Chapter 3:

Logical Operators

Our formulas so far have used a fixed set of logical operators: three binary operators, '|', '&', and '→', a single unary operator, '~', and two constants, 'T' and 'F', which can be viewed as **nullary** operators (operators that take zero operands—see below), and to which we will indeed refer as operators in this chapter. In this chapter we take a wider point of view of the allowed set of operators. This wider view has two aspects: on the one hand we may want to gain richness by adding more operators, and on the other hand, for the sake of minimalism and succinctness we may want to remove operators. We will explore both options, but the bottom line of this chapter will be that this does not matter much and any "reasonable" set of operators will basically give rise to the "same kind" of logic. We will thus be confident that there is no loss of generality in continuing to stick with the set of operators that we have originally chosen.

1 More Operators

Abstractly speaking, an ***n*-ary** operator, i.e., an operator that takes n operands—where n is called the **arity** of the operator—is simply a function from n-tuples of Boolean values to a Boolean value. Since there are 2^n different n-tuples of Boolean values and since each of them can get mapped to one of the two possible Boolean values, the total number of n-ary operators is 2^{2^n}.

Let us start by considering $n = 0$: nullary operators. There are two of these: one that maps the 0-length-tuple to *True* and the other that maps it to *False*. These two nullary operators are already in our language: 'T' and 'F'. Let us move to considering $n = 1$: unary operators. There are four of these. Two of them completely disregard their input and always output the same value: one of them always outputs *True* and the other always outputs *False*. In this sense they are equivalent to the nullary operators 'T' and 'F'. A third operator is the identity operator that maps *True* to *True* and *False* to *False*, so we require no notation for it. Only the fourth unary operator is interesting: it maps *True* to *False* and *False* to *True*, i.e., it is none other than our beloved negation operator '~'.

We finally get some interesting new operators when considering $n = 2$: binary operators. Indeed, there are 16 possible binary operators, including the binary operators that we already met: | (or), & (and), and → (implies).[1] We will now give names and assign symbols to four of the new ones among these:

[1] Similarly to what we observed with unary operators, eight of these 16 possible binary operators are no more than unary (or nullary) operators in disguise: two such operators disregard their input and always output the same value and correspond to the nullary operators; another two simply output one of the inputs (one always outputs its first input, the always other outputs its second input), disregarding the other input—these correspond to the identity operator; and another two simply output the negation of one of the inputs, disregarding the other—these correspond to the negation operator.

- '$\oplus$' (**xor**, short for *exclusive or*): *True* whenever exactly one of its two operands is *True* and the other is *False*.
- '$\leftrightarrow$' (**iff**, short for *if and only if*): *True* whenever either both operands are *True* or both are *False*. (This operator is sometimes also called **double implication**.)
- '$\overline{\&}$', (**nand**, short for *not and*): negation of the conjunction ("and") of its two operands. (This operator is sometimes also called the **Sheffer stroke** and denoted by '$\uparrow$' or, confusingly enough, by '|'.)
- '$\bar{|}$' (**nor**, short for *not or*): negation of the disjunction ("or") of its two operands. (This operator is sometimes also called the **Quine arrow** and denoted by '$\downarrow$'.)

The following table lists these new binary operators, the way we denote each of them in our Python code (note that most are denoted by a sequence of two or even three characters), and the truth table by which their semantics are defined:

Operator:	$\oplus$	$\leftrightarrow$	$\overline{\&}$	$\bar{\|}$
name:	xor	iff	nand	nor
Python representation:	`+`	`<->`	`-&`	`-|`
Value of '*False* operator *False*':	*False*	*True*	*True*	*True*
Value of '*False* operator *True*':	*True*	*False*	*True*	*False*
Value of '*True* operator *False*':	*True*	*False*	*True*	*False*
Value of '*True* operator *True*':	*False*	*True*	*False*	*False*

Task 1. Extend the `Formula` class from Chapter 1 to also support the four new binary operators defined above. Start by changing the function `is_binary()` in the file `propositions/syntax.py` from:

`propositions/syntax.py` (old)

```
def is_binary(string: str) -> bool:
    """Checks if the given string is a binary operator.

    Parameters:
        string: string to check.

    Returns:
        ``True`` if the given string is a binary operator, ``False`` otherwise.
    """
    return string == '&' or string == '|' or string == '->'
```

to:

