**Staff Solutions to In-Class Problems Week 2, Fri.**

**Problem 1.**
Prove by truth table that OR distributes over AND, namely,

\[ P \text{ OR } (Q \text{ AND } R) \text{ is equivalent to } (P \text{ OR } Q) \text{ AND } (P \text{ OR } R) \]  

(1)

**Solution.**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>P</th>
<th>Q</th>
<th>R</th>
<th>P OR (Q AND R)</th>
<th>(P OR Q) AND (P OR R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

The highlighted column giving the truth values of the first formula is the same as the corresponding column of the second formula, so the two propositional formulas are equivalent. ■
Problem 2.
This problem\footnote{From Rosen, 5th edition, Exercise 1.1.36} examines whether the following specifications are satisfiable:

1. If the file system is not locked, then
   (a) new messages will be queued.
   (b) new messages will be sent to the messages buffer.
   (c) the system is functioning normally, and conversely, if the system is functioning normally, then the file system is not locked.

2. If new messages are not queued, then they will be sent to the messages buffer.

3. New messages will not be sent to the message buffer.

(a) Begin by translating the five specifications into propositional formulas using four propositional variables:

\[
\begin{align*}
L & \text{ := file system locked,} \\
Q & \text{ := new messages are queued,} \\
B & \text{ := new messages are sent to the message buffer,} \\
N & \text{ := system functioning normally.}
\end{align*}
\]

Solution. The translations of the specifications are:

\[
\begin{align*}
\neg L & \implies Q & \text{(Spec. 1.(a))} \\
\neg L & \implies B & \text{(Spec. 1.(b))} \\
\neg L & \iff N & \text{(Spec. 1.(c))} \\
\neg Q & \implies B & \text{(Spec. 2.)} \\
\neg B & & \text{(Spec. 3.)}
\end{align*}
\]

(b) Demonstrate that this set of specifications is satisfiable by describing a single truth assignment for the variables \(L, Q, B, N\) and verifying that under this assignment, all the specifications are true.

Solution. An assignment that works is

\[
\begin{align*}
L & = \text{T} \\
N & = \text{F} \\
Q & = \text{T} \\
B & = \text{F}.
\end{align*}
\]

To find this assignment, we could have started constructing the sixteen line truth table—one line for each way of assigning truth values to the four variables \(L, N, Q, \text{ and } B\)—and calculated the truth value of the AND of all the five specifications under that assignment, continuing until we got one that made the AND-formula true.

If for every one of the sixteen possible truth assignments, the AND-formula was false, then the system is not satisfiable.\footnote{From Rosen, 5th edition, Exercise 1.1.36}
Staff Solutions to In-Class Problems Week 2, Fri.

(c) Argue that the assignment determined in part (b) is the only one that does the job.

Solution. We can avoid calculating all 16 rows of the full truth-table calculation suggested in the solution to part (b) by reasoning as follows. In any truth assignment that makes all five specifications true,

- $B$ must be false, or the last specification, (Spec. 3.), would be false.
- Given that $B$ is false, (Spec. 2.) and (Spec. 1.(b)) can be true only if $Q$ and $L$ are true.
- Given that $L$ is true, (Spec. 1.(c)) can be true only if $N$ is false.

Thus, in order for all five specifications to be true, the assignment has to be the one in the solution to part (b).

Problem 3.
When the mathematician says to his student, “If a function is not continuous, then it is not differentiable,” then letting $D$ stand for “differentiable” and $C$ for continuous, the only proper translation of the mathematician’s statement would be

$$\text{NOT}(C) \text{ IMPLIES NOT}(D),$$

or equivalently,

$$D \text{ IMPLIES } C.$$

But when a mother says to her son, “If you don’t do your homework, then you can’t watch TV,” then letting $T$ stand for “can watch TV” and $H$ for “do your homework,” a reasonable translation of the mother’s statement would be

$$\text{NOT}(H) \text{ IFF NOT}(T),$$

or equivalently,

$$H \text{ IFF } T.$$

Explain why it is reasonable to translate these two IF-THEN statements in different ways into propositional formulas.

STAFF NOTE: If a discussion about the watching TV and homework doesn’t get going on its own, ask students to come up with rationales for both the IFF and the IMPLIES interpretations, like those below for example.

