Y = s(s(zero)) ;

X = s(s(zero))
Y = s(zero) ;

X = s(s(s(zero)))
Y = zero ;

no
```

It is a characteristic of logic programming that programs written in pure Prolog will always be able to function in many ways. However this last example shows us that we must be very careful about writing our functions. For example if we were to rewrite the *plus* predicate as

```
badplus(zero,X,X) :- isnumber(X).
badplus(X,s(Y),s(Z)).
```

then it will not work at all. This is because although it reads almost the same in English it is not accurate enough because it does not allow the possibility of using *zero* as the second variable. to add *zero* as the second argument.

Now consider the following definition for multiplication. It is to be another ternary predicate.

```
times(zero,X,zero) :- isnumber(X).
times(s(X),Y,Z) :- times(X,Y,Q), plus(Y,Q,Z).
```

This will now successfully multiply.

```
| ?- times(s(s(zero)),s(s(zero)),X).

X = s(s(s(s(zero))))

yes
```

On the principle of multiple uses, let us see if it will also divide for us

```
| ?- times(s(s(zero)),X,s(s(s(s(zero))))).

X = s(s(zero))

yes
```

Fine if there is a divisor, but if not then we get stuck in an infinite loop, only ending when the stack is exhausted

```
| ?-  times(s(s(zero)),X,s(s(s(zero)))).

! Out of local stack during execution

[ execution aborted ]
```

We can continue in this fashion to produce pure Prolog programs for many other arithmetic functions. The tutorials contain many such examples.

4.4 Arithmetic in Prolog

Of course performing arithmetic in pure Prolog is tremendously inefficient. The representation of numbers by terms is itself very slow, and the functions we devised do not use any of the computers built in ability to perform fast arithmetic. Therefore practical programs use various non-logical constructs for arithmetic. Although we shall be forced to use these we will try to stick to pure Prolog for the all else.

The first of these constructs is the assignment operator `is`. The statement

```
N is X
```

forces X to be evaluated as a number and then assigned to N.

The usual arithmetic operators are also defined:

$+$	addition
$-$	subtraction]
$*$	multiplication
$/$	division
mod	remainder after exact division

They are invoked by using the `is` command

```
| ?- X is 3 + 2.

X = 5

| ?-  X is 17 * 23.

X = 391

| ?-  X is 17/89.

X = 0.191011
```

Notice that the assignment will fail unless the right hand side is an arithmetic expression that evaluates to a number. In particular because these operations are not logical constructs they *cannot* be used in multiple ways. Logically for example, subtraction is the same as addition by reordering the arguments. However we cannot say

```
| ?-  4 is 3 + X.

! Error in arithmetic expression: not a number
   (1) 1 Fail: 4 is 3+_0

no
```

because the right hand side cannot be evaluated, and even if it could then the `is` operator cannot assign a value to the number 4.

Also used in a non-logical way are the comparison operators

$X > Y$	X is greater than Y
$X < Y$	X is less than Y
$X >= Y$	X is greater than or equal to Y
$X =< Y$	X is less than or equal to Y
$X =:= Y$	X is equal to Y
$X = \backslash = Y$	X is not equal to Y

It is important to realise that these are not logical because they all invoke the arithmetic operations to evaluate expressions. For example consider equality. In logic the expression $X = 3$ is perfectly valid, and has the solution $X = 3$.

```
| ?- X = 3.

X = 3
```

whereas the arithmetic version tries to evaluate the term X and fails

```
| ?-  X =:= 3.

! Error in arithmetic expression: not a number
   (1) 1 Fail: _0=:=3

no
```

In short, practical arithmetic in Prolog is very far removed from the logical goal. However if used with care it does not introduce too many problems, and it is absolutely vital for any sort of efficiency.

4.5 Lists in Prolog

Having dealt with the numerical problems of Prolog, we turn to the fundamental unit of non-numerical programming, which is the list. Lists are represented in Prolog as compound terms composed with the . functor. A list is defined recursively to be a binary term with functor ., the first argument being the first element of the list and the second argument being the remainder of list.

Therefore consider the following list

$$.(a, .(b, .(c, .(d, []))))$$

The term $.(d, [])$ refers to a list with d as first element and the empty list for the remainder. Thus it is simply the list $[d]$. So, the term $.(c, (d, []))$ refers to a list with c as its first element and with $.(d, [])$ as the remainder of the list. Therefore it represents the list $[c, d]$. Clearly the whole expression therefore refers to the list $[a, b, c, d]$. Although it is important for us to realize that Prolog represents a list as the *head* and *tail* of the list being the arguments to the . functor, it is clearly cumbersome for us to use this notation. Therefore Prolog accepts two alternatives. Firstly we can simply specify the list as $[a, b, c, d]$ in the natural fashion. Secondly we can specify it as $[a \mid [b, c, d]]$, where the vertical bar separates the head of the list from the tail of the list. Remember though, whatever notation we use for a list, Prolog immediately converts to the functor notation for internal use.

There are three fundamental functions to be performed on lists

1. checking membership in a list
2. concatenating two lists
3. adding elements or deleting elements from lists

We start by checking membership. Clearly an object is a member of a list if it is equal to the head of the list, or if it is a member of the tail of a list. We shall use a binary predicate *member(X,L)* which is to be true if the element X is a member of the list L.

```
member(Head,[Head|Tail]).
member(X,[Head|Tail]) :- member(X,Tail).
```

Notice how we have used Prolog's unification algorithm itself to check whether an element is equal to the head of the list. Consider the following simple examples of its use:

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

yes
| ?- member(e,[a,b,c]).

no
```

In the first example the first clause matched, and therefore the membership test was immediately successful, whilst in the second the head of the list was repeatedly deleted until the list was empty. In general the head of the list is repeatedly deleted until it matches with X and the clause succeeds, or the list is emptied and the clause fails.

Of course we can use this predicate in many other ways — to *find* elements of a list.

```
| ?-  member(X,[a,b,c]).

X = a ;

X = b ;

X = c ;

no
```

This use of *member* is extremely important for many "generate-and -test" algorithms, as we shall see later.

Stranger things happen if we try and use this to find lists for which a is a member.

```
| ?-  member(a,X).

X = [a|_6] ;

X = [_5,a|_10] ;

X = [_5,_9,a|_14]
```

Although this seems strange it is actually correct. The tokens that start with the underscore are variables for which Prolog has not been given a name, so internal names are chosen. Therefore this is saying that a is a member of the list $[a \mid \_6]$ where _6 is anything, and this is perfectly true. There are of course infinitely many lists of which a is a member.

Now consider concatenating two lists. We use the following predicate *concat(L1,L2,L3)* which is to be true whenever $L3$ is the list obtained by concatenating $L1$ and $L2$. We will need two clauses again. We use the following reasoning to derive the necessary clauses:

1. If $L1$ is empty then $L3 = L2$
2. If $L1$ is not empty, then $L3$ is obtained by putting the head of $L1$ at the beginning of the list obtained by concatenating the tail of $L1$ with $L2$

Notice that as usual we have a boundary case, and then a recursive case in which the problem is expressed as a smaller case of the same problem.

```
concat([],L2,L2).
concat([Head|Tail],L2,[Head|L3]) :- concat(Tail,L2,L3).
```

This works exactly as one may expect in the normal situation

```
| ?- concat([a,b],X,[a,b,c,d]).

X = [c,d]
```

and as usual with declarative programming has many other uses also

```
| ?- concat(X,Y,[a,b,c,d]).

X = []
Y = [a,b,c,d] ;

X = [a]
Y = [b,c,d] ;

X = [a,b]
Y = [c,d] ;

X = [a,b,c]
Y = [d] ;

X = [a,b,c,d]
Y = [] ;

no
```

Finally we turn our attention to inserting or deleting elements from a list. We will consider the predicate *delete(X,L,L1)* which is to be true if $L1$ is the list L after one occurrence of the element X has been deleted. Deletion is simply performed by observing that if X is the head of L then the result is simply the tail of L, and otherwise we must perform the delete from the tail of L.

```
delete(Head,[Head|Tail],Tail).
delete(X,[Y|Tail],[Y|L1]) :- delete(X,Tail,L1).
```

This program successfully deletes all occurrences of a letter from a list.

```
| ?- delete(a,[a,b,c,d,a,b],X).

X = [b,c,d,a,b] ;

X = [a,b,c,d,b] ;

no
```

Inserting an element into a list is exactly the opposite of deleting an element. Therefore it is not even necessary to write a separate program, but merely use the *delete* predicate in a different way.

```
| ?- delete(a,X,[b,c,d]).

X = [a,b,c,d] ;

X = [b,a,c,d] ;

X = [b,c,a,d] ;

X = [b,c,d,a] ;

no
```

Our final example in this section shows that it can be useful for the sake of efficiency to think about several different ways of solving a problem. We consider the problem of reversing a list, that is *reverse(L1,L2)* will be true provided that $L2$ is the list $L1$ reversed.

```
reverse([],[]).
reverse([Head|Tail],L2) :- reverse(Tail,L1), concat(L1,[Head],L2).
```

This works as planned

```
| ?- reverse([a,b,c,d],X).

X = [d,c,b,a]
```

However it is fairly inefficient in that for each recursive call to *reverse* one must also make a call to the recursive predicate *concat*. We avoid this by introducing an additional data structure called an *accumulator*. The accumulator is an intermediate data-structure that will be used to store part of the reversed list as it is built up. So, the predicate *reverse2(L1,L2,L3)* will be true if $L3$ is the result of concatenating $L2$ with the reverse of $L1$.

```
reverse2([Head|Tail],Accumulator,L3) :- reverse2(Tail,[Head|Accumulator],L3).
reverse2([],Accumulator,Accumulator).
```

```
| ?- reverse2([a,b,c,d],[],X).

X = [d,c,b,a] ;

no
```

It is instructive to see how this works. Each call to *reverse2* is replaced by another call to *reverse2* but with one element removed from the head of the list to be reversed, and added to the head of the accumulator. Finally when the initial list is empty the accumulator contains precisely the reverse of the initial list. The second clause simply accounts for this case. Simple timings show that this is much faster because it is essentially linear in the length of the list.

4.6 Sorting in Prolog

Sorting is one of the primary benchmark activities of computer science. Many many algorithms have been devised for sorting a list; we will consider some of these implemented in Prolog. We will consider four different algorithms for sorting a list: naive sort, insertion sort, bubble sort and quicksort. Each program will use a predicate *order(X,Y)* which will succeed if $X \leq Y$. Therefore simply by changing this predicate we will be able to sort different types of things - numbers, or words or whatever.

The naive sort simply generates permutations of a list, and then checks it for being sorted. First we give the clauses for the predicate *sorted(L)*.

```
sorted([]).
sorted([X]).
sorted([X,Y|Tail]) :- order(X,Y), sorted([Y|Tail]).
```

Next we must include the predicates for generating a permutation of a list. We shall use the *delete* predicate from the last section and the observation that a permutation of a list is obtained by permuting the tail of the list, and then inserting the head in every possible position.

```
permutation([],[]).
permutation([Head|Tail],L) :-  permutation(Tail,L1), delete(Head,L,L1).
```

We can combine these into the simple predicate *naivesort(L1,L2)* as follows:

```
naivesort(L1,L2) :- permutation(L1,L2), sorted(L2).
```

As one can see this is a very easy specification of a sorting program. However it is hopelessly inefficient. The reason for this is that there are $n!$ permutations of n objects and this grows extremely quickly. For example $5! = 120$, so Prolog potentially has to check only 120 possibilities. However, $10! = 3628800$ and $25! = 15511210043330985984000000$ which is clearly impossible for Prolog to examine. Clearly the problem lies in the fact that an entire permutation is generated before we check that it is sorted, whereas it should be possible very early to determine that a partial list is not sorted and reject that possibility.

