Mathematical Foundations of Computer Science
Jean Gallier

Mini Project
October 18, 2010; Due December 9, 2010
This project must be done individually

This project consists of a blend of theory and computer implementation. You are only required to give answers to Parts 1 and 2. These constitute 10% of your total grade. Part 3, the programming part is optional and will count only for extra credit (up to 10% of your total grade).

If you decide do do part 3, the code implementing the various required algorithms may be written in Java, C++, Python, Mathematica or Matlab. Please submit the source for every program in addition to every required output. Please pay attention to the readability of every output (and make sure your code is suitably documented). You may submit the various requested proofs in a separate document.

Even though natural deduction proof systems for classical propositional logic are complete (with respect to the truth values semantics), they are not adequate for designing algorithms searching for proofs (because of the amount of nondeterminism involved).

Gentzen designed a different kind of proof system using sequents (later refined by Kleene, Smullyan and others) that is far better suited for the design of automated theorem provers. Using such a proof system (a sequent calculus), it is relatively easy to design a procedure that terminates for all input propositions, $P$, and either certifies that $P$ is (classically) valid or else returns some (or all) falsifying truth assignment(s) for $P$. In fact, if $P$ is valid, the tree returned by the algorithm can be viewed as a proof of $P$ in this proof system.

For this mini-project, we describe a Gentzen sequent-calculus, $G'$, for propositional logic that lends itself well to the implementation of algorithms searching for proofs or falsifying truth assignments of propositions.

Such algorithms build trees whose nodes are labeled with pairs of sets called sequents. A sequent is a pair of sets of propositions denoted by

$$P_1, \ldots, P_m \rightarrow Q_1, \ldots, Q_n,$$

with $m, n \geq 0$. Symbolically, a sequent is usually denoted $\Gamma \rightarrow \Delta$, where $\Gamma$ and $\Delta$ are two finite sets of propositions (not necessarily disjoint).
For example,
\[ \rightarrow P \Rightarrow (Q \Rightarrow P), \quad P \lor Q \rightarrow, \quad P, Q \rightarrow P \land Q \]
are sequents. The sequent \( \rightarrow \), where both \( \Gamma = \Delta = \emptyset \) corresponds to falsity.

The choice of the symbol, \( \rightarrow \), to separate the two sets of propositions \( \Gamma \) and \( \Delta \) is commonly used and was introduced by Gentzen but there is nothing special about it. If you don’t like it, you may replace it by any symbol of your choice as long as that symbol does not clash with the logical connectives (\( \Rightarrow, \land, \lor, \neg \)). For example, you could denote a sequent \( P_1, \ldots, P_m; Q_1, \ldots, Q_n \)
using the semicolon as a separator.

Given a truth assignment, \( v \), to the propositional letters in the propositions \( P_i \) and \( Q_j \), we say that \( v \) satisfies the sequent, \( P_1, \ldots, P_m \rightarrow Q_1, \ldots, Q_n \), iff
\[ v((P_1 \land \cdots \land P_m) \Rightarrow (Q_1 \lor \cdots \lor Q_n)) = \text{true}, \]
or equivalently, \( v \) falsifies the sequent, \( P_1, \ldots, P_m \rightarrow Q_1, \ldots, Q_n \), iff
\[ v(P_1 \land \cdots \land P_m \land \neg Q_1 \land \cdots \land \neg Q_n) = \text{true}, \]
iff
\[ v(P_i) = \text{true}, \quad 1 \leq i \leq m \quad \text{and} \quad v(Q_j) = \text{false}, \quad 1 \leq j \leq n. \]
A sequent is valid iff it is satisfied by all truth assignments iff it cannot be falsified.

