General Information and Communication Technology 1

Course Number 320211

Jacobs University Bremen

Herbert Jaeger

Second module of this course:

Boolean logic

and

Some elements of computational complexity

1 Boolean logic

1.1 What it is and what it is good for in CS

There are two views on Boolean logic (BL) which are relevant for CS. In informal terms they can be described as follows:

1. In digital computers, information is coded and processed just by "zeros" and "ones" – concretely, by certain voltages in the electronic circuitry which are switching (veeeery fast!) between values of 0 Volt (corresponding to "zero") and some system-specific positive voltage V^+, for instance $V^+ = 5$ Volts (corresponding to "one"). $V^+ = 5$ Volts was a standard in old digital circuits of the first personal computers, today the voltages are mostly lower. You know that much of the computation in a computer is done in a particular microchip called the *central processing unit* (CPU). The CPU contains thousands (or millions) of transistor-based electronic mini-circuits called *logic gates*. At the output end of such a gate one can measure a voltage which is approximately either 0 or V^+. The specs of a computer contain a statement like "CPU speed is 2.5 GHz". GHz stands for Giga-Hertz, a measure of frequency. 1 Hertz means a frequency of "1 event per second", 2.5 GHz means "2.5×10^9 events per second". This means that 2.5×10^9 times in a second, all the gates may switch their output voltage from 0 to V^+ or vice versa (or stay on the same level). Boolean logic is the elementary mathematical formalism to describe (and design) such super-complex, super-fast switching circuits. And it's clear that one absolutely needs some mathematical theory here: it would be plainly impossible to describe, design or "program" such hugely complex systems without a suitable mathematical abstraction of all the things that may happen when one wires together zillions of little gating circuits. Boolean logic is thus the theory of the "low end" of computation where it is closest to hardware, of its "micro-mechanics". BL here describes how complex switching circuits (e.g., entire CPUs) can be decomposed into, or built from, some elementary logic gates.
2. But Boolean logic is also the mathematical theory which forms the basis of what one might call the "highest" end of computation: Artificial Intelligence. A chess-playing computer, or a robot control program, must be able to "think logically": *if I move this pawn one forward, my opponent may beat my queen in two moves*, or *in order to place that cup on the table, I* [the robot] *must first grasp it with my gripper*. In its elementary versions, such logical thinking is about ascertaining facts and planning the consequences of actions – factual statements may change from `True` to `False` or vice versa (or their *truth value* may stay the same) after an action. On this high level of knowledge processing, the "zeros" and "ones" of Boolean logic turn into `True` or `False`. BL here describes how complex logical arguments (e.g., a robot planning to assemble a car from parts) can become decomposed into, or composed from, a sequence of elementary logical arguments.

So, in sum, Boolean logic is the mathematical framework for describing anything that is *binary*, that is, anything which can have only two values – for instance 0 and V^+, or

`False` and `True`. In textbooks which explain BL one finds mostly two notations for these two events. They are either denoted by "0" and "1", or by "F" and "T". I will use the first convention.

These two values are called *truth values* in BL. This terminology comes from the "high-end" view on BL: a factual statement (a *proposition*) can be either `True` or `False`. Sometimes BL is also called *propositional logic*.

In this course I present the basic concepts of Boolean logic in some detail. If you want a more comprehensive but also more compressed treatment I can recommend the first chapters in the following textbook:

Uwe Schoening: Logic for Computer Scientists (Progress in Computer Science and Applied Logic, Vol 8), (Birkhauser). The IRC has some copies of this book: QA9 .S363 1989. You don't need to consult that book however, this handout is enough.

1.2 Elementary Boolean operations

BL describes objects which can take one of two values, or switch between these two values. In the "low-end" electronic circuit use of BL, these objects are the outputs of the logic gate circuits, which can take the values 0 or V^+. In the "high-end" Artificial Intelligence use of BL, these objects are propositions, which can be `True` or `False`. In the mathematical formalism of BL, which abstracts away from the final concrete use, these objects are called *Boolean variables*. We use the symbols $X_1, X_2, X_3, \ldots$ for Boolean variables. The set $\boldsymbol{X} = \{X_1, X_2, X_3, \ldots \}$ is the set of all Boolean variables – there are no others besides these. Sometimes, however, for convenience (when we are tired of writing subscript indices) we will also use X, Y, Z, ... to denote Boolean variables. Be aware that this is a only a convenience thing: even if we write Y, we actually mean one of the $X_1, X_2, X_3, \ldots$. Abstractly speaking, a Boolean variable X is a formal object which can have the value 0 or the value 1.

Example: When analyzing a complex electronic circuit, an electronic contact point (the output end of a logical gate circuit embedded in the complex circuit) where we can currently measure a voltage (and obtain a measurement of 0 or V^+) can be formalized as a Boolean variable X_i, taking Boolean values 0 or 1.

Example: When a robot inspects its environment, planning an action sequence, a proposition like *the cup is on the table* can currently be true or false. Again, this proposition can be formalized as a Boolean variable X_i which can take Boolean values 0 or 1.

Whether a Boolean variable is currently taking the value 0 or the value 1 is expressed through an *interpretation* (also called *truth value assignment*). Formally, an interpretation is a function

(1) $\mathcal{I}: \boldsymbol{D} \rightarrow \{0, 1\}$,

where $\boldsymbol{D}$ is a subset of $\boldsymbol{X}$ (in any real application we will not need all the infinitely many $X_1, X_2, X_3, \ldots$ but only a finite subset of them).

Example: In a CPU with 10000 gates we will choose $\boldsymbol{D} = \{X_1, X_2, \ldots, X_{10000}\}$ and if we measure the 10000 voltages at some time t, we have an interpretation $\mathcal{I}_t$ where $\mathcal{I}_t(X_i) = 0$ if the voltage measured at the i-th gate is 0 and $\mathcal{I}_t(X_i) = 1$ if the voltage measured at the i-th gate is V^+.

Example: our table-setting robot may have to think about 20 different situational circumstances like *the cup is on the floor*, *the cup is held by the gripper*, *the gripper is positioned above the table* etc., which can be modelled by $\boldsymbol{D} = \{X_1, X_2, \ldots, X_{20}\}$. In any concrete situation S, every one of these 20 propositions is either true or false. If the proposition X_i is true in situation S we would have $\mathcal{I}_S(X_i) = 1$.

The main purpose of Boolean logic is to describe (or design) interdependencies between Boolean variables.

Example: a particular electronic gate with output X_i in a CPU may get two input signals X_j, X_k from two other gates. The gate X_i responds lawfully to these inputs. For instance, the gate X_i may be constructed such that whenever at least one of the two feeding signals is V^+, then the output of X_i is V^+, too. We will formalize this in Boolean logic as "if $\mathcal{I}_t(X_j) = 1$ OR $\mathcal{I}_t(X_k) = 1$, then $\mathcal{I}_t(X_i) = 1$" (this holds for any interpretation $\mathcal{I}_t$). Since this holds for all interpretations, we can also write $X_i = X_j$ OR X_k.

Example: in our little robot world, the gripper may be designed such that it is either opened or closed. Then if $X_1 \equiv$ *the gripper is opened* and $X_2 \equiv$ *the gripper is closed*, we could describe that the two propositions are mutually exclusive as $X_1 =$ NOT X_2.

Traditionally (a tradition that goes back to ancient Greece!) one uses three elementary operations as building blocks for BL: the AND, OR, and NOT operations. Let us consider AND first. AND describes that a Boolean variable Z depends on two other Boolean variables X, Y such that Z is 1 if and only if both X AND Y are 1. Using the customary symbol $\wedge$ for AND, this is specified in a *truth table* as follows:

X	Y	$Z = X \wedge Y$
0	0	0
0	1	0
1	0	0
1	1	1

The AND operation can be regarded as a *function* which assigns to every pair (x, y) of truth values a new truth value $z = x \wedge y$. Notice that we use lowercase letters $x, y, \ldots$ to denote truth values and uppercase letters $X, Y, \ldots$ to denote the variables that can take those values. In mathematical terminology, the set of such pairs of values from $\{0, 1\}$ is called the *cross product* of $\{0, 1\}$ with itself, written as $\{0, 1\} \times \{0, 1\}$. The AND operation can thus mathematically be understood as a function from truth value pairs to truth values:

(2) $\wedge: \{0, 1\} \times \{0, 1\} \to \{0, 1\}$,

with the values for each argument pair given in the table above.