Insertion sort is probably the most natural technique for human beings to use manually. The head of the list is removed, the remainder is sorted and then the head is inserted into the appropriate position.

```
insort([],[]).
insort([Head|Tail],L) :- insort(Tail,L1), insert(Head,L1,L).
```

and the inserting is done by the following predicate

```
insert(X,[],[X]).
insert(X,[Y|Tail],[Y|L1]) :- order(Y,X), insert(X,Tail,L1).
insert(X,[Y|Tail],[X,Y|Tail]) :- order(X,Y).
```

This sorting program sorts lists as expected, with one slightly strange feature.

```
| ?- insort([4,1,3,1,7,2],L).

L = [1,1,2,3,4,7] ;

L = [1,1,2,3,4,7] ;

no
```

Prolog produced two solutions to the above query. The reason for this was that there are two 1s in the list, and therefore in the *insert* predicate there is a point when both of the clauses are satisified. In other words the two 1s get inserted in both possible orders. This could be avoided by using a different definition of *order*.

Bubble-sort is another sorting technique that works by looking at pairs of adjacent elements and exchanging them if they are out of order. After all pairs have been looked at the resulting list will be sorted. The following program, which performs this is an example of bad programming style in Prolog.

```
bubsort(L,S) :-
concat(X,[A,B|Y],L),
order(B,A),
concat(X,[B,A|Y],M),
bubsort(M,S).
bubsort(L,L) :- sorted(L).
```

This program works somewhat

```
| ?- bubsort([5,4,3,2,1],X).

X = [1,2,3,4,5]
```

but has several major problems. The first problem is that it is not written in a declarative style. It heavily uses knowledge of the order in which Prolog will attempt its resolution. Secondly, the goal *order(B,A)* will always succeed if $A = B$, and therefore this program will endlessly loop if two elements of the list are equal.

The problem is based in the fact that bubble sort is an inherently *procedural* concept. It is therefore eminently suitable for implementation in a procedural language such as **C**. As the above example shows it is possible to make Prolog do an inherently procedural algorithm, but the result is rather inelegant.

The *quicksort* algorithm is far more suitable for Prolog implementation. This algorithm is based on the following clever idea. Consider a list $[H|T]$. We can split the tail T into two lists L and M, such that L contains all the elements less than H, and M contains all the elements that are more than H. Then the sorted list will consist of the sorted version of L, then H and then the sorted version of M. Each of L and M are recursively sorted by quicksort.

First consider the algorithm to split the list. We use the predicate *split(H,T,L,M)* to split the list $[H|T]$ into L and M.

```
split(H,[A|X],[A|Y],Z) :- order(A,H), split(H,X,Y,Z).
split(H,[A|X],Y,[A|Z]) :- order(H,A), split(H,X,Y,Z).
split(H,[],[],[]).
```

and then the quicksort algorithm is

```
quicksort([H|T],S) :-
split(H,T,A,B),
```

```
quicksort(A,A1),
quicksort(B,B1),
concat(A1,[H|B1],S).
quicksort([],[]).
```

This algorithm performs very well, particularly on large lists, where the number of splits needed is likely to be small.

4.7 Searching in Prolog

Searching is an area in which Prolog programs are very easy to write. Unfortunately although these programs are easy to write they are not usually fast enough for practical problems. However as the price of fast hardware keeps dropping, and the price of developing software keeps increasing, it seems likely that Prolog will only increase in practicality.

Let us start by giving an example of a searching problem, before discussing ways of attacking it. One of the most famous of all artificial intelligence puzzles is the 8 puzzle. Consider a 3×3 array of 9 squares 8 of which contain one of the digits from 1 to 8. Initially the digits are placed in the array at random. A *move* consists of moving one of the horizontally or vertically adjacent digits into the empty square. The aim is to perform a sequence of such moves to go from the initial configuration to the goal configuration

1	2	3
8		4
7	6	5

Almost all searching problems of this nature are best described using the language of graph theory. We define the *state-space* to be the set consisting of all possible configurations. In this case there are $9! = 362880$ possible states. Now we can consider a graph with 362880 vertices, each one labelled with one of these possible states. We join two states with an edge provided that there is a legal move that allows one to move from one state to the other. For example the goal configuration is joined to 4 other vertices.

1		3
8	2	4
7	6	5

1	2	3
8	4	
7	6	5

1	2	3
	8	4
7	6	5

1	2	3
8	6	4
7		5

Therefore we can regard the general problem in graph theory terms as the problem of finding a path from an arbitrary starting vertex to the goal vertex. One standard technique for performing such a search is depth-first search, in which the current path is lengthened by one move, and then checked to see if the goal state has been reached. If not then lengthen the path again. To avoid looping, a history of the moves made so far is maintained and only new states are examined. If the path has looped around itself so much that there are no new states left to move to, then backtracking is performed. Specifying such a program is easy in Prolog. We use the predicate *solve_dfs(State,History,Moves)* which will be true if *Moves* is a sequence of moves leading from *State* to the goal configuration that does not use any moves in *History*.

```
solve_dfs(State,History,[]) :- final_state(State).

solve_dfs(State,History,[Move|Moves]) :-
move(State,Move,State1),
not member(State1,History),
solve_dfs(State1,[State|History],Moves).
```

There are several things to notice here. The most important one is the use of the word `not`. In this case the goal $\leftarrow$*not member(State1,History)* will succeed if $\leftarrow$*member(State1, History)* fails, in other words if *State1* is not a member of the list *History*. This is an example of using Prolog to

get negative information, which as we shall see later must be treated very cautiously. For now it is enough to observe that this predicate will only behave as we expect if both *State1* and *History* are instantiated to values when it is called. This is clearly a procedural requirement, not a logical one. We would avoid this if we could. The second thing to notice is that we still require an additional predicate *move(State,Move,State1)*, which will be true if there is a move *Move* that takes one from *State* to *State1*. This predicate is basically a move *generator* — it produces all the moves possible from the position *State*, and the backtracking action of Prolog ensures that when one move has led to a failed path, then that goal will have to be resatisfied, and hence an alternative move will be tried.

As we can see even a puzzle that is seemingly this simple has a very large state-space graph. Our search as it is written is completely directionless and will usually find solutions that are thousands of moves long, despite the fact that shortest solutions are tens of moves long. In order to make the problem feasible we must make the search more intelligent in some way.

4.8 Structured Information in Prolog

Programs in Prolog can be written to exploit all the features of Prolog. In particular the unification mechanism of Prolog is very powerful, and many useful programs can be made that essentially run entirely using the unification mechanism. One such example of this is the retrieving of structured information from a database. For example a database can be represented in Prolog using compound terms. For example we might have a term that represents a book in the following fashion

```
book(title(spectra_of_graphs), authors([cvetkovic, doob, sachs]),
published(academic_press,1980),isbn(0-12-195-150-2)).

book(title(genetic_algorithms), authors([goldberg]),
published(addison_wesley,1989),isbn(0-210-15767-2)).

book(title(programming_in_prolog), authors([clocksin,mellish]),
published(springer,1981),isbn(0-387-11046-1)).
```

Notice clearly that here we are using the Prologs predicate names in a different way here — the predicate *book* is not really being used to represent true or false, but more to be a place-holder for structured information. Predicate symbols being used in this way are called *functors*.

Now we can simply use Prolog's matching facility to query this database. For example, suppose we want the titles of all books that Springer has published.

```
| ?-  book(title(X),_,published(springer,_),_).

X = programming_in_prolog
```

It is clear how to write simple predicates using *member* to access information about a books authors and so on. Certain special types of predicate are called selector predicates, in that they use unification to access particular components of the structure.

```
publisher(book(_,_,published(Publisher,_),_),Publisher).
```

is a predicate that will extract the publisher of a given book. It is possible to write quite a useful database just using the functors and a suite of simple utility programs such as the various selector predicates.

Another area in which Prolog's unification can be exploited is in the area of logic puzzles. Consider the following problem

> Three children ran in a race in their school sports day. Peter did better than the person who ran in red. Jack, wearing gold, did better than the child in green. Who won the race?

This problem can be solved easily using Prolog's unification and backtracking provided the structures to be used are chosen carefully.

First we must choose a representation for a child. We shall use the binary functor *child* with the two arguments being the child's name and shirt colour. So we would represent Jack in gold by

```
child(jack,gold)
```

The result for which we are searching is to be the order of the three children, so we shall represent that by a functor *order* with three arguments (which will be the children). Hence a possible result would be

```
order(child(jack,gold), child(peter,red), child(john,green))
```

Our aim is to get Prolog to instantiate a term of that type with an answer that is consistent with our clues.

A complete program would be

```
didbetter(X,Y,order(X,Y,_)).
didbetter(X,Y,order(X,_,Y)).
didbetter(X,Y,order(_,X,Y)).

clue1(S) :- didbetter(child(peter,_),child(_,red),S).
clue2(S) :- didbetter(child(jack,gold),child(_,green),S).
```

and the program is then run by asking for an S that satisfies all the clues

```
| ?- clue1(S), clue2(S).

S = order(child(jack,gold),child(peter,green),child(_15,red)) ;

no
```

Notice that the work of fitting the child's names and colours into the appropriate places was all done by Prolog.

5 Substitutions and Unification

In this section we examine more carefully the heart of logic programming — the unification mechanism. We recall that Prolog often had to match two terms — one was in the body of the goal clause, and the other being the head of the program clause. This process is called *unification* of the two terms. The unification algorithm is the mechanism by which the variables are bound, and hence the mechanism by which the answers are produced.

5.1 Substitutions

Definition. 5.1 *An* expression *is a term, a literal or a conjunction or disjunction of literals. A* simple expression *is either a term or an atom.*

Definition. 5.2 *A* substitution θ *is a finite set of the form* $\{v_1/t_1, v_2/t_2, \ldots, v_n/t_n\}$ *where each* v_i *is a variable, each* t_i *is a term (not equal to* v_i*) and the variables* $v_1, v_2, \ldots, v_n$ *are distinct. Each element* v_i/t_i *of the substitution is called a* binding *for* v_i*. The substitution is called a* ground substitution *if all the* t_i *are ground terms, and* variable-pure *if all the* t_i *are variables.*

Definition. 5.3 *Let* $\theta = \{v_1/t_1, v_2/t_2, \ldots, v_n/t_n\}$ *be a substitution and* E *be an expression. Then* $E\theta$*, the* instance *of* E *by* θ *is the expression obtained from* E *by simultaneously replacing all occurrences of the variable* v_i *in* E *by the term* t_i*. If* $E\theta$ *is ground, then* $E\theta$ *is called a* ground instance *of* E*.*

Example. 5.4 If $E = parent(X, Y)$ and $\theta = \{X/pam, Y/bob\}$, then $E\theta = parent(pam, bob)$. ■

Example. 5.5 If $E = p(X, Y, g(A, B), c)$ and $\theta = \{X/g(A), Y/c, A/f(X, Y)\}$ then simply applying the definition yields $E\theta = p(g(A), c, g(f(X, Y), B), c)$ ■

If $S = \{E_1, E_2, \ldots, E_n\}$ is a finite set of expressions, then $S\theta = \{E_1\theta, E_2\theta, \ldots, E_n\theta\}$.
We consider the effect of performing two substitutions one after the other.