Note that a sequent, \( P_1, \ldots, P_m \rightarrow Q_1, \ldots, Q_n \), can be falsified iff some truth assignment satisfies all of \( P_1, \ldots, P_m \) and falsifies all of \( Q_1, \ldots, Q_n \). In particular, if \{\( P_1, \ldots, P_m \)\} and \{\( Q_1, \ldots, Q_n \)\} have some common proposition (they have a nonempty intersection), then the sequent, \( P_1, \ldots, P_m \rightarrow Q_1, \ldots, Q_n \), is valid. On the other hand if all the \( P_i \)'s and \( Q_j \)'s are propositional letters and \{\( P_1, \ldots, P_m \)\} and \{\( Q_1, \ldots, Q_n \)\} are disjoint (they have no symbol in common), then the sequent, \( P_1, \ldots, P_m \rightarrow Q_1, \ldots, Q_n \), is falsified by the truth assignment, \( v \), where \( v(P_i) = \text{true} \), for \( i = 1, \ldots, m \) and \( v(Q_j) = \text{false} \), for \( j = 1, \ldots, n \).

In the special case where \( m = 0 \), the truth assignment \( v \) falsifies the sequent \( \rightarrow Q_1, \ldots, Q_n \) iff \( v(Q_j) = \text{false} \) for \( j = 1, \ldots, n \), and in the special case where \( n = 0 \), the truth assignment \( v \) falsifies the sequent \( P_1, \ldots, P_m \) iff \( v(P_i) = \text{true} \) for \( i = 1, \ldots, m \). Thus, an empty left-hand side of a sequent is interpreted as \text{true} and an empty right-hand side of a sequent is interpreted as \text{false}. In particular, the degenerate sequent \( \rightarrow \) is falsified by all truth assignments.

The main idea behind the design of the proof system \( G' \) is to systematically try to falsify a sequent. If such an attempt fails, the sequent is valid and a proof tree is found. Otherwise, all falsifying truth assignments are returned. In some sense

\[ \text{failure to falsify is success (in finding a proof)!} \]
The rules of $G'$ are designed so that the conclusion of a rule is falsified by a truth assignment, $v$, iff its single premise of one of its two premises is falsified by $v$. Thus, these rules can be viewed as two-way rules that can either be read bottom-up or top-down.

Here are the axioms and the rules of the sequent calculus $G'$:

Axioms: $\Gamma, P \rightarrow P, \Delta$

Inference rules:

\[
\begin{align*}
\frac{\Gamma, P, Q, \Delta \rightarrow \Lambda}{\Gamma, P \land Q, \Delta \rightarrow \Lambda} & \quad \wedge: \text{left} \\
\frac{\Gamma \rightarrow \Delta, P, \Lambda \quad \Gamma \rightarrow \Delta, Q, \Lambda}{\Gamma \rightarrow \Delta, P \land Q, \Lambda} & \quad \wedge: \text{right} \\
\frac{\Gamma, P, Q, \Delta \rightarrow \Lambda}{\Gamma, P \lor Q, \Delta \rightarrow \Lambda} & \quad \lor: \text{left} \\
\frac{\Gamma \rightarrow \Delta, P, \Lambda \quad \Gamma \rightarrow \Delta, Q, \Lambda}{\Gamma \rightarrow \Delta, P \lor Q, \Lambda} & \quad \lor: \text{right} \\
\frac{\Gamma, \Delta \rightarrow P, \Lambda \quad Q, \Gamma, \Delta \rightarrow \Lambda}{\Gamma, P \Rightarrow Q, \Delta \rightarrow \Lambda} & \quad \Rightarrow: \text{left} \\
\frac{P, \Gamma \rightarrow Q, \Delta, \Lambda}{\Gamma \rightarrow \Delta, P \Rightarrow Q, \Lambda} & \quad \Rightarrow: \text{right} \\
\frac{\Gamma, \Delta \rightarrow P, \Lambda}{\Gamma, \neg P, \Delta \rightarrow \Lambda} & \quad \neg: \text{left} \\
\frac{P, \Gamma \rightarrow \Delta, \Lambda}{\Gamma \rightarrow \Delta, \neg P, \Lambda} & \quad \neg: \text{right}
\end{align*}
\]

where $\Gamma, \Delta, \Lambda$ are any finite sets of propositions, possibly the empty set.

A deduction tree is either a one-node tree labeled with a sequent or a tree constructed according to the rules of system $G'$. A proof tree (or proof) is a deduction tree whose leaves are all axioms. A proof tree for a proposition, $P$, is a proof tree for the sequent, $\rightarrow P$ (with an empty left-hand side).

For example,

\[P, Q \rightarrow P\]

is a proof tree.

