LOGICAL AGENTS

Prepared By:
Dipanita Saha
Assistant Professor
Institute of Information Technology(IIT)
Noakhali Science and Technology University
.

KNOWLEDGE-BASED AGENTS

- The central component of a knowledge-based agent is **its knowledge base, or kb**.
- A knowledge base is a set of **sentences**. Each sentence is expressed in a language called a **knowledge representation language** and represents some knowledge representation assertion about the world.
- Sometimes we dignify a sentence with the name **axiom**, when the sentence is taken as given without being derived from other sentences.
- There must be a way to add new sentences to the knowledge base and a way to query what is known. The standard names for these operations are TELL and ASK, respectively.
- Both operations may involve **inference**—that is, deriving new sentences from old.
- Inference must obey the requirement that when one ASKs a question of the knowledge base, the answer should follow from what has been told (or TELLed) to the knowledge base previously.
- The agent maintains a knowledge base, KB, which may initially contain some **background knowledge**.

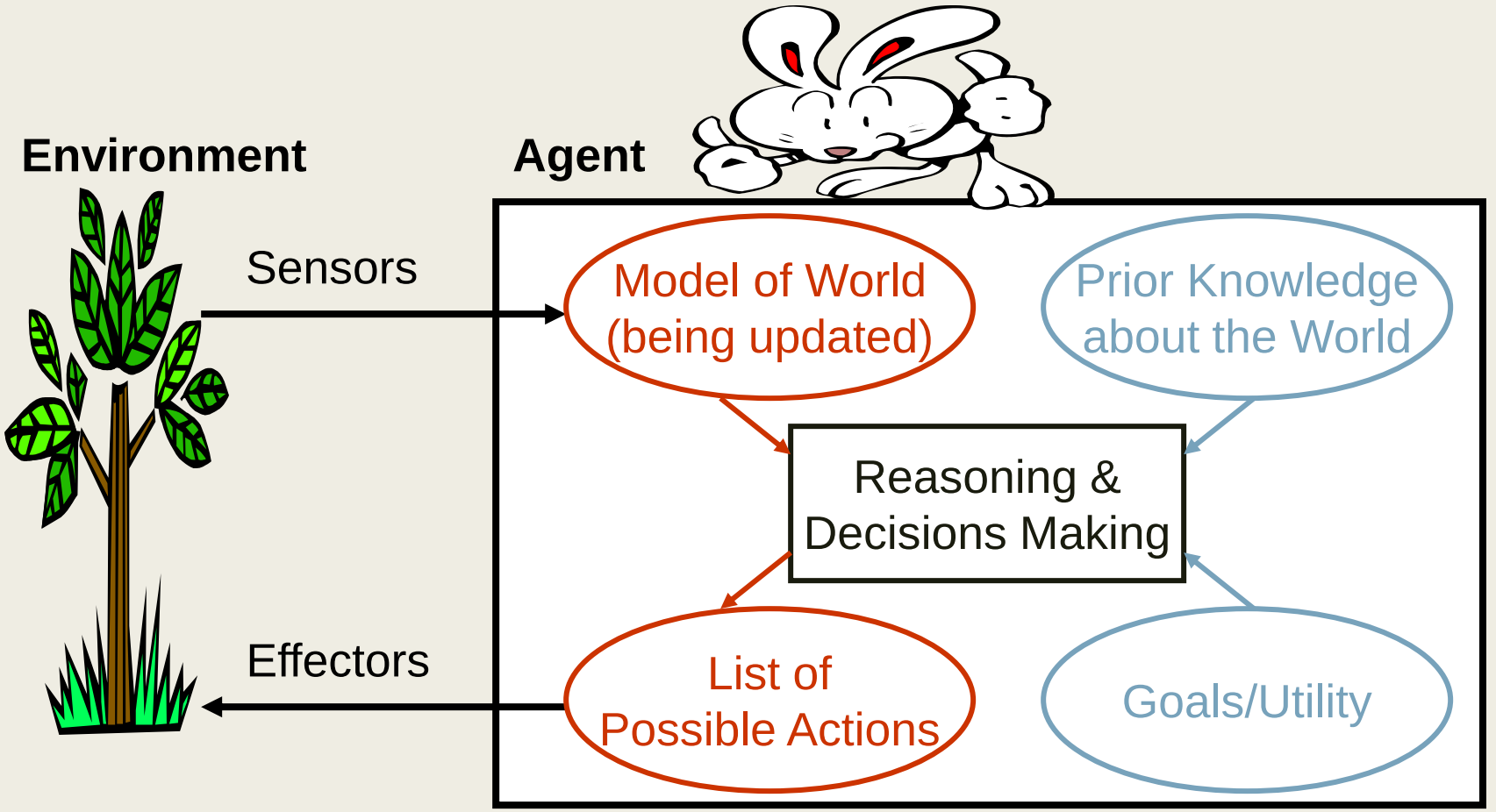
- Each time the agent program is called, it does three things.
- **First**, it TELLs the knowledge base what it perceives. **Second**, it ASKs the knowledge base what action it should perform. **Third**, the agent program TELLs the knowledge base which action was chosen, and the agent executes the action.
- In the process of answering this query, extensive reasoning may be done about the current state of the world, about the outcomes of possible action sequences, and so on.
- MAKE-PERCEPT-SENTENCE constructs a sentence asserting that the agent perceived the given percept at the given time.
- MAKE-ACTION-QUERY constructs a sentence that asks what action should be done at the current time.
- Finally, MAKE-ACTION-SENTENCE constructs a sentence asserting that the chosen action was executed.

```
function KB-AGENT(percept) returns an action
  persistent: KB, a knowledge base
              t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  action ← ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action
```

Figure 7.1 A generic knowledge-based agent. Given a percept, the agent adds the percept to its knowledge base, asks the knowledge base for the best action, and tells the knowledge base that it has in fact taken that action.

