CITS3005 Knowledge Representation

Lecture 3: Logic Programming

Tim French

The University of Western Australia

2023

Overview

In this lecture we will consider the basic elements of logic programming.

Prolog began as an implementation of a resolution engine that grew into a programming language in its own right.

Prolog has pragmatic implemen tation choices that assist efficient and predictable computations, while still allowing intuitive deduction and reasoning.

Running Prolog

Prolog has many implementations, that all loosely conform to a standard and typical run through a terminal.

```
$> swipl program_name.pl
```

although you can also use Gnu Prolog (`$>gprolog`), or others.

A prolog program is a knowledge base:

```
red(apple).
green(apple).
red(tomato).

fruit(apple).

redFruit(X) :- fruit(X), red(X).
```

It is either loaded when the program is run, or the rules can be loaded with the command: `['program_name.pl'].`

To quit prolog type

```
halt.
```

Queries

To interact, or run a program, you typically submit queries at the prolog prompt:
`? redFruit(X).`
All prologs commands are interpreted as queries, and end with a full stop.
When prolog interprets a query, it applies resolution and unification with the knowledge base.
If it is able to derive the query from the knowledge base it responds with `True` and gives the first satisfying substitution it found.:

```
True.
X=apple
```

You can type ; to see if there are any more substitutions that work

```
True.
X=apple ;
False.
```

When no more substitutions can be found Prolog responds with `False`.

Elements of a Prolog Program

A prolog program is simply a set of clauses. Furthermore, traditionally these are *Horn Clauses* which are clauses with only one positive literal. i.e. `redFruit(X) :- fruit(X), apple(X).` is valid, but `fruit(X), apple(X) :- redFruit(X).` is not.
In the clauses there are elements:

1. atoms `apple, tomato`
2. Strings `'Hello World'`
3. Integers `0,1,2,....`
4. Floats `3.14`
5. Functions/Structures `date(11,8,2022)`
6. Lists `[apple, 0, 'Hello']`

and predicates, like `red(apple)`.
A conjunction of of predicates is written as `red(apple), fruit(apple)`, a disjunction is written `red(apple); green(apple)`
Every clause and query ends with a full stop.

Atoms, Functions and Predicates

The core of Prolog's power is through unification and resolution, and these are applied to *atoms*, *variables*, *functions* and *predicates*.

- an *atom* is a name of a thing that cannot be further divided, and begins with a lowercase letter.
- a *function* takes a thing (atom, or function applied to a thing).
- a *predicate* take arguments (atoms, functions, or even other predicates) and produces a true false value.
- a *variable* is a placeholder that can be unified with things (atoms, functions applied to atoms), but also predicates.

```
eq(zero, zero).
eq(f(X),f(Y)) :- eq(X,Y).

or(P,Q) :- P.
or(P,Q) :- Q.
```

There are several useful builtin predicates, including `true` (a zero place predicate that always succeeds), and `false` (a zero place predicate that always fails, also written as `fail`).

Comparison Operators

As in most programming languages there are several ways to compare different terms and predicates, and these themselves are predicates:

- `X = Y` is true if `X` unifies with `Y`.
- `X \= Y` is true if `X` does not unify with `Y`.
- `X == Y` is true if `X` and `Y` are identical terms.
- `X \== Y` is true if `X` and `Y` are not identical terms

.

Built-in Data-Types: Numbers

Prolog has numeric data types of integers (...,-1,0,1,2,...) and floats (e.g. 3.14). Prolog uses representations optimised for hardware, but this means unification cannot be applied if arithmetic expressions are evaluated.

```
?- A+1=B+2.
no
?- A+1=2+B.
A = 2
B = 1
yes
```

Comparison	Meaning		Operator	Meaning
`X > Y`	X is greater than Y		`+`	Addition
`X < Y`	X is less than Y		`-`	Subtraction
`X >= Y`	X is greater than or equal to Y		`*`	Multiplication
`X =< Y`	X is less than or equal to Y		`/`	Division
`X =:= Y`	the X and Y values are equal		`**`	Power
`X =\= Y`	the X and Y values are not equal		`//`	Integer Division
`X is Y`	**assign** Y to X		`mod`	Modulus

