Primers - Lecture notes 1

Artificial Intelligence (Imperial College London)

Institute for Logic,
Language and Computation

Lecture Notes
An Introduction to Prolog Programming

Ulle Endriss

Universiteit van Amsterdam

Version: 1 September 2014

Preface

These lecture notes introduce the declarative programming language Prolog. The emphasis is on learning how to program, rather than on the theory of logic programming. Nevertheless, a short chapter on the logic foundations of Prolog is included as well.

All examples have been tested using SWI-Prolog (`www.swi-prolog.org`) and can be expected to work equally well with most other Prolog systems. These notes have originally been developed for a course I taught at King's College London in 1999 and 2000.

Amsterdam, August 2005 U.E.

The present version corrects a number of minor errors in the text, most of which have been pointed out to me by students following a number of courses I have given at the University of Amsterdam since 2005.

Amsterdam, September 2014 U.E.

Contents

Chapter 1

The Basics

Prolog (*pro*gramming in *log*ic) is one of the most widely used programming languages in artificial intelligence research. As opposed to imperative languages such as C or Java (the latter of which also happens to be object-oriented) it is a *declarative* programming language. That means, when implementing the solution to a problem, instead of specifying *how* to achieve a certain goal in a certain situation, we specify *what* the situation (*rules* and *facts*) and the goal (*query*) are and let the Prolog interpreter derive the solution for us. Prolog is very useful in *some* problem areas, such as artificial intelligence, natural language processing, databases, ..., but pretty useless in others, such as graphics or numerical algorithms.

By following this course, you will learn how to use Prolog as a programming language to solve practical problems in computer science and artificial intelligence. You will also learn how the Prolog interpreter actually works. The latter will include an introduction to the logical foundations of the Prolog language.

These notes cover the most important Prolog concepts you need to know about, but it is certainly worthwhile to also have a look at the literature. The following three are well-known titles, but you may also consult any other textbook on Prolog.

- I. Bratko. *Prolog Programming for Artificial Intelligence.* 4th edition, Addison-Wesley Publishers, 2012.
- F. W. Clocksin and C. S. Mellish. *Programming in Prolog.* 5th edition, Springer-Verlag, 2003.
- L. Sterling and E. Shapiro. *The Art of Prolog.* 2nd edition, MIT Press, 1994.

1.1 Getting Started: An Example

In the introduction it has been said that Prolog is a declarative (or descriptive) language. Programming in Prolog means describing the world. Using such programs means asking Prolog questions about the previously described world. The simplest way of describing the world is by stating *facts*, like this one:

```
bigger(elephant, horse).
```

This states, quite intuitively, the fact that an elephant is bigger than a horse. (Whether the world described by a Prolog program has anything to do with our real world is, of course, entirely up to the programmer.) Let's add a few more facts to our little program:

```
bigger(elephant, horse).
bigger(horse, donkey).
bigger(donkey, dog).
bigger(donkey, monkey).
```

This is a syntactically correct program, and after having compiled it we can ask the Prolog system questions (or *queries* in proper Prolog-jargon) about it. Here's an example:

```
?- bigger(donkey, dog).
Yes
```

The query `bigger(donkey, dog)` (i.e., the question "Is a donkey bigger than a dog?") succeeds, because the fact `bigger(donkey, dog)` has previously been communicated to the Prolog system. Now, is a monkey bigger than an elephant?

```
?- bigger(monkey, elephant).
No
```

No, it's not. We get exactly the answer we expected: the corresponding query, namely `bigger(monkey, elephant)` fails. But what happens when we ask the other way round?

```
?- bigger(elephant, monkey).
No
```

According to this elephants are not bigger than monkeys. This is clearly wrong as far as our real world is concerned, but if you check our little program again, you will find that it says nothing about the relationship between elephants and monkeys. Still, we know that if elephants are bigger than horses, which in turn are bigger than donkeys, which in turn are bigger than monkeys, then elephants also have to be bigger than monkeys. In mathematical terms: the bigger-relation is transitive. But this has also not been defined in our program. The correct interpretation of the negative answer Prolog has given is the following: from the information communicated to the system *it cannot be proved* that an elephant is bigger than a monkey.

If, however, we would like to get a positive reply for a query like `bigger(elephant, monkey)`, we have to provide a more accurate description of the world. One way of doing this would be to add the remaining facts, such as `bigger(elephant, monkey)`, to our program. For our little example this would mean adding another 5 facts. Clearly too much work and probably not too clever anyway.

The far better solution would be to define a new relation, which we will call `is_bigger`, as the transitive closure (don't worry if you don't know what that means)

of `bigger`. Animal `X` is bigger than animal `Y` either if this has been stated as a fact or if there is an animal `Z` for which it has been stated as a fact that animal `X` is bigger than animal `Z` and it can be shown that animal `Z` is bigger than animal `Y`. In Prolog such statements are called *rules* and are implemented like this:

```
is_bigger(X, Y) :- bigger(X, Y).
is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y).
```

In these rules `:-` means something like "if" and the comma between the two terms `bigger(X, Z)` and `is_bigger(Z, Y)` stands for "and". `X`, `Y`, and `Z` are variables, which in Prolog is indicated by using capital letters.

You can think of the the `bigger`-facts as data someone has collected by browsing through the local zoo and comparing pairs of animals. The implementation of `is_bigger`, on the other hand, could have been provided by a knowledge engineer who may not know anything at all about animals, but understands the general concept of something being bigger than something else and thereby has the ability to formulate general rules regarding this relation. If from now on we use `is_bigger` instead of `bigger` in our queries, the program will work as intended:

```
?- is_bigger(elephant, monkey).
Yes
```

Prolog still cannot find the fact `bigger(elephant, monkey)` in its database, so it tries to use the second rule instead. This is done by *matching* the query with the head of the rule, which is `is_bigger(X, Y)`. When doing so the two variables get instantiated: `X = elephant` and `Y = monkey`. The rule says that in order to prove the *goal* `is_bigger(X, Y)` (with the variable instantiations that's equivalent to `is_bigger(elephant, monkey)`) Prolog has to prove the two *subgoals* `bigger(X, Z)` and `is_bigger(Z, Y)`, again with the same variable instantiations. This process is repeated recursively until the facts that make up the chain between `elephant` and `monkey` are found and the query finally succeeds. How this goal execution as well as term matching and variable instantiation really work will be examined in more detail in Section 1.3.

Of course, we can do slightly more exiting stuff than just asking yes/no-questions. Suppose we want to know, *what* animals are bigger than a donkey? The corresponding query would be:

```
?- is_bigger(X, donkey).
```

Again, `X` is a variable. We could also have chosen any other name for it, as long as it starts with a capital letter. The Prolog interpreter replies as follows:

```
?- is_bigger(X, donkey).
X = horse
```

Horses are bigger than donkeys. The query has succeeded, but in order to allow it to succeed Prolog had to instantiate the variable `X` with the value `horse`. If this makes us

happy already, we can press Return now and that's it. In case we want to find out if there are more animals that are bigger than the donkey, we can press the semicolon key, which will cause Prolog to search for alternative solutions to our query. If we do this once, we get the next solution `X = elephant`: elephants are also bigger than donkeys. Pressing semicolon again will return a `No`, because there are no more solutions:

```
?- is_bigger(X, donkey).
X = horse ;
X = elephant ;
No
```

There are many more ways of querying the Prolog system about the contents of its database. As a final example we ask whether there is an animal `X` that is both smaller than a donkey *and* bigger than a monkey:

```
?- is_bigger(donkey, X), is_bigger(X, monkey).
No
```

The (correct) answer is `No`. Even though the two single queries `is_bigger(donkey, X)` and `is_bigger(X, monkey)` would both succeed when submitted on their own, their conjunction (represented by the comma) does not.

This section has been intended to give you a first impression of Prolog programming. The next section provides a more systematic overview of the basic syntax.

There are a number of Prolog systems around that you can use. How to start a Prolog session may differ slightly from one system to the next, but it should not be too difficult to find out by consulting the user manual of your system. The examples in these notes have all been generated using SWI-Prolog (in its 1999 incarnation, with only a few minor adjustments made later on).[1]

1.2 Prolog Syntax

This section describes the most basic features of the Prolog programming language.

1.2.1 Terms

The central data structure in Prolog is that of a *term*. There are terms of four kinds: *atoms*, *numbers*, *variables*, and *compound terms*. Atoms and numbers are sometimes grouped together and called *atomic terms*.

[1] One difference between "classical" Prolog and more recent versions of SWI-Prolog is that the latter reports `true` rather than `Yes` when a query succeeds and `false` rather than `No` when a query fails. There also a a few other very minor differences in how modern SWI-Prolog responds to queries. If you are interested in the finer subtleties of this matter, search the Internet for "Prolog toplevel".

Atoms. Atoms are usually strings made up of lower- and uppercase letters, digits, and the underscore, *starting with a lowercase letter*. The following are all valid Prolog atoms:

```
elephant, b, abcXYZ, x_123, another_pint_for_me_please
```

On top of that also any series of arbitrary characters enclosed in single quotes denotes an atom.

```
'This is also a Prolog atom.'
```

Finally, strings made up solely of special characters like `+ - * = < > : &` (check the manual of your Prolog system for the exact set of these characters) are also atoms. Examples:

```
+, ::, <------>, ***
```

Numbers. All Prolog implementations have an integer type: a sequence of digits, optionally preceded by a `-` (minus). Some also support floats. Check the manual for details.

Variables. Variables are strings of letters, digits, and the underscore, *starting with a capital letter or an underscore*. Examples:

```
X, Elephant, _4711, X_1_2, MyVariable, _
```

The last one of the above examples (the single underscore) constitutes a special case. It is called the *anonymous variable* and is used when the value of a variable is of no particular interest. Multiple occurrences of the anonymous variable in one expression are assumed to be distinct, i.e., their values don't necessarily have to be the same. More on this later.

Compound terms. Compound terms are made up of a *functor* (a Prolog atom) and a number of *arguments* (Prolog terms, i.e., atoms, numbers, variables, or other compound terms) enclosed in parentheses and separated by commas. The following are some examples for compound terms:

```
is_bigger(horse, X), f(g(X, _), 7), 'My Functor'(dog)
```

It's important not to put any blank characters between the functor and the opening parentheses, or Prolog won't understand what you're trying to say. In other places, however, spaces can be very helpful for making programs more readable.

The sets of compound terms and atoms together form the set of Prolog *predicates*. A term that doesn't contain any variables is called a *ground term*.

1.2.2 Clauses, Programs and Queries

In the introductory example we have already seen how Prolog programs are made up of *facts* and *rules*. Facts and rules are also called *clauses*.

Facts. A fact is a predicate followed by a full stop. Examples:

```
bigger(whale, _).
life_is_beautiful.
```

The intuitive meaning of a fact is that we define a certain instance of a relation as being true.

Rules. A rule consists of a *head* (a predicate) and a *body.* (a sequence of predicates separated by commas). Head and body are separated by the sign `:-` and, like every Prolog expression, a rule has to be terminated by a full stop. Examples:

```
is_smaller(X, Y) :- is_bigger(Y, X).
aunt(Aunt, Child) :-
  sister(Aunt, Parent),
  parent(Parent, Child).
```

The intuitive meaning of a rule is that the goal expressed by its head is true, if we (or rather the Prolog system) can show that all of the expressions (subgoals) in the rule's body are true.

Programs. A Prolog program is a sequence of clauses.

Queries. After compilation a Prolog program is run by submitting queries to the interpreter. A query has the same structure as the body of a rule, i.e., it is a sequence of predicates separated by commas and terminated by a full stop. They can be entered at the Prolog prompt, which in most implementations looks something like this: `?-`. When writing about queries we often include the `?-`. Examples:

```
?- is_bigger(elephant, donkey).
?- small(X), green(X), slimy(X).
```

Intuitively, when submitting a query like the last example, we ask Prolog whether all its predicates are provably true, or in other words whether there is an `X` such that `small(X)`, `green(X)`, and `slimy(X)` are all true.

1.2.3 Some Built-in Predicates

What we have seen so far is already enough to write simple programs by defining predicates in terms of facts and rules, but Prolog also provides a range of useful built-in predicates. Some of them will be introduced in this section; all of them should be explained in the user manual of your Prolog system.

Built-ins can be used in a similar way as user-defined predicates. The important difference between the two is that a built-in predicate is not allowed to appear as the principal functor in a fact or the head of a rule. This must be so, because using them in such a position would effectively mean changing their definition.

Equality. Maybe the most important built-in predicate is `=` (equality). Instead of writing expressions such as `=(X, Y)`, we usually write more conveniently `X = Y`. Such a goal succeeds, if the terms `X` and `Y` can be matched. This will be made more precise in Section 1.3.

Guaranteed success and certain failure. Sometimes it can be useful to have predicates that are known to either fail or succeed in any case. The predicates `fail` and `true` serve exactly this purpose. Some Prolog systems also provide the predicate `false`, with exactly the same functionality as `fail`.

Consulting program files. Program files can be compiled using the predicate `consult/1`.[2] The argument has to be a Prolog atom denoting the program file you want to compile. For example, to compile the file `big-animals.pl` submit the following query to Prolog:

```
?- consult('big-animals.pl').
```

If the compilation is successful, Prolog will reply with `Yes`. Otherwise a list of errors will be displayed.

Output. If besides Prolog's replies to queries you wish your program to have further output you can use the `write/1` predicate. The argument can be any valid Prolog term. In the case of a variable its value will get printed to the screen. Execution of the predicate `nl/0` causes the system to skip a line. Here are two examples:

```
?- write('Hello World!'), nl.
Hello World!
Yes

?- X = elephant, write(X), nl.
elephant
X = elephant
Yes
```

In the second example, first the variable `X` is bound to the atom `elephant` and then *the value of* `X`, i.e., `elephant`, is written on the screen using the `write/1` predicate. After skipping to a new line, Prolog reports the variable binding(s), i.e., `X = elephant`.

Checking the type of a Prolog term. There are a number of built-in predicates available that can be used to check the type of a given Prolog term. Here are some examples:

[2] The `/1` is used to indicate that this predicate takes one argument.

```
?- atom(elephant).
Yes

?- atom(Elephant).
No

?- X = f(mouse), compound(X).
X = f(mouse)
Yes
```

The last query succeeds, because the variable `X` is bound to the compound term `f(mouse)` at the time the subgoal `compound(X)` is being executed.

Help. Most Prolog systems also provide a help function in the shape of a predicate, usually called `help/1`. Applied to a term (like the name of a built-in predicate) the system will display a short description, if available. Example:

```
?- help(atom).
atom(+Term)
    Succeeds if Term is bound to an atom.
```

1.3 Answering Queries

We have mentioned the issue of term matching before in these notes. This concept is crucial to the way Prolog replies to queries, so we present it before describing what actually happens when a query is processed (or more generally speaking: when a goal is executed).

1.3.1 Matching

Two terms are said to *match* if they are either identical or if they can be made identical by means of variable instantiation. Instantiating a variable means assigning it a fixed value. Two free variables also match, because they could be instantiated with the same ground term.

It is important to note that the same variable has to be instantiated with the same value throughout an expression. The only exception to this rule is the *anonymous variable* `_`, which is considered to be unique whenever it occurs.

We give some examples. The terms `is_bigger(X, dog)` and `is_bigger(elephant, dog)` match, because the variable `X` can be instantiated with the atom `elephant`. We could test this in the Prolog interpreter by submitting the corresponding query to which Prolog would react by listing the appropriate variable instantiations:

```
?- is_bigger(X, dog) = is_bigger(elephant, dog).
```

```
X = elephant
Yes
```

The following is an example for a query that doesn't succeed, because `X` cannot match with `1` and `2` at the same time.

```
?- p(X, 2, 2) = p(1, Y, X).
No
```

If, however, instead of `X` we use the anonymous variable `_`, matching is possible, because every occurrence of `_` represents a distinct variable. During matching `Y` is instantiated with `2`:

```
?- p(_, 2, 2) = p(1, Y, _).
Y = 2
Yes
```

Another example for matching:

```
?- f(a, g(X, Y)) = f(X, Z), Z = g(W, h(X)).
X = a
Y = h(a)
Z = g(a, h(a))
W = a
Yes
```

So far so good. But what happens, if matching is possible even though no specific variable instantiation has to be enforced (like in all previous examples)? Consider the following query:

```
?- X = my_functor(Y).
X = my_functor(_G177)
Y = _G177
Yes
```

In this example matching succeeds, because `X` could be a compound term with the functor `my_functor` and a non-specified single argument. `Y` could be any valid Prolog term, but it has to be the same term as the argument inside `X`. In Prolog's output this is denoted through the use of the variable `_G177`. This variable has been generated by Prolog during execution time. Its particular name, `_G177` in this case, may be different every time the query is submitted.

In fact, what the output for the above example will look like exactly will depend on the Prolog system you use. For instance, some systems will avoid introducing a new variable (here `_G177`) and instead simply report the variable binding as `X = my_functor(Y)`.

1.3.2 Goal Execution

Submitting a query means asking Prolog to try to prove that the statement(s) implied by the query can be made true provided the right variable instantiations are made. The search for such a proof is usually referred to as *goal execution*. Each predicate in the query constitutes a (sub)goal, which Prolog tries to satisfy one after the other. If variables are shared between several subgoals their instantiations have to be the same throughout the entire expression.

If a goal matches with the head of a rule, the respective variable instantiations are made inside the rule's body, which then becomes the new goal to be satisfied. If the body consists of several predicates the goal is again split into subgoals to be executed in turn. In other words, the head of a rule is considered provably true, if the conjunction of all its body-predicates are provably true. If a goal matches with a fact in our program, the proof for that goal is complete and the variable instantiations made during matching are communicated back to the surface. Note that the order in which facts and rules appear in our program is important here. Prolog will always try to match its current goal with the first possible fact or rule-head it can find.

If the principal functor of a goal is a built-in predicate the associated action is executed whilst the goal is being satisfied. For example, as far as goal execution is concerned the predicate

```
write('Hello World!')
```

will simply succeed, but at the same time it will also print the words `Hello World!` on the screen.

As mentioned before, the built-in predicate `true` will always succeed (without any further side-effects), whereas `fail` will always fail.

Sometimes there is more than one way of satisfying the current goal. Prolog chooses the first possibility (as determined by the order of clauses in a program), but the fact that there are alternatives is recorded. If at some point Prolog fails to prove a certain subgoal, the system can go back and try an alternative way of executing the previous goal. This process is known as *backtracking*.

We shall exemplify the process of goal execution by means of the following famous argument:

All men are mortal.
Socrates is a man.

Hence, Socrates is mortal.

In Prolog terms, the first statement represents a rule: `X` is mortal, if `X` is a man (for all `X`). The second one constitutes a fact: Socrates is a man. This can be implemented in Prolog as follows:

```
mortal(X) :- man(X).
man(socrates).
```

Note that `X` is a variable, whereas `socrates` is an atom. The conclusion of the argument, "Socrates is mortal", can be expressed through the predicate `mortal(socrates)`. After having consulted the above program we can submit this predicate to Prolog as a query, which will cause the following reaction:

```
?- mortal(socrates).
Yes
```

Prolog agrees with our own logical reasoning. Which is nice. But how did it come to its conclusion? Let's follow the goal execution step by step.

(1) The query `mortal(socrates)` is made the initial goal.

(2) Scanning through the clauses of our program, Prolog tries to match `mortal(socrates)` with the first possible fact or head of rule. It finds `mortal(X)`, the head of the first (and only) rule. When matching the two terms the instantiation `X = socrates` needs to be made.

(3) The variable instantiation is extended to the body of the rule, i.e., `man(X)` becomes `man(socrates)`.

(4) The newly instantiated body becomes our new goal: `man(socrates)`.

(5) Prolog executes the new goal by again trying to match it with a rule-head or a fact. Obviously, the goal `man(socrates)` matches the fact `man(socrates)`, because they are identical. This means the current goal succeeds.

(6) This, again, means that also the initial goal succeeds.

1.4 A Matter of Style

One of the major advantages of Prolog is that it allows for writing very short and compact programs solving not only comparatively difficult problems, but also being readable and (again: comparatively) easy to understand.

Of course, this can only work, if the programmer (you!) pays some attention to his or her programming style. As with every programming language, comments *do* help. In Prolog comments are enclosed between the two signs `/*` and `*/`, like this:

```
/* This is a comment.  */
```

Comments that only run over a single line can also be started with the percentage sign `%`. This is usually used within a clause.

```
aunt(X, Z) :-
  sister(X, Y),   % A comment on this subgoal.
  parent(Y, Z).
```

Besides the use of comments a good layout can improve the readability of your programs significantly. The following are some basic rules most people seem to agree on:

(1) Separate clauses by one or more blank lines.

(2) Write only one predicate per line and use indentation:

```
blond(X) :-
  father(Father, X),
  blond(Father),
  mother(Mother, X),
  blond(Mother).
```

(Very short clauses may also be written in a single line.)

(3) Insert a space after every comma inside a compound term:

```
born(mary, yorkshire, '01/01/1995')
```

(4) Write short clauses with bodies consisting of only a few goals. If necessary, split into shorter sub-clauses.

(5) Choose meaningful names for your variables and atoms.

1.5 Exercises

Exercise 1.1. Try to answer the following questions first "by hand" and then verify your answers using a Prolog interpreter.

(a) Which of the following are valid Prolog atoms?

```
f, loves(john,mary), Mary, _c1, 'Hello', this_is_it
```

(b) Which of the following are valid names for Prolog variables?

```
a, A, Paul, 'Hello', a_123, _, _abc, x2
```

(c) What would a Prolog interpreter reply given the following query?

```
?- f(a, b) = f(X, Y).
```

(d) Would the following query succeed?

```
?- loves(mary, john) = loves(John, Mary).
```

Why?

(e) Assume a program consisting only of the fact

```
a(B, B).
```

has been consulted by Prolog. How will the system react to the following query?

```
?- a(1, X), a(X, Y), a(Y, Z), a(Z, 100).
```

Why?

Exercise 1.2. Read the section on matching again and try to understand what's happening when you submit the following queries to Prolog.

(a) `?- myFunctor(1, 2) = X, X = myFunctor(Y, Y).`

(b) `?- f(a, _, c, d) = f(a, X, Y, _).`

(c) `?- write('One '), X = write('Two ').`

Exercise 1.3. Draw the family tree corresponding to the following Prolog program:

```
female(mary).
female(sandra).
female(juliet).
female(lisa).
male(peter).
male(paul).
male(dick).
male(bob).
male(harry).
parent(bob, lisa).
parent(bob, paul).
parent(bob, mary).
parent(juliet, lisa).
parent(juliet, paul).
parent(juliet, mary).
parent(peter, harry).
parent(lisa, harry).
parent(mary, dick).
parent(mary, sandra).
```

After having copied the given program, define new predicates (in terms of rules using `male/1`, `female/1` and `parent/2`) for the following family relations:

(a) father

(b) sister

(c) grandmother

(d) cousin

You may want to use the operator `\=`, which is the opposite of `=`. A goal like `X \= Y` succeeds, if the two terms `X` and `Y` cannot be matched.

Example: `X` is the brother of `Y`, if they have a parent `Z` in common and if `X` is male and if `X` and `Y` don't represent the same person. In Prolog this can be expressed through the following rule:

```
brother(X, Y) :-
  parent(Z, X),
  parent(Z, Y),
  male(X),
  X \= Y.
```

Exercise 1.4. Most people will probably find all of this rather daunting at first. Read the chapter again in a few weeks' time when you will have gained some programming experience in Prolog and enjoy the feeling of enlightenment. The part on the syntax of the Prolog language and the stuff on matching and goal execution are particularly important.

Chapter 2

List Manipulation

This chapter introduces a special notation for lists, one of the most important data structures in Prolog, and provides some examples for how to work with them.

2.1 Notation

Lists are contained in square brackets with the elements being separated by commas. Here's an example:

```
[elephant, horse, donkey, dog]
```

This is the list of the four atoms `elephant`, `horse`, `donkey`, and `dog`. Elements of lists could be any valid Prolog terms, i.e., atoms, numbers, variables, or compound terms. This includes also other lists. The empty list is written as `[]`. The following is another example for a (slightly more complex) list:

```
[elephant, [], X, parent(X, tom), [a, b, c], f(22)]
```

Internal representation. Internally, lists are represented as compound terms using the functor `.` (dot). The empty list `[]` is an atom and elements are added one by one. The list `[a,b,c]`, for example, corresponds to the following term:

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

2.2 Head and Tail

The first element of a list is called its *head* and the remaining list is called the *tail*. An empty list doesn't have a head. A list just containing a single element has a head (namely that particular single element) and its tail is the empty list.

A variant of the list notation allows for convenient addressing of both head and tail of a list. This is done by using the separator `|` (bar). If it is put just before the last term inside a list, it means that that last term denotes another list. The entire list is

then constructed by appending this sub-list to the list represented by the sequence of elements before the bar. If there is exactly one element before the bar, it is the head and the term after the bar is the list's tail. In the next example, `1` is the head of the list and `[2,3,4,5]` is the tail, which has been computed by Prolog simply by matching the list of numbers with the head/tail-pattern.

```
?- [1, 2, 3, 4, 5] = [Head | Tail].
Head = 1
Tail = [2, 3, 4, 5]
Yes
```

Note that `Head` and `Tail` are just names for variables. We could have used `X` and `Y` or whatever instead with the same result. Note also that the tail of a list (more generally speaking: the thing after `|`) is always a list itself. Possibly the empty list, but definitely a list. The head, however, is an element of a list. It *could* be a list as well, but not necessarily (as you can see from the previous example—`1` is not a list). The same applies to all other elements listed before the bar in a list.

This notation also allows us to retrieve the, say, second element of a given list. In the following example we use the anonymous variable for the head and also for the list after the bar, because we are only interested in the second element.

```
?- [quod, licet, jovi, non, licet, bovi] = [_, X | _].
X = licet
Yes
```

The head/tail-pattern can be used to implement predicates over lists in a very compact and elegant way. We exemplify this by presenting an implementation of a predicate that can be used to concatenate two lists.[1] We call it `concat_lists/3`. When called with the first two elements being instantiated to lists, the third argument should be matched with the concatenation of those two lists, in other words we would like to get the following behaviour:

```
?- concat_lists([1, 2, 3], [d, e, f, g], X).
X = [1, 2, 3, d, e, f, g]
Yes
```

The general approach to such a problem is a recursive one. We start with a base case and then write a clause to reduce a complex problem to a simpler one until the base case is reached. For our particular problem, a suitable base case would be when one of the two input-lists (for example the first one) is the empty list. In that case the result (the third argument) is simply identical with the second list. This can be expressed through the following fact:

[1] Note that most Prolog systems already provide such a predicate, usually called `append/3` (see Section 2.3). So you do not actually have to implement this yourself.

```
concat_lists([], List, List).
```

In all other cases (i.e., in all cases where a query with `concat_lists` as the main functor doesn't match with this fact) the first list has at least one element. Hence, it can be written as a head/tail-pattern: `[Elem | List1]`. If the second list is associated with the variable `List2`, then we know that the head of the result should be `Elem` and the tail should be the concatenation of `List1` and `List2`. Note how this simplifies our initial problem: We take away the head of the first list and try to concatenate it with the (unchanged) second list. If we repeat this process recursively, we will eventually end up with an empty first list, which is exactly the base case that can be handled by the previously implemented fact. Turning this simplification algorithm into a Prolog rule is straightforward:

```
concat_lists([Elem | List1], List2, [Elem | List3]) :-
  concat_lists(List1, List2, List3).
```

And that's it! The predicate `concat_lists/3` can now be used for concatenating two given lists as specified. But it is actually much more flexible than that. If we call it with variables in the first two arguments and instantiate the third one with a list, `concat_lists/3` can be used to decompose that list. If you use the semicolon key to get all alternative solutions to your query, Prolog will print out all possibilities how the given list could be obtained from concatenating two lists.

```
?- concat_lists(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
```

Recall that the `No` at the end means that there are no further alternative solutions.

2.3 Some Built-in Predicates for List Manipulation

Prolog comes with a range of predefined predicates for manipulating lists. Some of the most important ones are presented here. Note that they could all easily be implemented by exploiting the head/tail-pattern.

`length/2`: The second argument is matched with the length of the list in the first argument. Example:

```
?- length([elephant, [], [1, 2, 3, 4]], Length).
Length = 3
Yes
```

It is also possible to use `length/2` with an uninstantiated first argument. This will generate a list of free variables of the specified length:

```
?- length(List, 3).
List = [_G248, _G251, _G254]
Yes
```

The *names* of those variables may well be different every time you call this query, because they are generated by Prolog during execution time.

`member/2`: The goal `member(Elem, List)` will succeed, if the term `Elem` can be matched with one of the members of the list `List`. Example:

```
?- member(dog, [elephant, horse, donkey, dog, monkey]).
Yes
```

`append/3`: Concatenate two lists. This built-in works exactly like the predicate `concat_lists/3` presented in Section 2.2.