Solution. We know that a differentiable function must be continuous, so when a function is not continuous, it is also not differentiable. Now mathematicians use IMPLIES in the technical way given by its truth table. In particular, if a function is continuous then to a mathematician, the implication

$$\text{NOT}(C) \text{ IMPLIES NOT}(D),$$

is automatically true since the hypothesis (left hand side of the IMPLIES) is false. So whether or not continuity holds, the mathematician could comfortably assert the IMPLIES statement knowing it is correct. And of course a mathematician does not mean IFF, since she knows a function that is not differentiable may well be continuous.

On the other hand, while the mother certainly means that her son cannot watch TV if he does not do his homework, both she and her son most likely understand that if he does do his homework, then he will be allowed watch TV. In this case, even though the Mother uses an IF-THEN phrasing, she really means IFF.

On the other hand, circumstances in the household might be that the boy may watch TV when he has not only done his homework, but also cleaned up his room. In this case, just doing homework would not imply...
being allowed to watch TV — the boy won’t be allowed to watch TV if he hasn’t cleaned his room, even if he has done his homework, so in this case the Mother really means IMPLIES.

The general point here is that semantics (meaning) trumps syntax (sentence structure): even though the mathematician’s and mother’s statements have the same structure, their meaning may warrant different translations into precise logical language.

Problem 4.

Propositional logic comes up in digital circuit design using the convention that $T$ corresponds to 1 and $F$ to 0. A simple example is a 2-bit half-adder circuit. This circuit has 3 binary inputs, $a_1, a_0$ and $b$, and 3 binary outputs, $c, s_1, s_0$. The 2-bit word $a_1a_0$ gives the binary representation of an integer, $k$, between 0 and 3. The 3-bit word $cs_1s_0$ gives the binary representation of $k + b$. The third output bit, $c$, is called the final carry bit.

So if $k$ and $b$ were both 1, then the value of $a_1a_0$ would be 01 and the value of the output $cs_1s_0$ would 010, namely, the 3-bit binary representation of 1 + 1.

In fact, the final carry bit equals 1 only when all three binary inputs are 1, that is, when $k = 3$ and $b = 1$. In that case, the value of $cs_1s_0$ is 100, namely, the binary representation of 3 + 1.

This 2-bit half-adder could be described by the following formulas:

\[
\begin{align*}
c_0 &= b \\
s_0 &= a_0 \text{ XOR } c_0 \\
c_1 &= a_0 \text{ AND } c_0 & \text{ the carry into column 1} \\
s_1 &= a_1 \text{ XOR } c_1 \\
c_2 &= a_1 \text{ AND } c_1 & \text{ the carry into column 2} \\
c &= c_2.
\end{align*}
\]

(a) Generalize the above construction of a 2-bit half-adder to an $n+1$ bit half-adder with inputs $a_n, \ldots, a_1, a_0$ and $b$ and outputs $c, s_n, \ldots, s_1, s_0$. That is, give simple formulas for $s_i$ and $c_i$ for $0 \leq i \leq n + 1$, where $c_i$ is the carry into column $i + 1$, and $c = c_{n+1}$.

Solution. The $n + 1$-bit word $a_n \ldots a_1a_0$ will be the binary representation of an integer, $s$, between 0 and $2^{n+1} - 1$. The circuit will have $n + 2$ outputs $c, s_n, \ldots, s_1, s_0$ where the $n + 2$-bit word $cs_n \ldots s_1s_0$ gives the binary representation of $s + b$. The variable $c_i$ will be the “carry” from the $i$th

Here are some simple formulas that define such a half-adder:

\[
\begin{align*}
c_0 &= b, \\
s_i &= a_i \text{ XOR } c_i & \text{ for } 0 \leq i \leq n, \\
c_{i+1} &= a_i \text{ AND } c_i & \text{ for } 0 \leq i \leq n, \\
c &= c_{n+1}.
\end{align*}
\]