Definition. 5.6 *Let $\theta = \{u_1/s_1, u_2/s_2, \ldots, u_m/s_m\}$ and $\sigma = \{v_1/t_1, v_2/t_2, \ldots, v_n/t_n\}$. Then the* composition *$\theta\sigma$ of the substitutions θ and σ is the substitution obtained from the set*

$$\{u_1/s_1\sigma, u_2/s_2\sigma, \ldots, u_m/s_m\sigma, v_1/t_1, v_2/t_2, \ldots, v_n/t_n\}$$

by deleting any binding $u_i/s_i\sigma$ for which $u_i = s_i\sigma$ and deleting any binding v_j/t_j for which $v_j \in \{u_1, u_2, \ldots, u_m\}$.

Let us consider this definition briefly. It is attempting to formalize the idea of performing two substitutions on an expression. Performing substitution $\theta\sigma$ should be the same as doing θ first and then σ. This explains the slightly strange definition. First θ is applied and all the u_i get replaced by the appropriate s_i. Then σ changes the newly formed s_i to $s_i\sigma$. Now unless we have just changed a variable back into itself (in which case that binding is deleted) then we get terms of the form $u_i/s_i\sigma$. Now consider occurrences of the variables v_i. If $v_i \in \{u_1, u_2, \ldots, u_m\}$ then it will be altered by θ and will already have been accounted for in the earlier terms. All the other occurrences of the v_i will be left unscathed by θ and bound according to σ alone.

Example. 5.7 Let $\theta = \{X/f(Y), Y/Z\}$ and $\sigma = \{X/a, Y/b, Z/Y\}$ and consider $\theta\sigma$. According to the definition, we consider the set

$$\{X/f(b), Y/Y, X/a, Y/b, Z/Y\}$$

and throw out the terms Y/Y, X/a and Y/b leaving $\{X/f(b), Z/Y\}$. ■

Definition. 5.8 *The substitution given by the empty set is called the identity substitution and is denoted ϵ.*

Composition of substitutions obeys some standard algebraic properties

Lemma. 5.9 *Let θ, σ and γ be substitutions. Then*

1. $\theta\epsilon = \epsilon\theta = \theta$
2. $(E\theta)\sigma = E(\theta\sigma)$ *for all expressions E.*
3. $(\theta\sigma)\gamma = \theta(\sigma\gamma)$

Definition. 5.10 *Let E and F be expressions. Then E and F are called* variants *of each other if there are substitutions θ and σ such that $E = F\theta$ and $F = E\sigma$.*

Definition. 5.11 *Let E be an expression and let V be the set of variables occurring in E. Then a* renaming substitution *is a variable-pure substitution $\{x_1/y_1, \ldots, x_n/y_n\}$ such that $\{x_1 \ldots, x_n\} \subseteq V$, the y_i are distinct and $(V \backslash \{x_1, x_2, \ldots, x_n\}) \cap \{y_1, y_2, \ldots, y_n\} = \emptyset$.*

This definition seems somewhat complex at first, all it is really saying is that a renaming substitution simply takes the variables in the first expression and gives them new names. The extra conditions are simply to make sure that the new variables do not get mixed up with the old ones.

Example. 5.12 Consider the expression $E = p(X, g(Y))$. Let $\theta = \{X/Z, Y/V\}$. Then $E\theta = p(Z, g(V))$. Clearly $E\theta$ is a variant of E, because we can use the substitution $\{Z/X, V/Y\}$ to turn $E\theta$ into E. ■

The previous example makes it clear that if an expression can be obtained from another by a renaming substitution, then the two expressions are variants. It is more surprising that the converse is also true.

Lemma. 5.13 *Let E and F be variants. Then there exist renaming substitutions θ and σ such that $E = F\sigma$ and $F = E\theta$.*

PROOF. See Lloyd (1987), p22 ■

We are interested in substitutions that make each expression in a set of expressions identical. If $S = \{E_1, E_2, \ldots, E_n\}$ is a set of expressions, then we define $S\theta = \{E_1\theta, E_2\theta, \ldots, E_n\theta\}$ Notice that as $S\theta$ is a set, it may have fewer elements than S. We are particularly interested in substitutions such that $S\theta$ contains only one element. Such a substitution is called a *unifier* for the set of expressions. In particular, Prolog attempts to unify an atom in a goal clause with an atom in the head of a program clause.

Definition. 5.14 *Let S be a finite set of simple expressions. A substitution θ is called a* unifier *for S if $S\theta$ contains only one expression. A unifier θ for S is called a* most general unifier (mgu) *if for every unifier σ of S there is a substitution γ such that $\sigma = \theta\gamma$.*

This is another definition that requires some thought. The concept of a unifier is easy to grasp as being any substitution such that $S\theta$ is a singleton. A most general unifier roughly corresponds to the idea that we do not bind any variables unless they have to be bound. The definition states that θ is a most general unifier provided that any other unifier can be obtained by using θ first and then a further substitution. So the most general unifier can be thought of as the smallest amount of substituting that must be done to ever unify the two expressions.

Example. 5.15 Let $S = \{p(X, f(X, g(Y))), p(X, Z)\}$. Then $\sigma = \{X/a, Y/b, Z/f(a, g(b))\}$ is certainly a unifier, because $S\sigma = \{p(a, f(a, g(b))\}$. However it is not a most general unifier. Consider $\theta = \{Z/f(X, g(Y))\}$. Then θ is also a unifier, because $S\theta = \{p(X, f(X, g(Y))\}$ but it is more general than σ because we did not unnecessarily bind the variables X and Y. In fact θ is a most general unifier. Notice here that $\sigma = \theta\{X/a, Y/b\}$. ■

5.2 A unification algorithm

We now present a full unification algorithm for unifying a finite set of expressions. The idea is simply to read along the two expressions until coming to a symbol where they differ. Then we try to unify the two subexpressions starting at that point. If that is successful, then we resume reading along the expressions, find the next symbol at which they differ, and unify the subexpressions starting at that point. If this process completes then the composition of all the substitutions made will be a most general unifier, and if at any stage unification cannot be successfully completed, then the two expressions cannot be unified.

Example. 5.16 Let $S = \{p(X), q(X)\}$. Then S cannot be unified, because the very first symbols of $p(X)$ and $q(X)$ are different, and neither is a variable. ■

Example. 5.17 Let $S = \{p(X), p(a)\}$. Then the first symbols at which they disagree are the X and the a. However these can be unified by the substitution $\{X/a\}$ and this substitution is hence an mgu for S. ■

To describe the algorithm more formally we need one more definition

Definition. 5.18 *Let S be a finite set of simple expressions. The* disagreement set *of S is the set of subexpressions, one from each member of S, each starting at the leftmost symbol at which not all expressions in S agree.*

Example. 5.19 The disagreement set of $\{p(X), q(X)\}$ is $\{p(X), q(X)\}$ whereas the disagreement set of $\{p(X), p(a)\}$ is $\{X, a\}$. ■

Algorithm. 5.20 *1. Put $k = 0$ and set $\sigma_0 = \epsilon$*

2. If $S\sigma_k$ is a singleton, then stop and output σ_k

3. Find D_k the disagreement set of $S\sigma_k$

4. If D_k contains a variable v and a term t not using v, then set $\sigma_{k+1} = \sigma_k\{v/t\}$, increment k and go to step 2.

5. Otherwise, stop with a message that S is not unifiable

Notice that this algorithm proceeds by eliminating one variable at a time, and hence is guaranteed to stop. However, it is non-deterministic because there may be several choices for v and t at step 4.

Let us consider several examples of its use.

Example. 5.21 Let $S = \{p(f(X), Z), p(Y, a)\}$. Then proceed as follows

$\sigma_0 = \epsilon$
$S\sigma_0 = \{p(f(X), Z), p(Y, a)\}$
$D_0 = \{f(X), Y\}$
$\sigma_1 = \sigma_0\{Y/f(X)\} = \{Y/f(X)\}$
$S\sigma_1 = \{p(f(X), Z), p(f(X), a)\}$
$D_1 = \{Z, a\}$
$\sigma_2 = \sigma_1\{Z/a\} = \{Y/f(X), Z/a\}$
$S\sigma_2 = \{p(f(X), a)\}$

therefore we exit with the result that $\sigma_2 = \{Y/f(X), Z/a\}$ is a most general unifier for S. ■

Example. 5.22 Let $S = \{p(X, X, g(X), h(a)), p(f(Y), f(h(Z)), g(X), Y)\}$. Then proceeding with the algorithm we get

$\sigma_0 = \epsilon$
$S\sigma_0 = \{p(X, X, g(X), h(a)), p(f(Y), f(h(Z)), g(X), Y)\}$
$D_0 = \{X, f(Y)\}$
$\sigma_1 = \sigma_0\{X/f(Y)\} = \{X/f(Y)\}$
$S\sigma_1 = \{p(f(Y), f(Y), g(f(Y)), h(a)), p(f(Y), f(h(Z)), g(f(Y)), Y)\}$
$D_1 = \{Y, h(Z)\}$
$\sigma_2 = \sigma_1\{Y/h(Z)\} = \{X/f(h(Z)), Y/h(Z)\}$
$S\sigma_2 = \{p(f(h(Z)), f(h(Z)), g(f(h(Z))), h(a)), p(f(h(Z)), f(h(Z)), g(f(h(Z))), h(Z)\}$
$D_2 = \{Z, a\}$
$\sigma_3 = \sigma_2\{Z/a\} = \{X/f(h(a))), Y/h(a), Z/a\}$
$S\sigma_3 = \{p(f(h(a)), f(h(a)), g(f(h(a))), h(a))\}$

therefore we exit with the result that $\sigma_3 = \{X/f(h(a))), Y/h(a), Z/a\}$ is a most general unifier for S. ■

Now consider the following example

Example. 5.23 Let $S = \{p(X, X), p(Y, f(Y))\}$. Then the algorithm yields

$\sigma_0 = \epsilon$
$S\sigma_0 = \{p(X, X), p(Y, f(Y))\}$
$D_0 = \{X, Y\}$
$\sigma_1 = \sigma_0\{X/Y\}$
$S\sigma_1 = \{p(Y, Y), p(Y, f(Y))\}$
$D_1 = \{Y, f(Y)\}$

and here the algorithm fails because we cannot find a variable v and a term t *not involving* v. Therefore the expressions are not unifiable. ■

This last example illustrated the use of the *occur check*. It is vitally important at every stage that every new binding does not bind a variable v to a term that actually contains v. The occur check can be a very expensive part of the unification algorithm. The following is a classic example of a very expensive occur check.

Example. 5.24 Let $S = \{p(X_1, X_2, \ldots, X_n), p(f(X_0, X_0), f(X_1, X_1), \ldots, f(X_{n-1}, X_{n-1}))\}$. Then the algorithm gives us

$$
\begin{aligned}
&D_0 = \{X_1, f(X_0, X_0)\} \\
&\sigma_1 = \{X_1/f(X_0, X_0)\} \\
&D_1 = \{X_2, f(f(X_0, X_0), f(X_0, X_0))\} \\
&\sigma_2 = \{X_1/f(X_0, X_0), X_2/f(f(X_0, X_0), f(X_0, X_0))\} \\
&D_2 = \{X_3, f(f(f(X_0, X_0), f(X_0, X_0)), f(f(X_0, X_0), f(X_0, X_0)))\}
\end{aligned}
$$

and so on. The disagreement set D_k contains a term with 2^{k+1} occurrences of X_0, and hence in this case the occur check alone takes an exponential amount of time. ■

The previous example demonstrates that any algorithm that gives the most general unifier explicitly can never be better than exponential complexity in the worst case. In practice however fast unification algorithms can be written by such techniques as careful choice of data structure for th expressions, and representing the unifier as a composition of substitutions rather than an explicit substitution. It is in fact possible to perform unification in linear time.