Notice that each row in the truth table corresponds to an interpretation. For instance, the first row in the table above corresponds to the interpretation $\mathcal{I}_{\text{firstrow}}: \{X, Y\} \rightarrow \{0, 1\}$, which maps X to 0 and Y to 0. That is, $\mathcal{I}_{\text{firstrow}}(X) = 0$ and $\mathcal{I}_{\text{firstrow}}(Y) = 0$.

Similarly, the OR operation (denoted by $\vee$) has the intuitive interpretation "Z is 1 if and only if at least one of X or Y is 1", as detailed in the truth table

X	Y	$Z = X \vee Y$
0	0	0
0	1	1
1	0	1
1	1	1

Again, OR can be seen as a function of the type $\vee: \{0, 1\} \times \{0, 1\} \rightarrow \{0, 1\}$.

Finally, the NOT operation flips truth values, turning 1 into 0 and vice versa. Using the traditional symbol $\neg$ for NOT, this turns out to be a function of the type $\neg: \{0, 1\} \rightarrow \{0, 1\}$ with the following truth table:

X	$Z = \neg X$
0	1
1	0

These are the three Boolean functions which are standardly used in mathematics and logic as basic building blocks for Boolean logic. The AND, OR, and NOT functions are also known as *conjunction*, *disjunction*, and *negation*, respectively.

Some other elementary Boolean functions are of interest too, either because they correspond to intuitive logic operations or because they are often hard-wired as elementary logic gates in microchips. Here are some examples:

The implication ("if – then"), symbolized by $\rightarrow$:

X	Y	$Z = X \rightarrow Y$
0	0	1
0	1	1
1	0	0
1	1	1

This logical implication seems often a little counter-intuitive to logic beginners. Consider the claim *if cats are dogs, then the sun shines*. For ordinary mortals, this makes little sense. However, for logicians this not only makes sense, but it is true: cats are not dogs, so the precondition *cats are dogs* is false (i.e., it is 0). Then regardless of whether the sun shines or not (that is regardless of whether $\mathcal{I}$(*the sun shines*) = 0 or = 1), the truth table returns 1. A logical implication (→) statement is only false if its precondition is true, but the consequence that it asserts is false.

The logical equivalence ("if and only if"), symbolized by ↔:

X	Y	$Z = X \leftrightarrow Y$
0	0	1
0	1	0
1	0	0
1	1	1

The exclusive or ("either-or"), abbreviated XOR:

X	Y	$Z =$ XOR(X,Y)
0	0	0
0	1	1
1	0	1
1	1	0

The not-AND operation, abbreviated NAND:

X	Y	$Z =$ NAND(X,Y)
0	0	1
0	1	1
1	0	1
1	1	0

The NAND operator is also called *Sheffer stroke* and is sometimes represented by the symbol ↑.

Altogether there are $2^4 = 16$ different Boolean functions which transform pairs of truth values into truth values (why?). Not all of them have names or are standardly hard-wired in microchips or are used in AI applications of logic. You can find an instructive overview at http://en.wikipedia.org/wiki/Logic_gate.

1.3 Composite Boolean functions and Boolean expressions

Our elementary Boolean functions ¬, ∧, ∨ can be composed to give more complex functions. A simple example is

(3) $\varphi(X, Y) := \neg (X \wedge Y)$.

Here we defined a new function φ: $\{0, 1\} \times \{0, 1\} \to \{0, 1\}$ by the composite operation "first compute the AND of X and Y, then apply the NOT on what you got from the AND". When you think about it you will find that φ is the NAND function. We generally use Greek letters φ, ψ,... to denote Boolean functions.

Here is a more involved example:

(4) $\psi(X, Y, Z) := (\neg (X \wedge Y) \vee (Z \wedge Y))$.

This function assigns a truth value to every *triple* of truth values, that is, this ψ is of the type ψ: $\{0, 1\} \times \{0, 1\} \times \{0, 1\} \rightarrow \{0, 1\}$, for which we also write ψ: $\{0, 1\}^3 \rightarrow \{0, 1\}$. The truth table for this ψ looks like this:

X	Y	Z	$V = \psi(X, Y, Z)$
0	0	0	1
0	0	1	...
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

I only filled the result for the first combination of arguments $X = Y = Z = 0$. I computed it by computing partial results "from the inside outwards". Concretely, I first computed $(X \wedge Y) = (0 \wedge 0) = 0$ and $(Z \wedge Y) = (0 \wedge 0) = 0$. Then I applied the NOT on the result of first subformula: $\neg (X \wedge Y) = \neg 0 = 1$. Then I applied the OR on the two subformulas $\neg (X \wedge Y)$ and $(Z \wedge Y)$: $(\neg (X \wedge Y) \vee (Z \wedge Y)) = 1 \vee 0 = 1$. Filling in the remaining rows is left as an exercise.

Abstracting from these examples, we define Boolean functions in general as follows:

Definition 1.1 A Boolean function φ is any function of the type φ: $\{0, 1\}^k \rightarrow \{0, 1\}$, where $k \geq 0$.

This deserves some comments. When $k = 1$, a Boolean function assigns truth values to truth values (like the NOT function). When $k = 2, 3, ...$ it assigns truth values to pairs, triples, ... of truth values. The number k of arguments is called the *arity* of the function. What about the arity $k = 0$? Well, for mathematicians this is not extraordinary. A Boolean function with arity $k = 0$ function assigns truth values to... nothing! That is, it depends on nothing and simply returns a truth value. There are two such functions, one always returning 0 and the other always returning 1. We simply *identify* these two arity-0 functions with the *truth value constants* 0 and 1.

It is easy to see that the truth table of a Boolean function with arity k has 2^k rows. As k increases, truth tables clearly become an infeasible way to specify Boolean functions. Specifically, consider a CPU chip. It has k input wires (called "pins" by the chip people) and m output pins. Let us model the k input pins by k Boolean variables X_1, X_2, ... , X_k. If we fix one of the m output pins, calling it Z, and ignore the other outputs, the chip realizes in hardware a Boolean function χ: $\{0, 1\}^k \rightarrow \{0, 1\}$. In real CPU chips, k can easily be as large as 128. The truth table for the chip (only considering a single output pin) would have 2^{128} rows, in the order of the number of atoms in the universe. Chip designers need more efficient methods to specify Boolean functions than by way of truth tables...

... but that seems within reach. Consider again the ψ function from above. We actually specified it in two ways: first, by equation (4), and second, by the truth table. The first way of specifying it clearly is more compact. Formulas like (4) are the preferred way to specify Boolean functions, not truth tables. We now give a formal definition of a these important and helpful objects. The definition uses a common trick in theoretical CS and proceeds by *induction*.

Definition 1.2 (*Boolean formulas*, also known as *Boolean expressions*).

Basis of inductive definition:

1a. Every Boolean variable X_i is a Boolean formula.
1b. The two Boolean constants 0 and 1 are Boolean formulas.

Induction step:

2a. If φ and ψ are Boolean formulas, then $(\varphi \wedge \psi)$ is a Boolean formula.
2b. If φ and ψ are Boolean formulas, then $(\varphi \vee \psi)$ is a Boolean formula.
2c. If φ is a Boolean formula, then $\neg \varphi$ is a Boolean formula.

Again we are relaxed about what we may use as Boolean variables. While in theory only $X_1, X_2, \ldots$ qualify for step 1a, in practice we also use $X, Y, \ldots$.

This is all we need to verify whether a particular writeup of symbols qualifies as a Boolean formula. Consider again the example $(\neg (X \wedge Y) \vee (Z \wedge Y))$. This symbol string can be ascertained to be a valid Boolean formula by going step-wise from the inside to the outside:

- X, Y, Z are Boolean formulas (note our relaxed attitude w.r.t. Boolean variables) by rule 1a.
- $(X \wedge Y)$ and $(Z \wedge Y)$ are Boolean formulas by rule 2a.
- $\neg (X \wedge Y)$ is a Boolean formula by rule 2c.
- Hence, $(\neg (X \wedge Y) \vee (Z \wedge Y))$ is a correctly written Boolean formula by rule 2b.

The rigorous Definition 1.2 uses lots of brackets to make the nested structure of a Boolean formula clear. In practice one uses some bracket-saving conventions to have fewer of them, preventing bracket cluttering. The most important convention is to write $(X \wedge Y \wedge Z)$ instead of $(X \wedge (Y \wedge Z))$, or $(X \vee Y \vee Z \vee W)$ for $(X \vee (Y \vee (Z \vee W)))$, etc (repeated $\wedge$'s or repeated $\vee$'s may be merged).

