Introduction to Prolog

• Useful references:

Reviews

• A Prolog program consists of predicate definitions.
• A predicate denotes a property or relationship between objects.
• Definitions consist of clauses.
• A clause has a head and a body (Rule) or just a head (Fact).
• A head consists of a predicate name and arguments.
• A clause body consists of a conjunction of terms.
• Terms can be constants, variables, or compound terms.
• We can set our program goals by typing a command that unifies with a clause head.
• A goal unifies with clause heads in order (top down).
• Unification leads to the instantiation of variables to values.
• If any variables in the initial goal become instantiated this is reported back to the user.

Tests

• When we ask Prolog a question we are asking for the interpreter to prove that the statement is true.
  ?- $< 7, integer(bob).
  yes = the statement can be proven.
  no = the proof failed because either
    – the statement does not hold, or
    – the program is broken.
  Error = there is a problem with the question or program.
  "nothing" = the program is in an infinite loop.
• We can ask about:
  – Properties of the database: mother(jane,alan).
  – Built-in properties of individual objects: integer(bob).
  – Absolute relationships between objects:
    – Unification: =/2
    – Arithmetic relationships: <, >, =<, >=, =:=, +, -, *, /

Arithmetic Operators

• Operators for arithmetic and value comparisons are built-in to Prolog.
  – always accessible / don’t need to be written
• Comparisons: <, >, =<, >=, =:=, =
• Equality is different from unification
  – = compares if two terms unify
  – =/2 checks if two terms unify

Arithmetic Operators (2)

• Arithmetic Operators: +, -, *, /
  – Infix operators but can also be used as prefix.
  – Need to use is/2 to access result of the arithmetic expression otherwise it is treated as a term:

    | ?- X = 5+4. |?- X is 5+4.
    | X = 5+4 ? X = 9 ?
    yes yes

• Mathematical precedence is preserved: *<, before +,
• Can make compound sums using round brackets
  – Impose new precedence
  – Inner-most brackets first

Tests within clauses

• These operators can be used within the body of a clause:
  – To manipulate values,
    write('Numbers are the same').
  – To distinguish between clauses of a predicate definition

    bigger(N,M):-
    N < M,
    write('The bigger number is '), write(M).
    bigger(N,M):-
    N > M,
    write('The bigger number is '), write(N).
    bigger(N,M):-
    N =:= M,
    write('Numbers are the same').
bigger(5,4).  
\[ \text{The bigger number is 5.} \]

bigger(5,5).  
\[ \text{Numbers are the same.} \]

bigger(X,Y,Answer):- X>Y.

bigger(X,Y,Answer):- X=<Y.

\[ \text{Answer = X.} \]

\[ \text{Answer = Y.} \]

\[ \text{Satisfy subgoals - recursion.} \]

\[ \text{Most rules contain calls to other predicates in their body. These are known as subgoals.} \]

\[ \text{These subgoals can match: - facts, - other rules, or - the same rule - a recursive call.} \]

\[ \text{Success/failure - any computation can "succeed" or "fail", and this is used as a test mechanism.} \]

\[ \text{Matching - two data items can be compared for similarity, and values can be bound to variables in order to allow a match to succeed.} \]

\[ \text{Searching - the whole activity of the Prolog system is to search through various options to find a combination that succeeds.} \]

\[ \text{Backtracking - when the system fails during its search, it returns to previous choices to see if making a different choice would allow success.} \]

\[ \text{Why use recursion? - It allows us to define very clear and elegant code.} \]

\[ \text{Relationships may be recursive.} \]

\[ \text{Data is represented recursively and best processed recursively.} \]

\[ \text{Grammatical structures can contain themselves.} \]

\[ \text{Ordered data: each element requires the same processing.} \]

\[ \text{Allows Prolog to perform complex searches of a problem space without any dedicated algorithms.} \]
Prolog Data Objects (Terms)

- **Simple Objects**
  - Constants
    - Integers
    - Atoms
    - Symbols
    - Strings
    - Signs
  - Variables
  - Structures
  - Lists

- **Structured Objects**
  - To create a single data element from a collection of related terms we use a structure.
  - A structure is constructed from a functor (a constant symbol) and one or more components.

