

07-380 Notes
Pre-reading: Propositional Logic

Carnegie Mellon University
School of Computer Science

Contents

1	Propositional Logic	2
1.1	Syntax: Symbols, Operators, and Sentences	2
1.2	Truth Tables	3
1.3	More Propositional Logic Vocabulary	4
1.4	Conjunctive Normal Form (CNF)	4
1.4.1	Definition	4
1.4.2	Steps to Convert a Sentence to CNF	5
1.4.3	Any Propositional Logic Sentence to CNF	5
2	Models, Knowledge Bases, and Queries	6
2.1	Example: Candy, Sickness, and Lecture	6
3	Satisfiability and Entailment	7
3.1	Satisfiability	7
3.2	Entailment	7
3.3	Example: Before and After an Observation	8
4	Model Checking	10
4.1	Evaluating a Sentence in a Model	10
4.2	Enumerating Models with Depth-First Search	11
4.3	Satisfiability by Model Checking	11
4.4	Entailment by Model Checking	12
4.5	Time and Space Complexity	13

1 Propositional Logic

So far our agents have found solutions by *searching*: enumerate states, apply actions, and look for a goal. Sometimes, though, the hard part is not finding a sequence of actions but figuring out *what is true* given what we already know.

Propositional logic is a language for writing down what an agent knows, precisely enough that a computer can reason about it. Rather than searching for a path, a logical agent keeps everything it has been told in a knowledge base and works out what must follow from it.

These notes cover the language itself (this section), the vocabulary for talking about what an agent knows and what we want to ask it (Section 2), the two central questions we ask of a knowledge base (Section 3), and the most direct algorithmic way to answer them (Section 4).

1.1 Syntax: Symbols, Operators, and Sentences

A **symbol** is a variable that is either True or False. By convention we write symbols with a capital letter, using a longer name when it helps us remember what the symbol stands for:

$A, B, C, \text{ Rain, Windy}$

Each symbol should stand for a single, unambiguous statement about the world. *Rain* might mean “it is raining here today.” There is no notion of a symbol being “partly” true; propositional logic is strictly two-valued. It is also common to include the two constants *True* and *False* for convenience.

Operators (also called connectives) combine sentences into larger sentences:

Operator	Read as	Name	Arity
$\neg\alpha$	“not α ”	negation	unary
$\alpha \wedge \beta$	“ α and β ”	conjunction	binary
$\alpha \vee \beta$	“ α or β ”	disjunction	binary
$\alpha \Rightarrow \beta$	“ α implies β ”	implication	binary
$\alpha \Leftrightarrow \beta$	“ α if and only if β ”	biconditional	binary

Notice that the operators are defined in terms of α and β , not in terms of symbols. Greek letters are our convention for *arbitrary sentences*, which may themselves be large. This is what lets us build sentences of any size.

A **sentence** is defined constructively:

- **Base case:** any symbol X is a sentence (as are the constants *True* and *False*).
- **Recursive cases:** if α and β are sentences, then so are $\neg\alpha$, $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $(\alpha \Rightarrow \beta)$, and $(\alpha \Leftrightarrow \beta)$.
- Nothing else is a sentence.

So $\neg A \vee (B \wedge \neg C)$ is a sentence, and so is $(A \Rightarrow B) \Leftrightarrow (\neg B \Rightarrow \neg A)$, but $A \wedge \vee B$ is not. This recursive definition matters for more than tidiness: every algorithm we write for propositional logic will recurse over sentences following exactly this structure.

Precedence. When parentheses are omitted, the standard precedence from tightest to loosest is

$\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$

so $\neg A \wedge B \vee C \Rightarrow D$ means $((\neg A) \wedge B) \vee C \Rightarrow D$. When in doubt, add parentheses.

1.2 Truth Tables

Syntax tells us which strings are sentences; it says nothing about what they *mean*. The meaning of each operator is given by a **truth table**, which lists the value of the compound sentence for every combination of values of its parts.

