Worksheet 4

Pure Prolog

In this worksheet, we add function symbols to DataLog, thus giving us Pure Prolog. The aims are that

- you will learn how function symbols can give us some mechanism for adding structure to our data;
- you will learn that functions are simply structuring devices and are *not* evaluated;
- you will try to get your head around the kinds of programs that manipulate structured data;
- you will learn that the data structures they create can be partial (by having variables in them).

4.1 Preliminaries

You run Pure Prolog just as you ran PropLog and DataLog.

PropLog is a restricted version of DataLog, and DataLog is a restricted version of Pure Prolog: everything you typed in to PropLog and DataLog can also be typed into Pure Prolog. But Pure Prolog generalises DataLog. Specifically, the arguments to predicates can be more complicated.

4.2 Structured objects

So far the arguments to predicates have all been simple constant symbols or variables. Constant symbols and variables are called *terms*. There is a third type of term, which is allowed in Prolog but not in DataLog. This third type of term consists of a *function symbol* applied to other terms. Function symbols begin with lowercase letters.

Fact (1) is represented using simple constants; fact (2) represents the 'same' information using some function symbols:

```
(1) person(william, shakespeare, playwright).

(2) person(name(first_name(william), surname(shakespeare)),
           occupation(playwright)).
```

The function symbols in the second clause are:

- `name`, which is a function taking two arguments,
- `first_name`, which takes one argument,
- `surname`, which takes one argument, and
- `occupation`, which takes one argument.

You can see that functions allow us to structure our data in a richer way.

If you use functions, you have to be careful in rules and queries if you want to ensure a match: you must supply the right number of arguments in the right order. For example, the following queries will all match the fact given above that uses function symbols (fact (2)):

```
?- person(Name, Occ).
?- person(name(First, Sur), Occ).
?- person(name(First, Sur), occupation(Occ)).
```

However, the following won't match and will return the answer ***no***:

```
?- person(name(Name), Occ).
```

This fails to match because `name` requires two arguments.

You'll remember that you cannot use variables where predicate symbols are supposed to appear. Equally you cannot put variables where function symbols should appear:

```
?- person(X(First, Sur), Y(Occ)).
Syntax error
```

The `X` and `Y` are bad. (But `?- person(X, Y)` is OK because here the variables stand for the whole terms, not just the function symbols).

□ Load fact (2) about Shakespeare into Prolog and try out all the queries given in the text above.

You can see that this business of 'matching' about which we've been fairly informal so far is crucial to the way Prolog works. Let's try to spell it out a bit more (this is an informal characterisation; the full method will be given later in the course and will be referred to as *unification*):

- Two terms match if substitutions can be made for any variables in the terms so as to make the terms identical.
- To match two terms, determine whether their corresponding parts match. Two constants match if they are identical. A variable will match anything. Two structured objects will match if their function symbols are identical and their components match.
- If two terms match, then Prolog will need to know what was substituted for the variables in the terms.

Example: `f(X, g(b, c))` and `f(d, g(Z, c))` are terms which match. `d` is substituted for `X` and `b` is substituted for `Z`.

Exercise 4.1 *Say whether the following pairs of terms match, and if they do, say what substitutions are made for the variables:*

(a) `f(X, g(b, c))` *and* `f(Z, g(Y, c))`

(b) `f(X, g(b, c))` *and* `f(Z, g(Z, c))`

(c) `f(c, g(b, c))` *and* `f(Z, g(Z, c))`

(d) `g(Z, f(X, a, Y), h(X, Y), a)` *and* `g(W, f(V, V, U), W, U)`

4.3 Functions Are Not Evaluated

For good theoretical reasons, functions in logic programming languages are not evaluated to give a result, rather they are simply used to construct new compound terms. These compound terms are like record structures in other languages, i.e. they are used to represent structured information. To illustrate this, consider the function `surname` in the Shakespeare example. This has as its argument `shakespeare`. Although `surname` was a function, it didn't return any result. All that we got was a compound object `surname(shakespeare)`.

