Worksheet 14

Evaluable Predicates and Functions: Arithmetic and Types

We can do arithmetic in pure Prolog (using Peano's axiomatisation), and we can check types in pure Prolog (see worksheets 4 and 5). But doing these in pure Prolog can be inefficient. Prolog offers in-built predicates and function symbols which, while offering no more power as such (indeed, they are less polymodal so there is a loss of flexibility), execute efficiently (not by resolution!). The aims of this worksheet are

- that you see that Prolog does provide extralogical arithmetic by means of evaluable function symbols;
- that you should see the consequences that this provision has on efficiency and declarativeness; and
- that you see that Prolog provides evaluable predicate symbols for type checking.

The worksheet ends with a section on Artificial Intelligence search, which brings together ideas from many of worksheets 1-14 inclusive.

14.1 Introduction

□ Try everything out as you read through.

Logic programs compute truths not values. Therefore, the following query returns *no*, not 4:

```
?- 2+2.
no
```

Prolog takes the `+` to be an (infix) predicate symbol and looks for a fact in the database that matches this query (or a rule that allows it to be concluded).[1]

But logical reasoning may need to be quantitative, and it may therefore require calculation.[2] The axiomatisation of arithmetic (using Peano's axioms) given in worksheet 4 is elegant but not practical. The implementation of `add` there was recursive. It takes time proportional to the magnitude of the first of the numbers being added. Modern computers provide efficient arithmetic operations directly in hardware, and so the time taken by, for example, addition operations can be pretty much independent of the size of the numbers. These facilities ought to be exploited by the language.

Logic programming languages, and Prolog in particular, *do* provide some operators that exploit the underlying capabilities of the machine. Specifically, the set of constants is extended so that as well as covering sequences of symbols beginning with lowercase letters, it includes numbers such as:

-999, 0, +999, 999, 0.23, 0.23E5, 0.23E+5, 0.23E-5

And a set of operations is also provided, such as addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), integer division (`//`), and possibly others (consult your manual). The operators are *evaluable*. That is to say that procedures are attached to them ('procedural attachment') and uses of these operators (at least when used in certain syntactic contexts) causes these procedures to be run.

A question about these 'operators' is: are they to be provided as binary *function symbols* or are as ternary *predicate symbols*? This is a point of debate. It varies between versions of Prolog (though the 'standard' Edinburgh syntax goes for the former). Here are arguments one might advance in favour of the two approaches.

14.1.1 Why arithmetic operations should be predicates

- `append` is like an addition operation over lists: it takes two small lists and gives a bigger one; given that addition takes two small numbers and gives a bigger one, it, like `append`, should be defined as a ternary predicate.
- We saw when we defined `add` as a predicate in Peano's axiomatisation of the natural numbers that we did not need to define subtraction separately. Addition and subtraction were *modal variants* of the same relation. Similarly, if the logic programming language offered us `+` as a ternary predicate

[1] So the answer would be *yes* if you had loaded the fact `+(2, 2)` or `2+2` into the database.

[2] Although, obviously, if your task is *primarily* numeric, then use a more appropriate language.

symbol, i.e. `+(X, Y, Z)`, we would expect to be able to use this (with different instantiations) to achieve `X + Y` ⇒ `Z`, `Z - X` ⇒ `Y` and `Z - Y` ⇒ `X`. Similar arguments apply to multiplication and division.

- Even if arithmetic is provided as evaluable predicate symbols, the programmer can still go on to use them to define function symbol arithmetic for him/herself in terms of the predicate arithmetic operators.

14.1.2 Why arithmetic operations should be functions

- The composition of arithmetic operators is of second nature to us, and only functions compose, e.g. we want to write `Answ = B*B - 4*A*C` rather than `prod(B, B, Temp1), prod(4, A, Temp2), prod(Temp2, C, Temp3), add(Answ, Temp3, Temp1).`
- There seems to be an obvious relationship of *evaluation* between an arithmetic *term* and its *value*, arithmetic terms being built out of function symbols.
- Even if arithmetic is provided as evaluable function symbols, the programmer can still go on to use them to define predicate symbol arithmetic for him/herself in terms of the function arithmetic operators.

