COSC 2P93 Prolog Assignment #5

Due date: Friday April 5, 12:00 noon
Lates: accepted until 12:00 Monday April 8, -25%; not accepted afterwards.
Comments: See Assignment 1 comments for program requirements.
Hand in: Printouts of your programs and adequate examples of their execution. Electronic submission of source code and files.

1. Mathematical expression simplifier.

The 2P93 example directory has an elegant little program that performs symbolic differentiation (from *Programming in Prolog 4e*, W.Clocksin and C.Mellish, Springer, 2003):

http://www.cosc.brocku.ca/Offerings/2P93/examples/derivative

An example execution is as follows:

?- d(2*x - 3*(x^2)/log(2*x), x, A).

A = 2*1-(-1*log(2*x)^(-1-1)*(2*1*(2*x)^ -1)*(3*x^2)+3*(2*x^(2-1)*1)*log(2*x)^ -1)

If you examine the output expression, there is a lot of opportunity for simplification. For example, the following replacements can be done on terms within it:

2*1 → 2
-1-1 → -2
-1*log(...) → -log(...)
(2*x^(**2-1**)*1) → (2*x^**1*1**) → (2***x^1**) → 2*x

In the last line above, the terms that are simplified are in boldface. The entire output might be simplified to:

A = 2-(-log(2*x)^(-2)*(4*x)^ -1)*(3*x^2)+(6*x*log(2*x)^ -1)

Note the following:

i. When certain patterns are found in expressions (left of arrow), they can be replaced with simplified forms (right of arrow). Pattern matching and simplification can be implemented in almost **the same way** as the derivative rules in the calculus program.
ii. The application of simplifications is iterative (recursive). A simplification can result in a new term, which can then be re-simplified. This can carry on until no more simplifications arise.

Write a simplifier program, that takes an output expression from the derivative program, and generates a simplified version of it. You should think of as many "simplification rules" as you can, that have general forms such as the following:

X – X → 0
X / 1 → X
X ^ 0 → 1

Your program should be recursive and/or iterative, and reapply simplification to expressions repeatedly. It should stop when all simplifications have been applied. Don't worry about making the *simplest* expression possible, because that is a difficult search problem!

2. Random Prose Generator

The Policeman's Beard is Half Constructed is a book that came out in the early 1980's. It contains prose generated entirely with computer software named Racter. Two excerpts:

> *Slide and tumble and fall among*
> *The dead. Here and there*
> *Will be found a utensil.*
>
> *Tomatoes from England and lettuce from Canada are eaten by*
> *cosmologists from Russia.*

(You get the idea.) You are to write a random prose generator in Prolog. To do this, you will use a DCG for a small subset of English. To get started, there are some DCG grammars in the 2P93 example folder, and the Bratko text. (You don't need to create a parse tree or "logical" meaning with your grammar, however, so those parts can be removed). You will implement a grammar that will be used to generate random sentences (vs. parsing sentences). You should do the following:

a) Extend the grammar to include some interesting English grammar rules. You should consider adjectives, adverbs, and conjunctions (and, or). Feel free to include anything else you can think of... (eg. questions might be interesting to include!). Impress the marker for full marks!
b) Use a variety of words for your grammar. However, rather than have 50 rules for "noun" (one per word), use one rule for singular nouns, and put all your nouns in a list. You can then write something that randomly selects a noun from this list.
c) Make sure your grammar respects singular/plural constructions (see example online).
d) Some rules may be recursive. You should make sure that infinite recursion won't happen. This can be stopped by including a depth value, that will terminate a rule if the depth exceeds a certain limit. (The following item might also help...)
e) You might have 2 rules for a grammatical part of speech. You can randomly select one rule to use as follows:

```
noun_phrase --> {maybe(0.33)}, < 1st noun phrase rule..>.
noun_phrase --> {maybe(0.5)}, < 2nd noun phrase rule...>.
noun_phrase --> < 3rd noun phrase rule...>
```

maybe(P) is a builtin predicate in the random library module, that will succeed P% of the time. Therefore, each rule above will be selected roughly 33% of the time.

f) Put a random seed *setrand/1* call at the beginning of your program. The user can supply a seed, to start a new random number sequence.
g) Finally, include a high-level predicate that will generate 3 sentences of random prose. It will call your grammar 3 times to make these sentences. Each sentence will be written to the screen in a proper format. For example, the list [a,cat,hits,the,senator] will be written:

A cat hits the senator.

3. CLP and number puzzles: Re-do question 2 from assignment 3 ("Symbo-logical") using constraint logic programming.