Worksheet 16

Metaprogramming

The aims of this worksheet are

- that you learn what is meant by metaprogramming and see why it is useful;
- that you see how well-suited to metaprogramming Prolog is.

16.1 Introduction

Metaprograms treat other programs as data. Compilers and interpreters are obvious examples. But examples from previous worksheets include the simulation of finite state machines (for Sun headlines) and all the A.I. planners that we have built. Here we concentrate on *interpreters* of various kinds.

16.2 Prolog Interpreters

Metaprogramming (writing metaprograms) is easy in Prolog because Prolog terms can encode Prolog clauses (or other programs) quite easily.

Here we show how to easily write an interpreter for Prolog in Prolog. (Interpreters for a language written in the language itself are sometimes called meta-interpreters or meta-circular interpreters.) Writing these interpreters enables us to add new constructs to the language and/or to implement new evaluation strategies (e.g. breadth-first search rather than depth-first search).

16.2.1 A One-Clause Interpreter

We will define a predicate `solve(Goal)` which is true if `Goal` is true with respect to the program being interpreted. Here it is:

```
solve(Goal) :- call(Goal).
```

and that's it.

This uses the extralogical inbuilt primitive predicate `call`, described in worksheet 13, which executes the term to which its argument is instantiated as if that term were a normal predicate textually substituted in place of the use of `call`.

This interpreter is of little use since it delegates all its work to the normal Prolog interpreter. Having said that, it can form the basis of marginally more interesting versions. For example, in the file `haha`, I have supplied a highly witty interpreter that takes over the work normally done by the Prolog interpreter of supplying a prompt-read-evaluation-write loop. If this interpreter is loaded into Prolog along with, say, the definition of `max` from the previous worksheet, hours of fun can be had. The prompt is different and the way it displays answers is different, e.g.:

```
?- prompt_me.
Yes, oh great one?  max(1, 2, X).
max(1, 2, 2)
That's your lot
Yes, oh great one?  max(2, 1, X).
max(2, 1, 2)
That's your lot
Yes, oh great one?
```

Ha ha!

□ Try it. (If you look at the code, you will see that it uses the trivial interpreter given earlier. The rest of the code might not make much sense: it is full of Prolog funnies plus a particular Prolog programming trick.)

(In Sterling & Shapiro a simple interpreter like this is extended so that it logs all interaction to a file, which seems a fairly sensible exploitation of this idea.)

16.2.2 A Three-Clause Interpreter

A more interesting meta-interpreter is now given, which simulates the Prolog execution of a program. It is an interpreter for *pure* Prolog. But to implement it needs an impurity: it makes use of the inbuilt predicate `clause`.

In `clause(P, Q)`, `P` must be bound to a non-variable term, i.e. it must be at least partially instantiated, and the Prolog database is searched for a clause whose head matches `P`. The head and body of such clauses are unified with `P` and `Q` respectively. In the case of Prolog *facts* (unit clauses), `Q` will be unified with `true`.

For example, if the following are loaded into Prolog:

```
p(a).
p(X) :- q(X).
p(X) :- r(X), s(X).
```

we can use `clause` as follows:

```
?- clause(p(Z), Q).

Q = true
Z = a ;

Q = q(X)
Z = X ;

Q = r(X), s(X)
Z = X ;

no
```

(Variables will actually get renamed.)

Note that the first argument to clause cannot be simply a variable:

```
?- clause(P, Q).
Error
```

Now we present the meta-interpreter.

Clearly, the trivial goal `true` is always solvable:

```
solve(true).
```

Any other goal is solvable only if there is some clause in the database which matches it and whose body is solvable:

```
solve(Goal) :-
   clause(Goal, Body),
   solve(Body).
```

Finally, if the body consists of the conjunction of two subgoals, they must be solved independently:

```
solve((Body1, Body2)) :-
   solve(Body1),
   solve(Body2).
```

□ Try this interpreter out. It is in the file `solve`. You need to load it plus a normal Prolog program such as `append` and then issue queries as follows:

```
?- solve(append([a, b, c], [d, e], L)).

L = [a, b, c, d, e] ;
no
```

It should mimic normal Prolog.