`last/2`: This predicate succeeds, if its second argument matches the last element of the list given as the first argument of `last/2`.

`reverse/2`: This predicate can be used to reverse the order of elements in a list. The first argument has to be a (fully instantiated) list and the second one will be matched with the reversed list. Example:

```
?- reverse([1, 2, 3, 4, 5], X).
X = [5, 4, 3, 2, 1]
Yes
```

`select/3`: Given a list in the second argument and an element of that list in the first, this predicate will match the third argument with the remainder of that list. Example:

```
?- select(bird, [mouse, bird, jellyfish, zebra], X).
X = [mouse, jellyfish, zebra]
Yes
```

2.4 Exercises

Exercise 2.1. Write a Prolog predicate `analyse_list/1` that takes a list as its argument and prints out the list's head and tail on the screen. If the given list is empty, the predicate should put out an message reporting this fact. If the argument term isn't a list at all, the predicate should just fail. Examples:

```
?- analyse_list([dog, cat, horse, cow]).
This is the head of your list: dog
This is the tail of your list: [cat, horse, cow]
Yes

?- analyse_list([]).
This is an empty list.
Yes

?- analyse_list(sigmund_freud).
No
```

Exercise 2.2. Write a Prolog predicate `membership/2` that works like the built-in predicate `member/2` (without using `member/2`).

Hint: This exercise, like many others, can and should be solved using a recursive approach and the head/tail-pattern for lists.

Exercise 2.3. Implement a Prolog predicate `remove_duplicates/2` that removes all duplicate elements from a list given in the first argument and returns the result in the second argument position. Example:

```
?- remove_duplicates([a, b, a, c, d, d], List).
List = [b, a, c, d]
Yes
```

Exercise 2.4. Write a Prolog predicate `reverse_list/2` that works like the built-in predicate `reverse/2` (without using `reverse/2`). Example:

```
?- reverse_list([tiger, lion, elephant, monkey], List).
List = [monkey, elephant, lion, tiger]
Yes
```

Exercise 2.5. Consider the following Prolog program:

```
whoami([]).
```

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

Under what circumstances will a goal of the form `whoami(X)` succeed?

Exercise 2.6. The objective of this exercise is to implement a predicate for returning the last element of a list in two different ways.

(a) Write a predicate `last1/2` that works like the built-in predicate `last/2` using a recursion and the head/tail-pattern for lists.

(b) Define a similar predicate `last2/2` solely in terms of `append/3`, without using a recursion.

Exercise 2.7. Write a predicate `replace/4` to replace all occurrences of a given element (second argument) by another given element (third argument) in a given list (first argument). Example:

```
?- replace([1, 2, 3, 4, 3, 5, 6, 3], 3, x, List).
List = [1, 2, x, 4, x, 5, 6, x]
Yes
```

Exercise 2.8. Prolog lists without duplicates can be interpreted as sets. Write a program that given such a list computes the corresponding power set. Recall that the power set of a set S is the set of all subsets of S. This includes the empty set as well as the set S itself.

Define a predicate `power/2` such that, if the first argument is instantiated with a list, the corresponding power set (i.e., a list of lists) is returned in the second position. Example:

```
?-  power([a, b, c], P).
P = [[a, b, c], [a, b], [a, c], [a], [b, c], [b], [c], []]
Yes
```

Note: The order of the sub-lists in your result doesn't matter.

Chapter 3

Arithmetic Expressions

If you've tried to use numbers in Prolog before, you might have encountered some unexpected behaviour of the system. The first part of this section clarifies this phenomenon. After that an overview of the arithmetic operators available in Prolog is given.

3.1 The `is`-Operator for Arithmetic Evaluation

Simple arithmetic operators such as `+` or `*` are, as you know, valid Prolog atoms. Therefore, also expressions like `+(3, 5)` are valid Prolog terms. More conveniently, they can also be written as infix operators, like in `3 + 5`.

Without specifically telling Prolog that we are interested in the *arithmetic* properties of such a term, these expressions are treated purely syntactically, i.e., they are not being evaluated. That means using `=` won't work the way you might have expected:

```
?- 3 + 5 = 8.
No
```

The terms `3 + 5` and `8` *do not match*—the former is a compound term, whereas the latter is a number. To check whether the sum of 3 and 5 is indeed 8, we first have to tell Prolog to arithmetically evaluate the term `3 + 5`. This is done by using the built-in operator `is`. We can use it to assign the value of an arithmetic expression to a variable. After that it is possible to match that variable with another number. Let's rewrite our previous example accordingly:

```
?- X is 3 + 5, X = 8.
X = 8
Yes
```

We could check the correctness of this addition also directly, by putting `8` instead of the variable on the lefthand side of the `is`-operator:

```
?- 8 is 3 + 5.
Yes
```

But note that it doesn't work the other way round!

```
?- 3 + 5 is 8.
No
```

This is because `is` only causes the argument *to its right* to be evaluated and then tries to match the result with the lefthand argument. The arithmetic evaluation of `8` yields again `8`, which doesn't match the (non-evaluated) Prolog term `3 + 5`.

To summarise, the `is`-operator is defined as follows: It takes two arguments, of which the second has to be a valid arithmetic expression with all variables instantiated. The first argument has to be either a number or a variable representing a number. A call succeeds if the result of the arithmetic evaluation of the second argument matches with the first one (or in case of the first one being a number, if they are identical).

Note that (in SWI-Prolog) the result of an arithmetic calculation will be a float (rather than an integer) whenever one of the input parameters is a float. This means, for example, that the goal `1 is 0.5 + 0.5` would not succeed, because `0.5 + 0.5` evaluates to the float `1.0`, not the integer `1`. However, other Prolog systems may do this differently. In general, it is better to use the operator `=:=` (which will be introduced in Section 3.2) instead whenever the left argument has been instantiated to a number already. That is, do not use `is/2` to *compare* the values of two arithmetic expressions; only use it to *evaluate* arithmetic expressions, i.e., only use it with a variable on the lefthand side.

3.2 Predefined Arithmetic Functions and Relations

The arithmetic operators available in Prolog can be divided into *functions* and *relations*. Some of them are presented here; for an extensive list consult your Prolog reference manual.

Functions. Addition or multiplication are examples for arithmetic functions. In Prolog all these functions are written in the natural way. The following term shows some examples:

```
2 + (-3.2 * X - max(17, X)) / 2 ** 5
```

The `max/2`-expression evaluates to the largest of its two arguments and `2 ** 5` stands for "2 to the 5th power" (2^5). Other functions available include `min/2` (minimum), `abs/1` (absolute value), `sqrt/1` (square root), and `sin/1` (sinus).[1] The operator `//` is used for integer division. To obtain the remainder of an integer division (modulo) use the `mod`-operator. Precedence of operators is the same as you know it from mathematics, i.e., `2 * 3 + 4` is equivalent to `(2 * 3) + 4` etc.

You can use `round/1` to round a float number to the next integer and `float/1` to convert integers to floats.

All these functions can be used on the righthand side of the `is`-operator.

[1] Like `max/2`, these are all written as functions, not as operators.

Relations. Arithmetic relations are used to compare two evaluated arithmetic expressions. The goal `X > Y`, for example, will succeed if expression `X` evaluates to a greater number than expression `Y`. Note that the `is`-operator is not needed here. The arguments are evaluated whenever an arithmetic relation is used.

Besides `>` the operators `<` (less than), `=<` (less than or equal), `>=` (greater than or equal), `=\=` (non-equal), and `=:=` (*arithmetically* equal) are available. The differentiation of `=:=` and `=` is crucial. The former compares two evaluated arithmetic expressions, whereas the later performs logical pattern matching.

```
?- 2 ** 3 =:= 3 + 5.
Yes
?- 2 ** 3 = 3 + 5.
No
```

Note that, unlike `is`, arithmetic equality `=:=` also works if one of its arguments evaluates to an integer and the other one to the corresponding float.

3.3 Exercises

Exercise 3.1. Write a Prolog predicate `distance/3` to calculate the distance between two points in the 2-dimensional plane. Points are given as pairs of coordinates. Examples:

```
?- distance((0,0), (3,4), X).
X = 5
Yes

?- distance((-2.5,1), (3.5,-4), X).
X = 7.81025
Yes
```

Exercise 3.2. Write a Prolog program to print out a square of $n \times n$ given characters on the screen. Call your predicate `square/2`. The first argument should be a (positive) integer, the second argument the character (any Prolog term) to be printed. Example:

```
?- square(5, '*  ').
*  *  *  *  *
*  *  *  *  *
*  *  *  *  *
*  *  *  *  *
*  *  *  *  *
Yes
```

Exercise 3.3. Write a Prolog predicate `fibonacci/2` to compute the nth Fibonacci number. The Fibonacci sequence is defined as follows:

$$\begin{aligned} F_0 &= 1 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \quad \text{for } n \geq 2 \end{aligned}$$

Examples:

```
?- fibonacci(1, X).
X = 1
Yes

?- fibonacci(2, X).
X = 2
Yes

?- fibonacci(5, X).
X = 8
Yes
```

Exercise 3.4. Write a Prolog predicate `element_at/3` that, given a list and a natural number n, will return the nth element of that list. Examples:

```
?- element_at([tiger, dog, teddy_bear, horse, cow], 3, X).
X = teddy_bear
Yes

?- element_at([a, b, c, d], 27, X).
No
```

Exercise 3.5. Write a Prolog predicate `mean/2` to compute the arithmetic mean of a given list of numbers. Example:

```
?- mean([1, 2, 3, 4], X).
X = 2.5
Yes
```

Exercise 3.6. Write a predicate `range/3` to generate all integers between a given lower and a given upper bound. The lower bound should be given as the first argument, the upper bound as the second. The result should be a list of integers, which is returned in the third argument position. If the upper bound specified is lower than the given lower bound, the empty list should be returned. Examples:

```
?- range(3, 11, X).
X = [3, 4, 5, 6, 7, 8, 9, 10, 11]
Yes

?- range(7, 4, X).
X = []
Yes
```

Exercise 3.7. Polynomials can be represented as lists of pairs of coefficients and exponents. For example the polynomial

$$4x^5 + 2x^3 - x + 27$$

can be represented as the following Prolog list:

```
[(4,5), (2,3), (-1,1), (27,0)]
```

Write a Prolog predicate `poly_sum/3` for adding two polynomials using that representation. Try to find a solution that is independent of the ordering of pairs inside the two given lists. Likewise, your output doesn't have to be ordered. Examples:

```
?- poly_sum([(5,3), (1,2)], [(1,3)], Sum).
Sum = [(6,3),  (1,2)]
Yes

?- poly_sum([(2,2), (3,1), (5,0)], [(5,3), (1,1), (10,0)], X).
X = [(4,1),  (15,0),  (2,2),  (5,3)]
Yes
```

Hints: Before you even start thinking about how to do this in Prolog, recall how the sum of two polynomials is actually computed. A rather simple solution is possible using the built-in predicate `select/3`. Note that the list representation of the sum of two polynomials that don't share any exponents is simply the concatenation of the two lists representing the arguments.

Chapter 4

Operators

In the chapter on arithmetic expressions we have already seen some operators. Several of the predicates associated with arithmetic operations are also predefined operators. This chapter explains how to define your own operators, which can then be used instead of normal predicates.

4.1 Precedence and Associativity

Precedence. From mathematics and also from propositional logic you know that the *precedence* of an operator determines how an expression is supposed to be interpreted. For example, $\wedge$ binds stronger than $\vee$, which is why the formula $P \vee Q \wedge R$ is interpreted as $P \vee (Q \wedge R)$, and not the other way round.

In Prolog every operator is associated with an integer number (in SWI-Prolog between 0 and 1200) denoting its precedence. The lower the precedence number, the stronger the operator is binding. The arithmetic operator `*`, for example, has a precedence of 400, `+` has a precedence of 500. This is why, when evaluating the term `2 + 3 * 5`, Prolog will first compute the product of 3 and 5 and then add it to 2.

The *precedence of a term* is defined as 0, unless its principal functor is an operator, in which case the precedence is the precedence of this operator. Examples:

- The precedence of `3 + 5` is 500.
- The precedence of `3 * 3 + 5 * 5` is also 500.
- The precedence of `sqrt(3 + 5)` is 0.
- The precedence of `elephant` is 0.
- The precedence of `(3 + 5)` is 0.
- The precedence of `3 * +(5, 6)` is 400.

Associativity. Another important concept with respect to operators is their *associativity*. You probably know that there are *infix* operators (like +), *prefix* operators (like ¬ in logic), and sometimes even *postfix* operators (like the factorial operator ! in mathematics). In Prolog the associativity of an operator is also part of its definition.

But giving precedence and indicating whether it's supposed to be infix, prefix, or postfix is not enough to fully specify an operator. Take the example of subtraction. This is an infix operator and in SWI-Prolog it is defined with precedence 500. Is this really all we need to know to understand Prolog's behaviour when answering the following query?

```
?- X is 10 - 5 - 2.
X = 3
Yes
```

Why didn't it compute $5-2=3$ and then $10-3=7$ and return `X = 7` as the result? Well, it obviously did the right thing by first evaluating the left difference $10-5$ before finally subtracting 2. But this must also be part of the operator's definition. The operator `-` is actually defined as an infix operator, for which the righthand argument has to be a term of strictly lower precedence than 500 (the precedence of `-` itself), whereas the lefthand argument only needs to be of lower or equal precedence. Given this rule, it is indeed impossible to interpret `10 - 5 - 2` as `10 - (5 - 2)`, because the precedence of the righthand argument of the principal operator is 500, i.e., it is not strictly lower than 500. We also say the operator `-` "associates to the left" or "is left-associative".

In Prolog associativity (together with such restrictions on arguments' precedences) is represented by atoms like `yfx`. Here `f` indicates the position of the operator (i.e., `yfx` denotes an infix operator) and `x` and `y` indicate the positions of the arguments. A `y` should be read as *on this position a term with a precedence less than or equal to that of the operator has to occur*, whereas `x` means that *on this position a term with a precedence strictly less than that of the operator has to occur.*

Checking precedence and associativity. It is possible to check both precedence and associativity of any previously defined operator by using the predicate `current_op/3`. If the last of its arguments is instantiated with the name of an operator it will match the first one with the operator's precedence and the second with its associativity pattern. The following example for multiplication shows that `*` has precedence 400 and the same associativity pattern as subtraction.

```
?- current_op(Precedence, Associativity, *).
Precedence = 400
Associativity = yfx
Yes
```

Here are some more examples. Note that `-` is defined twice; once as subtraction (infix) and once as negative sign (prefix). The same is true for `+`.[1]

```
?- current_op(Precedence, Associativity, **).
Precedence = 200
Associativity = xfx ;
No

?- current_op(Precedence, Associativity, -).
Precedence = 200
Associativity = fy ;
Precedence = 500
Associativity = yfx ;
No

?- current_op(Precedence, Associativity, <).
Precedence = 700
Associativity = xfx ;
No

?- current_op(Precedence, Associativity, =).
Precedence = 700
Associativity = xfx ;
No

?- current_op(Precedence, Associativity, :-).
Precedence = 1200
Associativity = fx ;
Precedence = 1200
Associativity = xfx ;
No
```

As you can see, there aren't just arithmetic operators, but also stuff like `=` and even `:-` are declared as operators. From the very last example you can see that `:-` can also be a prefix operator. You will see an example for this in the next section.

Table 4.1 provides an overview of possible associativity patterns. Note that it is not possible to nest non-associative operators. For example, `is` is defined as an `xfx`-operator, which means a term like `X is Y is 7` would cause a syntax error. This makes sense, because that term certainly doesn't (make sense).

[1] When generating these examples I always pressed `;` to get all alternatives. This is why at the end of each query Prolog answered with `No`.

Pattern	Associativity		Examples
`yfx`	infix	left-associative	`+, -, *`
`xfy`	infix	right-associative	`,` (for subgoals)
`xfx`	infix	non-associative	`=, is, <` (i.e., no nesting)
`yfy`	makes no sense, structuring would be impossible		
`fy`	prefix	associative	`-` (i.e., `- - 5` allowed)
`fx`	prefix	non-associative	`:-` (i.e., `:- :- goal` not allowed)
`yf`	postfix	associative	
`xf`	postfix	non-associative	

Table 4.1: Associativity patterns for operators in Prolog

4.2 Declaring Operators with `op/3`

Now we want to define our own operators. Recall the example on big and not so big animals from Chapter 1. Maybe, instead of writing terms like `is_bigger(elephant, monkey)` we would prefer to be able to express the same thing using `is_bigger` as an infix operator:

```
elephant is_bigger monkey
```

This is possible, but we first have to declare `is_bigger` as an operator. As precedence we could choose, say, 300. It doesn't really matter as long as it is lower than 700 (the precedence of `=`) and greater than 0. What should the associativity pattern be? We already said it's going to be an infix operator. As arguments we only want atoms or variables, i.e., terms of precedence 0. Therefore, we should choose `xfx` to prevent users from nesting `is_bigger`-expressions.

Operators are declared using the `op/3` predicate, which has the same syntax as `current_op/3`. The difference is that this one actually *defines* the operator rather than retrieving its definition. Therefore, all arguments have to be instantiated. Again, the first argument denotes the precedence, the second one the associativity type, and the third one the name of the operator. Any Prolog atom could become the name of an operator, unless it is one already. Our `is_bigger`-operator is declared by submitting the following query:

```
?- op(300, xfx, is_bigger).
Yes
```

Now Prolog knows it's an operator, but doesn't necessarily have a clue how to evaluate the truth of an expression containing this operator. This has to be programmed in terms of facts and rules in the usual way. When implementing them you have the choice of either using the operator notation or normal predicate notation. That means we can use the program from Chapter 1 in its present form. The operator `is_bigger` will be associated with the functor `is_bigger` that has been used there, i.e., after having compiled the program file we can ask queries like the following:

```
?- elephant is_bigger donkey.
Yes
```

As far as matching is concerned, predicate and operator notation are considered to be identical, as you can see from Prolog's reply to this query:

```
?- (elephant is_bigger tiger) = is_bigger(elephant, tiger).
Yes
```

Query execution at compilation time. Obviously, it wouldn't be very practical to redefine all your operators every time you re-start the Prolog interpreter. Fortunately, it is possible to tell Prolog to make the definitions at compilation time. More generally speaking, you may put any query you like directly into a program file, which will cause it to be executed whenever you consult that file. The syntax for such queries is similar to rules, but without a head. Say, for instance, your program contains the following line:

```
:- write('Hello, have a beautiful day!').
```

Then every time you consult the file, this will cause the goal after `:-` to be executed:

```
?- consult('my-file.pl').
Hello, have a beautiful day!
my-file.pl compiled, 0.00 sec, 224 bytes.
Yes
?-
```

You can do exactly the same with operator definitions, i.e., you could add the definition for `is_bigger`

```
:- op(300, xfx, is_bigger).
```

at the beginning of the big animals program file and the operator will be available directly after compilation. This means that you can use `is_bigger` in infix-notation to define clauses in your program file and you can use it for queries when you run your program.

4.3 Exercises

Exercise 4.1. Consider the following operator definitions:

```
:- op(100, yfx, plink),
   op(200, xfy, plonk).
```

(a) Copy the operator definitions into a program file and compile it. Then run the following queries and explain what is happening.

(i) `?- tiger plink dog plink fish = X plink Y.`

(ii) `?- cow plonk elephant plink bird = X plink Y.`

(iii) `?- X = (lion plink tiger) plonk (horse plink donkey).`

(b) Write a Prolog predicate `pp_analyse/1` to analyse `plink/plonk`-expressions. The output should tell you what the principal operator is and which are the two main sub-terms. If the main operator is neither `plink` nor `plonk`, then the predicate should fail. Examples:

```
?- pp_analyse(dog plink cat plink horse).
Principal operator: plink
Left sub-term: dog plink cat
Right sub-term: horse
Yes

?- pp_analyse(dog plonk cat plonk horse).
Principal operator: plonk
Left sub-term: dog
Right sub-term: cat plonk horse
Yes

?- pp_analyse(lion plink cat plonk monkey plonk cow).
Principal operator: plonk
Left sub-term: lion plink cat
Right sub-term: monkey plonk cow
Yes
```

Exercise 4.2. Consider the following operator definitions:

```
:- op(100, fx, the),
   op(100, fx, a),
   op(200, xfx, has).
```

(a) Indicate the structure of this term using parentheses and name its principal functor:

```
claudia has a car
```

(b) What would Prolog reply when presented with the following query?

```
?- the lion has hunger = Who has What.
```

(c) Explain why the following query would cause a syntax error:

```
?- X = she has whatever has style.
```

Exercise 4.3. Define operators in Prolog for the connectives of propositional logic. Use the following operator names:

– Negation: `neg`

- Conjunction: `and`
- Disjunction: `or`
- Implication: `implies`

Think about what precedences and associativity patterns are appropriate. In particular, your declarations should reflect the precedence hierarchy of the connectives as they are defined in propositional logic. Define all binary logical operators as being left-associative. Your definitions should allow for double negation without parentheses (see examples).

Hint: You can easily test whether your operator declarations work as intended. Recall that Prolog omits all redundant parentheses when it prints out the answer to a query. That means, when you ask Prolog to match a variable with a formula whose structure you have indicated using parentheses, those that are redundant should all disappear in the output. Parentheses that are necessary, however, will be shown. Examples:

```
?- Formula = a implies ((b and c) and d).
Formula = a implies b and c and d
Yes

?- AnotherFormula = (neg (neg a)) or b.
AnotherFormula = neg neg a or b
Yes

?- ThirdFormula = (a or b) and c.
ThirdFormula = (a or b)and c
Yes
```

Exercise 4.4. Write a Prolog predicate `cnf/1` to test whether a given formula is in conjunctive normal form (using the operators you defined for the previous exercise). Examples:

```
?- cnf((a or neg b) and (b or c) and (neg d or neg e)).
Yes

?- cnf(a or (neg b)).
Yes

?- cnf((a and b and c) or d).
No

?- cnf(a and b and (c or d)).
Yes
```

```
?- cnf(a).
Yes

?- cnf(neg neg a).
No
```

Hint: Propositional atoms correspond to atoms in Prolog. You can test whether a given term is a valid Prolog atom by using the built-in predicate `atom/1`.

Chapter 5

Backtracking, Cuts and Negation

In this chapter you will learn a bit more on how Prolog resolves queries. We will also introduce a control mechanism (cuts) that allows for more efficient implementations. Furthermore, some extensions to the syntax of Prolog programs will be discussed. Besides conjunction (remember, a comma separating two subgoals in a rule-body represents a conjunction) we shall introduce negation and disjunction.

5.1 Backtracking and Cuts

In Chapter 1 the term "backtracking" has been mentioned already. Next we are going to examine backtracking in some more detail, note some of its useful applications as well as problems, and discuss a way of overcoming such problems (by using so-called cuts).

5.1.1 Backtracking Revisited

During proof search, Prolog keeps track of choicepoints, i.e., situations where there is more than one possible match. Whenever the chosen path ultimately turns out to be a failure (or if the user asks for alternative solutions), the system can jump back to the last choicepoint and try the next alternative. This is process is known as *backtracking*. It is a crucial feature of Prolog and facilitates the concise implementation of many problem solutions.

Let's consider a concrete example. We want to write a predicate to compute all possible permutations of a given list. The following implementation uses the built-in predicate `select/3`, which takes a list as its second argument and matches the first argument with an element from that list. The variable in the third argument position will then be matched with the rest of the list after having removed the chosen element. Here's a very simple recursive definition of the predicate `permutation/2`:

```
permutation([], []).

permutation(List, [Element | Permutation]) :-
```

```
  select(Element, List, Rest),
  permutation(Rest, Permutation).
```

The simplest case is that of an empty list. There's just one possible permutation, the empty list itself. If the input list has got elements, then the subgoal `select(Element, List, Rest)` will succeed and bind the variable `Element` to an element of the input list. It makes that element the head of the output list and recursively calls `permutation/2` again with the rest of the input list. The first answer to a query will simply reproduce the input list, because `Element` will always be assigned to the value of the head of `List`. If further alternatives are requested, however, backtracking into the `select`-subgoal takes place, i.e., each time `Element` is instantiated with another element of `List`. This will generate all possible orders of selecting elements from the input list, in other words, this will generate all permutations of the input list. Example:

```
?- permutation([1, 2, 3], X).

X = [1, 2, 3] ;

X = [1, 3, 2] ;

X = [2, 1, 3] ;

X = [2, 3, 1] ;

X = [3, 1, 2] ;

X = [3, 2, 1] ;

No
```

We have also seen other examples for exploiting the backtracking feature before, e.g., in Section 2.2. There we used backtracking into `concat_lists/3` (which is the same as the built-in predicate `append/3`) to find all possible decompositions of a given list.

5.1.2 Problems with Backtracking

There are cases, however, were backtracking is not desirable. Consider, for example, the following definition of the predicate `remove_duplicates/2` to remove duplicate elements from a given list.

```
remove_duplicates([], []).

remove_duplicates([Head | Tail], Result) :-
  member(Head, Tail),
```

```
  remove_duplicates(Tail, Result).

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

The *declarative meaning* of this predicate definition is the following. Removing duplicates from the empty list yields again the empty list. There's certainly nothing wrong with that. The second clause says that if the head of the input list can be found in its tail, the result can be obtained by recursively applying `remove_duplicates/2` to the list's tail, discarding the head. Otherwise we get the tail of the result also by applying the predicate to the tail of the input, but this time we keep the head.

This works almost fine. The *first solution* found by Prolog will indeed always be the intended result. But when requesting alternative solution things will start going wrong. The two rules provide a choicepoint. For the first branch of the search tree Prolog will always pick the first rule, if that is possible, i.e., whenever the head is a member of the tail it will be discarded. During backtracking, however, also all other branches of the search tree will be visited. Even if the first rule *would* match, sometimes the second one will be picked instead and the duplicate head will remain in the list. The (semantically wrong) output can be seen in the following example:

```
?- remove_duplicates([a, b, b, c, a], List).

List = [b, c, a] ;

List = [b, b, c, a] ;

List = [a, b, c, a] ;

List = [a, b, b, c, a] ;

No
```

That is, Prolog not only generates the correct solution, but also all other lists we get by keeping *some* of the elements that should have been deleted. To solve this problem we need a way of telling Prolog that, even when the user (or another predicate calling `remove_duplicates/2`) requests further solutions, there are no such alternatives and the goal should fail.

5.1.3 Introducing Cuts

Prolog provides a solution to the sort of problems discussed above. It is possible to explicitly "cut out" backtracking choicepoints, thereby guiding the proof search and prohibiting unwanted alternative solutions to a query.

A *cut* is written as `!`. It is a predefined Prolog predicate and can be placed anywhere inside a rule's body (or similarly, be part of a sequence of subgoals in a query). Executing the subgoal `!` will always succeed, but afterwards backtracking into subgoals placed *before* the cut inside the same rule body is not possible anymore.

We will define this more precisely a bit later. Let's first look at our example about removing duplicate elements from a list again. We change the previously proposed program by inserting a cut after the first subgoal inside the body of the first rule; the rest remains exactly the same as before.

```
remove_duplicates([], []).

remove_duplicates([Head | Tail], Result) :-
  member(Head, Tail), !,
  remove_duplicates(Tail, Result).

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

Now, whenever the head of a list is a member of its tail, the first subgoal of the first rule, i.e., `member(Head, Tail)`, will succeed. Then the next subgoal, `!`, will also succeed. Without that cut it would be possible to backtrack, that is, to match the original goal with the head of the second rule to search for alternative solutions. But once Prolog went past the cut, this isn't possible anymore: alternative matchings for the parent goal[1] will not be tried.

Using this new version of the predicate `remove_duplicates/2`, we get the desired behaviour. When asking for alternative solutions by pressing `;` we immediately get the right answer, namely `No`.

```
?- remove_duplicates([a, b, b, c, a], List).
List = [b, c, a] ;
No
```

Now we are ready for a more precise *definition of cuts* in Prolog: Whenever a cut is encountered in a rule's body, all choices made between the time that rule's head has been matched with the parent goal and the time the cut is passed are final, i.e., any choicepoints are being discarded.

Let's exemplify this with a little story. Suppose a young prince wants to get married. In the old days he'd simply have saddled his best horse to ride down to the valleys of, say, Essex[2] and find himself the sort of young, beautiful, and intelligent girl he's after. But, obviously, times have changed, life in general is becoming much more complex these days, and most importantly, our prince is rather busy defending monarchy against

[1] With "parent goal" we mean the goal that caused the matching of the rule's head.

[2] This story has been written with a British audience in mind. Please adapt to your local circumstances.

communism/anarchy/democracy (pick your favourite). Fortunately, his royal board of advisors consists of some of the finest psychologists and Prolog programmers in the country. They form an executive committee to devise a Prolog program to automatise the prince's quest for a bride. The task is to simulate as closely as possible the prince's decision if he actually were to go out there and look for her by himself. From the expert psychologists we gather the following information:

- The prince is primarily looking for a *beautiful* girl. But, to be eligible for the job of a prince's wife, she'd also have to be *intelligent*.
- The prince is young and very romantic. Therefore, he will fall in love with the *first* beautiful girl he comes across, love her for ever, and never ever consider any other woman as a potential wife again. Even if he can't marry that girl.

