Worksheet 13

Increasing 'Logical' Expressiveness: Negation and Higher-Order Logic

Any Turing-machine-computable function can be computed by a pure Prolog program. We do not, therefore, need any extra 'power'. However, some programs might be difficult to write or inelegant if we restrict ourselves to pure Prolog. In this worksheet, we look at the introduction into the language of negation and certain constructs of higher-order logic. The aims of the worksheet are

- to show you the form of negation that most Prologs provide;
- to show you that this negation is different from logical negation and is not a sound implementation of the theory of negation it claims to be based on;
- to show you that Prolog negation is nevertheless very useful; and
- to show you the way that most Prologs allow the use of some expressions of higher-order logic (although, again, these extensions are not strictly logical).

13.1 Introduction to Negation

As we know, Prolog programs consist of Horn clauses (clauses with at most one positive literal). The database/program, in particular, contains positive Horn clauses (having exactly one positive literal). Negative facts cannot be represented. Yet representing negative information seems like a very useful thing to be able to do. For example, the following rule:

```
sibling(X, Y) :- parent(P, X), parent(P, Y).
```

defines siblings as people with a parent in common and so defines a person to be his/her own sibling. To rule this situation out we would want to say that `X` and `Y` must be different. Adding this to the rule in a general way would seem to require negation.

Indeed negation is so useful that most Prolog's do provide some form of it. But given what we have said about Prolog programs being positive Horn clauses, Prolog negation cannot be logical negation. It must be *extralogical*, i.e. something defined *outside* of the logic.

The symbol used in our dialect of Prolog for its negation is `\+`. So `\+ P` can be read as 'not P'. The reason our dialect uses such a peculiar symbol rather than, e.g. `not`, `neg` or $\sim$, is to emphasise its non-logical, extralogical nature.

We will explain why Prolog negation is not logical negation shortly, but first we illustrate in a more theoretical way that the addition of some form of negation is very advantageous.

The form of a completely general clause is

$$\neg Q_1 \vee \ldots \vee \neg Q_n \vee P_m \vee \ldots \vee P_2 \vee P_1$$

which is equivalent to

$$\neg(Q_1 \wedge \ldots \wedge Q_n \wedge \neg P_m \wedge \ldots \wedge \neg P_2) \vee P_1$$

which is equivalent to

$$(Q_1 \wedge \ldots \wedge Q_n \wedge \neg P_m \wedge \ldots \wedge \neg P_2) \rightarrow P_1$$

And with some form of negation added into Prolog we get the following:

$$P_1 \texttt{:-} Q_1, \ldots, Q_n, \texttt{\textbackslash+} P_m, \ldots, \texttt{\textbackslash+} P_2.$$

and this is equivalent to the above *except for the fact that* `\+` *isn't logical negation.*

13.2 Negation as Failure

□ Try out all the examples presented here as you read through it.

The `\+` operator uses the *negation as failure* rule. A goal `\+` P will be assumed to be true if P cannot be proved.

For example, suppose we have the following (simple) program:

```
elephant(clyde).
```

Here are some queries:

```
?- \+ elephant(flopsy).
yes
```

The goal `?- \+ elephant(flopsy)` follows from the program by negation as failure: `elephant(flopsy)` cannot be proved so its 'negation' is assumed to be true. On the other hand, the goal

```
?- \+ elephant(clyde).
no
```

does not succeed because `elephant(clyde)` can be proved.

As another example, we can put a negation into the body of a rule. Here we (approximately) capture the clause that says that Clyde is happy if he does not belong to a circus, and we add this to our program:

```
happy(clyde) :- \+ belong_circus(clyde).
```

In this query:

```
?- happy(clyde).
yes
```

to prove that Clyde is happy, it has to prove that he doesn't belong to a circus; first it tries to prove that he does belong to a circus, which it cannot; therefore, it concludes that he doesn't belong to one and returns the answer *yes*.

If we augmented the program with the following:

```
belongs_circus(X) :- elephant(X).
```

The query

```
?- happy(clyde).
no
```

returns *no* because now Prolog can prove that Clyde belongs to a circus so the negative goal fails.