Rightly or wrongly, 'standard' Prolog offers its arithmetic through binary function symbols. We look at this now.

14.2 Prolog Arithmetic

One thing that we hammered home in worksheet 4 is that function symbols are not evaluated in Prolog. For example, if you have the fact

```
prime(5).
```

i.e. `5` is a prime number, and you issue the goal

```
?- prime(2+3).
no
```

Prolog does not evaluate `2+3` to give `5` and reply *yes*. Rather it determines whether the terms `2+3` and `5` match, which they don't so the reply is *no*.

Now that we know that function symbols such as `+`, `-`, `*` and `/` have procedures attached to them (i.e. are evaluable), we need a way of telling Prolog when we *do* want it to evaluate them. There are two syntactic contexts that signal this to Prolog: the inbuilt predicate `is` and the comparison operators.

14.2.1 Value is Expression

Introduction

Prolog comes with an inbuilt infix predicate `is` which is used in goals (whether they be user queries or subgoals in the bodies of rules) of the following form:

```
Value is Expression
```

To prove this goal, `Expression` is evaluated as an arithmetic expression. The result of a successful evaluation is then unified with `Value`. If the unification succeeds, then the goal as a whole succeeds; otherwise the goal as a whole fails.

Here are some examples:

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

`3+5` is evaluated to `8` and then Prolog determines whether `8` unifies with `8`, which it does, so the answer is *yes*.

In the following `3+5` is evaluated to `8`. Then Prolog unifies `X` and `8`, which is successful with the binding `X` $\mapsto$ `8`:

```
?- X is 3+5.
X=8 ;
no
```

In the following:

```
?- 3+5 is 3+5.
no
```

Prolog evaluates the second `3+5` to `8`. Then it tries to unify the first `3+5` with `8`. This fails (just as trying to unify `f(a, b)` with `c` would fail).

A restriction on its use

As we discussed in the previous section, choosing to have arithmetic operations as *predicate* symbols would have allowed them to be used polymodally with a variety of instantiations for a variety of effects (e.g. using `add` to do subtraction). Unfortunately, and perhaps inevitably since Prolog doesn't use predicates but uses function symbols for arithmetic, the use of `is` does not have this flexibility. For example, the following raises an error condition:

```
?- 8 is X+3.
Error
```

This query does not return `X = 5`; first Prolog tries to evaluate `X+3` and cannot, so the whole goal is erroneous.

This doesn't mean that you can't have variables in these goals. You can. But the variables must have values bound to them by earlier parts of the program by the time Prolog tries to evaluate the expression. For example, if the system contains the following fact:

```
prime(5).
```

then the following goal succeeds:

```
?- prime(X), 8 is X+3.
X = 5 ;
no
```

Here `prime(X)` binds `5` to `X`. Then Prolog evaluates `X+3` given that `X` $\mapsto$ `5`, giving `8`. Finally, it tries to unify `8` with `8`, which obviously succeeds.

In the light of the above, the behaviour of Prolog with the following queries should be clear.

```
?- X is X+1.
Error
```

This gives an error because `X+1` is not an evaluable function. On the other hand,

```
?- prime(X), X is X+1.
no
```

gives *no* because after finding `X` $\mapsto$ `5`, the system evaluates `X+1` to 6 and then tries to unify `5` (the value of `X`) with `6` (the value of `X+1`), which fails.

In summary, there is a mode restriction on uses of `is`. Prolog can solve a goal `V is E` only if `E` is *fully instantiated*.[3] Of course, this means that programs that *use* `is` will also not be fully polymodal.

Example: Summing the numbers in a list

The task is to define `sumlist(L, N)`, which is true if the sum of the numbers in list `L` is equal to `N`.

This version is *wrong* (it has no successful modes of use):

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

[3] Once operators become functional, the only way that this restriction could be lifted is if Prolog were to contain a sophisticated equation solver, which it does not.

