Boolean functions and expressions

We consider Boolean functions and expressions. Circuits realizing boolean functions are the hardware basis of modern computers. The next segment of the course introduces the ideas of computer hardware and architecture.

Boolean functions

Boolean values in Scheme are `#t` and `#f` for true and false. In hardware, it is customary to use 1 for true and 0 for false, so we will shift to that convention for this topic. A Boolean function is a function that takes some fixed number of arguments, each of which may be 0 or 1, and returns a 0 or a 1 as its value.

For concreteness, we consider Boolean functions of two arguments. The domain of such a function is all ordered pairs in which each element is either 0 or 1. Listed as a set, the domain is

$$\{(0,0),(0,1),(1,0),(1,1)\}.$$

We will display Boolean functions in truth tables giving every domain element and the value of the function for that domain element. We'll use `(x * y)` to denote the "and" function, true when both of its arguments are true, and false otherwise. Its truth table is as follows.

```
x y  | (x * y)
---------------
0 0  |    0
0 1  |    0
1 0  |    0
1 1  |    1
```

The "or" function, true when either or both of its arguments are true will be denoted `(x + y)`, and has the following truth table.

```
x y  | (x + y)
---------------
0 0  |    0
0 1  |    1
1 0  |    1
1 1  |    1
```

We'll use the notation `x'` for the "not" function, which has only one argument and is true when that argument is false and false when that argument is true. Its truth table is as follows.

```
x |    x'
---------------
0 |    1
1 |    0
```

Note that this truth table has only two lines because it displays the values of a function with only one argument. Here is another truth table.

```
x y  | (x XOR y)
---------------
0 0  |    0
0 1  |    1
1 0  |    1
1 1  |    0
```

This function is the "exclusive or", which is true when exactly one of its two arguments is true and the other is false. Note that it differs from `(x + y)` when both `x` and `y` are 1.

How many two-argument Boolean functions are there? If we consider the possible tables of values, we have:

```
x y  | (x OP y)
---------------
0 0  |    ?
0 1  |    ?
1 0  |    ?
1 1  |    ?
```

where each of the `?`'s can be replaced (independently) by a 0 or a 1. Thus, we have 2 choices for the first `?`, 2 choices for the second `?`, 2 choices for the third `?`, and 2 choices for the fourth `?`, or

$$2 \cdot 2 \cdot 2 \cdot 2 = 16$$

possible choices of ways to fill in the value column of the table. Each possible choice gives us a different function, so there are exactly 16 different Boolean functions of two arguments.

How many Boolean functions of one argument are there? Just 4, listed here:

```
x  |  0     x  |  1     x  |  x     x  |   x'
-------     -------     -------     -----------
0  |  0     0  |  1     0  |  0     0  |   1
1  |  0     1  |  1     1  |  1     1  |   0
```

The first one is the constant 0 function, the second one is the constant 1 function, the third one is the identity, and the fourth one is negation.

How many Boolean functions of n arguments are there? There are 2^n elements in the domain, because each of the n arguments takes on value 0 or 1 independently, so we multiply n 2's together to get

$$2 \cdot 2 \cdots 2 = 2^n$$

different lines in its truth-table. Then, for each line of the truth-table, we can fill in the value of the Boolean function on that domain element as either 0 or 1, so we multiply 2^n 2's together to get

$$2^{2^n}$$

different possible Boolean functions of n arguments.

This is a very rapidly growing number.

$$\begin{aligned} 2^{2^1} &= 4 \\ 2^{2^2} &= 16 \\ 2^{2^3} &= 256 \\ 2^{2^4} &= 65536 \end{aligned}$$

How can we possibly deal with the more than 65,536 possible Boolean functions of 4 arguments? The good news is that most of them aren't interesting to us. The other good news is that if we allow ourselves to combine Boolean functions of one and two arguments in certain ways, we can actually get *all* the Boolean functions of any number of arguments.

There are many notations in use for the common Boolean functions. Alternative notations for x' include $\neg x$, and $\overline{x}$. Alternative notations for `(x * y)` include $x\&y$, $x \wedge y$, $x \cdot y$, and xy. Alternative notations for `(x + y)` include $x \vee y$. `(x XOR y)` is often written: $x \oplus y$. A common convention is: $\overline{x}$, $x \cdot y$, and $x + y$, for not, and, or.

Boolean expressions

We may combine constants, variables, and operators to get more complicated Boolean expressions. Our definition of Boolean expressions is inductive.

1. 0 and 1 are Boolean expressions
2. Variables x, y, z, x_1, y_1, z_1, ... are Boolean expressions
3. If E is a Boolean expression, then $(E)'$ is a Boolean expression, the *negation* of E.
4. If E_1 and E_2 are Boolean expressions, then $(E_1 \cdot E_2)$ is a Boolean expression, the *conjunction* of E_1 and E_2.
5. If E_1 and E_2 are Boolean expressions, then $(E_1 + E_2)$ is a Boolean expression, the *disjunction* of E_1 and E_2.

Note the similarity of this to a recursive definition of a procedure. There are base cases (the constants 0 and 1 and individual variables like `x` and `y`), and there are ways of taking previously constructed Boolean expressions and combining them into new ones.