`propositions/syntax.py` (new)

```
def is_binary(string: str) -> bool:
    """Checks if the given string is a binary operator.

    Parameters:
        string: string to check.

    Returns:
        ``True`` if the given string is a binary operator, ``False`` otherwise.
    """
    return string in {'&', '|',  '->', '+', '<->', '-&', '-|'}
```

Next, make the necessary changes to the method `_parse_prefix()` of class `Formula`. Finally, verify that all the methods (both instance methods and static methods) that you have implemented in this class in Chapter 1 now function properly for all of the seven (old and new) binary operators; with some luck (and good software engineering) you will require no further changes in your code beyond the above changes to `is_binary()` and `_parse_prefix()`. (As with every task, we provide tests also for the above, to help you check your implementation of this extended functionality.)

Task 2. Extend the `evaluate()` function from Chapter 2 to also support the four new operators defined above. Verify that the functions `truth_values()`, `print_truth_table()`, `is_tautology()`, `is_contradiction()`, and `is_satisfiable()` from that chapter "inherit" the support for the above operators from your changes to the `evaluate()` function.

We may, of course, decide to add operators with even higher **arity**. For example the **ternary** (taking three operands) operator **mux** (short for *multiplexer*) takes three Boolean values a, x, and y, and evaluates to x if a is *True* and to y if a is *False*. There is no semantic difficulty with adding such higher-arity operators, and the only issue to consider is which syntax to choose to represent them. A simple choice is to use a functional notation like 'mux(a,x,y)', a solution that can apply to any operator of any arity and only requires that we fix a name for the operator. Another option is to use some specific notation for ternary operators. For example, in some programming languages the notation `a?x:y` is used for the ternary mux operator. Everything that we discuss in the rest of the chapter also applies to any additional operators of any arity. As you have shown in Chapter 2, though, that any operator of any higher arity can be expressed (that is, its truth table can be synthesized) using operators of arity two or less, in order to avoid the technical details of fixing a syntax for higher-arity operators, our discussion will safely avoid operators of higher arity.

2 Substitutions

So, adding operators to our Logic turned out to be quite easy. We now start looking at doing the opposite: removing operators. The idea is very simple: if you can "express" one operator using others, then you can take any formula that uses the former operator and convert it to one that does not use it by substituting each instance of this operator with the appropriate combination of other operators. In this section we will focus on the mechanics of this type of substitution, which as we will see can be done in a completely syntactic manner.

Before we consider the task of substituting for an operator, we handle a simpler task of substituting for variable names, which is essentially a way of *composing* formulas. Suppose that we have a formula ϕ that uses some variable names $v_1, \ldots, v_n$ and a sequence of formulas $\psi_1, \ldots, \psi_n$ (over some set of other variable names), then we can compose ϕ with the ψ_is by (simultaneously) replacing in ϕ each occurrence of any v_i with the (sub)formula ψ_i, obtaining a single formula of which we can think as $\phi(\psi_1, \ldots, \psi_n)$, over the set of the variable names of the ψ_is (without the original v_is). For example, taking the formula '((x→x)|y)' and replacing the variable name 'x' with the formula '(q&r)' and the variable name 'y' with the formula 'p', we obtain the single formula '(((q&r)→(q&r))|p)'.

Task 3. Implement the missing code for the method `substitute_variables(substitution_map)` of class `Formula`, which takes a dictionary that maps some variable names to formulas, and returns the composition of the current formula with these formulas.

propositions/syntax.py

```
class Formula:
        ⋮
    def substitute_variables(self, substitution_map: Mapping[str, Formula]) -> \
            Formula:
        """Substitutes in the current formula, each variable name `v` that is a
        key in `substitution_map` with the formula `substitution_map[v]`.

        Parameters:
            substitution_map: mapping defining the substitutions to be
                performed.

        Returns:
            The formula resulting from performing all substitutions. Only
            variables name occurrences originating in the current formula are
            substituted (i.e., variable name occurrences originating in one of
            the specified substitutions are not subjected to additional
            substitutions).

        Examples:
            >>> Formula.parse('((p->p)|r)').substitute_variables(
            ...     {'p': Formula.parse('(q&r)'), 'r': Formula.parse('p')})
            (((q&r)->(q&r))|p)
        """
        for variable in substitution_map:
            assert is_variable(variable)
        # Task 3.3
```