Architecture of a Simple Intelligent Agent



- The agent must be able to:
  - represent states, actions, etc.
  - incorporate new percepts
  - update internal representation of world
  - deduce hidden properties of world
  - deduce appropriate actions
- One of the core problems in developing an intelligent agent is knowledge representation:
  - how to represent knowledge
  - how to reason using that knowledge
- View of agent (levels of abstraction):
  - **knowledge level**:
    what the agent knows at a high level
  - **logic level**:
    level of sentence encoding
  - **implementation level**:
    level that runs on the architecture,
    detail of data structures and algorithms.

THE WUMPUS WORLD

- Here we describe an environment in which knowledge-based agents can show their worth.
- The **wumpus world** is a cave consisting of rooms connected by passageways. Lurking somewhere in the cave is the terrible wumpus, a beast that eats anyone who enters its room.
- The wumpus can be shot by an agent, but the agent has only one arrow. Some rooms contain bottomless pits that will trap anyone who wanders into these rooms
- The precise definition of the task environment is given by the PEAS description

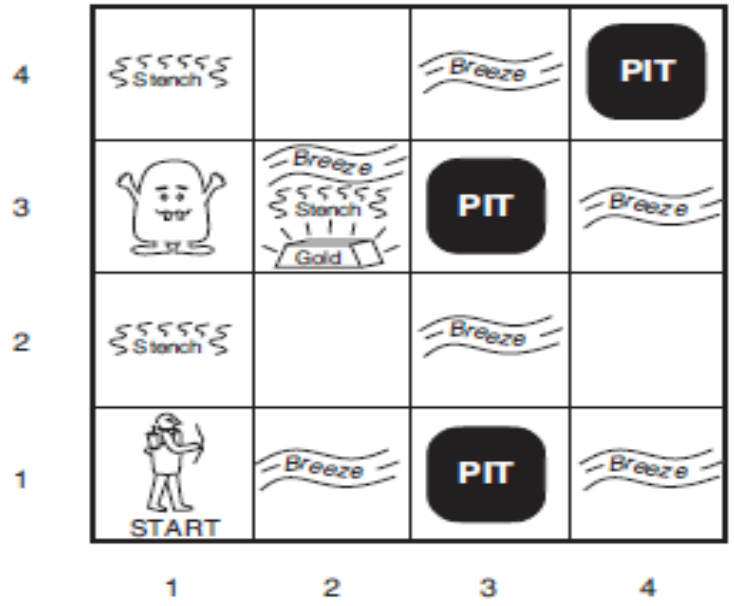


Figure 7.2 A typical wumpus world. The agent is in the bottom left corner, facing right.

- **Performance measure**: +1000 for climbing out of the cave with the gold, –1000 for falling into a pit or being eaten by the wumpus, –1 for each action taken and –10 for using up the arrow. The game ends either when the agent dies or when the agent climbs out of the cave.
- **Environment**: A 4×4 grid of rooms. The agent always starts in the square labeled [1,1], facing to the right. The locations of the gold and the wumpus are chosen randomly, with a uniform distribution, from the squares other than the start square. In addition, each square other than the start can be a pit, with probability 0.2.
- **Actuators**: The agent can move *Forward*, *TurnLeft* by 90◦, or *TurnRight* by 90◦. The agent dies a miserable death if it enters a square containing a pit or a live wumpus. If an agent tries to move forward and bumps into a wall, then the agent does not move.

The action *Grab* can be used to pick up the gold if it is in the same square as the agent. The action *Shoot* can be used to fire an arrow in a straight line in the direction the agent is facing.

The arrow continues until it either hits (and hence kills) the wumpus or hits a wall. The agent has only one arrow, so only the first *Shoot* action has any effect.

Finally, the action *Climb* can be used to climb out of the cave, but only from square [1,1].

- **Sensors**: The agent has five sensors, each of which gives a single bit of information:

1. In the square containing the wumpus and in the directly (not diagonally) adjacent squares, the agent will perceive a *Stench*.
2. In the squares directly adjacent to a pit, the agent will perceive a *Breeze*.
3. In the square where the gold is, the agent will perceive a *Glitter*.
4. When an agent walks into a wall, it will perceive a *Bump*.
5. When the wumpus is killed, it emits a woeful *Scream* that can be perceived anywhere in the cave.

- For an agent in the environment, the main challenge is its initial ignorance of the configuration of the environment; overcoming this ignorance seems to require logical reasoning.
- We use an informal knowledge representation language consisting of writing down symbols in a grid