Here are some examples of Boolean expressions according to these definitions.

$$0, 1, x, y, (x)', ((x)')', (x \cdot y), (x + y), ((x + y))', \ldots$$

Here I have been quite pedantic about including all the parentheses given in the definitions. Computers don't mind pedantic parentheses; we might like to do without some of them. In order to know how an expression like

$$x \cdot y' + z$$

is to be interpreted, we have to know how to "put back in" the missing parentheses unambiguously. To do this, we use precedence rules to say which operations take precedence over which other operations. (You might recall a PEMDAS rule from algebra, which specifies the precedence of parentheses, exponentiation, multiplication, division, addition, and subtraction.) In Boolean algebra, "not" takes precedence over "and", which in turn takes precedence over "or". Thus, using this rule to restore the parentheses to the expression above we get the following expression.

$$((x \cdot (y)') + z)$$

Now, we'd like to know how to figure out which Boolean function is denoted by a given Boolean expression. One way is to specify how to construct a truth table from and expression. For example, if the expression is $(x + y')'$, we proceed as follows. The variables in the formula are x and y, so these are the labels of the arguments to the function. We can proceed by making truth tables for the subexpressions of the expression, as in the following example.

```
x y  |    y'   (x + y')  (x + y')'
-----------------------------------
0 0  |    1       1         0
0 1  |    0       0         1
1 0  |    1       1         0
1 1  |    0       1         0
```

The final column above gives the desired values for $(x+y')'$. Note that we found the values for the y' column by applying the "not" operation to the column for y. In turn, we found the values for the $(x + y')$ column by applying the "or" operation to the columns for x and y'. Finally, we appplied the "not" operation to the values in the column for $(x + y')$ to get the column for $(x + y')'$. Proceeding in this way, we can compute a truth table for any Boolean expression.

Note that there are many (infinitely many) Boolean expressions that denote the same Boolean function. Two Boolean expressions that denote the same Boolean function are said to be "equivalent." For example, the expression $x' \cdot y$ is equivalent to $(x+y')'$, as can be confirmed by constructing the truth table for $x'y$, as follows.

```
x y  |    x'    x' * y
-----------------------
0 0  |    1        0
0 1  |    1        1
1 0  |    0        0
1 1  |    0        0
```

Boolean algebra

Equivalences between Boolean expressions can be characterized by a set of axioms for Boolean algebra. Here is a particular set of axioms. (Note that here we use the equal sign (=) to denote the equivalence of Boolean expressions.)

```
; Some axioms for Boolean algebra
; (Birkhoff and MacLane, A Survey of Modern Algebra)
; Using * for "and", + for "or", ' for "not"

; (A1)  a * a = a
; (A2)  a + a = a
; (A3)  a * b = b * a
; (A4)  a + b = b + a
; (A5)  a * (b * c) = (a * b) * c
; (A6)  a + (b + c) = (a + b) + c
; (A7)  a * (a + b) = a
; (A8)  a + (a * b) = a
; (A9)  a * (b + c) = (a * b) + (a * c)
; (A10) a + (b * c) = (a + b) * (a + c)
; (A11) 0 * a = 0
; (A12) 0 + a = a
; (A13) 1 * a = a
; (A14) 1 + a = 1
; (A15) a * a' = 0
```

```
    ; (A16) a + a' = 1
```

To check the first axiom, we can build a truth table for `a * a` and compare it with the truth table for `a`:

```
a  |  a * a
-----------
0  |    0
1  |    1
```

Since the columns for `a` and for `a * a` are the same, we see that these expressions do denote the same Boolean function. Note the two distributive laws, (A9) and (A10). We'll check the latter one, which is NOT true in the algebra of numbers.

```
a b c |  (b * c)   a + (b * c)  (a + b)   (a + c)   (a + b) * (a + c)
-------------------------------------------------------------------
0 0 0 |     0         0            0         0             0
0 0 1 |     0         0            0         1             0
0 1 0 |     0         0            1         0             0
0 1 1 |     1         1            1         1             1
1 0 0 |     0         1            1         1             1
1 0 1 |     0         1            1         1             1
1 1 0 |     0         1            1         1             1
1 1 1 |     1         1            1         1             1
```

Note that the two columns labelled `a + (b * c)` and `(a + b) * (a + c)` are the same, that is, these two expressions represent the same Boolean function. You might have noticed that the axioms seem to come in pairs, with `*` and `+` interchanged, and 0 and 1 interchanged.

What do we do with these axioms? We use them to justify steps in proofs of the equivalence of two Boolean expressions. Here is an example of a proof of this kind. It derives the axiom (A2) from other axioms, showing that we could have done without (A2) in our initial set.

```
    ; Example: proof of (A2):  x + x = x using other axioms

    ; x = 0 + x           (by A12)
    ;   = x + 0           (by A4)
    ;   = x + (xx')       (by A15)
    ;   = (x + x)(x + x') (by A10)
    ;   = (x + x) * 1     (by A16)
    ;   = 1 * (x + x)     (by A3)
    ;   = x + x           (by A13)
```

The first step (from x to $0 + x$) is justified by (A12); in fact, it is obtainable from (A12) by substituting x for a. The second step (from $0 + x$ to $x + 0$) is justified by (A4); again, if we substitute 0 for a and x for b, then the left and right hand sides of the axiom are the two expressions in the step. The third step (from $x + 0$ to $x + (xx')$) is justified by (A15). In this case, the two expressions are not substitution instances of the two sides of the axiom, but corresponding subexpressions of them are. More long-windedly, $0 = xx'$ is a substitution instance of (A15) so we may replace the 0 in $x + 0$ by xx'. Then we use the "strange" distribution law (A10), and so on, eventually proving the original expression, x, equal to the final expression, $x + x$.

What do we want from a set of axioms? Two relevant properties are soundness and completeness. Soundness means that the proofs using the axioms do not prove false assertions. In our case, the requirement is that if we can use the axioms to prove two Boolean expressions equal, the expressions really do denote the same Boolean function. This axiom system is sound. Completeness means that every possible true assertion has a proof; this is a rather rarer property in axiom systems, but the axiom system above is also complete. That is, every pair of Boolean expressions that denote the same function can be proved equal in this system.