Here are two traces, side by side, showing that this interpreter mimics or parallels what the normal Prolog interpreter would do on the same goal (in this case on a call to `append`):

```
?- append([a, b], [c, d], L).   ?- solve(append([a, b], [c, d], L)).

append([a,b],[c,d],_10) ?       solve(append([a,b],[c,d],_10)) ?
                                clause(append([a,b],[c,d],_10),L102) ?
                                clause(append([a,b],[c,d],[a|_35]),append([b],[c,d],_3
append([b],[c,d],_28) ?         solve(append([b],[c,d],_35)) ?
                                clause(append([b],[c,d],_35),L141) ?
                                clause(append([b],[c,d],[b|_61]),append([],[c,d],_61))
append([],[c,d],_35) ?          solve(append([],[c,d],_61)) ?
                                clause(append([],[c,d],_61),L180) ?
                                clause(append([],[c,d],[c,d]),true)
                                solve(true) ?
                                solve(true)
append([],[c,d],[c,d])          solve(append([],[c,d],[c,d]))
append([b],[c,d],[b,c,d])       solve(append([b],[c,d],[b,c,d]))
append([a,b],[c,d],[a,b,c,d])   solve(append([a,b],[c,d],[a,b,c,d]))

L = [a,b,c,d]                   L = [a,b,c,d]

yes                             yes
```

16.2.3 Absorption

The difference between the three-clause interpreter and the trivial one-clause interpreter resides in the amount of the work that is being done directly by the underlying Prolog interpreter:

- in the case of the one-clause interpreter, all of the work is *absorbed* by the underlying Prolog;
- in the case of the three-clause interpreter, much, but not all, of the work is being absorbed. For example, unification of goals with clause heads is

absorbed, as is the order in which clauses are tried. The choice of subgoals ('selection of literals') is not absorbed.

The portions of the interpretation process that are not being absorbed are open to being changed by us in our interpreter. Thus, the three-clause interpreter allows us to experiment with subgoal choice since this isn't absorbed. A simple example of this would be to investigate the consequences of choosing to execute them in reverse order by changing the third clause to:

```
solve((Body1, Body2)) :-
   solve(Body2),
   solve(Body1).
```

If the order in which *clauses* are tried were not absorbed, we would be able to experiment with different search strategies (e.g. breadth-first). If the *unification* were not absorbed, we would be able to experiment with an alternative implementation of unification.

Exercise 16.1 *Why might the latter be useful?*

Exercise 16.2 *The interpreter we've written is good only for pure Prolog. It cannot handle impurities, the primitive predicates. Add extra clauses to the interpreter so that it traps and correctly executes a number of primitives, e.g.* `clause`, `<`, `=<`, `>`, `>=`, `=:=`, `=\=`, *and* `Value is Expression`.

Exercise 16.3 *In this exercise we modify the interpreter so that it prints a sort of trace as it goes along, e.g.:*

```
?- tracer(append([a, b, c], [d, e], L)).

append([a,b,c],[d,e],[a|_37])
  append([b,c],[d,e],[b|_66])
    append([c],[d,e],[c|_98])
      append([],[d,e],[d,e])

L = [a,b,c,d,e] ;

no
```

The trace only shows successful calls, not all the fails as well like the normal tracer.

To start you off, here are some of the things you need:

```
tracer(Goal) :- tracer(Goal, 0).
```

```
display(Goal, Depth) :-
   write_n_spaces(Depth), write(Goal), nl.

write_n_spaces(0).
write_n_spaces(N) :- N > 0, write(' '), N1 is N - 1, write_n_spaces(N1).
```

(In fact `tab(N)` *is a primitive that does what* `write_n_spaces` *does.)* `read` *and* `write` *read terms from and write terms to the terminal;* `nl` *writes a newline.*

Your job is to take `solve` *and use it as the basis of a definition for* `tracer/2` *which displays indented goals as they are called. (This won't be interactive like the normal Prolog* `trace` *facility.)*

16.3 Program Transformation and Compilers

Compilers translate from source programs to object programs, where the object programs are programs that can be more readily and more efficiently executed. For example, the SICStus Prolog compiler produces machine code that, it is claimed, runs about 8 times faster and requires less runtime storage than interpreted Prolog. (The cost is that you cannot, e.g., trace the compiled code.) We will not pursue this further, but will instead look at another form of compilation.

One can imagine writing a compiler that translates one Prolog program to another Prolog program which is logically equivalent, but which the Prolog interpreter would typically execute using less time and space.[1] Such a compiler would use *program transformation* techniques, and, while we are not going to look at the construction of such a compiler, we will review some well-known program transformation techniques. Our presentation will be informal, giving enough information for you to be able to manually use such techniques yourself.

16.3.1 Goal order

An obvious example is swapping the order of subgoals in a rule to put the most restrictive ones earlier. We discussed this in worksheet 12 and noted the difficulties, e.g. that the best order can be query-specific and that changing the goal order can affect the available modalities.