Because the occur check is so expensive, most common Prolog implementations overcome the problem simply by omitting it! In practice this is not a severe problem because often an occur check violation will cause the program to quickly crash, or be unable to produce any output. However the omission of the occur check destroys the soundness of Prolog's resolution, so is theoretically disastrous.

6 Herbrand's theorem

We now return to more of our theoretical discussion of logic programming. We recall once again the definition of logical implication for predicate logic

Definition. 6.1 *Let S be a set of closed formulas. Then the closed formula F is called a* logical consequence *of S if every model for S is also a model for F.*

Now our aim is to have the system find logical consequences of a given set of clauses. Therefore the above definition raises two seemingly difficult problems. Firstly it seems that it will be necessary to check *every* model for S in order to prove logical consequence. Secondly we want our system to perform its work by mechanical manipulation of symbols, and it seems that the requirement to check every possible model is inconsistent with this aim. In this section we shall consider a special kind of interpretation, called a Herbrand interpretation which to a large extent solves these problems.

6.1 Herbrand interpretations

So far we have always used interpretations to make some connection between logic programs and the real world. However, our definition of interpretation was purely mathematical, and so we can consider a very formal type of interpretation. This is a purely symbolic interpretation: in effect we are setting up a situation where the "meaning" of a formula in predicate logic is just a string of symbols. We start with some definitions

Definition. 6.2 *Let L be a first order language. The* Herbrand universe U_L *of the language is the set of all ground terms which can be made out of the constant symbols and function symbols of L. (If there are no constant symbols in L then we arbitrarily choose a symbol to represent a constant).*

Example. 6.3 Consider a language with one constant symbol a, one unary function symbol f and one binary function symbol g. Then the Herbrand universe is all the terms that can be made from these symbols according to the rules for forming terms.

$$U_L = \{a, f(a), f(f(a)), g(a,a), b), g(a, f(a)), g(f(a), a)), g(f(a), f(a)), f(g(a,a)), \ldots\}$$

■

Definition. 6.4 *Let L be a first order language. Then the* Herbrand base B_L *consists of all the ground atoms that can be formed from the predicate symbols of L using the ground terms of the Herbrand universe as arguments.*

Example. 6.5 Suppose the language L used in the previous example has only one predicate symbol, a binary predicate called p. Then the *Herbrand base* is the set of all possible atoms made from p and the terms in the Herbrand universe

$$B_L = \{p(a,a), p(a, f(a)), p(f(a), a), p(f(a), f(a)), p(g(a,a), a), p(a, g(a,a)), \ldots\}$$

■

Therefore the Herbrand base is all the possible ground atoms that can be expressed in the language L. Now we come to the slightly tricky concept of a Herbrand interpretation

Definition. 6.6 *An interpretation of a first order language L is a* Herbrand interpretation *provided that*

1. *The domain D is the Herbrand universe U_L*
2. *Constants in L are assigned to themselves in U_L*
3. *If f is an $n-$ary function then the mapping $f' : D^n \to D$ is given by $f'(t_1, t_2, \ldots, t_n) = f(t_1, t_2, \ldots, t_n)$.*

It is important to try to grasp the ideas behind this definition. The Herbrand universe consists of all the terms that can be made from the language L. In all interpretations of L the terms will represent elements of the domain. Sometimes two terms will represent the same element of the domain, and sometimes there will be elements of the domain that are not represented by any term at all. Taking the domain to actually *be* the set of terms gives us an interpretation in which each term represents exactly one element of the domain and every element of the domain is represented. In some sense a Herbrand interpretation is a "most general interpretation" of a language L. This allows us to abstractly consider *all* the interpretations of a language L without being too specific, yet still retaining the limitations inherent in the expressiveness of L.

The mappings associated to the predicates are not specified, so in order to finish the specification of a Herbrand interpretation we must assign a predicate to each predicate symbol. As a predicate is merely a function which returns the values *true* or *false* it is sufficient to specify the predicate merely by indicating the arguments for which it returns *true* with the convention that it returns *false* for all other arguments. (For example, we can specify the predicate `odd(X)` in either of the following two fashions (1) `odd(X)` is true when X is odd and false otherwise (2) `odd(X)` is true when $X = \{1, 3, 5, 7, \ldots\}$)

The Herbrand base contains all the ground atoms that can be formed from the language L and therefore the predicates can be completely specified by selecting which ground atoms are to be true (with the convention that the other ground atoms are false). Therefore a Herbrand interpretation can be completely specified merely selecting a *subset* of the Herbrand base B_L, to represent the true statements. Therefore there are many different Herbrand interpretations for any given language, one for each subset of B_L. This is sufficiently important to state as a theorem.

Theorem. 6.7 *There is a 1-1 correspondence between the subsets of B_L and the Herbrand interpretations of L.*

This correspondence allows us to *identify* Herbrand interpretations with subsets of the Herbrand base. This means that we actually describe a Herbrand interpretation merely by giving the subset of the Herbrand base representing the true statements.

Example. 6.8 Let L be a language with one constant symbol a, one unary function symbol f and one predicate symbol p. Then

$$U_L = \{a, f(a), f(f(a)), f(f(f(a))), \ldots\}$$

$$B_L = \{p(a), p(f(a)), p(f(f(a))), p(f(f(f(a)))), \ldots\}$$

Therefore in a Herbrand interpretation we regard the domain as simply being the set of strings in U_L, the function f' acts on this set of strings by forming a new string. The Herbrand base gives all the formal statements that can be made using that language. To obtain a Herbrand interpretation, we simply choose a subset of those statments to be true. So we might consider the following three interpretations.

$$\mathcal{I}_1 = \{p(a)\}$$

$$\mathcal{I}_2 = \{p(f(f(a))), p(f(f(f(a)))), \ldots\}$$

$$\mathcal{I}_3 = \{p(f^n(a)) \mid n \text{ is prime}\}$$

At this stage these are merely arbitrarily chosen sets or interpretations and none is any "better" than the others. However, if we consider a particular set of clauses then some interpretations will be models and some will not. ■

Definition. 6.9 *Let S be a set of closed formulas of a language L. Then a* Herbrand model *of S is a Herbrand interpretation that is a model for S.*

Example. 6.10 Let L be as before, and let S be the set of clauses

p(f(f(a)))
p(f(X)) ←p(X)

Now $\mathcal{I}_1$ is not a model for S, because any model for S must contain the atom $p(f(f(a))$. On the other hand $\mathcal{I}_2$ is also not a Herbrand model for S. It does contain $p(f(f(f(a))))$, but it does not contain $p(f(f(f(f(a)))))$ and yet $p(f(f(f(f(a)))))\leftarrow p(f(f(f(a))))$ is a ground instance of a clause in S. However $\mathcal{I}_3$ is a model for S. ■

The concept of Herbrand interpretation is often hard to get used to. It is often useful to think of Herbrand interpretations and models as just applying to formal strings of symbols. This is consistent with our aim of having the system mimic natural reasoning by mechanical manipulation of symbols. A Herbrand interpretation can therefore just be thought of as a listing of all the strings of symbols that we wish to regard as true. We can extend this idea and think of a Herbrand model for S as being all the strings of symbols that are true provided the ones in S are true.

Our reason for concentrating on this seemingly artificial and restricted class of interpretations is the remarkable fact that it is all that is necessary to prove logical implication *provided we restrict our attention to clauses.*

Theorem. 6.11 (Herbrand's theorem) *Let S be a set of clauses, and suppose that S has a model. Then S has a Herbrand model.*

PROOF. Let $\mathcal{I}$ be an interpretation of S. Then define a Herbrand interpretation $\mathcal{I}'$ in the following manner.

$$\mathcal{I}' = \{p(t_1, t_2, \ldots, t_n) \mid p(t_1, t_2, \ldots, t_n) \text{ is true with respect to } \mathcal{I}\}$$

Now we must show that $\mathcal{I}'$ is a model for S. This involves demonstrating that every clause in S is true under the interpretation $\mathcal{I}'$. So, consider a clause in S.

$$A \leftarrow B_1, B_2, \ldots, B_n$$

We must show that for every possible choice for the variables in the clause that it is true under $\mathcal{I}'$. Consider a particular choice of variables so that the clause is ground. If any of the B_i are false then the clause itself is true, so we need only consider the case where they are all true. Then however A must also be true under the interpretation $\mathcal{I}$, and hence it is in $\mathcal{I}'$. ■

It is extremely important to note that this result does not apply in the case where S contains formulas that are not clauses.

Example. 6.12 The set $S = \{p(a), \exists x \sim p(x)\}$ is a satisfiable set of formulas, but it does not have a Herbrand model. We see this because the Herbrand base is $\{p(a)\}$ and hence there are only two interpretations, $\emptyset$ and $\{p(a)\}$ itself. However neither of these two interpretations is a model. ■

Essentially all the theoretical logic programming that we shall be doing is based on Herbrand's theorem, and it is for this reason that we are restricted to clauses.

We restate Herbrand's theorem in two further ways.

Corollary 6.13 *If S is a satisfiable set of clauses then S has a Herbrand model.*

Stating the contrapositive of this statement (which is just the same statement) we obtain

Corollary 6.14 *If S does not have a Herbrand model, then S is unsatisfiable.*

Therefore we have proved that it is only necessary to consider all the Herbrand models of a set S of clauses in order to prove unsatisfiability.

6.2 The Least Herbrand Model

Consider a logic program P, that is, a finite set of program clauses. The clauses in P use certain function symbols, predicate symbols, and constants, and we can consider a language L that just uses these symbols. We do not lose anything by supposing that L is actually the language from which P arose. Therefore we shall extend our terminology and define U_P the Herbrand universe of P, and B_P the Herbrand base of P to be the Herbrand universe and base of that language.

Now consider models for P. Recall that any interpretation (and hence model) can be regarded as a subset of B_P. Then the next result is clear:

Theorem. 6.15 *Let P be a logic program and let $\{M_i\}_{i \in I}$ be a set of Herbrand models for P. Then*

$$M = \bigcap_{i \in I} M_i$$

is a Herbrand model for P.

PROOF. We must show that any statement of P is true under the interpretation M. Let $A \leftarrow B_1$, B_2, ..., B_n be an arbitrary ground instance of a clause in P, and we shall show that this is true under the interpretation M. If one or more of B_1, B_2, ..., B_n is not in M, then they are false, and so the statement is true. On the other hand, if all of B_1, B_2, ..., B_n are in M, then they are all in M_i for all i. However every M_i is a model for $A \leftarrow B_1$, B_2, ..., B_n and so A is in every M_i and

thus also in M. Hence M is a model for the clause, and as it was picked arbitrarily M is a model for every clause. ■

Therefore we can go further and consider the intersection of *all* Herbrand models for P. This will again be a Herbrand model, and is the smallest such model. For this reason it is called the *least Herbrand model* and is denoted M_P.

Definition. 6.16 *Let P be a logic program. Then the intersection of all the Herbrand models for P is a model called the* least Herbrand model *of P and denoted M_P.*

The least Herbrand model is of great theoretical importance. Recall that Herbrand interpretations are the "most general interpretations"; therefore the Herbrand models are the most general models and the least Herbrand model consists of those statements that are true in *every* Herbrand model for P. Therefore M_P contains precisely of those ground atoms that are logical consequences of P.