Note that you cannot put the negation operator into the head of a rule and hence, equally, you cannot put a negative fact into the database. The reason is that negation as failure requires a failure to prove. You cannot *store* that something isn't provable (this is a fact we can only write in the meta-language); you can only try to prove it and fail to do so: negation as failure is procedural.

The goal `\+ P` requires Prolog to try to prove P: If P is provable, Prolog concludes that the negated goal fails (and so it returns *no*); if P is not provable, Prolog concludes that the negated goal succeeds (and so it returns *yes*). A problem is that trying to prove P may not terminate. There are two reasons why trying to prove P might not terminate.

The first reason is that determining whether a goal is provable is in general an undecidable problem[1]. So if a goal P is unprovable, a logic program might terminate with failure or it may loop forever (or until it runs out of resources). Therefore, negation as failure is really negation as *finite* failure (NFF), i.e. `\+ P` succeeds if P cannot be finitely proved (i.e. if P finitely fails).

But, and here is why Prolog is incomplete, Prolog uses depth-first SLD-resolution and *this* gives rise to infinite computations (on left-recursive rules, for example). Yet in these cases, some other theorem-prover, with a different search strategy would terminate with failure.

So, if the Prolog search tree has infinite branches, Prolog *might* have a problem getting the same answer as the theory of negation as finite failure would predict. More specifically, it gets no answer at all if it 'hits' an infinite search path.

We can think of negation as finite failure (NFF) as an additional inference rule (additional to resolution). But, if you extend the proof theory, you must extend the semantics. This is done using the closed world assumption (CWA).

Logic programs are collections of facts and rules describing what is true. Untrue facts are not expressed explicitly; they are omitted. The *closed world assumption* is that: everything that is true is in the program or can be derived from the program. Or, if you prefer, everything that is not known to be true, must be false. (So we need not say that something is not true.) (Note that the CWA can only be applied to definite clauses (positive Horn clauses); applied to other clauses it gives problems.) NFF is then a procedural interpretation of the CWA: everything that is not provable is assumed to be false. Accordingly, if a clause is not in the program or cannot be derived from it, it is assumed false.

Thus, if we have the following program (and nothing else):

```
course(dec).
course(toc).
course(ssd).
course(isd).

isa_prereq_of(isd, ssd).
```

[1] See TOC —since any Turing machine can be represented by a pure Prolog program, the undecidability of the Turing machine halting problem implies that the determination of whether a goal is provable is in general an undecidable problem.

```
can_take_with_impunity(X) :- \+ isa_prereq_of(Y, X).
```

we get the following results:

```
?- course(ops).
no

?- can_take_with_impunity(dec).
yes
```

Despite Prolog's answers you should not interpret the *no* to mean that `ops` isn't a course, only that Prolog couldn't prove that it was and so, as justified by the closed world assumption, assumed that it wasn't. Nor should you conclude that you can take `dec` with impunity. Again Prolog couldn't prove there was a prerequisite, and so concluded there wasn't and hence said that you could take `dec` with impunity.

It is up to you either to ensure, if this is possible, that you include enough facts and rules so that whenever some P is true it can be proved true, or you must change your intuitive understanding of the predicate and the results Prolog gives when you use the predicate (perhaps you should even change predicate names to remind you of what is going on, e.g. `courses_typed_in_so_far`, `can_take_with_impunity_as_long_as_i_have_not_forgotten_any_prerequisite_facts`, etc.).

13.3 A Problem with Prolog's NFF

There is a problem with using Prolog's implementation of negation as failure: it is unsound. Specifically, there are instantiation problems: it does not compute what it ought if the argument to `\+` contains unbound variables.