[1] Another use of compilers would be to take in programs that are written in *extensions* of Prolog (e.g. object-oriented Prolog) and convert them into normal Prolog for direct execution by the Prolog interpreter. This will, generally, be more efficient than using a (meta-)interpreter written in Prolog to interpret the extended language.

16.3.2 Tail-recursive optimisation

Prolog offers us recursion, rather than iterative constructs (unless you count failure-driven loops). Iteration is more space efficient than recursion. In recursion, a stack frame has to be maintained for every recursive call that has not yet terminated. So, if there are n recursive calls, the program needs space linear in n. Iterative programs, however, typically use only a constant amount of memory, independent of the number of iterations (you don't cost yourself a stack frame each time you go round a `while`-loop, for example).

However, certain Prolog implementations contain an optimisation, called *tail-recursive optimisation*, that allows them to execute certain recursive Prolog programs (ones that satisfy certain conditions), the so-called *tail-recursive programs*, using constant space. Obviously, if you have such an interpreter, it make sense to write tail-recursive programs or to transform non-tail-recursive programs into tail-recursive ones, where possible.

The optimisation that these interpreters use (roughly) is, for a clause such as

```
P :- Q1, ..., Qn.
```

when just about to call `Qn` (which, in our case, will, in fact, be a recursive call to `P`), to pop any stack frames for `P, Q1,...,Qn-1` and then re-use `P`'s space for the call to `Qn`. Obviously, this can only be done under certain conditions, otherwise you lose paths of the search tree! The conditions are that, at the point when `Qn` is to be called, `P` has no alternative definitions left to try, and that `Q1,...,Qn-1` also have no choice points left, i.e. they were deterministic.

Assuming we have an interpreter that can spot (at least some) cases of tail recursion (using backtracking information and indexing information) and which will re-use stack space in the way indicated, we will want to write programs that can be subject to this optimisation.

Some already are, e.g.

```
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

By the time the recursive call is reached, the space can be re-used.

But, even when they are not, there is a program transformation we might be able to apply, namely to use *accumulators*. Compare the definition of `reverse/2` from exercise 15.3 with the one using an accumulator, exercise 15.4. The latter is tail recursive.

Or compare:

```
sumlist([], 0).
sumlist([N|Ns], S) :- sumlist(Ns, SNs), S is N + SNs.
```

with

```
sumlist(Ns, S) :- sumlistacc(Ns, 0, S).

sumlistacc([], S, S).
sumlistacc([N|Ns], Temp, S) :- Temp1 is Temp + N, sumlistacc(Ns, Temp1, S).
```

16.3.3 Composition

We can sometimes combine two programs that process the same data structure. For example, the program for computing the sum of a list's members and the program for computing the length of a list can be composed, as follows (using the definitions that do not use accumulators here):

```
sum_length([], 0, 0).
sum_length([N|Ns], S, L) :- sum_length(X, XsS, XsL),
   S is X + XsS, L is 1 + XsL.
```

Now, to compute the mean, for example, requires only one traversal of the list (using `sum_length`) rather than a traversal by `length` followed by a traversal by `sumlist`.

16.3.4 Incorporating testers into generators

'Generate-and-test' is a common problem-solving technique, much used as the basis of some logic programs: one routine generates a candidate solution to the problem, and another tests that candidate, trying to find whether it does actually solve the problem.

For example, to find whether two lists have a member in common, we might write:

```
common_member(Xs, Ys) :- member(X, Xs), member(X, Ys).
```

The first subgoal *generates* a candidate (a member of `Xs`), the second *tests* whether that candidate is a solution (is it also a member of `Ys`). (In Prolog, backtracking will ensure we keep generating and testing until the test succeeds.)

A technique for improving generate-and-test efficiency is to 'push' the tester into the generator as much as possible. You thus get a more 'intelligent' generator whose proposed candidate solutions are more often genuine solutions. (Ideally, you merge the two totally, so that only correct solutions are generated.)

Insertion sort, for example, can be viewed as the optimised version of permutation sort. Read Sterling Shapiro (2nd edn.) pp. 252-261 for other examples that are too lengthy to include here.

16.3.5 Partial execution

Partial execution in logic programming is also called *partial evaluation*, *partial reduction* or *partial deduction*.

In essence, the idea is that predictable steps in the execution of a program (in the case of logic programs, predictable resolutions) can be carried out (manually or automatically by a compiler) in a pre-processing (compilation) step, thus saving on their execution time. The most common form of this in logic programming is the use of *unfolding*.

Suppose we have the following program:

```
grandparent(A, C) :- parent(A, B), parent(B, C).