Here is a proof tree for $(P \Rightarrow Q) \Rightarrow (\neg Q \Rightarrow \neg P)$:

\[
\begin{align*}
P, \neg Q & \rightarrow P \\
\neg Q & \rightarrow \neg P, P \\
\rightarrow & \quad P, (\neg Q \Rightarrow \neg P) \\
\end{align*}
\]

\[
\begin{align*}
Q & \rightarrow Q, \neg P \\
\neg Q, Q & \rightarrow \neg P \\
\rightarrow & \quad Q \rightarrow (\neg Q \Rightarrow \neg P) \\
\end{align*}
\]

\[
\begin{align*}
(P \Rightarrow Q) & \rightarrow (\neg Q \Rightarrow \neg P) \\
\rightarrow & \quad (P \Rightarrow Q) \Rightarrow (\neg Q \Rightarrow \neg P)
\end{align*}
\]

The following is a deduction tree but not a proof tree:
\[
\begin{array}{c}
P, R \rightarrow P \\
\hline
R \rightarrow \neg P, P \\
\hline
\rightarrow P, (R \Rightarrow \neg P) \\
\end{array}
\quad
\begin{array}{c}
R, Q, P \rightarrow \\
\hline
R, Q \rightarrow \neg P \\
\hline
Q \rightarrow (R \Rightarrow \neg P) \\
\hline
\end{array}
\]

\[
(P \Rightarrow Q) \Rightarrow (R \Rightarrow \neg P) \\
\rightarrow (P \Rightarrow Q) \Rightarrow (R \Rightarrow \neg P)
\]

since its rightmost leaf, \( R, Q, P \rightarrow \), is falsified by the truth assignment \( v(P) = v(Q) = v(R) = \text{true} \), which also falsifies \( (P \Rightarrow Q) \Rightarrow (R \Rightarrow \neg P) \).

Let us call a sequent, \( P_1, \ldots, P_m \rightarrow Q_1, \ldots, Q_n \), \textit{finished} if either it is axiom (\( P_i = Q_j \) for some \( i \) and some \( j \)) or all the propositions \( P_i \) and \( Q_j \) are atomic and \( \{P_1, \ldots, P_m\} \cap \{Q_1, \ldots, Q_n\} = \emptyset \). We will also say that a deduction tree is finished if all its leaves are finished sequents.

The beauty of the proof system \( G' \) is that for every sequent, \( P_1, \ldots, P_m \rightarrow Q_1, \ldots, Q_n \), the process of building a deduction tree from this sequent \textit{always terminates with a tree where all leaves are finished, independently of the order in which the rules are applied}. Therefore, we can apply any strategy we want when we build a deduction tree and we are sure that we will get a deduction tree with all its leaves finished. If all the leaves are axioms, then we have a proof tree and the sequent is valid, or else all the leaves that are not axioms yield a falsifying assignment, and all falsifying assignments for the root sequent are found this way.

If we only want to know whether a proposition (or a sequent) is valid, we can stop as soon as we find a finished sequent that is not an axiom since in this case, the input sequent is falsifiable.

(1) (Required) Prove that for every sequent, \( P_1, \ldots, P_m \rightarrow Q_1, \ldots, Q_n \), any sequence of applications of the rules of \( G' \) terminates with a deduction tree whose leaves are all finished sequents (a finished deduction tree).

\textit{Hint.} Define the number of connectives, \( c(P) \), in a proposition, \( P \), as follows:

(1) If \( P \) is a propositional symbol, then
\[
c(P) = 0.
\]

(2) If \( P = \neg Q \), then
\[
c(\neg Q) = c(Q) + 1.
\]

(3) If \( P = Q \ast R \), where \( \ast \in \{\Rightarrow, \lor, \land\} \), then
\[
c(Q \ast R) = c(Q) + c(R) + 1.
\]
Given a sequent,
\[ \Gamma \rightarrow \Delta = P_1, \ldots, P_m \rightarrow Q_1, \ldots, Q_n, \]
define the number of connectives, \( c(\Gamma \rightarrow \Delta) \), in \( \Gamma \rightarrow \Delta \) by
\[
c(\Gamma \rightarrow \Delta) = c(P_1) + \cdots + c(P_m) + c(Q_1) + \cdots + c(Q_n).
\]

Prove that the application of every rule decreases the number of connectives in the premise(s) of the rule.