Consider the following Prolog program, which is commonly used to illustrate this:

```
student(barry).
student(gary).

married(barry).
```

We want to find out who the unmarried students are. You would agree that the answer is `gary`. The following goal gives the expected answer:

```
?- student(X), \+ married(X).
X = gary ;
no
```

The first subgoal, `student(X)` succeeds with `X = barry`, then Prolog tries to prove `\+ married(barry)`, which means that it tries to prove `married(barry)`, which succeeds, so the negated goal `\+ married(barry)` fails. So Prolog backtracks and re-tries `student(X)`, succeeding with `X = gary`. Then it has to prove `\+ married(gary)`. When it then tries to prove `married(gary)`, this fails, so the negated goal `\+ married(gary)` is successful and the answer is `X = gary`.

But now consider the following:

```
?- \+ married(X), student(X).
no
```

This ought to get the same result (after all the comma is supposed to mean $\wedge$ and logical and is commutative). But it doesn't. First Prolog tries to prove `married(X)`. This succeeds with `X = barry`. So the negative goal fails. Thus the whole conjunctive query fails and the answer returned is *no*.

The problem in the second case is that the negative goal has unbound variables in it. In general, Prolog's implementation of NFF does not work (is not sound) for non-ground goals. It is the responsibility of the programmer to ensure that negated goals are ground before they are solved.

Here's another example. Compare the following alternative definitions (they are logically equivalent).

```
cousin1(X, Y) :-
   grandparent(G, X), grandparent(G, Y),
   \+ (parent(P, X), parent(P, Y)).

cousin2(X, Y) :-
   \+ (parent(P, X), parent(P, Y)),
   grandparent(G, X), grandparent(G, Y).

grandparent(GP, GC) :- parent(GP, P), parent(P, GC).

parent(liz, chas).
parent(liz, andy).
parent(chas, willie).
parent(andy, eugenie).
```

The use of negation in the definitions of cousinhood are designed to rule out someone being his/her own cousin. But only the first definition works correctly:

```
?- cousin1(X, Y).
Y = eugenie
X = willie ;
```

```
Y = willie
X = eugenie ;
no

?- cousin2(X, Y).
no
```

In general you avoid these problems by swapping the goals so that negated ones come after others.

There are two ways of explaining why this happens.

1) The implementation of `\+` P behaves as if any unbound variables in P were existentially quantified *inside* the negation operator, i.e. `?- \+ p(X)` is executed as if it meant `?- \+ (∃ X p(X))`, but to be consistent with normal Prolog variable scoping, it ought to be treated as quantified universally at the outermost level. This makes a difference for conjoined goals: `\+ p(X), q(X)` ought to be equivalent to $\forall X(\neg \mathrm{p}(X) \wedge \mathrm{q}(X))$, but is actually implemented as $\forall X((\exists X \neg \mathrm{p}(X)) \wedge \mathrm{q}(X))$, which is not right.

2) For a goal P, P is treated as a *completely separate query* in a computational environment of its own. If P can be satisfied, then `\+` P fails. If P fails then `\+` P is successful *without causing any substitutions.*

So substitutions can be communicated into negated goals (as in `?- student(X), \+ married(X)`) but not out of them (as shown by the problems in `?- \+ married(X), student(X)`).

The reason instantiations are not communicated *out* of negative goals is clear when you think about it. In the following goal:

```
?- \+ p(X), q(X).
```

The only way Prolog will go on to try to prove `q(X)` is if the goal `\+ p(X)` succeeds. But the only way that `\+ p(X)` will succeed is if `p(X)` fails. And we know that if a goal fails no bindings are created.

We see that the introduction of NFF into Prolog, being an extralogical operator, makes subgoal ordering in Prolog programs even more important than it was previously. It also reduces polymodality: it introduces an operator into Prolog that does not give correct answers for all patterns of instantiation.

Nevertheless, introducing some form of negation is obviously very useful as illustrated by the exercises and section 5, which shows a clever use of negation as failure.

13.4 Exercises

Before the exercises, here is a notational warning.

When applying `\+` to conjoined goals be sure to leave a space between `\+` and the bracket. `\+ (p, q)` is a negated conjoined goal. `\+(p, q)` is not. This is shown by the following programs (assume that the database is empty):

```
?- \+ (p, q).
yes

?- \+(p, q).
no
```

The former is treated as the negation of a conjoined goal (and since `p, q` cannot be proved, the negation succeeds); the latter is treated as a predicate applied to arguments `p` and `q`. And since there are no clauses in the database with `\+` as their predicate, this query fails.