Theorem. 6.17 *Let P be a logic program. Then $M_P = \{A \in B_P \mid A$ is a logical consequence of $P\}$.*

PROOF. We must show that everything in M_P is a logical consequence of P, and also that every logical consequence of P is actually in M_P. So, consider a clause A that is in M_P. Then A is true in every model for P. Therefore in any model for P, $\sim A$ is false, and hence $P \cup \{\sim A\}$ is unsatisfiable. Conversely, suppose that $P \cup \{\sim A\}$ is unsatisfiable. Then A is in every model for P, and hence $A \in M_P$. ■

As the least Herbrand model contains every ground atom that is a logical consequence of the program P, we can say in a broad sense that the aim of logic programming is to find the least Herbrand model of a program (or at least specific elements of the least Herbrand model). This is not quite correct because as we discussed earlier, we are usually not specifically interested in ground atoms, but more what substitutions need to be made to result in a logical consequence.

Definition. 6.18 *Let P be a logic program, and G a goal $\leftarrow A_1, A_2, \ldots, A_n$. A* correct answer *for $P \cup \{G\}$ is a substitution θ for (some of) the variables of G such that*

$$\forall x_1 \forall x_2 \ldots \forall x_k((A_1 \wedge A_2 \wedge \ \ldots \ \wedge A_n)\theta)$$

is a logical consequence of P.

We can see that this definition is consistent with our view of what a correct answer should be. Recall the basic set-up of logic programming. We wish to determine the truth of the proposition

$$\exists x_1 \exists x_2 \ldots \exists x_k(A_1 \wedge A_2 \wedge \ \ldots A_n)$$

We do this by negating that clause to form the goal G

$$\forall x_1 \forall x_2 \ldots \forall x_k \sim(A_1 \wedge A_2 \wedge \ \ldots A_n)$$

which in clausal form is the goal

$$\leftarrow A_1, A_2, \ldots, A_n$$

Then Prolog shows that the $P \cup \{G\}$ is unsatisfiable, by demonstrating that in fact there ARE choices of variables that make the original proposition true. A correct answer says that once the bindings of variables in θ are done, then any choice of the remaining variables will produce a logical consequence of P.

7 SLD Resolution

In this section we delve deeper into the formal description of how logic programming systems find correct answers. We have already seen that logic programming uses resolution to produce refutation proofs. There are many ways of using resolution; we shall be examining SLD-resolution (SL-resolution for Definite clauses).

First we use the terminology developed in the last section to give a proper definition of resolvent.

Definition. 7.1 *Let G be the goal $\leftarrow A_1, A_2, \ldots, A_m, \ldots, A_k$ and let C be the program clause $A \leftarrow B_1, B_2, \ldots, B_q$. Then G' is* derived from G and C using θ *if*

1. *A_m is an atom, called the* selected atom *in G*
2. *θ is an mgu of A_m and A*
3. *G' is the goal $\leftarrow (A_1, A_2, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$*

This is simply a formalizing of our previous definition of resolution using substitutions. Recall that G' is then called the resolvent of G and C.

Definition. 7.2 *Let P be a logic program and G be a goal. Then an* SLD-derivation *of $P \cup \{G\}$ is a sequence $G = G_0, G_1, \ldots$ of goals, a sequence $C_1, C_2, \ldots$ of variants of program clauses and a sequence $\theta_1, \theta_2, \ldots$ of substitutions such that each G_i is derived from C_i and G_{i-1} using θ_i.*

The only part of this definition that requires comment is the part about the C_i being variants of program clauses. This is simply to avoid clashing names of variables. In any closed formula the names of the variables are not relevant. One simple way of producing variants in such a way that the variables will never clash is to always use the subscript i on every variable in clause C_i. Otherwise it is clear that an SLD-derivation is simply a sequence where each goal is the resolvent of the previous goal and some program clause. Obviously we are interested in such sequences that end in the empty clause $\Box$.

Definition. 7.3 *An* SLD-refutation *of $P \cup \{G\}$ is an SLD-derivation with final goal being the empty clause $\Box$.*

Figure 2 gives a pictorial view of an SLD-derivation.

Consider the following logic program P

```
p(X,Z) ←q(X,Y), p(Y,Z)
p(X,X) ←
q(a,b) ←
```

and the goal $G = \leftarrow p(X,b)$. An SLD-refutation for $P \cup \{G\}$ is shown in Figure 3.

A *successful* SLD-derivation is a refutation, and a *failed* SLD-derivation is one in which the selected atom in the goal does not unify with any of the program clauses. As well as these two possibilities an SLD-derivation can be infinite.

Definition. 7.4 *The* success set *of the logic program P is the set of all $A \in B_P$ such that $P \cup \{\leftarrow A\}$ has an SLD-refutation.*

The success set therefore contains all the ground atoms that can be proved by an SLD-refutation. So, we now have two concepts — one declarative, and one procedural. The least Herbrand model is the set of all ground atoms that are logical consequences of P, and the success set is the set of all ground atoms that have SLD-refutations, and hence can be proved by the system. Once again therefore we have the important questions of *soundness* and *completeness*. SLD-resolution is *sound* if the success set is a subset of the least Herbrand model; in other words, everything that has a refutation actually *is* a logical consequence. SLD-resolution is *complete* if the least Herbrand model is a subset of the success set; in other words, we can find a refutation for *every* logical consequence.

Therefore SLD-resolution will be sound and complete provided that the least Herbrand model is actually equal to the success set.

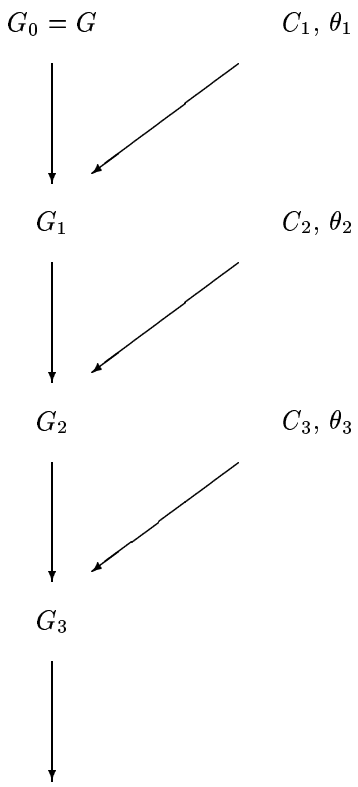

Figure 2: An SLD-derivation

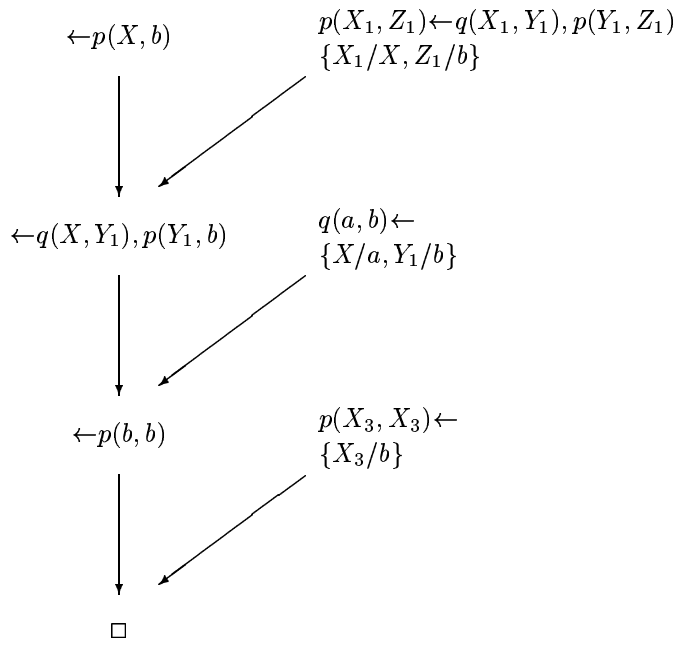

Figure 3: An SLD-refutation

Definition. 7.5 *Let P be a logic program and G a goal. A* computed answer *θ for $P \cup \{G\}$ is the substitution obtained by restricting the composition $\theta_1\theta_2 \ldots \theta_n$ to the variables of G, where θ_1, θ_2, $\ldots$, θ_n is the sequence of mgus used in an SLD-refutation of P.*

Example. 7.6 *Consider the SLD-refutation shown in Figure 3. The composition of the sequence of mgus is $\{X_1/a, Z_1/b, X/a, Y_1/b, X_3/b\}$, and the restriction of this to the variables in G gives us $\{X/a\}$ which is the computed answer.*

7.1 Soundness and completeness of SLD-resolution

Here we present the results that demonstrate the soundness of SLD-resolution, and discuss, but do not prove the completeness of SLD-resolution.

Theorem. 7.7 *Let P be a logic program and G be a goal. Then every computed answer for $P \cup \{G\}$ is a correct answer for $P \cup \{G\}$.*

PROOF. See Lloyd (1987)p 43 ■

Corollary 7.8 *Let P be a logic program and G be a goal. If there is an SLD-refutation for $P \cup \{G\}$ then $P \cup \{G\}$ is unsatisfiable.*

Corollary 7.9 *The success set of a logic program is contained in its least Herbrand model.*

Therefore we can conclude that SLD-resolution is sound; everything that has an SLD-refutation is actually a logical consequence of the logic program.

We have not yet introduced the tools to prove it, but it is the case that SLD-resolution is also complete. We simply state the main results without proof.

Theorem. 7.10 *The success set of a logic program is equal to its least Herbrand model.*

This theorem demonstrates that for ground atoms, SLD-resolution is complete. However, we are not usually dealing with ground atoms, and are particularly interested in the answer substitutions that a logic programming system produces.

Theorem. 7.11 *Let P be a logic program and G be a goal. If $P \cup \{G\}$ is unsatisfiable, then there is an SLD-refutation of $P \cup \{G\}$.*

Theorem. 7.12 *Let P be a logic program and G be a goal. For every correct answer θ for $P \cup \{G\}$ there is a computed answer σ for $P \cup \{G\}$ and a substitution γ such that $\theta = \sigma\gamma$.*

7.2 Prolog is not sound

Unfortunately, having gone to all the trouble of proving that SLD-resolution is sound, we must now examine the fact that common implementations of Prolog are not sound. The reason for Prolog's lack of soundness is simply the omission of the occur check.

Consider the following logic program

test $\leftarrow$*p(X,X)*
p(X,f(X)) $\leftarrow$

Now, given the goal $\leftarrow$*test* the Prolog system without the occur check will mistakenly answer `yes`. This will occur because the atom *p(X,X)* will be unified with the atom *p(X,f(X))*. However the atom *test* is not a logical consequence of this program. Therefore, practical Prolog implementations can sometimes provide "refutations" for things that are not logical consequences of the logic program. Therefore, we conclude that Prolog is not sound.

7.3 SLD-trees

To this point it may appear that the goal of using logic as a programming language is not too remote. We have described a technique for proving logical implication, and indicated that it is sound and complete. The only major problems that we have indicated in Prolog are problems of efficiency, such as the expense of performing the occur check. The results to date indicate that for any goal that actually is a logical consequence of the program P, there is an SLD-refutation that demonstrates that fact. In this section we consider the problem of how to actually *find* the SLD-refutation.

The first problem we have to consider is how the system should choose which atom in the goal to try to unify.

Definition. 7.13 *A* computation rule *R is a function from the set of all goals to the set of all atoms, such that the value of the function on any particular goal is one of the atoms in that goal, called the* selected atom.

This definition is simply a formal way of stating that there is some rule that will select one of the atoms from that goal. Prolog systems usually use the rule that simply selects the left-most atom in the goal.

Definition. 7.14 *An* SLD-derivation via rule *R is an SLD-derivation in which the selected atom at each stage is that given by R.*

Although SLD-refutations using a particular rule R are more restricted that general SLD-refutations it can be shown that the completeness of SLD-resolution is independent of which rule is actually chosen.