Guidelines: When replacing occurrences of a variable name v with the formula ψ to which v is mapped by the given dictionary, you should not duplicate the formula ψ for each substitution of an occurrence of v, but rather have the *same* `Formula` object for ψ (the object from the given dictionary) be pointed to from all locations in the new formula tree that correspond to places where v appears in the original formula. For example, say that the current formula is '((p&q)|(~p&~q))', and you are asked to replace 'p' in this formula with the formula '(x→y)', to obtain '(((x→y)&q)|(~(x→y)&~q))'. While in the textual representation of the formula, each occurrence of '(x→y)' is fully spelled out, when building the `Formula` data structure for the composition, we ask you to use a single `Formula` object (the one from the given substitution map) for both occurrences of '(x→y)', by pointing to this object from both places in the returned formula where '(x→y)' appears. While this means that the returned formula data structure is no longer a tree, but a **directed acyclic graph (DAG)**, this is completely fine and causes no problems in our implementation since `Formula` objects are immutable. (So for example, we would never find ourselves modifying one occurrence of '(x→y)' in the formula and by doing so unintentionally modify the other as well.) Moreover, this sharing of sub-data-structures clearly leads to a data structure that is more memory-efficient than straight-forward parsing of the composed textual formula.[2]

[2]Conceptually, either of these data-structure representations are just representations of the composed formula, and in this book we continue to just think about the formula and ignore any such "internal sharing" in the code. For the benefit of interested readers, we do note, though, that the field of **Computational Complexity** that studies the *size* of formulas, does however care much about the distinction of whether

We are now ready to substitute formulas for operators. Suppose that we have a general substitution rule that allows us to replace '(p∗q)', where '∗' is some given operator, by some formula ϕ whose variable names are 'p' and 'q'; for example this rule may be to replace '(p&q)' with '~(~p|~q)'. Now we can take any formula of the form '$(\psi_1 * \psi_2)$' where ψ_1 and ψ_2 are arbitrary subformulas, and replace it with $\phi(\psi_1, \psi_2)$ (as defined in the beginning of this section); for example, if ψ_1 is '(x|y)' and ψ_2 is '~x', then '$(\psi_1 \& \psi_2)$' is '((x|y)&~x)', and we can replace it with '~(~(x|y)|~~x)'. Despite the fact that this substitution is completely syntactic ("copy-paste", if you wish), it should be clear that if '(p∗q)' and ϕ both have the same truth table, then it is also true that '$(\psi_1 * \psi_2)$' and $\phi(\psi_1, \psi_2)$ both have the same truth table, but this semantic discussion will be left for the next section. In this section we only ask you to perform this syntactic substitution, but in a general way, that is, finding all matching subformulas and (simultaneously) replacing them, and also handling substitution rules for several different operators at the same time.

Task 4. Implement the missing code for the method `substitute_operators(substitution_map)` of class `Formula`. This method takes as input a dictionary that maps some operators to formulas, where each of these formulas is over the variable names 'p' and 'q', where 'p' is agreed to serve as a placeholder for the first operand of the operator mapped to the formula (if the operator is unary or binary), and 'q' is agreed to serve as a placeholder for the second operand of the operator (if the operator is binary). The method returns a formula obtained from the current formula by replacing, for each operator that is a key in the given dictionary, each occurrence of this operator with the formula to which the dictionary maps this operator, where in that formula every occurrence of 'p' is replaced with the first operand of the operator in the original formula and every occurrence of 'q' is replaced with the second operand of the operator in the original formula. (In particular, if the operator is 'T' or 'F', then it is simply replaced with the formula to which the dictionary maps it, without making any substitutions in that formula.)