The MI5 provides the committee with a database of women of the appropriate age. The entries are ordered according to the order the prince would have met them on his ride through the country. Written as a list of Prolog facts it looks something like this:

```
beautiful(claudia).
beautiful(sharon).
beautiful(denise).
...

intelligent(margaret).
intelligent(sharon).
...
```

After some intensive thinking the Prolog sub-committee comes up with the following ingenious rule:

```
bride(Girl) :-
  beautiful(Girl), !,
  intelligent(Girl).
```

Let's leave the cut in the second line unconsidered for the moment. Then a query of the form

```
?- bride(X).
```

will succeed, if there is a girl `X` for which both the facts `beautiful(X)` and `intelligent(X)` can be found in the database. Therefore, the first requirement identified by the psychologists will be fulfilled. The variable `X` would then be instantiated with the girl's name.

In order to incorporate the second condition the Prolog experts had to add the cut. If the subgoal `beautiful(Girl)` succeeds, i.e., if a fact of the form `beautiful(X)` can

be found (and it will be the first such fact), then that choice will be final, even if the subgoal `intelligent(X)` for the same `X` should fail.

Given the above database, this is rather tragic for our prince. The first beautiful girl he'd meet would be Claudia, and he'd fall in love with her immediately and forever. In Prolog this corresponds to the subgoal `beautiful(Girl)` being successful with the variable instantiation `Girl = claudia`. And it stays like this forever, because after having executed the cut, that choice cannot be changed anymore. As it happens, Claudia isn't the most amazingly intelligent young person that you might wish her to be, which means they cannot get married. In Prolog, again, this means that the subgoal `intelligent(Girl)` with the variable `Girl` being bound to the value `claudia` will not succeed, because there is no such fact in the program. That means the entire query will fail. Even though there is a name of a girl in the database, who is both beautiful and intelligent (Sharon), the prince's quest for marriage is bound to fail:

```
?- bride(X).
No
```

5.1.4 Problems with Cuts

Cuts are very useful to "guide" the Prolog interpreter towards a solution. But this doesn't come for free. By introducing cuts, we give up some of the (nice) declarative character of Prolog and move towards a more procedural system. This can sometimes lead to unexpected results.

To illustrate this, let's implement a predicate `add/3` to insert an element into a list, if that element isn't already a member of the list. The element to be inserted should be given as the first argument, the list as the second one. The variable given in the third argument position should be matched with the result. Examples:

```
?- add(elephant, [dog, donkey, rabbit], List).
List = [elephant, dog, donkey, rabbit] ;
No

?- add(donkey, [dog, donkey, rabbit], List).
List = [dog, donkey, rabbit] ;
No
```

The important bit here is that there are no wrong alternative solutions. The following Prolog program does the job:

```
add(Element, List, List) :-
  member(Element, List), !.

add(Element, List, [Element | List]).
```

If the element to be inserted can be found in the list already, the output list should be identical with the input list. As this is the only correct solution, we prevent Prolog from backtracking by using a cut. Otherwise, i.e., if the element is not already in the list, we use the head/tail-pattern to construct the output list.

This is an example for a program where cuts can be problematic. When used as specified, namely with a variable in the third argument position, `add/3` works fine. If, however, we put an instantiated list in the third argument, Prolog's reply can be different from what you might expect. Example:

```
?- add(a, [a, b, c, d], [a, a, b, c, d]).
Yes
```

Compare this with the definition of the `add/3`-predicate from above and try to understand what's happening here. One possible solution would be to explicitly say in the second clause that `member(Element,List)` should not succeed, rather than using a cut in the first clause. We are going to see how to do this using negation in the next section. An alternative solution would be to rewrite the definition of `add/3` as follows:

```
add(Element, List, Result) :-
  member(Element, List), !,
  Result = List.

add(Element, List, [Element | List]).
```

Try to understand how this solves the problem. Note that from a declarative point of view the two versions of the program are equivalent, but procedurally they behave differently. So be careful with those cuts!

5.2 Negation as Failure

In the marriage example from before, from the fact `intelligent(claudia)` not appearing in the database we concluded that beautiful Claudia wasn't intelligent. This touches upon an important issue of Prolog semantics, namely that of negation.

5.2.1 The Closed World Assumption

In order to give a positive answer to a query, Prolog has to construct a proof to show that the set of facts and rules of a program implies that query. Therefore, precisely speaking, answering `Yes` to a query means not only that the query is true, but that it is *provably true*. Consequently a `No` doesn't mean the query is necessarily false, just *not provably true*: Prolog failed to derive a proof.

This attitude of negating everything that is not explicitly in the program (or can be concluded from the information provided by the program) is often referred to as the

closed world assumption. That is, we think of our Prolog program as a little world of its own, assuming nothing outside that world does exist (or is true).

In everyday reasoning we usually don't make this sort of assumption. Just because the duckbill might not appear in even a very big book on animals, we cannot infer that it isn't an animal. In Prolog, on the other hand, when we have a list of facts like

```
animal(elephant).
animal(tiger).
animal(lion).
...
```

and `animal(duckbill)` does not appear in that list (and there are no rules with `animal/1` in the head), then Prolog would react to a query asking whether the duckbill was an animal as follows:

```
?- animal(duckbill).
No
```

The closed world assumption might seem a little narrow-minded at first sight, but you will appreciate that it is the only admissible interpretation of a Prolog reply, as Prolog clauses only give sufficient, not necessary conditions for a predicate to hold. Note, however, that if you have *completely* specified a certain problem, i.e., when you can be sure that for every case where there is a positive solution Prolog has all the data to be able to construct the respective proof, then the notions of *not provable* and *false* coincide. A `No` then really does mean no.

5.2.2 The \+-Operator

Sometimes we might not want to ask whether a certain goal succeeds, but whether it fails. That is, we want to be able to *negate* goals. In Prolog this is possible using the `\+`-operator. This is a prefix operator that can be applied to any valid Prolog goal. A goal of the form `\+ Goal` succeeds, if the goal `Goal` fails and *vice versa*. In other words, `\+ Goal` succeeds, if Prolog fails to derive a proof for `Goal` (i.e., if `Goal` is not provably true). This semantics of the negation operator is known as *negation as failure*. Prolog's negation is defined as the failure to provide a proof. In real life this is usually not the right notion (though it has been adopted by judicature: "innocent unless proven guilty").

Let's look at an example for the use of the `\+`-operator. Assume we have a list of Prolog facts with pairs of people who are married to each other:

```
married(peter, lucy).
married(paul, mary).
married(bob, juliet).
married(harry, geraldine).
```

Then we can define a predicate `single/1` that succeeds if the argument given can neither be found as the first nor as the second argument in any of the `married/2`-facts. We can use the anonymous variable for the other argument of `married/2`, because its value would be irrelevant:

```
single(Person) :-
  \+ married(Person, _),
  \+ married(_, Person).
```

Example queries:

```
?- single(mary).
No

?- single(claudia).
Yes
```

Again, we have to read the answer to the last query as "Claudia is assumed to be single, because she cannot be shown to be married". We are only allowed to shorten this interpretation to "Claudia *is* single", if we can be sure that the list of `married/2`-facts is exhaustive, i.e., if we accept the closed world assumption for this example.

Now consider the following query and Prolog's response:

```
?- single(X).
No
```

This means, that Prolog cannot provide any example for a person `X` that would be single. This is so, because our little database of married people is all that Prolog knows about in this example.

Where to use `\+`. We have mentioned already that the `\+`-operator can be applied to any valid Prolog goal. Recall what this means. Goals are either (sub)goals of a query or subgoals of a rule-body. Facts and rule-heads aren't goals. Hence, it is not possible to negate a fact or the head of a rule. This perfectly coincides with what has been said about the closed world assumption and the notion of negation as failure: it is not possible to explicitly declare a predicate as being false.

5.3 Disjunction

The comma in between two subgoals of a query or a rule-body denotes a *conjunction*. The entire goal succeeds if both the first *and* the second subgoal succeed.

We already know one way of expressing a *disjunction*. If there are two rules with the same head in a program then this represents a disjunction, because during the goal execution process Prolog could choose either one of the two rule bodies when the current

goal matches the common rule-head. Of course, it will always try the first such rule first, and only execute the second one if there has been a failure or if the user has asked for alternative solutions.

In most cases this form of disjunction is the one that should be used, but sometimes it can be useful to have a more compact notation corresponding to the comma for conjunction. In such cases you can use `;` (semicolon) to separate two subgoals. As an example, consider the following definition of `parent/2`:

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

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

This means, "`X` can be shown to be the parent of `Y`, if `X` can be shown to be the father of `Y` *or* if `X` can be shown to be the mother of `Y`". The same definition can also be given more compactly:

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

Note that the precedence value of `;` (semicolon) is higher than that of `,` (comma). Therefore, when implementing a disjunction inside a conjunction you have to structure your rule-body using parentheses.

The semicolon should only be used in exceptional cases. As it can easily be mixed up with the comma, it makes programs less readable.

5.4 Example: Evaluating Logic Formulas

As an example, let's try to write a short Prolog program that may be used to evaluate a row in a truth table. Assume appropriate operator definitions have been made before (see for example the exercises at the end of the chapter on operators). Using those operators, we want to be able to type a Prolog term corresponding to the logic formula in question (with the propositional variables being replaced by a combination of truth values) into the system and get back the truth value for that row of the table.

In order to compute the truth table for $A \wedge B$ we would have to execute the following four queries:

```
?- true and true.
Yes

?- true and false.
No
```

```
?- false and true.
No

?- false and false.
No
```

Hence, the corresponding truth table would look like this:

A	B	$A \wedge B$
T	T	T
T	F	F
F	T	F
F	F	F

One more example before we start writing the actual program:

```
?- true and (true and false implies true) and neg false.
Yes
```

In the examples we have used the Prolog atoms `true` and `false`. The former is actually a built-in predicate with exactly the meaning we require, so that's fine. Also `false` will be available as a built-in on some Prolog systems (with the same meaning as `fail`). If not, we can easily define it ourselves:

```
false :- fail.
```

Next are conjunction and disjunction. They obviously correspond to the Prolog operators `,` (comma) and `;` (semicolon), respectively. We use the built-in predicate `call/1` to invoke the subformulas represented by the variables as Prolog goals:

```
and(A, B) :- call(A), call(B).


or(A, B) :- call(A); call(B).
```

Our own negation operator `neg` again is just another name for the built-in `\+`-Operator:

```
neg(A) :- \+ call(A).
```

Defining implication is a bit more tricky. One way would be to exploit the classical equivalence of $A \rightarrow B \equiv \neg A \vee B$ and define `implies` in terms of `or` and `neg`. A somewhat nicer solution (this, admittedly, depends on one's sense of aesthetics), however, would be to use a cut. Like this:

```
implies(A, B) :- call(A), !, call(B).
implies(_, _).
```

How does that work? Suppose `A` is false. Then the first rule will fail, Prolog will jump to the second one and succeed whatever `B` may be. This is exactly what we want: an implication evaluates to *true* whenever its antecedent evaluates to *false*. In case `call(A)` succeeds, the cut in the first rule will be passed and the overall goal will succeed if and only if `call(B)` does. Again, this is precisely what we want in classical logic.

Remark. We know that in classical logic $\neg A$ is equivalent to $A \rightarrow \bot$. Similarly, instead of using `\+` in Prolog we could define our own negation operator as follows:

```
neg(A) :- call(A), !, fail.
neg(_).
```

5.5 Exercises

Exercise 5.1. Type the following queries into a Prolog interpreter and explain what happens.

(a) `?- (Result = a ; Result = b), !, Result = b.`

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

Exercise 5.2. Consider the following Prolog program:

```
result([_, E | L], [E | M]) :- !,
  result(L, M).

result(_, []).
```

(a) After having consulted this program, what would Prolog reply when presented with the following query? Try answering this question first without actually typing in the program, but verify your solution later on using the Prolog system.

```
?- result([a, b, c, d, e, f, g], X).
```

(b) Briefly describe what the program does and how it does what it does, when the first argument of the `result/2`-predicate is instantiated with a list and a variable is given in the second argument position, i.e., as in item (a). Your explanations should include answers to the following questions:

- What case(s) is/are covered by the Prolog fact?
- What effect has the cut in the first line of the program?
- Why has the anonymous variable been used?

Exercise 5.3. Implement Euclid's algorithm to compute the greatest common divisor (GCD) of two non-negative integers. This predicate should be called `gcd/3` and, given two non-negative integers in the first two argument positions, should match the variable in the third position with the GCD of the two given numbers. Examples:

```
?- gcd(57, 27, X).
X = 3
Yes
```

```
?- gcd(1, 30, X).
X = 1
Yes

?- gcd(56, 28, X).
X = 28
Yes
```

Make sure your program behaves correctly also when the semicolon key is pressed.

Hints: The GCD of two numbers a and b (with $a \geq b$) can be found by recursively substituting a with b and b with the rest of the integer division of a and b. Make sure you define the right base case(s).

Exercise 5.4. Implement a Prolog predicate `occurrences/3` to count the number of occurrences of a given element in a given list. Make sure there are no wrong alternative solutions. Example:

```
?- occurrences(dog, [dog, frog, cat, dog, dog, tiger], N).
N = 3
Yes
```

Exercise 5.5. Write a Prolog predicate `divisors/2` to compute the list of all divisors for a given natural number. Example:

```
?- divisors(30, X).
X = [1, 2, 3, 5, 6, 10, 15, 30]
Yes
```

Make sure your program doesn't give any wrong alternative solutions and doesn't fall into an infinite loop when the user presses the semicolon key.

Exercise 5.6. Check some of your old Prolog programs to see whether they produce wrong alternative solutions or even fall into a loop when the user presses ; (semicolon). Fix any problems you encounter using cuts (*one* will often be enough).

Chapter 6

Logic Foundations of Prolog

From using expressions such as "predicate", "true", "proof", etc. when speaking about Prolog programs and the way goals are executed when a Prolog system attempts to answer a query it should have become clear already that there is a very strong connection between logic and Prolog. Not only is Prolog the programming language that is probably best suited for implementing applications that involve logical reasoning, but Prolog's query resolution process itself is actually based on a logical deduction system. This part of the notes is intended to give you a first impression of the logics behind Prolog.

We start by showing how (basic) Prolog programs can be translated into sets of first-order logic formulas. These formulas all have a particular form; they are known as *Horn formulas*. Then we shall briefly introduce *resolution*, a proof system for Horn formulas, which forms the basis of every Prolog interpreter.

6.1 Translation of Prolog Clauses into Formulas

This sections describes how Prolog clauses (i.e., facts, rules, and queries) can be translated into first-order logic formulas. We will only consider the very basic Prolog syntax here, in particular we won't discuss cuts, negation, disjunction, the anonymous variable, or the evaluation of arithmetic expressions at this point. Recall that given their internal representation (using the dot-functor, see Section 2.1) lists don't require any special treatment, at least not at this theoretical level.

Prolog predicates correspond to predicate symbols in logic, terms inside the predicates correspond to functional terms appearing as arguments of logic predicates. These terms are made up of constants (Prolog atoms), variables (Prolog variables), and function symbols (Prolog functors). All variables in a Prolog clause are implicitly universally quantified (that is, every variable could be instantiated with any Prolog term).

Given this mapping from Prolog predicates to atomic first-order formulas the translation of entire Prolog clauses is straightforward. Recall that `:-` can be read as "if", i.e., as an implication from right to left; and that the comma separating subgoals in a clause constitutes a conjunction. Prolog queries can be seen as Prolog rules with an empty

head. This empty head is translated as $\bot$ (falsum). Why this is so will become clear later. When translating a clause, for every variable `X` appearing in the clause we have to put $\forall x$ in front of the resulting formula. The universal quantification implicitly inherent in Prolog programs has to be made explicit when writing logic formulas.

Before summarising the translation process more formally we give an example. Consider the following little program consisting of two facts and two rules:

```
bigger(elephant, horse).
bigger(horse, donkey).
is_bigger(X, Y) :- bigger(X, Y).
is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y).
```

Translating this into a set of first-order logic formulas yields:

$$\begin{array}{ll}\{ & bigger(elephant, horse),\\ & bigger(horse, donkey),\\ & \forall x.\forall y.(bigger(x,y) \rightarrow is\_bigger(x,y)),\\ & \forall x.\forall y.\forall z.(bigger(x,z) \wedge is\_bigger(z,y) \rightarrow is\_bigger(x,y)) \quad \}\end{array}$$

Note how the head of a rule is rewritten as the consequent of an implication. Also note that each clause has to be quantified independently. This corresponds to the fact that variables from distinct clauses are independent from each other, even when they've been given the same name. For example, the `X` in the first rule has nothing to do with the `X` in the second one. In fact, we could rename `X` to, say, `Claudia` throughout the first but not the second rule—this would not affect the behaviour of the program. In logic, this is known as the *renaming of bound variables*.

If several clauses form a program, that program corresponds to a set of formulas and each of the clauses corresponds to exactly one of the formulas in that set. Of course, we can also translate single clauses. For example, the query

```
?- is_bigger(elephant, X), is_bigger(X, donkey).
```

corresponds to the following first-order formula:

$$\forall x.(is\_bigger(elephant, x) \wedge is\_bigger(x, donkey) \rightarrow \bot)$$

As you know, queries can also be part of a Prolog program (in which case they are preceded by `:-`), i.e., such a formula could also be part of a set corresponding to an entire program.

To summarise, when translating a Prolog program (i.e., a sequence of clauses) into a set of logic formulas you have to carry out the following steps:

(1) Every Prolog predicate is mapped to an atomic first-order logic formula (*syntactically*, both are exactly the same: you can just rewrite them without making any changes).

(2) Commas separating subgoals correspond to conjunctions in logic (i.e., you have to replace every comma between two predicates by a $\wedge$ in the formula).

(3) Prolog rules are mapped to implications, where the rule body is the antecedent and the rule head the consequent (i.e., rewrite `:-` as $\rightarrow$ and *change the order* of head and body).

(4) Queries are mapped to implications, where the body of the query is the antecedent and the consequent is $\bot$ (i.e., rewrite `:-` or `?-` as $\rightarrow$, which is put after the translation of the body and followed by $\bot$).

(5) Each variable occurring in a clause has to be universally quantified in the formula (i.e., write $\forall x$ in front of the whole formula for each variable `X`).

6.2 Horn Formulas and Resolution

The formulas we get when translating Prolog rules all have a similar structure: they are implications with an atom in the consequent and a conjunction of atoms in the antecedent (this implication again is usually in the scope of a sequence of universal quantifiers). Abstracting from the quantification for the moment, these formulas all have the following structure:

$$A_1 \wedge A_2 \wedge \cdots \wedge A_n \rightarrow B$$

Such a formula can be rewritten as follows:

$$\begin{array}{ll} A_1 \wedge A_2 \wedge \cdots \wedge A_n \rightarrow B & \equiv \\ \neg(A_1 \wedge A_2 \wedge \cdots \wedge A_n) \ \vee \ B & \equiv \\ \neg A_1 \vee \neg A_2 \vee \cdots \vee \neg A_n \vee B & \end{array}$$

Note that if B is $\bot$ (which is the case when we translate queries) we obtain the following:

$$\neg A_1 \vee \neg A_2 \vee \cdots \vee \neg A_n \vee \bot \quad \equiv \quad \neg A_1 \vee \neg A_2 \vee \cdots \vee \neg A_n$$

Hence, every formula we get when translating a Prolog program into first-order formulas can be transformed into a universally quantified disjunction of literals with at most one positive literal. Such formulas are called *Horn formulas*.[1] (Sometimes the term Horn formula is also used to refer to conjunctions of disjunctions of literals with at most one positive literal each; that would corresponds to an entire Prolog program.)

As $A \rightarrow \bot$ is logically equivalent to $\neg A$, by translating queries as implications with $\bot$ in the consequent we are basically putting the negation of the goal in a query into the set of formulas. Answering a query in Prolog means showing that the set corresponding to the associated program together with the translation of that query is logically inconsistent.

[1] A Prolog fact is simply translated into an atomic formula, i.e., a positive literal. Therefore, formulas representing facts are also Horn formulas.

This is equivalent to showing that the goal logically follows from the set representing the program:

$$\mathcal{P}, (A \rightarrow \bot) \vdash \bot \quad \text{if and only if} \quad \mathcal{P} \vdash A$$

In plain English: to show that A follows from $\mathcal{P}$, show that adding the negation of A to $\mathcal{P}$ will lead to a contradiction.

In principle, such a proof could be accomplished using any formal proof system (e.g., natural deduction or semantic tableaux), but usually the *resolution* method is chosen, which is particularly suited for Horn formulas. We are not going to present the resolution method in its entirety here, but the basic idea is very simple. This proof system has just one rule, which is exemplified in the following argument (all formulas involved need to be Horn formulas):

$$\frac{\neg A_1 \vee \neg A_2 \vee B_1 \qquad \neg B_1 \vee \neg B_2}{\neg A_1 \vee \neg A_2 \vee \neg B_2}$$

If we know $\neg A_1 \vee \neg A_2 \vee B_1$ and $\neg B_1 \vee \neg B_2$, then we also know $\neg A_1 \vee \neg A_2 \vee \neg B_2$, because in case B_1 is false $\neg A_1 \vee \neg A_2$ has to hold and in case B_1 is true, $\neg B_2$ has to hold. In the example, the first formula corresponds to this Prolog rule:

```
b1 :- a1, a2.
```

The second formula corresponds to a query:

```
?- b1, b2.
```

The result of applying the resolution rule then corresponds to the following new query:

```
?- a1, a2, b2.
```

And this is exactly what we would have expected. When executing the goal `b1, b2` Prolog starts by looking for a fact or a rule-head matching the first subgoal `b1`. Once the right rule has been found, the current subgoal is replaced with the rule body, in this case `a1, a2`. The new goal to execute therefore is `a1, a2, b2`.

In Prolog this process is repeated until there are no more subgoals left in the query. In resolution this corresponds to deriving an "empty disjunction", in other words $\bot$.

When using variables in Prolog, we have to move from propositional to first-order logic. The resolution rule for first-order logic is basically the same as the one for propositional logic. The difference is, that it is not enough anymore just to look for complementary literals (B_1 and $\neg B_1$ in the previous example) that can be found in the set of Horn formulas, but now we also have to consider pairs of literals that can be made complementary by means of unification. Unification in logic corresponds to matching in Prolog (but see the exercise section for some important subtleties). The variable instantiations returned by Prolog for successful queries correspond to the unifications made during a resolution proof.

This short presentation has only touched the very surface of what is commonly referred to as the *theory of logic programming*. The "real thing" goes much deeper and has been the object of intensive research for many years all over the world. More details can be found in books on automated theorem proving (in particular resolution), more theoretically oriented books on logic programming in general and Prolog in particular, and various scientific journals on logic programming and alike.

6.3 Exercises

Exercise 6.1. Translate the following Prolog program into a set of first-order logic formulas:

```
parent(peter, sharon).
parent(peter, lucy).

male(peter).

female(lucy).
female(sharon).

father(X, Y) :-
  parent(X, Y),
  male(X).

sister(X, Y) :-
  parent(Z, X),
  parent(Z, Y),
  female(X).
```

Exercise 6.2. Type the following query into Prolog and try to explain what happens:

```
?- X = f(X).
```

Hint: This example shows that matching (Prolog) and unification (logic) are in fact not exactly the same concept. Take your favourite Prolog book and read about the "occurs check" to find out more about this.

Exercise 6.3. As we have seen in this chapter, the goal execution process in Prolog can be explained in terms of the resolution method. (By the way, this also means, that a Prolog interpreter could be based on a resolution-based automated theorem prover implemented in a low-level language such as Java or C++.)

Recall the mortal Socrates example from the introductory chapter (page 10) and what has been said there about Prolog's way of deriving a solution to a query. Translate that

program and the query into first-order logic and see if you can construct the corresponding resolution proof. Compare this with what we have said about the Prolog goal execution process when we first introduced the Socrates example. Then, sit back and appreciate what you have learned.

Appendix A

Recursive Programming

Recursion has been mentioned over and over again in these notes. It is not just a Prolog phenomenon, but one of the most basic and most important concepts in computer science (and mathematics) in general.

Some people tend to find the idea of recursive programming difficult to grasp at first. If that's you, maybe you'll find the following helpful.

A.1 Complete Induction

The concept of recursion closely corresponds to the induction principle used in mathematics. To show a statement for all natural numbers, show it for a base case (e.g., $n = 1$) and show that from the statement being true for a particular n it can be concluded that the statement also holds for $n + 1$. *This proves the statement for all natural numbers n.*

Let's look at an example. You might recall the formula for calculating the sum of the first n natural numbers. Before one can use such a formula, it has to be shown that it is indeed correct.

Claim: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ (induction hypothesis)

Proof by complete induction:

$n = 1:$ $\sum_{i=1}^{1} i = 1 = \frac{1(1+1)}{2}$ ✓ (base case)

$n \rightsquigarrow n+1:$ $\sum_{i=1}^{n+1} i = \sum_{i=1}^{n} i + (n+1)$ (induction step, using the hypothesis)

$= \frac{n(n+1)}{2} + (n+1) = \frac{(n+1)(n+2)}{2}$ ✓

A.2 The Recursion Principle

The basic idea of recursive programming, the *recursion principle* is the following: To solve a complex problem, provide the solution for the simplest problem of its kind and provide a rule for transforming such a (complex) problem into a slightly simpler problem.

In other words, provide a solution for the *base case* and provide a *recursion rule* (or *step*). You then get an algorithm (or program) that solves every problem of this particular problem class.

Compare: Using *induction*, we prove a statement by going from a base case *"up"* through all cases. Using *recursion*, we compute a function for an arbitrary case by going through all cases *"down"* to the base case.

Recursive definition of functions. The factorial $n!$ of a natural number n is defined as the product of all natural numbers from 1 to n. Here's a more formal, recursive definition (also known as an inductive definition):

$$\begin{array}{lcll} 1! & = & 1 & \text{(base case)} \\ n! & = & (n-1)! \cdot n \quad \text{for } n > 1 & \text{(recursion rule)} \end{array}$$

To compute the actual value of, say, 5! we have to pass through the second part of that definition 4 times until we get to the base case and are able to calculate the overall result. That's a recursion!

Recursion in Java. Here's a Java method to compute the factorial of a natural number. It is recursive (for didactic reasons; note that this is not the best way of implementing the factorial in Java).

```
public int factorial(int n) {
  if (n == 1) {
    return 1;                    // base case
  } else {
    return factorial(n-1) * n;  // recursion step
  }
}
```

Recursion in Prolog. The definition of a Prolog predicate to compute factorials:

```
factorial(1, 1).          % base case

factorial(N, Result) :-  % recursion step
  N > 1,
  N1 is N - 1,
  factorial(N1, Result1),
  Result is Result1 * N.
```

Take an example, say the query `factorial(5, X)`, and go through the goal execution process step by step, just as Prolog would—and just as you would, if you wanted to compute the value of 5! systematically by yourself.

Another example. The following predicate can be used to compute the length of a list (it does the same as the built-in predicate `length/2`):

```
len([], 0).             % base case

len([_ | Tail], N) :-  % recursion step
  len(Tail, N1),
  N is N1 + 1.
```

A.3 What Problems to Solve

You can only use recursion if the class of problems you want to solve can somehow be parametrised. Typically, parameters determining the complexity of a problem are (natural) numbers or, in Prolog, lists (or rather their lengths).

You have to make sure that every recursion step will really transform the problem into the next simpler case and that the base case will eventually be reached.

That is, if your problem complexity depends on a number, make sure it is striving towards the number associated with the base case. In the `factorial/2`-example the first argument is striving towards 1; in the `len/2`-example the first argument is striving towards the empty list.

Understanding it. The recursion principle itself is very simple and applicable to many problems. Despite the simplicity of the principle the actual execution tree of a recursive program might become rather complicated.

Make an effort to really understand at least one recursive predicate definition, such as `concat_lists/3` (see Section 2.2) or `len/2` completely. Draw the Prolog goal execution tree and do whatever else it takes.

After you got to the stage where you are theoretically capable of understanding a particular problem in its entirety, it is usually enough to look at things more abstractly:

> "I know I defined the right base case and I know I defined a proper recursion rule, which is calling the same predicate again with a simplified argument. Hence, it will work. This is so, because I understand the recursion principle, I believe in it, and I am able to apply it. Now and forever."

A.4 Debugging

In SWI-Prolog (and most other Prolog systems) it is possible to debug your Prolog programs. This *might* help you to understand better how queries are resolved (it might

however just be really confusing). This is a matter of taste.