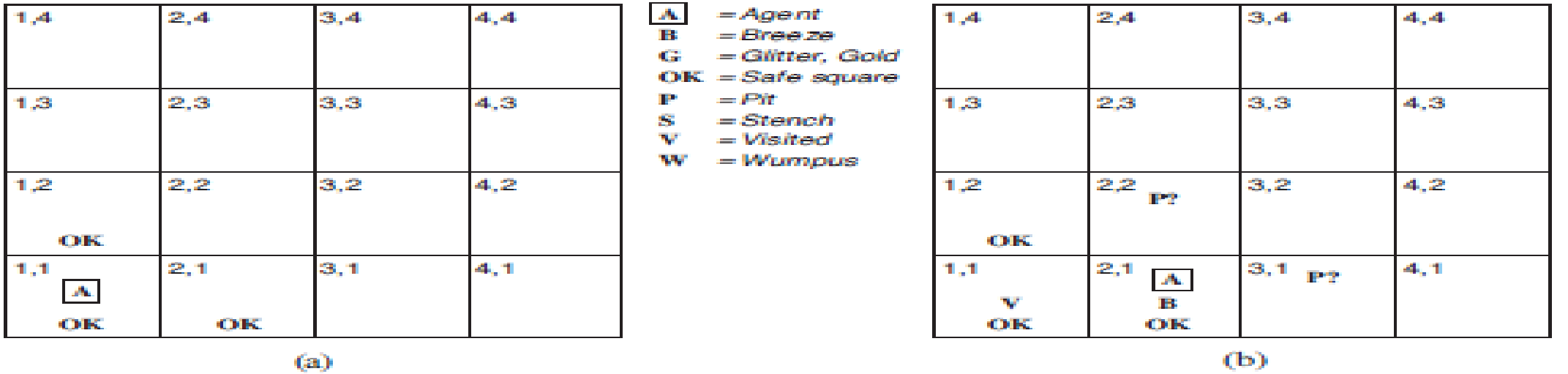


Figure 7.3 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept [*None*, *None*, *None*, *None*, *None*]. (b) After one move, with percept [*None*, *Breeze*, *None*, *None*, *None*].

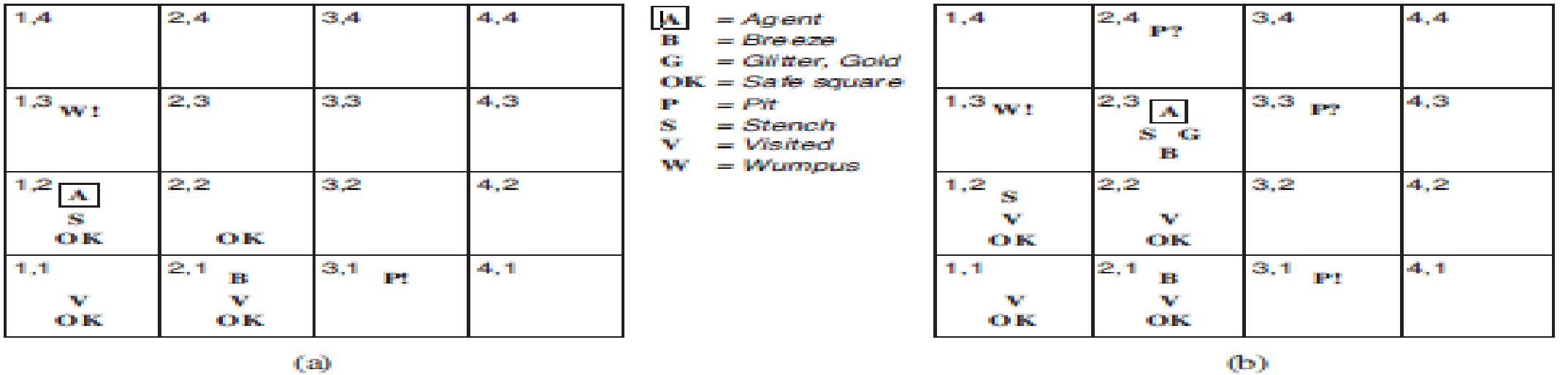


Figure 7.4 Two later stages in the progress of the agent. (a) After the third move, with percept [*Stench*, *None*, *None*, *None*, *None*]. (b) After the fifth move, with percept [*Stench*, *Breeze*, *Glitter*, *None*, *None*].

- The agent's initial knowledge base contains the rules of the environment, in particular, it knows that it is in [1,1] and that [1,1] is a safe square; we denote that with an "A" and "OK," respectively, in square [1,1].
- The first percept is [None, None, None, None, None], from which the agent can conclude that its neighboring squares, [1,2] and [2,1], are free of dangers—they are OK.
- A cautious agent will move only into a square that it knows to be OK.
- Let us suppose the agent decides to move forward to [2,1]. The agent perceives a breeze (denoted by "B") in [2,1], so there must be a pit in a neighboring square.
- The pit cannot be in [1,1], by the rules of the game, so there must be a pit in [2,2] or [3,1] or both.
- The notation "P?" in Figure 7.3(b) indicates a possible pit in those squares.
- At this point, there is only one known square that is OK and that has not yet been visited. So the prudent agent will turn around, go back to [1,1], and then proceed to [1,2].
- The stench in [1,2] means that there must be a wumpus nearby. But the wumpus cannot be in [1,1], by the rules of the game, and it cannot be in [2,2] (or the agent would have detected a stench when it was in [2,1]).

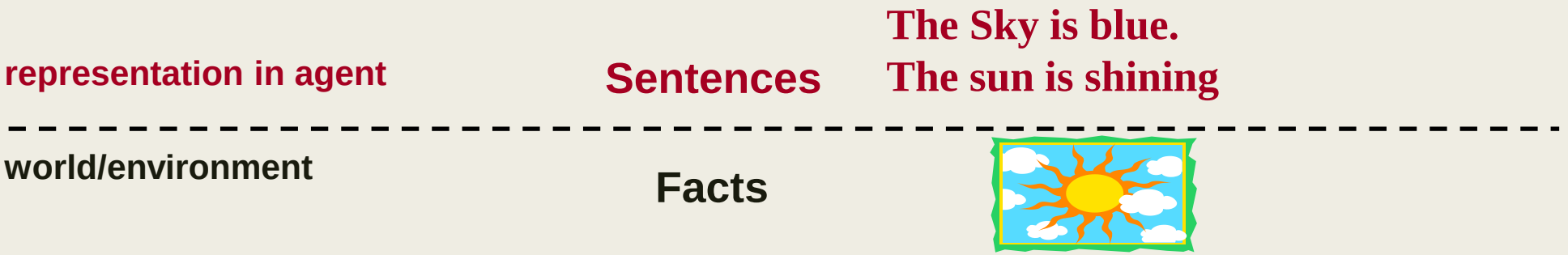
- Therefore, the agent can infer that the wumpus is in [1,3].
- The notation W! indicates this inference. Moreover, the lack of a breeze in [1,2] implies that there is no pit in [2,2].
- Yet the agent has already inferred that there must be a pit in either [2,2] or [3,1], so this means it must be in [3,1].
- The agent has now proved to itself that there is neither a pit nor a wumpus in [2,2], so it is OK to move there.
- We do not show the agent's state of knowledge at [2,2]; we just assume that the agent turns and moves to [2,3], giving us Figure 7.4(b).
- In [2,3], the agent detects a glitter, so it should grab the gold and then return home