Theorem. 7.15 *Let P be a logic program, G a goal, and R a computation rule. Then for every correct answer θ for $P \cup \{G\}$ there is an $R-$computed answer σ and a substitution γ such that $\theta = \sigma\gamma$.*

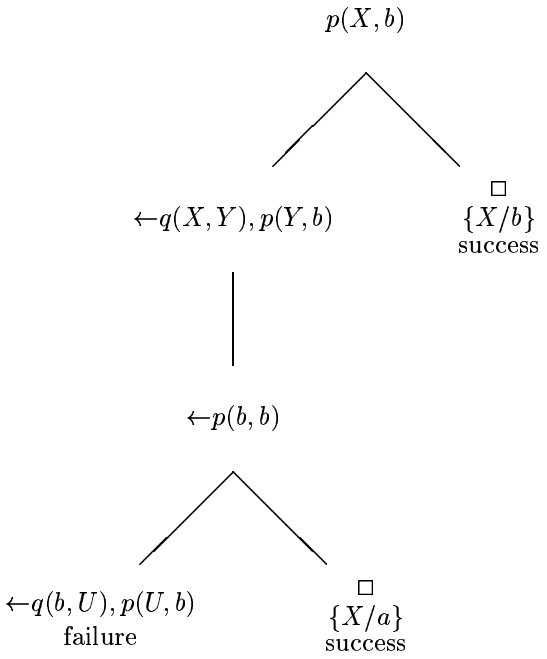

Figure 4: An SLD-tree using left-most rule

PROOF. The proof is fairly technical, and may be found in §9 of LLoyd (1987) ■

This is simply a counterpart of the general completeness theorem for SLD-resolution, stating that if a particular rule R is used, then completeness still holds. Therefore theoretically we lose nothing if we choose a particular rule R at the beginning and stick to it. In practice of course, there are differences depending on which rule is actually chosen, but these cannot be predicted in advance.

Now we consider the SLD-tree, which may be considered as all the possible derivations of a goal given a particular computation rule R. The description of a practical logic programming system, for example Prolog, is the details of how it searches the SLD-tree in attempting to find an SLD-refutation.

Definition. 7.16 *Let P be a logic program, G a goal and R a computation rule. Then an* SLD-tree for $P \cup \{G\}$ via R *is a tree such that*

1. *Each node of the tree is a goal*
2. *The root of the tree is G*
3. *Let N be a node. For each program clause C, if the selected atom in N can be unified with the head of C, the resolvent N' of N and C is a child of N*

Consider the example we used earlier, the program

p(X,Z) $\leftarrow$*q(X,Y), p(Y,Z)*
p(X,X) $\leftarrow$
q(a,b) $\leftarrow$

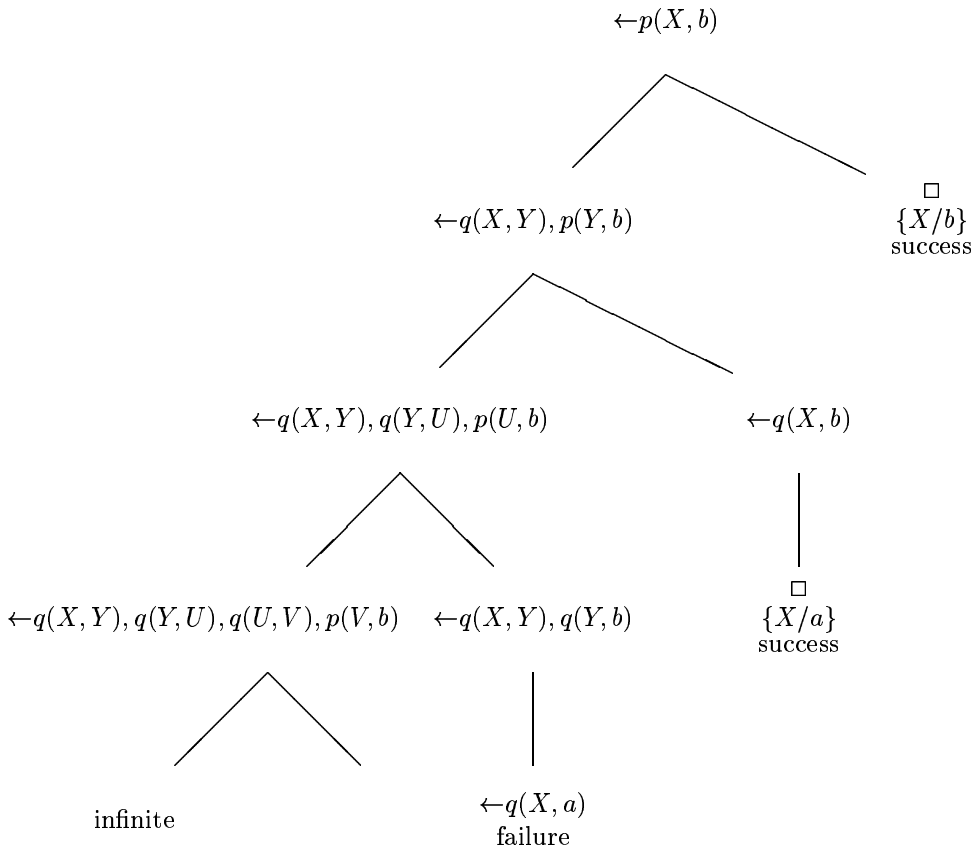

Figure 5: SLD-tree using right-first rule

and the goal $G = \leftarrow p(X,b)$. An SLD-tree using the rule that selects the left-most atom can be seen in Figure 4. Firstly notice that each path from the node to a leaf of this tree is a possible SLD-derivation. Furthermore each leaf of the tree is either the empty clause, or a node from which no further resolution can be performed using the selected atom. The leaves are therefore referred to as success leaves or failure leaves.

Now a different computation rule can lead to a tree that is very different in size and structure. Figure 5 shows the tree for the very same goal.

The importance of SLD-trees is that they display all the possible SLD-derivations that the system can perform. What a logic programming system actually does when it tries to find an SLD-refutation is to search the SLD-tree for success leaves. When the system finds a success leaf it may either stop and report success or continue to search for other success leaves.

Notice that although the trees are very different they both have the same number of success leaves. This is a straightforward consequence of the completeness theorem — every correct answer has a variant that is computed using the rule R. Unfortunately it is not possible in advance to see how the choice of rule will alter the SLD-tree, so Prolog merely sticks with the left-first rule. In this

case it is a good choice but it is easy to construct examples where the left-first rule produces much larger trees than right-first rule.

7.4 Searching the SLD-tree

We have seen that to produce a proof by refutation a logic programming system must search the SLD-tree for success leaves. Therefore we must consider exactly what strategy the system should follow in searching the tree - we must specify a *search rule*.

Now ideally we want a search rule that is fair and efficient, where by *fair* we mean that any success leaf will eventually be found. The two main techniques for searching a tree are breadth-first search and depth-first search.

In a depth-first search the search proceeds from a node to a child of that node, lengthening that branch, until a leaf is reached. If the leaf is a success leaf then all is well, but if it is a failure leaf the search *backtracks* to the previous node and tries the next child of that node. In this fashion the whole tree is searched. Depth-first search is very efficient to implement, but however it is not fair. Examination of the SLD-tree shown in Figure 5 shows that a depth first search will go along an infinite branch — never reaching a leaf of any sort!

The alternative is breadth-first search, in which all branches of a certain length are considered at the same time. First all of the children of the root are examined, then all of these branches are extended by one step, and all the nodes at distance 2 from the root are examined, and so on. This rule is fair, but is very inefficient to implement. In particular, the storage of all the branches is very expensive.

In practical Prolog systems the efficiency considerations force the adoption of the depth first search rule. Therefore Prolog systems are not fair, and if they get lost down an infinite branch never find solutions on different branches of the tree.

Therefore we are forced to the conclusion that although SLD-resolution is both sound and complete, Prolog is neither sound (due to the lack of occur check) nor complete (due to the depth first search rule). In short, practical logic programming is still well short of the ideal of theoretical logic programming. Therefore practical Prolog systems are forced to use many non-logical or extra-logical features to provide working programs. An example of such a feature is the cut, which gives the programmer control over how the SLD-tree is searched, by pruning certain branches. This is a non-logical feature because it requires the programmer to know how the search is being performed, and to determine that the portion of the SLD-tree being pruned does not contain any success leaves.

8 Negative information

We have indicated that for proving logical implication SLD-resolution is sound and complete (though the practical implementations are neither). However, logical implication allow us only to deduce positive information. That is, given a program P and a ground atom $A \in B_P$ it is possible to show that A is a logical consequence of P, but it is not possible to show that $\sim A$ is a logical consequence of P. In fact the negation of an atom is *never* a logical consequence of a program, because $P \cup \{ \sim \sim A\} = P \cup \{A\}$ is always satisfiable (unless P is inconsistent itself).

However there are many situations in real life in which we wish to infer negative information. In our propositional logic example where we listed the requirements for obtaining a mathematics degree, if somebody does not satisfy them then we wish to infer $\sim mathReq$. This is because we believe that we have given complete information about the mathematics requirements. This is an important distinction to make.

Consider the logic program

married(jim).
married(tom).

Now consider the atom *married(bob)*. If we regard the logic program as having full information about the predicate *married* then we would feel entitled to say that Bob was not married, and that we should infer $\sim$*married(bob)*. On the other hand it may just be the case that the information is incomplete, and we do not have *any* information about Bob, and therefore cannot infer anything.

In many contexts however it is natural to suppose that we do in fact have complete information, and that information not explicitly or implicitly present in the logic program is actually false. This assumption is given the name *closed world assumption*, and when it seems appropriate to make the closed world assumption, we have an additional inference rule

Definition. 8.1 *Given a logic program P, and an atom $A \in B_P$. Under the closed world assumption we deduce $\sim A$ provided that A is not a logical consequence of P.*

This is a very straightforward rule in theory. Given any logic program P we have the set of all ground atoms B_P. Now $M_P \subseteq B_P$ is the least Herbrand model, and consists of all the atoms that are logical consequences of P. The remainder $B_P \backslash M_P$ consists of all the atoms that are not logical consequences of P and under the closed world assumption we take those to be false.

So, given a logic program P and an atom $A \in B_P$, in order to infer $\sim A$, all that is necessary is to see whether A is a logical consequence of P or not. Unfortunately, even in theory this is an impossible task, in that logic programming is asymmetric. We have seen that SLD-resolution is sound and complete, in that there is a refutation for atom that *is* in the least Herbrand model M_P. However here we are asking that we should be able to detect that an atom is *not* in M_P. This is not in general possible, because it may be the case that the SLD-tree has infinite branches, and they can never be exhaustively searched.

8.1 Finite failure

In the last section we indicated that in general we cannot be certain that an atom is not in M_P. However this can be ascertained for certain atoms.

Definition. 8.2 *A finitely-failed SLD-tree for $P \cup \{\leftarrow A\}$ is a finite SLD-tree that has no success branches.*

If there is a finitely failed SLD-tree for $P \cup \{\leftarrow A\}$ then by the completeness of SLD-resolution, we can be certain that $A \notin M_P$. This leads us to the following definition

Definition. 8.3 *Let P be a logic program. Then the* SLD finite failure *set F_P of P is the atoms $A \in B_P$ such that $P \cup \{\leftarrow A\}$ has a finitely failed SLD-tree.*

The set F_P is the procedural counterpart of the set $B_P \backslash M_P$. To see this consider an atom A. We give the system the goal $\sim A$. If $A \in M_P$ then there is an SLD-refutation and we can conclude A. If $A \notin M_P$, but $A \in F_P$ then there is a finitely failed SLD-tree, and we can conclude $\sim A$. However there is a grey area, containing the atoms in $B_P \backslash (F_P \cup M_P)$. An atom A in this set is not a logical consequence, but the SLD-tree has some infinite branches and no matter how long the search, we will never be able to detect whether there is a refutation that has not yet been found, or whether there really is no refutation.