A query such as `?- sumlist([1, 2, 3], X)` would raise an error condition for this definition of `sumlist` because the variable `SNs` would not be bound when encountered in the first subgoal of the second clause. (Similarly for all other goals.)

To ensure that `SNs` is instantiated requires changing the order of the subgoals, as follows:

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

We see another example of a case where a left-recursive definition (similar in structure to `anc3`) has to be used. This is due (as with negation) to the behaviour of an inbuilt predicate.

This new version works on the following:

```
?- sumlist([1, 2, 3], X).
X = 6 ;
no
```

but cannot be used with the first argument other than fully instantiated, i.e. the first argument cannot be a variable or contain a variable:

```
?- sumlist(X, 6).
Error

?- sumlist([1, 2 | Tail], 6).
Error
```

(In fact, in these, on discovering the error, Prolog allows you to ask it to continue through backtracking, and it can go on to launch an infinite computation with an infinite number of errors.)[4]

14.2.2 Numeric comparison operators

The other syntactic context that tells Prolog to evaluate function symbols is the numeric comparison operators. Prolog provides the following predefined infix

[4] The following version, which you might want to look at, is supplied by Colin Runciman:

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

posno(1).
posno(N) :- posno(M), N is M+1.
```

It can be used with a variety of queries, e.g. including those where the list is a variable. However, as he says himself, asking for alternative solutions is asking for trouble!

predicates for numeric comparisons: `<`, `>`, `=<`, `>=`. Equality uses the symbol `=:=` and inequality uses `=\=`.[5]

In `A<B`, for example, `A` and `B` are evaluated as arithmetic expressions. The two resultant numbers are compared, and the goal as a whole succeeds if the result of evaluating `A` is less than the result of evaluating `B`. (If `A` or `B` is not an arithmetic expression, or *is not fully instantiated*, the goal raises an error condition.)

Here are examples:

```
?- 1 < 2.
yes

?- (3-2 < 2*3+1).
yes

?- 2<1.
no

?- clyde < 3.
Error

?- (X<1).
Error

?- prime(X), (X<1).
no
```

(assuming that `prime(5)` is a fact).

Example: Greatest common divisor

Given two positive numbers, `X` and `Y`, their GCD is computed as follows:

1. If `X` and `Y` are equal then their GCD is equal to `X` (or `Y`!);
2. If `X<Y` then their GCD is equal to the GCD of `X` and the difference `Y-X`;

[5] Be warned of this! The normal equality symbol, `=`, has a different use in Prolog. It is used to check whether two terms *unify*, not whether they are equal. Therefore,

```
?- 2+2 = 3+1.
no
```

returns *no* because `=` is not numeric equality, it is unification and `2+2` and `3+1` do not unify. The following returns *yes*:

```
?- 2+2 =:= 3+1.
yes
```

3. If `Y<X` then their GCD is equal to the GCD of `Y` and `X-Y`.

For example if `X` = 20 and `Y` = 25

1. GCD(20, 25)
2. = GCD(20, 5)
3. = GCD(5, 15)
4. = GCD(5, 10)
5. = GCD(5, 5)

Therefore, GCD(20, 25) = 5.

The Prolog follows the above definition closely:

```
gcd(X, X, X).
gcd(X, Y, D) :- X<Y, Y1 is Y-X, gcd(X, Y1, D).
gcd(X, Y, D) :- Y<X, X1 is X-Y, gcd(X1, Y, D).
```

Example: Computing primes

A fuller example is the computation of prime numbers up to some specified number:

```
primes(Limit, Ps) :- integers(2, Limit, Is), sift(Is, Ps).

integers(Low, High, [Low | Rest]) :-
   Low =< High,
   M is Low + 1,
   integers(M, High, Rest).

integers(Low, High, []) :-
   Low > High.

sift([], []).

sift([I | Is], [I | Ps]) :-
   remove(I, Is, New),
   sift(New, Ps).

remove(P, [], []).

