EE473—Introduction to Artificial Intelligence
Spring 2014
Problem Set #3
Due: Friday 28–February–2014

This problem set is an exercise in *symbolic manipulation*. It will teach you about *syntax*, *compilation*, *macros*, and *rewriting*. From an electrical-engineering perspective, you will build a simple logic simplifier. Logic simplifiers (aka *logic synthesis* tools) are an important part of modern VLSI CAD-tool suites. From an AI perspective, you will build a rule-based system for simplifying propositional-logic formulas.

We adopt the same syntactic and semantic definition of propositional logic as in the handout for problem set 2. Recall that the semantics of propositional logic is defined by way of the valuation function $\mathcal{V}(\Phi, I)$. Further recall that we use the symbol $\rightsquigarrow$ to mean *rewrites to*. We will define the following system of rewrite rules:

$$
\begin{array}{rcl}
\texttt{(not \#t)} & \rightsquigarrow & \texttt{\#f} \\
\texttt{(not \#f)} & \rightsquigarrow & \texttt{\#t} \\
\texttt{(not (not } \Phi\texttt{))} & \rightsquigarrow & \Phi \\
\texttt{(and)} & \rightsquigarrow & \texttt{\#t} \\
\texttt{(and } \Phi\texttt{)} & \rightsquigarrow & \Phi \\
\texttt{(and } \Phi_1 \ldots \Phi_m \texttt{ \#t } \Phi_{m+1} \ldots \Phi_n\texttt{)} & \rightsquigarrow & \texttt{(and } \Phi_1 \ldots \Phi_m \ \Phi_{m+1} \ldots \Phi_n\texttt{)} \\
\texttt{(and } \Phi_1 \ldots \Phi_m \texttt{ \#f } \Phi_{m+1} \ldots \Phi_n\texttt{)} & \rightsquigarrow & \texttt{\#f} \\
\texttt{(and } \Phi_1 \ldots \Phi_l \texttt{ (and } \Phi_{l+1} \ldots \Phi_m\texttt{) } \Phi_{m+1} \ldots \Phi_n\texttt{)} & \rightsquigarrow & \texttt{(and } \Phi_1 \ldots \Phi_l \ \Phi_{l+1} \ldots \Phi_m \ \Phi_{m+1} \ldots \Phi_n\texttt{)} \\
\texttt{(and } \Phi_1 \ldots \Phi_l \ \Phi \ \Phi_{l+1} \ldots \Phi_m \ \Phi \ \Phi_{m+1} \ldots \Phi_n\texttt{)} & \rightsquigarrow & \texttt{(and } \Phi_1 \ldots \Phi_l \ \Phi \ \Phi_{l+1} \ldots \Phi_m \ \Phi_{m+1} \ldots \Phi_n\texttt{)} \\
\texttt{(and } \Phi_1 \ldots \Phi_l \texttt{ (not } \Phi\texttt{) } \Phi_{l+1} \ldots \Phi_m \ \Phi \ \Phi_{m+1} \ldots \Phi_n\texttt{)} & \rightsquigarrow & \texttt{\#f} \\
\texttt{(and } \Phi_1 \ldots \Phi_l \ \Phi \ \Phi_{l+1} \ldots \Phi_m \texttt{ (not } \Phi\texttt{) } \Phi_{m+1} \ldots \Phi_n\texttt{)} & \rightsquigarrow & \texttt{\#f} \\
\texttt{(or)} & \rightsquigarrow & \texttt{\#f} \\
\texttt{(or } \Phi\texttt{)} & \rightsquigarrow & \Phi \\
\texttt{(or } \Phi_1 \ldots \Phi_m \texttt{ \#f } \Phi_{m+1} \ldots \Phi_n\texttt{)} & \rightsquigarrow & \texttt{(or } \Phi_1 \ldots \Phi_m \ \Phi_{m+1} \ldots \Phi_n\texttt{)} \\
\texttt{(or } \Phi_1 \ldots \Phi_m \texttt{ \#t } \Phi_{m+1} \ldots \Phi_n\texttt{)} & \rightsquigarrow & \texttt{\#t} \\
\texttt{(or } \Phi_1 \ldots \Phi_l \texttt{ (or } \Phi_{l+1} \ldots \Phi_m\texttt{) } \Phi_{m+1} \ldots \Phi_n\texttt{)} & \rightsquigarrow & \texttt{(or } \Phi_1 \ldots \Phi_l \ \Phi_{l+1} \ldots \Phi_m \ \Phi_{m+1} \ldots \Phi_n\texttt{)} \\
\texttt{(or } \Phi_1 \ldots \Phi_l \ \Phi \ \Phi_{l+1} \ldots \Phi_m \ \Phi \ \Phi_{m+1} \ldots \Phi_n\texttt{)} & \rightsquigarrow & \texttt{(or } \Phi_1 \ldots \Phi_l \ \Phi \ \Phi_{l+1} \ldots \Phi_m \ \Phi_{m+1} \ldots \Phi_n\texttt{)} \\
\texttt{(or } \Phi_1 \ldots \Phi_l \texttt{ (not } \Phi\texttt{) } \Phi_{l+1} \ldots \Phi_m \ \Phi \ \Phi_{m+1} \ldots \Phi_n\texttt{)} & \rightsquigarrow & \texttt{\#t} \\
\texttt{(or } \Phi_1 \ldots \Phi_l \ \Phi \ \Phi_{l+1} \ldots \Phi_m \texttt{ (not } \Phi\texttt{) } \Phi_{m+1} \ldots \Phi_n\texttt{)} & \rightsquigarrow & \texttt{\#t}
\end{array}
$$

Note that in the above rule set, there are rules where the same pattern variable, say Φ, appears more than once in the left hand side. This means that that the two (or more) subformulas that match that pattern variable must be syntactically the same (i.e. `equal?`).

We want you to implement the following procedures:

`boolean-simplify` Φ [*Procedure*]

Φ is a formula. Repeatedly applies the above rewrite rules to Φ and its subformulas until no more rules are applicable and then returns the resulting formula.

`truth-tables-match?` Φ Φ' [*Procedure*]

Φ and Φ' are formulas. Φ' will be the result of `(boolean-simplify` Φ`)`. Compares the truth tables of Φ and Φ' and returns `#t` if they are the same and `#f` if they are different.

The procedure `truth-tables-match?` will help you debug your `boolean-simplify` procedure. We therefore suggest that you write it first. You can use your solution to problem set 2 (or the provided solution) to construct your `truth-tables-match?` procedure.

To help debug and test your implementation, we have provided the GUI `(p3)`. This GUI is derived from the GUI for problem set 2 with several differences. First, the window is larger to allow display of two formulas and two truth tables. Second, there is a Simplify button. Like before, you create and edit formulas that appear on the left hand side of the window. And like before, when there are no empty subformulas, their truth table is displayed on the left hand side of the window. (For this, you will need a working `truth-table` procedure. You can use your own or the one provided.) When they are complete, you can click on Simplify. (Simplify is highlighted when the formula is complete to show that you can click on it.) When you click on Simplify, the formula on the left hand side of the window is passed to `boolean-simplify`. The output of `boolean-simplify` is displayed on the right hand side of the window along with its truth table. If both truth tables are the same, as indicated by `truth-tables-match?`, you will get an indication of that fact at the bottom of the window. If the truth tables differ, the GUI will tell you so. You may be tempted to create a `truth-tables-match?` procedure that lies and always reports a match. Such deviant behavior will not fool our grading procedure. Thus it is in your best interest to write a good `truth-tables-match?` procedure to help you catch bugs in your `boolean-simplify` procedure.

Good luck and have fun!