α	β	$\neg\alpha$	$\alpha \wedge \beta$	$\alpha \vee \beta$	$\alpha \Rightarrow \beta$	$\alpha \Leftrightarrow \beta$
F	F	T	F	F	T	T
F	T	T	F	T	T	F
T	F	F	F	T	F	F
T	T	F	T	T	T	T

Three points about this table are worth dwelling on.

Disjunction is inclusive. $\alpha \vee \beta$ is true when *both* α and β are true. It is not the exclusive “or” of everyday English (“you can have soup or salad”). If you want exclusive or, you have to write it: $(\alpha \vee \beta) \wedge \neg(\alpha \wedge \beta)$.

Implication is not causation. $\alpha \Rightarrow \beta$ makes no claim that α brings about β ; it only rules out the single combination α true and β false. In particular *False* $\Rightarrow$ *anything* is true. The sentence “if the moon is made of cheese then $2 + 2 = 5$ ” is true in propositional logic, which feels strange until you read $\alpha \Rightarrow \beta$ as the promise “I am not claiming a case where α holds but β fails.” This is exactly the behavior captured by the equivalence

$$\alpha \Rightarrow \beta \equiv \neg\alpha \vee \beta$$

which you can verify column by column:

α	β	$\alpha \Rightarrow \beta$	$\neg\alpha$	$\neg\alpha \vee \beta$
F	F	T	T	T
F	T	T	T	T
T	F	F	F	F
T	T	T	F	T

The third and fifth columns match in every row. That is what we mean when we say two sentences are **logically equivalent**: they take the same truth value in *every* assignment of values to their symbols. We will use this particular equivalence constantly, because it is how we get rid of implications when converting to a normal form.

The biconditional is two implications. Similarly,

$$\alpha \Leftrightarrow \beta \equiv (\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$$

α	β	$\alpha \Leftrightarrow \beta$	$\alpha \Rightarrow \beta$	$\beta \Rightarrow \alpha$	$(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
F	F	T	T	T	T
F	T	F	T	F	F
T	F	F	F	T	F
T	T	T	T	T	T

A truth table with n distinct symbols has 2^n rows, because each symbol can independently be True or False. Keep that number in mind; it comes back as the running time of our first algorithm.

1.3 More Propositional Logic Vocabulary

A handful of terms describe sentences with particular shapes. These shapes matter because several important algorithms only accept input of a particular shape.

Literal. A **literal** is an atomic sentence: a symbol, a negated symbol, or one of the constants. A , $\neg A$, *True*, and *False* are all literals; $A \wedge B$ is not. A literal is **positive** if the symbol is not negated (A) and **negative** if it is ($\neg A$).

Conjunction and disjunction. A **conjunction** is sentences joined by $\wedge$ (an “AND of things”); a **disjunction** is sentences joined by $\vee$ (an “OR of things”). The pieces being joined are called the *conjuncts* and *disjuncts* respectively.

Clause. A **clause** is a disjunction of literals. For example:

$$\alpha_1 : A \vee \neg B \vee \neg C \qquad \alpha_2 : A \vee B \vee \neg C$$

Both are clauses. A single literal on its own also counts as a clause (a *unit clause*).

Definite clause. A **definite clause** is a clause with *exactly one* positive literal. Above, α_1 is a definite clause (only A is positive) while α_2 is not (both A and B are positive). Definite clauses are important because they can always be rewritten as an implication whose premise is a conjunction of positive literals:

$$A \vee \neg B \vee \neg C \equiv \neg(B \wedge C) \vee A \equiv (B \wedge C) \Rightarrow A$$

That is, a definite clause is exactly a rule of the form “if all of these things hold, then this other thing holds.” Knowledge bases built only from definite clauses admit much faster inference algorithms, which you will see in lecture.

Horn clause. A **Horn clause** is a clause with *at most one* positive literal. Every definite clause is a Horn clause; the extra cases are clauses with no positive literal at all, such as $\neg A \vee \neg B$, which can be read as the constraint $(A \wedge B) \Rightarrow \text{False}$.