remove(P, [I | Is], [I | Nis]) :-
   \+ (0 is I mod P),
```

```
   remove(P, Is, Nis).

remove(P, [I | Is], Nis) :-
   0 is I mod P,
   remove(P, Is, Nis).
```

Investigate this for yourself (it is in the file `primes`); we might also discuss how it works in lectures.

14.3 Exercises

Exercise 14.1 *Define* `listlength(L, N)` *so that* `N` *is the number of items in list* `L`. *(For the reasons given above, your use of* `Value is Expression` *will come last in your definition.)*

Exercise 14.2 *Use the idea of an accumulator to give another definition of* `listlength` *(call it llength). Here* `Value is Expression` *must come first! Make sure you understand why this differs between the two definitions.*

Exercise 14.3 *Define* `max(X, Y, Max)` *so that* `Max` *is the greater of the two numbers* `X` *and* `Y`.

Exercise 14.4 *Define* `maxlist(List, Max)` *so that* `Max` *is the greatest number in the list of numbers* `List`.

14.4 Types

Re-read the material on types in worksheets 4 and 5. There you see that we can check, using definitions such as `list/1` and `peano_no/1`, that the right kind of arguments have been given in goals. But doing so considerably worsens efficiency, particularly on recursive objects such as lists and Peano numbers, where the type checker must recurse through the structure to check it. The consequential neglect of type checking arguably didn't matter too much: if you used the wrong type of object in a goal, the goal would either fail (return `no`) or return silly results.

But now, it has become more important because of the use of evaluable function symbols (`+, -` etc.) and evaluable predicates (`is, >, <`, etc.). When these are faced by the wrong types of arguments, the program crashes, with an error message. We ought to add checks to avoid this.

What we want is the equivalent of an infinite table of facts:

```
number(0).
number(1).
number(2).
   .
   .
   .
```

Since we obviously cannot create an infinite table in this way, Prolog provides another evaluable predicate, `number/1`, which checks that its argument is currently instantiated to a number.

We can now use `number/1` to 'protect' some of our programs from mis-use:

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

Prolog provides other such predicates, e.g. `integer/1` and `atom/1` (which checks that its argument is currently instantiated to a constant other than a number, e.g. `clyde`).

Before leaving this, we ought to consider the modality of this evaluable predicate. Consider a query such as:

```
?- number(X), Y is X + 1.
```

What would you expect?

- Perhaps since `X` is not a number, this should fail.
- Perhaps `X` should be instantiated to a number. On backtracking, it could be instantiated to another number. This, after all, would be what would happen if `number/1` behaved like a table of facts as suggested above. So we would get answers such as:

  ```
  Y = 1
  X = 0 ;

  Y = 2
  X = 1 ;
  ```

  etc.
- Perhaps since `X` could be a number, this should succeed, but leaving `X` still uninstantiated. (Of course, the addition will then fail due to `X` being uninstantiated.)

What will happen is that Prolog will fail. (This has the advantage, unlike the third option above, of not passing a variable on to the subsequent arithmetic operations.)

14.5 State Space Search

A lot of Artificial Intelligence problems are based on searching a *state space*. Basically this means we envisage a problem as having a *starting state* (e.g. the initial state of a chess board), a *goal condition* for detecting solutions to the problem (e.g. the definition of checkmate) and a *set of successor operations* that can be applied to one state to generate new states (e.g. the legal moves of chess).

Thus the problem is conceptualised as a tree or graph of connected states which our algorithm must search to find a path from the starting state to a state that satisfies the goal condition. It should be clear that chess can be conceptualised as a search problem, as can noughts-and-crosses (tic-tac-toe). Even diagnosing a mechanical fault in a car can be represented as a search problem: each state is a state of partial knowledge about the problem; examining symptoms and inducing their cause(s) is a process of search through states of increasing knowledge.

Here we construct one general state space searching program. It searches a state space using depth-first search implemented as a backtracking algorithm. We define a predicate `solve(Start, ActionList)` which is true if `ActionList` is a list of actions from `Start` to a state satisfying the goal condition. `solve` calls `depthfirst`, initialising the solution state path to one that contains only the start state and the solution action path to the empty list:

```
solve(Start, ActionList) :-
   depthfirst([Start], [], RevActionList), reverse(RevActionList, ActionList).
```

The definition of `depthfirst` is recursive. In the base case we determine whether the path has reached a state satisfying the goal condition (bearing in mind that the most recent state is the *head* of the list), and if so the accumulated actions (2nd argument) are 'copied' into the final action list (third argument):

```
depthfirst([State | States], ActionList, ActionList) :-
   goal(State).
```

The recursive case finds a successor to the state we are in, extends the path with this provided it is not a state we have visited before on this path, and then recurses:

```
depthfirst([State | States], ActionList, FinalActions) :-
   successor(State, Action, NewState),
   \+ visitedBefore(NewState, [State | States]),
   depthfirst([NewState, State | States],
              [Action | ActionList], FinalActions).
```

The definition of `visitedBefore` is really just a kind of `member` definition:

```
visitedBefore(X, [Y|Ys]) :- sameState(X, Y).
visitedBefore(X, [Y|Ys]) :- visitedBefore(X, Ys).
```

All the above code can be found in the file `dpf`.

All that is needed to use this general search algorithm on a particular problem is a definition of `successor`, `goal` and `sameState` for that problem.

Here's an example, which you'll probably know. A Tory, a Rottweiler, a Liberal Democrat and a Nut Roast are on one side of a river and want to get to the other side. There is a boat, which the Tory knows how to handle (it's more of a yacht really). But it can hold only the Tory and one other. (It's quite a big Nut Roast.) We want to know the sequence of trips needed in order to transport everything across to the other side given that certain pairs cannot be left out of the Tory's supervision: specifically, if the Rottweiler and the Liberal Democrat are left alone, the Rottweiler will 'worry' the Liberal Democrat; if the Liberal Democrat and the Nut Roast are left alone then the Liberal Democrat will eat the Nut Roast (or is it the other way round?).

A representation of states of the problem is needed. I suggest we have two sets (represented as lists), one for the left bank and one for the right, which will show the position of the Rottweiler (represented as `rw`), the Liberal Democrat (`ld`) and the Nut Roast (`nr`). And we will have another variable whose values `l` or `r` will say where the Tory and his boat are. Here, for example, is the initial state, the goal state and a couple of legal intermediate states, respectively:

```
state([rw, ld, nr], l, [])
state([], r, [rw, ld, nr])
state([rw, nr], l, [ld])
state([rw], r, [ld, nr])
```

We can now define `goal` as follows:

```
goal(state([], r, Rs)) :-
   setEquals(Rs, [rw, ld, nr]).
```

And we can enumerate instances of the `successor` relation:

```
successor(state([rw, ld, nr], l, []), row(ld, right), state([rw, nr], r, [ld])).
successor(state([rw, nr], r, [ld]), row(ld, left), state([rw, ld, nr], l, [])).
successor(state([rw, nr], r, [ld]), row(empty, left), state([rw, nr], l, [ld])).
```

And so on!

Of course, better is to define `successor` using some compact rules, rather than as a very long enumeration of facts (i.e it give it an *intensional* definition, rather than an *extensional* one). The problem is essentially about four types of move: rowing from left-to-right or right-to-left, with a cargo or without a cargo.

Here's how (and this, by the way, explains why I used sets to represent the two banks):

```
successor(state([L|Ls], l, Rs), row(X, right), state(NewL, r, NewR)) :-
   setMember(X, [L|Ls]),
   setDelete(X, [L|Ls], NewL),
   setInsert(X, Rs, NewR),
   legal(NewL).

successor(state(Ls, r, [R|Rs]), row(X, left), state(NewL, l, NewR)) :-
   setMember(X, [R|Rs]),
   setDelete(X, [R|Rs], NewR),
   setInsert(X, Ls, NewL),
   legal(NewR).