Furthermore, because logical implication and equivalence occur frequently in applications, by convention one may write in shortcut notation

(5) $(X \to Y)$ for $(\neg X \vee Y)$ and
(6) $(X \leftrightarrow Y)$ for $((\neg X \vee Y) \wedge (\neg Y \vee X))$.

Now we know how we may write down Boolean formulas, and we know it *precisely*. In CS/math terminology, we have fixed the *syntax* of Boolean formulas. In a sense, Definition 1.2 could be called the grammar of Boolean formulas. I said above that Boolean formulas are compact tools for specifying Boolean functions. When I first introduced Boolean formulas for specifying Boolean functions in the example from equation (4) above, I used plain English to explain by an intuitive argument how that formula $\psi(X, Y, Z) = (\neg (X \wedge Y) \vee (Z \wedge Y))$ gives rise to a truth table. We proceed to make this connection precise. The following definition specifies precisely, again in an inductive way, what is the Boolean function associated with a Boolean formula. The definition thus defines what a Boolean formula "means", or "denotes", – it is the definition of the *semantics* of Boolean formulas.

Definition 1.3 (Semantics of Boolean formulas). Let $\boldsymbol{D} \subseteq \boldsymbol{X}$ be a set of Boolean variables and $\mathcal{I}: \boldsymbol{D} \to \{0, 1\}$ an interpretation. Let $\Phi(\boldsymbol{D})$ be the set of all Boolean formulas which contain only Boolean variables that are in $\boldsymbol{D}$. We define a generalized version of an interpretation $\mathcal{I}^*: \Phi(\boldsymbol{D}) \to \{0, 1\}$ again by induction, repeating the syntactical construction of Boolean formulas:

Basis of inductive definition:

1a. For every Boolean variable $X \in \boldsymbol{D}$, $\mathcal{I}^*(X) = \mathcal{I}(X)$.
1b. For the two Boolean constants 0 and 1, we set $\mathcal{I}^*(0) = 0$ and $\mathcal{I}^*(1) = 1$.

Induction step:

2a. If φ and ψ are in $\Phi(\boldsymbol{D})$, then

$$\mathcal{I}^*\left((\varphi \wedge \psi)\right) = \begin{cases} 1 & \textit{if } \mathcal{I}^*(\varphi) = 1 \quad \textit{and} \quad \mathcal{I}^*(\psi) = 1 \\ 0, & \textit{all other cases} \end{cases}$$

2b. If φ and ψ are in $\Phi(\boldsymbol{D})$, then

$$\mathcal{I}^*\left((\varphi \vee \psi)\right) = \begin{cases} 1 & \textit{if } \mathcal{I}^*(\varphi) = 1 \quad \textit{or} \quad \mathcal{I}^*(\psi) = 1 \\ 0, & \textit{all other cases} \end{cases}$$

2c. If φ is in $\Phi(\boldsymbol{D})$, then

$$\mathcal{I}^*\left(\neg \varphi\right) = \begin{cases} 1 & \textit{if } \mathcal{I}^*(\varphi) = 0 \\ 0 & \textit{if } \mathcal{I}^*(\varphi) = 1 \end{cases}$$

Because this generalized interpretation $\mathcal{I}^*$ is the same as $\mathcal{I}$ on the Boolean variables $X \in \boldsymbol{D}$, we say that $\mathcal{I}^*$ *extends* $\mathcal{I}$ from the domain $\boldsymbol{D}$ to the domain $\Phi(\boldsymbol{D})$. Following the standard practice in the field, we will henceforth re-use the symbol $\mathcal{I}$ for the generalized interpretation too, that is, we understand $\mathcal{I}: \Phi(\boldsymbol{D}) \to \{0, 1\}$ as the natural extension of the original $\mathcal{I}: \boldsymbol{D} \to \{0, 1\}$ to the larger domain $\Phi(\boldsymbol{D})$. Furthermore, because the set $\boldsymbol{D}$ of variables in use will usually be clear from the context, we mostly will not specify it explicitly.

There is another customary slack in terminology which I would like to point out. Strictly speaking, we have introduced Boolean formulas in Definition 1.2 as mere strings of symbols. The formula $(\neg (X \wedge Y) \vee (Z \wedge Y))$ is just that – a string made of bracket symbols, variable symbols, and the logical connective symbols $\neg$, $\wedge$, $\vee$. On the other hand, Definition 1.3 directly assigns a Boolean function to every such formula. By an abuse of terminology and notation, we don't use a different notation for the function obtained from the formula $(\neg (X \wedge Y) \vee (Z \wedge Y))$. We would simply say things like "the function $(\neg (X \wedge Y) \vee (Z \wedge Y))$", although strictly speaking, $(\neg (X \wedge Y) \vee (Z \wedge Y))$ isn't a function but a formula, that is a symbol string denoting a function by virtue of Definition 1.3. And when we use the symbols φ, ψ, etc., we may refer to formulas or to functions.

1.4 First steps toward Boolean logic

"Logic" is a very ancient theme indeed. Even in antiquity, hundreds of years before the Western calendar started counting years at 0, Greek philosophers contemplated the truth or falsity of propositions and started to spell out rules for correct logical thinking. From those ancient times, a number of logical concepts have survived to the present day, and they are still relevant for the foundations of mathematical logic, theoretical CS and Artificial Intelligence. Two of these venerable concepts are *tautologies* and *contradictions*.

In modern Boolean parlance, a tautology is a Boolean formula which is always true, and a contradiction is never true. Before we give the precise definition, we introduce a little helper notion:

Definition 1.4 (adapted interpretations). An interpretation $\mathcal{I}: \boldsymbol{D} \rightarrow \{0, 1\}$ is *adapted* to a Boolean formula φ if all Boolean variables that occur in φ are contained in $\boldsymbol{D}$.

In other words, interpretations adapted to φ provide truth value assignments for all variables in φ.

Definition 1.5 (tautologies and contradictions). A Boolean formula φ is a *tautology* if for all interpretations $\mathcal{I}$ which are adapted to φ it holds that $\mathcal{I}(\varphi) = 1$. A Boolean formula φ is a *contradiction* if for all interpretations $\mathcal{I}$ which are adapted to φ it holds that $\mathcal{I}(\varphi) = 0$.

The classical example for a tautology is

(7) $(X \vee \neg X)$

and the classical example for a contradiction is

(8) $(X \wedge \neg X)$.

But also very complex formulas with many Boolean variables can be tautologies or contradictions. One way to find out whether a given Boolean formula φ is a tautology or contradiction would be to compute the complete truth table of φ. Recall that each row in the truth table corresponds to one of the possible interpretations of the variables in φ. Recall furthermore that if φ contains k Boolean variables, the truth table has 2^k many rows – its size "explodes" with growing k and very soon it becomes practically impossible to compute this HUGE table. Unfortunately there is no known general procedure to find out whether a given φ is a tautology/contradiction which is less costly than computing the entire truth table. In fact, logicians have reason to believe that no faster method exists (but that is an unproven conjecture!).

Here are two elementary and (I would say) obvious facts relating to tautologies and contradictions:

- For any Boolean expression φ, $(\varphi \vee \neg \varphi)$ is a tautology and $(\varphi \wedge \neg \varphi)$ is a contradiction.
- If φ is a tautology, then $\neg \varphi$ is a contradiction and vice versa.

The following definition introduces a concept which later will turn out to lie at the core of "correct logical reasoning". I introduce it at this early stage because it is related to tautologies/contradictions.

Definition 1.6 (satisfying a Boolean formula). An interpretation $\mathcal{I}$ which is adapted to a Boolean formula φ is said to *satisfy* the formula φ if $\mathcal{I}(\varphi) = 1$. A formula φ is called *satisfiable* if there exists an interpretation which satisfies φ.

If you think about it, you will find that the following two are equivalent characterizations of satisfiability:

- A Boolean formula is satisfiable iff its truth table contains at least one row that results in 1.
- A Boolean formula is satisfiable iff it is not a contradiction.

Here I used the abbreviation "iff" for "if and only if".

Another basic notion is the equivalence of two formulas:

Definition 1.7 (equivalence of Boolean formulas). Let φ, ψ be two Boolean formulas. Then φ is *equivalent* to ψ, written $\varphi \equiv \psi$, if for all interpretations $\mathcal{I}$ which are adapted both to φ and ψ it holds that $\mathcal{I}(\varphi) = \mathcal{I}(\psi)$.

