Harvey Mudd College
Computer Science 65
Fall 2008

Assignment 5
Propositional Logic
Due. 11:59 p.m., Wed., 14 Oct. 2009

Note that problem 4 requires paper submission, due in class on the date above.
In this assignment, we will use the Scheme language to simplify expressions in proposition logic, also known as switching logic or "Boolean algebra". I have provided a parser for a specific logical language in:

http://www.cs.hmc.edu/courses/2009/fall/cs42/logic-parser-simplifier.scm

The grammar in that file defines both the input and output language for these problems. You are free to use a different parser of your own construction if you wish. However, the language accepted by your parser should be the same as this one. The parser converts an input expression into a syntax tree. In the example code, some very simple simplifications are performed on the tree. Your problem will entail making a system that gives better simplifications.

1. [10 points] Construct function **`get-variable`** that extract a variable from a logic syntax tree of the type produced by your parser. For example,

   ```
   (get-variable '(+ (- x) (* z (+ y x))))
   ```

   might return `x`. If there is no variable, return the empty list to so indicate.

2. [10 points] Construct function **`substitute`** that substitutes an expression E for all occurrences of a specified variable x in another expression F. For example,

   ```
   (substitute '(+ (- x) (* z (+ y x))) 'x '(+ a b))
   ```

   would return

   ```
   (+ (-(+ a b)) (* z (+ y (+ a b))))
   ```

   If the variable does not occur in F, then F itself is returned.

3. [50 points, or more] Construct function **`simplify`** that will simplify a logic expression. This problem is open-ended and you get extra credit for doing better simplifications. Here is the **minimum requirement**:

   - 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, giving your simplifier various syntax trees:

```
(test (simplify '(+ x (- x))) 1)
(test (simplify '(* x (- x))) 0)
(test (simplify '(+ (* x (- y)) y)) '(+ x y))
```

There are many ways to approach this problem. Problems 1 and 2 are hints to use the Boole-Shannon expansion. You can couple this expansion with judicious application of various logical identities for further simplification. (Although you could also implement a full Quine-McCluskey algorithm, I am not suggesting that you go that far.)

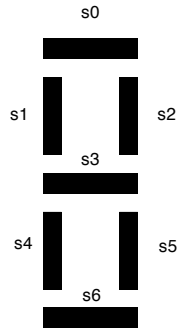
The test examples in

http://www.cs.hmc.edu/courses/2009/fall/cs42/a05tests.txt

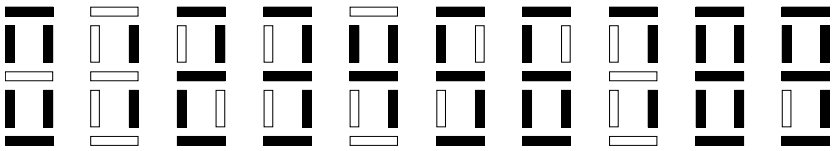
are all in the form expected by the parser. Your "read-eval-print" loop should call the simplifier on each example

4. [30 points]

A seven-segment display consists of seven LEDs (light-emitting diodes), numbered s0 through s6, that display the digits from 0 through 9 as shown:



Seven segment LED display



Display of 0 through 9 with a seven-segment display

In the following, we assume that the digits are coded in BCD (Binary-Coded Decimal, see below). This uses only ten of sixteen possible combinations of four bits. The remaining combinations are *don't-cares*. The seven segments correspond to seven logic functions, for which you are asked to provide simplified versions.

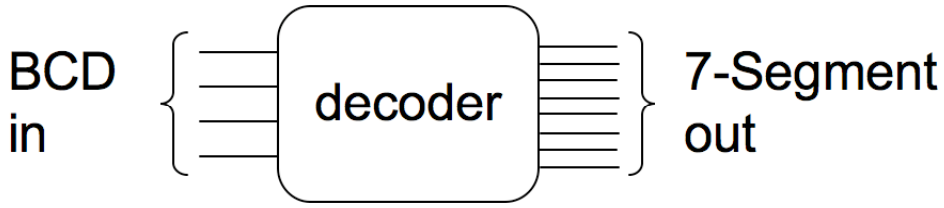


diagram of the decoder

Aligned to the BCD encodings are seven segment functions shown in the table below:

	BCD	seven-segment						
	wxyz	**s_0**	**s_1**	**s_2**	**s_3**	**s_4**	**s_5**	**s_6**
0	0000	1						
1	0001							
2	0010	1						
3	0011	1						
4	0100							
5	0101	1						
6	0110	1						
7	0111	1						
8	1000	1						
9	1001	1						

fill in for yourself
or, better, transcribe directly
to a Karnaugh map

a. Simplify each of the switching functions, using don't cares to advantage, to achieve a 2-level SOP (sum-of-products) realization. This means that each function is represented as the **+** of several terms, with each term being an ***** of a variable or the *negation* of a variable. For example, using the syntax of the language for problems 1-3, the function s_0 can be expressed as follows:

$$s_0 = w + y + x*z + -x*-z$$

Your answer should entail expressions for each of the functions s_0 to s_6. Show your derivation on paper; don't just give the answers.

b. Using your simplifier from problem 3, verify that each of your functions is correct by submitting to it the following type of expressions, representing the truth-table entries, to the simplifier (shown here for s_0). Each should simplify to 1. Use this to fix errors.

```
0.   -w*-x*-y*-z >  (w + y + x*z + -x*-z)
1.   -w*-x*-y* z > -(w + y + x*z + -x*-z)
2.   -w*-x* y*-z >  (w + y + x*z + -x*-z)
3.   -w*-x* y* z >  (w + y + x*z + -x*-z)
4.   -w* x*-y*-z > -(w + y + x*z + -x*-z)
5.   -w* x*-y* z >  (w + y + x*z + -x*-z)
6.   -w* x* y*-z >  (w + y + x*z + -x*-z)
7.   -w* x* y* z >  (w + y + x*z + -x*-z)
8.    w*-x*-y*-z >  (w + y + x*z + -x*-z)
9.    w*-x*-y* z >  (w + y + x*z + -x*-z)
```