Similarly, if I create a term from the function `plus` and the arguments `2` and `3`, e.g. `plus(2, 3)`, this will not return `5`, i.e. it won't be evaluated. It will simply stay as the compound object `plus(2, 3)`.

This is an issue that we need to come back to as it lies at the heart of fundamental questions in logic programming.

For now, I want to show that even though functions are not evaluated, we can still do some useful work with them.

Firstly, I give two exercises which use function symbols to structure the data in a richer way.

4.3.1 Simple examples

Exercise 4.2 *Copy the file* `flights`*, which contains facts of the following form, showing which airports have direct flights between them on which days:*

```
flight(from(name(lhr), city(london)),
       to(name(bsl), city(basel)),
       day(tuesday)).

flight(from(name(bsl), city(basel)),
       to(name(cdg), city(paris)),
       day(tuesday)).
```

(The file deliberately doesn't contain any cyclic routes.)

Define the predicate `connects(dep_city(D), arr_city(A), day(DY))` *which is satisfied if there is a sequence of flights that connects some airport in city* `D` *to some airport in city* `A` *on day* `DY`*. (Your definition should be fully polymodal.)*

In the exercise above, the function symbols are not really needed; they simply make the clauses that we write more self-documenting. For example, writing the two clauses in the fashion shown above is (perhaps) clearer than writing:

```
flight(lhr, london, bsl, basel, tuesday).

flight(bsl, basel, cdg, paris, tuesday).
```

where it is not completely clear which airport is the starting place of the flight and which is the destination. Other than this, the definition of `connects` will have been very similar to the recursive definitions given in the previous chapter.

The next exercise is similar in this regard: again the function symbols make the clauses more self-documenting, but the heart of the program is a simple recursive definition like those given in the previous chapter.

Exercise 4.3 *Simple A.I. Planning*

There is a monkey at the door into a room. In the middle of the room a banana is hanging from the ceiling. The monkey is hungry and wants to get the banana, but he cannot stretch high enough from the floor. At the window of the room there is a box the monkey may use.

The monkey can perform the following actions: walk on the floor, climb onto the box, push the box around (if the monkey is already at the box), and grasp the banana if standing on the box directly under the banana.

Write a Prolog program which can determine whether the monkey can get the banana.

Some help...

We are going to use a complex term to represent a state of the world. States have 4 components: the place of the monkey in the room, whether it is on the box or not, the position of the box in the room, and whether the monkey has the banana or not.

The initial state has the monkey at the door and on the floor, the box by the window, and the monkey not having the banana. This is represented as:

```
state(at_door, on_floor, at_window, has_not)
```

(You don't need to do anything with this: it will be included in your query, not in your file.)

a) You have been given the initial state, now define the goal state. (You don't need to do anything with it until part (c).)

Now consider moves which change the world from one state to another. These can be represented as 3-place predicates, `move(State1, M, State2)`*.* `State1` *is the state before the move (and so must capture any preconditions of the move),* `M` *is the move itself, and* `State2` *is the state after the move (and so must capture any effects of the move).*

For example, grasp the banana. This is a move that can only happen if the monkey is standing on the box, is directly under the banana, and does not have the banana yet. So grasp is defined with its necessary preconditions and effects as follows:

```
move(state(under_banana, on_box, under_banana, has_not),
     grasp,
     state(under_banana, on_box, under_banana, has)).
```

b) Define 3 more `move` *clauses for (1) climbing on the box, (2) moving the box from* `Place1` *to* `Place2`*, and (3) the monkey walking from* `Place1` *to* `Place2`*. (Put the four* `move` *clauses into the file in the order I have specified them!)*

We will want to use the query:

```
?-canget(state(at_door, on_floor, at_window, hasnot)).
```

and get the answer `yes` *if the monkey can get the banana and* `no` *if it can't from the starting state specified in the query.*

c) Define `canget` *and try out your program. (Remember, it's just a simple recursive definition like those given in the previous chapter. Ask yourself what the base case is, i.e. what state would be the argument of* `canget` *for it to return* `yes`*? Ask yourself what recursive call would take you nearer to a solution.)*

(d) Is your definition polymodal? Suppose the order of the grasp, climb, push and walk clauses were different (e.g. if walk appears first); what ramifications does this have for polymodality?

