Logic Programming | Alan Smaill
21/10/15

Tutorial for week 6 (26-30 Oct)

Definite clause grammars, Generate and Test

1. **DCG example** (This example is from LPN.)

   Let $a^n b^{2n}$ be the formal language which contains all strings of the following form: an unbroken block of letters a of length n followed by an unbroken block of bs of length $2n$, and nothing else. For example, abb , aabbbb, and aaabbbbbb belong to $a^n b^{2n}$, and so does the empty string. Write a DCG that generates this language.

2. **DCGs with parameters**
   Recall that the Kleene star expression a^* denotes the set of all strings of the form a^n, where $n \geq 0$. In Prolog, we can use lists to model sequences, interpreting such a regular language as: $\{ [\,], [a], [a, a], [a, a, a], \dots \}$

   (a) Write a DCG (using one or two rules) that defines a nonterminal `astar` that accepts the regular language a^*. (Be careful to avoid left-recursion).

   (b) Write a DCG that defines a parametrised nonterminal `star(X)` such that, for any atom a, $star(a)$ accepts the regular language a^*.

3. **Parsing expressions** Consider the following simple expression language:

   - A number $n = 1, 2, 3, \dots$ is an expression
   - If e_1 and e_2 are expressions then so is $e_1 + e_2$
   - If e_1 and e_2 are expressions then so is $e_1 - e_2$
   - If e_1 and e_2 are expressions then so is $e_1 * e_2$
   - If e_1 and e_2 are expressions then so is e_1/e_2
   - If e is an expression then so is (e)

   The input to the parser is provided as a list of tokens. A token is either a number or an atom of the form:

   `’+’ ’-’ ’/’ ’*’ ’(’ ’)’`

   The predicate `token/1` recognises tokens:

```
token(X) :- number(X).
token('+'). token('-'). token('*').
token('/'). token('('). token(')').
```

   (a) (*) Write a grammar defining nonterminal exp that correctly parses fully-parenthesised expressions.

(b) (*) Building on the expression parser in the previous question, parametrise the nonterminals in the grammar with a number `V` that is the value of the expression (evaluated using `is`).
Thus, evaluating `exp(X,['(',2,'+',2,')'],[])` should yield `X=4`.

4. **Generate and Test**

Recall that it is not safe to use Prolog negation as failure where the query when called is not ground. Here you are asked to provide a generate and test way round this, where candidate ground solutions are generated so that negation can be safely used as part of the test.

Alice, Bob, Charlie, David, and Eve live on a row of houses on the same street, and each have different jobs: one doctor, one teacher, one dentist, one lawyer, and one firefighter. The following facts declaratively describe them:

```
male(bob).                  male(charlie).
male(david).
female(alice).              female(eve).
neighbour(alice, bob).      neighbour(bob, charlie).
neighbour(charlie, david). neighbour(david, eve).
```

The following additional constraints hold:

- Bob is not the doctor.
- The teacher and firefighter are both male.
- The firefighter and lawyer are neighbours.
- The dentist has a female neighbour.

(a) Being neighbours is symmetric, but the `neighbour/2` predicate is not. Write a predicate `neighbour_sym/2` such that `neighbour_sym(A,B)` holds if `A` is a neighbour of `B` or vice versa.

(b) An *assignment* of jobs to people is a 5-tuple `(Doc,Tea,Den,Law,Fir)` where each element is a different person's name, and `Doc` is the name of the doctor, `Tea` is the name of the teacher, etc. Define a predicate `test/5` such that `test(Doc,Tea,Den,Law,Fir)` succeeds when given a ground assignment that satisfies the constraints.

(c) Define a generator predicate `generate/5` that succeeds repeatedly by generating all possible assignments. (**Hint:** It may be easier to do this by working with lists.)

(d) Write a predicate `solve/5` that succeeds by finding solutions to the problem. The `solve/5` predicate should succeed repeatedly to find **all** solutions (there are more than one).