FP Project:
An Evaluator in Haskell for a Logic Programming Language

In this project you have to write in Haskell evaluators for logic programming languages of various levels of complexity:

- The *propositional* case, i.e., every clause is (a combination of) elemntary propositions (that is: predicates with zero variables).
- The case in which every clause is (a combination of) predicates with *one* variable.
- The general case, i.e., a predicate may have *any number* of variables.

For each case, the evaluator should be written as a Haskell function *eval* which gets as arguments a logic program (i.e., a list of clauses) and a query. The result of the *eval* function should be either *True* or *False*, or, in case the query contains one or more variables, a sequence of possible values for the variable(s) that make the query *true*.

The emphasis of this project is on the reasoning process in a logic programming language, i.e., you may assume that there are no side effects. Hence, you may ignore I/O. Also, arithmetic and lists do not have to be taken into account, i.e., the only terms you have to consider are *constants* and *variables*. Furthermore, no optimizations have to be investigated, so you may ignore the "!" from Prolog.

Deadline: Monday, June 12, 23:59. Submit your code on Blackboard under *FP Project group enroll*, together with a short explanation of your program.

Level 1 – Propositional Case – 3 points

In the propositional case an *atom* consists of a proposition letter only, i.e., atoms do not contain variables. Hence, unification and subsitutions are not yet required in this part of the project. For example, the following is a small logic program in such a restricted language:

$$
\begin{array}{lll}
a_0. & & \\
a_1. & & \\
a_2. & & \\
b_0 & :- & a_0,\ a_1. \\
b_1 & :- & a_1,\ a_2. \\
b_2 & :- & a_1,\ a_2,\ d. \\
c_0 & :- & b_0,\ b_1. \\
c_1 & :- & b_0,\ b_1,\ b_2.
\end{array}
$$

According to this program, the query c_0? should give the answer *True*, whereas the query c_1? should give *False* (because of the *closed world assumption*).

A possible starting point to program an evaluator in Haskell for Prolog programs like this might be given by data types like the following:

$$\begin{array}{lcl} \textbf{data}\ \mathit{Atom} & = & A_0 \mid A_1 \mid A_2 \mid B_0 \mid B_1 \mid B_2 \mid C_0 \mid C_1 \mid D \\ & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}) \\ \textbf{type}\ \mathit{Clause} & = & (\mathit{Atom}, [\mathit{Atom}]) \\ \textbf{type}\ \mathit{Program} & = & [\mathit{Clause}] \\ \textbf{type}\ \mathit{Query} & = & [\mathit{Atom}] \end{array}$$

Elementary atoms are indicated by constructors in the algebraic type *Atom*. Note that constructors stand for *constants*, and start with a capital letter, which is different from the convention in Prolog.

The type *Clause* defines 2-tuples of the form (p, ps), representing a clause in Prolog-syntax:

$$p \ :\!-\ ps.$$

Note that a *fact* in Prolog corresponds to $ps = []$.

Even though an initial query is a single atom, during the evaluation process more atoms can be added. Hence, in general a query should be a list of atoms, as expressed by the type *Query*.

Task: Write a function *evalProp* which evaluates a query for the propositional case to *True* or *False*.

Hint: Use list comprehension to "walk" through all clauses in a program.

Level 2 – One-variable case – 5 points

In this case every atom consists of a *predicate* with *one* variable (or constant). The following program is an example of this case:

$$\begin{array}{l} p(a). \\ p(b). \\ p(c). \\ \\ q(a). \\ q(b). \\ \\ r(X) \ :\!-\ p(X), q(X). \end{array}$$

Note that an atom now consists of a *predicate* together with a *variable* or a *constant* Now the queries $r(a)$? and $r(b)$? should yield *True*, whereas the query $r(c)$? should give *False*. The query $r(X)$? contains the variable X and should yield all values for X for which $r(X)$ is true. In this case, these values are a and b, whereas the value c is not correct.

In the above program, there is no immediate match for, e.g., $r(a)$, only after *substituting* a for X in the last clause we get the clause:

$$r(a) \ :- \ p(a), q(a).$$

against which $r(a)$ can be matched. We remark that substitution may be done for variables *only*.

A possible type definition useful for substitution might be

$$\textbf{type} \ Substitution \ = \ (Term, Term)$$

where a *Term* may be a constant or a variable (you'll have to define the type *Term* yourself).

Tasks:

- Extend or adapt the types given above such that atoms may consist of a predicate together with a single constant or variable.
- Define an operation $\Leftarrow$ for substitution (with (x, a) of type $Subsitution$, i.e., both x and a should be of type *Term*):

  $$e \Leftarrow (x, a)$$

  in which e can be an expression of various kinds (see below), x a variable, and a a constant or a variable. The result should be that a is substituted for x in e. For example, the result of (X is a variable)

  $$X \Leftarrow (X, a)$$

  should be a, and the result of (X, Y are variables)

  $$Y \Leftarrow (X, a)$$

  should be Y. Remember that in Haskell you may define $\Leftarrow$ directly as an infix operation.

  In your definition of the substitution operation ($\Leftarrow$) it should be possible that e is a *term* (i.e., a variable or a constant), an *atom*, or a *clause*. Hence, it is practical to define a *type class* such that this operation can be *instantiated* for for all these things.
- Since variables in a clause should be kept seperate from the variables in the query, you should first write a function *rename* to replace the variables in the clause by new variables. Hence, you will have to generate *new* variables and substitute these for the variables in a clause.

- Write a function *unify* that finds the right substitution (in the form of a tuple (x, a)) which can make two atoms equivalent by applying that substitution. For example, the two atoms $p(a)$ and $p(X)$ (with a a constant and X a variable) can be *unified* by the substitution (X, a):

  $$\begin{array}{lll} p(a) \Leftarrow (X, a) & = p(a) & \text{substitution for } a \text{ leaves } a \text{ unchanged} \\ p(X) \Leftarrow (X, a) & = p(a) & \text{substituting } a \text{ for } X \text{ does replace } X \text{ with } a \end{array}$$

  Note that neither $p(a)$ and $q(X)$, nor $p(a)$ and $p(b)$ can be unified.

- write a function *evalOne* that evaluates a query for a program in which clauses may consist of a predicate together with one variable or constant.

  Your function should work in case that the query contains a constant (yielding *True* or *False*), and also in the case that it contains a variable (yielding the substitutions for which the query becomes true).

Remark: again, the Prolog convention to let constants begin with a lowercase letter and a variable with a capital letter, does not need to bother us. Or, more precisely, in the above X, Y are *specific* variables, and x stands for *any* variable (i.e., x is a "meta variable").

Level 3 – Multi-variable case – 2 points

This case is a generalization of the one-variable case, i.e., a predicate may have any number of variables. The example program on the royal family is an example of this case.

Tasks: The extra difficulty in this case is that an atom may contain the same variable more than once, such that substitution and unification become more tricky: in order to unify $p(X, X)$ and $p(a, Y)$ *both* X and Y have to be replaced by a.

Write a function *evalMulti* for this case. As before, your program should work for queries containing constants and/or variables.