Harvey Mudd College
Computer Science 65
Fall 2008

Assignment 4
Propositional Logic
Due. 11:59 p.m., Wed., 1 Oct. 2008

In this assignment, we will use the Scheme language to implement some functions relating to proposition logic, also known as switching logic or Boolean logic. The target is 100 points for this assignment, but you can get up to 75 points of extra credit on the last problem by exercising your creativity.

The basic logic functions used in this assignment are ∧ (*and*), ∨ (*or*), → (*implies*), and ¬ (*not*). For the first two functions, we mostly use the 2-ary version, except in problem 5, where the output is n-ary. We will be concerned with two forms of expressing logic functions:

a. Truth tables
b. Scheme expressions

As discussed in class, logic functions of interest are expressible as truth tables. For example, consider the function f(x, y, z) = ((x ∧ y) → z) → (y → (z ∨ x)). This function has the following truth table, as you may check for yourself:

x	**y**	**z**	**f**
0	0	0	**1**
0	0	1	**1**
0	1	0	**0**
0	1	1	**1**
1	0	0	**1**
1	0	1	**1**
1	1	0	**1**
1	1	1	**1**

The "stub" part of the truth table, i.e. the columns for variable values x, y, and z, have a standard form corresponding to the numbering of the row values in increasing binary order. Given that we use this standard by convention, only the column for f is interesting, and we can thus capture the entire table more compactly by giving the column as a list:

```
(1 1 0 1 1 1 1 1)
```

The left-to-right reading of the list corresponds to the top-to-bottom reading of the right-hand column. We adopt this convention because then the stub row, interpreted as a binary numeral, can be used as an index into the list to get the value of the function. For example, the stub row 0 1 0 is binary 2, and indexing the list by 2 (starting from 0) gives the function value 0.

1. [20 points] Construct function **truth-table-column** that returns the truth-table column for a function given as a Scheme expression. To make the truth-table representation unique, we provide the list of variables to this function as well. An example of the arguments to this function are the list of variables in order:

```
(x y z)
```

and the symbolic expression for the function's value:

```
(implies (implies (and x y) z) (implies y (or z x))))
```

Thus a test case for the current example would appear:

```
(test
  (truth-table-column
    '(x y z)
    '(implies (implies (and x y) z) (implies y (or z x)))
    )
   '(1 1 0 1 1 1 1 1))
```

You can make use the evaluator constructed in class, and available on the website. I highly recommend a recursive approach toward this problem, using the 0-variable case as a basis. (A 0-variable function must have a constant value 0 or 1.)

2. [10 points] Construct function **get-vars** that extracts the variables from a logic expression as a list of symbols. The order of variables can be arbitrary, but make sure that no variable is listed more than once.

```
(test (get-vars '(or (not x) (and z (or y x)))) '(z y x))
```

3. [10 points] Construct function **equiv?** that determines whether or not two logic expressions are equivalent. Two expressions are equivalent iff, for each substitution of an element in {0, 1} for their corresponding variables, the values of the expressions are the same.

```
(test (equiv? '(or x y)       '(or y (and (not y) x)) '#t)
(test (equiv? '(implies x y) '(implies y x))          '#f)
```

4. [20 points] Construct function **boole-shannon-expansion** that expands a logic function in the form of a truth-table column along with its list of variables, according to the Boole-Shannon expansion explained in class. Leave the result in raw, unsimplified form, as awful as that may appear. You get a chance to make it prettier in problem 6. An example is given below.

```
(test (boole-shannon-expansion '(x y) '(1 1 1 0))
      '(or
        (and (not x)
             (or (and (not y) 1)
                 (and y 1)))
        (and x
             (or (and (not y) 1)
                 (and y 0)))))
```

Let recursion do the work for you.

5. [30 points] Construct function **`minterm-expansion`** that expands a truth-table right-hand column with a list of variables according to the *minterm expansion*. The minterm expansion is the following form, where both **`or`** and **`and`** are allowed to have an *arbitrary number of arguments*:

(**`or`** (**`and`** L_1 L_2 … L_n) (**`and`** L_1' L_2' … L_n') …(**`and`** L_1" L_2" … L_n")

where each L_i is a *literal*, meaning a variable or its negation. *Every* variable in the original expression must be present in every term either as itself or negated. Example:

```
(test (minterm-expansion '(x y z) '(0 1 1 0 0 0 0 1))
      '(or
         (and (not x) (not y) z)
         (and (not x) y (not z))
         (and x y z)))
```

The minterm expansion is obtainable by "reading off" from the truth table the rows corresponding to a 1 output. However, this problem may be done recursively, splitting the truth table in half repeatedly.

6. [10 - 85 points] Construct function **`simplify`** that will simplify a logic expression. This problem is open-ended. The better the simplifications, the more points you get. Here are the **minimum requirements**:

- If an expression is equivalent to 1, it must simplify to 1.
- If an expression is equivalent to 0, it must simplify to 0.

For example:

```
(test (simplify '(or x (not x))) 1)

(test (simplify '(and x (not x))) 0)

(test (simplify '(or (and x (not y)) y)) '(or x y))
```

There are many ways to approach this problem. One suggestion is to use the Boole-Shannon expansion, coupled with judicious application of various logical identities.