PROBLEM

Boolean expressions on n variables allow us to compute the result of nested true-false operations. They consist of boolean *variables* operated on by the following three functions: NOT, OR, AND. In infix (human readable) notation, consider the following boolean expression e:

$$e = \text{NOT}(a \text{ AND } (b \text{ OR } a)) = \neg(a \wedge (b \vee a))$$

where we adopt the conventions NOT=$\neg$, OR=$\vee$, and AND=$\wedge$. In OCaml, we can define these expressions recursively with types:

```
type expression =
  |Var of int
  |Not of expression
  |And of expression * expression
  |Or of expression * expression;;
```

Hence, the expression above may be written as

```
let e = Not(And(1, Or(2, 1)))
```

We wish to do the following...

1. Convert a recursively-defined boolean expression into human readable text.
2. Extract an expression's variable names in a well-ordered fashion.
3. Evaluate an expression on fixed inputs values.
4. Compute the values of a boolean expression over all possible input combinations (i.e. make a "truth table"). Provide a readable summary of an expression using (1), (2), and the truth table.
5. Determine if a boolean expression is identically true, or if a solution for it exists, and, if so, on what inputs.
6. Determine if two boolean expressions have an implication relation, i.e. $e_1 \implies e_2$, $e_1 \impliedby e_2$, or $e_1 \iff e_2$.

SOLUTION

The third and fourth questions have been solved on OCaml (*ocaml.org* $\rightarrow$ *exercises* $\rightarrow$ *intermediate*), in both the $n = 2$ and general cases. However, these solutions are not tail-recursive and are not space efficient. We provide the following solution outlines for the problems above, which, if recursive, should be made tail recursive using continuations.

1. `printExpression`
   One should approach this by pattern matching on the expression `e`: if `e` is a variable, return it; otherwise, place the appropriate operator (i.e. $\neg, \vee, \wedge$) between (or in front) of a recursive call on the expression(s) contained in it.

2. `inputList, trInputList`
   We pattern match on an expression `e`: if `e` is a variable, check if we've seen it already (via an accumulator, say), and take note of it if not. Otherwise, recursively call on its arguments. At the end, use `List.sort` to get arguments in order.

3. `evaluateExpression, trEvaluateExpression, memoEvaluateExpression`
   Similarly, given a list of bools associated with variables, we return the appropriate bool if we run into the variable. Otherwise, we perform the given logical operation (i.e. `not`, `||`, `&&`) on the recursive call.

4. `truthTable`
   Generate a 2D list of 2^n rows consisting of all length-n true-false combinations. Ideally one uses a helper function to do this. Then, associate each element in an input combo with a variable in the expression. We hence use (3) from above to evaluate the expression on this combination. Use higher order functions to adjust which evaluator you use. `Printf.printf` provides the functionality to display the results from (1), (2), and (4).

5. `alwaysTrue, existsSolution, findSolutions`
   After generating (3) (the "truth table"), we simply analyze the solutions (are they all true/false?; if a combination evaluates to true, what was it)?

6. `satSolverImplies, satSolverImpliedBy, satSolverIff`
   For $e_1 \implies e_2$ to hold, we require exactly that e_2 be true when e_1 is true. This is encoded in the verification of $e := (\neg e_1) \vee e_2$. Generate the truth table for e, and use `alwaysTrue` to see if it always holds. $e_1 \impliedby e_2$ and $e_1 \iff e_2$ follow similarly.