Sentence	Clause?	Definite?	Horn?
A	yes	yes	yes
$\neg A$	yes	no	yes
$A \vee \neg B \vee \neg C$	yes	yes	yes
$\neg A \vee \neg B$	yes	no	yes
$A \vee B \vee \neg C$	yes	no	no
$A \wedge B$	no	n/a	n/a
$A \vee (B \wedge C)$	no	n/a	n/a

1.4 Conjunctive Normal Form (CNF)

1.4.1 Definition

A sentence is in **conjunctive normal form (CNF)** if it is a conjunction of clauses. Unpacking the definitions one level at a time:

- A CNF sentence is a **conjunction (AND)** of **clauses**.
- Each clause is a **disjunction (OR)** of **literals**.
- Each literal is a symbol or a negated symbol.

In other words, CNF is an *AND of ORs*, and nothing may nest more deeply than that. For example,

$$(A \vee \neg B \vee \neg C) \wedge (A \vee B \vee \neg C) \wedge (\neg A)$$

is in CNF (three clauses), while $A \vee (B \wedge C)$ is not, because an AND appears inside an OR.

Why do we care about a specific syntactic shape? Because several of the algorithms we rely on, including the SAT solvers and theorem provers you will meet in lecture, require their input in CNF. They exploit the flat structure: a CNF sentence is true in a model exactly when *every* clause has at least one true literal, which is a very easy condition to check and to reason about incrementally.

The reason this restriction costs us nothing is that **every sentence of propositional logic has a logically equivalent sentence in CNF**. So we never have to choose between expressiveness and being able to run our algorithms. We can write knowledge down in whatever form is natural and convert it afterwards.

1.4.2 Steps to Convert a Sentence to CNF

The conversion is a fixed sequence of rewrites, each of which replaces a sentence with a logically equivalent one:

1. **Eliminate $\Leftrightarrow$** . Replace $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.
2. **Eliminate $\Rightarrow$** . Replace $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$.
3. **Move $\neg$ inward** until it applies only to symbols, using De Morgan's laws and double-negation elimination:

$$\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta \quad \neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta \quad \neg\neg\alpha \equiv \alpha$$

4. **Distribute $\vee$ over $\wedge$** to flatten the sentence into an AND of ORs:

$$\alpha \vee (\beta \wedge \gamma) \equiv (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$$

5. **Tidy up**. Flatten nested $\wedge$ s and $\vee$ s, and optionally drop duplicate literals and clauses containing both X and $\neg X$ (such clauses are always true, so they can be removed).

The order matters. Steps 1 and 2 remove the operators that step 3 does not know how to push a negation through, and step 3 must finish before step 4, since distribution assumes negations already sit on the symbols.

1.4.3 Any Propositional Logic Sentence to CNF

Here is a full worked example. Take the sentence

$$(A \wedge B) \Leftrightarrow C$$

which is small but still needs all four of the main conversion steps.

Step 1: eliminate $\Leftrightarrow$.

$$((A \wedge B) \Rightarrow C) \wedge (C \Rightarrow (A \wedge B))$$

Step 2: eliminate $\Rightarrow$ (both of them):

$$(\neg(A \wedge B) \vee C) \wedge (\neg C \vee (A \wedge B))$$

Step 3: move $\neg$ inward. De Morgan's law turns $\neg(A \wedge B)$ into $\neg A \vee \neg B$:

$$(\neg A \vee \neg B \vee C) \wedge (\neg C \vee (A \wedge B))$$

Step 4: distribute $\vee$ over $\wedge$. The first conjunct is already a clause. In the second, $\neg C$ distributes across $A \wedge B$:

$$(\neg A \vee \neg B \vee C) \wedge (\neg C \vee A) \wedge (\neg C \vee B)$$

The result is a conjunction of three clauses, each a disjunction of literals: CNF. It is worth checking that this says what the original said. If C is true, the last two clauses force A and B to be true; if C is false, the first clause forbids A and B from both being true. That is exactly the claim that $A \wedge B$ and C have the same truth value.

Note also that the sentence got longer. That is typical, and in the worst case step 4 can blow the sentence up exponentially, which is worth remembering when you are tempted to write a very deeply nested sentence.