Examples:

- $(X \vee Y) \equiv (Y \vee X)$
- $(X \vee Y) \equiv (Y \vee X) \wedge (Z \vee \neg Z)$

The second example shows that two equivalent formulas need not contain the same set of variables.

There are numerous "laws" of Boolean logic which are stated as equivalences. Here is a choice:

Proposition 1. For any Boolean formulas φ, ψ, χ the following equivalences hold:

1. $(\varphi \wedge \varphi) \equiv \varphi$ and $(\varphi \vee \varphi) \equiv \varphi$ *("idempotency")*
2. $(\varphi \wedge \psi) \equiv (\psi \wedge \varphi)$ and $(\varphi \vee \psi) \equiv (\psi \vee \varphi)$ *("commutativity")*
3. $((\varphi \wedge \psi) \wedge \chi) \equiv (\varphi \wedge (\psi \wedge \chi))$ and $((\varphi \vee \psi) \vee \chi) \equiv (\varphi \vee (\psi \vee \chi))$ *("associativity")*
4. $(\varphi \wedge (\psi \vee \chi)) \equiv (\psi \wedge \varphi) \vee (\psi \wedge \chi)$ and $(\varphi \vee (\psi \wedge \chi)) \equiv (\psi \vee \varphi) \wedge (\psi \vee \chi)$ *("distributivity")*
5. $\neg\neg\varphi \equiv \varphi$ *("double negation")*
6. $\neg(\varphi \wedge \psi) \equiv (\neg\varphi \vee \neg\psi)$ and $\neg(\varphi \vee \psi) \equiv (\neg\varphi \wedge \neg\psi)$ *("de Morgan's law")*

Each of these laws can be proven by writing down the corresponding truth table. I illustrate this for the first of the two de Morgan's laws:

φ	ψ	$\neg\varphi$	$\neg\psi$	$(\varphi \wedge \psi)$	$\neg(\varphi \wedge \psi)$	$(\neg\varphi \vee \neg\psi)$
0	0	1	1	0	**1**	**1**
0	1	1	0	0	**1**	**1**
1	0	0	1	0	**1**	**1**
1	1	0	0	1	**0**	**0**

The two last columns marked in bold font are identical, showing equivalence of $\neg(\varphi \wedge \psi)$ and $(\neg\varphi \vee \neg\psi)$.

Such equivalence laws can be used, among other things, to "calculate" with logics, executing stepwise transformations from a starting formula to some target formula, where each step applies one such equivalence law. This is similar to algebraic calculations where a starting formula is stepwise transformed to a target formula. Designing automated routines to transform starting formulas into target formulas is a practically important topic for AI applications, but we will not further pursue this theme of "automated reasoning" in this course.

1.5 Normal forms of Boolean formulas

Boolean formulas can be structured in very complex, highly nested ways – consider, for instance the monster

$$(((X_1 \wedge \neg X_4) \wedge (X_2 \vee X_1)) \vee \neg((X_6 \wedge X_{100}) \vee 0 \vee \neg((X_3 \vee \neg X_4 \vee X_2) \vee ((X_1 \wedge \neg X_4) \vee (X_2 \wedge \neg X_9 \wedge X_1)))))$$

This is a syntactically correct formula but I dare say you would have difficulties "understanding" it – and likewise, a computer program processing this formula would have difficulties disentangling it. Fortunately, every Boolean formula is equivalent to a Boolean formula which has a very simple and transparent structure. Such simple, standardized structures are called *normal forms* of Boolean formulas. They are absolutely instrumental for any practical (human or computer-driven) work with Boolean logic. I will describe in some detail the so-called *conjunctive normal form* (CNF) and briefly hint out another normal form called *disjunctive normal form* (DNF).

To prepare the grounds for normal forms, the following notion is helpful:

Definition 1.8 (literals). A *literal* is a Boolean formula that has one of the forms X_i, $\neg X_i$, 0, 1, $\neg 0$, or $\neg 1$. That is, a literal is just a Boolean variable or a constant, or its negation. The literals X_i, 0, 1 are called *positive literals* and $\neg X_i$, $\neg 0$, $\neg 1$ are called *negative literals*. We write L_i for any literal.

Definition 1.9 (CNF). A Boolean formula is said to be in *conjunctive normal form* if it is a conjunction of disjunctions of literals.

Example: $(X_2 \vee \neg X_2 \vee X_3) \wedge X_1 \wedge (\neg X_2 \vee \neg X_3 \vee 0)$ is a conjunction of the disjunctions $(X_2 \vee \neg X_2 \vee X_3)$, X_1, and $(\neg X_2 \vee \neg X_5 \vee 0)$.

Example: X_1 is in CNF, too. This is the effect of the mathematical way of thinking about conjunctions and disjunctions: an isolated Boolean formula φ is considered to be a conjunction (you could say, a conjunction with itself); it is also considered to be a disjunction (with itself). So X_1 can be considered a conjunction of a disjunction (smile).

The practically important convenience feature about formulas in CNF is that such formulas are "shallow" – they don't have nested bracketing levels.

Proposition 2. Every Boolean formula φ is equivalent to a Boolean formula χ in CNF.

Proof. We don't usually go through proofs in this lecture, but here I make an exception. First, because Proposition 2 is important, and second, because the proof is instructive – the method used to prove this proposition is used in theoretical CS in many places. The proof method is called "proof by induction over the structure of Boolean expressions". We repeat the inductive make-up of the syntax of Boolean formulas from Definition 1.2 as follows. I will not give the fully abstract proof but illustrate the important steps by examples.

Basis of induction:

1a. Let $\varphi = X_i$ be a Boolean formula that just consists of a single Boolean variable. Then φ is in CNF according to what I said above about "isolated" formulas. Putting $\chi = \varphi$ gives us the (trivial) CNF of φ.

1b. Case $\varphi = 0$ or $\varphi = 1$: similar.

Induction step:

2a. Assume that $\varphi = (\chi_1 \wedge \chi_2)$ is a conjunction of two formulas χ_1, χ_2 which (by induction) we may assume to be in CNF already. Then simply by dropping brackets we see that φ is in CNF too. Illustration by example: Let for instance $\chi_1 = (X \wedge (\neg Y \vee 0))$ and $\chi_2 = ((\neg X \vee Z) \wedge (Y \vee W))$. Then

$$\begin{aligned}\varphi = (\chi_1 \wedge \chi_2) &= ((X \wedge (\neg Y \vee 0)) \wedge ((\neg X \vee Z) \wedge (Y \vee W))) \\ &= (X \wedge (\neg Y \vee 0) \wedge (\neg Y \vee Z) \wedge (Y \vee W)) \\ &=: \chi.\end{aligned}$$

2b. Assume that $\varphi = (\chi_1 \vee \chi_2)$ is a disjunction of two formulas χ_1, χ_2 which are in CNF already. By a repeated application of distributivity we can transform $(\chi_1 \vee \chi_2)$ into a CNF formula. Example: Let for instance $\chi_1 = X \wedge (\neg Y \vee 0)$ and $\chi_2 = (\neg X \vee Z) \wedge W$. Then we apply distributivity several times as follows:

$$\begin{aligned}\varphi &= (X \wedge (\neg Y \vee 0)) \vee ((\neg X \vee Z) \wedge W) \\ &= (X \vee ((\neg X \vee Z) \wedge W)) \wedge ((\neg Y \vee 0) \vee ((\neg X \vee Z) \wedge W)) \\ &= (X \vee \neg X \vee Z) \wedge (X \vee W) \wedge (\neg Y \vee 0 \vee \neg X \vee Z) \wedge (\neg Y \vee 0 \vee W)\end{aligned}$$

which is in CNF.

2c. Assume that $\varphi = \neg\chi$ is a negation of a formula χ which by induction assumption is in CNF. By applications of de Morgan's rule, double negation and distributivity we can transform $\neg\chi$ into a CNF formula. Example: let $\chi = (\neg X \vee Z) \wedge W$. Then

$$\begin{aligned}\varphi &= \neg((\neg X \vee Z) \wedge W) && \\ &= \neg(\neg X \vee Z) \vee \neg W && \text{[de Morgan]} \\ &= (\neg\neg X \wedge \neg Z) \vee \neg W && \text{[de Morgan again]} \\ &= (X \wedge \neg Z) \vee \neg W && \text{[double negation]} \\ &= (X \vee \neg W) \wedge (\neg Z \vee \neg W) && \text{[distributivity]}\end{aligned}$$

where the last line is a formula in CNF.

With a little additional formalism it can be shown that these transformations, which I here demonstrated only in examples, are always possible. We have thus shown that every Boolean formula can be transformed into a formula in CNF.