propositions/syntax.py

```
class Formula:
        ⋮
    def substitute_operators(self, substitution_map: Mapping[str, Formula]) -> \
            Formula:
        """Substitutes in the current formula, each constant or operator `op`
        that is a key in `substitution_map` with the formula
        `substitution_map[op]` applied to its (zero or one or two) operands,
        where the first operand is used for every occurrence of 'p' in the
        formula and the second for every occurrence of 'q'.

        Parameters:
```

sharing sub-data-structures is or is not allowed: in the language of that field, the term **formula size** refers to the size of a tree representation that does *not* allow sharing, while the term **circuit size** is used for a DAG representation that is allowed to share intermediate results. Naturally, the latter may be smaller than the former, and in fact it turns out that sometimes even significantly so. It is still a central open problem of computational complexity theory whether the gap between the formula size and the circuit size can be *super-polynomial* (grow faster than any polynomial). This is called the "$NC^1 = P$?" question (an equality here would mean that the gap cannot be super-polynomial), and it turns out to precisely capture the question of whether every computation can be significantly parallelized. This is already the second example that we see (the first was the question of efficient satisfiability, discussed in Chapter 2) of a question that may seem at first glance to be a purely theoretical question of interest mainly to logic aficionados, but in fact turns out to have far-reaching applied implications to virtually any computationally heavy task that you could imagine.

```
            substitution_map: mapping defining the substitutions to be
                performed.

        Returns:
            The formula resulting from performing all substitutions. Only
            operator occurrences originating in the current formula are
            substituted (i.e., operator occurrences originating in one of the
            specified substitutions are not subjected to additional
            substitutions).

        Examples:
            >>> Formula.parse('((x&y)&~z)').substitute_operators(
            ...     {'&': Formula.parse('~(~p|~q)')})
            ~(~~(~x|~y)|~~z)
        """
        for operator in substitution_map:
            assert is_constant(operator) or is_unary(operator) or \
                   is_binary(operator)
            assert substitution_map[operator].variables().issubset({'p', 'q'})
        # Task 3.4
```

Guidelines: You should once again be frugal in your duplications of subformulas. That is, whichever subformulas (subtrees) of the original formula that you can use in their entirety in the returned formula, you should point to instead of duplicating.

3 Complete Sets of Operators

Now that we have the mechanics of substituting for operators in place, we can proceed to the heart of the issue of "getting rid" of unnecessary logical operators. The following is a key definition that will guide us to make sure that we don't "get rid" of too many operators:

Definition (Complete Set of Operators)**.** A set of logical operators is called **complete** if every Boolean function (of any arity) can be expressed using only the operators in the set.

The notion of "expressed" here is a semantic one: there exists a formula that uses only these operators, whose truth table is the one of the given Boolean function. In this language, the main result proved in Chapter 2 was that the set of operators {'~','&','|'} is complete (since the DNF Theorem states that every Boolean function can be represented in disjunctive normal form, a form that only uses these three operators).

In this chapter we will demonstrate other sets of operators that are complete, a task that will turn out to be easier using the fact that we already have one such set in our toolbox. The basic idea is simple: once we know that we can represent a Boolean function using one set of operators, if we can in turn represent each of the operators in this set using a second set of operators, then using substitution we can get a representation of the original Boolean function using (only) the latter set of operators.

To get used to this type of substitution, your next task is to take an arbitrary formula that uses any of the extended family of operators introduced so far and convert it to a formula that uses only the set of operators {'~','&','|'}. Of course, from Chapter 2 we already know that this is possible as there exists a DNF formula that has the same truth table as the original formula. Indeed, one semantic way to achieve this would be to calculate the truth table of the given formula, and then synthesize a formula in DNF that has the same truth table:

```
    synthesize(formula.variables(),
               truth_values(formula(), all_models(formula.variables())))
```

The problem with this approach is that it takes exponential time. Therefore, in the next task, you are asked to implement the same functionality in a syntactic rather than semantic way that, beyond being a good exercise, has the additional advantage of being far more efficient.

The file `propositions/operators.py` contains a list of functions that you are asked to implement in the remainder of this chapter.

Task 5. Implement the missing code for the function `to_not_and_or(formula)`, which returns a formula that is equivalent to the given formula (i.e., a formula having the same truth table), but may only use the operators *not*, *and*, and *or* in it.

propositions/operators.py

```
def to_not_and_or(formula: Formula) -> Formula:
    """Syntactically converts the given formula to an equivalent formula that
    contains no constants or operators beyond '~', '&', and '|'.

    Parameters:
        formula: formula to convert.

    Returns:
        A formula that has the same truth table as the given formula, but
        contains no constants or operators beyond '~', '&', and '|'.
    """
    # Task 3.5
```

Hint: Substitute (using your solution for Task 4) each other operator with an equivalent formula that uses only the allowed operators.