greatgrandparent(D, F) :- parent(D, E), grandparent(E, F).
```

The call to `grandparent/2` from within `greatgrandparent/2` can be unfolded by substituting the definition of `grandparent/2` into the body of the definition of greatgrandparent/2, to give:

```
grandparent(A, C) :- parent(A, B), parent(B, C).

greatgrandparent(D, C) :- parent(D, A), parent(A, B), parent(B, C).
```

Execution of queries on the revised `greatgrandparent/2` ought now to be (slightly) faster than on the original program because a resolution step that would have been carried out at execution time has now been carried out at compile-time. (An unfold operation is therefore a use of resolution, so a more theoretical definition would make mention of a most general unifier, etc.)

There are limits to it applicability. For example, we cannot always unfold recursive calls (how many times would you unfold them?). And the interaction with Prolog impurities is complicated. For example, the following program:

```
p(X) :- var(X), r(X).
r(3).
```

succeeds for queries such as `?- p(X)`. The naïve unfolding:

```
p(3) :- var(3).
```

always fails. The correct unfolding is:

```
p(X) :- var(X), X = 3.
```

16.3.6 Difference lists

Appending two lists in Prolog takes time proportional to the length of the first list. In languages that allow node-and-pointer representations, appending two lists need only take constant time. (Question: how?) There is a way of getting a similar constant time performance in logic programming, exploiting the idea of *incomplete data structures*, i.e. data structures with variables in them.

In Prolog programming, an alternative way to represent sequences is as the *difference* between a pair of *lists*. We can represent the sequence $\langle a, b, c \rangle$, for example, as the difference between the pair of lists `[a, b, c | L]` and `L`. This representation is known as *difference list* representation. It is an example of an 'incomplete data structure' because it uses variables in its representation. One way to think of the second item of the pair is as a pointer to the end of the list that is the first item of the pair.

We can define an append relation that uses Prolog unification to append difference lists in constant time. In this definition, each pair of arguments (i.e. (`Front`, `Back1`), (`Back1`, `Back2`) and (`Front`, `Back2`)) is a difference list. The definition is simply the following:

```
dl_append( Front, Back1,   Back1, Back2,   Front, Back2 ).
```

To append the sequences $\langle a, b, c \rangle$ and $\langle d, e \rangle$, `dl_append/6` is called as follows:

```
?- dl_append([a, b, c | X], X, [d, e | Y], Y, F, B).
```

As the dotted lines in the figure show by its use of repeated instances of the

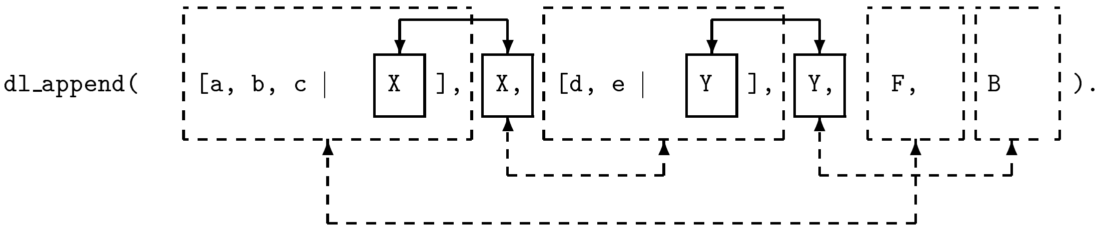


Figure 16.1: The operation of `dl_append`

same variable, the definition itself forces certain unifications to take place (the first argument with the fifth, the second with the third, and the fourth with the sixth). But equally, in the call to `dl_append/6`, some parts of the input structures are already constrained to be equal by virtue of shared variables (as shown by the solid lines in the figure). The effect of both sets of constraints is that `F` will be bound to `[a, b, c, d, e | Y]` and `B` will be bound to `Y`,[2] which

[2] In fact, Prolog will renames all variables whenever it retrieves clauses from the database, and so the answer will not contain `Y` but some renamed version of `Y`.

is indeed the difference list representation corresponding to the concatenation of the sequences $\langle a, b, c\rangle$ and $\langle d, e\rangle$. (To understand this, you will need to carefully work out the instantiations that this diagram describes for yourself.)

Any program that does a lot of appending of lists might usefully be transformed into one that uses difference lists.

A small example is `reverse/2`. The following:

```
reverse([], []).

reverse([X|Xs], Ys) :-
   reverse(Xs, Zs), append(Zs, [X], Ys).
```

would become

```
dl_reverse([], Ys, Ys).