LOGIC

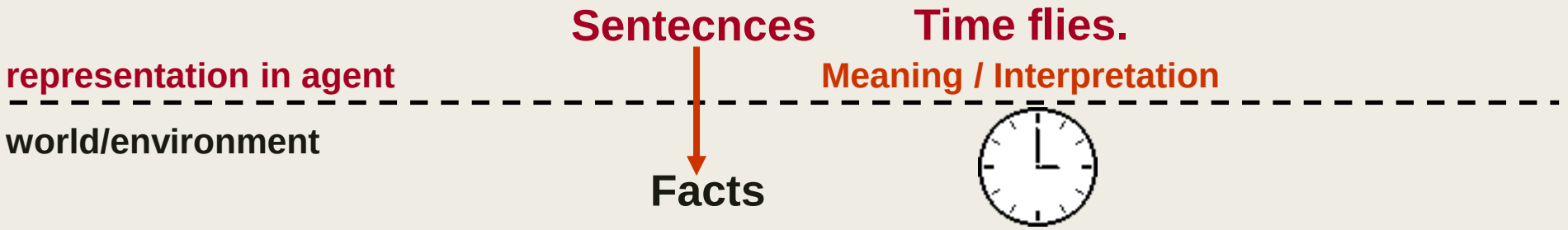
- Knowledge bases consist of sentences. These sentences are expressed according to the **syntax** of the representation language, which specifies all the sentences that are well formed.
- The notion of syntax is clear enough in ordinary arithmetic: “x + y = 4” is a well-formed sentence, whereas “x4y+ =” is not.
- A logic must also define the **semantics** or meaning of sentences. The semantics defines the **truth** of each sentence with respect to each **possible world**.
- For example, the semantics for arithmetic specifies that the sentence “x + y =4” is true in a world where x is 2 and y is 2, but false in a world where x is 1 and y is 1.
- In standard logics, every sentence must be either true or false in each possible world—there is no “in between.”
- When we need to be precise, we use the term **model** in place of “possible world.”
- If a sentence α is true in model m, we say that m **satisfies** α or sometimes m **is a model of** α. We use the notation M(α) to mean the set of all models of α.
- we have a notion of truth, we are ready to talk about logical reasoning. This involves the relation of logical **entailment** between sentences—the idea that a sentence *follows logically* from another sentence. In mathematical notation, we write α |= β

Logic

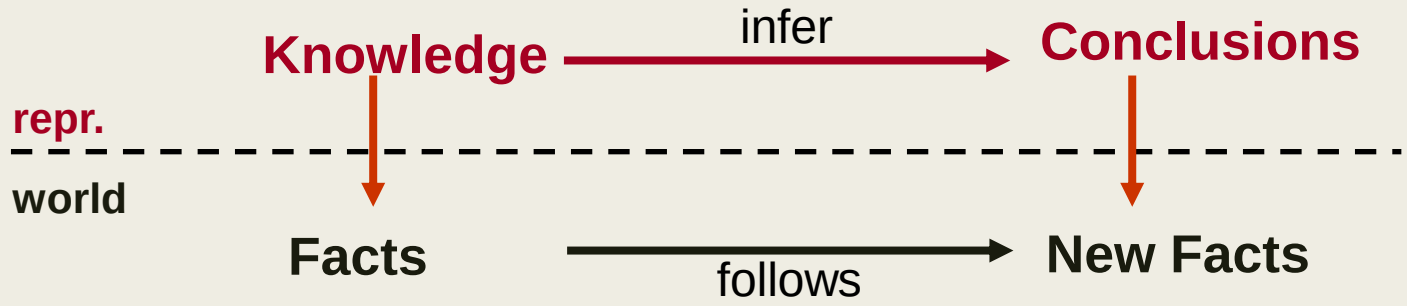
- The agent internally represents its world/environment in its knowledge base.
- **Sentences are representations in some language.**
- **Facts are claims about the world that are true/false.**



- Sentences represent facts in the world.
- **Meaning** connects sentences to their facts.
- A sentence is true if what it represents is actually the case in the current state of world.

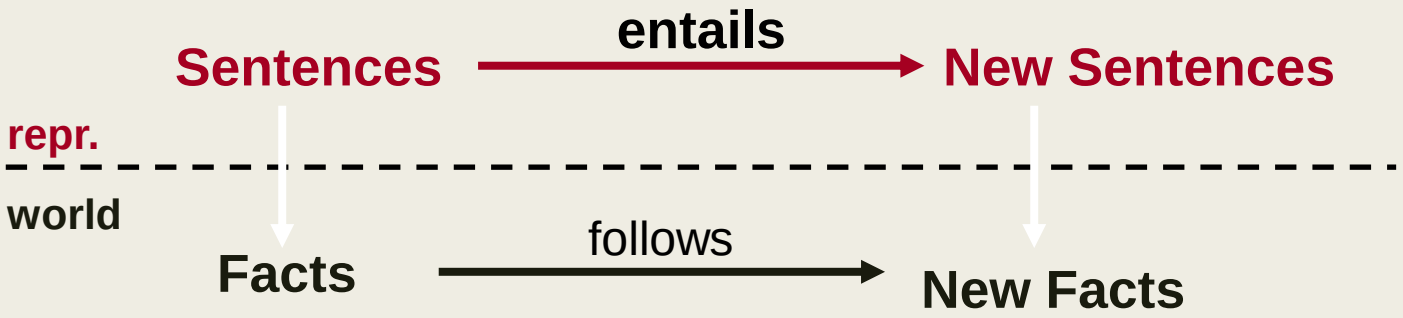


- Proper reasoning ensures that conclusions inferred from the KB are consistent with reality.
- That means they represent new facts that actually follow from the original facts (represented by sentences in the KB).