We are now ready for the crux of this chapter: showing that a few other sets of operators are complete. As we know that the set {'~','&','|'} is complete, to show that another set of operators is complete, it suffices to represent only each of the three operators '~', '&', and '|' using operators from that latter set.

Task 6. Implement the missing code for the four functions `to_not_and(formula)`, `to_nand(formula)`, `to_implies_not(formula)`, and `to_implies_false(formula)`, each of which returns a formula that is equivalent to the given formula, but may only use the operators listed in the name of that function.

propositions/operators.py

```
def to_not_and(formula: Formula) -> Formula:
    """Syntactically converts the given formula to an equivalent formula that
    contains no constants or operators beyond '~' and '&'.

    Parameters:
        formula: formula to convert.

    Returns:
        A formula that has the same truth table as the given formula, but
        contains no constants or operators beyond '~' and '&'.
    """
    # Task 3.6a

def to_nand(formula: Formula) -> Formula:
    """Syntactically converts the given formula to an equivalent formula that
```

```
    contains no constants or operators beyond '-&'.

    Parameters:
        formula: formula to convert.

    Returns:
        A formula that has the same truth table as the given formula, but
        contains no constants or operators beyond '-&'.
    """
    # Task 3.6b

def to_implies_not(formula: Formula) -> Formula:
    """Syntactically converts the given formula to an equivalent formula that
    contains no constants or operators beyond '->' and '~'.

    Parameters:
        formula: formula to convert.

    Returns:
        A formula that has the same truth table as the given formula, but
        contains no constants or operators beyond '->' and '~'.
    """
    # Task 3.6c

def to_implies_false(formula: Formula) -> Formula:
    """Syntactically converts the given formula to an equivalent formula that
    contains no constants or operators beyond '->' and 'F'.

    Parameters:
        formula: formula to convert.

    Returns:
        A formula that has the same truth table as the given formula, but
        contains no constants or operators beyond '->' and 'F'.
    """
    # Task 3.6d
```

Your solution to Task 6 proves the following theorem:

Theorem. *The following four sets of operators are each (separately) a complete set of operators:*

a. *The set containing only* '$\sim$' *(not) and* '$\&$' *(and)*

b. *The singleton set containing only* '$\overline{\&}$' *(nand)*

c. *The set containing only* '$\rightarrow$' *(implies) and* '$\sim$' *(not)*

d. *The set containing only* '$\rightarrow$' *(implies) and* 'F' *(False)*

This is not an exhaustive list of complete sets of operators, of course: any superset of a complete set of operators is itself complete. There are also a few other minimal complete sets of operators; in particular the singleton set of '$\overline{|}$' (nor) by itself is also a complete set. There are also minimal complete sets of operators that contain operators of arity higher than 2, such as the set containing only *mux* and '$\sim$' (not). Nevertheless, the above are arguably the most interesting complete sets of operators.

Once we know that a set of operators is complete, we can limit ourselves to using only this set, knowing that nothing of the essence is lost in the sense that still every truth table can be expressed using some formula in our logic. This will be our approach in the rest of our discussion of Propositional Logic, where we will restrict ourselves to the complete set that only contains '$\rightarrow$' (implies) and '$\sim$' (not), shunning not only the new operators introduced in this chapter but also the familiar disjunction and conjunction binary operators '&' and '|', as well as the constants 'T' and 'F'. This will naturally simplify all technicalities without affecting anything substantial.

4 Proving Incompleteness

It is natural to wonder which other sets of operators are complete. For example, as we have seen that the set containing only '$\rightarrow$' and 'F' is complete, it is natural to ask whether the set containing only '$\rightarrow$' and 'T' is also complete, or maybe even the singleton set containing only '$\rightarrow$'. Similarly, as we have seen that the set containing only '&' and '$\sim$' is complete, it is natural to ask whether the set containing only '&' and '|' is also complete, etc. In this section we will prove negative answers to these and other questions. As expected, since these are negative results, they will not be proven by programming, but rather by classical mathematical arguments.

