CSE 341, Spring 2004, Assignment 3
Due: Monday 26 April, 9:00AM

Last updated: 14 April

You will write several functions over propositional logic formulas, using these definitions:

```
type var = string
type truthtable = (var * bool) list
datatype formula = Constant of bool
                 | Var of var
                 | Not of formula
                 | And of formula * formula
                 | Or of formula * formula
                 | Implies of formula * formula
exception UnboundVar of var
```

Some hints describe the approximate length of the sample solution. Consider this length a "rough guide", not an exact requirement. We assume some knowledge of logic; just ask if you have questions.

We are using ML strings to represent logic variables. You can compare two strings with `=`.

1. Define a function `evaluate` that takes a truth-table and a formula, and evaluates to `true` if and only if the formula is true given the truth-table. If a formula includes a variable v not in the table, raise the exception `UnboundVar` v. As usual, `Implies(`*f1*`,`*f2*`)` is true unless *f1* is true and *f2* is false. Hint: Use another function to lookup variables in a truth-table. Hint: Sample solution is 15 lines.

2. Define a function `demorgan` that takes a formula and evaluates to an *equivalent* formula (a formula that evaluates to the same result as the input for all truth-tables).

   The *result* must satisfy these requirements:

   - Not use `Implies`
   - Only use `Not` on subformulas of the form `Var` v

   The *implementation* must satisfy these requirements:

   - Use only two mutually recursive helper functions, called `pos` and `neg`, both of type `formula->formula`
   - `pos` has the same behavior as `demorgan`. `neg` is like `demorgan` except it produces a formula equivalent to the *negation* of its input.
   - Any formula you build must be part of the result of `demorgan`.

   Hints: $\neg(f_1 \wedge f_2)$ is equivalent to $(\neg f_1) \vee (\neg f_2)$ and $\neg(f_1 \vee f_2)$ is equivalent to $(\neg f_1) \wedge (\neg f_2)$, but you use these facts indirectly because $\neg f_1$ and $\neg f_2$ may not be appropriate subformulas for the output. Hint: Sample solution is 18 lines.

3. Define a function `fold_vars` of type `'a * ('a * var -> 'a) * formula -> 'a`. Intuitively, the first argument is an accumulator and the second argument is a function applied to every `var` occurring in the third argument. Your function must be suitable for use in the next 3 problems. Hint: Sample solution is 11 lines.

4. Define a function `all_vars` that takes a formula and evaluates to a list containing all the variables in a formula. The order the variables appear in the list is irrelevant. If a variable appears n times in a formula, it should appear n times in the list. Your solution must use `fold_vars` and otherwise be nonrecursive. Hint: Sample solution is 1 line.

5. Define a function `has_var` that takes a formula and a variable and returns true if and only if the variable occurs in the formula. Your solution must use `fold_vars` and otherwise be nonrecursive. It should not use `all_vars`. Hint: Sample solution is 1 line.

6. Define a function `has_repeat` that takes a formula and returns true if and only if there exists a variable that occurs more than once in the formula. The sample solution finds this definition useful:

   ```
   datatype seen = Repeat | Sofar of var list
   ```

   Your solution must use `fold_vars` and the library function `List.exists` and otherwise be nonrecursive. Hints: `Sofar` is pronounced "so far" and indicates there are no repeats yet. Sample solution is 11 lines.

7. **Extra Credit** Define a function `smaller` that takes a formula and evaluates to an equivalent formula. The result must never have more occurrences of variables (subformulas of the form `Var` v) than the input. The output must never have any constants unless the *entire* output is a constant. The more *tautologies* (always true statements) and *contradictions* (always false statements) you find, the smaller your result may be. Warnings:

   - The sample solution did not do the extra credit.
   - If you find a solution that always produces the smallest possible result and always runs in time less than $O(2^n)$ where n is the size of the input, you will win the Turing Award, which is roughly the "Nobel Prize of Computer Science".

Type Summary: Evaluating a correct homework solution should include these bindings (`datatype seen` is not strictly required):

```
val evaluate = fn : (var * bool) list * formula -> bool
val demorgan = fn : formula -> formula
val fold_vars = fn : 'a * ('a * var -> 'a) * formula -> 'a
val all_vars = fn : formula -> var list
val has_var = fn : formula * var -> bool
datatype seen = Repeat | Sofar of string list
val has_repeat = fn : formula -> bool
```

Also remember `pos` and `neg` must have type `formula->formula`.

Turn-in Instructions

- Put all your solutions in one file, `lastname_hw3.sml`, where `lastname` is replaced with your last name.
- Line 1 of your `.sml` file should include an ML comment with your name and the phrase `homework 3`.
- Email your solution to `daverich@cs.washington.edu`.
- The subject of your email should be *exactly* `[cse341-hw3]`.
- Your `.sml` file should be an *attachment*.