For completeness I mention the "mirror" version of CNF, the disjunctive normal form: a Boolean formula is in DNF if it is a disjunction of conjunctions of literals. For example, $(X_2 \wedge \neg X_2 \wedge X_3) \vee X_1 \vee (\neg X_2 \wedge \neg X_3 \wedge 0)$ is in DNF. Every Boolean formula can be transformed into an equivalent formula in DNF.

CNFs can be used to demonstrate that *every Boolean function can be represented by a Boolean expression*. I illustrate this important fact with an example. Any Boolean function is fully specified by its truth value table. Consider for example the following arbitrary Boolean function φ: $\{0, 1\}^3 \to \{0,1\}$ given by this truth table:

X	Y	Z	$\varphi(X, Y, Z)$

0	0	0	**1**
0	0	1	**0**
0	1	0	**0**
0	1	1	**1**
1	0	0	**1**
1	0	1	**0**
1	1	0	**1**
1	1	1	**0**

We will now create a Boolean formula for the function φ. The CNF helps us to do so. In order to find a Boolean formula χ in CNF for this language, we inspect all rows in the table that result in a 0. The first of these rows is the second row in the table, $(X, Y, Z) = (0, 0, 1)$. In order to make sure that our to-be-constructed formula χ evaluates to 0 when $X = 0$, $Y = 0$, $Z = 1$, we add a disjunction to χ which ensures exactly this: $(X \vee Y \vee \neg Z)$ (think about it!). Doing this for all rows which have a 0 on the right hand side in the table gives us

$$\chi = (X \vee Y \vee \neg Z) \wedge (X \vee \neg Y \vee Z) \wedge (\neg X \vee Y \vee \neg Z) \wedge (\neg X \vee \neg Y \vee \neg Z).$$

The fact that every Boolean function can be represented by a Boolean formula could also be stated as "every Boolean function can be obtained by a (possibly nested) composition of the elementary functions AND, OR, NOT". These three functions are a *universal* "basis" for constructing *all* other Boolean functions.

The NAND (also known as the Sheffer stroke, written $\uparrow$) is, surprisingly a stand-alone universal operation: AND, OR, NOT can all be expressed in terms of $\uparrow$, hence any Boolean function can be constructed from $\uparrow$ alone. This renders the NAND gate particularly important for the physical realization of digital microchips.

1.6 An interim summary, and universality of AND, OR, NOT

Here is a collection of take-home facts that we have collected so far:

- Boolean functions are defined to be "binary" functions from φ: $\{0, 1\}^k \rightarrow \{0,1\}$.
- A Boolean function of arity k can be specified by a truth table with 2^k rows.
- Boolean formulas are inductively constructed by creating compound formulas from given ones using the logical connective symbols $\neg$, $\wedge$, $\vee$, starting the process from the Boolean variables X_i and the truth value constants 0 and 1.
- A Boolean formula specifies a Boolean function, and every Boolean function can be represented by a Boolean formula, even by a Boolean formula in CNF.

Note that in the make-up of Boolean formulas we only used the elementary Boolean functions AND, OR and NOT. These three functions AND, OR, NOT are a universal basis for constructing *all* Boolean functions.

Other choices of elementary Boolean functions are possible which yield a universal basis. Electrical engineers typically design their binary electronic switching circuits not from AND, OR, NOT but from other elementary "gates" because other gates are easier to create from electronic devices (like transistors) than AND, OR, NOT. However, the textbook theory of Boolean logic usually chooses AND, OR, NOT because these three operations are closest to "natural" human logic. The ancient Greek philosophers as well as contemporary mathematicians think in terms of AND, OR, NOT and not in terms of logical operations like NAND and the like.

1.7 Boolean circuits

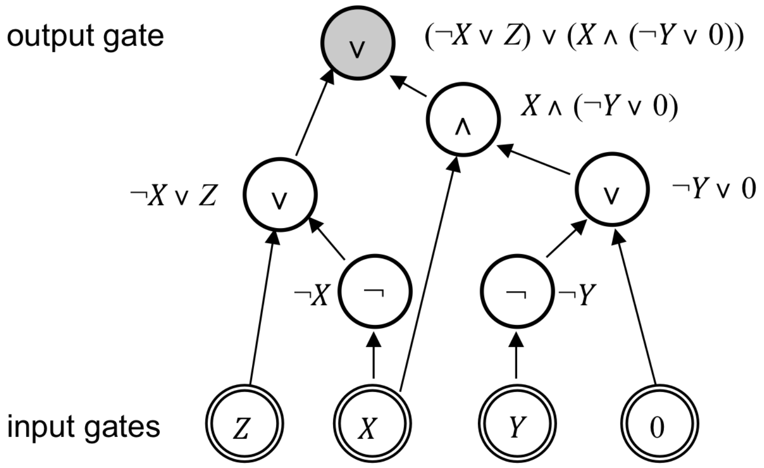
So far we have met two ways of specifying Boolean functions: truth tables and Boolean formulas. By a little twist on the latter we get a third way, *Boolean circuits*. This is a representation format that is closer to electronic circuits than tables or formulas. A Boolean circuit in a sense just "unfolds" a Boolean formula (which is a one-dimensional string of symbols) into 2 dimensions. This is best explained with an example. Consider the Boolean formula

(9) $\varphi = (X \wedge (\neg Y \vee 0)) \vee (\neg X \vee Z)$

In its inductive construction, it is composed from its *subformulas*

$0, X, Y, Z, \neg Y, \neg X, (\neg Y \vee 0), (\neg X \vee Z), (X \wedge (\neg Y \vee 0)), (X \wedge (\neg Y \vee 0)) \vee (\neg X \vee Z)$

where in good math spirit we call the entire formula a "subformula" too. A Boolean circuit is a graphical representation of all the subformulas, which are now called *gates*. Depending on whether the subformula is "atomic" (a constant or a single Boolean variable) or whether it is "composite" (made from other subformulas by use of $\neg$, $\wedge$, $\vee$), the gates are called *input gates* (for atomic subformulas) or *logic gates* (for composite subformulas). Logic gates come in three sorts: $\wedge$-gates, $\vee$-gates and $\neg$-gates. Gates are drawn as circles, and the composition relationships between them as arrows. The formula (9) becomes "exploded" into the following diagram:



output gate
$(\neg X \vee Z) \vee (X \wedge (\neg Y \vee 0))$
$X \wedge (\neg Y \vee 0)$
$\neg X \vee Z$
$\neg Y \vee 0$
$\neg X$
$\neg Y$
input gates
Z
X
Y
0

You could interpret this as an electronic circuit which has four inputs (X, Y, Z, 0) of which the last is always equal to 0 and the first three can be fed with 0's or 1's. Formally input patterns are interpretations $\mathcal{I}$: $\{X, Y, Z\} \rightarrow \{0, 1\}$. The circuit has a binary output which corresponds to $\mathcal{I}(\varphi)$ – check again Definition 1.3 to make sure you understand this claim.

Here is a formal definition:

Definition 1.10 (Boolean circuits). A *Boolean circuit* (or simply circuit) is a finite directed graph whose nodes i are of sort $s_i \in \{0, 1, \neg, \wedge, \vee, X_1, X_2, X_3, ...\}$. Nodes are called *gates*. The indegree (= number of ingoing edges) of a gate is 0 for gates of sort 0, 1 or X_i; 1 for gates of sort $\neg$ and 2 for gates of sort $\wedge$ or $\vee$. Gates are numbered such that if (i, j) is an edge, then $i < j$. Gates of sort X_i or 0 or 1 are *input* gates. The gate with largest index n is the *output* gate of the circuit.
Students of Electrical Engineering actually build such digital switching circuits from electronic components in their first-year labs. Digital microchips can in general and with some abstraction effort be regarded as Boolean circuits. The main differences between real microchips and Boolean circuits are the following:

- A microchip usually has more than one output pin. Formally, such systems realize *multi-valued* Boolean functions φ: $\{0, 1\}^k \rightarrow \{0, 1\}^m$. Note that a multi-valued Boolean function φ: $\{0, 1\}^k \rightarrow \{0, 1\}^m$ can be understood as a set of m single-valued Boolean functions φ_i: $\{0, 1\}^k \rightarrow \{0, 1\}$, one such φ_i for each of the m outputs of the circuit.
- As I already mentioned, physical transistor-based microchips typcially use other elementary Boolean functions than AND, OR, NOT for their gate nodes because those other functions are easier to realize with semiconductor elements.
- A microchip may have additional electronic subsystems that provide infrastructure to the "logic" parts, like clocks, voltage stabilizers etc.