How can one go about proving that there is *no way* to represent some function using a given set of operators? Well, one very useful proof technique is actually quite simple: identify a property of Boolean functions that must hold for all Boolean functions that can be represented using only this given set of operators, and then exhibit some Boolean function that does not have this property. As we will see, though, pinpointing the right property to use is somewhat of an art. The following theorem, which answers (among others) the two specific questions raised above, has an especially simple proof of this form:

Theorem. *The set of operators containing* '&', '|', '$\rightarrow$', 'T', '$\leftrightarrow$', *and* 'mux'[3] *is not complete. In particular it cannot represent the constant function with value False, or the negation function.*

Proof. The property that all functions composed only of the operators '&', '|', '$\rightarrow$', 'T', '$\leftrightarrow$', and 'mux' share is the following:

Definition (T-Preservation)**.** A Boolean function f is called **T-preserving** if whenever the values of all its input values are *True*, the function also outputs *True*.

To see that every function composed of these operators is T-preserving, we need to note two things: first that all six operators in the list have this property (this can be seen directly by simply plugging in all-*True* values into each of these and evaluating), and second that composing functions with this property also maintains the property. The second point is true since if you take T-preserving functions f and g and look at their composition $f(g(\cdot),\cdot)$ (where $\cdot$ denotes an arbitrary set of values), then if all input values are *True* then so is $g(\cdot)$ and then all input values to f are *True* so thus is also $f(\cdot)$. Formally, this is then proved by induction on the length of the formula.[4]

[3]Recall that 'mux' is the ternary operator such that '$\text{mux}(x,y,z)$' evaluates to the same value as '$((x\&y)|(\sim x\&z))$'.

[4] Here is the formalistic argument: We are given a formula composed only of these operators and need to show that when the input values are all *True* the formula also evaluates to *True*. The base case is if the formula is either a variable name or the nullary operator 'T', and in these two cases clearly the value

We next need to identify a function that does not have this property, and conclude that it cannot be composed of the above operators (so they do not form a complete set). As this function we can choose, e.g., the constant function that always has value *False*, or the negation function, both of which are not T-preserving. □

The above theorem provides a rather satisfactory answer to the question of why '→' and 'T' do not comprise a complete set of operators, and while it also shows that '&' and '|' (and 'T') do not comprise a complete set, the really interesting question is whether the set containing '&' and '|' in addition to both constants 'T' and 'F' is complete. As we will now prove, it turns out that it is not the case:

Theorem. *The set of operators containing* '&', '|', 'T', *and* 'F' *is not complete. In particular it cannot represent the negation function.*

Proof. The key property here will be that of **monotonicity**:

Definition (Monotonicity)**.** A Boolean function f is called **monotone** if whenever for some input (that is, for some tuple of input values) f evaluates to *True*, it also continues to evaluate to *True* even if we modify the input by changing some of the input values from *False* to *True*.[5]

For a monotone function, changing some of the input values from *False* to *True* can only lead to the function's value changing from *False* to *True* (or not changing at all) and never to the value changing in the opposite direction, from *True* to *False*. It is clear that '&', '|', 'T' and 'F' are all monotone, and that composition maintains monotonicity: if f and g are monotone, then looking at the effect of changing the value of some input from *False* to *True* on the value of $f(g(\cdot),\cdot))$, we observe that since $g(\cdot)$ can only change from *False* to *True* then this is true for all of the input values to f and thus also for its output.

In contrast, the negation function is not monotone, since when its input changes from *False* to *True* its output changes form *True* to *False*. We conclude that the negation function cannot be composed of the above operators, and so they do not form a complete set. □

All of the sets that we have so far proven to be incomplete do not contain the negation operator (and moreover, negation cannot be composed of any of these sets, and in fact adding the negation operator to any of these sets yields a complete set of operators). We will now show that a different set, which contains also the negation operator, is incomplete. The property that we will use to achieve this is more subtle, and to pinpoint it will require somewhat more thought.

Theorem. *The set of operators containing* 'T', 'F', '∼', '⊕', *and* '↔' *is not complete. In particular it cannot represent the conjunction ("and") function or the disjunction ("or") function.*

of the function is *True*. Otherwise, we apply the induction hypothesis to each of the two (or three in the case of 'mux') operands of the root operator to obtain that they evaluate to *True*, which then implies that the value of the whole formula is *True*.