successor(state(Ls, l, Rs), row(empty, right), state(Ls, r, Rs)) :-
   legal(Ls).

successor(state(Ls, r, Rs), row(empty, left), state(Ls, l, Rs)) :-
   legal(Rs).

legal(Xs) :-
   \+ (setMember(rw, Xs), setMember(ld, Xs)),
   \+ (setMember(ld, Xs), setMember(nr, Xs)).
```

Other formulations are, of course, possible. I hope this is a clear one (once you have given it your detailed consideration) if nothing else. Finally, here's the definition of `sameState`:

```
sameState(state(L1s, B, R1s), state(L2s, B, R2s)) :-
   setEquals(L1s, L2s), setEquals(R1s, R2s).
```

All the above code can be found in the file `river`.

Once both the `dpf`, `river` and (since `river` makes use of them) your set definitions from the previous worksheet are loaded, the problem can be solved:

```
?- solve(state([rw, ld, nr], l, []), A).

A = [row(ld, right),row(empty, left),row(rw, right),
       row(ld, left),row(nr, right),row(empty, left),
       row(ld, right)] ;

A = [row(ld, right),row(empty, left),row(nr, right),
       row(ld, left),row(rw, right),row(empty, left),
       row(ld, right)] ;
```

no

Try it our for yourself.

(By the way, the Labour voter would have taken them all across in one go using the subsidised ferry that I forgot to mention.)

Exercise 14.5 *Here's a problem in which you must decide the Prolog representation of states, the definition of the* `goal` *predicate, the definition of* `successor` *and the definitions of* `sameState`*. (Your definition of* `successor` *should be intensional, not extensional, and will make use of Prolog arithmetic.)*

You are given two jugs, a 4-gallon one and a 3-gallon one. Neither has any measuring markers on it. There is a tap from which the jugs can be filled with water. How can you get exactly 2 gallons of water into the 4-gallon jug?

We can show states as pairs of integers, (X, Y)*.* X *represents the amount of water in the 4-gallon jug;* Y *represents the amount of water in the 3-gallon jug. Hence the domain of* X *is a number between 0 and 4 inclusive and the domain of* Y *is a number between 0 and 3 inclusive. The start state is (0, 0). The goal state is* $(2, N)$*. (This is how I'm showing states below. That's not to say that you should necessarily represent them in this way in your program.)*

The successor operation is based on the following:

1. *If* $X < 4$ *then* $(4, Y)$*, i.e. fill the 4-gallon jug (from the tap)*
2. *If* $Y < 3$ *then* $(X, 3)$*, i.e. fill the 3-gallon jug (from the tap)*
3. *If* $X > 0$ *then* $(0, Y)$*, i.e. empty the 4-gallon jug (onto the ground)*
4. *If* $Y > 0$ *then* $(X, 0)$*, i.e. empty the 3-gallon jug (onto the ground)*
5. *If* $X + Y \geq 4 \wedge Y > 0$ *then* $(4, Y - (4 - X))$*, i.e. pour water from the 3-gallon jug into the 4-gallon jug until the 4-gallon jug is full*
6. *If* $X + Y \geq 3 \wedge X > 0$ *then* $(X - (3 - Y), 3)$*, i.e. pour water from the 4-gallon jug into the 3-gallon jug until the 3-gallon jug is full*
7. *If* $X + Y \leq 4 \wedge Y > 0$ *then* $(X + Y, 0)$*, i.e. pour all the water from the 3-gallon jug into the 4-gallon jug*
8. *If* $X + Y \leq 3 \wedge X > 0$ *then* $(0, X + Y)$*, i.e. pour all the water from the 4-gallon jug into the 3-gallon jug.*

Encode this problem for use by the depthfirst search routine and try it out. (There are quite a lot of solutions, although some are somewhat silly.)

Exercise 14.6 *Modify the definitions of* `solve` *and* `depthfirst` *from above so that they work in the fashion described below.*

Where, in the definition of `depthfirst`, *there is a check that the most recent state is not a state that has already been visited on that path, remove that check. Then add an extra argument to* `solve` *and to* `depthfirst` *whereby the user can supply a maximum path length. The system should not pursue a path any further if pursuing it would make its length exceed that maximum. (This gives an alternative way of avoiding going round in circles.)*

Check your new definition using `river` *and your answer to the jugs problem, neither of which should need changing. (Note that the answers from the two versions of this depthfirst search program will, of course, give different answers: the one that checks whether a state has been visited before allows no cycles in a path; the one with a depth bound allows a finite number of cycles if the bound is large enough.)*

Exercise 14.7 *Recode the Sun headlines solution so that it is expressed as a problem that can be used by the depth-bounded depthfirst search program that you produced in exercise 14.6.*

The nice things about this will then be that you can now have cycles in the graph that defines the headlines, and you can specify a maximum length for the headlines.

Exercise 14.8 *Make the same changes as you made in exercise 14.6 to the forwards and backwards planners that you developed in the last worksheet. Once you do this, you will find that you can now have robot action repertoires that can contain variables in the action specifications. For example, the following will work:*