Exercise 13.1 *Create some clauses that define censored words, e.g.:*

```
censored(bother).
censored(drat).
censored(damned).
censored(blast).
censored(fiddlesticks).
```

Write a predicate that takes a list of words and produces a new list with the censored words beeped out. For example, for the facts above:

```
?- beep([prolog, is, a, damned, shitty, language], B).
B = [prolog, is, a, beep, shitty, language] ;
no
```

Exercise 13.2 *Give definitions for the following predicates that manipulate lists as if they were sets.* `setMember(Item , Set)` *is true if* `Item` *is an element in the list* `Set`*;* `setInsert(Item, Set1, Set2)` *is true if* `Set2` *is the result of adding* `Item` *to the list* `Set1` as long as it is not already there[2]*;* `setDelete(Item, Set1, Set2)` *is true if* `Set2` *is the result of deleting* `Item` *from* `Set1`*;* `subset(Set1, Set2)` *is true if all the elements of* `Set1` *are in* `Set2`*;* `setEquals(Set1, Set2)` *is true if the two sets contain the same elements not necessarily in the same order*[3] *(you can use* `subset` *to do this);* `setUnion(Set1, Set2, Set3)` *is true if* `Set3` *is the union of the two other sets (again without duplicates);* `setIntersect(Set1, Set, Set3)` *is true if* `Set3` *is the intersection of* `Set1` *and* `Set2`*; and* `setDiff(Set1, Set2, Set3)` *is true if* `Set3` *is the set difference* `Set1` *minus* `Set2`*.*

Chances are that your definition of `setInsert` *(and probably* `setUnion`*) are not fully polymodal: they can give* wrong *answers for one particular instantiation pattern. Discover what this is.*

[2] Remember, sets do *not contain duplicates.*

[3] Remember, sets are *not ordered.*

Exercise 13.3 *In this exercise you construct in Prolog simple robot planners that are considerably more general than the one you built for monkey planning.*

States of affairs will be represented using a list of facts. For example, the following

```
[at(door), on(floor)]
```

represents a state of affairs in which the robot is located at the door of a room and is standing on the floor.

The robot has a repertoire of actions which it can perform. These actions are defined by the `action` *predicate, which has four arguments:* `action(ActionName, PrecondList, AddList, DeleteList)`*. The action,* `ActionName`*, can be carried out in a state in which all the facts in the precondition list,* `PrecondList`*, are true. Carrying out the action creates a new state of affairs which is the same as the old state of affairs except that the facts in the add list,* `AddList`*, are now additionally true, and the facts in the delete list,* `DeleteList`*, are no longer true. (For any action, the facts in its delete list will be a subset of the facts in its precondition list.)*

Here are some example actions for a robot which can move around a room, climb on and off a box which is fixed beneath a shelf in the room, and take a jar of honey from the shelf:

```
action(move(door, shelf), [at(door), on(floor)], [at(shelf)], [at(door)]).

action(move(shelf, door), [at(shelf), on(floor)], [at(door)], [at(shelf)]).

action(climb(floor, box), [at(shelf), on(floor)], [on(box)], [on(floor)]).

action(climb(box, floor), [on(box)], [on(floor)], [on(box)]).

action(get(honey, shelf), [on(box)], [have(honey)], []).
```

The first of the above clauses, for example, says that the action of moving from the door to the shelf can be performed in a state in which the robot is at the door and on the floor. The resulting state will include the additional fact that the robot is at the shelf, and will have lost the fact that the robot is at the door.

A predicate `forwards(InitialState, GoalConditions, Actions)` *is to be defined. An example of its use is:*

```
?- forwards([at(door), on(floor)], [have(honey)], Actions).
Actions = [move(door, shelf), climb(floor, box),
           get(honey, shelf)]
```

etc.