(b) Write similar definitions for the digits and carries in the sum of two $n + 1$-bit binary numbers $a_n \ldots a_1a_0$ and $b_n \ldots b_1b_0$. 
Solution. Define

\[
\begin{align*}
  c_0 &= 0 \\
  s_i &= a_i \text{ XOR } b_i \text{ XOR } c_i \quad \text{for } 0 \leq i \leq n, \\
  c_{i+1} &= (a_i \text{ AND } b_i) \text{ OR } (a_i \text{ AND } c_i) \text{ OR } (b_i \text{ AND } c_i) \quad \text{for } 0 \leq i \leq n, \\
  c &= c_{n+1}.
\end{align*}
\]

Visualized as digital circuits, the above adders consist of a sequence of single-digit half-adders or adders strung together in series. These circuits mimic ordinary pencil-and-paper addition, where a carry into a column is calculated directly from the carry into the previous column, and the carries have to ripple across all the columns before the carry into the final column is determined. Circuits with this design are called \textit{ripple-carry} adders. Ripple-carry adders are easy to understand and remember and require a nearly minimal number of operations. But the higher-order output bits and the final carry take time proportional to \(n\) to reach their final values.

(c) How many of each of the propositional operations does your adder from part (b) use to calculate the sum?

Solution. The scheme given in the solution to part (b) uses \(3(n + 1)\) AND’s, \(2(n + 1)\) XOR’s, and \(2(n + 1)\) OR’s for a total of \(7(n + 1)\) operations.\(^2\)

And if you have time...

\textbf{STAFF NOTE}: A programming problem—easy and not specially important—to raise the morale of students who are better at programming than at proofs.

\textbf{Supplemental Problem}\(^3\)

As a break from proofs, here’s a short programming exercise related to truth tables:

Problem 5.
Describe a simple procedure which, given a positive integer argument, \(n\), produces a width \(n\) array of truth-values whose rows would be all the possible truth-value assignments for \(n\) propositional variables. For example, for \(n = 2\), the array would be:

\[
\begin{array}{cc}
  T & T \\
  T & F \\
  F & T \\
  F & F
\end{array}
\]

Your description can be in English, or a simple program in some familiar language such as Python or Java.

---

\(^2\)Because \(c_0\) is always 0, you could skip all the operations involving it. Then the counts are \(3n + 1\) AND’s, \(2n + 1\) XOR’s, and \(2n\) OR’s for a total of \(7n + 2\) operations.

\(^3\)There is no need to study supplemental problems when preparing for quizzes or exams.
A simple recursive approach is to start with an \( n = 1 \) array, namely a \( 1 \times 2 \) array whose first row has the entry \( T \) and whose second row has the entry \( F \). Build the width-\((n + 1)\) array for \( n + 1 \) variables recursively by making two copies of the width-\( n \) array for \( n \) variables. Then add a \( F \) entry at the beginning of each row of the first copy to form an array of width-\((n + 1)\), add a \( T \) entry at the beginning of each row of the second copy to form a second array of width-\((n + 1)\), and finally, place the first width-\((n + 1)\) array on top of the second width-\((n + 1)\) array, forming a width-\((n + 1)\) array of twice the height of the width-\( n \) array. This is the answer.

Here’s a Python program that carries generates the rows of the array for \( n \) variables in this way:

```python
def truth_values(n):
    if n <= 0:
        return [[]]
    else:
        def choice(elems):
            return [[False] + e for e in elems] + \
                   [[True] + e for e in elems]
        return choice(truth_values(n - 1))

>>> print truth_values(3)
[[False, False, False], [False, False, True], 
 [False, True, False], [False, True, True], 
 [True, False, False], [True, False, True], 
 [True, True, False], [True, True, True]]
```

An alternative approach makes the observation that the rows of the array for \( n \) variables as constructed above consist of the \( n \)-bit binary representations of the successive integers 0 to \( 2^n - 1 \), when we treat \( F \) as a code for 0 and \( T \) as a code for 1. For example, if, in the above array for \( n = 3 \), we number successive rows starting with zero, then row number three is \((f \; t \; t)\) which codes \( 011 \), namely, the binary representation of three. So we generate the \( k \)th row of the array for \( n \) variables by finding the length-\( n \) binary representation of \( k \) and converting 0’s back to \( F \) and 1’s back to \( T \).

There are much faster adder circuits than the simple ripple carry adder in Problem 4. If you have time and are interested, you can try tackling the following description of an ingenious (half) adder design.

