Logic Programming
Tutorial 6: Definite clause grammars

Week 8 (Nov 3–7)

1. **DCG translation** Translate the following DCG grammar rules into ordinary definite clauses using difference lists:

   (a) `s --> t, u(a), v.`

   (b) `t --> [a], u(c), [b].`

   (c) `u(X) --> []; ([X], [X]).`

   (d) `v --> [a]; ([a],v).`

   Then, try to optimize the resulting clauses to perform as much pattern-matching as possible in the head of the clause, rather than via explicit unification subgoals.

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:

   $$\{[], [a], [a, a], [a, a, a], \ldots\}$$

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

   (b) Write a DCG that defines a parameterized 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 $n = 1, 2, 3, \ldots$ 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 and n is a number then so is $e\hat{}n$
   - 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` recognizes tokens:

```
token(X) :- number(X).
token(X) :- member(X,['+','-','*','/','^','(',')']).
```

(a) (*) Write a grammar defining nonterminal `exp` that correctly parses fully-parenthesized expressions. (That is, expressions like `(1+2)*3` or `1+(2*3)` where enough parentheses have been added to eliminate any ambiguity.)

(b) (*) Building on the expression parser in the previous question, parameterize 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`.

(c) (**) In the absence of parentheses we want to treat multiplication and division as having higher precedence than addition and subtraction. For example, we want to treat the token sequence `[1,'+',2,'*',3]` the same as `['(',1,'+','(',2,'*',3,')',')']`, and both should evaluate (only) to 7. Modify the DCG from the previous part to handle expressions with parentheses omitted according to the usual precedence rules. Each valid token sequence should evaluate to a unique value. Avoid left-recursion.