PROBLEM

Boolean expressions define how a series of *operations* (NOT, OR, AND) act on set of boolean *variables*. For example, consider the following:

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

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

```
type exp =
  |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 understand when two boolean expressions imply eachother, i.e. when the statement "if e_1 then e_2" holds. Mathematicians have a particular interest in machines which can verify logical implications, called *SAT solvers*, and we will implement a simple one below by solving the following sub-problems:

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, 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, their solution is not tail-recursive or space efficient. We provide the following solution outlines for the problems above, which, if recursive, should be made tail recursive using continuations.

1. Implement `printExpression`, as described above. One should approach this by pattern matching on the expression `e`: if `e` is a variable, return it; otherwise, place the appropriate operator character (i.e. $\neg, \vee, \wedge$) between (or in front) of a recursive call on the expression(s) contained in it.
2. Implement `inputList` and `trInputList`, which take an expression `e` and returns a sorted list of integers. We pattern match: if `e` is a variable, check if we've seen it already (via an accumulator, say), and add it if not. Otherwise, recursively call on its arguments. At the end, use `List.sort` to get input names in increasing order.

3. Implement `evaluateExpression` in 3 ways: recursively, tail-recursively, and with memoization. Pattern match on the expression: if a variable, return the variable's assignment. Otherwise, perform the appropriate logical operation (i.e. `not`, `||`, `&&`) on the recursive call of its sub-expressions. Use a hash-table to accomplish memoization.
4. Implement `generateCombinations`, which generates a 2D list of 2^n rows consisting of all length-n true-false combinations. Then implement `truthTable`, which returns a list of input-output combinations. Use higher order functions to allow the user to choose which evaluator to use. `Printf.printf` provides the functionality to display the results from (1), (2), and (4).
5. Implement `alwaysTrue, existsSolution`, and `findSolutions`. After generating the truth table in (3), we analyze the solutions (are they all true/false?; if a combination evaluates to true, what was it)?
6. Implement `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.

   We look for the following types, where `<evaluator> = exp -> (int*bool) list -> bool`.

Name	Type
`printExpression (r/tr)`	`exp -> string`
`inputList (r/tr)`	`exp -> int list`
`evaluateExpression (r/tr/memo)`	`exp -> (int * bool) list -> bool`
`generateCombinations`	`int -> bool list list`
`truthTable`	`<evaluator> -> exp -> (bool list * bool) list`
`alwaysTrue, existsSolution`	`<evaluator> -> exp -> bool`
`findSolutions`	`<evaluator> -> exp -> (bool list * bool) list`
`satSolverImplies/ImpliedBy/Iff`	`<evaluator> -> exp -> exp -> bool`

Good luck ☺