**Problem 6.**

There are adder circuits that are much faster, and only slightly larger, than the ripple-carry circuits of Problem 3.5 of the course text. They work by computing the values in later columns for both a carry of 0 and a carry of 1, in parallel. Then, when the carry from the earlier columns finally arrives, the pre-computed answer can be quickly selected. We’ll illustrate this idea by working out the equations for an \((n + 1)\)-bit parallel half-adder.

Parallel half-adders are built out of parallel add1 modules. An \((n + 1)\)-bit add1 module takes as input the \((n + 1)\)-bit binary representation, \( a_n \ldots a_1 a_0 \), of an integer, \( s \), and produces as output the binary representation, \( c \ p_n \ldots p_1 p_0 \), of \( s + 1 \).

(a) A 1-bit add1 module just has input \( a_0 \). Write propositional formulas for its outputs \( c \) and \( p_0 \).

**Solution.**

\[
p_0 = a_0 \text{ XOR } 1 = \text{NOT}(a_0) \quad (2)
\]
\[
c = a_0. \quad (3)
\]
(b) Explain how to build an \((n + 1)\)-bit parallel half-adder from an \((n + 1)\)-bit \textit{add1} module by writing a propositional formula for the half-adder output, \(o_i\), using only the variables \(a_i\), \(p_i\), and \(b\).

**Solution.**

\[
o_i = (b \ \text{AND} \ p_i) \ \text{OR} \ (\text{NOT}(b) \ \text{AND} \ a_i)
\]

We can build a double-size \textit{add1} module with \(2(n + 1)\) inputs using two single-size \textit{add1} modules with \(n + 1\) inputs. Suppose the inputs of the double-size module are \(a_{2n+1}, \ldots, a_1, a_0\) and the outputs are \(c, p_{2n+1}, \ldots, p_1, p_0\). The setup is illustrated in Figure 1.

Namely, the first single size \textit{add1} module handles the first \(n + 1\) inputs. The inputs to this module are the low-order \(n + 1\) input bits \(a_n, \ldots, a_1, a_0\), and its outputs will serve as the first \(n + 1\) outputs \(p_n, \ldots, p_1, p_0\) of the double-size module. Let \(c_{(1)}\) be the remaining carry output from this module.

The inputs to the second single-size module are the higher-order \(n + 1\) input bits \(a_{2n+1}, \ldots, a_{n+2}, a_{n+1}\). Call its first \(n + 1\) outputs \(r_n, \ldots, r_1, r_0\) and let \(c_{(2)}\) be its carry.

(c) Write a formula for the carry, \(c\), in terms of \(c_{(1)}\) and \(c_{(2)}\).

**Solution.**

\[
c = c_{(1)} \ \text{AND} \ c_{(2)}.
\]

(d) Complete the specification of the double-size module by writing propositional formulas for the remaining outputs, \(p_i\), for \(n + 1 \leq i \leq 2n + 1\). The formula for \(p_i\) should only involve the variables \(a_i, r_{i-(n+1)},\) and \(c_{(1)}\).

**Solution.** The \(n + 1\) high-order outputs of the double-size module are the same as the inputs if there is no carry from the low-order \(n + 1\) outputs, and otherwise is the same as the outputs of the second single-size \textit{add1} module. So

\[
p_i = (\text{NOT}(c_{(1)}) \ \text{AND} \ a_i) \ \text{OR} \ (c_{(1)} \ \text{AND} \ r_{i-(n+1)}).
\]

for \(n + 1 \leq i \leq 2n + 1\).

(e) Parallel half-adders are exponentially faster than ripple-carry half-adders. Confirm this by determining the largest number of propositional operations required to compute any one output bit of an \(n\)-bit \textit{add} module. (You may assume \(n\) is a power of 2.)

**Solution.** The most operations for an output are those specified in formula (4). So it takes at most 4 additional operations to get any one double-size output bit from the single-size output bits that it depends on. It takes \(\log_2 n\) doublings to get from 1-bit to \(n\)-bit modules, so the largest number of operations needed for any one output bit is \(4 \log_2 n\).

This observation also shows that the total number of operations used in the parallel adder to calculate all the output digits is proportional to \(n \log_2 n\). This is larger than the total for a ripple-carry adder by a factor proportional to \(\log_2 n\).
Figure 1  Structure of a Double-size add1 Module.