Use `spy/1` to put a spypoint on a predicate (typed into the interpreter as a query, after compilation). Example:

```
?- spy(len).
Spy point on len/2
Yes
[debug]  ?-
```

For more information on how to use the Prolog debugger check your reference manual. Here's an example for the `len/2`-predicate defined before.

```
[debug]  ?- len([dog, fish, tiger], X).
*  Call:  (  8) len([dog, fish, tiger], _G397) ? leap
*  Call:  (  9) len([fish, tiger], _L170) ? leap
*  Call:  ( 10) len([tiger], _L183) ? leap
*  Call:  ( 11) len([], _L196) ? leap
*  Exit:  ( 11) len([], 0) ? leap
*  Exit:  ( 10) len([tiger], 1) ? leap
*  Exit:  (  9) len([fish, tiger], 2) ? leap
*  Exit:  (  8) len([dog, fish, tiger], 3) ? leap
X = 3
Yes
```

Your Prolog system may also provide more sophisticated tools (e.g., graphical tools) to help you inspect what Prolog is doing when you ask it to resolve a query.

Index

Planning and Problem Solving

Jeremy Pitt

Draft v0.2: October 2020

Abstract

This is the primer to accompany the lecture slides on Planning for Problem Solving. It covers search space representation, graph theory foundations, and the Prolog specification (and implementation) of the general graph search engine (GGSE). The customisation of the engine for different search algorithms is not covered in this version (v0.2).

1 Introduction

The aim of this primer in Planning and Problem Solving is to introduce the idea of 'problem solving search'. The basic issue is this: suppose that we are in a state, or at a position S (say), and we want to get into state (or be at position) G, then how do we solve this problem – how do we transform, or make the transition, from state (position) S to state (position) G? Or a bit more precisely, how do we transform, or make the transition, from state (position) S to state (position) G, given a set of possible actions to perform $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$?

Therefore, since there is unlikely to be a single transformation which turns state S into state G, we need to work out the sequence of actions (each one taken from the set $\mathcal{A}$) which transforms state S into state G, i.e.:

$$S \xrightarrow{A_i} S' \xrightarrow{A_j} S'' \xrightarrow{A_k} \ldots \xrightarrow{A_l} S''\ldots' \xrightarrow{A_m} G$$

In other words, we are looking for a *path*, from state S, through various intermediate states (S', S'', etc.), until we reach the final, *goal* state G.

To automate problem solving search requires three things. Firstly, it requires a symbolic representation of a *state*. Secondly, it requires the formal specification of *state-change rules* which transform one state into another. This means that we can generate a *graph*, where each of the nodes in the graph represents a state of the problem, and a (directed) link between two nodes represents that one state can be transformed into another by application of one of the state-change rules. Then we can identify all the *paths* in the graph, as sequences of state transitions from one state to another; and of course we are looking for a path which starts with state S and ends in state G. Thirdly and finally, we need a systematic method for generating and testing these paths: this is an algorithm. In fact, we will be more general than this: we will define an *engine* called the *General Graph Search Engine (GGSE)* which can be modified in various ways to give a number of different algorithms.

This primer is therefore structured into two parts. The first part contains the study of state representation and state-change rule specification, and some elementary graph theory, leading to the specification of the GGSE; and secondly, the study of various different algorithms for graph search – breadth first, depth first, iterative-deepening depth first, uniform cost, beam,

Figure 1: The Bear-Box-Honey Problem

best first and A^* — including criteria for evaluating different algorithms, and the modifications required to implement the GGSE. Both parts will be illustrated with examples: in fact, we will start with an example in the next section.

2 Two Motivating Examples

This section presents two motivating examples: first the Bear-Box-Honey (BBH) Problem, and secondly, the FWGC (Farmer-Wolf-Goat-Cabbage) Problem. The latter is a typical example of a *river-crossing* problem.

2.1 Example: Bear-Box-Honey (BBH) Problem

2.1.1 Problem Description

The situation is as illustrated in Figure 1. There is a bear. The bear is in a room. The room has a door and window, but both are locked. So the bear is going nowhere, but the bear is getting hungry.

In the middle of the room there is a hook. From the hook hangs a pot of honey. Bears like honey, but the honey is out of reach.

By the window there is a box. It does not look so heavy that it cannot be moved by a sufficiently hefty beast, a bear say; yet it looks sturdy enough to take the same bear's weight, if the bear sat or stood on the box.

The bear would not be hungry if it could eat the honey. The bear could eat the honey, if only the bear could reach the honey. The bear could reach the honey, If only the bear had something to stand on ...

So the problem is: how does the bear get the honey?

The 'AI planning' way to solve this sort of problem is to use *symbolic* representation and manipulation, rather than numerical processing. In other words, we will use symbols to denote objects in the micro-world description; we will use information structures to represent particular configurations of those objects, which we call *states*; and then we will define logical

rules which define a relation between states. These logical rules define all the ways that one state can be transformed into another by performing an action.

2.1.2 Symbolic State Representation

In general, then, for a problem of this kind, it requires addressing the questions of representation and manipulation, with various sub-questions:

- Representation
  - What information needs to be represented in the state?
  - How is this information best to be represented?
  - What is the initial (start) state?
  - What is the goal (end, target) state?
- Manipulation
  - What actions can be done, and by whom?
  - What are the effects of actions?
  - What are the constraints on actions?

When we have a satisfactory answer to the questions of representation and manipulation, we can address the question of formulating a solution.

So let us consider these questions for the BBH problem. First, for the issue of representation, what are the objects that need to be represented, and what are the properties of the object that are significant? So the 'things' that are mentioned in the description are the bear, the box, the jar of honey, the door, the window, and hook, etc.; the property of the object in which we are interested is its location, i.e. by the door, by the window, on the hook, and so on. From this analysis, we can see that some things in the problem description are significant, and some are just providing contextual richness. So we are interested in the location of the bear and the box, for example; but not so much the door and window, or in fact whether they are locked or not. The door and the window are just providing handles for talking about locations in more 'humanly meaningful' terms rather than just calling them 'location 1', 'location 2', and so on.

So then we could represent information by a set of *attribute-value* pairs, for example $(bear, door)$, $(box, window)$, $(jar, hook)$, meaning the bear is by the door, the box is by the window, the jar is on the hook, etc. Alternatively, we could just record the value of the property and remember whose property it is by virtues of its position in an information structure. This decision-making process is involves decomposition and abstraction, and is an issue that we will return to later. But for now, what would one answer to the question of representation and manipulation for the BBH problem look like.

So to begin with, to define the state, we decide what information needs to be represented in the state, by answering the following questions

- Where's the bear?
  - A location in the room: by the window, in the middle, by the door, which will be represented by the symbols *window*, *middle*, *door*.

- Where's the box?
  - A location in the room: by the window, in the middle, by the door; represented as before.
- Is the bear on the box?
  - This is a boolean value: true or false.
- Has the bear got the honey?
  - Also a boolean.

Note that the middle of the room is considered to be under the hook and we are abstracting away from any numerical precision.

The to determine the representation of the state, in Prolog, we use a 4-tuple, where the position in the state determines which object is being referred to (we need to know this, but the computer does not care, so we can leave it out of the state representation).

So the 4-tuple of state information is this:

(*location-of-bear*, *location-of-box*, *bear-on-box*, *bear-has-honey*)

For example (*door*, *window*, *false*, *false*) is a state: in fact this is the start state. Similarly, (*middle*, *middle*, *true*, *true*) is a state, and this is a goal state, because in the end state, we want the bear to have the honey. In fact, we do not care about any of the other three values, all we want is the bear to have the honey. Therefore the most general characterisation of the goal state is (_, _, _, *true*).

Note that now it should be possible to take any state description and map it to a configuration of the micro-world, as shown in Figure **??**, and equally, given any such configuration, map it into a state.

2.1.3 Symbolic State Manipulation

Having decided on an appropriate state representation, the second step is to specify how to manipulate those states, by defining the *state change* rules which transform one state into another.

Generally, these state change rules define a relation R on states, and it is true statement that two states are in this relation (i.e. $(s, s') \in R$) if (and only if) s is the 'before' state, s' is the 'after' state, and there is some action $A \in \mathcal{A}$ which transforms s into s'. This requires firstly, identifying what actions can be performed, (and possibly by whom); and secondly, identifying the conditions (or constraints) have to be satisfied for the action to be performed in that state, and what effects those have actions on the 'before' state to transform it into the after stTE

Specifically, for BBH problem, the box can't do anything, but the bear can do the following actions:

- Move from an 'old' location to a 'new' location ...
  - ... provided the locations are 'connected', and ...
  - ... afterwards, the bear is in the new location;

- Push the box from one location to another ...
  - ... provided the bear and the box are in the same location, and ...
  - ... provided the locations are 'connected', and ...
  - ... afterwards, the bear and the box are in the new location;
- Climb on the box ...
  - ... provided the bear and the box are in the same location, and ...
  - ... provided the bear is not already on the box, and ...
  - ... afterwards, the bear is on the box;
- Grab the honey
  - ... provided the bear and the box are in the middle location, and ...
  - ... provided the bear is on the box, and ...
  - ... provided the bear does not already have the honey, and ...
  - ... afterwards, the bear does have the honey.

Since it is only the bear that can perform an action in this problem, the actor can be left implicit.

For a more Prolog-like specification of this English description, let us represent the state as a 4-tuple with 4 Prolog 'variables', `(Bear, Box, OnBox, Has)`, each of whose instantiate vales denotes (respectively) the location of the bear, the location of the box, if the bear is on the box (or not) and if the bear has the honey (or not). Using the notation that an underscore represents the value of a variable in new state, the effects and constraints of actions can be summarised as:

- `(Bear, Box, OnBox, Has)` $\overset{move}{\Rightarrow}$ `(Bear_, Box, OnBox, Has), if`
  `connected( Bear, Bear_ )`
- `(Bear, Box, OnBox, Has)` $\overset{push}{\Rightarrow}$ `(Bear_, Box_, OnBox, Has), if`
  `Bear = Box` $\wedge$ `connected( Bear, Bear_ )` $\wedge$ `Bear_ = Box_`
- `(Bear, Box, false, Has)` $\overset{climb}{\Rightarrow}$ `(Bear, Box, true, Has), if`
  `Bear = Box`
- `(middle, middle, true, false)` $\overset{grab}{\Rightarrow}$ `(middle, middle, true, true)`

The conversion of such a specification directly into Prolog is straightforward: for example the first rule becomes:

```
state_change( move, (Bear, Box, OnBox, Has), (Bear_, Box, OnBox, Has) ) :-
    connected( Bear, Bear_ ).
```

and similarly for the other three rules (left as a simple exercise).

We would need the following facts about locations, of course:

```
connected( door, middle ).
connected( middle, window ).
```

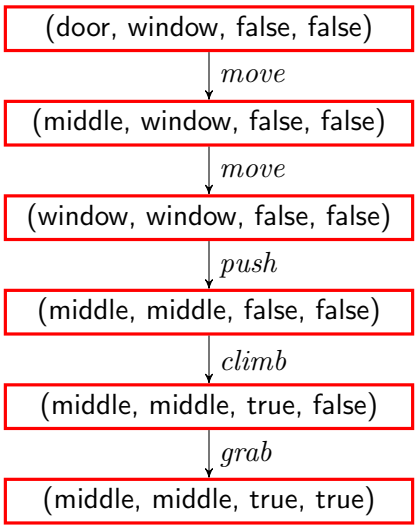


Figure 2: Solution to the BBH problem

```
connected( window, middle ).
connected( middle, door ).
```

Now, we could query the state changes rules with a query of the form:

```
?- state_change( Action, Old_State, New_State ).
```

and if we picked an action and instantiated `Old_State` with a 4-tuple, representing any state, Prolog would compute not only whether or not it is was a true statement, that this old state was in the state-change relation with the new state, but it would also compute what the value of `New_State` would have to be.

For example, the query with the initial state:

```
?- state_change( move, (door, window, false, false), New_State ).
```

would instantiate `(Bear, Box, OnBox, Has)` with `(door, window, false, false)`, so in the new state `Bear_` would be instantiated to `middle`, as the middle is connected to the door, and the other three elements remain the same, so that `(Bear_, Box, OnBox, Has)` is instantiated to `(middle, window, false, false)`.

2.1.4 Solution Formulation

We are now in a position to formulate a solution to the problem. What is the sequence of *permissible* actions – according to the constraints on the state change rules, which can convert the initial state into the final state. Of course, this fiendishly difficult problem could require hours to solve, so the answer is as illustrated in Figure 2.

So to get the computer to do what it as taken us, literally, hours to do, we need a formal representation of the states of the micro-world, and we need to know for what we are searching. All the possible state configurations of the micro-world is going to constitute a *search space*, and what we are searching for in this space is a *path*. We begin with a definition and overview of search spaces.

2.2 The Farmer-Wolf-Goat-Cabbage (FWGC) Problem

2.2.1 Problem Specification

The farmer-wolf-goat-cabbage (FWGC) problem is specified as follow:

- A farmer, wolf, goat and cabbage are on one side of a river with a boat, and they all want to get to the other side.
- The farmer can transport one of the wolf, goat or cabbage in the boat across the river, or he can go by himself.
- If he leaves the wolf with the goat, the wolf will eat the goat; if he leaves the goat with the cabbage, the goat will eat the cabbage.
- How can the farmer transport the wolf, goat and cabbage across the river without anything getting eaten?

2.2.2 Symbolic State Representation

As in the previous problem, we need to decide on a representation of the state, specify the initial state, and specify the goal state.

The representation of the state is:

- 4-tuple = (F, W, G, C)
  - Corresponding to which bank farmer/wolf/goat/cabbage are on, respectively;
  - Can be instantiated to l (left) or r right;
  - There is representation of boat – it is assumed to be on the same bank as the farme.

The initial state:

- (l, l, l, l)

The goal state

- (r, r, r, r)

2.2.3 Symbolic State Representation

The general specification of the state transformers is as before:

- `statechange(RuleName, OldState, NewState) :-`
  computation, *constraint_checking*.

Then there is on one state change (transformer) for each action. Note that we are assuming that only the farmer can row the boat, therefore the only significant actor is the farmer and the only significant actions are what he can do in terms of transporting himself, and possibly something else across the river.

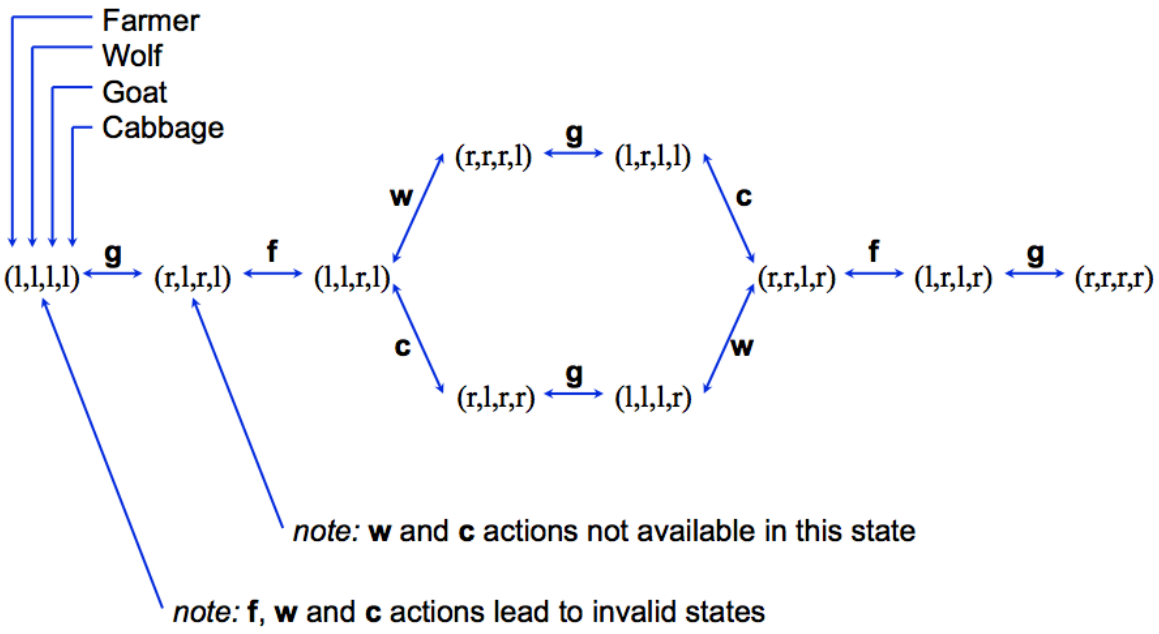


Figure 3: States and Actions for the FWGC problem

- ```
  state_change( farmer, (B,W,G,C), (O,W,G,C) ) :-
     opposite(B,O), opposite(W,G), opposite(G,C).
  state_change( wolf, (B,B,G,C), (O,O,G,C) ) :-
    opposite(B,O), opposite(G,C).
  state_change( goat, (B,W,B,C), (O,W,O,C) ) :-
     opposite(B,O).
  state_change( cabbage, (B,W,G,B), (O,W,G,O) ) :-
     opposite(B,O), opposite(W,G).

  opposite( l, r ).
  opposite( r, l ).
  ```

2.2.4 Search Space

If we applied all the state change rules to all the possible states, and drew an arrow between the states when the constraints were satisfied: the result would be as shown in Figure **??**.

This is the explicit representation again – we can now solve the problem by inspection. But how do we (get Prolog to) generate and inspect the graph, given only the initial state and the state transformers? So we need a way of representing and generating the search space, and a way of inspecting elements so generated.
  ```

3 Search Spaces

3.1 Search Spaces: Some Terminology

In the BBH problem there are $3 \times 3 \times 2 \times 2 = 36$ different possible states. This defines the total number of states in the search space. As we said, actions transform one state into another, so we could feed each of our 36 possible states as an 'old state' into each of the four rules, and see what 'new state' is computed by the rule. Then we could a directed arc between the old state and the new state, and label it with the action A that produced the transformation.

With only 36 states, and only 4 actions, it would be possible to draw it out. Furthermore, by inspection, we could trace a *path* between any one node and any other node by following arcs through a series of intermediate notes. If the first node we picked was the initial state and the last node in the path is a (the) goal state, then this path is a *solution path*, i.e. a sequence of actions which solves the problem.

Unfortunately, if it were so simple, why do we need a computer to do it? And if you were to get a computer to do it, would you – could you? – do it like this, anyway?

To answer these questions, we first need to define some terminology about search spaces, and then need to know some things about search spaces.

For terminology, Firgure 4 shows an example search space. The circles (*nodes* represent states labelled $A, B, \ldots$, the arcs represent transformations between states, labelled $1, 2, \ldots$. Then we define

- the *start state* as a distinguished node in the search space, where the search for a solution path originates;
- the *goal state* as a distinguished node in the search space, where the search for a solution path terminates;
- state change rules, transfomers, operators, are all inter-changeable names for functions from the set of states to the set of states;
- a *search path* (or just *path*) is a sequence of states;
- a *solution path* is a sequence of states which has the initial state as its first state, and the goal state as its last state; and
- the *search frontier* is an imaginary 'contour' drawn in the search space, which defines that amount of the search space has actually been 'seen' or inspected.

.

So, example, the search frontier in Figure 4 would be given by drawing an imaginary line from C, through E, through F, and onto B. In other words we are assuming that what is shown here is a fragment of the search space, and there are operators which act on the states in the search frontier to produce new states (which we don't yet know about so they are not yet drawn).

3.2 Search Spaces: A Few Caveats

There are some features of search spaces that should be borne in mind. Search spaces may:

- get very big, very quickly;

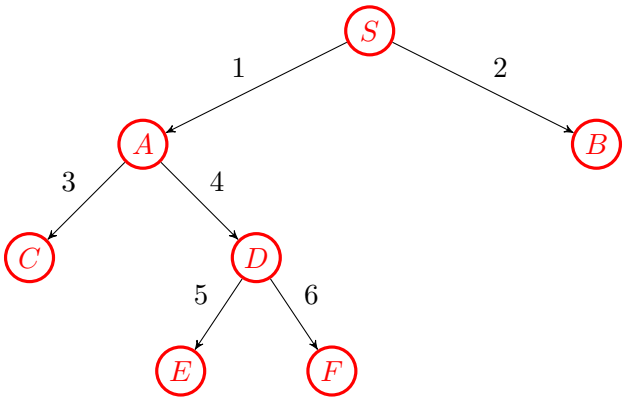


Figure 4: An Example Search Space

- have loops;
- have infinite paths;
- have constraints on the the state transformation;
- have local maxima;
- have multiple paths to the goals state;
- have multiple goal states; and
- have no paths to any goals states, i.e. there are states which are inaccessible from the goal state.

.

3.3 Formulating a Search Space

However, just as there are caveats concerning search spaces, there are also caveats about the *process* of formulating a search space.

If we are going to formulate a search space as graph, we need to:

- Define the state representation;
- Specify the initial state;
- Specify the state transformers (operators).

As alluded to above, there are some general, and specific considerations to be taken in to account in the process. For example deciding what goes in the state description and what is left out; or deciding what effects of actions are significant and what are not. In general, the problem is to find a balance between granularity and abstraction.

Granularity determines what 'depth' of detail is required for the search space formulation to be a 'valid' model of the problem. For example, in the BBH problem, there was no requirement to start measuring the distance of the box from the hook and the arm's reach of the bear standing on tip-toe; it was enough to assume that the bear could not reach the box

standing on the floor, but could if it was standing on the box, and the bear was smart enough to manoeuvre the box into a position under the hook/honey combo to make this possible.

Abstraction determines what 'height' of detail is required for the search space formulation to be a 'valid' model of the problem. Abstraction is the opposite pf granularity: it is the process of removing unnecessary detail from the state representation. A 'good' abstraction removes as much detail as possible, while retaining validity and ensuring that actions are as 'simple' as possible to execute. A good abstraction, balanced data representation and efficient state transformation rules may be the difference to being able to find a useful solution with these methods, and not.

Bearing all this in mind, though, if we can find a way to represent search spaces mathematically, we can then define a way to search for solution paths algorithmically (specifically taking advantage of out Prolog formulation of the problem).

The mathematical representation we will use is a **graph**.

4 Some Graph Theory

A search space can be mathematically represented by a *graph* $\mathcal{G}$. where a graph $\mathcal{G}$ defines a set of paths between nodes. In theory, then, if we had an explicit representation of all the paths in the graph, by inspection, we could check each path to see if its first node represented the initial state and its last node represented the final state. In practice, we don't have such an explicit representation: all we 'know' is the start state, the goal state, and the state change rules. Therefore we need to do two things: firstly, we need to show that this implicit representation of the graph – as a single node and a set of state-change rules – is equivalent to the explicit representation; and secondly, give this implicit representation, we need a systematic method of generating the paths emanating from the start state, and testing each of these paths to see if it ends in a goal state.

4.1 Preliminaries

In the following, we assume a familiarity with set theory, first-order logic and inductive definition, but will give a basic recap.

Set theory: a set is, informally, a collection of objects. Formally, a set is a binary relation between any object x a set A. The notation $x \in A$ means that x is a member of the set A, (it is true that x is a member of A). It is also said that x is an *element* of A. Note that a set is an object as well, so one set can be a member of another set as well. A set can also have no members: this is the empty set, denoted by $\emptyset$. A set can be defined explicitly, either by listing its members between braces, for example $\{1, 4, 9, 16\}$, or generatively, by specifying a type of object and the condition that it must satisfy, written $\{object|condition\}$, for example $\{x|\exists y \in N_1.x = y^2 \wedge x < 20\}$, i.e. this is the set of objects x such that it is the square of a member y in the positive natural numbers, and is less than 20. There are some standard operations on sets, in particular intersection of two sets A and B, written $A \cap B$, which is the set of objects which are members of A and members of B, and the union of two sets, written $A \cup B$, which is the set of objects which are either a member of A, or a member of B, or of both. We will also need set difference, $A - -B$, which is the set of objects in A but not in B. (For fun, intersection can be defined in terms of set difference thus: $A \cap B \equiv A - -(A - -B)$, see if you can work it out.)

First order logic: since it already cropped up in the overview of set theory, we need the existential quantifier, $\exists$, which is used to construct a statement which is true if there is an object in a domain which satisfies a condition, as we have just seen. So, $\exists x.P(x)$ is true, if the predicate P is true of at least one constant c substituted for the variable x, so $P(c)$ is true). Eventually we will need the universal quantifier, $\forall$, which is used to construct a statement which is true if every object in a domain satisfies a condition, i.e. $\forall x.P(x)$ is true, if the predicate P is true of every constant c substituted for the variable x. We also need the logical connectives 'and' ($\wedge$), 'or' ($\vee$), 'implies' ($\rightarrow$), and 'if and only if' ($\leftrightarrow$). (We study logic much more deeply in the second part of the course, on reasoning.)

Finally, we need inductive definition of sets. To define a set A inductively, we need to specify three things: firstly, we need to give the basis, i.e. we specify some elements which are in the set A. Secondly, we need to state one or more rules which specify how to construct a new element of the set from an existing element of the set. Thirdly, we need to specify the closure, that no other elements are in the set; this is usually implicitly assumed.

4.2 Explicit Graph Representation

A graph $\mathcal{G}$ can be defined as a 3-tuple:

$$\mathcal{G} = \langle N, E, R \rangle$$

where:

- N is the set of nodes
- E is the set of edges
- R is the **incidence relation** $R : N \times E \times N$

Referring back to the example graph given in Figure 4, we can see:

$$N = \{S, A, B, C, D, E, F\}$$
$$E = \{1, 2, 3, 4, 5, 6\}$$
$$R = \{(S, 1, A), (S, 2, B), (A, 3, C), (A, 4, D), (D, 5, E), (D, 6, F)\}$$

4.3 Paths

Given the definition of a graph as 3-tuple, we can also define a path by all of its *paths*.

In general, we will represent a path as a single node or a comma-separated list of nodes enclosed in square brackets, so, for example, $[S]$, $[A, S]$, $[D, A, S]$, $[E, D, A, S]$, $[F, A, S]$ are all paths. Note that:

- Some sequences of nodes are *not* paths in the graph;
- A singleton node is a path. Otherwise, every pair of adjacent nodes in a path have to be connected by an edge in the incidence relation for this path to be a path in the graph (i.e. of the example paths given above, the path $[F, A, S]$ is *not* a path in the graph, the others are.
- We're going to read these paths 'back to front', i.e. right to left, so in the path $[D, A, S]$, S is the first node and D is the last node

We also need a couple of useful functions for the development in the sequel. These two functions are *frontier* and *cons*:

$$frontier : path \to N$$
$$cons : N \times path \to path$$

So *frontier* takes a path and returns a node; this node will be frontier node of the path, which is the last node in the path, so the first node (the head) of the list. The function *cons* returns a new path by prepending (prefixing) a path with a node. Note that instead of writing $cons(n, p) = \ldots$, we will write $[n \mid p]$, i.e. we will use Prolog list notation with the head and tail of a list.

From the definition of G, we can give an inductive definition of the paths (and sub-paths) in G, starting from a specific node $S \in N$. Generally this we be written $P_{\mathcal{G}(S)}$, but when S is obvious from context, we will just write $P_{\mathcal{G}}$:

$$P_{\mathcal{G}(S)} = \bigcup_{i=0}^{\infty} P_i$$
$$P_0 = \{[S]\}$$
$$P_{i+1} = \{[n' \mid p] \mid \exists p \in P_i.(frontier(p), e, n') \in R\}$$

This states that the paths in a graph $\mathcal{G}$ are given by the distributed union (from $i = 1$ to $i = \infty$), of all the sets P_i. The basis is given by P_0, which is defined by the path consisting of just the start state, i.e. $[S]$. Then the construction rule is that the members of the P_{i+1}th set can be constructed from members of the P_ith set by finding a path $p \in P_i$ such that if $frontier(p) = n$ and $(n, e, n') \in R$ (i.e. n is connected to n' by e in the incidence relation R), then $[n' \mid p] \in P_{i+1}$, i.e. the path $[n' \mid p]$ is a member of the P_ith set.

Again, referring back to the example graph given in Figure 4, we can see:

$$P_0 = \{[S]\}$$
$$P_1 = \{[A, S], [B, S]\}$$
$$P_2 = \{[C, A, S], [D, A, S], \}$$
$$P_3 = \{[E, D, A, S], [F, D, A, S]\}$$
$$P_4 = \emptyset$$
$$\ldots$$

We are given P_0, which is just $\{[S]\}$, the single path consisting of just the start state S. If we look at the condition for generating members pf P_1, this is the only path that satisfies the existential quantifier, and $frontier([S]) = S$. There are two members of R that make the quantified statement true, namely $(S, 1, A)$ and $(S, 2, B)$. So we have two new paths and $P_1 = \{[A, S], [B, S]\}$. We can continue this process with successive increments of i, looking first at each path p in the $P_{i-1}th$ set, and for each path seeing what values (if any) of e and n' make $(frontier(p), e, n') \in R$ true, and making a new path for the P_ith set by prefixing p with n', i.e. $[n' \mid p]$.

For some value of i, basically when we try to look for where to go from B, C, E and F, there will no members of R to make the existentially quantified statement true, so there will be no paths generated, so the set for this value of i will be empty. Then there will be no value that can can substituted for $\exists p \in P_i$, so P_{i+1} will be empty too, and so on.

As a result for this example:

$$P_G = \bigcup_{i=0}^{\infty} P_i = P_0 \cup P_1 \cup P_2 \cup P_3 \cup P_4 \ldots = \ldots$$

but P_4 is the emptyset, and so for any value $i > 4, P_i = \emptyset$. The union of any set with the emptyset is the same set, so in fact:

$$P_G = \bigcup_{i=0}^{\infty} P_i = P_0 \cup P_1 \cup P_2 \cup P_3$$

4.4 Implicit Graph Representation

Generally, though, search spaces are far too large to generated exhaustively like this, and in any case, we might not be interested in the whole space, and we still need a procedure for generating and testing paths in $P_{\mathcal{G}}$. So instead of defining a Graph as a 3-tuple with all the nodes, edges and incidence relation explicitly listed, we are going to give an implicit graph definition as a 2-tuple, based on identifying one node, and a set of state transformation rules.

First we define a set Op of operators (i.e. state-change rules, transformers, etc)

$$Op = \{op_1, op_2, \ldots, op_n\}$$

Each $op_i \in Op$ is a partial function:

$$op_i : N \mapsto (N \times E)$$

Then, a graph $\mathcal{G}'$ can be (implicitly) defined as a 2-tuple:

$$\mathcal{G}' = \langle S, Op \rangle$$

where:

- S is a node ($S \in N$)
- Op is a set of operators (defined as above)

4.5 Paths (Revisited)

We can use the implicit definition of a graph to give inductive definitions of N and R, and indeed the paths in the graph.

The nodes (and edges) of the graph are defined by:

$$\begin{aligned} N_G &= \bigcup_{i=0}^{\infty} N_i \\ N_0 &= \{S\} \\ N_{i+1} &= \{n' \mid \exists n \in N_i . \exists op \in Op . op(n) = (n', e)\} \end{aligned}$$

The incidence relation is defined by:

$$\begin{aligned} R_G &= \bigcup_{i=0}^{\infty} R_i \\ R_0 &= \{(S, e, n) \mid \exists op \in Op . op(S) = (n, e)\} \\ R_{i+1} &= \{(n, e, n') \mid \exists (\_, \_, n) \in R_i . \exists op \in Op . op(n) = (n', e)\} \end{aligned}$$

Finally, the paths in the graph are defined by:

$$P_{G'} = \bigcup_{i=0}^{\infty} P'_i$$
$$P'_0 = \{[S]\}$$
$$P'_{i+1} = \{[n \mid p] \mid \exists p \in P'_i . \exists op \in Op . op(\mathit{frontier}(p)) = (n, e)\}$$

Referring back to the example, suppose $Op = \{l, r\}$ and

$$l(S) = (A, 1) \quad r(S) = (B, 2)$$
$$l(A) = (C, 3) \quad r(A) = (D, 4)$$
$$l(D) = (E, 5) \quad r(D) = (F, 6)$$

No surprises: the paths defined $\mathcal{G}'$ are the same in $\mathcal{G}(S)$:

$$P'_0 = \{[S]\}$$
$$P'_1 = \{[A, S], [B, S]\}$$
$$P'_2 = \{[C, A, S], [D, A, S]\}$$
$$P'_3 = \{[E, D, A, S], [F, D, A, S]\}$$
$$P'_4 = \emptyset$$
$$\ldots$$

4.6 Equivalence of $P_{\mathcal{G}}$ and $P_{\mathcal{G}'}$

We can show in general that $\mathcal{G}' = \langle S, Op \rangle$ defines the same graph (as a set of paths) as $\mathcal{G}(S) = \langle N, E, R \rangle$ provided:

$$(n, e, n') \in R \leftrightarrow \exists op_i \in Op . op_i(n) = (n', e)$$

Proposition. Defining $\mathcal{G} = \langle N, E, R \rangle$, and $\mathcal{G}' = \langle S, Op \rangle$ $(S \in N)$, then if $(n, e, n') \in R \leftrightarrow \exists op_i \in Op . op_i(n) = (n', e)$, $P_{\mathcal{G}}(S) = P_{\mathcal{G}'}$.
Proof. By induction on paths.
Base case. $P_0 = \{[S]\} = P'_0$, by definition, and we are done.
Induction Step. Assume (by induction) that $P_{i-1} = P'_{i-1}$. We need to show that for any path p, $p \in P_i \leftrightarrow p \in P'_i$.

$\rightarrow$: $[n'|p] \in P_i \rightarrow [n'|p] \in P'_i$. If $[n'|p] \in P_i$ then there must have been a path $p \in P_{i-1}$, with frontier node n (i.e. $\mathit{frontier}(p) = n$) and $(n, e, n') \in R$. If $(n, e, n') \in R$ then $\exists op_i \in Op . op_i(n) = (n', e)$. Since $p \in P'_{i-1}$ (by induction), and there is an operator in Op such that $op_i(n) = (n', e)$, then $[n'|p] \in P'_i$, as required.

$\leftarrow$: $[n'|p] \in P'_i \rightarrow [n'|p] \in P_i$. If $[n'|p] \in P'_i$ then there must have been a path $p \in P'_{i-1}$, such that $\mathit{frontier}(p) = n$ and $\exists op_i \in Op . op_i(n) = (n', e)$. If $op_i(n) = (n', e)$ then $(n, e, n') \in R$. Since $p \in P_{i-1}$ (by induction), $\mathit{frontier}(p) = n$ and $(n, e, n') \in R$, then $[n'|p] \in P_i$, as required.

QED. Black box etc. etc.

We now have a mathematic basis for the formal specification of the search spaces defined by out formulation of the search spaces for, for example, the BBH and FWGC problems.

We now need a method for generating (using the state transformers) all the paths originating from the specified start state, and checking if a path so generated ends in a goal state. Such a path is called a *solution path*.

In fact, we can be more general: we can make an *engine*, and then we can parameterise that engine in such a way that we can find many different ways of generating all the paths originating from from the specified start state. This is the **General Graph Search Engine**.

5 The General Graph Search Engine (GGSE)

5.1 Some Definitions

To begin with, we need some definitions, and these definitions are represented in Prolog. We define:

- **State**: a data structure, representing the state of a problem;
- **Node**: a data structure, including at least the problem-state, but possibly other information as well;
- **Path**: a sequence of nodes;
- **Graph**: a set of paths.

The Prolog representation of these definitions is as follows:

- A problem-state is represented by a **term**, depending on the formulation of the problem;
- A node is represented by a **tuple**, including at least the problem-state;
- A path is represented by a **list** (of nodes);
- A graph is represented by a **list** (of paths, so a list of (lists of nodes)).

5.2 The GGSE in Pseudo-Declarative Specification

A more-formal-than-English but less-formal-than-Prolog declarative specification of the GGSE can be given as follows:

- A graph G can be searched for a Solution Path SP, if
  - we can pick a path P in G, AND
  - the frontier node of path P is N, AND
  - the problem-state of node N is S, AND
  - IF S is a goal state, THEN P is a Solution Path SP.
- A graph G can be searched for a Solution Path SP, if
  - we can pick a path P in G, AND
  - we set $OtherPaths \leftarrow G - \{P\}$, AND
  - the frontier node of path P is N, AND
  - the set S of one step extensions of P is given by $S = \{N' | op(N, \_) = N'\}$, AND
  - we set $OtherPaths \leftarrow \{[N'|P] | N' \in S\}$, AND
  - We can make a bigger graph $G^+ = NewPaths \cup OtherPaths$, AND
  - Graph G^+ can be (recursively) searched for a Solution Path SP.

5.3 The GGSE in Prolog

Given that declarative specification, the GGSE is very succinctly expressed in Prolog, as follows:

- ```
  search( Graph, SolutionPath ) :-
      choose( Graph, Path, _ ),
      frontier_node( Path, Node ),
      state_of( Node, State ),
      goal_state( State ),
      Path = SolutionPath.
  search( Graph, SolutionPath ) :-
      choose( Graph, Path, OtherPaths ),
      one_step_extensions( Path, NewPaths ),
      add_to_paths( NewPaths, OtherPaths, GraphPlus ),
      search( GraphPlus, SolnPath ).
  ```

  The intuitive reading of the predicates used in GGSE is as follows

- `choose`: given as input a graph, return one path as the second argument and the rest of the paths in the graph as the third argument;
- `frontier_node`: given a path, returns the last node in the path;
- `state_of`: given a node, returns the problem-state;
- `goal_state`: tests if a state is a goal state or not;
- = is Prolog's unfiy;
- `one_step_extensions`: given a path, compute all the paths that can be generated by applying each of the state-change rules to the path's frontier node;
- `add_to_paths`: combine the paths left over from `choose` with the paths generated by `one_step_extensions` to make a bigger graph.

  Then, we will query the GGSE with queries of this form:

```
?- search( +Graph, ?SolutionPath ).
```

where `Graph` will be instantiated by a graph, normally a list of just one path, that path being a list of just one node, the start state; and `SolutionPath` will be an uninstantiated variable. Then, when (or if) the query succeeds, `SolutionPath` will be instantiated to to a path that starts with the start state and ends with the goal state.

Typical examples would be, for the BBH problem:

```
?- search( [ [ (door,window,false,false) ] ], SP ).
```

and for the FWGC problem:

```
?- search( [ [ (1,1,1,1) ] ], SP ).
```
```

How the GGSE processes these queries, declaratively-speaking, is as follows – recalling that Prolog will try to prove the first clause, and if that does not succeed, it will backtrack and try the second clause. The first clause is the base case: we have found a path in the graph which is a solution path. The second clause is the recursive call: we expand one of the frontier nodes on one of the paths to make a 'bigger' graph which covers more of the search space, and then search this 'bigger' graph for a solution path, and so the process repeats.

- A graph G(=`Graph`) can be searched for a Solution Path SP(=`SolutionPath`), if
  - `choose` succeeds, picking a path P(=`Path`) in G (we don't care, for the moment, about the others, hence the anonymous variable _),
  - AND the frontier node of path P is N(=`Node`),
  - AND the problem-state of node N is S(=`State`),
  - AND S is a goal state,
  - AND P unifies with Solution Path SP, which it does, since `SolutionPath` is a variable; so the query succeeds, and returns yes with `SolutionPath` (at every recursive call up to the query) unified with `Path`.
- A graph G(=`Graph`) can be searched for a Solution Path SP(=`SolutionPath`), if
  - `choose` succeeds, by:
    - Picking a path P(=`Path`) in G, AND
    - Setting *OtherPaths*(= `OtherPaths`) $\leftarrow G - \{P\}$
  - AND `one_step_extensions` succeeds, by:
    - Getting the frontier node N of path P, AND
    - Computing the set N' of all the new nodes reachable from N, by applying all the state change rules to N, AND
    - Setting *NewPaths*(= `NewPaths`) $\leftarrow \{[n' \mid P] \mid \exists n' \in N'\}$,
  - AND `add_to_paths` succeeds, by:
    - Making a bigger graph G^+(=`GraphPlus`) from *NewPaths* and *OtherPaths*
  - AND Graph G^+ can be (recursively) searched for a Solution Path SP.

The definition of the supplementary predicates is some case straightforward, and in others, critical. We will will start with the straightforward cases.

The predicates `state_of` and `frontier_node` are 'access' predicates and depend on how the nodes of the graph have been formulated and how the paths are being represented. In the simplest case, there is no extra information being stored in the nodes, and so the state representation *is* the node representation; and we represent paths as a list of modes in reverse order, so the frontier node is the head of the list:

```
state_of( S, S ).

frontier_node( [H|_], H ).
```

Checking that a state is a goal state (or not) depends on the problem formulation and its specification: the definition of the `goal_state` predicate may need extra processing or it may just unify.

The definition of `one_step_extensions` requires the Prolog meta-predicate `findall/3`, which returns the set of answers that satisfy a query. It is effectively like set generation (as discussed earlier), i.e.:

$$S = \{x \mid p(x) \wedge q(X)\} \quad \equiv \quad \texttt{findall(X, (p(X), q(X)), S)}$$

Thus the definition is given by:

```
one_step_extensions( [Node|Path], NewPaths ) :-
        state_of( Node, State ),
        findall( [NewNode,Node|Path],
                 ( state_change( _, State, NewState ),
                   make_node (Node, NewState, NewNode ) ),
                 NewPaths ).
```

Note that the set-membership condition here is actually two clauses, the application of the state change operators ($p(X)$), and the node constructor predicate `make_node` ($q(X)$). `finall` works just as well with one condition, two (as here), three, four, or more. More refined versions of the engine will actually have four conditions, as it will also consider the *costs* associated with this path, and the detection of *loops*.

For now though, in the simplest cases, where we do not need to consider any information in the previous node and the node *is* the state, note the definition of the second condition is trivial:

```
make_node( _, S, S ).
```

As an aside, this explains why, in the graph theory section, sequences of nodes were represented in reverse order; and the | notation for prepending was introduced. The Prolog list representation makes it very easy to access the head of a list, both when it was required to get at the frontier node; and when it was required to compute the 'one step extensions' of a path (i.e. by prepending each of the nodes reachable from its frontier node by applying the state change rules to that node)

This also means that GGSE can be more succinctly expressed in Prolog, as follows:

- ```
  search( Graph, [Node|Path] ) :-
      choose( Graph, [Node|Path], _ ),
      state_of( Node, State ),
      goal_state( State ).
  search( Graph, SolnPath ) :-
      choose( Graph, Path, OtherPaths ),
      one_step_extensions( Path, NewPaths ),
      add_to_paths( NewPaths, OtherPaths, GraphPlus ),
      search( GraphPlus, SolnPath ).
  ```