dl_reverse([X|Xs], Ys, YsT) :-
  dl_reverse(Xs, Zs, ZsT), dl_append(Zs, ZsT, [X|XT], XT, Ys, YsT).
```

where only the second argument of `reverse/2` has bee turned into a pair of arguments (a difference list), and the call to `append/3` has become a call to `dl_append/6`.

In fact, we can improve this program further using the idea from the previous subsection, partial execution. The effect of `dl_append/6` is to carry out three unifications (the dotted lines in the diagram). We can do these in advance:

```
dl_reverse([], Ys, YsT).
dl_reverse([X|Xs], Ys, YsT) :-
   dl_reverse(Xs, Ys, [X|YsT]).
```

This has a nice way of rounding this material off: we see that difference lists have a constant-time append operation. We then see that a further advantage is that this append operation can often be partially executed. And, what we've ended up with is tail-recursive. (Indeed, other than the order of the second and third arguments, the version we've ended up with is the same as the version of `reverse` that uses an accumulator, so there is even more of a tying together of these ideas!)

□ Trace all three definitions to see the difference.

Of course, all this can be hidden from the user by adding a definition that converts the answer back to a normal list:

```
reverse(Xs, Ys) :- dl_reverse(Xs, Ys, []).
```

Sterling & Shapiro (2nd ed.) pp.291-293 show difference structures other than difference lists, having similar advantages to difference lists.

16.4 Expert System Shells

Expert system shells, as described below, also turn out to be interpreters, since they execute databases (or knowledge bases) of rules and facts.

16.4.1 What is an Expert System?

Expert systems are systems that are used to interpret, diagnose, predict, instruct, monitor, analyse, consult, plan or design. Each operates in some limited domain for which there are human experts. If any part of AI has ‘caught on’, it is this, although industry is not making full use of the ideas developed in universities. Some instances of their commercial use include:

- R1, or, to use its commercial name, XCON, has been used by DEC since 1984. Its task is to check customer orders for completeness, to complete incomplete orders and to decide on a spatial configuration of the items in an order. It has been used on over 500000 orders from 80000 customers. It misconfigures fewer than 1 in 1000 orders. The errors it makes are minor and are caused by knowing about only 5500 of the more than 100000 computer parts that might appear on an order.
- A US bank has developed the Personal Financial Planning System (PFPS). Using this, for a couple of hundred dollars per consultation, a family can get plentiful personal finance advice. The bank says that a similar quality consultation with a human advisor would cost $10000 each time.
- PROSPECTOR helps geologists to evaluate mineral sites for potential ore deposits. This system gained respect when in 1985 it found a molybdenum deposit worth $100 million.

16.4.2 Why build an expert system?

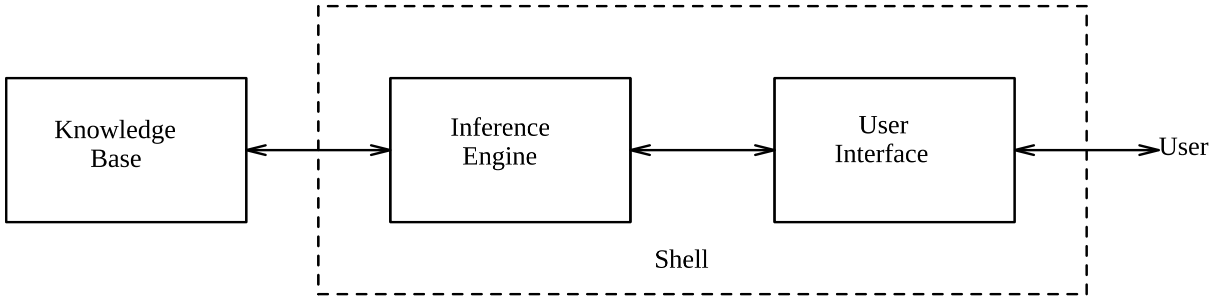
Possible advantages include:

- To disseminate rare and costly expertise.
- To formalise expert knowledge.
- To integrate diverse sources of knowledge.

Of course, expert systems have many disadvantages too, which should restrict their use to certain domains. They lack insight, compassion, an understanding of human motivations, the ability to guess, the ability to learn (usually), commonsense knowledge and the human experts’ sense of a “gut feeling”. All of this might rule out the deployment of an expert system from the point of view that it wouldn't work and/or might not be wholly ethical to apply it. In fact, in general, expert systems are best thought of as adjuncts to experts, not replacements of experts.[3] In this case, the user can provide the compassion and the commonsense.

16.4.3 The Architecture of an Expert System

It is convenient to divide expert systems into three main modules:



The Knowledge Base contains general knowledge about the domain (i.e. the area to which the expert system is being applied) and will also hold specific facts about the current case. The Inference Engine contains control knowledge that captures how to make active use of the Knowledge Base to carry out some reasoning/problem-solving task (e.g. diagnosis, tutoring, etc.). The User Interface caters for communication with the user.

The Shell is the domain-independent part of the system. In principle, it is re-usable: you can disconnect your bacterial infections knowledge base and connect an Ada program bugs knowledge base in its place, and the shell should be able to reason with this just as well. (Obviously, this is an exaggeration: the inference engine and user interface will only be *fairly* domain-independent. But they ought to be usable for a range of similar domains.)

We now look at the most common way that expert systems are implemented: as *rule-based systems*. It is important that you understand that those who *equate* expert systems with rule-based systems are out-of-touch. There are lots of other ways of implementing expert systems even if the use of rule bases is the most common.

16.4.4 Rule-Based Systems

Rule-based systems are often also referred to as *production systems*.

[3]Interesting in this regard is that the Japanese for "expert system" literally translates to the English "job assistant system".

In a rule-based system, the knowledge base consists of rules that capture general knowledge about the domain and facts that pertain to the particular case currently being investigated. While, in general, the syntax of the rules need not be different from that used for Prolog, the implementation of one of the shells is easier if they are. A small example might have the following rules, where the first argument is the consequent and the second argument is a list of the antecedents of a rule:

```
rule(mammal(X), [gives_milk(X)]).
rule(mammal(X), [has_hair(X)]).

rule(ungulate(X), [mammal(X), chews_cud(X)]).

rule(giraffe(X), [ungulate(X), has_long_neck(X)]).

rule(zebra(X), [ungulate(X), has_black_and_white_stripes(X)]).
```

Facts might be represented as follows:

```
true(chews_cud(gertie)).
untrue(chews_cud(bertie)).
```

At least two types of inference engine are possible for these sorts of expert systems:

Backwards-chaining: In backwards-chaining or goal-driven reasoning, we begin with the top-level goal and we must establish that this holds. Certain rules enable us to establish this. We adopt the conditions in the premiss of one such rule as subgoals, i.e. if these conditions are established, then the rule will fire and the top-level goal will be established. Each of these subgoals may themselves be established by some other rule(s), which sets up further subgoals (or sub-subgoals, if you prefer). And so we work down through the rules. Finally, we come to the basic literals representing facts. At which point, if the facts are already in memory, we are successful. If they are not, we ask the user about them. If the user confirms them, we are successful; else we must try some other path.

Forwards-chaining: In forwards-chaining or data-driven reasoning, we begin with the facts. The user supplies some facts at the start of the session. Once we have a few facts in the knowledge base, we will find that we have enough facts to match the premiss of some rule, which allows us to perform the action of that rule. Since this action may put further facts into the knowledge base, further rules may now be usable. This process continues until we reach the root.

Backwards-chaining is what Prolog does so the expert systems that you developed in worksheets 1 and 2 used a backwards-chaining inference engine on a Prolog representation of facts and rules. However, using Prolog directly does not give us an *interactive* expert system and so we need to implement our own interpreters or shells.

Before reading about the Prolog shells, you should quickly read the appendix to this worksheet, which lists some of the Prolog impurities we will be using.

16.4.5 A Backwards Chaining Rule-Based System Shell

Here, in Prolog, is the essence of a simple backwards chaining rule-based expert system shell (based on program 19.7 in Sterling & Shapiro 1st edition, p.312 —see program 17.19, p.345 in the 2nd edition, but this is an overly complicated revision):

```
solve([]).                    /* Trivially solved. */

solve([X|Xs]) :-              /* You can solve a list of goals by solving */
   solve(X),                  /* the head goal and the tail goals.        */
   solve(Xs).

solve(X) :-                   /* If there is a rule with head matching X  */
   rule(X, Xs), solve(Xs).    /* and body Xs, then solve Xs.              */

solve(X) :-                   /* Try asking the user.                     */
   askable(X),                /* See notes for explanation of askable.    */
   \+ known(X),               /* Make sure answer is not already known.   */
   ask(X, Answer),            /* Ask the user.                            */
   process_reply(Answer, X).  /* Process the reply.                       */


ask(X, Answer) :-             /* Display the question and get the answer. */
   display_query(X),
   read(Answer).