This leads us to the rule commonly used in Prolog systems: the negation as failure rule. This is easily implemented by reversing the notions of success and failure. To try to prove the goal $\leftarrow \sim A$, the Prolog system tries to prove the goal $\leftarrow A$. If it succeeds in proving the goal $\leftarrow A$, then the goal $\leftarrow \sim A$ fails, and if it fails (finitely) to prove $\leftarrow A$, then the goal $\leftarrow \sim A$ succeeds.

8.2 Negation in Prolog

Finite failure is implemented in Prolog as described above by using the symbol `not` in front of the predicate in question.

Example. 8.4 The following is a very simple example of using `not` in Prolog.

```
married(jim).
married(tim).

| ?- not married(bob).

yes
```

■

So, in the above example what Prolog actually tried to do was to see whether *married(bob)* was a logical consequence of the program, and then simply to reverse its answer by exchanging `no` for `yes`.

It is extremely important to be very careful indeed about using negation in Prolog because it can have some unexpected effects. Firstly it is crucial even in theory to decide whether the closed-world assumption is actually valid for the problem at hand. For example the *member* predicate that we used earlier has complete information about the properties of membership. We can be pretty certain that if Prolog cannot prove that something is a member of a list, then in fact it is not a member of that list.

```
| ?- not member(a,[a,b,c]).

no
| ?- not member(e,[a,b,c]).

yes
```

However, one must still be careful. Suppose there are no clauses at all defining the predicate *member* and one tries to use it.

```
| ?- not member(a,[a,b,c]).

yes
```

Then because Prolog cannot prove *member(a,[a,b,c])* (because the system has no clauses) it will erroneously return `yes`. Of course such errors are the responsibility of the programmer, but using the `not` is a very easy way to introduce such errors.

Even more subtle errors are introduced when one uses `not` with predicates with uninstantiated variables.

Suppose there is a single clause in the database which is

```
pet(cat)
```

Now consider the following queries

```
| ?- not pet(lion).

yes
| ?- not pet(X).

no
```

Now the first question was answered correctly, if Prolog was not told that a lion is a pet then it should report that it is not a pet. However it is not clear whether the second question even has any meaning!

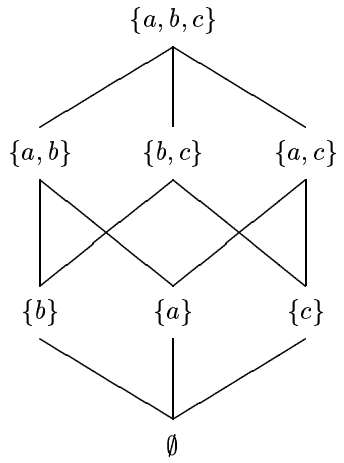

Figure 6: The Hasse diagram for $2^{\{a,b,c\}}$

9 Fixed point modelling

In this section we will be looking back at the concepts developed so far from a very different point of view. This will be a very mathematical and formal way of considering these ideas, and is quite an advanced excursion into very theoretical computer science.

We shall need the concept of a lattice of subsets of a set, which in turn requires the concept of a *partial order*.

Definition. 9.1 *A* partially ordered set *or* poset *P is a set on which a binary relation called "less than" and denoted $\preceq$ has been defined, and which satisfies the following axioms:*

1. $\forall x \in P,\ x \preceq x$
2. $\forall x, y \in P,\ x \preceq y$ *and* $y \preceq x$ *implies that* $x = y$
3. $\forall x, y, z \in P,\ x \preceq y$, *and* $y \preceq z$ *implies that* $x \preceq z$

There are many partial orders, but here we shall only consider the special case of subsets of a set.

Definition. 9.2 *The* subset lattice 2^X *of a set X is a partial order P where the elements of P are the subsets of X, and the partial order is given by $A \preceq B$ if and only if $A \subseteq B$.*

Example. 9.3 *Let $X = \{a, b, c\}$. Then the subset lattice of X consists of the elements $\emptyset$, $\{a\}$, $\{b\}$, $\{c\}$, $\{a, b\}$, $\{b, c\}$, $\{a, c\}$, $\{a, b, c\}$. Examples of the partial order are $\{a\} \preceq \{a, b\}$, $\{a, b\} \preceq \{a, b, c\}$. Notice that $\{a\}$ and $\{b, c\}$ are* incomparable *— that is neither one is less than the other.*

All partial orders P can be described by a *Hasse* diagram in which the points of P are drawn on a page, in such a way that every comparable pair $x \preceq y$ are drawn such that x is lower than y. Then lines are drawn between comparable pairs, provided the lines cannot be inferred already by the transitivity axiom. The Hasse diagram for the subset lattice above is shown in Figure 6.

Definition. 9.4 *Let X be a set of elements in a poset P. Then an* upper bound *for X is an element $a \in P$ such that $x \preceq a$ for all $x \in X$. A* lower bound *for X is an element $a \in P$ such that $a \preceq x$ for all $x \in X$.*

Example. 9.5 In the subset lattice for $\{a, b, c\}$ consider the set of elements $X = \{\{a\}, \{a, b\}\}$. Then an upper bound for X is $\{a, b\}$; another upper bound for X is $\{a, b, c\}$. ■

Definition. 9.6 *Let X be a set of elements in a poset P. Then a* least upper bound *for X is an element $a \in P$ such that*

1. *a is an upper bound for X*
2. *For any other upper bound b for X we have $a \preceq b$*

A greatest lower bound *for X is an element $a \in P$ such that*

1. *a is a lower bound for X*
2. *For any other lower bound b for X we have $b \preceq a$*

If a set X has a least upper bound, then it is unique and denoted $lub(X)$, and if it has a greatest lower bound then it is unique and denoted $glb(X)$.

Example. 9.7 In the subset lattice for $\{a, b, c\}$ consider the set of elements $X = \{\{a\}, \{a, b\}\}$. The least upper upper bound for X is $\{a, b\}$, and $glb(X) = \{a\}$. ■

Definition. 9.8 *A poset P is called a* complete lattice *if every subset of P has a least upper bound and a greatest lower bound.*

We have pre-empted this definition somewhat by referring to the poset $2^{\{}a, b, c\}$ as the subset lattice of $\{a, b, c\}$. It is easy to check that the subsets of any set do in fact form a complete lattice. The least upper bound of a collection of subsets is their union, and the greatest lower bound of a collection of subsets is their intersection. There is an extensive theory of lattices (a good introduction may be found in [3]), but we shall be content to consider only the subset lattice situation.

Lemma. 9.9 *Let L be a complete lattice. Then L contains an element $\top$, called the* top *of the lattice such that $x \preceq \top$ for all $x \in L$, and and element $\bot$, called the* bottom *of the lattice such that $\bot \preceq x$ for all $x \in L$.*

Example. 9.10 For the subset lattice of $\{a, b, c\}$, we have $\top = \{a, b, c\}$ and $\bot = \emptyset$. In fact, in general for the subset lattice 2^X we have $\top = X$ and $\bot = \emptyset$. ■

Now, let us see how this theory relates to logic programming. Let P be a logic program and let B_P be the Herbrand base of this program. We have already observed that any interpretation of P can be identified with a subset of B_P. Therefore 2^{B_P}, the subset lattice of B_P is a lattice containing all possible interpretations of P. Now, some of these interpretations are models for P, so we can identify some of the elements of the lattice as being models. In particular one element of the lattice is the least Herbrand model M_P. It is sometimes helpful to take a more abstract view of logic programming in the context of this lattice. To do this we use the concept of a logic program P corresponding to a certain mapping on the lattice.

Definition. 9.11 *Let L be a complete lattice and let $T : L \to L$ be a mapping. Then T is called* monotonic *if $T(x) \preceq T(y)$ whenever $x \preceq y$.*

Now, given a program P we shall define a mapping T_P on the subset lattice of B_P.

Definition. 9.12 *Let P be a logic program. Define $T_P : 2^{B_P} \to 2^{B_P}$ as follows. Let $\mathcal{I}$ be an interpretation (that is, an element of the lattice) Then $T_P(\mathcal{I}) = \{A \mid A \leftarrow A_1, A_2, \ldots, A_n\}$ is a ground instance of a clause in P and $\{A_1, A_2, \ldots, A_n\} \in \mathcal{I}$.*

Example. 9.13 Consider the logic program

p(X,Y) ←*q(X),q(Y)*
q(b)
p(a,a)

Then the Herbrand base $B_P = \{p(a,a), p(a,b), p(b,a), p(b,b), q(a), q(b)\}$. So the lattice 2^{B_P} consists of all the 64 subsets of B_P. Let us choose one such subset and apply the mapping T_P. Suppose we take the subset $\mathcal{I}_1 = \{q(b)\}$. Then consider $\mathcal{I}_2 = T_P(\{q(b)\})$. It certainly contains $q(b)$ and $p(a,a)$ because they are ground instances of clauses with empty bodies. Also, the clause $p(X,Y) \leftarrow q(X), q(Y)$ has a ground instance $p(b,b) \leftarrow q(b), q(b)$ and $q(b) \in \mathcal{I}_1$. Thus $\mathcal{I}_2 = T_P(\mathcal{I}_1) = \{q(b), p(a,a), p(b,b)\}$. Now consider $T_P(\mathcal{I}_2)$. Using the same arguments as previously we get that $T_P(\mathcal{I}_2) = \mathcal{I}_2$, that is, $\mathcal{I}_2$ is a fixed point. In fact $\mathcal{I}_2$ is also the least Herbrand model for this program, the proof of this is left as an exercise. ■

We now consider the consequences of applying the map T_P repeatedly starting with the bottom of the subset lattice.

Definition. 9.14 *Let $T : L \to L$ be a monotonic mapping of a complete lattice. Then define*

$T \uparrow 0 = \bot$
$T \uparrow n = T(T \uparrow (n-1))\ \forall n \in N$
$T \uparrow \omega = lub(\{T \uparrow n \mid n \in N\})$

Therefore $T \uparrow 0$ is simply the bottom of the lattice, $T \uparrow 1 = T(T \uparrow 0)$, $T \uparrow 2 = T(T(T \uparrow 0))$, ldots.

Hence for a program P we see that $T_P \uparrow 0 = \emptyset$. Then $T_P \uparrow 1$ is the set of all atoms that are ground instances of clauses with no body. In the language of forward chaining inference, $T_P \uparrow 1$ is all the atoms that can be produced with a chain of length 1. Then $T_P \uparrow 2$ is the set of all atoms that can be deduced with a forward chain of length 1 or 2. Notice that $T_P \uparrow n \subseteq T_P \uparrow (n+1)$. We can say something more explicit

Theorem. 9.15 *Let P be a program and G a goal $\leftarrow A_1, A_2, \ldots, A_m$. Then if $P \cup \{G\}$ has a refutation of length n with mgus θ_1, θ_2, ..., θ_n then every ground instance of $A_i\theta_1\theta_2\ldots\theta_n$ is in $T_P \uparrow n$ for all i.*

PROOF. See Lloyd (1987) Theorem 7.4 ■

That the mapping T_P is intimately connected with models for a program is shown by this theorem that demonstrates that models can be characterised using T_P.