✱ *Computers don't know the meaning.*

●A mechanical inference procedure is needed that derives conclusions without needing to know the meaning of the sentences.



- Logics are characterized by
  what they commit to as "primitives".

Logic	What Exists in World	Knowledge States
Propositional	facts	true/false/unknown
First-Order	facts, objects, relations	true/false/unknown
Temporal	facts, objects, relations, times	true/false/unknown
Probability Theory	facts	degree of belief 0..1
Fuzzy	degree of truth	degree of belief 0..1

PROPOSITIONAL LOGIC

- Propositions: assertions about an aspect of a world that can be assigned either a true or false value
  - *e.g. SkyIsCloudy, JimIsHappy*
  - *True, False are propositions meaning true and false*

Syntax:

- The **syntax** propositional logic defines the allowable sentences.
- The **atomic sentences** consist of a single **proposition symbol**. Each such symbol stands for a proposition that can be true or false.
- We use symbols that start with an uppercase letter and may contain other letters or subscripts, for example: P, Q, R, W and North.
- The names are arbitrary but are often chosen to have some mnemonic value.
- There are two proposition symbols with fixed meanings: True is the always-true proposition and False is the always-false proposition.
- **Complex sentences** are constructed from simpler sentences, using parentheses and **logical connectives**.

- There are five connectives in common use:
- ¬ **(not):** A sentence such as ¬W1,3 is called the **negation** of W1,3. A **literal** is either an atomic sentence (a **positive literal**) or a negated atomic sentence (a **negative literal**).
- ∧ **(and):** A sentence whose main connective is ∧, such as W1,3 ∧ P3,1, is called a **con junction**; its parts are the **conjuncts**. (The ∧ looks like an "A" for "And.")
- ∨ **(or):** A sentence using ∨, such as (W1,3∧P3,1)∨W2,2, is a **disjunction** of the **disjuncts** (W1,3 ∧ P3,1) and W2,2.
- ⇒ **(implies):** A sentence such as (W1,3∧P3,1) ⇒ ¬W2,2 is called an **implication** (or conditional). Its **premise** or **antecedent** is (W1,3 ∧P3,1), and its **conclusion** or **consequen** is ¬W2,2. Implications are also known as **rules** or **if–then** statements. The implication symbol is sometimes written in other books as ⊃ or →.
- ⇔ **(if and only if):** The sentence W1,3 ⇔ ¬W2,2 is a **biconditional**. Some other books write this as ≡.

- **Semantics**
- The semantics defines the rules for determining the truth of a sentence with respect to a particular model.
- In propositional logic, a model simply fixes the **truth value**—true or false—for every proposition symbol.
- For example, if the sentences in the knowledge base make use of the proposition symbols P1,2, P2,2, and P3,1, then one possible model is $m_1 = \{P_{1,2} = false,\ P_{2,2} = false,\ P_{3,1} = true\}$.
- With three proposition symbols, there are 2^3 =8 possible models.
- P1,2 is just a symbol; it might mean "there is a pit in [1,2]" or "I'm in Paris today and tomorrow."
- The semantics for propositional logic must specify how to compute the truth value of *any* sentence, given a model.
- All sentences are constructed from atomic sentences and the five connectives. Atomic sentences are easy:

❖ True is true in every model and False is false in every model.

❖ The truth value of every other proposition symbol must be specified directly in the model. For example, in the model m1 given earlier, P1,2 is false.

- For complex sentences, we have five rules, which hold for any subsentences P and Q in any model m (here "iff" means "if and only if"):

- $\neg P$ is true iff P is false in m.
- $P \wedge Q$ is true iff both P and Q are true in m.
- $P \vee Q$ is true iff either P or Q is true in m.
- $P \Rightarrow Q$ is true unless P is true and Q is false in m.
- $P \Leftrightarrow Q$ is true iff P and Q are both true or both false in m.

- The rules can also be expressed with **truth tables.** From these tables, the truth value of any sentence *s* can be computed with respect to any model *m* by a simple recursive evaluation. For example

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

Figure 7.8 Truth tables for the five logical connectives. To use the table to compute, for example, the value of $P \vee Q$ when P is true and Q is false, first look on the left for the row where P is *true* and Q is *false* (the third row). Then look in that row under the $P \vee Q$ column to see the result: *true*.

- the sentence ¬P1,2 ∧ (P2,2 ∨ P3,1), evaluated in m1, gives true ∧ (false ∨ true)=true ∧ true =true

A simple knowledge base

- For now, we need the following symbols for each [x, y] location:

$P_{x,y}$ is true if there is a pit in $[x, y]$.
$W_{x,y}$ is true if there is a wumpus in $[x, y]$, dead or alive.
$B_{x,y}$ is true if the agent perceives a breeze in $[x, y]$.
$S_{x,y}$ is true if the agent perceives a stench in $[x, y]$.