You've now built a little robot planner, or in this case a monkey planner.

The two exercises have, as we have said, used function symbols to make the programs more self-documenting. Now we want to show some cases where they are essential. We will consider how a program might actually *create* some structured object as it runs and return the structured object as its result. This will help to show how relational programs such as logic programs differ from functional ones.

4.3.2 Peano's Axiomatisation of Arithmetic

We will use Peano's axiomatisation of arithmetic as a small case study in the use of Prolog function symbols.

When answering the exercises, please do not adopt a 'guess-and-test' mentality to Prolog program writing. This is a disastrous approach. I hope that this subsection will show you that there are a number of issues to think about that can make it more likely that you will write a correct program first time (provided you 'think first, code later').

Here are some of the ways that we are accustomed to writing down symbols that represent the number 4 in our everyday work:

4	arabic number
IV	roman numeral
||||	tallying
four	English
quatre	French
vier	German
100	binary number

Since there are already many ways, there can be little harm in inventing another:

```
s(s(s(s(0))))
```

In this notation, the simple constant `0` is intended to represent the number 0; the term `s(0)`, constructed out of the function symbol `s` (to be thought of as the successor function) represents the number 1; `s(s(0))` represents the number 2, and so on.

We can write Prolog programs that do arithmetic on numbers represented in this fashion. Here's a simple example in which we define `peano_no(X)`, which will be true if `X` is instantiated to a number as represented above:

```
peano_no(0).

peano_no(s(X)) :- peano_no(X).
```

Hear are examples of its use:

```
?- peano_no(0).
yes

?- peano_no(s(s(s(0)))).
yes

?- peano_no(3).
no

?- peano_no(s(s(1))).
no
```

The definition is not polymodal: if its argument is uninstantiated, an infinite computation is launched:

```
?- peano_no(X).
X = 0 ;
X = s(0) ;
X = s(s(0)) ;
X = s(s(s(0))) ;
etc.
```

(It's unlikely that you'll have the patience to type enough semi-colons to get it to run out of stack space!) That it launches an infinite computation should come as no surprise for two reasons. Firstly, remember my rule of thumb that you should not put the recursive call leftmost. But here, since there is only one subgoal in the body of the recursive rule, we had no choice other than to put it leftmost. Secondly, it is clear that in posing the goal `peano_no(X)` you are asking a question to which there is an infinite number of answers. Once we add function symbols to DataLog to give us Pure Prolog, we move into a system in which we can, in principle, construct an infinite number of objects.

Note how the definition reflects the structure of our representation of the natural numbers: there is a clause for `0` and a clause for use of the successor function, `s`. This is a basic principle of writing recursive programs for structured objects:

RULE OF THUMB

Your program structure should reflect the structure of the data.

Exercise 4.4 *Define* `lt(X, Y)` *which is true if the number* `X` *is less than the number* `Y`*. (Make use of the rule of thumb!) Test the program with all four different patterns of instantiation.*

Exercise 4.5 *Modify your definition of* `lt(X, Y)` *to give* `lte(X, Y)`*, which is true if* `X` *is less than or equal to* `Y`*.*

The definitions you will have given as answers to the previous exercises will have a recursive rule whose recursive call recurses on both the first and second argument. This is very rare: usually, you recurse on only one argument:

RULE OF THUMB

Recurse on only one argument unless there are very good reasons to do otherwise.

Here's another example in which we define `add(A, B, C)`, whose intended meaning is that `C` is the sum of `A` and `B`:

```
add(0, Y, Y).
add(s(X), Y, s(Z)) :- add(X, Y, Z).
```

Look at the first argument. You see that this is the one being recursed on. You also see that again the recursive structure of our representation of natural numbers is reflected in the definition: the first clause deals with `0`, the second with the successor of some number.

