Concepts of programming languages: Prolog exercise

October 18, 2007

1 Prolog implementation

For this exercise, we use the SWI-Prolog [1] implementation of the Prolog language.

The Prolog interpreter has been compiled for Linux/x86 systems, and is available as `/home/csalp/bin.linux/prolog/pl`. Versions for other systems are avaible; see the webpage.

You can run the interpreter using the following commands:

`/home/csalp/bin.linux/prolog/pl`	run the Prolog interpreter
`/home/csalp/bin.linux/prolog/pl -f myprog.pl`	...with `myprog.pl` being the name of a Prolog program

NOTE:

When using the first version, be sure to type `consult('myprog.pl').` at the beginning and every time you change the program.

Once you have started the interpreter, you can test your program or edit it.
You can check Prolog out by typing a goal:

```
?- voegsamen([1,2,3,4,5], [6,7], X).
```

Presuming `voegsamen` is defined, the interpreter will produce:

```
X = [1,2,3,4,5,6,7]
```

Now there are two possibilities: you can enter a ';' to let the interpreter explore alternative values for X. If none are found, it will produce 'No'. By just pressing 'enter' the interpreter will stop searching for alternative values for X. The interpreter will produce a 'Yes' and you can enter a new command.

If you want to edit in `vi` you can enter:

```
?- edit.
```

This option will only work if you run Prolog using the '-f' option. Otherwise you'll have to enter:

```
?- edit('myprog.pl').
```

NOTE:

- Be sure to use capital letters when using variables: use 'Name' instead of 'name'.
- When using functions, DO NOT put a space between the function name and the first bracket: `voegsamen(...)` instead of `voegsamen (...)`.
- Be sure to put a '.' at the end of every command, otherwise they won't be executed.
- To use the implication symbol ($\leftarrow$) in SWI-Prolog, you have to use '`:-`':

  $bird(X) \leftarrow lays\_eggs(X) \wedge has\_wings(X)$ **becomes**

  `bird(X) :- lays_eggs(X), has_wings(X).`

You can exit SWI-Prolog bij entering CTRL-C followed by an 'e'.

2 Exercises

Exercise 1: Finite Directed Graphs

We shall consider three problems on finite directed graphs. To this end, first some preliminary notions on graphs are given. Recall that a *directed graph* G is a pair (N, A), where N is a set and A is a binary relation such that $A \subseteq N^2$. The elements of N are called the *nodes* of G and the pairs which belong to A are called the *arcs* of G. If the set N is finite, the directed graph is *finite*. A graph is called acyclic if it contains no cycles. DAG stands for "Directed Acyclic Graph".

Prolog does not have any built in facilities that deal with graphs. We represent here a finite directed graph (in short: a *graph*) by a (ground) list of its arcs, where an arc from node a to node b is represented by the list $[a, b]$. In this representation the isolated nodes of the graph are omitted. However, we consider here only problems dealing with paths in graphs, and consequently such a (mis)representation is adequate for our purposes.

A *path in a graph g from a to b* is the sequence $a_1, \ldots, a_n$ $(n > 1)$ such that
$-(a_i, a_{i+1}) \in g$ for $i \in \{1, 2, \ldots, n-1\}$,
$-a_1 = a$,
$-a_n = b$.
An *acyclic path* is a path consisting of distinct nodes.

1. We begin with the problem of computing the transitive closure of a DAG. The transitive closure of a DAG is obtained by adding all arcs to the graph that are combinations of other arcs, for example if [A,B], [B,C], [C,D] are in the list then also [A,D]. Implement the following predicate by means of a Prolog program.

   `trans_dag(X, Y, Graph)` `:-` the pair [X,Y] is in the transitive closure of the DAG Graph.

2. Next, consider the general case, and implement the following predicate by means of a Prolog program.

   `trans(X, Y, Graph)` `:-` the pair [X,Y] is in the transitive closure of Graph.

   Hint: Define a predicate `trans(X, Y, Graph, Avoids)` that uses the argument Avoids to collect the list of elements that should be avoided when searching a path from X to Y.

3. Finally, consider the problem of generating, for each pair of nodes belonging to the transitive closure, a path which connects them. In general, it cannot be claimed that this path will always be acyclic, because pairs of the form $[a, a]$ can belong to the graph. However, for each pair of nodes we can always find a connecting path $a_1, \ldots, a_n$ $(n > 1)$, whose *tail* $a_2, \ldots, a_n$ is acyclic. (For $n = 2$ we stipulate here that a sequence of one element is acyclic.) Call such a path *semi-acyclic*. Implement the following predicate by means of a Prolog program.

   `path(X, Y, Graph, Path)` `:-` Path is a semi-acyclic path which connects X and Y in the graph Graph

   Hint: a program that solves the above problem can be obtained by a slight modification of the relation `trans(X, Y, Graph, Z)` obtained by adding an argument to it (i.e., `trans(X, Y, Graph, Z, Path)`) that is used to incrementally construct a path.

Exercise 2: Binary Search Trees

We shall consider binary trees whose nodes are (labelled with) natural numbers, and use the term *void* to denote the empty tree, and the term *tree(x, left, right).* to denote the tree with root x, left subtree *left* and right subtree *right*. For example, the term *tree(1, tree(2, void, void), tree(3, void, void))* represents the tree with root *1* and children *2* and *3*.

We call a binary tree *tree(x, left, right) nice* if:

1. if *left* is not empty then x is greater than all the elements in *left*;

2. if *right* is not empty then x is less than all the elements in *right*.

A binary tree is called a *search tree* if every subtree of it is nice. Write a program which tests whether a ground term is a search tree.

Hint: Use the following predicate `is_search_tree(T)` in the definition of search tree:

```
is_search_tree(void).
is_search_tree(T)      :-  is_search_tree(T, Min, Max).
```

where `Min` and `Max` are the minimum and maximum element of the tree `T`. Then implement the predicate `is_search_tree(T, Min, Max)`.

3 How to submit

Your programs should be submitted together with a written report in which you explain your programs, to Thijs van Ommen (mvommen@liacs.nl).

References

[1] `http://www.swi-prolog.org/`