`forwards` *finds a sequence of actions that would take the robot from an initial state that is fully specified as the first argument of* `forwards` *to a state in which the facts in the second argument of* `forwards` *are among the true facts.*

This path is found by forward-chaining *from the initial state to a state satisfying the goal condition, i.e. by finding an action that can be applied to the initial state, carrying it out to get a new state, finding an action that can be applied to this new state, carrying this out, and so on until the goal condition is satisfied.*

Give a Prolog definition of `forwards` *making use of the set operations from the previous exercise.*

Here's some stuff to start you off.

`forwards/3` *calls an auxiliary predicate* `forwards/4`*, which initialises the actions so far to* `[ ]`*:*

```
forwards(InitialState, GoalState, FinalActions) :-
   forwards(InitialState, GoalState, [], FinalActions).
```

`forwards/4` *can stop when the conditions of the goal state are a subset of those in the state we've reached at this point. In this case, the actions so far are copied (in reverse*[4]*) into* `FinalActions`*.*

```
forwards(CurrentState, GoalState, RevActions, FinalActions) :-
   subset(GoalState, CurrentState),
   reverse(RevActions, FinalActions).
```

In the third clause of `forwards/4`*, which you must complete, you get an action and see whether it is one that you have used before (to avoid infinite loops). If it is not, you check it is applicable in this state, apply it and then recursively call* `forwards/4`*:*

```
forwards(CurrentState, GoalState, RevActions, FinalActions) :-
   FOR YOU TO COMPLETE
```

(Don't forget that you must include your set operator definitions and definitions of `reverse` *and hence* `append` *in your file.)*

Similarly, define `backwards(InitialState, GoalConditions, Actions)` *which solves the same problem but does so by* backwards-chaining *from the goal conditions to the initial state, i.e. it finds an action that could cause the goal condition to arise, carries the action out 'in reverse' as it were, to find out what the state prior to the goal state would need to be like, then finds an action that could have resulted in this state, and so on until it reaches the initial state.*

[4] They are reversed assuming that, as you collect new actions, you stick them onto the front of the list of actions so far. This means that the most recent action is at the front of the list, hence the need to reverse it. This isn't needed in the definition of `backwards`.

You can play with your planners by giving them different action repertoires. (Don't put variables into the action specifications.)

13.5 Failure-Driven Loops using Negation as Failure

We can use Prolog's negation, and sometimes double negation, to achieve 'effects' that are not strictly licensed by the logic. In particular, here we show how to implement failure-driven loops using negation. Failure-driven loops were described in worksheet 12, where we pointed out their use in processing non-recursive data structures. In that worksheet, we exploited Prolog's search rule. The implementation given here exploits the non-logical nature of Prolog's negation. The result is tidier than before. Nevertheless, the all-solutions predicates introduced in the next section are an even better way of writing programs that process non-recursive data structures.

An employment agency has a number of people on its books, where each of a person's qualifications is represented by a clause:

```
qual(pers1, typing5).
qual(pers1, frenchA).
qual(pers1, piano4).
qual(pers2, csBSc).
```

Equally it records job opportunities and their requisite qualifications:

```
entry(job12, piano4).
entry(job12, frenchA).
entry(job17, csBSc).
```

We want to define `shouldApply(Person, Opportunity)` which is true if the person has all the qualifications needed for the job.

Here's how:

```
shouldApply(P, J) :-
   \+ (entry(J, Q),
      \+ (qual(P, Q))).
```

Explaining how and why this works is hard. First one can try to see it as a piece of logic, bearing in mind that unbound variables in negated goals are existentially quantified within the negation. The corresponding logic is along the lines of:

$$\forall P \forall J (\neg \exists Q (\text{entry}(J, Q) \wedge \neg \text{qual}(P, Q)) \rightarrow \text{shouldApply}(P, J))$$

which is equivalent to

$$\forall P \forall J (\forall Q (\text{entry}(J, Q) \rightarrow \text{qual}(P, J)) \rightarrow \text{shouldApply}(P, J))$$

which seems right!