We can give some queries to show that these clauses do indeed define addition of the first two arguments in the notation for integers that we introduced above:

```
?- add(0, 0, R).
R = 0 ;
no

?- add(0, s(0), R).
R = s(0) ;
no

?- add(s(0), s(s(0)), R).
R = s(s(s(0))) ;
```

```
no

?- add(s(s(s(0))), s(s(0)), R).
R = s(s(s(s(s(0))))) ;
no
```

These correspond (in our heads) to computing $0+0=0, 0+1=1, 1+2=3$ and $3+2=5$ respectively.

Note that the answers do not reduce to (or get simplified to or evaluate to) values like 0, 1, 2, 3, etc. The only relationship between the definitions above and what we know about arithmetic is entirely in our heads: it's our way of interpreting what's going on. It is not as if the machine knows what meaning we are ascribing to its symbol manipulation. (It's asking a bit much to expect the machine to work this out and then make use of its low-level capacity for doing arithmetic directly.)

□ Try out the `add` rules as above. Also, explore the polymodality. (Since the recursive call is leftmost and also since some queries will ask questions that have an infinite number of answers anyway, you won't get full polymodality, but there will be some polymodality. Since, there are three arguments, there are at least eight patterns of instantiation to try.)

Exercise 4.6 *In the light of what you have discovered about the polymodality of* `add`*, define* `sub(X, Y, Z)` *in terms of* `add`*, where* `X` *minus* `Y` *gives* `Z`*.*

We still haven't explained how the definition of `add` works. Doing so is very revealing as it exemplifies the difference between functional programming and relational programming. In a functional program, `add` would be a binary function: it would take in the two numbers to be added, the answer would be computed in the body of the definition, and it would then be 'returned' from the body. This is not how things happen in relational programs. The logic program is ternary (three arguments), the first two being the numbers being added, the third being the 'answer'. So note that the answer is 'built up' in the third argument, it is not returned by the body.

Specifically, the first clause of the definition

```
add(0, Y, Y).
```

states that `0` added to any number, `Y`, yields that number, `Y`.

The second clause

```
add(s(X), Y, s(Z)) :- add(X, Y, Z).
```

states that the successor of any number, `X`, added to some other number, `Y`, is the successor of `Z`, where `Z` is the sum of `X` and `Y`.

I repeat: the answer is not being returned from the body of the rule as it would be in a functional program; it is being constructed in one of the arguments in the head of the rule.

Exercise 4.7 *Define* `incr(X, Y)`*, which is true if* `Y` *is one more than* `X`*, i.e.* `Y = X+1`*. Do* not *use* `add` *to do this. The answer is really very simple: there's no recursion involved; just remember to build the answer in the second argument. Note that your answer can also be used to decrement* `Y` *giving* `X`*.*

Exercise 4.8 *Define* `double(X, Y)`*, which is true if* `Y` *is twice* `X`*, i.e.* `Y = X+X`*. Do* not *use* `add` *to do this. The answer will be similar in style to the definition of* `add`*.*

`add` and `double` show one style of recursive program for manipulating structured data, building up an answer in one of the argument positions. Now we show another (related) style.

Here's another definition of doubling (the one I told you not to give):

```
timestwo(X, Y) :- add(X, X, Y).
```

This one feels different to the one above: the answer *is* computed in the body (as the sum of `X` and `X`, called `Y`) and then 'copied' into the second argument in the head.

Here's another example: `mult(X, Y, Z)` is true if `Z` is the product of `X` and `Y`:

```
mult(0, Y, 0).

mult(s(X), Y, Z) :- add(Y, Int, Z), mult(X, Y, Int).
```

(Again note the recursion in the first argument, reflecting the definition of the structured object.) The first clause says that any number, `Y`, multiplied by `0` is `0` (the third argument). The second clause says that any number, `Y`, multipled by the successor of `X` is `Z`, this being the result of adding `Y` to the product of `X` and `Y`. Here, as in `timestwo` above, the answer is built in the body and 'copied' into the head.

(Of course, the style of this definition of `mult` is not really very different from that of the definition of `add`: the only difference really is that in `add` the thing computed in the body, `Z`, is 'copied' into the head but has further structure built around it to give `s(Z)`, whereas in `mult`, the thing computed in the body, `Z`, is 'copied' into the head without further structure being built around it.)