A useful exercise: pick any sentence you like, convert it to CNF by these steps, and then build truth tables for the original and the result to confirm that the two columns match in every row.

2 Models, Knowledge Bases, and Queries

We now have a language. Next we need vocabulary for talking about *what an agent knows* and *what it wants to find out*.

Model. A **model** is a complete assignment of every symbol to True or False. “Complete” is the important word: a model leaves nothing unassigned. If our world has symbols $\{X, Y, Z\}$, then

$$m = \{X=\text{True}, Y=\text{False}, Z=\text{False}\}$$

is one model, and there are $2^3 = 8$ models in total. In general n symbols give 2^n models. You can think of a model as one complete “possible world”: one specific way things might be.

Given a model, every sentence has a definite truth value: just look up each symbol and apply the operator truth tables. If sentence α is true in model m , we say m **satisfies** α , and we call m an α -*model*. We write $\text{models}(\alpha)$ for the set of all models that satisfy α .

Knowledge base. A **knowledge base** (KB) is a collection of sentences representing everything the agent knows about the world. The sentences in a KB are implicitly conjoined: a KB containing $\{\alpha_1, \alpha_2, \alpha_3\}$ is just shorthand for the single sentence $\alpha_1 \wedge \alpha_2 \wedge \alpha_3$. A knowledge base typically mixes two kinds of content:

- **Rules:** general statements that are always true, such as the rules of the game being played or the physics of the environment.
- **Facts:** specific things this agent has been told or has perceived, describing the particular situation it is in.

Query. A **query** is a sentence we want to know about. Given the KB , we want to determine whether the query follows from what the agent knows. Section 3 makes “follows from” precise.

2.1 Example: Candy, Sickness, and Lecture

Let us make this concrete with a small knowledge base about a day in the life of Dippy the diplodocus. We use three symbols:

- C : “Dippy ate too much candy today”
- S : “Dippy is sick today”

- L : “Dippy goes to lecture today”

The agent has been told two general rules and one specific fact:

	Sentence	In English
Rule	$C \Rightarrow S$	Eating too much candy makes you sick
Rule	$S \Rightarrow \neg L$	When you are sick you do not go to lecture
Fact	C	Dippy ate too much candy today

Conjoining them gives the knowledge base

$$KB = C \wedge (C \Rightarrow S) \wedge (S \Rightarrow \neg L)$$

That is the whole knowledge base. It is worth being clear about what it is and what it is not: it is a record of the sentences the agent has been told, written in a language with precise meaning. Nothing has been combined, simplified, or concluded. In particular, the knowledge base does not contain a sentence saying that Dippy is sick, even though that seems to follow from what is there. Working out what follows is a separate job, and the rest of these notes is about how to do it.

3 Satisfiability and Entailment

3.1 Satisfiability

A sentence α is **satisfiable** if there exists at least one model that makes it true, i.e. $models(\alpha) \neq \emptyset$. It is **unsatisfiable** if no model makes it true, and **valid** if *every* model makes it true.

- $A \wedge \neg B$ is satisfiable: the model $\{A=T, B=F\}$ satisfies it. To show satisfiability we only need to exhibit *one* model.
- $A \wedge \neg A$ is unsatisfiable: no assignment to A makes it true.
- $A \vee \neg A$ is valid: every assignment makes it true.

Satisfiability is a property of a *single* sentence. Note also the asymmetry in how much work the two answers take: proving satisfiability requires one witness, while proving *unsatisfiability* requires ruling out every model. The problem of deciding whether a sentence in CNF is satisfiable is the **Boolean satisfiability problem (SAT)**, and a program that answers it is a **SAT solver**.