Theorem. 9.16 *Let P be a logic program, and I be a Herbrand interpretation. Then $\mathcal{I}$ is a model for P if and only if $T_P(\mathcal{I}) \subseteq \mathcal{I}$.*

PROOF. Let $\mathcal{I}$ be a model for P. Now let us consider $T_P(\mathcal{I})$. If $A \in T_P(\mathcal{I})$ then there must be a clause in P with a ground instance $A \leftarrow A_1, A_2, \ldots, A_n$ and also $\{A_1, A_2, \ldots, A_n\} \subseteq \mathcal{I}$. But then because $\mathcal{I}$ is a model, we must also have $A \in \mathcal{I}$. Thus $T_P(\mathcal{I}) \subseteq \mathcal{I}$. Now to prove the converse, suppose that $T_P(\mathcal{I}) \subseteq \mathcal{I}$. We must show that $\mathcal{I}$ is a model for P. Consider a clause in P, say $A \leftarrow A_1, A_2, \ldots, A_n$. We must show that it is true under every variable assignment, so consider any specific variable assignment. If any of the A_i are false then the clause holds, so we need only consider the situation where every A_i is true. Then however $A \in T_P(\mathcal{I})$ by the definition of T_P, and because $T_P(\mathcal{I}) \subseteq \mathcal{I}$, we have $A \in \mathcal{I}$. Therefore the clause $A \leftarrow A_1, A_2, \ldots, A_n$ is true, and hence $\mathcal{I}$ is a model for P. ■

We can also observe that if $\mathcal{I}$ is a model, then so is $T_P(\mathcal{I})$.

Theorem. 9.17 *Let P be a logic program. Then if $\mathcal{I}$ is a Herbrand model for P, then so is $T_P(\mathcal{I})$.*

PROOF. We must show that every clause in P is true under the interpretation $T_P(\mathcal{I})$. Consider any particular ground instance of the clause $A \leftarrow A_1, A_2, \ldots, A_n$. If any of the A_i are false, then the clause is true, so we need only concern ourselves with the case where the A_i are all in $T_P(\mathcal{I})$. Then however they are in $\mathcal{I}$ because $T_P(\mathcal{I}) \subseteq \mathcal{I}$ and hence $A \in T_P(\mathcal{I})$, and hence the clause is true under $T_P(\mathcal{I})$. ■

We can now give the fixed point characterisation of the least Herbrand model

Definition. 9.18 *Let L be a complete lattice and let T be a monotonic mapping Then a* fixed point *of L is a point $x \in L$ such that $T(x) = x$. A* least fixed point *and* greatest fixed point *are defined in the natural way.*

Theorem. 9.19 *Let P be a logic program. Then the least Herbrand model M_P is the least fixed point of the mapping $T_P : 2^{B_P} \rightarrow 2^{B_P}$.*

PROOF. First we observe that M_P is a fixed point. As M_P is a model we have that $T_P(M_P) \subseteq M_P$, and because M_P is the least model we have $M_P \subseteq T_P(M_P)$. Therefore M_P is a fixed point of T_P. Now consider any other fixed point. Then it too is a model and hence contains M_P. Thus M_P is the least fixed point of T_P. ■

Finally we can give another characterisation of the least Herbrand model

Theorem. 9.20 *Let P be a logic program. Then the least Herbrand model $M_P = T_P \uparrow \omega$.*

PROOF. We shall demonstrate that $T_P \uparrow \omega$ is a fixed point. Then because $\bot \preceq lfp(T_P)$ it is clear that $T_P \uparrow \omega \preceq lfp(T_P)$ and the result will then follow. So, to demonstrate that $T_P \uparrow \omega$ is a fixed point we consider $T_P(T_P \uparrow \omega)$. Let $A \in T_P(T_P \uparrow \omega)$. Then there is a ground instance of a clause $A \leftarrow A_1, A_2, \ldots, A_n$, such that $\{A_1, A_2, \ldots, A_n\} \subseteq T_P \uparrow \omega$. Now

$$T_P \uparrow \omega = \bigcup_{n \in N} T_P \uparrow n$$

and thus there is some specific k such that $\{A_1, A_2, \ldots, A_n\} \subseteq T_P \uparrow k$. Then $A \in T_P \uparrow (k+1)$ and hence $A \in T_P \uparrow \omega$. Therefore we have shown that anything in $T_P(T_P \uparrow \omega)$ is in $T_P \uparrow \omega$ and thus $T_P(T_P \uparrow \omega) \subseteq T_P \uparrow \omega$ and by the monotonicity of T_P they are actually equal. Hence $T_P \uparrow \omega$ is the least fixed point of T_P and is thus equal to M_P. ■

These concepts now enable us to prove a result that we only stated earlier, the completeness of SLD-resolution.

Theorem. 9.21 *(Completeness of SLD-resolution) The success set of a logic program P is equal to its least Herbrand model M_P.*

PROOF. Let $A \in T_P \uparrow k$. Then we shall prove by induction on k that $P \cup \{ \sim A\}$ has a refutation. Suppose first that $k = 1$. Then there is a ground instance of a clause $A \leftarrow$ and there is an SLD-refutation of length 1. Now suppose that the result holds for $T_P \uparrow k$, and consider $T_P \uparrow (k+1)$. Suppose that $A \in T_P \uparrow (k+1)$. Then there is a ground instance of a clause $A \leftarrow A_1, A_2, \ldots, A_n$ such that $\{A_1, A_2, \ldots, A_n\} \subseteq T_P \uparrow k$. Therefore by the induction hypothesis there are refutations for $P \cup \{ \sim A_i\}$ for each i. As each of the A_i is ground we can find an SLD-refutation for $P \cup \{ \sim A\}$ simply by resolving immediately with the clause $A \leftarrow A_1, A_2, \ldots, A_n$ and then combining the refutations of all the atoms A_i. ■

10 Non-logical constructs in Prolog

10.1 Cuts in Prolog

One of the fundamental problems with Prolog is that it follows the fixed left-most first, depth-first search approach to searching the SLD-tree, and therefore all too frequently gets caught down infinite branches. As we have discussed previously however, these problems are not easy to rectify — although the computation rule does change the SLD-tree no particular rule is generally better than another, and it seems difficult to dynamically vary the computation rule according to which particular goal is being refuted. The depth-first nature of Prolog's search is essentially forced upon us by considerations of efficiency. Breadth-first search places huge demands on the amount of storage required, and is more difficult to implement. However, one possible area of research would be to try and find some way of identifying infinite branches, and then backtracking immediately in that situation.

In many situations however, the programmer can see in advance that certain clauses are going to cause difficulties. For these situations Prolog provides a *control facility* called the *cut* and written !. The cut provides a method of pruning certain branches from the SLD-tree; that is, the cut prevents Prolog from examining certain sub-trees of the SLD-tree. If the programmer knows in advance that the pruned sub-tree contains no success branches, or is infinite, then clearly the cut can be used to great effect. However, it must be remembered that the cut is *not* a logical feature — it affects the control of the logic programming system, and therefore can only be interpreted procedurally rather than logically. This is contrary to the spirit of logic programming and therefore use of the cut is somewhat controversial and like *goto* statements in Pascal or C, a program is considered aesthetically more pleasing if it does not use the cut.

Now we shall consider the precise behaviour of the cut. It is used in a program exactly as though it were a goal. Define the *parent goal* to be the goal in which the selected atom matched the head of the clause containing the cut. The behaviour of Prolog on encountering the goal ! can be described in the following somewhat cryptic sentence

> The goal ! succeeds and commits Prolog to all the choices made since the parent goal was unified with the head of the clause containing the cut.

An alternative phrasing would be to say that as a goal the cut immediately succeeds, but that if the system backtracks and attempts to resatisfy the cut, then Prolog prunes the entire subtree with the parent goal as the root.

An example is the best way to make this clear. Consider the following logic program.

$A \leftarrow B, C$

$\vdots$

$B \leftarrow D,\ !,\ E$

$\vdots$

$D \leftarrow$

$\vdots$

Now consider the SLD-tree for the goal $\leftarrow A$ (you should draw this tree to understand this).

Notice that $\leftarrow B, C$ is the parent goal here, as this is the goal that unified with the head of the clause $B \leftarrow D, !, E$. After this unification the goal is $\leftarrow D, !, E, C$. The selected atom D unifies and the goal becomes $\leftarrow !, E, C$. Now the cut immediately succeeds, leaving the goal $\leftarrow E, C$. *but commits Prolog to all the choices made since the parent goal was unified.* Now suppose that the goal E eventually fails, and the system backtracks to the cut. Then the system does *not* try to resatisfy D, and it does *not* try to choose another clause to unify with B instead it prunes the entire subtree rooted at $\leftarrow B, C$.

Therefore even if there are other clauses with head B, passing through the cut ensured that Prolog was committed to using that particular clause. Thus the backtracking resumes with at the point of trying to resatisfy the goal $\leftarrow A$.

So, the cut may be used to prune the SLD-tree. Clearly, problems can arise if the pruned subtree contains success branches, a cut used in such a situation is said to be *unsafe*. Safe cuts reduce the search space, possibly making otherwise infeasible problems possible.

One of the main places to use a cut is the situation where there are two or more clauses for a predicate, but only one of them can possibly be satisfied. For example consider the following partial Prolog program for determining if a term is a polynomial in x.

```
polynomial(x,x).
polynomial(Term,x) :- constant(Term).
polynomial(Term1+Term2,x) :- polynomial(Term1,x), polynomial(Term2,x).
polynomial(Term1-Term2,x) :- polynomial(Term1,x), polynomial(Term2,x).
polynomial(Term1*Term2,x) :- polynomial(Term1,x), polynomial(Term2,x).
```

Assuming that *constant* is defined appropriately the program will perform in the following way.

```
?- polynomial(x*x*x+2*x*x+3,x).

yes
```

Now, observe that any given expression will only match at most one of the clauses. If the clause it matches fails, then backtracking will occur as Prolog uses its unification mechanism to try and match one of the other clauses — a procedure that we *know* cannot succeed. Therefore we can improve efficiency by placing cuts so that once a particular clause matches no others will be tried in the event of failure.

```
polynomial(x,x) :- !.
polynomial(Term,x) :- constant(Term), !.
polynomial(Term1+Term2,x) :- !, polynomial(Term1,x), polynomial(Term2,x).
polynomial(Term1-Term2,x) :- !, polynomial(Term1,x), polynomial(Term2,x).
polynomial(Term1*Term2,x) :- !, polynomial(Term1,x), polynomial(Term2,x).
```

It is often tempting to use the cut to write "procedural" programs. For example consider the following program for minimum.

```
minimum(X,Y,X) :- X <= Y, !.
minimum(X,Y,Y).
```

If the program is used in the following fashion it produces correct answers

```
?- minimum(2,5,Q).

Q = 2.
```

The procedural reasoning behind this is as follows: if $X \leq Y$ then the first clause will succeed and Q will be correctly instantiated to X. Then the cut will prevent the other clause being examined. Therefore if the other clause *is* examined then it must be the case that $X > Y$ and hence we do not need to waste another comparison trying to check this.

However this reasoning is completely contrary to the spirit of logic programming, because the declarative meaning of the above program is *not* the correct meaning of *minimum*. For example

```
?- minimum(2,5,5).

yes
```

Programmers should attempt to use the cut only to prune subtrees known to have no success branches, and not in this 'procedural' way.

10.2 Set-of constructs in Prolog

References

[1] Bratko, I. *PROLOG Programming for Artificial Intelligence*, Addison-Wesley, 1990.

[2] Clocksin, W.F and Mellish, C.S. *Programming in Prolog* Springer-Verlag, 1981.

[3] Davey, B and Priestly, H. *Introduction to lattices and order* 1990.

[4] LLoyd, J.W. *Foundations of Logic Programming*, Springer-Verlag, 1987.

[5] Maier and Warren *Computing with Logic*, Benjamin-Cummings. 1988.

[6] Sterling, L and Shapiro, E. *The Art of Prolog*, MIT Press. 1986.