Secondly, one can try to understand it procedurally. This is perhaps best done with a couple of example goals.

```
?- shouldApply(pers1, job12).
yes
```

What happens here is that first Prolog picks up an entry qualification needed for `job12`: `piano4`. It then sees whether the person has that qualification: `pers1` does have this qualification, so the goal `qual(pers1, piano4)` succeeds. But it is a negated goal, so this success is changed into a failure. This forces Prolog to backtrack to pick up another entry qualification for `job12`: `frenchA`. Again it sees whether `pers1` has this qualification; s/he does, so the goal `qual(pers1, frenchA)` succeeds, but being negated, this is changed to failure and so Prolog backtracks again. It now tries to pick up another qualification needed for `job12`. But there are no more, so the goal `entry(job12, Q)` fails, so the whole conjunctive goal `(entry(job12, Q), \+ qual(pers1, Q))` fails; but this whole conjunctive goal is a negated one, so this failure is turned into success, and Prolog returns *yes*. And this is correct: for every entry qualification we picked up, we found a corresponding qualification for `pers1`.

Consider instead:

```
?- shouldApply(pers2, job12).
no
```

First Prolog finds an entry qualification for `job12`: `piano4`. Then it determines whether `pers2` has this qualification; s/he doesn't, so the goal `qual(pers2, piano4)` fails. But it is a negated goal, so this failure is turned into success. So the whole conjunction has been successful. But the conjunction is negated, so this success is turned into failure, and Prolog returns *no*. And this is correct: this is at least one entry qualification that the person didn't have.

☐ Trace the above to see how my English description of what is going on corresponds to the goals Prolog forms.

You can see that we are relying on the behaviour of Prolog negation to get a looping behaviour into our program without using recursion: we make successful goals fail in order to force a backtrack.

Here's another example:

```
cleverSubset(Xs, Ys) :-
   \+ (member(Z, Xs),
      \+ (member(Z, Ys))).
```

The behaviour is similar: you take a member of `Xs` and check it is in `Ys`; if it is, the negation forces a backtrack, to get another member of `Xs`. And so on, until all members of `Xs` have been tried and the failure to find another member of `Xs` is reported as the success of the definition as a whole. On the other hand, if at any time, a member of `Xs` is not a member of `Ys`, failure is immediately reported.

```
?- cleverSubset([a, b, c], [c, d, a, b]).
yes
```

There are some dangers with this approach. For example, if I forget to load the definition of `member` when loading the definition of `cleverSubset`, and then call, e.g.

```
?- cleverSubset([c, d, a, b], [a, b, c]).
yes
```

the answer is *yes*!! The problem is that Prolog tries the first call to `member`, but `member` has no definition and so the call fails, and this failure is turned into success by the negation.[5]

These definitions therefore need to be used with care.

Exercise 13.4 *The Achin' Hearts Dating Agency rides again (as their advertising slogan ambiguously puts it). This time their representation is in terms of people and lists of interests, e.g.:*

```
person(case1, [pop, walks, cinema]).
person(case2, [walks]).
person(case3, [scifibooks, roleplaygames, festeringanoraks]).
```

Define, using negation, `mightDate(Person1, Person2)`*, which is true if* `Person1`*'s interests are a subset of* `Person2`*'s interests.*

Use this to define `wildParty(People1, People2)`*, which is true if the* combined *interests of the people in the list* `People1` *are a subset of the* combined *interests of the people in the list* `People2`*.*

Exercise 13.5 *The following are logically equivalent:*

```
check_for_giraffes1(X) :-
  giraffe(X).

check_for_giraffes2(X) :-
  \+ ( \+ giraffe(X)).
```

[5]Actually, this depends on your version of Prolog. There are versions that, in an effort to protect programmers from inadvertent errors, will report an error when a predicate that does not appear in the facts or in the heads of rules is used in a goal.

But due to the way Prolog's negation works they actually do different things. What's the difference?

