Homework 1

Due May 5th

April 28, 1999

Problem 1.1. Write a definition of `times/3` so that `times(X,Y,Z)` is true for natural numbers X, Y and Z just in case $Z = X * Y$. Assume that we have already defined a predicate `plus/3` such that `plus(X,Y,Z)` is true if $Z = X + Y$. You can find a definition in the coursenotes. The definiton of `times/3` should look a lot like the definitions for `plus/3` and `exp/3` provided in the coursenotes.

You may have trouble with infinite searches in this problem. Some perfectly sensible definitions of `times/3` will contain an infinite branch in their search trees. First and foremost: do not worry about this as long as you get all solutions to your goals before Prolog goes off on an infinite search. If you want your code to be better behaved, try either (a) reordering the arguments in your call(s) to `plus/3` or (b) reordering the subgoals in the body of the recursive clause(s) of `times/3` itself.

1 Excursus: On Abstract Data Types.

Exercises (1.2) and (1.3) introduce the concept of *abstract data types* in the context of Prolog. I will briefly mention the background ideas here, but you may want to look at the problems first and then read this afterwards.

The study of data types in computer science is mainly about the proper description of data items, that is, how to represent things and concepts in such a way that a computer can manipulate them. The same problem is of course familiar from within linguistics: development of descriptively adequate grammar formalisms presupposes some way of describing linguistic objects.

Any given programming language will come with a collection of built in data types. For instance, integer and character types are built in to C and C++: compilers for C programs already "know what you mean" when you say that x is an integer. Furthermore, all programming languages come with a collection of type constructors that allow the programmer to construct more complex data types from simple ones. For example, most procedural programming languages have some built in mechanism for defining *arrays* over elementary data types. A powerful additional technique for extending a language's built in stock of data types is the definition of *abstract data types.* Instead of saying directly what the pieces of a given data structure should be, we define a set of operations which must be implemented for that type. For example, we could define a *list* abstract data type by defining what an empty list is (so we can construct brand new lists, as well as knowing when we have reached the end of an old one), and defining a function that returns the head of the list (its first element) and the tail of the list (a list containing all the elements except the first). The power of this technique comes from the fact that we can now implement any imaginable procedure over lists using these three predicates—let us refer to them as `empty_list/1`, `list_head/2` and `list_tail/2`—without having the faintest idea what concrete data structure actually underlies their implementation. As a result, whenever clever programmers discover a new and better concrete data structure for lists, we can use it in our applications simply by redefining `empty_list/1`, `list_head/2` and `list_tail/2`. All the programs which we have written using these three predicates remain the same, however. All the (potentially) thousands of lines of code we might have written using lists remain unchanged; only the few lines of code implementing those three predicates need to be changed.

For example, we could implement the abstract data type for lists using Prolog's built-in `'.'/2` operator. Then the definitions for `empty_list/1`, `list_head/2` and `list_tail/2` would look like the following:

```
empty_list(L) :- % L is the empty list if
    L = [].

list_head(H,L) :- % H is the head of L if
    L = [H|_].

list_tail(T,L) :- % T is the tail of L if
    L = [_|T].
```

To implement an abstract data type you need to do two things. First, and most importantly, you need to decide what kinds of predicates you will need in order to construct, operate on, and examine your data structures. These constitute an *interface* between your ADT and the programs that will use it. In a logic programming language like Prolog, this interface acts as a logical *definition* of the relevant structures: they can be anything at all, as long as they satisfy the constraints specified by the interface predicates. Second, you must decide on some concrete data type that you will use internal to the ADT definition. In Prolog this second step may seem somewhat trivial, since there is only one data type, technically speaking. Still, there are a lot of different expression trees that you could use to accomplish the same thing, and some there is considerable scope for cleverness and ingenuity here.

Problem 1.2. In this exercise we illustrate the basic elements of this technique as they appear in Prolog.

Suppose for example that we are building a natural language processing application and we know that the language in question contains some expressions which are nouns and some which are verbs. Suppose furthermore that we know that nouns can be marked for person, number and gender, and that verbs can be marked for tense and aspect, and also for person and number. Typically, we would expect to discover later some other features which are relevant (we might need to distinguish count nouns from mass nouns; we might need to distinguish "stage-level" predicates from "individual-level" predicates), but we would not expect to know in advance what those other features will be. For all we know, the theoreticians will invent next week distinctions that we cannot at the moment even imagine! Therefore, it is best to implement nouns and verbs as abstract data types.