1.8 Further steps toward Boolean logic

In this final section I want to provide a glimpse on the "high-end" side of Booleanism, namely AI-style applications in intelligent reasoning.

The main and eternal type of question that intelligent reasoning is facing is to find out whether some *conclusion* follows logically from some *premises*. One could say without too much simplification that logical reasoning is all about "**If** ... **then** ..." arguments. Here are some examples:

- The most classical, venerable example which has been discussed already by ancient Greek philosophers is the following argument: ***If*** *all men are mortal* ***and*** *Socrates is a man,* ***then*** *Socrates is mortal*. The Greek philosophers compiled a set of about 20 such arguments, called *syllogisms*. A classical syllogism has two premises and one conclusion. Here are two examples of syllogisms, expressed in an abstract form:

  If all B are C, and all A are B, then all A are C. (For instance, ***If*** *all humans are mortal* ***and*** *all Greeks are human,* ***then*** *all Greeks are mortal.*)

  If some B are not C, and all A are B, then some A are not C (e.g., ***If*** *some cats have no tails* ***and*** *all cats are mammals,* ***then*** *some mammals have no tails*).

Check out http://en.wikipedia.org/wiki/Syllogism if you want to learn more about these ancient roots of logic.

- In mathematical textbooks (and in the minds of mathematicians) the core item of interest are *theorems*. A mathematical theorem is always of the form ***If** premises* A_1, ..., ***and*** A_k *hold, **then** conclusion Z holds*. The necessary premises are not always listed explicitly in the theorem's statement; often they tacitly include premises and definitions which are understood in the context where the theorem is stated. For example, I stated our Proposition 1 above as follows:

  For any Boolean formulas φ, ψ, χ the following equivalences hold: $(\varphi \wedge \varphi) \equiv \varphi$ *and* $(\varphi \vee \varphi) \equiv \varphi$, ...

  If I would have stated this theorem in a fully spelled-out version, its premise would have been very long and would have included all of the preceding definitions. It would then have looked something like:

  If** φ, ψ, χ are Boolean formulas **and** a Boolean formula is either a Boolean constant or a Boolean variable or can be obtained from a Boolean formula by an iterated application of the operations $\neg$, $\wedge$, $\vee$, **and** a Boolean constant is either* 0 *or* 1*, **and** the $\neg$ operations are unary and the operations $\wedge$, $\vee$ are binary, **and** ... **and** ... **and**...* [essentially repeating all of the definitions introduced before], ***then $(\varphi \wedge \varphi) \equiv \varphi$ *and* $(\varphi \vee \varphi) \equiv \varphi$, ...

- An AI-empowered autonomous mobile robot incessantly has to *plan* its next actions. Many implementations of high-level robot planning systems are based on logical reasoning. Such logic-based planning algorithms execute reasoning steps of the kind

  If** in the current situation facts* F_1, ... ***and F_m *hold, **and** if I* [the robot] *now make this action A*, ***then*** *the situation will change such that fact E holds.*

  Concrete example: ***If** the cup is on the floor **and** I lift it on the table, **then** the cup will be on the table.*

Abstracting from these examples, it turns out that one of the main tasks for philosophers, mathematicians, AI programmers and AI programs is to decide whether a given argument of the kind

If** premises* P_1 ***and** ... **and P_m *hold, **then** conclusion C holds*

is true. The premises P_i and the conclusion C will be expressed in some logic formalism (of which there are many), the simplest of which is Boolean logic. So the question that we will consider in the remainder of this section is whether a Boolean formula of the structure

(10) $(\varphi_1 \wedge \ldots \wedge \varphi_m) \rightarrow \psi$

is true, that is, whether it is a tautology. Deciding whether formulas of this kind are tautologies lies at the heart of logical reasoning!

Remember that a tautology is a Boolean formula whose truth table has only 1's in the result column. The brute-force way to check (10) for being a tautology would be to compute its truth table and find out whether all results are 1. When (10) has many Boolean variables in it, this is practically impossible because the truth table would be larger than the universe. Therefore, one proceeds indirectly and first re-formulates the problem in a clever way.

A Boolean formula τ is a tautology iff $\neg\tau$ is a contradiction. A Boolean formula is a contradiction iff is not satisfiable. Thus in order to to check whether (10) is a tautology we may equally well check whether $\neg((\varphi_1 \land \ldots \land \varphi_m) \rightarrow \psi)$ is unsatisfiable. If we find it is unsatisfiable, then (10) is proven to be a tautology.

Inspecting the truth table for the logical implication $\rightarrow$ reveals that $(\varphi \rightarrow \psi)$ is equivalent to $\neg(\varphi \land \neg\psi)$. Hence, $((\varphi_1 \land \ldots \land \varphi_m) \rightarrow \psi)$ is equivalent to $\neg(\varphi_1 \land \ldots \land \varphi_m \land \neg\psi)$, and by double negation, $\neg((\varphi_1 \land \ldots \land \varphi_m) \rightarrow \psi)$ is equivalent to $(\varphi_1 \land \ldots \land \varphi_m \land \neg\psi)$. Therefore, in order to to check whether (10) is a tautology we may equally well check whether

(11) $(\varphi_1 \land \ldots \land \varphi_m \land \neg\psi)$

is satisfiable. If we find it is, then (10) is not a tautology: we have *disproven* the claim $(\varphi_1 \land \ldots \land \varphi_m) \rightarrow \psi$.

There are a number of reasons why one proceeds in this indirect way of trying to find out whether (11) is satisfiable, instead of trying to find out directly whether (10) is a tautology:

- One (often) does not have to compute the entire truth table of (11). Once one has found a single line in that table with a result of 1, (11) has been shown to be satisfiable and hence (10) is disproven – end of procedure.
- The formula (11) has a homogeneous structure: it is a single conjunction. If one transforms the elements $\varphi_1, \ldots, \varphi_m, \neg\psi$, each into CNF, the entire formula is in CNF. Often the premises $\varphi_1, \ldots, \varphi_m$ will already be in CNF, so the computational cost is small. Then one can invoke clever algorithms which are specifically designed to search for interpretations $\mathcal{I}$ which satisfy a CNF formula.

In fact, *much* research has gone into sophisticated algorithms that decide whether a Boolean function in CNF is satisfiable. Entire books and scientific workshops are devoted to this theme. Because this question is of such a fundamental importance for CS, I devote an explicit definition to it:

Definition 1.11 (satisfiability problem). The satisfiability problem in theoretical CS, abbreviated standardly by SAT, is the following computational problem: given as input a Boolean formula in CNF, compute as output a "yes" or "no" response according to whether the input formula is satisfiable or not.

In the worst case, finding out whether a Boolean formula in CNF is satisfiable amounts to computing the entire truth table. In practice, this worst case often does not strike and cleverly designed algorithms find "yes" or "no" answers much earlier. It is decidedly the most famous open problem in CS – and possibly in mathematics in general – whether some (as yet undiscovered) algorithm exists which can decide the SAT problem always faster than by computing the entire truth table. Check out http://en.wikipedia.org/wiki/Boolean_satisfiability_problem to read more. If you can find an algorithm which always is faster than computing the entire truth table, or if conversely you can prove that no such algorithm exists, then you will be the most famous mathematician of the decade if not the century, and you will have secured a one Million Dollar prize – you will have solved what is known as the N = NP problem (http://en.wikipedia.org/wiki/Millennium_Prize_Problems).