The sentences we write will suffice to derive $\neg P_{1,2}$ (there is no pit in [1,2]), as was done informally in Section 7.3. We label each sentence R_i so that we can refer to them:

- There is no pit in [1,1]:

$$R_1 : \quad \neg P_{1,1} .$$

- A square is breezy if and only if there is a pit in a neighboring square. This has to be stated for each square; for now, we include just the relevant squares:

$$R_2 : \quad B_{1,1} \quad \Leftrightarrow \quad (P_{1,2} \vee P_{2,1}) .$$
$$R_3 : \quad B_{2,1} \quad \Leftrightarrow \quad (P_{1,1} \vee P_{2,2} \vee P_{3,1}) .$$

- The preceding sentences are true in all wumpus worlds. Now we include the breeze percepts for the first two squares visited in the specific world the agent is in, leading up to the situation in Figure 7.3(b).

$$R_4 : \quad \neg B_{1,1} .$$
$$R_5 : \quad B_{2,1} .$$

PROPOSITIONAL THEOREM PROVING

- **Theorem proving**—applying rules of inference directly to the sentences in our knowledge base to construct a proof of the desired sentence without consulting models.
- If the number of models is large but the length of the proof is short, then theorem proving can be more efficient than model checking.
- we will need some additional concepts related to entailment.
- The first concept is **logical equivalence**: two sentences α and β are logically equivalent if they are true in the same set of models. We write this as $\alpha \equiv \beta$.
- For example, we can easily show (using truth tables) that $P \wedge Q$ and $Q \wedge P$ are logically equivalent; other equivalences are shown in Figure

$$
\begin{aligned}
(\alpha \wedge \beta) &\equiv (\beta \wedge \alpha) && \text{commutativity of } \wedge \\
(\alpha \vee \beta) &\equiv (\beta \vee \alpha) && \text{commutativity of } \vee \\
((\alpha \wedge \beta) \wedge \gamma) &\equiv (\alpha \wedge (\beta \wedge \gamma)) && \text{associativity of } \wedge \\
((\alpha \vee \beta) \vee \gamma) &\equiv (\alpha \vee (\beta \vee \gamma)) && \text{associativity of } \vee \\
\neg(\neg\alpha) &\equiv \alpha && \text{double-negation elimination} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\beta \Rightarrow \neg\alpha) && \text{contraposition} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\alpha \vee \beta) && \text{implication elimination} \\
(\alpha \Leftrightarrow \beta) &\equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) && \text{biconditional elimination} \\
\neg(\alpha \wedge \beta) &\equiv (\neg\alpha \vee \neg\beta) && \text{De Morgan} \\
\neg(\alpha \vee \beta) &\equiv (\neg\alpha \wedge \neg\beta) && \text{De Morgan} \\
(\alpha \wedge (\beta \vee \gamma)) &\equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) && \text{distributivity of } \wedge \text{ over } \vee \\
(\alpha \vee (\beta \wedge \gamma)) &\equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) && \text{distributivity of } \vee \text{ over } \wedge
\end{aligned}
$$

Figure 7.11 Standard logical equivalences. The symbols α, β, and γ stand for arbitrary sentences of propositional logic.

- An alternative definition of equivalence is as follows: any two sentences α and β are equivalent only if each of them entails the other:

$$\alpha \equiv \beta \quad \text{if and only if} \quad \alpha \models \beta \text{ and } \beta \models \alpha .$$

- The second concept we will need is **validity**. A sentence is valid if it is true in *all* models. For example, the sentence P ∨ ¬P is valid. Valid sentences are also known as **tautologies**—they are *necessarily* true.
- Because the sentence True is true in all models, every valid sentence is logically equivalent to True
- From our definition of entailment, we can derive the **deduction theorem**, which was known to the ancient Greeks

For any sentences α and β, $\alpha \models \beta$ if and only if the sentence $(\alpha \Rightarrow \beta)$ is valid.

- The final concept we will need is **satisfiability**. A sentence is satisfiable if it is true in, or satisfied by, *some* model.
- For example, the knowledge base given earlier, (R1 ∧ R2 ∧ R3 ∧ R4 ∧ R5), is satisfiable because there are three models in which it is true.
- Satisfiability can be checked by enumerating the possible models until one is found that satisfies the sentence.

Inference and proofs

- **inference rules** that can be applied to derive a **proof**—a chain of conclusions that leads to the desired goal.
- The best-known rule is called **Modus Ponens** and is written $\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}$.
- The notation means that, whenever any sentences of the form $\alpha \Rightarrow \beta$ and α are given, then the sentence β can be inferred.
- Another useful inference rule is **And-Elimination**, which says that, from a conjunction, any of the conjuncts can be inferred:

$$\frac{\alpha \wedge \beta}{\alpha}$$

- By considering the possible truth values of α and β, one can show easily that Modus Ponens and And-Elimination are sound once and for all.
- These rules can then be used in any particular instances where they apply, generating sound inferences without the need for enumerating models.
- For example, the equivalence for biconditional elimination yields the two inference rules

$$\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)} \quad \text{and} \quad \frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}.$$

- Let us see how these inference rules and equivalences can be used in the wumpus world.
- We start with the knowledge base containing R1 through R5 and show how to prove ¬P1,2, that is, there is no pit in [1,2]. First, we apply biconditional elimination to R2 to obtain

$$R_6 : \quad (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}) .$$

Then we apply And-Elimination to R_6 to obtain

$$R_7 : \quad ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}) .$$

Logical equivalence for contrapositives gives

$$R_8 : \quad (\neg B_{1,1} \Rightarrow \neg(P_{1,2} \vee P_{2,1})) .$$

Now we can apply Modus Ponens with R_8 and the percept R_4 (i.e., $\neg B_{1,1}$), to obtain

$$R_9 : \quad \neg(P_{1,2} \vee P_{2,1}) .$$

Finally, we apply De Morgan's rule, giving the conclusion

$$R_{10} : \quad \neg P_{1,2} \wedge \neg P_{2,1} .$$

That is, neither [1,2] nor [2,1] contains a pit.