```
action(pickup(X),       [ontable(X), clear(X), handempty],  /* Pre */
                        [holding(X)],                       /* Add */
                        [ontable(X), clear(X), handempty]). /* Del */
action(putdown(X),      [holding(X)],                       /* Pre */
                        [ontable(X), clear(X), handempty],  /* Add */
                        [holding(X)]).                      /* Del */
action(stack(X, Y),     [holding(X), clear(Y)],             /* Pre */
                        [handempty, on(X, Y), clear(X)],    /* Add */
                        [holding(X), clear(Y)]).            /* Del */
action(unstack(X, Y), [handempty, clear(X), on(X, Y)],      /* Pre */
                        [holding(X), clear(Y)],             /* Add */
                        [handempty, clear(X), on(X, Y)]).   /* Del */
```

These actions define a robot that can pick boxes off a table, put them down on the table, put one box on top of another, and take one box off another.

The following goal defines the starting state as one in which there is a stack of boxes (a red one at the bottom, a blue one on top of that, and a green one on *top of that) and defines the goal state as having the stack the other way up (the red on top, etc.): on goals such as:*

```
?- forwards([ontable(red), on(blue, red), on(green, blue),
            clear(green), handempty],
            [ontable(green), on(blue, green), on(red, blue),
            clear(red)],
            Actions, 6).