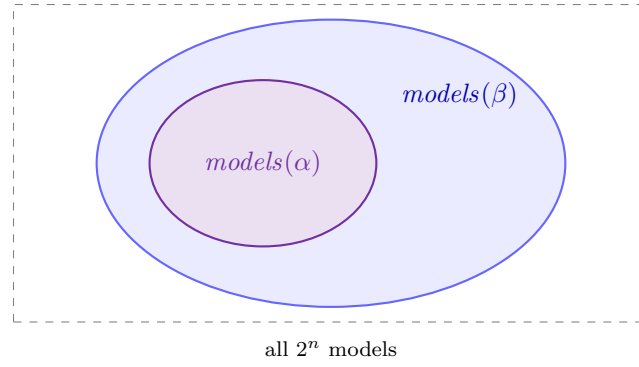
3.2 Entailment

Entailment relates *two* sentences. We say α entails β , written

$$\alpha \models \beta$$

if and only if every model that satisfies α also satisfies β . Equivalently, in terms of model sets:

$$\alpha \models \beta \iff models(\alpha) \subseteq models(\beta)$$



Read the picture carefully: $\alpha \models \beta$ says the α -models are *contained in* the β -models. There may be plenty of β -models outside $models(\alpha)$; entailment does not care. Note also that a *smaller* model set means a *stronger* sentence: the more a sentence says, the fewer worlds satisfy it, and the more it entails.

Entailment and satisfiability are two views of the same thing. Since $\alpha \models \beta$ means no model satisfies α while falsifying β , we have:

$$\alpha \models \beta \quad \text{if and only if} \quad \alpha \wedge \neg\beta \text{ is unsatisfiable}$$

This identity is the bridge between the two problems, and it is how a SAT solver gets used to answer entailment questions.

3.3 Example: Before and After an Observation

The knowledge base of Section 2.1 was not handed to the agent all at once. The two rules are general facts about the world that the agent has known all along, while C is something it observed at some point during the day. Looking at what the agent could conclude before and after that observation separates our two questions cleanly.

Before the observation. Early in the day the agent knows only the two rules:

$$KB_{\text{before}} = (C \Rightarrow S) \wedge (S \Rightarrow \neg L)$$

Which models satisfy it? A model falsifies $C \Rightarrow S$ exactly when C is true and S is false, and it falsifies $S \Rightarrow \neg L$ exactly when S and L are both true. Two models fall into the first case and two into the second, and no model falls into both, so four of the eight survive:

$$(C, S, L) \in \{(T, T, F), (F, T, F), (F, F, T), (F, F, F)\}$$

Four possible worlds remain, and they disagree with one another. Rules alone do not pin down much: nothing here says that candy is the only way to get sick, or that a healthy Dippy has to show up. In one of the four worlds Dippy is at lecture; in the other three Dippy is not. So the two questions come apart:

- $KB_{\text{before}} \wedge L$ is **satisfiable**. The model (F, F, T) satisfies both, so nothing the agent knows rules out Dippy being at lecture.
- $KB_{\text{before}} \not\models \neg L$. That same model satisfies KB_{before} while falsifying $\neg L$, which makes it a counterexample.

This is the distinction in its sharpest form. The rules are perfectly consistent with Dippy staying home, but they do not *force* it. Satisfiability asks whether *some* world works; entailment asks

whether *every* world does. That same model settles another question along the way: (F, F, T) also falsifies S , so $KB_{\text{before}} \not\models S$ and the agent cannot yet conclude that Dippy is sick either.

After the observation. The agent now sees Dippy eat too much candy and adds C to what it knows, giving exactly the knowledge base of Section 2.1:

$$KB_{\text{after}} = KB_{\text{before}} \wedge C = C \wedge (C \Rightarrow S) \wedge (S \Rightarrow \neg L)$$

Adding a sentence can only remove models, never add them, since each model now has one more condition to meet. Of the four models above, only (T, T, F) has C true, so exactly one survives:

$$(C, S, L) = (T, T, F)$$

With a single possible world left, the answers change:

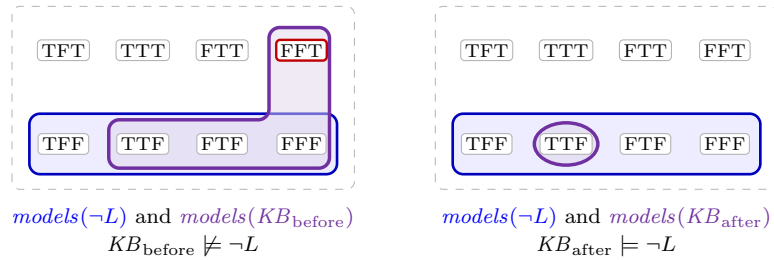
- $KB_{\text{after}} \models S$. Dippy is sick. Nobody told the agent this; it follows from the fact together with the first rule.
- $KB_{\text{after}} \models \neg L$. Dippy is not at lecture. This one took both rules and the fact.
- $KB_{\text{after}} \wedge L$ is **unsatisfiable**. The world in which Dippy was at lecture, (F, F, T), did not survive the new information.

Notice that the last two bullets say the same thing. That is the identity from Section 3.2: $KB_{\text{after}} \models \neg L$ holds exactly when $KB_{\text{after}} \wedge \neg(\neg L)$, which is $KB_{\text{after}} \wedge L$, is unsatisfiable.

Both knowledge bases are themselves satisfiable, which is worth checking in its own right. An unsatisfiable knowledge base means the agent has been told things that cannot all hold at once, and nothing useful can be concluded from it.

The same story as model sets

Here are both situations drawn in the style of the diagram in Section 3.2. The eight models are laid out so that the four with L false fill the bottom row, and each label gives (C, S, L) .



In (a) the single model lying outside $models(\neg L)$, outlined in red, is the counterexample that blocks the entailment.

In (a) the purple region pokes out of the blue one at exactly one model, (F, F, T): a world the rules allow in which Dippy is at lecture. That one model is the whole reason the entailment fails. Observing C shrinks the purple region to the single model (T, T, F), which lies inside the blue region, and the entailment holds.

This is the general shape of the thing. Every sentence added to a knowledge base shrinks its set of models, and a query becomes entailed at the moment that set has shrunk far enough to fit inside the query's region. Learning more does not change what the rules mean; it narrows the range of worlds still under consideration until they all agree on the answer.

4 Model Checking

We have definitions; now we need algorithms. **Model checking** is the most direct approach available: since a KB over n symbols admits only 2^n models, and since every sentence has a definite truth value in every model, we can simply *enumerate all the models and check*.

That plan needs two ingredients: a way to evaluate a sentence in a given model, and a way to enumerate models systematically. Throughout this section KB means the knowledge base of Section 2.1, the one that includes the fact C , which Section 3.3 called KB_{after} .

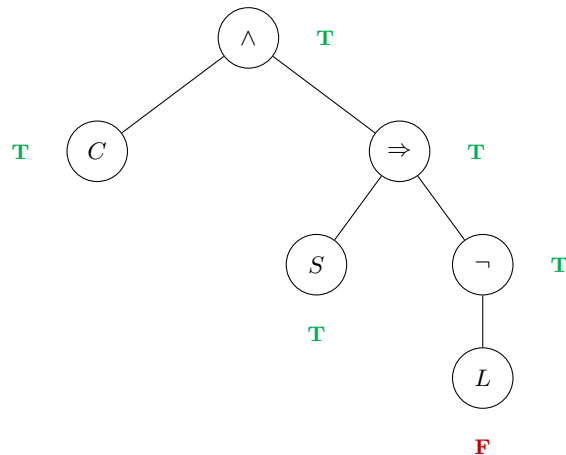
4.1 Evaluating a Sentence in a Model

The first ingredient is a function that takes a model and a sentence and returns whether the sentence is true in that model. We will call it `EVALUATE`; it is also commonly written `PL-TRUE?`. Its structure mirrors the recursive definition of a sentence exactly:

- **Base case.** If the sentence is a symbol, look up its value in the model and return it. (*True* and *False* return themselves.) The model is complete, so the lookup always succeeds.
- **Recursive case.** Otherwise the sentence has a top-level operator. Evaluate its argument (for $\neg$) or its two arguments (for $\wedge, \vee, \Rightarrow, \Leftrightarrow$) by calling `EVALUATE` on each, then combine those results using that operator's truth table.