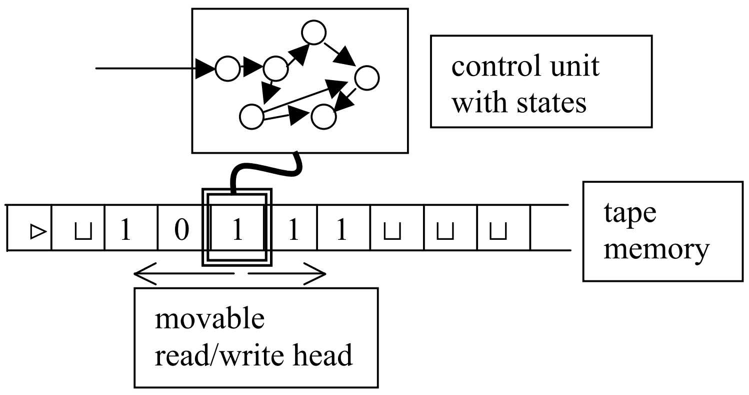
2 Some elements of computational complexity

In the preceding subsection 1.9 I made informal statements concerning the "speed" of certain computations. I said, for instance, *"... an algorithm which always is faster than computing the entire truth table…"*. Questions relating to the "speed" of a computation, or its "cost", or "effort" or the like, are of natural importance for CS. Some computations of great practical importance apparently require a large computational effort. A good example is weather prediction. Predicting the weather amounts to simulate the atmospheric dynamics of the entire planet. These simulations are done on supercomputers – in fact, national weather agencies maintain some of the largest existing computing facilities. And the computer scientists and mathematicians (!) employed there are naturally interested to find ways to speed up their computations. Another example where speed matters, which may be closer to your everyday experience, is internet search. Google started its epochal rise by a clever, innovative algorithm to rank pages that match the search keywords. Obviously this algorithm had to be fast as lightning! Have you ever thought that it comes close to a miracle that you get the results from a Google query so fast, even though this somehow must require comparing billions of webpages?

The importance of computational efficiency has been clear from the beginnings of modern CS. It has given rise to an entire subfield of theoretical CS called *complexity theory*. This is one of the most difficult and complex themes in theoretical CS. In a nutshell, the core question in complexity theory is *If I have an algorithm A which computes the problem P in time T, is there another algorithm B which solves the same problem in less time?* Other measures of computational cost besides time are also of interest, in particular the amount of *memory space* that is claimed by a given algorithm, but here I will only consider *time complexity*. In this course we cannot dig deep into complexity theory, but I want to outline some basic concepts.

First thing – how is the "time" needed for a computation measured? It cannot be reasonably measured in seconds – because with computers getting faster every year, the time measures would become a moving target. One wants a measure of "time" that is independent of a concrete computer.

The standard answer to finding a measure of computation time which is independent of concrete computing hardware is to define an abstract computing device which is as simple as possible (to admit a transparent mathematical formalization) yet as powerful as any other computer (to be relevant at all). This "elementary abstract total-power computer" is the *Turing machine* (TM). It is called a "machine" but in fact it is a mathematical concept. I will not give the mathematical definition but instead provide a intuitive sketch by way of a diagram, pretending that the TM were a mechanical device and not a mathematical abstraction.



A TM consists of three main components:

- A "control unit" (think of a CPU) which is a simple switching device (much like a Boolean circuit) which can switch between a finite number of *states*.
- A "memory tape" which is a right-infinite linear array of *cells*. Each cell can be empty or hold a 1 or a 0 symbol (other and more symbols can be used too but zeros and ones suffice). This is the Turing machine's "working memory".
- A read-write head which can move along the memory tape. When it is positioned over a cell (it can "see" only one cell at a time) it can *read* the symbol, and it can *write* a new symbol into the cell, replacing the previous one if the cell was not empty.

Like modern digital computers, a TM has a *working cycle* which is repeated again and again. A working cycle consists of the following substeps:

1. Read the symbol that is currently under the read/write head (or read "empty").
2. Depending on the current state of the control unit and the symbol that has been read, do the following three things:
    a. switch the control unit to the next cycle's state,

b. write a symbol in the currently seen cell,

c. move the head one step to the left or to the right (or remain on the same cell).

A *run* of a TM is a sequence of such working cycles which is obtained as follows:

1. At the beginning of a run, the control unit is in a specially designated *starting state*, and the read/write head is positioned at the leftmost tape cell.
2. At the beginning of a run, the "user" has written in *input word* into some cells, starting from the left end of the tape. An input word is just a finite sequence of 0's and 1's. To the right of the input words all tape cells are initially empty. Note that in theoretical CS, a "word" is the technical term for any finite sequence of symbols from some given alphabet.
3. From this starting configuration, the working cycle updates are activated. The TM starts "running", executing one working cycle after the other. In this process, the control unit passes through a sequence of states, the read/write head moves back and forth on the tape, reading/erasing/printing new symbols as it moves along.
4. One of the states of the control unit is called the *halting state*. If at the end of some working cycle the control unit is led into that state, the run stops. (And if the halting state is not entered, the run does never stop...! you know this behavior from your computer when it "freezes" – in fact it most likely did not "freeze" but entered an un-interruptible infinite loop of non-terminating actions.)
5. After the run has stopped, whatever is currently written on the tape is considered as the *output word* of that run.

This model of a TM can be enriched by convenience features. The most common of these add-on features is to endow the TM with more than a single memory tape. Multi-tape TMs often are granted a specific tape that is used exclusively for feeding the input word, and another special tape dedicated to write the output on. Such multi-tape TMs can't solve other or more computational tasks than single-tape TMs can, but they make it easier to design concrete TMs that can master a given computational task.

These simple "machines" can be mathematically proven to be capable of computing the same input-output functions as any digital computer. Therefore it is no loss of generality if one builds the formal theory of computation on the fundaments of just these TMs. In fact, the two fundamental branches of theoretical computer science, which investigate the twin questions

- of which input-output functions can be computed at all, regardless of cost (theory of *computability*), and
- and among the input-output functions which can be computed at all, which of them can be computed how fast (theory of *complexity*),

both build on TMs as the standard, fundamental, mathematical model of a computer.

The TM also serves as the standard model of an *algorithm*. While you likely have an intuitive grasp of this concept as a "computational procedure", the concept of a TM affords us with a rigorous, precise definition of this central item of CS. "Algorithms" can be identified with TMs:

$$\boxed{\text{Algorithm = Turing machine}}$$

Thus, an algorithm (in the sense how this concept is understood in theoretical CS) is a formal mechanism (typically a TM) which computes a word-input, word-output function. The input words are the symbol sequences that are written on the TM (input) tape at the beginning of a run, the corresponding output words are what is left on the (output) tape when the TM enters the halting state.

Example. Computing the square function $f(n) = n^2$ of integers n: in order to frame this formally as a formal TM-based algorithm, one needs to devise a way to *code* the input integer n as a word string of 0's and 1's. The natural way to do this is to use the binary representation of integers (about which you have learnt in the first part of this lecture). A TM computing this function must be designed in such a way that

1. On each input word (i.e. on every input sequence of 0's and 1's coding n) the TM embarks on a run that after a finite number of working cycles actually enters the halting state, and
2. at halting time, on the tape there is a single connected sequence of 0's and 1's (with no empty cells in between) which is the binary representation of n^2.

Many different TMs can be devised which obey these specifications – there are many different algorithms to compute the square function.

Example. Checking whether a Boolean formula φ in CNF is satisfiable. Again, in order to frame this formally as an algorithm, one first needs to fix a way to code Boolean formulas as binary words, in order to make them "inputtable" for a TM. There are many coding schemes that do this. For instance, one can first write down φ in human-readable style as we did in Section 1, giving a symbol string in the standard ASCII symbol set. Then one can replace each ASCII symbol in this string by its 7-bit binary representation, yielding a binary word that uniquely codes φ. One furthermore needs a way to code the desired outcome of a satisfiability check as a binary word. The two possible outcomes are "yes" (φ is satisfiable) and "no" (it's not). One could, for instance, declare that an output word "1" means "yes" and an output word "0" means "no". Like in the case of computing the square function, one must design a concrete TM (= design a concrete algorithm) which

1. On each input word (i.e. on every input sequence of 0's and 1's coding a Boolean expression φ) the TM embarks on a run that after a finite number of working cycles enters the halting state, and

2. at halting time, on the (output) tape there is a single non-empty cell which holds a 1 if φ is satisfiable and a 0 if not.

Again, there are many different TMs (= algorithms) which solve this task.

If you want to learn more about TMs, visit the well-written Wikipedia article https://en.wikipedia.org/wiki/Turing_machine. It is kind of funny to see a TM do its work. The read/write head moves left and right on its tape like crazy and prints and erases 0's and 1's (or other symbols), and the tape inscription can grow very large. There are numerous browser-runnable visualizations of these busy devices on the web, for instance at http://morphett.info/turing/turing.html (load the "primality test" TM!)