Built-in Data-Types: Strings

Strings are are immutable, and delimited by single or double quotes in Prolog: `'CITS3005'`. Strings with single quotes are atoms (names of things), while string with double quotes are Strings.
note: the Availability of some of these functions varies between prolog implementations
Characters are stored as ASCII codes, and can be declared as `0't`. There are functions available to convert from lower case to upper case:
`lower_upper('t', X).`
To convert an atom string to a list of characters use:
`atom_chars('cites3005',X).`
To compare atom strings use the following operators:

Comparison	Meaning
`T1 @< T2`	succeeds if T1 is alphabetically before T2
`T1 @=< T2`	succeeds if T1 is alphabetically before or equal to T2
`T1 @> T2`	succeeds if T1 is alphabetically after T2
`T1 @>= T2`	succeeds if T1 is alphabetically after or equal to T2
`T1 =:= T2`	the T1 and T2 values are equal
`T1 =\= T2`	the T1 and T2 values are not equal

Lists

Lists in prolog are treated as having two parts: the head element of the list and the tail of the list. This is a two place predicate which can be used to describe arbitrary lists.
Lists can be described in one of three ways:

- a comma separated list enclosed in square brackets, e.g. `[10, -1, 8, 20]`
- as two parts, the head and tail, separated by a vertical bar, e.g. `[ HeadElem | TailList ]`
- via the built in predicate `.` and empty list constant `[]`: `.(10,.(-1,.(8,20))))`

The elements within lists can be any collection of valid data items, including lists of lists, etc. For example:
`[ 3, 'foo', [ x, [ y ] ], 17 ]`
Beware of the difference between `[a,b|c]` and `[a,b,c]`.
This representation lends itself to a natural recursive definition of operations:

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

List Operations

Some of the built-in list operations include:

Operation	Description
`list(List)`	True if `List` is a list
`length(List,Length)`	The length of `List`
`min_list(List, Elem)`	The minimum element in a list
`max_list(List, Elem)`	The maximum element in a list
`append(List1, List2, Result)`	The concatenation of two lists
`member(Elem,List)`	True is `Elem` is in `List`
`delete(List, Elem, Result)`	`Result` is `List` with all `Elem` remove
`last(List,Elem)`	The last element of `List`.
`reverse(List, Result)`	`Result` is `List` reversed.
`permutation(List, PermutedList)`	`PermutedList` is a permutation of `List`
`nth0(Position, List, Elem)`	`Elem` is in position `Position` in `List`
`sort(List, Result)`	`Result` is `List` sorted

How would you implement a predicate `indexOf(Elem, List, N)` where `N` is the first index of `Elem` in `List`, or -1 if `Elem` is not in `List`?

Execution of a Logic Program

A logic program executes by repeated application of resolution. Once the knowledge base is loaded (`?-[my_program.pl].`), and some query is entered (`?-check(this,Out), see(Out).`) then:

1. Prolog will take the left most element of the query conjuncts, call this q.
2. Prolog will go through the knowledge base row by row until it finds a clause with a head that can unify with q.
3. Prolog will apply unification to both the matched clause and the entire query conjunct and then applies resolution, to get a new set of literals which are inserted at the front of the query conjunct (i.e. a *depth first search*).
4. Then prolog takes the element at the front of the query conjunct and repeats the entire process.
5. If Prolog cannot find a clause that matches q in step 2, it will reverse the last unification and resolution step it made, and continue searching top down through the set of clauses for the next match.
6. If Prolog cannot find a clause that matches q and there are no steps left to reverse, prolog reports `no.`
7. If Prolog resolves all the query conjuncts, prolog reports `yes` and returns the successful substitution.