Assume as a concrete underlying data type expression trees whose functors reflect the type of object they represent (so `n` for nouns, `v` for verbs), and which have argument positions for each of the features we will use (so `n`-type structures will have 3 arguments, `v`-type structures will have 4). As for which argument position is used for which feature, that is up to you.

We also consider the possible values of our various features as types. So for example we can represent the type of objects that have singular number as `sg`. The only difference between these types and the more complex types `n` and `v` is that the simple types have no arguments. That is, we have types of arity 0 (`sg`, `3`, `past`, `prog`, etc.) and also types of higher arities (`n` is a type of arity 3, `v` is a type of arity 4).

To work with nouns and verbs, then, we will at the very least need the following predicates as (a part of) our definition of the noun and verb abstract data types.

1. Definitions of our various types:
    - `n_type(DataItem)` should succeed just in case `DataItem` is a data structure of type noun (that is, a structure with functor `n` and 3 argument positions). This can be used both to recognize an input expression as being a noun (or not), and also to create new noun data items.
    - `v_type(DataItem)` should succeed just in case `DataItem` is a data structure of type verb (that is, a structure with functor `v` and 4 argument positions).
    - `person_type(DataItem)` should succeed just in case `DataItem` is a constant drawn from the set {`1`, `2`, `3`}.
    - `number_type(DataItem)` should succeed just in case `DataItem` is a constant drawn from the set {`sg`, `pl`}.
    - `gender_type(DataItem)` should succeed just in case `DataItem` is a constant drawn from the set {`f`, `m`, `n`}.
    - `tense_type(DataItem)` should succeed just in case `DataItem` is a constant drawn from the set {`past`, `pres`, `fut`}.
    - `aspect_type(DataItem)` should succeed just in case `DataItem` is a constant drawn from the set {`prog`, `perf`}.
2. Definitions of our various features:
    - `person(DataItem,P)` and `number(DataItem,N)` should succeed just in case `P` (`N`) is bound to the contents of the person field (number field) in `DataItem`. (This should succeed no matter whether `DataItem` is instantiated as a noun or verb type.)
    - `gender(DataItem,G)` should succeed just in case `G` is bound to the contents of the gender field in `DataItem`. (Obviously, this should only be defined for noun type data.)
    - `tense(DataItem,T)` and `aspect(DataItem,A)` should succeed just in case `T` (`A`) is bound to the contents of the tense (aspect) field in `DataItem`. (Obviously, this should only work for verb type data.)

Implement all the above listed predicates. If your implementation is correct, you should be able to issue the goal

```
?- n_type( X ).
```

and, by repeatedly pressing the ';' key, eventually enumerate all of the possible noun types. Similarly for verbs. You should also be able to create instances of noun types by giving Prolog goals like the following.

```
?- v_type(V),person(V,3),number(V,sg),tense(V,past),aspect(V,perf).
```

Problem 1.3. A *stack* is a data structure allowing only "first-in-last-out" access to data items. The classic model of a stack is the stack of trays at a cafeteria: the customer removes the top tray, the staff places newly washed trays back on top of the stack. The bottom tray is never used unless all the other trays have been used and none have been washed.

Abstractly, a stack is any data structure which supports the following operations (given here as predicates).

- `empty_stack(X)`—true whenever `X` is an empty stack.
- `push(Item,StackIn,StackOut)`—true whenever `StackOut` is the result of pushing `Item` onto the top of `StackIn`. (The kitchen staff has just washed a tray.)
- `pop(Item,StackIn,StackOut)`—true whenever `StackOut` is the result of popping `Item` off of the top of `StackIn`. (A customer has just taken a tray and gone to get some food.) If the stack is empty, your program should simply fail. (As an extra exercise, if you're feeling bored and restless after finishing all this other stuff, you might consider how to print out a helpful message saying why the program failed. The Prolog builtins `write/1` and `fail/0` might be useful there.)

Implement `empty_stack/1`, `push/3` and `pop/3` using lists as the concrete underlying data structure. Test your implementation with goals like:

```
?- empty_stack(S0), push(a,S0,S1), push(b,S1,S2), pop(b,S2,S3).
```

This should succeed, if you have implemented your predicates correctly, and leave you with `S3 = [a]`. The following goal, on the other hand, should fail.

```
?- empty_stack(S0), push(a,S0,S1), push(b,S1,S2), pop(a,S2,S3).
```

Make sure you save your stack code someplace safe: you'll need it again later!