Think about it first and then see what you get when you try them out in Prolog both with and without a giraffe fact (e.g. `giraffe(gertie)`*) added to the database.*

13.6 Higher-Order Logic

13.6.1 Quantification over predicates

Prolog, like most logic programming languages, is based on first-order logic. First-order logic allows quantification over individuals: (roughly) you can put variables in the argument positions of predicates. Second-order logic, which is an example of a higher-order logic, allows quantification over predicates: (roughly) you can put variables in predicate positions. For example,

$$\forall P(P(\text{clyde}) \rightarrow P(\text{flopsy})$$

i.e. for all 'properties' P, if P is true of Clyde, then P is also true of Flopsy. If we allow something like this in a logic programming language, then predicates become 'first-class' objects, capable of being passed around by our programs. (This clearly brings us strongly in line with functional programming, where the idea that functions are data is commonplace.)

Let us define the logic programming equivalent to the functional programming function *map*. In functional programming, the function *map* applies a function to each element of a list, returning the list of new elements as its result. In logic programming, the *ternary* predicate `map(P, Xs, Ys)` will denote a relation which is true if, for each element `X` of `Xs`, there is a corresponding element `Y` of `Ys` (preserving the original ordering) such that `P(X, Y)` is true.

Here's the definition:

```
map(P, [], []).
map(P, [X|Xs], [Y|Ys]) :- P(X, Y), map(P, Xs, Ys).
```

Note how `P` is being passed around as data, and note that the first subgoal of the second clause, `P(X, Y)`, uses the predicate variable `P`.

Some, though not many, Prologs will allow you to enter such a definition. Our will not! (You'll get a syntax error if you try.) For now, let's proceed as if it did.

Suppose the user had entered some facts:

```
likes(bill, ben).
```

```
likes(ben, bill).
likes(weed, weed).
```

then s/he can issue the following query:

```
?- map(likes, [weed, ben, bill], Rs).
Rs = [weed, bill, ben]
```

which is what we want.

Here's how we achieve this in Prologs, such as our own, which do not allow subgoals such as `P(X, Y)`.

We use another in-built predicate, called *univ* and written `=..`. In `X =.. Y`, `Y` is a list whose head is the atom corresponding to the principal functor of `X`, and whose tail is the argument list of that functor in `X`. Huh? Examples will make this clear:

```
?- likes(bill, ben) =..  G.
G = [likes, bill, ben]

?- likes(X, Y) =..  G.
G = [likes, X, Y]

?- F = [likes, bill, Y].
F = likes(bill, Y)
```

Here is how we use this:

```
map(P, [], []).
map(P, [X|Xs], [Y|Ys]) :- Goal =.. [P, X, Y],
                          call(Goal),
                          map(P, Xs, Ys).
```

Now we do not have a subgoal `P(X, Y)`. Instead, we construct a goal using uinv, and then execute that goal. This relies on `P` being instantiated by the time the univ is called (otherwise the univ will fail). In typical user queries, such as `?- maplikes, [weed, ben, bill], Rs)`), `P` is instantiated, so we're alright.

(`call` is an in-built predicate, whose argument must be instantiated to a term which would be acceptable as the body of a clause, and which will execute that argument exactly as if that argument appeared textually in place of the use of `call`. In fact, in most Prologs, including our own, you don't need to use `call`; the following will suffice:

```
map(P, [], []).
map(P, [X|Xs], [Y|Ys]) :- Goal =.. [P, X, Y],
                          Goal,
                          map(P, Xs, Ys).
```

I prefer to use `call` to emphasise that I know what I'm doing.)

As in the example where `P` is `likes`, `P` must be instantiated by the time it is being used in the first subgoal of the second clause, i.e. you cannot do the following (even in Prologs that accept the original definition of `map`):

```
?- map(P, [weed, ben, bill], Rs).
```

Arguably, if this were a proper implementation of higher-order logic, you might hope for an answer such as:

```
Rs = [weed, bill, ben]
P = likes
```

i.e. you might hope the system would find predicates that `P` can be instantiated to. This won't happen.

□ Load the definition into Prolog and try it out.