One final point about the definition of `mult`: how polymodal is it? It uses `add`, and so it is a matter of concern to know how polymodal `add` is, because if add is called in the wrong modes (i.e. if, by the time add is called, its arguments are not instantiated in a suitable pattern), then this can cause mult to fail. In fact, the definition of mult given above has four working modes; if you swap the order of the subgoals in the recursive rule, you get only three working modes. You can see how complicated the quest for polymodality can become!

Exercise 4.9 *Using the same style of definition, define* `factorial(X, Y)`*, which is true if* `Y` *is* `X` *factorial. (You will use* `mult` *in your answer; the order of the subgoals in the recursive clause requires some thinking about because of the only partial polymodality of* `mult`*!)*

Exercise 4.10 *In a similar style to* `mult`*, define* `exp(X, Y, Z)`*, which is true if* `X` *raised to the power* `Y` *is* `Z`*. (Use* `mult` *in the answer.) (And watch out: (i) assuming you take* $0^0 = 0$ *but* $n^0 = 1$ *for all* $n > 0$*, then there are two base cases, (ii) you must think very hard about which argument to recurse on, and (iii) the order of subgoals in the recursive clause requires some thinking about, as in* `factorial` *above.)*

There is a third style of definition for Prolog programs that build structured objects as their answers. You will have seen this idea in the functional programming part of the course: the use of an *accumulator*. Here, by way of example, is a definition of factorial that uses an accumulator:

```
fact(X, Y) :- fact_aux(X, s(0), Y).

fact_aux(0, Y, Y).
fact_aux(s(X), Y, Z) :- mult(s(X), Y, W), fact_aux(X, W, Z).
```

In `fact_aux`, the second argument position is the accumulator. In the call to `fact_aux`, made in the body of `fact`, the accumulator is initialised to `s(0)` (i.e. to 1). In `fact_aux`, if the first argument is `0`, i.e. we are computing the factorial of zero, then the value of the accumulator, the second argument `Y`, is 'copied' into the third argument. This will be the answer returned by `fact`. In `fact_aux`, if the first argument is not zero (it is `s(X)`), then the accumulator is multiplied by `s(X)` to give `W`, and we recurse using the new value of the accumulator.

Exercise 4.11 *Define* `product(X, Y, Z)`*, where* `Z` *is the product of* `X` *and* `Y`*, using this idea of an accumulator. As in the definition of* `fact`*, you will need an auxiliary definition to do all the hard work after* `product` *has done suitable initialisation. (You can use* `add` *in your answer.)*

In summary, don't 'guess-and-test'. Use the rules of thumb. State your definition in English first, and then code it up.

4.3.3 Differentiation

We will now use our axiomatisation of the natural numbers to write a program to do differentiation.

Differentiation is a process that takes in one polynomial and returns another polynomial. So, to begin with, we need to determine how we will represent polynomials.

Here is a simple polynomial:

$$2$$

This is easily represented as the term:

```
s(s(0))
```

This one:

$$2x$$

can be represented as the term:

```
times(s(s(0)), x)
```

i.e. a function, `times`, applied to the terms `s(s(0))` and `x`. Note two things about this:

- The variable x is being captured as a Prolog *constant* `x`, not as a Prolog variable `X`.
- In this, `times` is just like any of the other function symbols that we have made up so far (e.g. `name, occupation, s`). Therefore, its use simply creates a compound data structure. It does not cause any attempt at evaluation.

Equally well, instead of using the cumbersome symbol `times`, we could use the shorter symbol, `*`:

```
*(s(s(0)), x)
```

Again, `*` is just like other function symbols we have invented: no evaluation takes place (in particular, `*(2, 3)`, say, does not 'return' 6). Since it involves less typing, I will use `*` instead of `times` henceforth.

I will assume that all our polynomials are in terms of x, and that no other variables will appear in them.

We want to define a predicate, `derivative`, that will take in, e.g., `*(s(s(0)), x)` (or other polynomials expressed in this fashion) and will return the derivative of the original expression w.r.t. `x`, i.e. `s(s(0))`.