Examples:
- `man체: [a,b,9]`
- `a: [a,b,9]`
- `X: date(03,31,05)`

Structures

- **To create a single data element from a collection of related terms we use a structure.**
- **A structure is constructed from a functor (a constant symbol) and one or more components.**

Examples:
- `somerelationship(a,b,c,[1,2,3])`
- `person(bob,48)`

Lists

- **A collection of ordered data.**
- **Has zero or more elements enclosed by square brackets ([ ]) and separated by commas (,).**
  - `[a]` → a list with one element
  - `[]` → an empty list
  - `[a,b,c]` → a list with 3 elements where the 3rd element is a list of 2 elements.

Examples:
- `\[a\]` → a list with one element
- `\[]` → an empty list
- `\[a,b,c\]` → a list with 3 elements where the 3rd element is a list of 2 elements.

List Unification

- **Two lists unify if they are the same length and all their elements unify.**
- `\[[X,a]\] = [b,Y]` → a list with 3 elements where the 3rd element is a list of 2 elements.

Examples:
- `\[[a,b,c,d]\] = \[Head|Tail\]` → Head = a, X = a, Tail = \[b,c,d\]
- `\[[a,b,c,d]\] = \[X|[Y|Z]\]` → H = a, W = a, T = \[b,c,d\], X = b, Y = c, Z = \[d\]
- `\[[a]\] = \[H|T\]` → H = a, W = a, T = \[

Definition of a List

- **Lists are recursively defined structures.**
  - An empty list, [], is a list.
  - A structure of the form [X, …] is a list if X is a term and […] is a list, possibly empty.
- **This recursiveness is made explicit by the bar notation – \[Head|Tail\] (|' = bottom left PC keyboard character)**
  - Head must unify with a single term.
  - Tail unifies with a list of any length, including an empty list, []
- **The bar notation turns everything after the Head into a list and unifies it with Tail.**

Examples:
- `\[X,a\] = \[b,Y\]` → X = a, Y = b
- `\[a\] = \[H|T\]` → H = a, T = [], X = b, Y = c, Z = [d]
- `\[\] = \[Head|Tail\]` → \[X,a\] = \[X,b,c\]

Head and Tail

- **To ensure that the predicate terminates, the recursive case must move the problem closer to a solution.**
  - Without this the recursion would never stop!
  - This is called the base case:
  - is_a_list([],\[]).
  - is_a_list([X|T],\[X|T\]):- is_a_list(T).
- **With list processing this means stripping away the Head of a list and recursing on the Tail.**

Examples:
- `\[X,a\] = \[Head|Tail\]` → Head = a, W = a, T = [X|T], X = b, Y = c, Z = [d]
• This includes committing to the clause containing the cut.

The cut succeeds when it is
is_older(Ancestor,Young):-
parent(mary,tom).
parent(tom,jim).

Given this program:
a:- b, c, d, e,
a:- m .

no
?-(member(stuart,[john,paul,ringo,george]).
yes
?-(member(ringo,[john,paul,ringo,george]).
| ?- trace.

Focussed Recursion (2)

Tracing Member/2

Mutually Exclusive Clauses

Mutually Exclusive Clauses (2)
To ensure a logic friendly cut either:

- If you want to improve the efficiency of a program use green cuts to control backtracking.
- Red cuts make your code hard to read and are dependent on the specific ordering of clauses (which may change once you start writing to the database).