We found this proof by hand, but we can apply any of the search algorithms in Chapter 3 to find a sequence of steps that constitutes a proof. We just need to define a proof problem as follows:

- INITIAL STATE: the initial knowledge base.
- ACTIONS: the set of actions consists of all the inference rules applied to all the sentences that match the top half of the inference rule.
- RESULT: the result of an action is to add the sentence in the bottom half of the inference rule.
- GOAL: the goal is a state that contains the sentence we are trying to prove.

- A sentence expressed as a conjunction of clauses is said to be in **conjunctive normal form** or **CNF** . We now describe a procedure for converting to CNF.
- We illustrate the procedure by converting the sentence B1,1 ⇔ (P1,2 ∨ P2,1) into CNF. The steps are as follows:

1. **Eliminate $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.**

$$(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}) .$$

2. **Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$:**

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1}) .$$

3. CNF requires ¬ to appear only in literals, so we "move ¬ inwards" by repeated application of the following equivalences from Figure 7.11:

$$\neg(\neg\alpha) \equiv \alpha \quad \text{(double-negation elimination)}$$
$$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta) \quad \text{(De Morgan)}$$
$$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta) \quad \text{(De Morgan)}$$

In the example, we require just one application of the last rule:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1}) .$$

4. Now we have a sentence containing nested ∧ and ∨ operators applied to literals. We apply the distributivity law from Figure 7.11, distributing ∨ over ∧ wherever possible.

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1}) .$$

A resolution algorithm

- A resolution algorithm is shown. First, (KB ∧ ¬α) is converted into CNF. Then, the resolution rule is applied to the resulting clauses.
- Each pair that contains complementary literals is resolved to produce a new clause, which is added to the set if it is not already present.
- The process continues until one of two things happens:
  - there are no new clauses that can be added, in which case KB does not entail α; or,
  - two clauses resolve to yield the *empty* clause, in which case KB entails α.
- The empty clause—a disjunction of no disjuncts—is equivalent to False because a disjunction is true only if at least one of its disjuncts is true.

$$
\begin{aligned}
CNFSentence &\rightarrow Clause_1 \wedge \cdots \wedge Clause_n \\
Clause &\rightarrow Literal_1 \vee \cdots \vee Literal_m \\
Literal &\rightarrow Symbol \mid \neg Symbol \\
Symbol &\rightarrow P \mid Q \mid R \mid \ldots \\
HornClauseForm &\rightarrow DefiniteClauseForm \mid GoalClauseForm \\
DefiniteClauseForm &\rightarrow (Symbol_1 \wedge \cdots \wedge Symbol_l) \Rightarrow Symbol \\
GoalClauseForm &\rightarrow (Symbol_1 \wedge \cdots \wedge Symbol_l) \Rightarrow False
\end{aligned}
$$

Figure 7.14 A grammar for conjunctive normal form, Horn clauses, and definite clauses. A clause such as $A \wedge B \Rightarrow C$ is still a definite clause when it is written as $\neg A \vee \neg B \vee C$, but only the former is considered the canonical form for definite clauses. One more class is the k-CNF sentence, which is a CNF sentence where each clause has at most k literals.

```
function PL-RESOLUTION(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of KB ∧ ¬α
  new ← { }
  loop do
    for each pair of clauses C_i, C_j in clauses do
      resolvents ← PL-RESOLVE(C_i, C_j)
      if resolvents contains the empty clause then return true
      new ← new ∪ resolvents
    if new ⊆ clauses then return false
    clauses ← clauses ∪ new
```

Figure 7.12 A simple resolution algorithm for propositional logic. The function PL-RESOLVE returns the set of all possible clauses obtained by resolving its two inputs.

■ EFFECTIVE PROPOSITIONAL MODEL CHECKING

- **A complete backtracking algorithm**
- The first algorithm we consider is often called the **Davis–Putnam algorithm**, after the seminal paper by Martin Davis and Hilary Putnam (1960).
- it is essentially a recursive, depth-first enumeration of possible models.

• *Early termination*: The algorithm detects whether the sentence must be true or false, even with a partially completed model. A clause is true if *any* literal is true, even if the other literals do not yet have truth values; hence, the sentence as a whole could be judged true even before the model is complete.

For example, the sentence $(A \vee B) \wedge (A \vee C)$ is true if A is true, regardless of the values of B and C.

- a sentence is false if *any* clause is false, which occurs when each of its literals is false. Again, this can occur long before the model is complete.

- *Pure symbol heuristic*: A **pure symbol** is a symbol that always appears with the same "sign" in all clauses. For example, in the three clauses $(A \vee \neg B)$, $(\neg B \vee \neg C)$, and $(C \vee A)$, the symbol A is pure because only the positive literal appears, B is pure because only the negative literal appears, and C is impure.

Note that, in determining the purity of a symbol, the algorithm can ignore clauses that are already known to be true in the model constructed so far.

For example, if the model contains B =false, then the clause $(\neg B \vee \neg C)$ is already true, and in the remaining clauses C appears only as a positive literal; therefore C becomes pure.

- *Unit clause heuristic*: A **unit clause** was defined earlier as a clause with just one literal. In the context of DPLL, it also means clauses in which all literals but one are already assigned false by the model.