Using this will be cumbersome because our chosen representation of polynomials is hard to read. It would be moderately easier to read if our function symbols, such as `*`, could be infix instead of prefix. In fact, this is possible. Instead of writing:

```
*(s(s(0)), x)
```

we can write

```
s(s(0)) * x
```

(Our version of Prolog already allows us to do this with the function symbols `+, -, *, /` and `^`. In general, there is a way of telling Prolog when you want to use a function symbol in infix form using the built in operator `op`. Luckily we can ignore this here.)

We stress again, just in case the infix use of `*` might now make you think otherwise, the function is still *not* evaluated.

Here is a polynomial with the Prolog term that corresponds to it, making use of the infix operators:

$x^2 - 3x + 2$ is represented as
`x^s(s(0))-s(s(s(0)))*x+s(s(0))`

Now we can write our program.

The predicate we define is `derivative(Expression, DifferentiatedExpression)`, whose intended meaning is that `DifferentiatedExpression` is the derivative of `Expression` w.r.t. `x`.

We proceed by writing one clause per case (hence reflecting the recursive structure of the polynomial). The clauses that deal with cases where a term uses binary functions such as `^, *, +` or `-` will be recursive, so that the arguments to those functions also get differentiated.

Case 1:

$$\frac{d}{dx}c = 0$$

i.e. the derivative of any constant c, w.r.t. x is zero. The corresponding Prolog is:

```
derivative(0, 0).
derivative(s(N), 0).
```

Case 2:

$$\frac{d}{dx}x = 1$$

The corresponding Prolog is:

```
derivative(x, s(0)).
```

i.e. the derivative of `x` w.r.t. `x` is `s(0)` (i.e. 1).

Case 3:

$$\frac{d}{dx}x^{n+1} = (n+1)x^n$$

In Prolog this becomes:

```
derivative(x^0, 0).
derivative(x^s(N), s(N)*x^N).
```

Case 4:

$$\frac{du + dv}{dx} = \frac{du}{dx} + \frac{dv}{dx}$$

In Prolog this becomes:

```
derivative(F+G, DF+DG) :- derivative(F, DF), derivative(G, DG).
```

i.e. the derivative of a sum is the sum of the derivatives of the sum. Here we needed recursive calls to compute these derivatives.

At this stage, only very simple differentiation is possible, e.g.:

```
?- derivative(x^s(s(0)), Result).
Result = s(s(0))*x^s(0)
```

Exercise 4.12 *Expand the program by writing clauses for the following:*

(a) $\frac{d}{dx}(sinx) = cosx$
(b) $\frac{d}{dx}(cosx) = -sinx$
(c) $\frac{d}{dx}(e^x) = e^x$
(d) $\frac{d}{dx}(logx) = \frac{1}{x}$
(e) $\frac{du-dv}{dx} = \frac{du}{dx} - \frac{dv}{dx}$
(f) $\frac{d}{dx}(uv) = u\frac{dv}{dx} + v\frac{du}{dx}$
(g) $\frac{d}{dx}\frac{u}{v} = \frac{v\frac{du}{dx} - u\frac{dv}{dx}}{v^2}$

Test your resulting program. Here's an example:

```
?-derivative(s(s(s(0)))*x^s(s(0))+s(s(0))*x^s(0)+s(0)*x^0,
Result).
Result = x^s(s(0))*0+s(s(s(0)))*(s(s(0))*x^s(0))+(x*0+s(s(0))*s(0))+0
```

Note that (and here I am merely repeating the point about functions not being evaluated) the result is not simplified. The example above asked for the derivative of $3x^2 + 2x + 1$, which is simply $6x + 2$, yet the result doesn't look much like $6x + 2$. If you work through it, you will see that it 'amounts to' $6x + 2$, but the complex result is not simplified even to `s(s(s(s(s(s(0))))))*x+s(s(0))`. Why should it be? The system doesn't know that these are expressions of arithmetic.