Greener cuts are in the head of the clause.

\[ p(X) :\neg testN(X) \]
\[ p(X) :\neg test2(X) \]
\[ p(X) :\neg test1(X) \]

\[ f(X,2) :\text{if } 6 \leq X \]
\[ f(X,1) :\text{if } 3 \leq X \text{ and } X < 6, ! \]
\[ f(X,0) :\text{if } X < 3, ! \]

\[ 	ext{We need to combine a cut with the fail to stop the redundant call to the second clause on backtracking.} \]

\[ 	ext{This is sufficient to represent the fact.} \]

Green Cuts!

\[ 	ext{Red Cuts!} \]

\[ 	ext{Red Cuts!} \]

\[ \text{Red Cuts!} \]

Using the cut

- Red cuts change the logical behaviour of a predicate.
- TRY NOT TO USE RED CUTS!
- Red cuts make your code hard to read and are dependent on the specific ordering of clauses (which may change once you start writing to the database).
- If you want to improve the efficiency of a program use green cuts to control backtracking.
- Do not use cuts in place of tests.

To ensure a logic friendly cut either:

\[ p(X) :\neg testN(X) \]
\[ p(X) :\neg test2(X) \]
\[ p(X) :\neg test1(X) \]

\[ f(X,2) :\text{if } 6 \leq X \]
\[ f(X,1) :\text{if } 3 \leq X \text{ and } X < 6, ! \]
\[ f(X,0) :\text{if } X < 3, ! \]

\[ 	ext{This is sufficient to represent the fact.} \]

\[ 	ext{Red Cuts} \]

\[ 	ext{Red Cuts} \]

\[ 	ext{Red Cuts} \]

\[ 	ext{Red Cuts} \]

A larger example.

We'll define several versions of the disjoint partial map split. (split of integers, non-negative, negative):

1. A version not using cut. Good code (each can be read on its own as a fact about the program). Not efficient because choice points are retained. The first solution is desired, but we must ignore backtracked solutions.

\[ \text{split}(X, Y) :\text{if } H > 0, \text{split}(X, Z, R) \]
\[ \text{split}(X, Y) :\text{if } H < 0, \text{split}(X, R, Z) \]

\[ \text{split}(X, Y) :\text{if } H = 0, \text{split}(T, Z, R) \]

A larger example.

2. A version using cut. Most efficient, but not the best 'defensively written' code. This cut clause does not delay its own execution as a fail. Negative solution is also redundant. Positive solutions are not needed because the cut is in the head of the clause.

\[ \text{split}(X, Y) :\text{if } H > 0, \text{split}(T, Z, R) \]
\[ \text{split}(X, Y) :\text{if } H < 0, \text{split}(X, R, Z) \]

A larger example.

3. A version using cut which is also 'safe'. The only inefficiency is that the goal H = 0 will be executed unnecessarily whenever H > 0.

\[ \text{split}(X, Y) :\text{if } H > 0, \text{split}(T, Z, R) \]
\[ \text{split}(X, Y) :\text{if } H < 0, \text{split}(X, R, Z) \]

Recommended for practical use. Hidden problem: the third clause does not capture the idea that H < 0 is a commitment. Here commitment is the default because H > 0 is in the head clause. Some new compilers detect that H = 0 is redundant.
A larger example.

3. A version with unnecessary cuts

split([], [], []) :- !.

split([H|T], [H|Z], R) :- H >= 0, !, split(T, Z, R).

split([H|T], R, [H|Z]) :- H < 0, !, split(T, R, Z).

First cut unnecessary because anything matching first clause will not match anything else anyway. Most Prolog compilers can delete this.

Why is the third cut unnecessary? Because H=0 is in the last clause. Whether or not H>0 fails, there was no choice left for the caller of split. However, the above will work for any order of clauses.

Negation as Failure

Using cut together with the built-in predicate fail, we may define a kind of negation. Examples:

Many likes any animals except reptiles:

likes(mary, X) :- reptile(X), !, fail.

likes(mary, X) :- animal(X).