process_reply(yes, X) :-      /* If answer is yes, put the fact into the  */
   assert(true(X)).           /* database.                                */

process_reply(no, X) :-       /* If answer is no, assert that the fact    */
   assert(untrue(X)),         /* was untrue, and force an attempt to find */
   fail.                      /* another solution path.                   */

known(X) :- true(X).          /* You know whether X is true or not        */
known(X) :- untrue(X).        /* according to whether true(X) or untrue(X)*/
                              /* are in the database.                     */

display_query(X) :-           /* Ask the question rather uglily.          */
```

```
   write(X),
   write('? ').
```

`askable` is used to label things that we consider to be suitable to gather from the user. In the animals example, it seems reasonable to be able to ask whether the animal chews the cud, but not whether the animal is an ungulate (we wouldn't expect the user to know or be able to find out whether it is or not.) Therefore, we would have the following facts in our database:

```
askable(gives_milk(X)).
askable(has_hair(X)).
askable(chews_cud(X)).
askable(has_long_neck(X)).
askable(has_black_and_white_stripes(X)).
askable(mammal(X)).
```

It should be clear that the system will never ask the user whether the animal is an ungulate, preferring to chain through rules to try to prove whether it is or not. But it is quite ready to ask the user whether the animal is a mammal of has black and white stripes. If the user replies 'no', then the system will go on to look for a way of proving these facts using rules.

Here's an interaction:

```
?- solve(giraffe).
gives_milk(gertie)? yes.
chews_cud(gertie)? yes.
has_long_neck(gertie)? yes.

yes
```

□ In the file `kbase` you will find the rules and askable facts. In the file `shell1` you will find the code for the shell. Play with these. How can the interaction be improved? What is the effect of swapping the 3rd and 4th clauses of the definition of `solve`?

16.4.6 Adding Explanations

It is important to have explanation facilities in an expert system. Rule-based systems usually offer the user two very rudimentary (and possibly inadequate) types of explanation: WHY and HOW explanations. These explanations amount to little more than traces of reasoning.

Suppose an expert system is trying to establish whether a fact holds. It does this by asking the user a question. In most rule-based systems, the user can instead of answering the question, reply WHY, i.e. why are you asking me this question? To this the expert system replies by displaying the rules that could fire on the basis of the answer.

It is easy to add an extra argument to `solve` which carries around a list of rules that keeps track of ancestor rules that caused the current rule to be tried (based on Program 19.8 in Sterling & Shapiro 1st edition, p.315 —see program 17.23, pp352-353 in the 2nd edition, but this is overly complicated):

```
solve(Goal) :- solve(Goal, []).

solve([], Rules).

solve([X|Xs], Rules) :-
   solve(X, Rules),
   solve(Xs, Rules).

solve(X, Rules) :-
   rule(X, Xs),
   solve(Xs, [rule(X, Xs)|Rules]).

solve(X, Rules) :-
   askable(X),
   \+ known(X),
   ask(X, Answer),
   process_reply(Answer, X, Rules).

ask(X, Answer) :-
   display_query(X),
   read(Answer).

process_reply(yes, X, Rules) :-
   assert(true(X)).

process_reply(no, X, Rules) :-
   assert(untrue(X)),
   fail.

process_reply(why, X, [Rule|Rules]) :- /* If user asks ‘why’, then   */
   display_rule(Rule),                 /* display the parent rule,   */
   ask(X, Answer),                     /* repeat the question,       */
   process_reply(Answer, X, Rules).    /* and process the new reply. */

process_reply(why, X, []) :-           /* What to do if ‘why’ is     */
   write('No more explanation'), nl,   /* asked but there is no      */
   ask(X, Answer),                     /* parent.                    */
   process_reply(Answer, X, []).
```

```
display_rule(rule(X, Xs)) :-
   write('IF '), write_conjunction(Xs), write(' THEN '), write(X), nl.

write_conjunction([X1, X2 | Xs]) :-
   !, write(X1), write(' AND '), write_conjunction([X2 | Xs]).

write_conjunction([X]) :-
   write(X), nl.

write_conjunction([]) :-
   nl.

known(X) :- true(X).
known(X) :- untrue(X).

display_query(X) :-
   write(X),
   write('? ').
```

Here's a partial interaction:

```
?- solve(giraffe(bertie)).
gives_milk(bertie)? why.
IF gives_milk(bertie)
 THEN mammal(bertie)
gives_milk(bertie)? why.
IF mammal(bertie) AND chews_cud(bertie)
 THEN ungulate(bertie)
gives_milk(bertie)? why.
IF ungulate(bertie) AND has_long_neck(bertie)
 THEN giraffe(bertie)
gives_milk(bertie)? why.
No more explanation
gives_milk(bertie)? yes.

and so on.
```

□ You will find this shell in `shell2`. Try it out.

Suppose on the other hand, an expert system has finished its reasoning having established some goal node. On telling the user its conclusion, in many expert systems the user is allowed to ask HOW, i.e. how did you come to that conclusion? To this the expert system should show which rules did fire in reaching that conclusion.

For example, if the system concluded that `bertie` is an `ungulate`, you can ask how it concluded this:

```
?- how(ungulate(bertie)).
ungulate(bertie) is proved using the rule
IF mammal(bertie) AND chews_cud(bertie)
 THEN ungulate(bertie)

mammal(bertie) is proved using the rule
IF gives_milk(bertie)
 THEN mammal(bertie)

gives_milk(bertie) is a fact in the database

chews_cud(bertie) is a fact in the database
```

I don't give the code here (see Program 19.9 in Sterling & Shapiro 1st edition, p.316).

16.4.7 A Forwards Chaining Rule-Based System Shell

Now we look at a simple forwards chaining rule-based system shell.

Here is the code for the shell (simplified from Gazdar & Mellish, pp.335-337):

```
add(Fact) :- true(Fact), !.       /* If the fact being added is */
                                  /* already known to be true,  */
                                  /* then do nothing.           */

add(Fact) :-                      /* Otherwise,                 */
   assert(true(Fact)),            /* put the fact into the      */
   write('Adding: '),             /* database, tell the user    */
   write(Fact), nl,               /* that you have done so, and */
   find_new_consequences(Fact).   /* work out the consequences  */
                                  /* of this new fact.          */

find_new_consequences(Fact) :-    /* To compute consequences,   */
   rule(Head, Body),              /* for each rule in the base, */
   all_facts(Body),               /* if its body literals are   */
                                  /* all facts, then             */
   add(Head),                     /* add the head of this rule  */
                                  /* to the base.               */
   fail.                          /* Then backtrack to find     */
                                  /* other rules.               */

find_new_consequences(Fact).      /* This clause stops the      */
                                  /* failure-driven loop.       */

all_facts([Fact | Facts]) :-      /* All the facts in a list are*/
```

```
   true(Fact),                  /* true if the head is true   */
   all_facts(Facts).            /* and those in the tail are. */

all_facts([]).                  /* Simple case.               */

true(true).                     /* Extra fact that true is true*/

go :-                           /* Top-level routine that reads*/
   read(Fact),                  /* in facts and calls the      */
   add(Fact),                   /* inferencing system          */
   go.                          /* repeatedly.                 */
```

Here's an example dialogue:

```
?- go.
|: chews_cud(george).
Adding: chews_cud(george)
|: has_hair(george).
Adding: has_hair(george)
Adding: mammal(george)
Adding: ungulate(george)
|: has_long_neck(george).
Adding: has_long_neck(george)
Adding: giraffe(george)
|:
etc.
```

You can see that as you enter facts at the prompt, the system immediately forwards chains on them to derive new facts.

□ This shell is in `shell3`. Try it out.

I stress again that there are lots of other ways of implementing expert systems that use different knowledge representation and different inference engines.

Acknowledgements

Section B owes a lot to F.C.N.Pereira and S.M.Shieber, *Prolog and Natural Language Analysis*, CSLI 10, 1987; Section C owes a lot to Sterling & Shapiro (1st edition) with a tad of G.Gazdar and C.Mellish, *Natural Language Processing in Prolog*, Addison-Wesley, 1989, who provided an overly-complicated forward-chaining shell, which I simplified.

The following Appendix is based closely on the C-Prolog User's Manual.

Appendix

Here are the Prolog evaluable predicates introduced in this worksheet (though `retract` and `fail` were, in fact, first introduced in worksheet 12, and `call` was used in worksheet 13). There are lots of others (including those we have described in other worksheets, but also predicates for file handling). The descriptions I give are much abridged.

read(X): The next term, delimited by a full-stop, is read from the terminal and unified with `X`.

write(X): The term `X` is written to the terminal.

nl: A new line is written to the terminal.

tab(N): N spaces are written to the terminal. `N` may be an integer expression.

fail: always fails.

call(X): If `X` is instantiated to a term which would be acceptable as the body of a clause, the goal `call(X)` is executed exactly as if that term appeared textually in place of the `call(X)`. If `X` is not instantiated as described above, an error message is printed and `call` fails.

assert(C): The current instance of `C` is interpreted as a clause and is added to the program (with new private variables replacing any uninstantiated variables).

clause(P, Q): `P` must be bound to a non-variable term, and the program is searched for a clause whose head matches `P`. The head and body of those clauses are unified with `P` and `Q` respectively. If one of the clauses is a unit clause, `Q` will be unified with `true`.

retract(C): The first clause in the program that matches `C` is erased. `C` must be initially instantiated to a non-variable.