4.4 Polymodality revisited again

The introduction of function symbols complicates a discussion of polymodality somewhat. This is something I have ignored so far in this chapter. There are now three 'states' that an argument in a call to a predicate can be in:

Uninstantiated: the variable is mapped to itself or another variable;

Fully instantiated: the variable is mapped to a term containing no variables whatever;

Partially instantiated: the variable is mapped to a non-variable term that contains variables.

For example, in the following call to `add`, the first argument is fully instantiated, the second is partially instantiated and the third is uninstantiated:

```
?- add(s(s(s(0))), s(X), Y).
```

After execution, we get the following answer:

```
Y = s(s(s(s(_9))))
X = _9
```

The variable `X` is uninstantiated; the variable `Y` is partially instantiated (since `_9` is a variable).

This complicates polymodality as we now have many more patterns of instantiation. We want to know, for example, whether `add` works when all three arguments are partially instantiated, or when the first two are fully instantiated and the third is partially instantiated, and so on. (In practice, it is *often but not always* the case that if an argument position launched an infinite computation when the argument was uninstantiated, then it will launch an infinite computation when the argument is partially instantiated (e.g. `?-add(X, Y, Z)` is infinite, and so is `?-add(s(X), s(Y), s(Z))`).

It is common to express an available mode for a Prolog predicate (i.e. one which does not launch an infinite computation) in comments in the program that give an 'adornment' or 'decoration' of the predicate argument positions with the symbols +, ++, - and ?. The meanings are + for an instantiated argument (irrespective of whether fully or partially instantiated), ++ for a fully instantiated argument, - for an uninstantiated argument, and ? for an argument that may be either instantiated or uninstantiated. For example, `add` would have the following mode declarations:

```
/* add(?, ?, ++), add(++, ?, ?)   */
```

If the first argument contains variables, the third must be fully instantiated; if the first argument contains no variables, the other two can be whatever you like. This isn't too surprising: the first argument is the one that is being recursed on; if it contains variables, the only way that the recursion will bottom out is if the other two arguments constrain the recursion in some way; if, however, the first argument contains no variables, then recursion on it will stop at some point.

Exercise 4.13 *Give the mode declarations for* `mult`*.*

4.5 Types

Much of the functional programming part of the course was probably taken up with a discussion of types. Hardly any of this part of the course will be. Prolog is not a typed language. Logic programming languages typically are not (though the new language Gödel is an exception). There is nothing that prohibits us from having types and type checking in logic programming languages (there are logics, called sorted logics, which can provided the necessary theoretical underpinning), it just seems not to have been done in practice. This, of course, does lead to more fragile programs. For example, you will not get a type error is you misuse the programs we have developed in this worksheet:

```
?- add(3, 4, X).
no

?- add(s(s(0)), 4, X).
X = s(s(4))
```

The latter of the two examples is particularly bad.

There is nothing to stop us from writing programs that explicitly check for misuses. Here's is a revised version of `add`:

```
safe_add(0, Y, Y) :- peano_no(Y).
safe_add(s(X), Y, s(Z)) :- safe_add(X, Y, Z).
```

This uses the predicate `peano_no` that I defined earlier. Since the variable `Y` in the first clause could otherwise match with anything, we protect it by making sure that what it is bound to satisfies `peano_no`. There is no need to do this in the recursive clause: it only applies if the first and third argument are made up of the successor function, which is right, and the second argument will be checked when the recursion bottoms out in the first clause:

```
?- safe_add(3, 4, X).
no

?- safe_add(s(s(0)), 4, X).
no
```

An error message would still be better than the answer *no*, but at least there is some improvement.

But being the carefree hipsters that we are, Prolog programmers never bother to add these extra type-checking conditions onto their base clauses: since the checking that they request is done at run-time, they slow program execution down.

4.6 Concluding remarks

As we shall see, the introduction of functions has given us enormous power (much more than is hinted at in this worksheet). Indeed, they give us all the data structures we could need including lists and trees. More on this in the next worksheet.

Acknowledgements

- The monkey planner is based on material in Bratko.
- The Peano axiomatisation examples and the differentiation example are based on material in Sterling & Shapiro.