A utility predicate meaning something like "not equals":

different(X, Y) :- !, fail.

different(_, _).

Negation as Failure

We can use the same idea of "cut-fail" to define the predicate not, which takes a term as an argument, not will "call" the term, that is evaluate it as though it is a goal.

not(G) :- !, G, fail.

not(_).

Call is a built-in predicate.

Negation as Failure

Not makes things worse

Using not will not help you. Do not try to remedy this by defining:

\[ \text{guilty}(X) = \text{not}(\text{innocent}(X)). \]

This is useless, and makes matters even worse:

\[ ?- \text{guilty}(\text{francis}). \]

It is one thing to show that St Francis is innocent, but it is quite another thing to incorrectly show that he is guilty.

Negation as Failure can be Misleading

Some disturbing behaviour even more subtle than the innocent/guilty problem, and can lead to some extremely obscure programming errors.

Here is a restaurant database:

\[ \text{good_standard}(\text{goedels}). \]
\[ \text{good_standard}(\text{hilberts}). \]
\[ \text{expensive}(\text{goedels}). \]
\[ \text{reasonable}(R) :- \text{not}(\text{expensive}(R)). \]

Consider the following dialogue:

\[ ?- \text{good_standard}(X), \text{reasonable}(X). \]
\[ X = \text{hilberts} \]

But if we ask the logically equivalent question:

\[ ?- \text{reasonable}(X), \text{good_standard}(X). \]
\[ \text{no}. \]

The semantics of reasonable(X) differ depending on whether its argument is instantiated.

Negation-by-failure can be non-logical

The difference between the questions is as follows.

In the first question, the variable X is always instantiated when reasonable(X) is executed.

In the second question, X is not instantiated when reasonable(X) is executed.

The semantics of reasonable(X) differ depending on whether its argument is instantiated.

Question

Why do we get different answers for what seems to be logically equivalent queries?

The semantics of reasonable(X) differ depending on whether its argument is instantiated.
Not a Good Idea!

It is bad practice to write programs that destroy the correspondence between the logical and procedural meaning of a program without any good reason for doing so.

Negation-by-failure does not correspond to logical negation, and so requires special care.

How to fix it?

One way is to specify that negation is undefined whenever an attempt is made to negate a non-ground formula. A formula is "ground" if it has no unbound variables.

Some Prolog systems issue a run-time exception if you try to negate a non-ground goal.

Clauses and Databases

In a relational database, relations are regarded as tables, in which each element of an n-ary relation is stored as a row of the table having n columns.

supplier place chair red 10
smith desk black 50

Using clauses, a table can be represented by a set of unit clauses. An n-ary relation is named by an n-ary predicate symbol.

supplier(jones, chair, red, 10).
supplier(smith, desk, black, 50).

Clauses and Databases

Advantages of using clauses:
1. Rules as well as facts can coexist in the description of a relation.
2. Recursive definitions are allowed.
3. Multiple answers to the same query are allowed.
4. There is no role distinction between input and output.
5. Inference takes place automatically.

Negation and Representation

Like databases, clauses cannot represent negative information. Only true instances are represented.

The battle of Waterloo occurred in 1815.

How can we show that the battle of Waterloo did not take place in 1819?

1. Complete the database by adding clauses to specify the battle didn’t occur in 1819, 1813, 1802, ..., 1796, 1797, 1798, ...
2. Using the same database, add a clause saying the battle occurred in another year (the battle occurred in and only in 1815).
3. Make the "closed world assumption", implemented by "negation by failure".

Summary

• Controlling backtracking: the cut !
  – Efficiency: avoids needless REDO-ing which cannot succeed.
  – Simpler programs: conditions for choosing clauses can be simpler.
  – Robust predicates: definitions behave properly when forced to REDO.
• Green cut = cut doesn’t change the predicate logic = good
• Red cut = without the cut the logic is different = bad
• Cut – fail: when it is easier to prove something is false than true.