ExampleTrace

Suppose we build a simple system to perform arithmetic modulo 3.

```
limit((f(f(f(zero))))).

sum(X, zero, Y):- reduce(X,Y).
sum(X, f(Y), Z):- sum(X,Y,W), reduce(f(W),Z).

mult(X, zero, zero).
mult(X, f(Y), Z):- mult(X,Y,W), sum(X,W,W1), reduce(W1,Z).

equals(zero,zero).
equals(f(X),f(Y)):- reduce(X,A), reduce(Y,B), equals(A,B).

reduce(zero,zero).
reduce(X,zero):-limit(X).
reduce(f(X),f(Y)):-reduce(X,Y).
```

```
| ?- mult(f(zero),f(zero),X).
     1     1  Call: mult(f(zero),f(zero),_27) ?
     2     2  Call: mult(f(zero),zero,_98) ?
     2     2  Exit: mult(f(zero),zero,zero) ?
     3     2  Call: sum(f(zero),zero,_124) ?
     4     3  Call: reduce(f(zero),_148) ?
     5     4  Call: limit(f(zero)) ?
     5     4  Fail: limit(f(zero)) ?
     5     4  Call: reduce(zero,_135) ?
     5     4  Exit: reduce(zero,zero) ?
     4     3  Exit: reduce(f(zero),f(zero)) ?
     3     2  Exit: sum(f(zero),zero,f(zero)) ?
     6     2  Call: reduce(f(zero),_27) ?
     7     3  Call: limit(f(zero)) ?
     7     3  Fail: limit(f(zero)) ?
     7     3  Call: reduce(zero,_188) ?
     7     3  Exit: reduce(zero,zero) ?
     6     2  Exit: reduce(f(zero),f(zero)) ?
     1     1  Exit: mult(f(zero),f(zero),f(zero)) ?

X = f(zero) ?
```

Loops and Cuts

As Prolog uses backtracking and depth first search:

- it is easy to create infinite loops in the program.
- completing a full search of the proof tree can be inefficient.

A *cut* is a syntactic construct that can be inserted into a program that tells it to stop backtracking.

```
not(P) :- P, !, false.
not(P).
```

The `!` is a cut that causes the, program to stop any backtracking past the cut, to satisfy the head of the clause.

Special Operators

There a re a variety of non-logical operators in prolog, that “break” the representation of a logic program in first order logic. That is, they have side effects, some of which can alter the knowledge base. These include:

- `read(X)` read a string from the user and unify it with `X`.
- `write(X)` writes a string to the console.
- `asserta((P(X) :-Q(X)))` inserts a new rule to the *top* of the knowledge base.
- `assertz((P(X) :-Q(X)))` inserts a new rule to the *bottom* of the knowledge base.
- `consult(file)` adds a new set of clauses to the knowledge base, overwriting conflicting predicates.
- `halt` halts Prolog.
- `var(X)` true if `X` is an unbound variable at this point in the program.

Negation

Negation in logic programming is not as straightforward as it is in other languages. where there is no implicit quantification when evaluating logical statements (like Java or Python).

Understanding how *negation by failure* works is essential for both logic programming, and understanding the theory of knowledge bases.

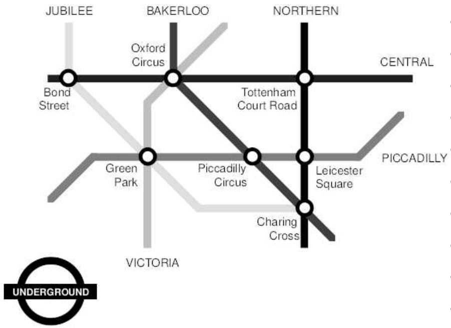
Prolog maintains a knowledge base, and when we pose a query, we are not asking "What is true in the world"; we are asking "What can be truly derived from the knowledge base".

If a fact, α cannot "be truly derived from the knowledge base", it does not necessarily follow the $\neg\alpha$ can be derived from the knowledge base.

Some of the notes are adapted from the book: *Simply Logical*, by Peter Flach
`https://book.simply-logical.space/src/simply-logical.html`.