For example, if the model contains B =true, then (¬B ∨ ¬C) simplifies to ¬C, which is a unit clause. Obviously, for this clause to be true, C must be set to false. The unit clause heuristic assigns all such symbols before branching on the remainder.

```
function PL-FC-ENTAILS?(KB, q) returns true or false
  inputs: KB, the knowledge base, a set of propositional definite clauses
          q, the query, a proposition symbol
  count ← a table, where count[c] is the number of symbols in c's premise
  inferred ← a table, where inferred[s] is initially false for all symbols
  agenda ← a queue of symbols, initially symbols known to be true in KB

  while agenda is not empty do
    p ← POP(agenda)
    if p = q then return true
    if inferred[p] = false then
      inferred[p] ← true
      for each clause c in KB where p is in c.PREMISE do
        decrement count[c]
        if count[c] = 0 then add c.CONCLUSION to agenda
  return false
```

Figure 7.15 The forward-chaining algorithm for propositional logic. The *agenda* keeps track of symbols known to be true but not yet "processed." The *count* table keeps track of how many premises of each implication are as yet unknown. Whenever a new symbol p from the agenda is processed, the count is reduced by one for each implication in whose premise p appears (easily identified in constant time with appropriate indexing.) If a count reaches zero, all the premises of the implication are known, so its conclusion can be added to the agenda. Finally, we need to keep track of which symbols have been processed; a symbol that is already in the set of inferred symbols need not be added to the agenda again. This avoids redundant work and prevents loops caused by implications such as $P \Rightarrow Q$ and $Q \Rightarrow P$.

Local search algorithms

- This choose the right evaluation function. Because the goal is to find an assignment that satisfies every clause, an evaluation function that counts the number of unsatisfied clauses will do the job. In fact,
- All these algorithms take steps in the space of complete assignments, flipping the truth value of one symbol at a time.
- The space usually contains many local minima, to escape from which various forms of randomness are required.
- One of the simplest and most effective algorithms to emerge from all this work is called WALKSAT.
- On every iteration, the algorithm picks an unsatisfied clause and picks a symbol in the clause to flip.
- It chooses randomly between two ways to pick which symbol to flip: (1) a “min-conflicts” step that minimizes the number of unsatisfied clauses in the new state and (2) a “random walk” step that picks the symbol randomly.

- When WALKSAT returns a model, the input sentence is indeed satisfiable but when it returns failure, there are two possible causes: either the sentence is unsatisfiable or we need to give the algorithm more time.
- If we set max flips =∞ and p > 0, WALKSAT will eventually return a model (if one exists), because the random-walk steps will eventually hit upon the solution. Alas, if max flips is infinity and the sentence is unsatisfiable, then the algorithm never terminates!

```
function WALKSAT(clauses, p, max_flips) returns a satisfying model or failure
  inputs: clauses, a set of clauses in propositional logic
          p, the probability of choosing to do a "random walk" move, typically around 0.5
          max_flips, number of flips allowed before giving up

  model ← a random assignment of true/false to the symbols in clauses
  for i = 1 to max_flips do
      if model satisfies clauses then return model
      clause ← a randomly selected clause from clauses that is false in model
      with probability p flip the value in model of a randomly selected symbol from clause
      else flip whichever symbol in clause maximizes the number of satisfied clauses
  return failure
```

Figure 7.18 The WALKSAT algorithm for checking satisfiability by randomly flipping the values of variables. Many versions of the algorithm exist.

AGENTS BASED ON PROPOSITIONAL LOGIC

- we show how the agent can use logical inference to construct plans that are guaranteed to achieve its goals.
  - **The current state of the world**
- a logical agent operates by deducing what to do from a knowledge base of sentences about the world.
- The knowledge base is composed of axioms—general knowledge about how the world works—and percept sentences obtained from the agent's experience in a particular world.
- The agent knows that the starting square contains no pit (¬P1,1) and no wumpus (¬W1,1).
- Furthermore, for each square, it knows that the square is breezy if and only if a neighboring square has a pit; and a square is smelly if and only if a neighboring square has a wumpus.

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$$
$$S_{1,1} \Leftrightarrow (W_{1,2} \vee W_{2,1})$$
$$\cdots$$

- The agent also knows that there is exactly one wumpus. This is expressed in two parts.
- First, we have to say that there is *at least one* wumpus:

$$W_{1,1} \vee W_{1,2} \vee \cdots \vee W_{4,3} \vee W_{4,4} .$$

- Then, we have to say that there is *at most one* wumpus. For each pair of locations, we add a sentence saying that at least one of them must be wumpus-free:

$$\begin{aligned} &\neg W_{1,1} \vee \neg W_{1,2} \\ &\neg W_{1,1} \vee \neg W_{1,3} \\ &\cdots \\ &\neg W_{4,3} \vee \neg W_{4,4} \ . \end{aligned}$$

- there is currently a stench, one might suppose that a proposition Stench should be added to the knowledge base.
- This is not quite right, however: if there was no stench at the previous time step, then ¬Stench would already be asserted, and the new assertion would simply result in a contradiction.
- The problem is solved when we realize that a percept asserts something *only about the current time*. Thus, if the time step is 4, then we add Stench4 to the knowledge base,
- The same goes for the breeze, bump, glitter, and scream percepts. The idea of associating propositions with time steps extends to any aspect of the world that changes over time.
- For example, the initial knowledge base includes $L^0_{1,1}$ the agent is in square [1, 1] at time 0. as well as

$$FacingEast^0, \ HaveArrow^0, \text{ and } \ WumpusAlive^0.$$

- We use the word **fluent** to refer an aspect of the world that changes. Symbols associated with permanent aspects of the world do not need a time superscript and are sometimes called **atemporal variables.**

- For any time step t and any square [x, y], we assert

$$L^t_{x,y} \Rightarrow (Breeze^t \Leftrightarrow B_{x,y})$$
$$L^t_{x,y} \Rightarrow (Stench^t \Leftrightarrow S_{x,y}) .$$