(2) (Required) Prove that for every sequent, \( P_1, \ldots, P_m \rightarrow Q_1, \ldots, Q_n \), for every finished deduction tree, \( T \), constructed from \( P_1, \ldots, P_m \rightarrow Q_1, \ldots, Q_n \) using the rules of \( G' \), every truth assignment, \( v \), satisfies \( P_1, \ldots, P_m \rightarrow Q_1, \ldots, Q_n \) iff \( v \) satisfies every leaf of \( T \). Equivalently, a truth assignment, \( v \), falsifies \( P_1, \ldots, P_m \rightarrow Q_1, \ldots, Q_n \) iff \( v \) falsifies some leaf of \( T \).

Deduce from the above that a sequent is valid iff all leaves of every finished deduction tree, \( T \), are axioms. Furthermore, if a sequent is not valid, then for every finished deduction tree, \( T \), for that sequent, every falsifying assignment for that sequent is a falsifying assignment of some leaf of the tree, \( T \).

(3) Programming Project; Extra Credit:

Design an algorithm taking any sequent as input and constructing a finished deduction tree. If the deduction tree is a proof tree, output this proof tree in some fashion (such a tree can be quite big so you may have to find ways of “flattening” these trees). If the sequent is falsifiable, stop when the algorithm encounters the first leaf which is not an axiom and output the corresponding falsifying truth assignment.

I suggest using a depth-first expansion strategy for constructing a deduction tree. What this means is that when building a deduction tree, the algorithm will proceed recursively as follows: Given a non-finished sequent
\[ A_1, \ldots, A_p \rightarrow B_1, \ldots, B_q, \]
if \( A_i \) is the leftmost non-atomic proposition if such proposition occurs on the left or if \( B_j \) is the leftmost non-atomic proposition if all the \( A_i \)'s are atomic, then

(1) The sequent is of the form
\[ \Gamma, A_i, \Delta \rightarrow \Lambda, \]
with \( A_i \) the leftmost non-atomic proposition. Then either

(a) \( A_i = C_i \land D_i \) or \( A_i = \neg C_i \), in which case either we recursively construct a (finished) deduction tree
\[
D_1 \\
\Gamma, C_i, D_i, \Delta \rightarrow \Lambda
\]
to get the deduction tree
\[
\mathcal{D}_1 \\
\Gamma, C_i, D_i, \Delta \rightarrow \Lambda \\
\Gamma, C_i \land D_i, \Delta \rightarrow \Lambda
\]

or we recursively construct a (finished) deduction tree
\[
\mathcal{D}_1 \\
\Gamma, \Delta \rightarrow C_i, \Lambda
\]

to get the deduction tree
\[
\mathcal{D}_1 \\
\Gamma, \Delta \rightarrow C_i, \Lambda \\
\Gamma, \neg C_i, \Delta \rightarrow \Lambda
\]

or

(b) \( A_i = C_i \lor D_i \) or \( A_i = C_i \Rightarrow D_i \), in which case either we recursively construct two (finished) deduction trees

\[
\mathcal{D}_1 \\
\Gamma, C_i, \Delta \rightarrow \Lambda \\
\mathcal{D}_2 \\
\Gamma, D_i, \Delta \rightarrow \Lambda
\]

and

\[
\mathcal{D}_1 \\
\Gamma, C_i, \Delta \rightarrow \Lambda \\
\mathcal{D}_2 \\
\Gamma, D_i, \Delta \rightarrow \Lambda
\]

or we recursively construct two (finished) deduction trees

\[
\mathcal{D}_1 \\
\Gamma, \Delta \rightarrow C_i, \Lambda \\
\mathcal{D}_2 \\
\Gamma, \Delta \rightarrow C_i, \Lambda \\
\mathcal{D}_1 \\
\Gamma, \Delta \rightarrow C_i, \Lambda \\
\mathcal{D}_2 \\
\Gamma, C_i \Rightarrow D_i, \Delta \rightarrow \Lambda
\]
(2) The non-finished sequent is of the form

