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 string (* in the general case, we'll use int *)
  |Not of expression
  |And of expression * expression
  |Or of expression * expression;;
```

Hence, the expression above may be written as

```
let e = Not(And(a, Or(b, a)))
```

We wish to understand the following...

1. Can we convert a recursively-defined boolean expression into human readable text?
2. Can we compute a boolean expression given fixed values for its inputs?
3. Can we compute all possible values of a boolean expression over its inputs (a "truth table")?
4. Can we find if a boolean expression is always true? Always false? Can we find if a boolean expression has a solution (i.e. a combination of inputs such that it is true), and, if so, list it/them?
5. Can we determine if two boolean expressions have an implication relation, i.e. $e_1 \implies e_2$, $e_1 \impliedby e_2$, or $e_1 \iff e_2$? This is a simple "SAT solver."
6. Can we do all of the above on a boolean expressions of arbitrary (i.e. not 2) inputs? Can we do all of the above tail-recursively, or, if appropriate, with memoization?

SOLUTION

The second and third 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 posed above:

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. `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.

3. `truthTable`: Generate a 2D list of 2^n rows consisting of all length-n true-false combinations. We associate each index of each row with a variable in the expression. We hence use (2) from above to evaluate the expression on each row. Use higher order functions to adjust which evaluator you use.

4. `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)?

5. `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 $e := (\neg e_1) \vee e_2$. Evaluate e on all inputs, and make sure it identically true (use functions from (4)). $e_1 \impliedby e_2$ and $e_1 \iff e_2$ follow similarly.

6. Only (1) and (2) require recursion, and tail-recursive versions may be accomplished easily via continuations. Generalizing to n, we wish to have a non-arbitrary ordering of variables. Strings are not good for this, so use integers. For example:

   ```
   let e = Or(And(Not(Var(3)), Var(1)), Var(2))
   ```

   then we need to extract these inputs recursively into a sorted list (i.e. `[1;2;3]`) and create pairings between a row of bools (recall: there are 2^n of these) and this list of variables. Finally, evaluate. Memoization may be accomplished by checking and storing sub-expressions in a hash table.