Concretely, to evaluate $\alpha \wedge \beta$ you recursively evaluate α , recursively evaluate β , and return the AND of the two answers; to evaluate $\neg \alpha$ you recursively evaluate α and negate the result; and so on for each operator.

It helps to picture the sentence as a tree, with operators at the internal nodes and literals at the leaves. `EVALUATE` walks that tree bottom-up: values flow from the leaves toward the root, and the value that arrives at the root is the truth value of the whole sentence. Here is a piece of our knowledge base, the fact together with the second rule, evaluated in the model $\{C=T, S=T, L=F\}$:

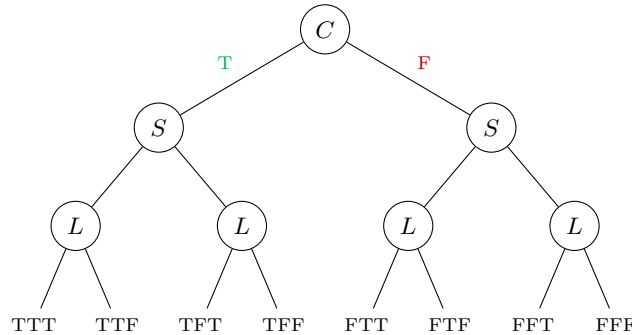


L is False in the model, so $\neg L$ is True. S is True, and $\text{True} \Rightarrow \text{True}$ is True. C is True, and $\text{True} \wedge \text{True}$ is True, so the sentence is true in this model.

`EVALUATE` is fast: it touches each node of the sentence once, so it runs in time linear in the size of the sentence. All the expense in model checking comes from the *number of models*, not from evaluating any one of them.

4.2 Enumerating Models with Depth-First Search

The second ingredient is a systematic way to walk through all 2^n models. The natural approach is depth-first search over partial assignments: pick an unassigned symbol, try it True, recurse; then try it False, recurse. This is the same recursion as backtracking search: the search tree is a complete binary tree of depth n , and its 2^n leaves are exactly the models.



Each internal node chooses a value for one symbol; the left branch assigns True and the right branch assigns False. Each root-to-leaf path fixes all three symbols, so each leaf is one complete model, written here as the triple (C, S, L) . With n symbols there are 2^n leaves, but only n frames on the recursion stack at any moment, so this enumeration takes exponential *time* but only linear *space*.

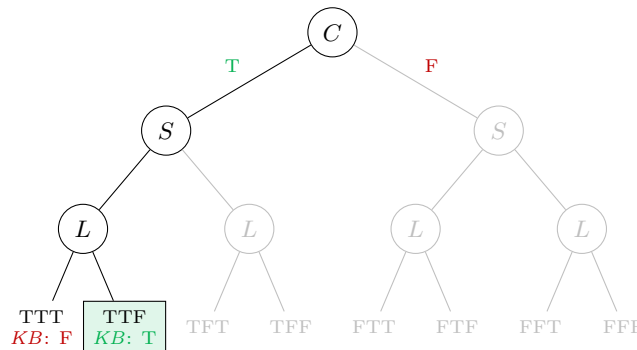
4.3 Satisfiability by Model Checking

Putting the ingredients together gives our first algorithm. To decide whether a sentence is satisfiable, run the depth-first enumeration above; at each leaf, call EVALUATE on the completed model.

- If the sentence is **true** in this model, we are done. The sentence is satisfiable, and we **return this model** as the witness. There is no reason to look at any remaining leaf.
- If the sentence is false, backtrack and continue with the next model.
- If the search exhausts every leaf without success, the sentence is unsatisfiable.

Note that returning the satisfying model, not just “yes,” is usually what we want in practice: for a planning problem encoded as a SAT instance, the model *is* the plan.

Running this on the *KB*, assigning symbols in the order C, S, L and trying True first:



The first leaf, (T, T, T) , falsifies the rule $S \Rightarrow \neg L$: Dippy is sick and yet at lecture. The second leaf,

(T, T, F), satisfies all three sentences, so the search returns immediately with that model. Six of the eight leaves, shown greyed out, are never visited.