The unification of the path selected by `choose` and the solution path in the head of the first clause is undone on backtracking, and we can access the frontier node as the head of the list directly, without needing another 'access predicate'.
  ```

5.4 Completing the GGSE

It will have been noticed that two predicates have not yet been further defined: so what about `choose` and `add_to_paths`?]

It turns out that different 'behaviours' of the GGSE are generated by different definitions of these two predicates, and this gives **different algorithms**.

The different algorithms, and their different performance characteristics, are studied next.

6 Summary

This primer has introduced a qualitative approach to problem-solving, based on a declarative specification of knowledge, by

- Formulating the problem as a search space, by
- Specifying a start state and a set of state transformers; which
- This defines a graph;
- The graph contains a solution-path if there is a path in the graph which originates with the start state and terminates with goal state.

We then specified a general engine for generating all the paths in a graph defined this way Next we will examine specific algorithms for generating and searching through those paths looking for the one we want.

Computational Logic Primer
(Supplement to Artificial Intelligence Course Notes)

Jeremy Pitt
Department of Electrical & Electronic Engineering
Imperial College London

This version: October, 2018

ABSTRACT

The aim of the Computational Logic (CL) primer is to introduce the idea of what is a valid logical argument, to demonstrate how an argument can be represented, and to show the how the process of making an argument can be mechanised, and hence automated in a Prolog program. The CL primer starts with the presentation of three, successively richer formal languages for knowledge representation (propositional, predicate and modal logics), and works through the mechanics of making an argument, culminating in the presentation of Prolog and explaining how Prolog works. Prolog is the basic computational logic tool for dealing with 'real' problems, and this is illustrated with a discussion of a general framework for representing and reasoning about actions and the effects of actions (the Event Calculus), how this general calculus can be realised directly in Prolog, and by way of example its application in dealing with a problem in regulatory compliance.

1 Introduction to CL Primer

Computational logic is concerned with the application of logic to computer science. It involves the *formalisation* of domain-specific information in logical form, and the *mechanisation* of rules of inference or reasoning, in order to compute new information, the answer to a query, a plan of action, and so on. When the formalisation is specified in a logical form technically called *horn clauses*, and mechanisation is given by an inference engine that implements a reasoning technique called *resolution*, then in effect we have the declarative programming language *Prolog*. With Prolog, we can write precise, concise, logical specifications of a problem: we focus on describing the properties of our domain (i.e. the formalisation of the nature of a problem), leaving unspecified the details of how to compute a solution to that problem. However, it turns out such a (declarative) specification can be its own implementation, and we can use the (implicit) mechanisation given by the Prolog inference engine to compute the solution. For this reason, declarative programming is often associated with the slogan "algorithm = logic + control".

Computational logic has been successfully applied in many domains, including traffic and transportation, medical and life sciences, and legal representation and reasoning. Indeed, some of the earliest applications of computational logic were in expert systems for medical diagnostics and the formalisation of the British Nationality Act. Since then scientific and technological advances have seen applications developed in the verification of safety-critical software and hardware, the specification and verification and debugging of embedded systems, and for verification of security protocols. As well as computer science, computational logic has also proved extremely useful in artificial intelligence (e.g. machine learning, natural language processing, intelligent agents, etc.), is the key to providing sufficient expressive power to unlock to full potential of the Semantic Web, and is extensively used in cognitive science to test theories of human cognition. Computational logic is a key

tool in the interdisciplinary study of Artificial Intelligence and Law, and therefore a key concern for applied inter-disciplinary research.

In this computational logic primer, we will take a gentle tour through logical form, beginning with propositional logic, and working our way through predicate logic and onto modal logic. We will then consider the issue of formalisation of domain-specific information in these languages, and then consider the issue of mechanization of rules of inference and reasoning. This will lead us on to the programming language Prolog, from where we can consider a general calculus of action, events and time called the Event Calculus [KS86] and it realization in Prolog (there are other representation languages and implementation routes, which are not covered by this primer). By way of example of the application of computational logic to dealing with 'real world' problems, we consider the issues of regulatory compliance, conflict prevention and alternative dispute resolution in law. This is illustrated through analysis of a detailed example applying computational logic to a problem of regulatory compliance.

2 Three Logical Languages

Like any language, Logic has its own syntax – for English, the rules for making correct sentences; in logic, the rules for making well-formed formulas – and its own semantics – the way to understand the meaning of the formulas. Furthermore, also in common with natural languages, Logic has its own 'dialects', and each 'dialect' has its own syntax and semantics. In this section we will give simple explanations of the syntax of three (progressively more expressive) logical languages: propositional logic, predicate logic, and modal logic.

2.1 Propositional Logic

In this section, we give the syntax and semantics of propositional logic.

2.1.1 Syntax

The syntax for writing well-formed formulas (*wff*) of propositional logic are:

$$
\begin{array}{lcl}
wff & = & wff\ connective\ wff \\
 & | & \neg\ wff \\
 & | & (\ wff\ ) \\
 & | & atomic\_formula \\
atomic\_formula & = & a \mid b \mid c \mid p \mid q \mid r \mid p1 \mid p2 \mid \ldots \\
connective & = & \wedge \mid \vee \mid \rightarrow \mid \leftrightarrow
\end{array}
$$

Example. p, q, $\neg p$, $p \wedge q$, $((p \rightarrow q) \wedge (\neg t \vee z \vee q1)) \leftrightarrow (a \rightarrow b)$ are all wff of propositional logic. $\wedge\ p\ q \leftrightarrow z$ and $a \rightarrow b) \leftrightarrow (c$ are not.
We will use P, Q, R etc. to represent arbitrarily complex wff, i.e the symbol P could represent any atomic or complex formula of propositional logic.
These are the rules for writing correct formulas (sentences, statements) of propositional logic. The language consists of a rule for writing propositional symbols (p, q, etc.) which are atomic wff, and rules for combining wff using connectives to make complex formulas. Note that $\neg p$ is an atomic formula, but $\neg(p \vee q)$ is a complex formula. Thus atomic formulas have no connectives.
The symbols $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$ are read 'not', 'and', 'or', 'implies', and 'if and only if' respectively. What this means is discussed in the next section.

2.1.2 Semantics

The semantics of propositional logic first requires assigning a meaning to atomic formulas. This requires doing two things:

- Mapping the propositional symbol to some real world statement,
- Assigning a truth value to that statement.

The semantics of propositional logic is completed by assigning a meaning to complex formulas. We do this by:

- The principle of *bivalence*: every statement (i.e. every atomic or complex formula) is *either* true *or* false, but never both. The two values are variously represented by the pairs T (true) and F (false), 1 and 0, $\top$ and $\bot$, etc.;
- Truth assignment: every atomic formula is assigned a meaning – i.e. a truth value, true or false – by a truth assignment (also called an interpretation, or a valuation);
- Compositionality: a complex wff gets its meaning (its truth value) from the meaning of the connective and the meaning of its component parts (i.e. the two wff joined by the connective as specified by the rule of syntax above).

The meaning of the connectives are given by truth tables, as shown below:

P	Q	$\neg Q$	$P \wedge Q$	$P \vee Q$	$P \rightarrow Q$	$P \leftrightarrow Q$
0	0	1	0	0	1	1
0	1	0	0	1	1	0
1	0	1	0	1	0	0
1	1	0	1	1	1	1

Table 1 - Truth Tables for standard connectives

There are logics where we relax this condition of bivalence, where the meaning of a statement may be 'fuzzy' (e.g. half true, so we have many values), or undefined (so we have three values, true, false and 'don't know'), but in no logic can a statement be both true and false at the same time.

With a truth assignment and truth tables we can define the notion of satisfaction. Given a truth assignment τ, a formula (proposition) P, and a set of formulas S, we say the following:

τ satisfies P	if and only if	P evaluates to true with assignment τ
τ falsifies P	if and only if	P evaluates to false with assignment τ
P is satisfiable	if and only if	there is an assignment that satisfies P
A set of formulas S is satisfiable	if and only if	there is an assignment that satisfies each and every formula in S
A formula P is a tautology	if and only if	for every possible assignment, τ satisfies P
A formula P is a contradiction	if and ony if	for every possible assignment, τ falsifies P (i.e. no assignment satisfies P)
A formula P is a logical consequence of a set of formulas S	if and only if	any assignment τ that satisfies S, also satisfies P

Example. Let P be the formula $p \rightarrow (q \rightarrow p)$. Given the truth assignment:

$$\tau = \{p > 1, q > 0\}$$

we can work out, using τ and compositionality, that P evaluates to true, so τ satisfies P, and P is satisfiable. Furthermore, using truth tables as well, we can check that P is satisfiable for all four possible (relevant) truth assignments, so P is also a tautology.

Example. Let S be the set of formulas $\{p \to q, q \to r\}$. The truth assignment:

$$\tau = \{p > 0, q > 1, r > 1\}$$

satisfies both formulas in S, so S is satisfiable. Let P be the formula $q \to r$. We can check that for each of the possible eight truth assignments, whenever S is satisfied, so is P. P is a logical consequence of S, or in other words S entails P.
There is a notational convention for logical consequence, denoted by the symbol $\models$. If P is a logical consequence of S, we write:

$$S \models P$$

If S is the empty set, we write simply:

$$\models P$$

In this case, we also say that P is a *theorem* of our language, in other words, P can be proved without assuming any premises at all.

Example. $p \to (q \to p)$ and $((p \to q) \to p) \to p$ are both theorems of propositional logic.

We can use truth tables to validate the claim of this example using truth tables. (Remember, if P is a theorem then P is a tautology; from the above, a tautology is a wff that is true under every possible truth assignment):

p	$\to$	$(q$	$\to$	$p)$
0	1	0	1	0
0	1	1	0	0
1	1	0	1	1
1	1	1	1	1

$((p$	$\to$	$q)$	$\to$	$p)$	$\to$	p
0	1	0	0	0	1	0
0	1	1	0	0	1	0
1	0	0	1	1	1	1
1	1	1	1	1	1	1

Mathematically, $\models$ defines a relation, the entailment relation, between sets of formulas and single formulas. This is important, because this is the logical relation that, eventually, we want to compute.

That's it: that is all there is to syntax and semantics of propositional logic. Don't be misled by the simplicity: propositional logic is a powerful tool, and for some applications, it's enough. Don't assume that propositional logic is just a stepping stone to the 'real thing', e.g. the languages we are going to discuss in the next two sections. However, equally don't assume that propositional logic is all powerful: the weak point might have been observed in the step where we made the mapping from the propositional symbol to the 'real world' statement. We

ignored the content and the structure of the statement. If we want to take this into account, we need a more expressive language. One such language is predicate logic.

2.2 Predicate Logic

In this section, we give the syntax and semantics of predicate logic.

2.2.1 Syntax

The syntax of predicate logic is rather more complex than that of propositional logic. This reflects that we are going to make more detailed statements about the 'real world'. However, because we don't want to talk about the whole of the real world, we are only going to make statements about that bit in which we are interested. We call that bit the *domain of discourse*.

So to begin with, in predicate logic, there are now two types of symbol:

- Variable symbols, which represent objects in the domain of discourse, without explicitly naming them, and
- Constant symbols, which are the objects in the domain of discourse.

Constants are further sub-divided into object constants, function constants, and variable constants.

Object constants are used to name a specific element of the domain of discourse. We will use lower case alphanumeric strings to denote constants.
A function constant is used to designate a function on elements of the domain of discourse. A function constant is denoted either by an operator (e.g. +, *, etc.) or a lower case alphanumeric string. A function is applied to other constants and returns an object constant as a result; therefore a function can be applied to another function. Every function has an arity, which is the number of arguments it takes.

A predicate constant is used to designate a relation on elements of the domain of discourse. A relation symbol is either a mathematical operator (e.g. <, >, etc.) or also a lower case alphanumeric string. Relations also have an arity, which is the number of places (i.e. elements) in the relation.

A variable will also be denoted by a lower case alphanumeric string (don't worry about everything being denoted by a lower case alphanumeric string – context and convention will make it clear whether the string denotes a variable or constant (and what sort of constant it denotes)). However, the value of a variable can only be an object constant. This is what we mean when we refer to first order predicate logic. If we have variables which can takes values of predicate constants, then we have a second order language (and we don't want to go there, just yet).

We are now in a position to define the terms of a first order language. These are given by:

- A variable is a term: x, y, z, …
- A constant symbol is a term: $3wood$, $5iron$, $putter$, …
- If t_1, t_2, …, t_n are all terms, and f is a function symbol of arity n, then $f(t_1, t_2, \ldots, t_n)$ is a term.

Notice that a function of arity 0 is just an object constant. A predicate of arity 0 is a sentence constant. Also, note that functions denoted by operators are usually written 'infix', i.e. as 2 + 2, rather than +(2, 2).

Thus variables, object constants and function constants are all terms, but not predicates. But don't go feeling sorry for predicates, because applying predicates to objects is what generates sentences (well-formed formulas of predicate logic), and it is sentences that have

truth values (i.e. a meaning, as we shall see in the next section). However, predicate logic allows us to have three types of sentence, i.e. three types of formula:

- Atomic formulas: if t_1, t_2, …, t_n are all terms and p is a predicate of arity n, then $p(t1, t2, , tn)$ and $\neg p(t1, t2, , tn)$ are formulas (atomic formulas);
- Logical formulas: if A and B are formulas, then $\neg A$, $A \wedge B$, $A \vee B$, $A \rightarrow B$, and $A \leftrightarrow B$ are formulas (logical formulas); and
- Quantified formulas: if x is a variable, and A is a formula, then $\forall x.A$ and $\exists x.A$ are both formulas (quantified formulas).

Intuitively, a formula of the form $\forall x.A$ is intended to mean that the formula A is true, no matter what object (in the domain of discourse) that the variable x stands for in that formula. Because it is (supposed to be) true for every object, this is called a universally quantified formula. A formula of the form $\exists x.A$ is intended to mean that there is at least one object that x can stand for, and A is true. Because it is (supposed to be) true for every object, this is called an existentially quantified formula. We will formalise this notion in the next section, when we discuss the semantics of predicate logic.

Example. Suppose we had a domain of discourse with two objects, 3wood and 5iron. Suppose we had three predicates, wood, iron, and golfclub. Then:

- $wood(3wood)$ and $\neg wood(5iron)$ are both atomic formulas;
- $wood(3wood) \wedge \neg wood(5iron)$ is a logical formula;
- $\forall x.iron(x) \rightarrow golfclub(x)$ is a universally quantified formula, stating that every object that is an iron is a golf club;
- $\exists x.wood(x) \wedge iron(x)$ is an existentially quantified formula, stating that there is an object which is a wood and an iron.

We can now give a syntax of wff (well-formed formulas) of predicate logic, in the style used for propositional logic in section 2.1.1:

wff = *wff connective wff*
| ¬ *wff*
| (*wff*)
| *atomic_formula*
| *quantifier variable* . *wff*

atomic_formula = *predicate_constant* (*term_list*)
| *predicate_constant*

term_list = *term* ',' *term_list*
| *term*

term = *variable* | *object_constant* | *function_constant*

function = *function_constant* (*term_list*)

connective = ∧ | ∨ | → | ↔

quantifier = ∀ | ∃

variable = [*alphanumeric*]

object_constant = [*alphanumeric*]

function_constant = [*alphanumeric*]

predicate_constant = [*alphanumeric*]

Given this syntax, it is possible to combine quantified formulas, just like atomic formulas, using the connectives. In this case, the scope of a quantifier only extends over the wff in which it is found, i.e. to the left or the right of the connective. In addition, a quantified formula is a wff, and so can be nested 'inside' another quantified formula. In this case, the scope of the quantifier extends over the entire formula, while the order of the nesting can be extremely important to the meaning. Compare:

$$\forall x.\exists y.hitsfurther(x,y)$$
$$\exists x.\forall y.hitsfurther(x,y)$$

The first formula states that for any object, there is another (at least one) object that it hits further than (and not necessarily the same one for each object); and the second formula states that there is an object that hits further than all the others.

One last point to note: $\forall x.p(y)$ is a wff formula of predicate logic, even though the quantified variable (x) does not occur in the formula, and the variable that does occur in the formula (y) is not bound to a quantifier. The former is called vacuous quantification, the latter is called a free variable (as opposed to a bound variable). One final syntactic terminology: a formula with no free variables is called a closed sentence; a formula with no variables is called a ground formula.

However, a formula such as this can still have a meaning, so that all wff do have a meaning (as we would expect), provided we define our semantics right.

2.2.2 Semantics

The meaning of a wff in predicate logic is the same as the meaning of a wff of propositional logic, i.e. a truth value, true or false. The problem is that given the extra complexity in the syntax (predicates rather than symbols, variables, and quantifiers, etc.) the semantics is correspondingly more complex. However, the basic idea is relatively simple: we build a model of the world, and then make statements about. We make those statements in first-order logic. The question is how we relate statements to the model.

The answer was first provided by the mathematician Alfred Tarski, who provided a definition of what it 'means' for a formula to be true with respect to a structure. This structure consists essentially of an interpretation (for the language), which provides an appropriate treatment for the terms of the language, and a valuation, which assigns values (objects) to the variables. Then we can give a meaning to a formula with respect to the interpretation.

So to begin, let L be a first-order language. An interpretation of L is a pair <D,I>, where:

- D is a non-empty set, the set of individuals in the domain of discourse;
- I is a mapping which maps constant, function and predicate symbols (of L) to actual constants, functions and relations:
    - If c is a constant symbol, then $I[c] \in D$
    - If f is an n-place function, then $I[f] \in D^n \rightarrow D$
    - If p is an n-place predicate, then $I[p] \in D^n$

A valuation V of L given D is a function from variable symbols (of L) into D. An important piece of notation: $V\{o/x\}$ denotes the valuation that maps the variable x to the object o, but is otherwise exactly the same as V.

A term assignment I_V corresponding to I and V is a mapping from terms to objects and is defined by:

- $I_V[x] =_{def} V(x)$
- $I_V[c] =_{def} I[c]$
- $I_V[f(t_1, t_2, \ldots, t_n)] =_{def} I[f](I_V[t_1], I_V[t_2], \ldots, I_V[t_n])$

A term assignment gives us a relative notion of truth that we require for predicate logic, called (as with propositional logic), satisfaction. As a matter of notational convention, the fact that a formula A is satisfied by and interpretation I and a variable assignment V is written:

$$|=_{I,V} A$$

We say that "A holds" or "A is true" in I under V, i.e. A holds/is true relative to the interpretation I and the valuation V. If we want to say "A does not hold" (in I under V), we write $|\!\neq_{I,V} A$.

We can the define satisfaction by induction on the structure of well-formed formulas.

Atomic formulas. A term assignment satisfies an atomic formula if the tuple formed by the designated terms is an element of the relation designated by the predicate constant, i.e.:

- $\models_{I,V} p(t_1, t_2, \ldots, t_n)$ if and only if $(I_V[t_1], I_V[t_2], \ldots, I_V[t_n]) \in I[p]$

Logical formulas. A term assignment satisfies a logical formula under the following conditions:

- $\models_{I,V} \neg A$ if and only if $\not\models_{I,V} A$
- $\models_{I,V} A \wedge B$ if and only if $\models_{I,V} A$ and $\models_{I,V} B$
- $\models_{I,V} A \vee B$ if and only if $\models_{I,V} A$ or $\models_{I,V} B$
- $\models_{I,V} A \rightarrow B$ if and only if $\not\models_{I,V} A$ or $\models_{I,V} B$
- $\models_{I,V} A \leftrightarrow B$ if and only if $\models_{I,V} A$ and $\not\models_{I,V} B$, or $\not\models_{I,V} \neg A$ and $\models_{I,V} \neg B$

Quantified formulas. A term assignment satisfies a logical formula under the following conditions:

- $\models_{I,V} \forall x.A$ if and only if for every $o \in D$, $\models_{I,V\{o/x\}} A$
- $\models_{I,V} \exists x.A$ if and only if there is at least one $o \in D$, $\models_{I,V\{o/x\}} A$

2.3 Modal Logic

Modal logic, like predicate logic, recognizes the fact that, truth is relative. In predicate logic, a formula was (or was not) true relative to an interpretation. Equally, modal logic recognises that there may be many different 'modes' of truth. Thus modal logic can be thought of the logic of qualified truth. The particular modes that provide the qualification are many and various: for example time, knowledge, deontics (duty), and the 'standard' mode, necessity and possibility.

In this section, we give the syntax and semantics of (propositional) modal logic. In doing so, we will gloss over a significant amount of historical development, philosophical discussion and technical machinery that has brought modal logics to the level of understanding that we enjoy today. For a full introduction, consult a good introductory textbook, like Hughes and Cresswell [HC96]; a good online resource is the Stanford Encyclopedia of Philosophy, http://plato.stanford.edu/contents.html.

2.3.1 Syntax

Propositional modal logic is an extension to, not an alternative to, ordinary propositional logic. In one sense the extension comes from, like predicate logic, 'peering into' the statement about the 'real world' that we want to formalise, and discerning that it has a content, or structure, that is not adequately expressed by a single, simple propositional symbol. The content that we discern is concerned with representing statements that have modalities, i.e. a particular mode with respect to which the statement is expressed.

As indicated above, there are many 'modes', each one yields a different modal logic. Commonly encountered modes include:

Logic	*Expression*
Modal Logic	It is necessary that … (Necessarily …)
	It is possible that … (Possibly …)
Temporal Logic	It will be the case that …
	It will always be the case that ….
	It was always the case that …
	It was once the case that …

	Since …
	Until …
Deontic Logic	It is obligatory that …
	It is permitted that …
	It is forbidden that …
Doxastic Logic	It is believed that …

Narrowly construed, modal logic is concerned with studying logical consequence between statements containing expressions of the form 'necessarily' and 'possibly'. Under this narrow reading, we simply need to extend syntax of ordinary propositional logic with 2 rules:

$$wff = modal\_operator\ wff$$
$$modal\_operator = \Box \vee \Diamond$$

Note that we are not going to discuss the extension of predicate logic to modal logic with quantifiers. It is straightforward, but not without complications which are a distraction for present purposes.

Example. As well as all the formulas of propositional logic, the following are all wff formulas of (propositional) modal logic:

$$\Box p \qquad \Box p \rightarrow \Box q \qquad \Box(p \rightarrow q) \qquad (\Box p \wedge \Diamond q) \leftrightarrow \Box\Diamond r \qquad \Box\Box\Box\Box p \rightarrow p$$

These operators allow us to express qualified truth. For example, in a certain domain, we might want the statement "Tom is a cat" to be necessarily true, i.e. $\Box p$, and the statement "Tom is a pet" to be possibly true, i.e $\Diamond p$, because we want to allow for the circumstances where Tom is still a cat, but gone all feral on us.

We can have a finer grained structure on the modal operators by parameterising them. Then we can write formulas of the form $[A]p$, with an appropriate reading for A. One reading gives dynamic logic where the parameters are actions, so the intuitive reading of the formula is "after action A, p is true.

So, now we have (a range of) possible readings of these operators, which provide an intuitive meaning; the next issue is to give these formulas a formal meaning.

2.3.2 Semantics

As we have seen, and unlike propositional and predicate logics the meaning of a modal logic formula may not be truth-functional: for example knowing that p is true does not imply that $\Box p$ is true; nor does knowing that p is false imply that $\Diamond p$ is false.

One way to tackle this problem is to observe that in propositional logic, one symbol 'picks out' one statement. In predicate logic, a constant symbol picked out an object from a domain of discourse, a predicate symbol picked out sets of objects, and so on. This is effectively an extensional state description, i.e. an explicit definition. What we require is an *intensional* reading, i.e. by defining the meaning implicitly. This is to what Leibniz referred to as possible worlds. Hence the semantics of modal logic is given in terms of possible worlds. Intuitively one could think of possible worlds as follows:

- In propositional logic: possible world is a truth assignment (ie one row of the truth table);
- In predicate logic: a possible world is an interpretation;
- In modal (propositional) logic: we have a *set* of propositional worlds (similarly modal predicate logic). One member of this set is 'the real world', the rest are orientations of the world as it might otherwise be. We have ways of accessing the other worlds from the real world, and a way of picking out which formulas are true in these worlds. Collectively, these three elements are known as a model.

We therefore define the semantics of modal logic relative to a model M = <W,R,V> where:

- W is a non-empty set, the set of all possible worlds;
- R is a binary relation on W, $R \subseteq (W \times W)$, called the accessibility relation;
- V is a function from propositional symbols to subsets of W. For each symbol, this is referred to as its denotation. For p, this is written as: $[\![p]\!]$

Generally, the pair <W,R> is referred to as a frame and together with a valuation function V this triple is called a model. The meaning of a wff of propositional modal logic is then given as follows.

Atomic formulas.

- $\models_{\mathcal{M},\alpha} p$ if and only if $\alpha \in [\![p]\!]$
- $\models_{\mathcal{M},\alpha} \neg p$ if and only if $\alpha \notin [\![p]\!]$

Logical formulas

- $\models_{\mathcal{M},\alpha} \neg A$ if and only if $\models_{\mathcal{M},\alpha} A$
- $\models_{\mathcal{M},\alpha} A \wedge B$ if and only if $\models_{\mathcal{M},\alpha} A$ and $\models_{\mathcal{M},\alpha} B$
- $\models_{\mathcal{M},\alpha} A \vee B$ if and only if $\models_{\mathcal{M},\alpha} A$ or $\models_{\mathcal{M},\alpha} B$
- $\models_{\mathcal{M},\alpha} A \rightarrow B$ if and only if $\models_{\mathcal{M},\alpha} A$ or $\models_{\mathcal{M},\alpha} B$
- $\models_{\mathcal{M},\alpha} A \leftrightarrow B$ if and only if $\models_{\mathcal{M},\alpha} A$ and $\models_{\mathcal{M},\alpha} B$, or $\models_{\mathcal{M},\alpha} \neg A$ and $\models_{\mathcal{M},\alpha} \neg B$

Modal formulas.

- $\models_{\mathcal{M},\alpha} \Box A$ if and only if for all worlds β in $\mathcal{M}$ such that $\alpha R \beta, \Box\Box\Box \models_{\mathcal{M},\beta} A$
- $\models_{\mathcal{M},\alpha} \Diamond A$ if and only if for some world β in $\mathcal{M}$ such that $\alpha R \beta, \Box \models_{\mathcal{M},\beta} A$

Example. Let $\mathcal{M}$ be the model $\mathcal{M}$ = <W,R, $[\![]\!]$ > where W = {α,β,γ}, R = {$\alpha\beta$, $\beta\alpha$, $\beta\gamma$}, and P is the function $[\![p]\!]$ = {α,β}, $[\![q]\!]$ = {β,γ}. We can represent this model diagrammatically, as shown in Figure 1- A Model for Modal Logic.

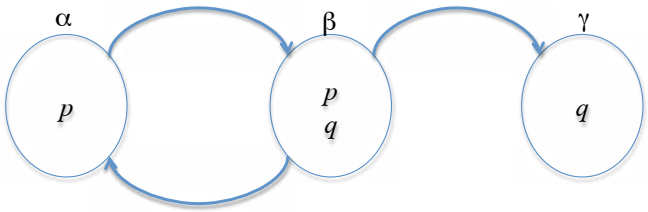


Figure 1- A Model for Modal Logic

Intuitively, we can think of each possible world, together with the denotation function P, picking out those propositions which are (contingently) true at that world. The accessibility relation can be thought of as defining the ways in which we can change the world 'we are in' to reach another configuration of the world (i.e. a possible world) in which different propositions may be true.

Example. Using the given semantics and the example model, we can determine which of the following wff are true (relative to this model):

$\models_{\mathcal{M},\alpha} p$	$\models_{\mathcal{M},\beta} p$	$\models_{\mathcal{M},\gamma} \neg p$
$\models_{\mathcal{M},\alpha} p \wedge q$	$\models_{\mathcal{M},\beta} p \rightarrow q$	$\models_{\mathcal{M},\gamma} p \rightarrow q$

$\models_{\mathcal{M},\alpha} \Box p$	$\models_{\mathcal{M},\beta} \Box p$	$\models_{\mathcal{M},\gamma} \Box p$
$\models_{\mathcal{M},\alpha} \Diamond q$	$\models_{\mathcal{M},\beta} \Diamond q$	$\models_{\mathcal{M},\gamma} \Diamond q$
$\models_{\mathcal{M},\gamma} \Diamond p \Box$	$\models_{\mathcal{M},\alpha} \neg\Diamond\neg p$	$\models_{\mathcal{M},\beta} \neg\Box\neg q$

These formulas, by row, are respectively true, true, and true; false, true, and true; true, false, and true; true, true, and false; and false, true, and true.

There are a couple of points to note on this example. The first is relatively straightforward: the equivalence of $\Box$ and $\neg\Diamond\neg$, and similarly, $\Diamond$ and $\neg\Box\neg$. The second may be harder. Note that $\Box p$ is true in γ, while $\Diamond p$ is false. This because we could re-express the semantics of the modal formulas as expressions of predicate logic, thus:

- $\models_{\mathcal{M},\alpha} \Box A \quad \leftrightarrow \forall\beta.\ \alpha R\beta \rightarrow \models_{\mathcal{M},\beta} A$
- $\models_{\mathcal{M},\alpha} \Diamond A \leftrightarrow \exists\beta.\ \alpha R\beta \wedge \models_{\mathcal{M},\beta} A$

Now note that for the $\Box$ rule, if there is no world β such that $\alpha R\beta$, so the antecedent of the implication is always false. If that is the case, then the formula (from truth tables) is always true. In the $\Diamond$ rule, the connective is an 'and'. If one component is always false, (i.e. $\alpha R\beta$ for all β) then the formula is never true.

In any case, we can now say what it means to be valid in modal logic.

A formula A of modal logic is valid:

- In a *model* $\langle W,R,V\rangle$ if $\models_{\mathcal{M},\alpha} A$ for all α in W;
- In a *frame* $\langle W,R\rangle$ if it is valid in all possible V;
- In a *class* of frames or models *C* if it is valid in every member of *C*.

2.3.3 'Normal' Modal Logics

So far, we have just concentrated on defining the syntax and semantics of three logical languages. The syntax defined which were the wff of the language; the semantics defines the meaning of the wff (in terms of true or false, with respect to a truth assignment, interpretation or model). The semantics is also important for allowing us to determine which wff and arguments are valid.

A calculus is a logical system for proving which formulas are valid, i.e. those formulas which can be proven from the empty set of premises, $\models P$, and those for proving arguments, i.e. a formula which can be proven from a non-empty set of premises, $S \models P$.

A *calculus* is simply a logical system for proving valid formulas and arguments. There are, however, many ways to do this. One way is to use an axiomatic system. An axiomatic system defines:

- A language (i.e. all the wff);
- A set of axioms (or rather axioms schemata), which are taken as given;
- A finite set of rules called inference rules.

A *proof* in an axiom system is then a sequence of wff $P_1, P_2, \ldots, P_n$ such that each P_i in the sequence is an axiom, or it follows from two formulas P_j and P_k, (j < i, k < i) by applying one of the inference rules.

Another way, called natural deduction, for the same language, has no axioms and just a set of inference rules. Yet another way, called the sequent calculus, has one axiom and several rules.

However, for any proof system, we will need to show the following: firstly, *soundness*: if we make an argument, we want to be sure that this is a valid entailment; and secondly, *completeness*: if there is a valid entailment, and we want to be sure we can make an argument for it.

The reason for this slight digression into calculi (it is, after all the focus of Section 4) is because it has been a subject of considerable debate which axioms you have to add to a modal logic to get a usable logical system, i.e. one which respects our intuitions about the 'meaning' of the modality and allows us to make 'meaningful' arguments in modal logic.

Let C be all the possible models, with every single possible accessibility relation you can think of. We can identify subsets of C in which the accessibility relation satisfies a certain mathematical relation or relations. It turns out that these conditions correspond to characteristic axioms, which we add to our calculus. It also turns out that these characteristic axioms correspond to an intuitive understanding of certain modalities. In effect, this gives us different logics. Although there are many possible modal logics, there are 15 'normal' modal logics (normal in the sense that the have intuitively appealing semantics, good formal proof systems, and reasonable decision procedures). We summarise these here:

Logic		Condition on R	Characteristic Axiom
K	none		$\Box(p \rightarrow q) \rightarrow (\Box p \rightarrow \Box q)$
T	reflexive	$\forall\alpha.\ \alpha R\alpha$	$\Box p \rightarrow p$
B	symmetric	$\forall\alpha\beta.\ \alpha R\beta \rightarrow \beta R\alpha$	$p \rightarrow \Box\Diamond p$
4	transitive	$\forall\alpha\beta\gamma.\ \alpha R\beta \wedge \beta R\gamma \rightarrow \alpha R\gamma$	$\Box p \rightarrow \Box\Box p$
D	seriality	$\forall\alpha.\exists\beta.\ \alpha R\beta$	$\Box p \rightarrow \Diamond p$
5	euclidean		$\Diamond p \rightarrow \Box\Diamond p$

Note that the Euclidean condition is the result when R is reflexive, symmetric and transitive (essentially it means that every world is accessible from every other). The other logics are obtained as different conditions on R are combined. Figure 2 - 'Map' of the 15 normal modal logics, shamelessly ripped from the Stanford Encyclopedia of Philosophy, shows a map of the 15 normal modal logics and their relation to each other.

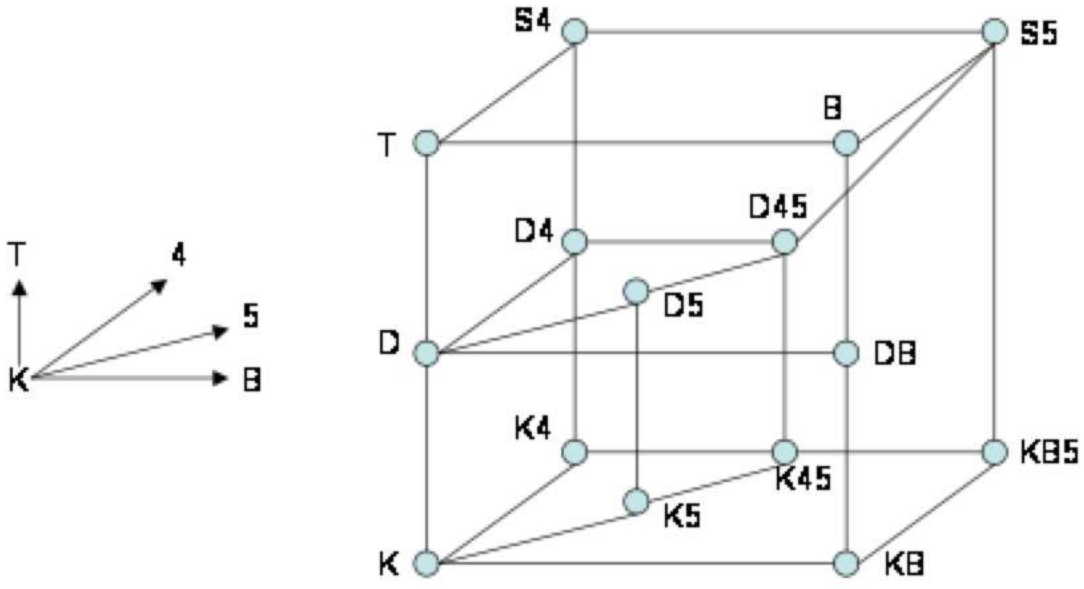


Figure 2 - 'Map' of the 15 normal modal logics

Thus each axiom corresponds to a condition on a frame. Therefore, once the intuitive 'interpretation' of a modality has been decided, the appropriate conditions on R can be identified. This defines the validity of arguments, and in turn determines the axioms that need to be added to the proof system. We can then proceed to hold valid arguments in this modal logic.

For example, let us suppose that the reading of the □ modality is required to be "it will always be the case that". This suggests that our understanding of possible worlds is "moments of time" and our reading of R is "earlier than". Now we can construct models where we have three worlds, say α, β and γ, and in R we have $\alpha R\beta$ and $\beta R\gamma$. Now we can quite happily have $\Box p$ true in α and p not true in γ, which violates our intuitions about the ordering of moments of time (if α is earlier than β and β is earlier than γ, then α should be earlier than γ) and in consequence our intended meaning of "it will always be the case that".

So what we have to do, is only worry about that class of models which do not violate these intuitions. Clearly, the class of models we want is that one where R is transitive, so we add the 4 axiom to our proof system. The logic which allows us to make valid arguments respecting the meaning of our modality is K4, i.e. K4 is the logic which is adequate (both sound and complete) for making 4-valid arguments.

We can carry on in this vein as we wish. One might also insist that there is no last moment in time (finessing any physical or metaphysical debate). In which case, we also require that our models satisfy the seriality condition, D. In this case, we add the D axiom to our proof system, and D4 (= K + D + 4) is the logic we will use for making valid KD4 arguments.

We can cement this relation between conditions and axioms as follows. First, we can consider the class of all models, and demonstrate that the axiom schema is *not* valid; then, we can show that the schema is valid in the class of all models in which the corresponding condition on R *does* hold.

3 Knowledge Representation

We now have a formidable amount of technical machinery at our disposal. The question then is: what is it good for. One answer, is that it is good for knowledge representation: that is the formalisation of information in a form that is better disposed to rigorous analysis than plain English statements. In one sense, the syntax and the semantics come together perfectly to provide us the tools for formalisation of information and mechanisation of the calculus (the axioms and rules) to make arguments. And not just any old arguments: correct, right and proper arguments, i.e. valid arguments.

In this section we will briefly consider the issue of conceptualisation of information, 'translating' English into logical form, and then the issues in formal deduction, before going on, in Sections 4 and 5, to consider the mechanisation of that formal deduction (i.e. putting the computation into computational logic).

3.1 Conceptualisation

3.1.1 Conceptualisation in Propositional Logic

In propositional logic, the conceptualisation is only applied to complete sentences, which are not analysed further, i.e. without regard for their internal structure.

Example. In the House at Pooh Corner, Winnie the Pooh seeks the advice of his friend Owl. On getting to Owl's house, he finds on the door a bell, a door-knocker, and a note, saying: "please ring if an answer is required. Please knock if an answer is not required". What should Winnie the Pooh to get Owl's attention?

One possible conceptualisation is:

$answer$	= an answer is required
$ring$	= ring if an answer is required
$knock$	= knock if an answer is not required

i.e. we conceptualise both statements as propositional symbols. However, this is not very helpful for reasoning, so we have to 'delve inside' the structure and look for key (logical) words like "if", "and" and "or" to analyse the sentences as complex formulas with connectives. In this case, the "if" suggests the following formalisation:

$$answer \rightarrow ring$$
$$\neg answer \rightarrow knock$$

Given that, in the story, Pooh wants an answer, he should be able to work out what to do; as indeed you can too. (Being a "bear of very little brain", he rings the ringer, knocks the knocker, rings the knocker and knocks the ringer – which is actually quite smart, but a reasoning process beyond mechanisation with current principles, as we shall see, but there you go.)

3.2 Conceptualisation in Predicate Logic

Using predicate logic, a more detailed conceptualisation is possible. As might be expected the conceptualisation identifies those elements required of an interpretation, as discussed above. In other words, a conceptualisation gives a concrete representation of 'things' in the 'the real world', as objects, functions and relations. Therefore a conceptualisation is a triple consisting of:

< domain of discourse, functional basis, relational basis >

Domain of Discourse:

- Consists of objects that are things that exist in the real world;
- Can be concrete (physical objects), like animals, blocks, golf clubs, etc.; or abstract (numbers, concepts like justice, and so on); or fictional like Winnie the Pooh, Piglet, etc.;
- Not every object in the real world needs to be represented in the conceptualisation, only those 'things' which are of interest. As we said earlier, the collection of those things is called the domain of discourse, or universe. Even though we may not represent everything, the domain may still be infinite.

Functional Basis:

- Define one kind of inter-relationship between objects, and maps sets of objects into objects;
- Functions are also objects that can be generated from other objects;
- Not every function needs to be represented in the domain of discourse; as before only those of interest need be represented. The set of functions that are represented is called the functional basis of the conceptualisation.

Relational Basis:

- Define another kind of inter-relationship between objects; these are important because these relationships have truth values, and truth values are our meanings;

- A relation can be between any number of objects (a relation between n objects is called an n-place relation);
- A domain of discourse of size b has b^n possible n-tuples. Every n-ary relation is a subset of these b^n tuples. Therefore an n-ary relation must be one of at most $2^{(bn)}$ sets. Those that are defined constitute the relational basis of the conceptualisation.

It follows from the last note on the relational basis that a domain of two objects has at most 16 possible relations. Therefore a conceptualisation of 2 objects and 17 relations must have at least two relations which are, to all logical intents and purposes, completely indistinguishable. You can argue if this is a good thing or not.

No conceptualisation on any particular topic is unique. Furthermore, two conceptualisations of a given topic may differ completely in the objects, functions and relations they choose to include in the domain, functional and relational bases, so that what can be said in one conceptualisation cannot be said in another.

However, three important issues in formulating a conceptualisation are as follows:

- Reification: there are different ways of doing the same thing. For example, suppose we had a domain dealing with colour. One conceptualisation may choose to make red a relation. Another conceptualisation may make red itself an object, and have a relation colour. Yet another could make colour a function. And so on.
- Grain size: the detail in the conceptualisation must fit the kind of arguments expected. A simple example of this has already been seen in the conceptualisation of propositional logic above. The same applies to conceptualisation in predicate logic. The process of abstraction has to decide what to put in, and what to leave out. Abstraction is often the key to be able to do any reasoning at all.
- Formalising English. Once we have our conceptualisation, we want to make statements with/about it. We could make these statements in English to begin with, to give an intuition, but ultimately we have to find a way of expressing these statements in predicate logic using the conceptualisation.

To address this last issue, in the next Section we look at 'translating' English into logical form. We complete this section with a brief consideration of conceptualisation in (propositional) modal logic.

3.2.1 Conceptualisation in Modal Logic

For conceptualisation in (propositional) modal logic, the same comments apply to formalising symbols and non-modal statements as apply about (ordinary) propositional logic

The only other issue is: what type of modality we have? As we have seen earlier, the intended reading of the modality will determine the particular modal logic we will use to do the formal representation, and by extension (because it picks out particular axioms) which logical reasoning system we will use.

3.3 English in Logical Form

The following is a set of informal guidelines for translating English sentences into statements (wff) of propositional and predicate logic. Note that in English grammar, sentences often of the form "subject predicate" (e.g. Winnie the Pooh [subject] is a bear of very little brain [predicate]". When translating this to logic, the predicate stays the same, but the subject (of the English sentence) is now an object (of the domain of discourse). Oh well.

statement1	p

statement2	q
subject predicate	$predicate(\ subject\ )$
statement1 and statement2	$p \wedge q$
statement1 or statement2	$p \vee q$
if statement1 then statement2	$p \rightarrow q$
statement1, if statement2	$q \rightarrow p$
statement1 only if statement2	$p \rightarrow q$
statement1 if and only if statement2	$p \leftrightarrow q$
statement1, unless statement2	$\neg q \rightarrow p$
every, all, for all, each	$\forall$
there exists, there is, for some, a	$\exists$
every noun predicate	$\forall x.\ noun(x) \rightarrow predicate(x)$
some noun predicate	$\exists x.\ noun(x) \rightarrow predicate(x)$

However, we emphasise that these are only guidelines; there is no substitute for common sense, intuition and experience in this 'translation process'.

So having got our conceptualisation, and made some statements about it, the question is, what are we going to do with it. What we want to do, is make arguments.

4 Argumentation

Earlier, we said that we were interested in describing the properties of our domain (i.e. the formalisation of the nature of a problem), leaving unspecified the details of how to compute a solution to that problem. This is declarative knowledge, and there are many arguments in favour of representing knowledge this way: ease of maintenance, re-use, ease of extension, introspection, and so on.

This representation can be done by a process we called conceptualisation: deciding which objects we wanted to talk about, and then fixing the relationship between them. In fact, conceptualisation can be done in many different ways: Artificial Intelligence is littered with them (semantic nets, frames, scripts, etc.). The motivation for producing a conceptualisation in logical form is that it can be directly reasoned with.

Given a conceptualisation in logical form, we then want to make a set of statements which are true about the world. Since we insist that these statements are true, we will call such a set, a set of assertions. A set of assertions may be consistent or inconsistent (i.e. they do or do not contradict themselves). Given a set of assertions, we want to be able carry out deduction, or inference, on it.

Thus the situation is as illustrated in Figure 3 - Logic for Knowledge Representation. Figure 3 - Logic for Knowledge Representation shows, on the left at back, the 'real world', or rather that bit of it in which we are interested, the domain of discourse (world). One way to think of this is, as Wittengenstein wrote at the start of the Tractatus, the world is the set of facts. Now, we know some of the facts, and using those facts, we can discover new (true) facts. This is the situation shown on the right at the back, world+. These facts (statements) about the world can be represented in English, and the processes of conceptualisation of translation into logical form give us the syntactic representation in our formal, logical language. The semantics of these statements is such that they are true.

As shown in Figure 3 - Logic for Knowledge Representation, we want to establish the connection between the left and the right hand sides of the diagram. One way to do this is to use the semantics, and define an entailment relation (whenever the facts on the left of the figure are true, so is the fact on the right). However, while we can compute this for propositional logic, we can't do it for predicate logic or modal logic (we can't check all the interpretations and possible worlds). However, we can do it if we use the syntactic form to make inferences, provided we make inferences in a way that respects the semantics.

Thus the syntactic form specifies all our (true) information about the world in what is called a knowledge base. Now, as we will see in Automated Reasoning, that specification can be its own implementation – this is a logic program, or Prolog. However, we don't 'run' this program, we 'query' it, or we 'consult' it, intuitively asking "given what we know, is such-and-such fact true"?

The answering process is called logical consequence or proof. It is about making an argument: an argument derives a new assertion from an initial set: we also call this argument a proof. However, it is essential to note that logic is concerned only with the validity of arguments, not with their objective, real world truth; i.e. if you start with garbage, you will prove garbage, but as far as logic is concerned if it is valid garbage, then it is OK.

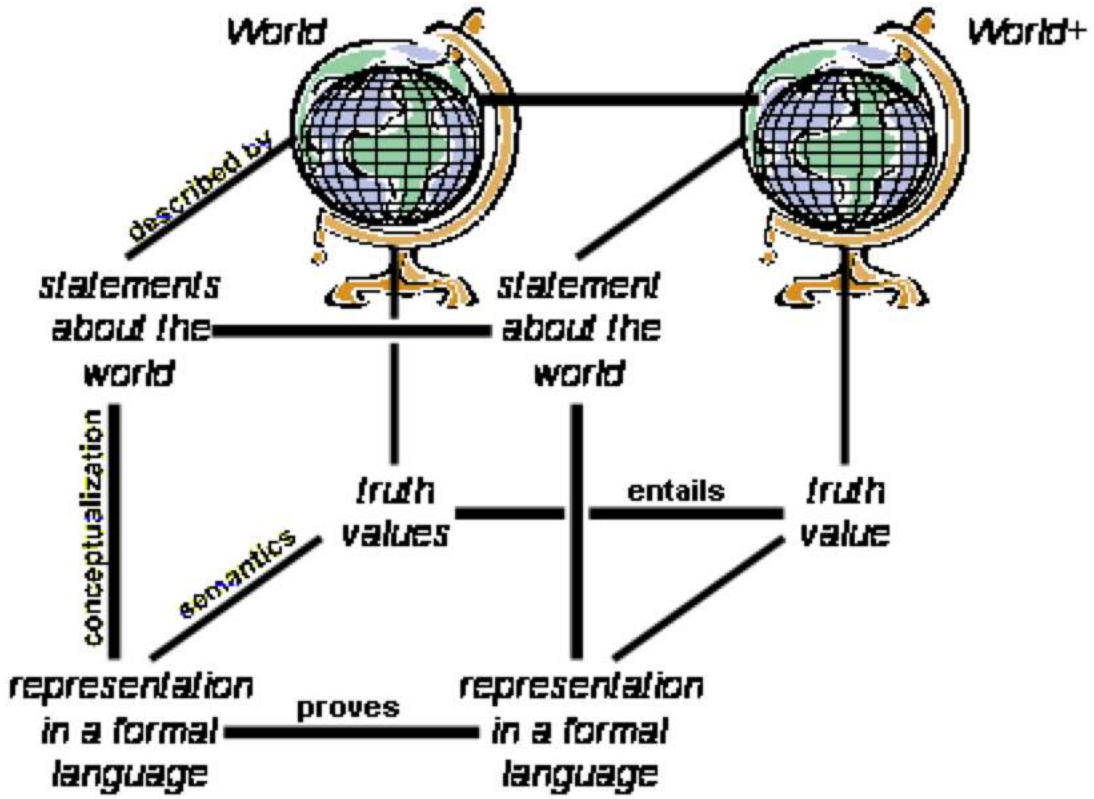


Figure 3 - Logic for Knowledge Representation

Therefore the syntax and semantics of logic allows us to characterize the difference between valid and invalid arguments. To repeat, a calculus or logical system for a language is a set of axioms and rules designed to prove *exactly* the valid arguments that can be stated in the language. In particular, as we said indicated earlier, it must be demonstrated that the reasoning system is *sound*, i.e. that every argument proven using the rules and axioms is in fact a valid entailment. Furthermore, the system should be *complete*, meaning that if there is

a valid entailment then it has a proof in the logical system. Demonstrating soundness and completeness of formal systems is a logician's central concern.

Such a demonstration cannot get underway until the concept of validity is defined rigorously. Formal semantics for a logic provides a definition of validity by characterizing the truth behavior of the sentences of the system. In propositional logic, validity can be defined using truth tables. A valid argument is, as we have seen, one where every truth table row that makes its premises true also makes its conclusion true. However truth tables cannot be used to provide an account of validity in predicate and modal logics because there are no truth tables for expressions such as 'for every', 'for some', 'it is necessary that', 'it is possible that', and so on. We need something else, and this is the business of the next section.

5 Automated Reasoning

Let us summarise so far: we are interested in declarative knowledge representation using formal logic. This requires formalisation of information in a logical language (what we called conceptualisation). We do this because we want to prove theorems (formulas which can show true without any premises) and make arguments (showing a formula follows from a set of formulas asserted to be true). To prove theorems and make arguments we had to have a proof system or calculus: that calculus had to be sound and complete with respect to the stated semantics of the language.

So we have the formalisation element of computational logic. We now want to address the mechanisation element of computational logic. This section takes us right from the basics of normal forms, through proof methods by 'reasoning forwards' from premises in search of conclusions and proof by refutation, leading eventually to Prolog (logic programming) and the automation of a powerful and efficient calculus for predicate and modal logics in Prolog (i.e. using logic programming to implement other logical calculi).

5.1 Normal Forms

At the root of normal forms there are two observations:

- Some formulas are equivalent to others, and
- As a result, we only need $\neg$ and one other connective, and we still have the same expressive power.

We will use the symbol $\cong$ to denote the equivalence of two formulas. Like $\models$, this is *not* a symbol in our language, it is part of the metalanguage that we use to talk about (relations between) formulas of the language. Therefore, note that $P \cong Q$ and $P \leftrightarrow Q$ are different 'kinds' of assertion:

- $P \leftrightarrow Q$ is material equivalence and refers to a single fixed interpretation;
- $P \cong Q$ is semantic equivalence and refers to all (possible) interpretations.

This does mean however that $\vDash P \leftrightarrow Q$ and $P \cong Q$ do mean the same thing.

(As an aside, note that $P \rightarrow Q$ and $P \models Q$ mean different things for the same reason as above, but rather usefully, $\models P \rightarrow Q$ and $P \models Q$ do mean the same thing. We will return to this.)

Some useful equivalences are as follows:

$$
\begin{array}{lcl}
P \rightarrow Q & \cong & \neg P \vee Q \\
\neg(P \wedge Q) & \cong & \neg P \vee \neg Q \\
\forall x.\, P & \cong & \neg \exists x.\, \neg P
\end{array}
$$

$$\exists x.\, P \quad \cong \quad \neg\forall x.\, \neg P$$
$$\neg\neg P \quad \cong \quad P$$

These equivalences are useful because we can reduce formulas of propositional and predicate logic to normal forms. We can do this because, as we said, propositional logic only needs $\neg$ and one other connective: we can define all the other connectives in terms of just this one. Then, by repeatedly applying certain equivalences, we can transform a formula in to its normal form. Typically, a normal form eliminates some connectives, uses others in a restricted manner, and can be used to check for tautologies *without* using truth tables.

This latter is important. We will illustrate this by converting a formula into a normal form of some sort. To do this, we first need some definitions:

- A literal is an atomic formula or its negation. A non-negated atomic formula is a positive literal, a negated formula is a negative literal;
- A maxterm is a literal or a disjunction of literals;
- A minterm is a literal or a conjuction of literals
- A formula is in negation normal form (NNF) if the only connectives are $\neg$, $\wedge$ and $\vee$, and the negation only applies to literal formulas;
- A formula is in conjunctive normal form (CNF) if it is of the form:
  $$P_1 \wedge P_2 \wedge \ldots \wedge P_n$$
  where each P_i is a maxterm (CNF is a conjunction of disjunctions);
- A formula is in disjunctive normal form (DNF) if it is of the form:
  $$P_1 \vee P_2 \vee \ldots \vee P_n$$
  where each P_i is a minterm (DNF is a disjunction of conjunctions).

A method for turning any formula into CNF is a follows:

- Step 1: eliminate $\leftrightarrow$ and $\rightarrow$ by repeatedly applying the equivalences:
  $$P \leftrightarrow Q \cong (P \rightarrow Q) \wedge (Q \rightarrow P)$$
  $$P \rightarrow Q \cong \neg P \vee Q$$
- Step 2: get NNF by pushing negations in, until $\neg$ only applies to literals, by applying the equivalences
  $$\neg\neg P \cong P$$
  $$\neg(P \wedge Q) \cong \neg P \vee \neg Q$$
  $$\neg(P \vee Q) \cong \neg P \wedge \neg Q$$
- Step 3: turn NNF into CNF by pushing disjunctions in, until $\vee$ only connects literals, using the equivalences:
  $$P \vee (Q \wedge R) \cong (P \vee Q) \wedge (P \vee R)$$
  $$(P \wedge Q) \vee R \cong (P \vee R) \wedge (Q \vee R)$$
- Step 4: simplify the CNF by:
  - Deleting any maxterm that contains P and $\neg P$ (since $P \vee \neg P \cong$ true);
  - Deleting any maxterm subsumed by another $P \wedge (P \vee Q) \cong P$;
  - Simplifying common terms, e.g. $(P \vee Q)$ and $(P \vee \neg Q)$ reduces to P.

Example. Construct the CNF of $((p \rightarrow q) \rightarrow p) \rightarrow p$

$$\begin{aligned}
& ((p \rightarrow q) \rightarrow p) \rightarrow p \\
\cong\ & \neg((p \rightarrow q) \rightarrow p) \vee p \\
\cong\ & \neg(\neg(p \rightarrow q) \vee p) \vee p \\
\cong\ & \neg(\neg(\neg p \vee q) \vee p) \vee p \\
\cong\ & (\neg\neg(\neg p \vee q) \wedge \neg p) \vee p \\
\cong\ & ((\neg p \vee q) \wedge \neg p) \vee p \\
\cong\ & ((\neg p \vee q \vee p) \wedge (\neg p \vee p) \\
\cong\ & \mathit{true} \wedge (\neg p \vee p) \\
\cong\ & \mathit{true} \wedge \mathit{true} \\
\cong\ & \mathit{true}
\end{aligned}$$

Now that is interesting. Recall, from a previous exercise, we have shown that the formula $((p \to q) \to p) \to p$ is a theorem of propositional logic. This suggests that we in fact have a procedure for tautology checking using normal forms. In other words, to prove a formula P is valid, all (all) we need to do is reduce the formula to its CNF. Each transformation and simplification preserves semantic equivalence, so if the CNF reduces to true, then the formula itself is semantically equivalent to true, and so is valid.

Otherwise, we have a valuation that falsifies the formula. To see why this is the case, if the formula is not a tautology, then its CNF is a conjunction of disjunctions. In other words, the CNF is of the form $(P_1 \wedge P_2 \wedge \ldots \wedge P_n)$. Each P_i is of the form $(l_1 \vee l_2 \vee \ldots \vee l_m)$. Then we can construct a truth assignment τ that falsifies each l_j:

- If l_j is a positive literal, then $\tau(l_j)$ = false
- If l_j is a negative literal, then $\tau(l_j)$ = true

Another approach is to reduce a formula F to its DNF. In its DNF, we have that:

$$F = P_1 + P_2 + \ldots + P_n$$

where each P_i is of the form $(l_1 \wedge l_2 \wedge \ldots \wedge l_m)$.

If for each P_i, there is an l_j and l_k such that $l_j = \neg l_k$, since $l_j \wedge \neg l_k \cong \mathit{false}$, then the DNF is semantically equivalent to false.

Therefore to prove F, we can refute $\neg F$. If F is a tautology, then $\neg F$ is a contradiction. If $\neg F$ is a contradiction, then $\neg F$ has DNF semantically equivalent to false. If $(\neg F)_{\text{DNF}} \cong \mathit{false}$, then $(F)_{\text{CNF}} \cong \mathit{true}$. If $(F)_{\text{CNF}} \cong \mathit{true}$, then F is indeed a tautology.

5.2 Proves Relation

We now have a way of doing syntactic proof. In fact this is an algorithm, and what this algorithm defines is another relationship between sets of formulas and single formulas.

We shall denote this new relation with the symbol $|\!-$.

As it turns out, there are many different ways of defining this relation, depending on the algorithm we choose to use to determine it. Therefore we should subscript the symbol with the proof system we are using, e.g. $|\!-_{\text{NF}}$ for the normal form proof. However, we will omit the subscript when it is obvious from context which system we are using.

Now there are two important properties of the proves relation, which are how it compares to the entailment relation, which, after all, is what it is supposed to compute. The first property is that, if our syntactic proof system computes a proof, then it really is an entailment. This is the property called *soundness*. Secondly, if there is a member of the entailment relation, we want to be sure that our proof system can compute it. This is the property called *completeness*.

In other symbols, we have:

Soundness:	if $S \,|\!-_{\text{NF}}\, P$, then $S \,|\!= P$	if $(S,P) \in |\!-_{\text{NF}}$, then $(S,P) \in |\!=$
Completeness:	if $S \,|\!= P$, then $S \,|\!-_{\text{NF}}\, P$	if $(S,P) \in |\!=$, then $(S,P) \in |\!-_{\text{NF}}$

Any proof system should come with a proof of soundness. This is essential, if the arguments to be had are going to make any sense. Ideally, the system should also have a proof of completeness. However, this property is highly desirable not completely-and-utterly essential, provided it is well understood why the system is incomplete.

5.3 Reasoning Forwards

Now that we have a *syntactic* proof system for propositional logic, which is sound and complete, we don't need to use truth tables at all (although the semantics justifies every application of the equivalence relations we used). There are however, three problems: firstly, the normal form may be exponentially larger than the original form, and certainly a lot harder to process for the human eye; secondly, constructing normal forms for quantified and modal formulas explicitly is just a little hard; and thirdly, our proof system, since it is based on equivalences, is using rules *about* the language, rather than inference rules themselves. We want something else, although normal forms turn out to be rather useful for that 'something else'. To see what that something else might be, let us try doing some proofs in the predicate calculus.

Let us start with a set of English statements, as follows (this will henceforth be referred to as the harry-ralph example):

Every horse is faster than every dog.
There is a greyhound that is faster than every rabbit.
Every greyhound is a dog.
Faster is transitive.
Harry is a horse.
Ralph is a rabbit.

Suppose we want to know if Harry is faster than Ralph. (This is also why this is known as a running example.)

The first thing to do is to formalise these English sentences in the language of the predicate calculus. Applying the 'translation' guidelines introduced previously, we come up with:

$$\forall x.\ horse(x) \rightarrow \forall y.\ dog(y) \rightarrow faster(x,y)$$
$$\exists x.\ greyhound(x) \wedge \forall y.\ rabbit(y) \rightarrow faster(x,y)$$
$$\forall x.\ greyhound(x) \rightarrow dog(x)$$
$$\forall x.\ \forall y.\ \forall z.\ faster(x,y) \wedge faster(y,z) \rightarrow faster(x,z)$$
$$horse(harry)$$
$$rabbit(ralph)$$

This is a consistent set of assertions. Whatever interpretations we have to consider must therefore make all these statements true. Clearly, even if it is not explicitly referenced, we are going to have to consider domains where there is at least a third object which is a greyhound, or either harry or ralph are greyhounds as well as being a horse and rabbit respectively.

However, just because there are interpretations which do not make (intuitive) sense does not invalidate the argument if it can be made in an interpretation which does make (intuitive) sense. So we will show that in *all* interpretations in which the premises (this set of assertions) hold, the conclusion (i.e. the formula entailed, the logical consequence, the thing to be proved – specifically, that Harry is faster than Ralph) also holds.

Now, we can't build all the possible interpretation and check each one, like we could (in principle) with truth tables. Instead, having a bunch of axioms and some inference rules, we derive a sequence of steps such that each step in sequence is instance of axiom schemata or follow from two earlier steps. Of course, we have to show that the calculus is sound and complete, but earlier logicians have done the hard work for us, we'll just leverage the calculus.

To do this, we will give ourselves a set of inference rules. This is a subset of the rules required for the full range of connectives, but it is enough for present purposes (it is certainly

sound, it just might not be complete). We will have an empty set of axioms. [Aside: I like it this way. If I had to express a preference, I regret to say that I *loathe* axiomatic systems, absolutely *abhor* the wretched things. I suppose this is due to the fact that I could never see which instance of which axiom schema I had to apply to push a proof forward, and the examples always seemed so contorted to me, even for very simple proofs, so much so that it seemed more like magic than logic; and logic should be like, well, logic, and definitely not at all like magic. Where, for example, is the logic in Harry Potter?]

Meanwhile, back on topic, here is the set of inference rules:

Modus Ponens	$\dfrac{P \rightarrow Q \quad P}{Q}$	If P implies Q, and P is true, then Q is true
And Elimination	$\dfrac{P \wedge Q}{P \quad Q}$	If 'P and Q' is true, then so is P, and so is Q
And Introduction	$\dfrac{P \quad Q}{P \wedge Q}$	If P is true, and Q is true, then so is 'P and Q'
Universal Instantiation	$\dfrac{\forall x.\ P}{P\{c/x\}}$	c is a constant (any constant from the domain) substituted for x in P
Existential Instantiation	$\dfrac{\exists x.\ P}{P\{f(v_1, ..., v_n)/x\}}$	f is a new function constant, and v_1, ..., v_n are all the free variables in P, and this function is substituted for x in P. If there are no free variables, then f is an object constant.

And off we go with the proof:

$\forall x.\ horse(x) \rightarrow \forall y.\ dog(y) \rightarrow faster(x,y)$
$\exists x.\ greyhound(x) \wedge \forall y.\ rabbit(y) \rightarrow faster(x,y)$
$\forall x.\ greyhound(x) \rightarrow dog(x)$
$\forall x.\ \forall y.\ \forall z.\ faster(x,y) \wedge faster(y,z) \rightarrow faster(x,z)$
$horse(harry)$
$rabbit(ralph)$

$greyhound(c) \wedge \forall y.\ rabbit(y) \rightarrow faster(c,y)$
$greyhound(c)$
$\forall y.\ rabbit(y) \rightarrow faster(c,y)$
$rabbit(ralph) \rightarrow faster(c,ralph)$
$faster(c,ralph)$
$horse(harry) \rightarrow \forall y.\ dog(y) \rightarrow faster(harry,y)$
$\forall y.\ dog(y) \rightarrow faster(harry,y)$
$dog(c) \rightarrow faster(harry,c)$
$greyhound(c) \rightarrow dog(c)$
$dog(c)$
$faster(harry,c)$
$faster(harry,c) \wedge faster(c,ralph)$
$\forall y.\ \forall z.\ faster(harry,y) \wedge faster(y,z) \rightarrow faster(harry,z)$
$\forall z.\ faster(harry,c) \wedge faster(c,z) \rightarrow faster(harry,z)$
$faster(harry,c) \wedge faster(c,ralph) \rightarrow faster(harry,ralph)$
$faster(harry, ralph)$

And we are done: we have made a (valid) argument.

This is more or less the basis of *forwards chaining* rule-based systems (RBS). An RBS (loosely) consists of an interpreter (which implements the reasoning mechanism, e.g. the 5 inference rules we gave ourselves above), a rule base (e.g. the 6 premises we gave ourselves above), and what is called working memory, where we add all the inferences we have made by applying the inference rules to the premises and inferences we have made. We keep going until one of the inferences we have made is the one we want, or there are no more inferences to be made. Some forwards chaining RBS often implement the rule base in terms of rules of the form IF <condition> THEN <action>, where the action might be to add or delete elements of working memory. The complication then comes from the requirement to have a conflict resolution strategy (since the conditions on many rules may be applicable, so which one should be fired), and the order in which rules are applied may be crucial, especially when deleting items from working memory. Further complications come from applying a rule and satisfying its action, only to have some later rule remove the condition: ideally we would also remove the effects of its action. Strategies to handle such situations are referred to as truth maintenance systems.

As an alternative, we could try backward chaining. In backward chaining, we take a goal to prove, and check to see if it is true. If not, we look for a rule that would make it true (i.e. what was in forward chaining the action), and check to see if its conditions are true, and so on. Instead of working memory, the system now has to store all the goals that we are trying to prove. We can also use backward chaining make the same move as we did with the normal forms, proof by refutation. Rather than checking to see if each inference derives the formula we are trying to prove, what we do is simply to add the negation of what we want to prove to a consistent set of assertions. If we then derive a contradiction, bivalence (remember that?) means that what we wanted to prove does indeed hold.

Proof by refutation and normal forms do indeed come together to do provide the mechanisation in computational logic. In the next sections, we shall specify a clausal form, an inference rule, and an algorithm for handling variables. Essentially, this is the basis of Prolog.

5.4 Clausal Form

It turns out (from bitter experience) that those proofs that can be most easily mechanised are those where the formulas are written in a uniform style and preferably only one inference rule. In the best known method the uniform style is called clausal form and the inference rule (see the next section) is called resolution.

Clausal form is a simplified form of predicate calculus. It has the same symbols and terms. However, instead of atomic formulas it has literals, which is an atomic formula or the negation of an atomic formula (referred to respectively as positive and negative literals). Instead of logical and quantified formulas it has clauses, where a clause is an implicitly quantified set of literals representing their disjunction.

Any formula of predicate calculus can be converted into a set of clauses that is equivalent to the original formula: by equivalent here we mean of course that the original formula is satisfiable if and only if the set of clauses is satisfiable (semantic equivalence. The conversion is completely algorithmic and the procedure follows these steps:

- Eliminate implication
- Reduce scope of negation
- Standardise variables
- Move quantifiers left
- Eliminate existential quantifiers
- Drop universal quantifiers

- Convert to conjunctive normal form
- Rename variables.

Example. We will illustrate this procedure by applying it to the premises of the harry-ralph example above. Let us start with: $\forall x.\ horse(x) \rightarrow \forall y.\ dog(y) \rightarrow faster(x,y)$

Eliminate implication. Use the familiar equivalences $p \rightarrow q \cong \neg p \vee q$, and $p \leftrightarrow q \cong (\neg p \wedge \neg q) \vee (p \wedge q)$.

$\forall x.\ horse(x) \rightarrow \forall y.\ dog(y) \rightarrow faster(x,y)$
$\cong \quad \forall x.\ horse(x) \rightarrow \forall y.\ \neg dog(y) \vee faster(x,y)$
$\cong \quad \forall x.\ \neg horse(x) \vee \forall y.\ \neg dog(y) \vee faster(x,y)$

Reduce scope of negation. Negations in this formula only apply to literals, but otherwise the usual equivalences would be applied ($\neg(p \wedge q) \cong \neg p \vee \neg q$, etc.).

Standardise variables. In this step, variables are renamed so that each quantifier has a uniquely identifiable variable. Here both quantifiers have different variable names so no more needs to be done.

Move quantifiers left, without changing order. The formula is now in prenex normal form.

$\cong \quad \forall x.\ \forall y.\ \neg horse(x) \vee \neg dog(y) \vee faster(x,y)$

Eliminate existential quantifiers. There are no existential quantifiers in this normal, so no more needs to be done. (However, this is a complicated step as we shall see when we convert the next formula.)

Drop universal quantifiers. Since any remaining variables (after the application of the previous step) are universally quantified, there is no need to explicitly refer to the quantifier.

$\cong \quad \neg horse(x) \vee \neg dog(y) \vee faster(x,y)$

We saw how to do this in Section 4.1). In this case, we are already done.

Rename variables. In the final step, we rename the variables so that no variable appears in more than one clause. This is the first clause, so:

$\cong \quad \neg horse(x1) \vee \neg dog(y1) \vee faster(x1,y1)$

And we are done. Now let us apply the procedure to the second clause in our set of premises.

Eliminate implication.

$\exists x.\ greyhound(x) \wedge \forall y.\ rabbit(y) \rightarrow faster(x,y)$
$\cong \quad \exists x.\ greyhound(x) \wedge \forall y.\ \neg rabbit(y) \vee faster(x,y)$

Reduce scope of negation. No more to do.

Standardise variables. No more to do.

Move quantifiers left, without changing order.

$\cong \quad \exists x.\ \forall y.\ greyhound(x) \wedge \neg rabbit(y) \vee faster(x,y)$

Eliminate existential quantifiers. Now comes the hard part.

Push the harry-ralph example onto the stack for a moment, and consider instead the following formula:

$$\forall x.\ \exists y.\ friend(x, y)$$

Let us take this a true statement. This means that any interpretation that satisfies this formula will have, for every x in the domain of course, at least one y in that domain who is his friend (or her friend). Now, instead of the existential quantifier, we can instead replace the y with a *function*, which for every x, picks out exactly one of those y's which that x was in the friend relation to. We call this function a skolem function, after the logician Thoralf Skolem who showed that if the original formula, with existential quantifiers, was satisfiable, then the derived formula, with skolem functions, was also satisfiable. So in this case, we see that:

$$\forall x.\ \exists y.\ friend(x, y) \qquad \cong \qquad \forall x.\ friend(x, sk(x))$$

Note that it would have been different had the order of the quantifiers been reversed, thus:

$$\exists y.\ \forall x.\ friend(x, y)$$

This formula asserts that there is just one object with whom everyone is a friend (including that object: s/he is a friend of him/herself). In this case, the skolem function has no arguments: and a function of no arguments is a constant. So skolemising this formula gives:

$$\forall x.\ friend(x, c)$$

where, in this interpretation the skolem constant c is now the symbol denoting the object that everyone was friends with in the original interpretation.

Returning finally to the harry-ralph example, we see that the existential quantifier being eliminated in this case is on the 'outside', so it can be replaced by a skolem constant:

$$\cong \qquad \forall y.\ greyhound(c) \wedge \neg rabbit(y) \vee faster(c,y)$$

Drop universal quantifiers. The only remaining variables (after the application of the previous step) is universally quantified, so we remove it explicitly:

$$\cong \qquad greyhound(c) \wedge \neg rabbit(y) \vee faster(c,y)$$

Convert to conjunctive normal form. No more to do.

Rename variables. In the final step, we rename the variables so that no variable appears in more than one clause. This is the second clause, so:

$$\cong \qquad greyhound(c) \wedge \neg rabbit(y2) \vee faster(c,y2)$$

We finish by converting the remaining formulas of the harry-ralph example to clausal form. Note that there are no more existential quantifiers, so no more skolemisation is required, and note that the two 'facts' ($horse(harry)$ and $rabbit(ralph)$) are already in clausal form.

Applying the conversion procedure to each formula in the harry-ralph example therefore yields:

$$\neg horse(x1) \vee \neg dog(y1) \vee faster(x1,y1)$$
$$greyhound(c)$$
$$\neg rabbit(y2) \vee faster(c,y2)$$

$\neg greyhound(x3) \vee dog(x3)$
$\neg faster(x4,y4) \vee \neg faster(y4,z4) \vee faster(x4,z4)$
$horse(harry)$
$rabbit(Ralph)$

So, we have a standard form of writing formulas. Now all we need is an appropriate inference rule.

5.5 Resolution

Resolution is a valid inference rule which given two clauses, produces a new clause implied by this pair, which, for propositional logic, looks like this:

$$\frac{\begin{array}{l}\neg P \vee Q \\ \neg Q \vee \neg R\end{array}}{\neg P \vee \neg R}$$

To show that this a valid inference rule, it is enough to inspect the truth table:

P	Q	R	$\neg P \vee Q$	$\neg Q \vee \neg R$	$\wedge$	$\neg P \vee \neg R$	
0	0	0	1	1	1	1	X
0	0	1	1	1	1	1	X
0	1	0	1	1	1	1	X
0	1	1	1	0	0	1	
1	0	0	0	1	0	1	
1	0	1	0	1	0	0	
1	1	0	1	1	1	1	X
1	1	1	1	0	0	0	

Intuitively, note that is we use our familiar equivalence for converting between or and implication, the inference rule simply confirms the transitivity of implies, i.e. $p \rightarrow q$ and $q \rightarrow \neg r$ implies $p \rightarrow \neg r$.

Alternatively, note that, If Q is true then:

$$(\neg P \vee Q) \wedge (\neg Q \vee \neg R) \;\cong\; (\neg P \vee true) \wedge (false \vee \neg R) \;\cong\; (true) \wedge (\neg R) \cong \neg R$$

Or, if Q is false then:

$$(\neg P \vee Q) \wedge (\neg Q \vee \neg R) \;\cong\; (\neg P \vee false) \wedge (true \vee \neg R) \;\cong (\neg P) \wedge (true) \;\cong\; \neg P$$

In other words, the truth of the statement $(\neg P \vee Q) \wedge (\neg Q \vee \neg R)$ is determined by the truth value of $\neg P$ or the truth value of $\neg R$, independent of the truth value of Q.

In it's most general form the resolution rule is:

$$\frac{\begin{array}{l}p_1 \vee p_2 \vee \ldots \vee p_{i-1} \vee p_i \vee p_{i+1} \vee \ldots \vee p_{m-1} \vee p_m \\ q_1 \vee q_2 \vee \ldots \vee q_{j-1} \vee q_j \vee q_{j+1} \vee \ldots \vee q_{n-1} \vee q_n\end{array}}{p_1 \vee p_2 \vee \ldots \vee p_{i-1} \vee p_{i+1} \vee \ldots \vee p_{m-1} \vee p_m \vee q_1 \vee q_2 \vee \ldots \vee q_{j-1} \vee q_{j+1} \vee \ldots \vee q_{n-1} \vee q_n}$$

where p_i and q_j are complementary literals (i.e. a positive and negative instances of the same propositional symbol).

In the form in which we will be most interested, the resolution rule looks like:

$\neg p1 \vee \neg p2 \vee \ ... \ \vee pm\text{-}1 \vee pm$ [rule]
$\neg pm \vee \neg q1 \vee \neg q2 \vee \ ... \ \vee \neg qn$ [goal]

$\neg p1 \vee \neg p2 \vee \ ... \ \vee pm\text{-}1 \vee \neg q1 \vee \neg q2 \vee \ ... \ \vee \neg qn$

In it's most specific form:-

p
$\neg p$

$\perp$

The $\perp$ symbol is an empty clause, a contradiction. We are trying to make p true and false at the same time, which of course we cannot do.

Contradiction is good, because this suggests an algorithm for deciding the validity of a formula. The algorithm is based on refutation (proof by contradiction) and the knowledge that we can convert any formula into a semantically equivalent normal form:

- Convert all premises into conjunctive normal form (CNF);
- Negate the desired conclusion, and convert that into CNF;
- Repeatedly apply the resolution rule, adding the new clauses generated, until either:
  - False is derived, i.e. a contradiction, or
  - There are no more instances of resolution to be applied.

In other words, we take a particular form of the premises (the formulas assumed to be true), and if we also assume the negation of what we want to show (the conclusion, in the same form), and then 'run' our algorithm for 'long enough', one of two things is going to happen. On the one hand, if we derive false, this means that the conclusion did, in fact, follow from the premises, from proof by contradiction. On the other hand, we stop because we've run out of clauses to apply resolution to. In which we infer that the formulas cannot be proved from the premises.

The nice thing about this is that, intuitively, every application of the inference rules takes two clauses and produces one clause; this means the number of clauses is always getting smaller. This means that we are bound to stop with an answer – in other words the algorithm is complete. Furthermore, from the demonstration of validity, if we start with two formulas which are satisfiable, we will produce a formula that is satisfiable. In other words, the algorithm is sound, and the answer (yes or no) is always correct.
That does it for propositional logic – but what about predicate logic.

5.6 Unification

What inferences can we make when we try to apply resolution to formulas of predicate logic converted to clausal form following the procedure given above?

Let us assume we are interested in a domain of discourse with two objects, *Tom* and *Jerry*. Let us further assume we 'know' the two following statements (are true): *Everything is either a cat or a mouse. Tom is not a mouse*. Neither of the two inferences *"Jerry is a cat"* and *"Jerry is a mouse"* is valid. *cat(jerry)* is not valid because a possible model could be {*cat(tom)*, *mouse(jerry)*}; *mouse(jerry)* is not valid because the model could be {*cat(tom)*,*cat(jerry)*}. In fact we can't say anything about Jerry at all; the correct inference is of course *cat(tom)* with the substitution *x* = *tom*.

In predicate logic, formulas are allowed to a structure, i.e. we have predicates which take arguments: variables, functions, or constants. To determine whether or not we can apply the

resolution inference rule in a step of our proof, we have to pay attention to how we handle the differing variables, functions, or constants that might appear in premises. As suggested by the example above, what we need to do is find a *substitution*, an assignment of values to variables, which will make two terms "the same". If one term is positive, and the other is negative, we can use them in resolution, provided we pass on the substitution to subsequent steps in the proof.

Unification is a procedure for searching for a consistent set of substitutions of elements. The algorithm to unify two terms *S* and *T* is relatively simply expressed:

- If *S* and *T* are constants, they unify if and only if they are identical;
- If *S* is a variable and *T* anything, then they unify with the substitution {*T*/*S*}; similarly if *T* is a variable and *S* anything, then they unify with the substitution {*S*/*T*}; if *S* and *T* are both variables then they unify and the substitution is {*S*=*T*}, in other words, if ever *S* gets a value, *T* gets the same value, and vice versa, in other other words, S and T are effectively the same variable;
- If *S* and *T* are both predicates, then they unify if and only if:
    - They have the same name;
    - They are of the same arity (same number of arguments);
    - Their pairwise corresponding arguments unify (recursively apply the procedure).

There are complications to this. For example, there may be several possible unifications, in which case procedure seeks the most general unifier. Moreover, there is a question over what to do with an attempt to unify a variable with a term containing itself: for example what happens if we try to unify *X* with *p*(*X*)? We get an infinite unification: *X* = *p*(*p*(*p*(…))). For this reason, some implementations of the algorithm apply what is called an *occurs check*, to ensure this can never happen.

Now, to prove $faster(harry,ralph)$ using resolution, we assume its negation, and show inconsistency, from which we conclude the original.

$\neg horse(x1) \vee \neg dog(y1) \vee faster(x1,y1)$
$greyhound(c)$
$\neg rabbit(y2) \vee faster(c,y2)$
$\neg greyhound(x3) \vee dog(x3)$
$\neg faster(x4,y4) \vee \neg faster(y4,z4) \vee faster(x4,z4)$
$horse(harry)$
$rabbit(ralph)$

1	$\neg faster(harry,ralph)$	
2	$\neg faster(harry,y4) \vee \neg faster(y4,ralph)$	$\{harry/x4, ralph/z4\}$
3	$\neg horse(harry) \vee \neg dog(y4) \vee \neg faster(y4,ralph)$	$\{harry/x1, y4=y1\}$
4	$\neg dog(y4) \vee \neg faster(y4,ralph)$	
5	$\neg greyhound(y4) \vee \neg faster(y4,ralph)$	$\{y4=x3\}$
6	$\neg faster(c,ralph)$	$\{c/y4\}$
7	$\neg rabbit(ralph)$	$\{ralph/y2\}$

5.7 Prolog

We define a subset of clausal form called Horn Clauses, which are formulas in clausal form containing at most one positive literal. We can see the clauses from the harry-ralph example are indeed horn clauses.

Now, given that the following are equivalent:

$$\neg p1 \vee \neg p2 \vee \neg p3 \vee q \quad \cong \quad \neg(p1 \wedge p2 \wedge p3) \vee q$$
$$\cong \quad (p1 \wedge p2 \wedge p3) \rightarrow q \quad \cong \quad q \leftarrow (p1 \wedge p2 \wedge p3)$$

and with a bit of judicious notation, swapping ← for ':-' and ',' for &, and introducing a '.' to separate clauses (to show where we have different variables), we can re-express these clauses as:

```
faster( X, Y ) :-
     horse( X ),
     dog( Y ).
greyhound( c ).
faster( c, Y ) :-
     rabbit( Y ).
dog( X ) :-
     greyhound( X ).
faster( X, Z ) :-
     faster( X, Y ),
     faster( Y, Z ).
horse( harry ).
rabbit( ralph).
```

And bingo, we have a Prolog program. Call this program `P`.

Now, as we said earlier, this is a declarative program. We don't run this program, we query it. Our query here, as before, is `P?faster(harry,ralph)`. This then becomes a matter of showing that `horse(harry)` and `dog(ralph)`, or, showing that `greyhound(harry)` and `rabbit(ralph)`, or, showing that `faster(harry,Y)` and `faster(Y,ralph)`… What do we do?

It may have been noticed that the pure resolution proof of the preceding section was really rather short; one of the reasons it was so short was that at each step we got *just the right* unification and resolution to work towards the conclusion we wanted. So we got there by 'insider dealing', as it were: we knew what we wanted and made the right choices. However, if we had to do it systematically, by trying out each of the possible resolutions, it would have been extremely laborious and easy to get lost. Computers, on the other hand, have no insight whatsoever, but do tend to be rather useful for the systematic side of things. So all we have to do is specify an algorithm for the order of computation:

1. the query is a conjunction of atoms or compound terms, referred to as the *goal* (if there is more than one conjunct, each conjunct is referred to as a *sub-goal*)
2. the subgoals are *satisfied*, i.e. return a 'yes' with any variable bindings, from *left to right*;
3. to satisfy a subgoal, look through the clauses in the program (in the order in which they are declared) for one with the same functor name and arity;
    a. if none is found, *fail* the subgoal, and start *backtracking*;
    b. if one is found, try to *unify* the subgoal with the head of the clause;
    c. if the unification fails, look for another clause with the same functor name & arity (i.e. the declarations are searched in *top to bottom* order);
    d. if no unification succeeds, *fail* the subgoal, and start *backtracking*;
4. if a unifiable relation is found (the unification *succeeds*), then:
    a. the values given to variable by the unification are transmitted to sub-goals in the body;
    b. the body replaces the original subgoal in the query conjunction;
5. if the query conjunction is empty, then the query *succeeds*, otherwise repeat from step 1;

6. if we are trying to *re-satisfy* a sub-goal because of backtracking, restart from step 3.c (as if the original unification had failed).

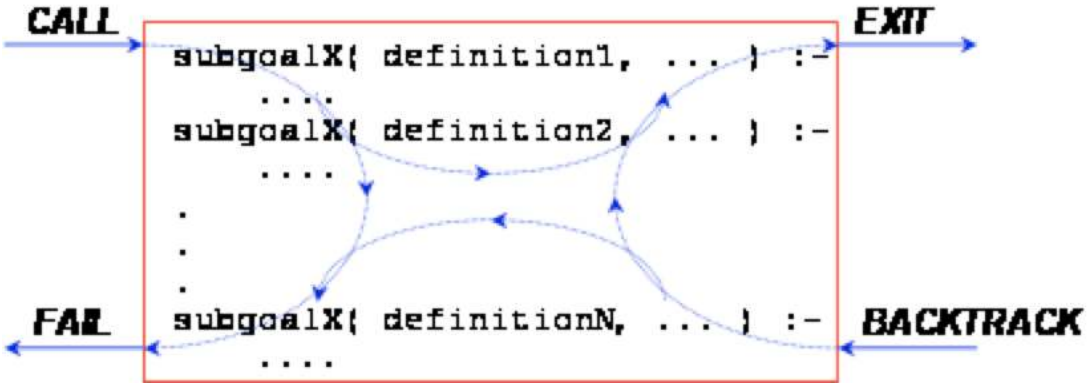


Figure 4 - Order of Computation in Prolog

In relation to the order of computation, we 'enter' the box (call) at step 3, we leave the box successfully (exit) at step 4, return to the box (backtrack) at step 6 (and pick up from wherever we were when last were in this box), and leave the box (fail) at steps 3.a and 3.d.

We can now see exactly what happens when we query our knowledge base with P?faster(harry,ralph), as shown in Figure 5 - A Prolog Computation.

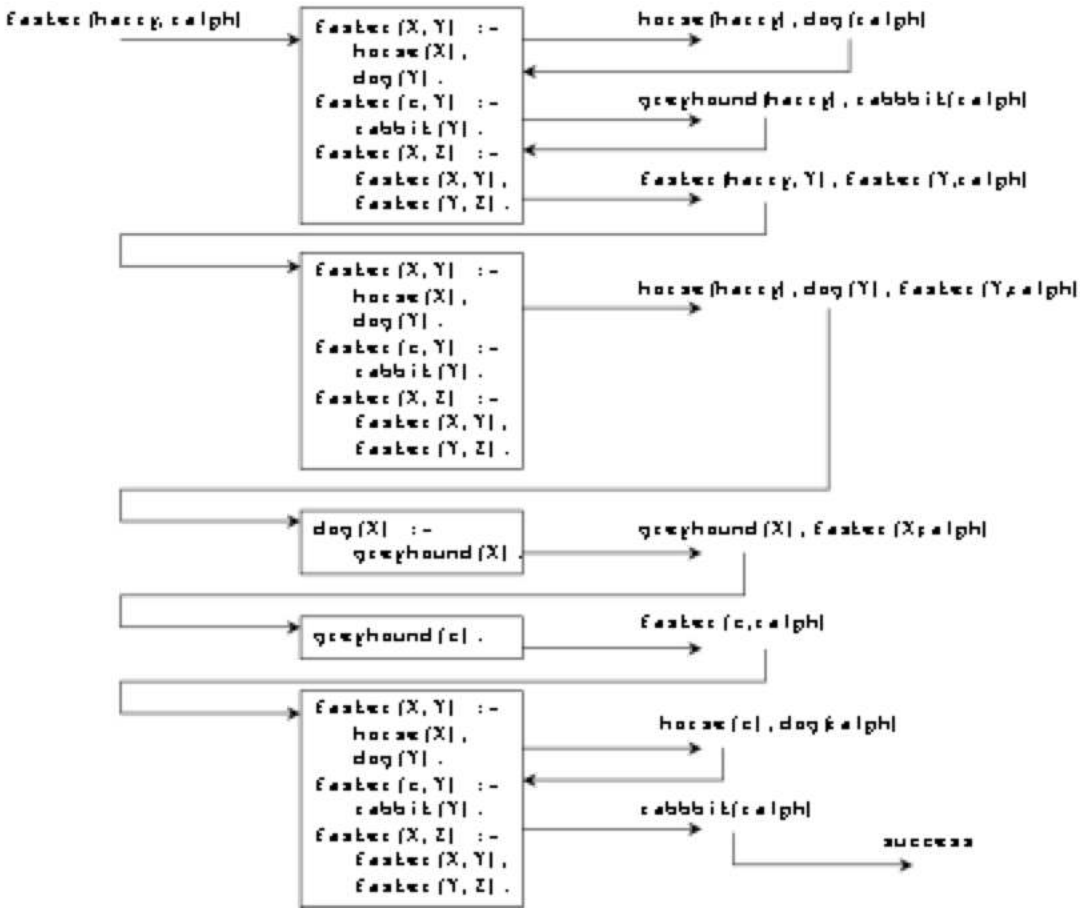


Figure 5 - A Prolog Computation

Note that we have omitted the box for some of the calls to simple facts (e.g. horse(harry)). It is instructive to follow the links through seeing how the unification algorithm fails or succeeds, and how variables acquire values which are passed on to subsequent goals. In particular note the unification of dog(Y) in the sub-goal (query) with dog(X) as the head of the clause. This is the same variable as in the goal faster(Y,ralph), which is why when X is bound to the constant c, this goal becomes faster(c,ralph). Thus variables are *bound* to values while the computation is going forwards (this is as close as Prolog gets to assignment), and they retain these values thereafter (i.e. they are not variable at all, in the sense that they cannot be re-assigned). On backtracking, the unification is 'undone' and the variable bindings are 'unbound', and now the terms can be unified in different ways to get new values for the variables.

This strategy of left-to-right, top-to-bottom search is known as *depth-first search with backtracking*. The irony is, that this procedure is well known for being vulnerable to loops and infinite paths. So it can be, that there are entailments, but Prolog cannot prove them because it gets stuck in an infinite loop. Hence, theoretically, it is not complete. Knowing this, and the source of the completeness, is enough to ensure that it need not be a problem in practice.

One final point to note is the difference between logical negation, which is an operation on formulas, and its procedural interpretation in logic programming. In Prolog, suppose we have a clause, say, 'not faster(ralph, harry)', to prove. The Prolog interpreter attempts to show that 'faster(ralph, harry)' is true, and when (if) that attempt is unsuccessful, then 'not faster(ralph, harry)' is considered to be true. This interpretation is called *negation-as-failure*, according to which the negation of a formula is true if and only if the formula cannot be proved to be true, or in other words, what is not known to be false is true. This is known as the *closed world assumption*, common in databases, whereby it is assumed that every record not explicitly contained in a table is implicitly assumed to be false, rather than unknown.

6 The Event Calculus

In this section, we define a general purpose language for representing and reasoning about events (actions) and their effects, which has a very convenient implementation in a logic programming framework. This language is called the Event Calculus [Kowalski and Sergot, 1986], and we will employ this language in our preliminary study of applying computational logic to regulatory compliance (see Section 7).

Notation. In the following we will use <- to denote 'if' or 'implies, & for logical 'and', and '+' for logical or.

6.1 The General Idea

The Event Calculus (henceforth EC) focuses on events, rather than situations, and local states rather than global ones. Rather than propositions, we reason about fluents, which are propositions whose truth value may change (from true to false, and vice versa) over time. One might therefore think of one local state for each fluent f which holds continuously over that period.

Events initiate and terminate periods of time for which fluents hold continuously. Events are assumed to be instantaneous, but it is not a particular complication to consider events with duration.

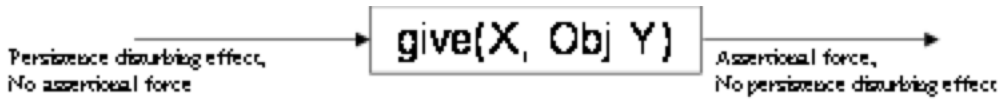
There are many versions of the Even Calculus: the one we will use here is referred to as the simplified EC, but we will continue to refer to EC.

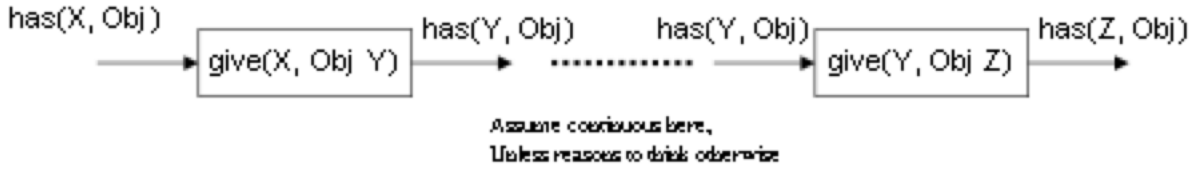
Thus we may have formulas of the following form:

give(X, Obj, Y) initiates owns(Y, Obj)
give(X, Obj, Y) terminates owns X, Obj)

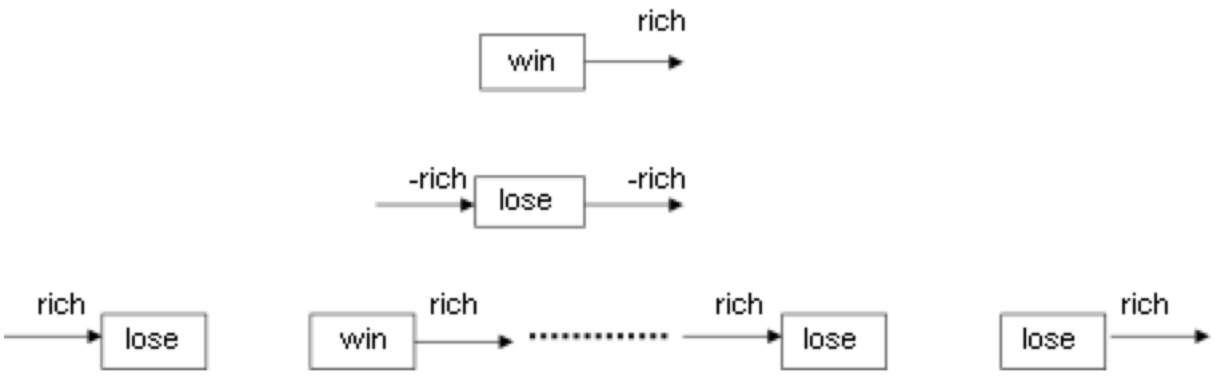
Clearly these are of the syntactic form 'event keyword fluent', where the keyword initiates makes the value of the fluent true, and terminate makes it false.

An event formula *E* terminates *F* does not say anything about the past. It simply blocks persistence of *F* beyond (in time) the event *E*. Thus the event *give(X, Obj, Y)* has, through terminates formulas, a persistence disturbing on fluents that were true before it, but no assertional force (we can not infer anything about values of fluents prior to the event), and similarly, through initiates formulas, it has assertional force on fluents that are true after the event, but no persistence disturbing effect 'back in time' on any fluent we know to be true after the event. We may represent this as shown below.

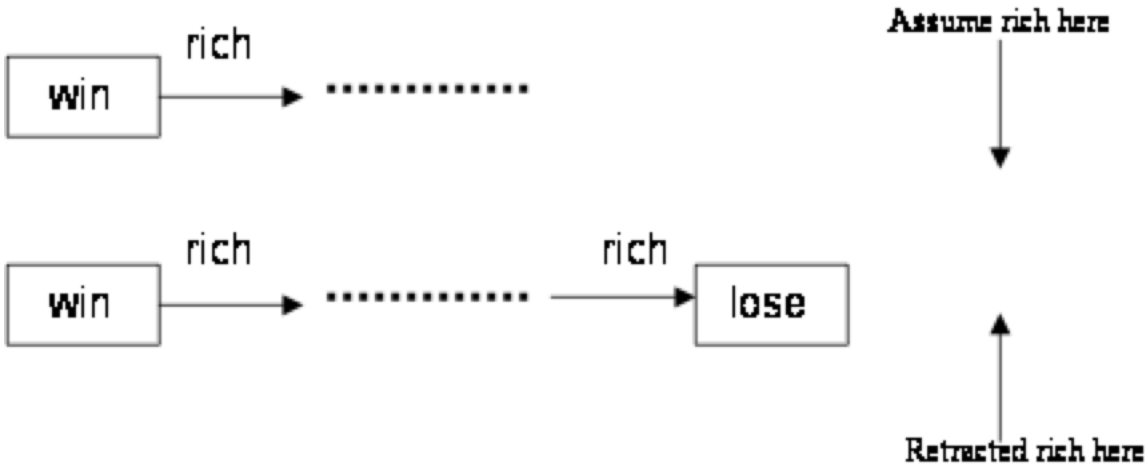




Example. Winning the lottery initiates rich (although we might already be rich). Similarly, losing your wallet might terminate rich (but we might not be rich when we lost it). In the spirit of the previous diagram, we might represent this as follows:



Finally, note that inference is non-monotonic. We might have a win event at time t after which rich holds. At time t+2, we assume rich holds still. But if we 'find out' about a new lose event at time t+1, later than time t but earlier than time t+2, we no longer assume (we retract) rich at the later time (and indeed all time points beyond t+1). This is useful because we can process events in any order we get them:



A collection of events is called a narrative.

6.2 EC General Formulation

In the (simplified) EC, we have the following:

- Events are given a time (when it happens);
- We assume, as above, that all events are instantaneous;
- We will use non-negative integer times – but this for (human) convenience. EC has no requirements that time is discrete and/or that time points are integers;
- This means that many events can happen at the same time;
- We can compute with the EC provided we have a relative ordering of events. The ordering itself could even be partial.

There are two useful computations that we can then perform:

1. Given a narrative, check that it is consistent, i.e., that the narrative is possible given actions pre-conditions (e.g. a fluent must hold for an action to be performed) and other constraints of our domain;
2. Given a (consistent) narrative, determine what holds when – now, in the past, and in the future.

The general formulation of the domain is then given by *holdsAt*, which allows us to compute what holds and when:

holdsAt(F, T)	*F* is a fluent, *T* is a time point, *F* is true (holds) at *T*

The narrative – what has happened and when – is represented by:

happens(E, T)	An action or event of type *E* occurred/happened at time *T*
initially(F)	*F* is a fluent, *F* holds at the first time point (usually 0)

The effects of actions are represented using the predicates initiates and terminates as introduced above:

E initiates F	The occurrence of event/action of type *E* initiates a period of time for which fluent *F* holds
E terminates F	The occurrence of event/action of type *E* initiates a period of time for which fluent *F* does not hold

Often, initiating and terminating events are context dependent (i.e. dependent on the local state). So there are 3-ary versions of both:

E initiates F at T if <conditions>
E terminates F at T if <conditions>

Of course, we also have:

If E initiates F then E initiates F at T

And

If E terminates F then E terminates F at T

The general rule for *holdsAt* is:

```
holdsAt(F, T ) <-
        happens(E, Ts) &
        Ts < T &
        initiates(E, F, Ts) &
        not broken(U, Ts, T)
holdsAt(F, T) <-
        0 <= T &
        initially(F) &
        not broken(U, 0, T)
```

In words, the first rule states that the fluent *F* holds at *T* if there occurred an event *E* at an earlier time *Ts*, the event initiated a period of time for which *F* holds at the time of occurrence (the action pre-conditions etc. were met), and there was not some other event which occurred between *Ts* and *T* which caused *F* to be terminated. The second rule deals with the initially statements in the narrative.

Of course, we also need a definition of broken:

```
broken(U, Ts, T) <-
        happens(Edash, Tdash) &
        Ts < Tdash &
        Tdash < T &
        terminates(Edash, U, Tdash)
```

Two last points. Firstly, note the way time comparisons have been written, especially in the first *holdsAt* rule. The strictly 'earlier than' means that that a fluent *does not* hold at the time of the event that initiated it.

Secondly, note that negation-as-failure for the 'not broken' ensures that the inference is non-monotonic. Not that this is a particularly efficient way of doing it, but converting the rule given directly into Prolog will work.

6.3 Multi-Valued Fluents

To date we have dealt with propositional fluents, i.e. fluents which have been either true or false. However, it is a simple EC extension to have multi-valued fluents, i.e. fluents which can have three or more values.

7 Summary

We have now completed a full coverage of a number of important concepts in computational logic, culminating in an understanding of the programming language Prolog and the application to an illustrative problem in regulatory compliance. We can now apply this knowledge and understanding of computational logic, to advanced topics such as automated reasoning, language processing and event recognition, but also in combination with planning and problem-solving techniques, as a platform for building intelligent software agents.

REFERENCES

[ASP02] A. Artikis, J. Pitt and M. Sergot. Animated Specifications of Computational Societies, in the *Proceedings of Autonomous Agents and Multi-Agent Systems* (AAMAS) Conference, pp. 1053-1062, Bologna, 2002.

[ASP03] A. Artikis, M. Sergot and J. Pitt . Specifying Electronic Societies with the Causal Calculator, in the *Proceedings of Agent-Oriented Software Engineering III* (AOSE) workshop, LNCS 2585, Springer, 2003.

[Hal03] J. Halpern. A Computer Scientist Looks at Game Theory. *Games and Economic Behaviour*, 45(1), pp114-131, 2003.

[HC96] G. Hughes and M. Cresswell. *A New Introduction to Modal Logic*. Routledge, London, 1996.

[KS86] R. Kowalski and M. Sergot. A Logic-based Calculus of Events. New Generation Computing 4(1), pp67-95, 1986.

[PBKN06] J. Pitt, J. Barria, L. Kamara and B. Neville. Imperial College & the ALIS Project: An Overiew of WP3. Presentation at *ALIS Project Meeting*, Paris, France, January, 2006.

[PKSA06] J. Pitt, L. Kamara, M. Sergot and A. Artikis. Voting in Multi-Agent Systems. *The Computer Journal*, 49:2, pp156—170, 2006.

[PKSA05a] J. Pitt, L. Kamara. M. Sergot, and A. Artikis. Voting in online deliberative assemblies. In A. Gardner and G. Sartor, eds., *Proc. ICAIL'05*, (to appear), 2005.

[PKSA05b] J. Pitt, L. Kamara. M. Sergot, and A. Artikis. Formalization of a voting protocol for virtual organizations. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, ACM Press, pp373—380, 2005.

[Rob00] H. Robert, et al. *Robert's Rules of Order Newly Revised, 10th Edition*, Perseus Publishing, 2000.

[vB06] J. van Bentham. Logic and Games, to appear, 2006.