Now we are equipped to tackle the main question of this chapter, time complexity. Having TMs at our disposal, we now can measure the time needed by an algorithm for computing a function: it is simply the number of TM update cycles that a run takes before the TM halts. For a mathematical analysis of TM runtimes, we make this (halfway) precise:

Definition 2.1 (measuring TM runtime). Let M be a TM with a tape alphabet $\{0, 1\}$. Let f: $A \to B$ be some function from some *countable* set $A = \{a_1, a_2, ...\}$ to some arbitrary set B. Let $\mathcal{C}$ be a *coding* function which translates every $a_i \in A$ into a unique binary codeword $\mathcal{C}(a_i)$. Let $\mathcal{D}$ be a *decoding* function which translates every binary word to some element of $B \cup \{👎\}$. Then we define the following items:

1. *M computes f* if for every $a_i \in A$, when M is started with $\mathcal{C}(a_i)$ on its (input) tape, M executes a finite number l of updates cycles, then enters the halting state, and the binary word w which is then on the (output) tape satisfies $f(a_i) = \mathcal{D}(w)$.
2. If M computes f, the number of l of update cycles before halting when M is started on input $\mathcal{C}(a_i)$ is called the *runtime* of M on input $\mathcal{C}(a_i)$. It is denoted by $l = T(M, \mathcal{C}, a_i)$.

In this definition, the image domain of the decoding function $\mathcal{D}$ was set to $B \cup \{👎\}$. The special value 👎 would be returned by $\mathcal{D}$ for those binary words which are not useful outputs of the TM.

Example. In the square function example, f: $\mathbb{N} \to \mathbb{N}$, $n \mapsto n^2$ is the ordinary square function on the natural numbers. The coding function $\mathcal{C}$ transforms each $n \in \mathbb{N}$ to its binary representation string, and the decoding $\mathcal{D}$ retransforms binary representations of natural numbers back to the corresponding $n \in \mathbb{N}$.

Example. In the satisfiability check task, we have

$$f\text{: \{set of all Boolean expressions } \varphi\} \to \{\text{"yes", "no"}\}.$$

The coding $\mathcal{C}$ transforms every Boolean expression φ to a binary codestring. The decoding $\mathcal{D}$ maps the word 1 to "yes", the word 0 to "no", and any other binary word to 👎.

Now let us take a closer look at the runtimes in the square function example. A natural strategy for designing a TM that computes this function would be to mimick on the TM the paper-and-pencil procedure for computing products that you know from elementary school. For example, in order to compute 5^2, where 5 in binary is written as `101`, you would write down a scheme like

(12)
```
101*101
_______
101
 000
  101
_______
11001
```

The bulk of this writeup is contained in the middle part. Let L = length($\mathcal{C}(n)$) be the length of the codeword of the argument n. In our example (12) we have $L = 3$. The scheme (12) requests to write down L^2 0's and 1's in that middle part. The length of the final output string (here `11001`) is at most $2L$. Altogether you have to write down $L^2 + 2L$ symbols (or one less if the length of the output word is $2L–1$). Without going into details, I state here that this elementary-school scheme can be mimicked quite closely on a 3-tape TM M_{square}. The TM needs some extra head moves (which invoke extra working cycles) for "administrative overhead", like moving the read-write head back to the beginning of a tape inscription when a substep of the scheme (12) has been completed. If one counts all the necessary cycles one finds that one can design a TM which needs no more than $3(L_n^2 + 2L_n) + 2$ for any input of length L_n = length($\mathcal{C}(n)$). Expressed formally, our TM needs at most time

(13) $T(M_{\text{square}}, \mathcal{C}, n) \leq 3 * (L_n^2 + 2L_n) + 2.$

Designing a TM that solves the satisfiability check problem is much more involved and I will refrain from doing this exercise in any detail. If one mimicks the "naive" procedure of computing the entire truth table, one will end up with some TM M_{SAT} that needs at most a number of working cycles which is exponential in the size L_φ = length($\mathcal{C}(\varphi)$):

(14) $T(M_{\text{SAT}}, \mathcal{C}, \varphi) \leq 2^{\alpha L_\varphi}$,

where α is some constant that depends on the concrete detail of the TM design. The reason for this exponential runtime is the sheer size of the truth table which grows exponentially with the number of Boolean variables in φ.

Here are two observations which constitute the basis for how computation runtimes are handled in theoretical CS:

1. The runtimes reported in (13) and (14) are *upper bounds*. We are not interested in the *exact* number of working cycles.
2. The runtimes are measured (or more precisely, the bounds are stated) as a function of the lengthes L of the input words. Different inputs which have the same length are treated identically with respect to their induced runtimes.

Furthermore, details don't matter. If we have a concrete TM M which computes f, then usually it is possible by some fine-tuning to improve the runtime of M a little. But we are not really interested in whether, for instance, $T(M_{\text{square}}, C, n) \leq 3 * (L_n^2 + 2L_n) + 2$ or whether $T(M_{\text{square}}, C, n) \leq 2.5 * (L_n^2 + 1.75 * L_n) + 4$. In fact, there exist generally applicable tuning methods for speeding up a TM by any constant factor – this is the *linear speedup theorem*:

Proposition 2.1 (linear speedup). If for some TM M which computes some function f with input coding C we have $T(M, C, a_i) \leq \gamma(L_{a_i})$, then for any *speedup factort* $s \geq 0$ one can design another TM M' whose runtime is bounded by

$$T(M', C, a_i) \leq s\gamma(L_{a_i}) + L_{a_i}.$$

This is actually not a deep theorem – the sped-up TM M' can be redesigned from M by admitting M to process entire blocks of symbols on its tape in a single working cycle, which leads to a new tape alphabet that has "block symbols".

In fact, an analog of the linear speedup theorem is effective in real digital computers. You may have heard of "32-bit" or "64-bit" CPUs. A 64-bit CPU processes "block symbols" which are binary strings of length 64 in single working cycles and is thereby twice as fast (needs half the number of working cycles) than a 32-bit CPU to carry out the same computation.

Because of the possibility of linear speedup, it is in many ways uninteresting to distinguish computation speeds which differ from each other only by a constant factor. To account for this indifference, a notation for specifying functions "up to some factor" has been introduced and is widely used in theoretical CS: the *Landau big-O* notation:

Definition 2.2 (Landau Big-O notation). Let $f, g: \mathbb{N} \to \mathbb{N}$ be two functions from the natural numbers to the natural numbers. Then we write $f = \mathrm{O}(g)$ if for some real-valued constant $c > 0$, and some $n_0 \in \mathbb{N}$, it holds that

(15) for all $n \geq n_0$: $f(n) \leq c\, g(n)$.

In words, $f = \mathrm{O}(g)$ means that $f(n)$ is "ultimately" (for n sufficiently large) bounded by $g(n)$ up to a factor c.

http://en.wikipedia.org/wiki/Big_O_notation gives a much fuller account if you are interested.

Examples:

- $3 * (L_n^2 + 2L_n) + 2 = O(L_n^2)$. Comment: the "growth rate" of $3 * (L_n^2 + 2L_n) + 2$ is ultimately dominated up to a factor by the quadratic term.
- More generally, if $f(n) = a_k n^k + ... + a_1 n + a_0$ is a polynomial of degree k, then $f(n) = O(n^k)$. Comment: the growth rate of any polynomial is ultimately determined by its highest-order term.
- For any polynomial order k and any exponential growth rate $\alpha > 0$, $n^k = O(2^{\alpha n})$. Comment: any exponential function ultimately outgrows any polynomial. For instance, $n^{1000} < (2^{0.0001 * n})$ if n is sufficiently large!

We have seen that the square function can be computed in polynomial runtime $O(L_n^2)$ and the Boolean satisfiability problem can be decided in exponential runtime $O(2^{\alpha L_\varphi})$. It is not known whether the Boolean satisfiability problem can be decided in polynomial runtime (most researchers believe it can't). Computational tasks that need exponential runtime are considered to be practically *intractable*. When input lengthes L grow, the required runtime quickly "explodes". In contrast, polynomial runtimes are in general considered *tractable*.

For many real-life problems, only exponential-time algorithms are known (and in most cases one suspects that polynomial-time algorithms don't exist). Specifically, this occurs very often for problems that have a combinatorial optimization flavor, for example

- Given a set of machines and raw products and desired end products, find a temporal schedule of passing raw and intermediate products to machines such that the overall production time is minimized (a case of *scheduling* or *queuing* problems).
- Given a set of axioms and premises and a desired conclusion, find a mathematical proof which derives the conclusion from the axioms and premises (*automated theorem-proving*)
- Given an electronic circuit diagram, find a concrete physical microchip layout that minimizes wiring length, or the number of necessary wire crossings (which are difficult/expensive to realize in hardware) (*chip configuration*)

In such situations, which abound, algorithm designers usually must resort to either simplifying the original task until it becomes computationally tractable, or be satisfied with *approximate* algorithms which most of the time return results that are close to the optimum.