# Homework 2

Due May 26th

May 12, 1999

**Problem 2.1.** Define a relation `cross_product/3` such that, given three lists `LeftCoordinates`, `RightCoordinates` and `Pairs`, the goal

```
?- cross_product( LeftCoordinates, RightCoordinates, Pairs ).
```

succeeds just in case `Pairs` is the complete list of pairs whose first element comes from `LeftCoordinates` and whose right coordinate comes from `RightCoordinates`.

Pairs can be represented any number of ways in Prolog, but very commonly the infix operator '`-`' is used, and there are several builtins and SICStus library predicates that expect as inputs lists of pairs in this format. So, for example, your solution should succeed on goals like the following.

```
?- cross_product([a,b],[q1,q2],[a-q1,a-q2,b-q1,b-q2]).
```

**Problem 2.2.** In exercise (1) you constructed a list of pairs. In this exercise we go in the opposite direction, and assume that we are given such a list of pairs to start with. Given such a list, one common problem is the *indexing problem*. The idea is that, given a data value `A` we want to find the list of all data values `B` such that `A-B` is a member of the input list. That is, given an expression `Key` and a list of pairs `Pairs`, the goal

```
?- all_values( Key, Pairs, Values ).
```

should succeed binding `Values` to a list of all the values `V` such that `Key-V` is a member of `Pairs`.

Note that the following naive implementation will not work.

```
all_values( Key, [], [] ).
all_values( Key, [Key-Val|Rest], [Val|MoreVals] ):-
    all_values( Key, Rest, MoreValues ).
all_values( Key, [Other-_|Rest], Vals ):-
    all_values( Key, Rest, Vals ).
```

The puzzle is still missing a piece, which you will have to find.

Assume for the purposes of this assignment that the keys are just single (lower-case) characters.

**Problem 2.3.** A *ranked alphabet* or *signature* is a set of symbols in which each symbol is associated with an *arity*. A convenient way to formalize this idea is to imagine that we have a set of sets of symbols $\Sigma_0, \Sigma_1, \ldots$, such that each $\Sigma_i$ contains all and only the symbols of arity $i$. So, for example, $\Sigma_0$ would be the set of all constant symbols (symbols of arity 0). Then the alphabet $\Sigma$ is conveniently defined as

$$\Sigma = \bigcup_{i<\omega} \Sigma_i$$

and saying "$f$ is of arity $k$" is synonymous with saying $f \in \Sigma_k$.

Given a ranked alphabet $\Sigma$, we can define the set of $\Sigma$-*terms* $T_\Sigma$ as the smallest set such that

1. $\Sigma_0 \subseteq T_\Sigma$: $T_\Sigma$ contains all the constant symbols, that is, a constant counts as a term.
2. If $t_1, \ldots, t_k \in T_\Sigma$ and $t \in \Sigma_k$, then $t(t_1, \ldots, t_k) \in T_\Sigma$. That is, if $t$ is a symbol of arity $k$, then we can make a new term out of $k$ old terms by applying $t$ to these $k$ arguments. The terms $t_1, \ldots, t_k$ are called (immediate) *subterms* of $t(t_1, \ldots, t_k)$.
3. Nothing else is in $T_\Sigma$.

This is a relatively precise way of defining a new sort of *language*. Now, instead of being a set of strings, our language is a set of *trees*, namely, those trees whose terminals are all constants and whose nonterminals all satisfy the arity constraint: they all have just as many children as their arities say they should have.

It is easy to define a ranked alphabet in Prolog via a set of unit clauses for a relation `sigma/2` that relates symbols to their arities. (Alternatively, instead of using unit clauses you can use clauses that call `member/2` in the way we are familiar with from studying the inequality relation `neq/2`.) So, you should be able to define the tree-language defined by this signature without too much trouble. That is, you are to define `sigma_term/1` such that it succeeds if and only if its argument is a term over the ranked alphabet given by `sigma/2`.

For the purposes of this assignment, assume the following definition of `sigma/2`.

```
sigma( t, 0 ).   sigma( f, 0 ).
sigma( not, 1 ).
sigma( and, 2 ).   sigma( or, 2 ).   sigma( then, 2 ).
```

(Feel free to extend this alphabet in any way you choose.) Obviously, $T_\Sigma$ in this case is the set of constant boolean expressions.

**Problem 2.4.** In practice we usually want a more general notion than $T_\Sigma$. We grant ourselves a set of variables $X$, which is generally assumed to be *countably infinite*, meaning there are as many members of $X$ as there are natural numbers. In practice we just need to be sure that there are "enough" variables, meaning that we can always get a brand new one whenever we need it, and that we can always effectively recognize a variable when we see one. Then we extend our notion of "term" to the set $T_\Sigma(X)$ of terms containing variables drawn from this fixed set $X$.

In first order languages, variables are always nullary, so we do not bother to assign them arities. Note that in higher order languages we may have variables of arbitrary arity. Assume from now on the following definition of $X$:

```
variable( v(_) ).
```

In practice you should probably stick to integers as subscripts for your variables, that is, use terms of the form `v(0)`, `v(1)`, etc.

The set denoted $T_\Sigma(X)$ is defined by simply throwing into the definition of $T_\Sigma$ the extra clause: "Variables are terms, too." So now our (extended) sigma terms can have variables as terminals as well as constant symbols. Once again, since we are sticking to a first order language, variables will never appear as non-terminals.

Given this extended definition of a $\Sigma$-term, you are to define the relation `substitute/4` as follows. Given a variable `Var` and three terms `PutMe`, `InMe`, `GetMe`, the goal

```
?- substitute( Var, PutMe, InMe, GetMe ).
```

should succeed just in case `GetMe` is the result of substituting `PutMe` for all occurrences of `Var` in `InMe`. Note that to do this you will need to define an inequality relation for variables. (Otherwise you will get solutions in which only some instances of `Var` in `InMe` get replaced.) You can use the Prolog builtin `'=\='/2` for this: it is only defined for numbers, but as long as you are using numbers for your variables' "subscripts" then you should be fine.

To test this, you should make sure you have more than one occurrence of the variable you are substituting for, and at least one other variable. Also be sure to press ';' until you run out of solutions. You should only get one solution.