Knowledge Bases

A knowledge base can be thought of as a collection of facts (much like a database), along with a general means of deriving new facts and constraining existing information.
For example, consider the following representations of a transport network:



```
connected(bond_street,oxford_circus,central).

connected(oxford_circus,tottenham_court_road,central).

connected(bond_street,green_park,jubilee).

connected(green_park,charing_cross,jubilee).

connected(green_park,piccadilly_circus,piccadilly).

connected(piccadilly_circus,leicester_square,piccadilly).

connected(green_park,oxford_circus,victoria).

connected(oxford_circus,piccadilly_circus,bakerloo).

connected(piccadilly_circus,charing_cross,bakerloo).

connected(tottenham_court_road,leicester_square,northern).

connected(leicester_square,charing_cross,northern).
```

Knowledge Bases

The two representations contain the same information, but if we want to express synthetic concepts, like stations *being nearby* (with two stops) or *on the same line*, what this concept *means* can be included in the knowledge base and used to populate these concepts:

```
nearby(X,Y):-connected(X,Y,_L).
nearby(X,Y):-connected(X,Z,L),connected(Z,Y,L).
```

We can use this to derive possibly derive that Bond Street and Charing Cross are not *nearby*, but probably not to derive that Subiaco and Leederville are not *nearby*

Unknown Unknowns

The is a significant asymmetry between what we know and can apply deduction to, and what is unknown.

> *Reports that say that something hasn't happened are always interesting to me, because as we know, there are known knowns; there are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns—the ones we don't know we don't know. And if one looks throughout the history of our country and other free countries, it is the latter category that tends to be the difficult ones.*

Donald Rumsfeld

The Closed World Assumption

The Closed World Assumption

The closed-world assumption (CWA), in knowledge representation is the presumption that a statement that is true is also known to be true, and conversely what is not currently known to be true, is false.

The opposite of the closed world assumption is the *Open World Assumption*, which presumes that only statements that can be said to be false, are those statements that are provably false.

In general, knowledge bases (both formal and informal) tend to record much more positive information than negative information, so there is a natural asymmetry in how deduction is applied

Aside: Belief Revision

What happens when our knowledge is wrong? Logically, if our knowledge base contains an incorrect fact, and this is not consistent with other facts in the knowledge base, all reasoning breaks down:

$$\mathcal{K} \models \bot \implies \forall \alpha \mathcal{K} \models \alpha$$

.

Belief Revision is the process of repairing a knowledge base when new information comes to light, or correcting a misconception.
Operators are defined to modify a knowledge base by:

- $\mathcal{K} \oplus \varphi$ - extend a knowledge base by a fact.
- $\mathcal{K} \ominus \varphi$ - retract a fact from a knowledge base.
- $\mathcal{K} \otimes \varphi$ - *revise* a knowledge base with new information.

It is non-trivial to decide which facts ought to be rescinded to allow new information to be added. Alchourron, Gardenfors and Makinson proposed the AGM postulates for *rational belief revision*.

Negation in Prolog

We can define a program to represent “negation” in Prolog, by using the *cut operation*

```
not(P) :- P, !, fail.
not(P).
```

Here we have a predicate `not` that takes any predicate, `P` as an argument, and if `P` can be satisfied, a cut is applied to prevent backtracking and the clause fails. Otherwise, `not(P)` will be true.

However, when we say `P` *can* be satisfied, we mean that there is *some* substitution for which `P` can be derived from our knowledge base.

Therefore `not(P)` is true when there is *no* substitution for which `P` can be derived from our knowledge base, so `P` and `not(P)` are quite different sorts of information.

Negation as Failure

This interpretation of negation is refereed to as *negation as failure*, and is a consequence of the closed world assumption. If something is not provable then its negation should be true.
This can lead to problems:

```
bachelor(X):-not(married(X)),man(X).
man(fred).
man(peter).
married(fred).
```

What does `?- bachelor(X).` return? Is there a way to correct this program?

As the closed world assumption rarely applies to knowledge bases, the preferred representation for negation as failure is

```
bachelor(X):- \+married(X), man(X).
```

which is read as *not provable that* `married(X)`, although its implementation is the same.