4.4 Entailment by Model Checking

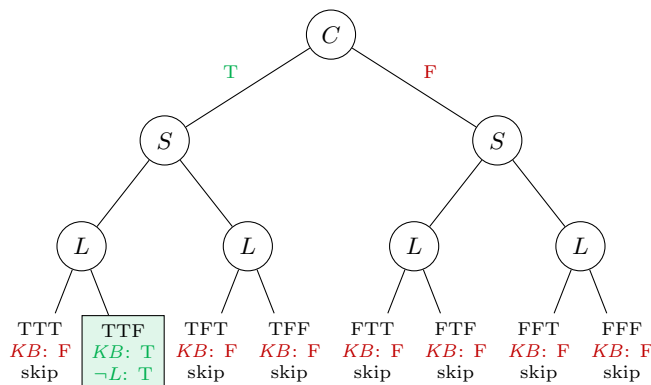
Deciding entailment uses the same enumeration but a different test at the leaves and, importantly, the opposite early-return condition. To decide whether $KB \models \alpha$, walk the same tree and at each leaf evaluate *both* sentences in the completed model:

- If KB is **false** in this model, the model is irrelevant. Entailment only constrains models where the KB holds, so skip it and continue.
- If KB is **true** and α is **true**, this model is consistent with entailment. Continue; we cannot conclude anything yet, because entailment is a claim about *all* models.
- If KB is **true** and α is **false**, we have found a **counterexample**: a world the KB permits in which α fails. **Return false** immediately; $KB \not\models \alpha$.
- If the search exhausts every leaf without finding a counterexample, then **return true**: $KB \models \alpha$.

Compare this with the satisfiability algorithm. Both walk the same tree, but they return early for opposite reasons, because they are answering opposite kinds of question:

	Satisfiability	Entailment
Claim being tested	<i>there exists</i> a model	<i>for all</i> models
Returns early on	the first satisfying model	the first counterexample
Early return means	satisfiable (returns that model)	not entailed (returns false)
Exhausting the tree means	unsatisfiable (false)	entailment holds (true)

Let us trace $KB \models \neg L$, the query asking whether Dippy stays home from lecture.



Seven of the eight leaves falsify the KB and are skipped: the four with C false contradict the fact, and three more contradict a rule. The one surviving leaf, (T, T, F), satisfies both the KB and the query, so the search runs out of leaves without ever finding a counterexample and returns true.

Note that unlike the satisfiability search, this one had to visit *every* leaf. That is the price of a “for all” claim. Had we instead asked whether $KB \models L$, the second leaf would have been a counterexample (KB true, L false) and the search would have returned false right there. Laying all eight leaves out in a table gives exactly the truth table we would have written by hand:

C	S	L	KB	S	$\neg L$	
T	T	T	F	T	F	
T	T	F	T	T	T	← the only KB -model
T	F	T	F	F	F	
T	F	F	F	F	T	
F	T	T	F	T	F	
F	T	F	F	T	T	
F	F	T	F	F	F	
F	F	F	F	F	T	

Reading only the rows where the KB is true, of which there is exactly one: both S and $\neg L$ are true there, so $KB \models S$ and $KB \models \neg L$. The depth-first algorithm computes precisely this, one row at a time, without ever storing the whole table.

4.5 Time and Space Complexity

Both algorithms visit up to 2^n leaves, where n is the number of symbols appearing in the KB and the query, and do work linear in the size of the sentences at each leaf. So model checking takes **exponential time** in the number of symbols. On the other hand, the depth-first recursion only ever holds one partial assignment, so it uses only **linear space**.

Exponential time is a serious limitation. Our example had three symbols and eight models, but a knowledge base with a mere 100 symbols has 2^{100} models, more than we could enumerate if we had started at the Big Bang. Yet real SAT solvers routinely handle problems with hundreds of thousands of symbols. How they manage that, and what other approaches exist for proving entailment without enumerating models at all, is the subject of lecture.