Unate Recursive Tautology Code
Due electronically on Monday, 22 October 2011

Write a simple program (in C, C++, Java, Perl, etc.) that implements the *unate recursive tautology*. The unate recursive tauology, discussed in lecture notes, is also repeated in Homework #1.

For simplicity, assume that the sum-of-products form you will read in to test has this simple format:

- First line has one (1) integer variable, labeled VARS that indicates the number of variables.
- The next lines each describe one product term. Assume the variables are just represented as single lower-case ASCII chars: if there are VARS=3 variables in your function, then the variables are labeled "*a*", "*b*" and "*c*".
- A true literal "*a*" is represented as "+ *a*", *i.e.*, a plus sign, a space, and an "*a*". A complemented literal "a'" is represented as "− *a*", *i.e.*, a minus sign, a space, and an "*a*".
- A product term is just a sequence of space-delimited literals terminated with a semicolon.
- An empty product term (no literals, just a semicolon) terminates the entire input.

For example, for the function $f = a + c'd' + a'b' + abc + ab'd$, the input would look like the following:

```
4                             (4 vars a b c d)
+ a ;                          (1st term is a)
- c - d ;                    (2nd term is c'd')
- a - b ;                    (3rd term is  a'b')
+ a + b + c ;                (4th term is   abc)
+ a - b + d ;               (5th term is   ab'd)
;              (empty product term ends input)
```

Use the termination rules given in the notes. You should print out something "enlightening" each time your recursive tautology routine gets called, like what SOP form it got called on, and what the result was of checking each rule, and what variable you are splitting on. This will generate a simple trace that shows what the function is doing as it runs. Ultimately it should print out YES or NO for the overall function. A sample intermediate output could like like:

```
[ 01 11 11 11 ]
[ 11 11 10 10 ]
[ 10 10 11 11 ]
[ 01 01 01 11 ]
[ 01 10 11 01 ]
This is not unate.
Rule 1 does not apply.
Rule 2 does not apply.
Rule 3 does not apply.
Splitting the function with respect to a...
```

Dont get hung up in fancy data structures for the cubes, some simple arrays of `char`s or `int`s will do fine. The goal is for this thing to be *small*, and *simple*.

Do the following:

1. Write and turn in a brief project report. Project report can include technical details (your development environment, how to run/compile your program, # of lines, your selection of data structures, algorithms etc.), the program flow chart, sample input/output and interpretation, verification of your results (by hand calculations, code debugging or graphics), and finally, your comments on the program, project assignment and your own evaluation of your submission.
2. Turn in your code (Yes, a good indentation and comments are recommended.).
3. Turn in the output of your code running on the function listed in this problem.
4. Also, pick one other random example with 5 or 6 variables that IS a tautology, and show the Kmap to show how the cubes make a tautology then run your code on the example and hand in the printout.
5. Pick one other random example with 5 or 6 variables that IS NOT a tautology, and show the Kmap to show how the cubes are arranged, then run your code on the example and hand in the printout.