\[ \Gamma \rightarrow \Delta, B_j, \Lambda, \]

with \( B_j \) the leftmost non-atomic proposition. Then either

(a) \( B_j = C_j \lor D_j \) or \( B_j = C_j \rightarrow D_j \), or \( B_j = \neg C_j \), in which case either we recursively construct a (finished) deduction tree

\[ D_1 \]
\[ \Gamma \rightarrow \Delta, C_j, D_j, \Lambda \]

to get the deduction tree

\[ D_1 \]
\[ \Gamma \rightarrow \Delta, C_j, D_j, \Lambda \]
\[ \Gamma \rightarrow \Delta, C_j \lor D_j, \Lambda \]

or we recursively construct a (finished) deduction tree

\[ D_1 \]
\[ C_j, \Gamma \rightarrow D_j, \Delta, \Lambda \]

to get the deduction tree

\[ D_1 \]
\[ C_j, \Gamma \rightarrow D_j, \Delta, \Lambda \]
\[ \Gamma \rightarrow \Delta, C_j \rightarrow D_j, \Lambda \]

or we recursively construct a (finished) deduction tree

\[ D_1 \]
\[ C_j, \Gamma \rightarrow \Delta, \Lambda \]

to get the deduction tree

\[ D_1 \]
\[ C_j, \Gamma \rightarrow \Delta, \Lambda \]
\[ \Gamma \rightarrow \Delta, \neg C_j, \Lambda \]

or
(b) \( B_j = C_j \land D_j \), in which case we recursively construct two (finished) deduction trees

\[
\begin{array}{c}
\mathcal{D}_1 \\
\Gamma \rightarrow \Delta, C_j, \Lambda
\end{array}
\quad \text{and} \quad
\begin{array}{c}
\mathcal{D}_2 \\
\Gamma \rightarrow \Delta, D_j, \Lambda
\end{array}
\]

to get the deduction tree

\[
\begin{array}{c}
\mathcal{D}_1 \\
\Gamma \rightarrow \Delta, C_j, \Lambda
\end{array}
\quad \begin{array}{c}
\mathcal{D}_2 \\
\Gamma \rightarrow \Delta, D_j, \Lambda
\end{array}
\quad \begin{array}{c}
\Gamma \rightarrow \Delta, C_j \land D_j, \Lambda
\end{array}
\]

If you prefer, you can apply a \textit{breadth-first expansion strategy} for constructing a deduction tree.

\textbf{Input Format.}

In order to make life easier, I suggest that the input propositions be given in postfix notation because it is easy to build a tree from an expression in postfix, using a stack.

Also, use “\&” for \( \land \), “\|” for \( \lor \), “\~” for \( \neg \) and “\>” for \( \Rightarrow \) (or any other reasonable choice of characters).

Each propositional letter \( P_i \) is represented by the string “Pi”. For example, \( P_{17} \) will be represented by P17.

Use spaces as delimiters.

The postfix representation is specified recursively as follows:

1. \text{postfix}(P_i) = P_i
2. \text{postfix}(P \lor Q) = \text{postfix}(P) \text{ postfix}(Q) \|
3. \text{postfix}(P \land Q) = \text{postfix}(P) \text{ postfix}(Q) \&
4. \text{postfix}(P \Rightarrow Q) = \text{postfix}(P) \text{ postfix}(Q) >
5. \text{postfix}(\neg P) = \text{postfix}(P) \sim

For example, the proposition

\[(P_1 \lor \neg P_2) \land (\neg P_2 \Rightarrow P_3)\]

has the following postfix representation:

\[P_1 \ P_2 \sim \ | \ P_2 \sim P_3 \ > \ &\]
Output Format.

Your program should output a Gentzen-deduction tree in legible form and specify whether this tree is a proof tree or not. The nodes of deduction trees are Gentzen sequents, that is, expressions of the form

\[ P_1, \ldots, P_m \Rightarrow Q_1, \ldots, Q_n, \]

where the \( P_i \)'s and \( Q_i \)'s are propositions in postfix notation (or if you work harder, in standard infix notation), separated by some delimiter such as \( \Rightarrow \).