Actions = [unstack(green, blue),putdown(green),
            unstack(blue, red),stack(blue, green),
            pickup(red),stack(red, blue)] ;
no
```

Exercise 14.9 *Use your solution to exercise 14.6 again. Modify it to give a new search strategy, one that is based on depth-first search and is called* depth-first iterative deepening.

The idea is that you perform a depth-bounded depth-first search with a depth bound of 1. Then, you start all over again and do one with a depth bound of 2. Then one with a depth bound of 3. And so on. (Obviously, the program runs for ever (or until Prolog runs out of memory), with ever increasing depth bounds.)

Its advantage is that it finds all goals at depth d and less before proceeding to depth $d+1$ and so cannot miss out any goals. Also, it will find shorter paths first. Its disadvantage is that 'upper' parts of the search space are searched more than once, and goals in those 'upper' parts are found more than once.

So far, all the search algorithms have been depth-first algorithms, and all have been implemented using backtracking. Indeed, they have exploited the underlying Prolog backtracking algorithm. We left it to Prolog to detect choices, remember them, and pursue them one at a time.

Prolog is a general-purpose programming language, so it ought to be possible to write other search programs, e.g. ones that search breadth-first. To do this, we must keep track of options that have not yet pursued for ourselves. Then, the *order* in which these unpursued options are tried is under the control of the program. The program can pursue them in depth-first order, in breadth-first order, or in just about any specifiable order.

To keep track of the unpursued options, we will use a data structure, often called an *agenda*. At any point in the search, the program will determine the new choices and put them *all* onto the agenda. It will then choose one of the options from the agenda, remove it from the agenda and pursue it. Pursuing it may itself cause more things to go onto the agenda. Once it has exhaustively tried all the choices, the agenda will have become empty.

(Given that there are n choices, it might seem strange to put them *all* onto the agenda, to choose one and then remove it again. It might seem more sensible to put $n-1$ onto the agenda and keep hold of the one that the program is about to pursue. But, if we are to afford ourselves maximum flexibility, we want to allow for the possibility that the one that it should pursue next might not be one of the n newly-computed choices. It might be one of those presently on the agenda.)

The search strategy of the program is determined by the method it adopts for putting items on and taking items off the agenda. There are many choices but the most common two are as follows:

Depth-first: If the program treats the agenda as a *stack*, search will be depth-first: because new items are only put onto the front of the agenda and items for processing are only taken off the front of the agenda, it follows that the program is always processing one of the most recently created items. Thus it pursues a particular hypothesis to exhaustion before looking at older hypotheses on the agenda.

Breadth-first: If the program treats the agenda as a *queue*, search will be breadth-first: because new items are only ever put onto the back of the agenda but items for processing are still taken off the front of the agenda, it follows that the program extends all existing hypotheses by one processing step before looking at any of the new hypotheses.

If you read up on heuristic search in any Artificial Intelligence book, then you will realise that by treating the agenda as a priority-ordered queue (where the ordering is determined by the heuristic), we can implement these heuristic search algorithms in this way too.

In our implementation of agenda-based search, the agenda will be a Prolog list. Our main requirement is to process every item on the agenda until the agenda is empty, so the following recursive definition is needed (and we copy the Solutions accumulated in the second argument into the third argument when we reach the base case):

```
process_agenda([], Solutions, Solutions).

process_agenda([Item | Items], Solutions, FinalSolutions) :-
   process_agenda_item(Item, Items, NewAgenda, Solutions, NewSolutions),
   process_agenda(NewAgenda, NewSolutions, FinalSolutions).
```

`process_agenda_item` does something to `Item`, which probably creates some more items and these are then stuck onto or into `Items` (in a way that determines what kind of search we get, as described above) to get `NewAgenda`.

Exercise 14.10 *Your job is to implement a breadth-first search algorithm. Your main task is to define* `process_agenda_item/5` *as used above. Essentially, this* *will use an all-solutions predicate (from the previous worksheet) to generate new items and then stick these onto the back of the remaining agenda items to get the new agenda.*

A question you must think hard about (and experiment with different solutions perhaps) is what sort of 'items' will the agenda hold (e.g. states, paths of states,...).

A suitable example to test it on is the following:

```
goal(state(2, 2)).

successor(state(X, Y), up, state(X1, Y)) :-
   X1 is X + 1,
   legal(state(X1, Y)).

successor(state(X, Y), left, state(X, Y1)) :-
   Y1 is Y - 1,
   legal(state(X, Y1)).

successor(state(X, Y), right, state(X, Y1)) :-
   Y1 is Y + 1,
   legal(state(X, Y1)).

successor(state(X, Y), down, state(X1, Y)) :-
   X1 is X - 1,
   legal(state(X1, Y)).

legal(state(X, Y)) :-
   X >= 0, X =< 2, Y >= 0, Y =< 2,  \+ (X = 1, Y = 1).

sameState(state(X, Y), state(X, Y)).
```

You can think of this as defining an insect that can move around a 3 by 3 grid using up, down, left and right actions until it moves from the start state (specified by the user in his or her goal) to the goal state (2,2), but having to avoid state (1,1) on the way.

Anything else (e.g. other examples used in this worksheet) are almost certainly too big: breadth-first search eats space!

Acknowledgements

Some material is taken from S&S and from Bratko. The predicate symbol versus function symbol argument comes from discussions with Colin Runciman. The agenda search material is based on a section from D.G.Bridge and S.J.Harlow, *An Introduction to Computational Linguistics*, Blackwell (forthcoming).