Now, in much the same way as you did in functional programming, you can express other programs in higher-order ways, e.g. the answer to exercise 5.8 (machine translation) becomes simply:

```
translate(Words, Mots) :- map(dict, Words, Mots).
```

Exercise 13.6 *Define a ternary predicate* `filter` *such that* `filter(P, Xs, Ys)` *is true if the elements of* `Ys` *are those elements of* `Xs` *that satisfy* `P`*.*

Our definitions of `map` (and `filter`) still lack generality. For example, your definition of `filter` will not allow a binary predicate with an argument already instantiated, e.g. to keep those items of a list of numbers that are greater than 4, you cannot write:

```
?- filter(X > 4, [5, 4, 2, 7, 1], Rs)
```

nor even the mathematically more appropriate:

```
?- filter(lambda(X, X > 4), [5, 4, 2, 7, 1], Rs)
```

This sort of thing requires a cleverer way of constructing the goals.

We will not pursue this further. Suffice it to say that it can be done, but the use of higher-order constructs is not as widespread in Prolog as it is in functional programming anyway.

However, there are some in-built higher-order facilities, which we will call 'all-solutions' predicates, that are widely used.

13.6.2 All-solutions predicates

The in-built predicates can, in particular, be useful for processing non-recursive data structures without resorting to either of the implementations of failure-driven loops that we saw in the previous worksheet and in the presentation of negation earlier in this worksheet.

`findall(Term, Goal, Bag)` can (roughly) be read as 'Bag is the list of all instances of Term such that Goal is provable'. If Goal is not provable at all, Bag will be the empty list.

So, consider this database:

```
father(phil, chas).
father(phil, andy).
father(phil, eddie).
father(chas, willie).
father(chas, henry).
```

Here are sample uses of `findall`:

```
?- findall(X, father(phil, X), Xs).
Xs = [chas, andy, eddie]

?- findall(X, father(X, chas), Xs).
Xs = [phil]

?- findall(X, father(eddie, X), Xs).
Xs = []

?- findall(pair(X, Y), father(X, Y), XYs).
XYs = [pair(phil, chas), pair(phil, andy), pair(phil, eddie),
       pair(chas, willie), pair(chas, henry)]

?- findall(Y, father(X, Y), Ys).
Ys = [chas, andy, eddie, willie, henry]
```

Note this last one. When a variable appears in the second argument that does not appear in the first argument (`X` in this case), that variable is treated as existentially quantified (so the example reads (roughly) as: find all `Y` such that there exists an `X` and `X` is the father of `Y`).

`bagof(Term, Goal, Bag)` is just like `findall` in all but two cases. In other words, if you use `bagof` in place of `findall` in the above examples, you will get the same answers in all of them except the following:

```
?- bagof(X, father(eddie, X), Xs).
no
```

Here `findall/3` returned `Xs = []`, whereas `bagof` fails, i.e. the two differ when there are no ways of proving the second argument.

And they differ in cases where there are variables appearing in the second argument and not in the first:

```
?- bagof(Y, father(X, Y), Ys).
Ys = [willie, henry]
X = chas ;

Ys = [chas, andy, eddie]
X = phil
```

i.e. instead of a single solution, all people who have a father, there is a solution for each father.

You can make `bagof` behave like `findall` in these cases by signalling that you want `X` to be existentially quantified rather than free, as follows:

```
?- bagof(Y, X^father(X, Y), Ys).
Ys = [chas, andy, eddie, willie, henry]
```

There is also `setof(Term, Goal, Bag)`, which is exactly like `bagof` except that, at the cost of slower execution, it sorts the answer list and removes duplicates from it.

□ Try out all the above in Prolog.

Exercise 13.7 *Define the predicate* `intersect(Xs, Ys, Zs)` *using an all-solutions predicate to compute the intersection* `Zs` *of the two lists* `Xs` *and* `Ys`*, (Think about what should happen if* `Xs` *and* `Ys` *have no elements in common.)*

Further exercises that use all-solutions predicates appear at the end of the next worksheet.

Acknowledgements

This material is an original concoction whose constituents came from a conversation with Colin Runciman, from some unpublished notes by Clocksin and from the books Bratko, Sterling and Shapiro, Clocksin and Mellish, and F.C.N.Pereira and S.M.Shieber, *Prolog and Natural Language Analysis*, CSLI, 1987.