[5]The name "monotonicity" comes from viewing these functions as functions from vectors of zeros and ones to zeros and ones, with *False* corresponding to zero and *True* corresponding to one. A function is monotone in the above sense if and only if when viewed this way it is monotone in each of its coordinates in the standard numeric sense.

Proof. The key property here, **affineness**, may seem at first glance to have been hard to come up with, but as we will discuss, it in fact can be seen to be quite natural from an algebraic point of view that views {*False*, *True*} as a field with two elements.

Definition (Affineness)**.** A Boolean function f is called **affine** if it is equivalent (in the sense of having the same truth table) either to the *xor* of a subset[6] of its inputs[7] or to the negation of the *xor* of a subset of its inputs.[8]

It is easy to see that each of the operators 'T', 'F', '~', and '$\oplus$' is affine since it can be written as a *xor* of a subset of its inputs, or as the negation of such a *xor*.

Affine functions are precisely those functions that have the property that each of their inputs either never affects their output (this is the case when this input is *not* one of the *xor*ed inputs), or always affects their output in the sense that regardless of the other inputs, changing this input changes the output (this is the case when this input is one of the *xor*ed inputs). We can therefore immediately conclude that the conjunction and disjunction functions are not affine: for example, for the conjunction function we note that if the first input is *False* then the second input does *not* affect the output but if the first input is *True* then the second input *does* affect the output, so it is not true that the second input either never affects the output regardless of the other inputs or always affects the output regardless of the other inputs. It therefore remains to check that any composition of affine functions is affine.

There are two ways to complete the proof. One of them is via the algebraic point of view that led us to consider affineness to begin with: it is a well-known fact from linear algebra that the composition of affine transformations is itself an affine transformation. For those who are unfamiliar with this property of affine transformations, the second way to complete this proof is that it is easy to reprove this property in our special case of Boolean functions: if we replace one of the inputs of an affine function by an affine function, then any input that is *xor*ed with itself simply "cancels out," as do two negations, leaving us with an affine function. □

We have seen that the set containing *not* and *and* is complete, and so is the singleton set containing only *nand*. We have seen that the set containing *not* and *or* is complete, and so is the singleton set containing only *nor*. We have seen that the set containing *not* and *implies* is complete. It is natural to therefore ask whether the singleton set containing only what one may be tempted to analogously call "*nimplies*", **non-implication**, is complete. That is, whether the singleton set containing only the binary operator '$\not\Rightarrow$', whose semantics we define by defining its output to be the negation of the output of the *implies* operator on the same inputs,[9] is complete. This turns out not to be true,

[6]Like the operators '&' and '|' (but unlike the operator '$\rightarrow$', for example), the *xor* operator is **commutative** and **associative**: if we *xor* several values, the order in which we *xor* them does not alter the output, so for any given set of values, their *xor* is well defined with no need to specify any ordering, placement of parentheses, etc.

[7]By convention, we consider the *xor* of an empty set of inputs to be *False*.

[8]The name "affineness" comes from viewing these functions (somewhat similarly to the explanation of monotonicity above) as mappings from the vector space over the two-element field (note that *xor* is simply the addition operation of this field), to that field. A function is affine in the above sense if and only if when viewed this way it is an affine transformation in the standard linear-algebraic sense, i.e., a linear transformation plus a constant (since over the two-element field, adding the constant one is equivalent to negating the value).

[9]Attentive readers may notice that this is the last binary operator that we have not discussed so far. Recall that there are 16 possible binary operators, eight of which are unary (or nullary) operators in disguise and eight "really are" binary operators (in the sense that none of them always disregards one or

and you are invited to try and prove this yourself, using the same technique of finding a property that every Boolean function that can be built from instances of this operator satisfies, and then exhibiting a Boolean function that does not satisfy this property. **Hint:** Search for a property, inspired by one of the properties above, that is common to '$\not\Rightarrow$', 'F', '&', '|', '$\oplus$', and 'mux', and to every function constructed using only them, that would enable you to show that the set of operators containing all of these is not complete (and hence, every subset of this set, including the singleton set containing only the operator '$\not\Rightarrow$', is not complete).

more of its inputs). Of the latter, we already defined three in Chapter 2 and four more in the beginning of this chapter, and the last remaining one is this so-called "nimplies" operator.