7.3

a. There is a `pl true` in the Python code, and a version of `ask` in the Lisp code that serves the same purpose. The Java code did not have this function as of May 2003, but it should be added soon.)

b. The sentences $\mathbf{True}$, $P \vee \neg P$, and $P \wedge \neg P$ can all be determined to be true or false in a partial model that does not specify the truth value for P.

c. It is possible to create two sentences, each with k variables that are not instantiated in the partial model, such that one of them is true for all 2^k possible values of the variables, while the other sentence is false for one of the 2^k values. This shows that in general one must consider all 2^k possibilities. Enumerating them takes exponential time.

d. The python implementation of `pl true` returns true if any disjunct of a disjunction is true, and false if any conjunct of a conjunction is false. It will do this even if other disjuncts/conjuncts contains uninstantiated variables. Thus, in the partial model where P is true, $P \vee Q$ returns true, and $\neg P \wedge Q$ returns false. But the truth values of $Q \vee \neg Q$, $Q \vee \mathbf{True}$, and $Q \wedge \neg Q$ are not detected.

e. Our version of `tt entails` already uses this modified `pl true`. It would be slower if it did not.

7.7 We use the truth table code in Lisp in the directory `logic/prop.lisp` to show eac sentence is valid. We substitute `P, Q, R` for α, β, γ because of the lack of Greek letters i ASCII. To save space in this manual, we only show the first four truth tables:

```
> (truth-table "P ^ Q <=> Q ^ P")
----------------------------------------
 P  Q  P ^ Q  Q ^ P  (P ^ Q) <=> (Q ^ P)
----------------------------------------
 F  F    F      F            \(true\)
 T  F    F      F             T
 F  T    F      F             T
 T  T    T      T             T
----------------------------------------
NIL

> (truth-table "P | Q <=> Q | P")
----------------------------------------
 P  Q  P | Q  Q | P  (P | Q) <=> (Q | P)
----------------------------------------
 F  F    F      F             T
 T  F    T      T             T
 F  T    T      T             T
 T  T    T      T             T
----------------------------------------
NIL

> (truth-table "P ^ (Q ^ R) <=> (P ^ Q) ^ R")
------------------------------------------------------------------------
 P  Q  R  Q ^ R  P ^ (Q ^ R)  P ^ Q ^ R  (P ^ (Q ^ R)) <=> (P ^ Q ^ R)
------------------------------------------------------------------------
 F  F  F    F         F           F                    T
 T  F  F    F         F           F                    T
 F  T  F    F         F           F                    T
```

```
 T  T  F    F         F           F                   T
 F  F  T    F         F           F                   T
 T  F  T    F         F           F                   T
 F  T  T    T         F           F                   T
 T  T  T    T         T           T                   T
-------------------------------------------------------------------------
NIL

> (truth-table "P | (Q | R) <=> (P | Q) | R")
-------------------------------------------------------------------------
 P  Q  R  Q | R  P | (Q | R)  P | Q | R  (P | (Q | R)) <=> (P | Q | R)
-------------------------------------------------------------------------
 F  F  F    F         F           F                   T
 T  F  F    F         T           T                   T
 F  T  F    T         T           T                   T
 T  T  F    T         T           T                   T
 F  F  T    T         T           T                   T
 T  F  T    T         T           T                   T
 F  T  T    T         T           T                   T
 T  T  T    T         T           T                   T
-------------------------------------------------------------------------
NIL
```

For the remaining sentences, we just show that they are valid according to the `validi` function:

```
> (validity "~~P <=> P")
VALID
> (validity "P => Q <=> ~Q => ~P")
VALID
> (validity "P => Q <=> ~P | Q")
VALID
> (validity "(P <=> Q) <=> (P => Q) ^ (Q => P)")
VALID
> (validity "~(P ^ Q) <=> ~P | ~Q")
VALID
> (validity "~(P | Q) <=> ~P ^ ~Q")
VALID
> (validity "P ^ (Q | R) <=> (P ^ Q) | (P ^ R)")
VALID
> (validity "P | (Q ^ R) <=> (P | Q) ^ (P | R)")
VALID
```

7.8 We use the `validity` function from `logic/prop.lisp` to determine the validity of each sentence:

```
> (validity "Smoke => Smoke")
VALID
> (validity "Smoke => Fire")
SATISFIABLE
> (validity "(Smoke => Fire) => (~Smoke => ~Fire)")
SATISFIABLE
> (validity "Smoke | Fire | ~Fire")
VALID
```

```
> (validity "((Smoke ^ Heat) => Fire) <==> ((Smoke => Fire) | (Heat => Fire))")
VALID
> (validity "(Smoke => Fire) => ((Smoke ^ Heat) => Fire)")
VALID
> (validity "Big | Dumb | (Big => Dumb)")
VALID
> (validity "(Big ^ Dumb) | ~Dumb")
SATISFIABLE
```

Many people are fooled by (e) and (g) because they think of implication as being causation, or something close to it. Thus, in (e), they feel that it is the combination of Smoke and Heat that leads to Fire, and thus there is no reason why one or the other alone should lead to fire. Similarly, in (g), they feel that there is no necessary causal relation between Big and Dumb, so the sentence should be satisfiable, but not valid. However, this reasoning is incorrect, because implication is *not* causation—implication is just a kind of disjunction (in the sense that $P \Rightarrow Q$ is the same as $\neg P \vee Q$). So $Big \vee Dumb \vee (Big \Rightarrow Dumb)$ is equivalent to $Big \vee Dumb \vee \neg Big \vee Dumb$, which is equivalent to $Big \vee \neg Big \vee Dumb$, which is true whether Big is true or false, and is therefore valid.

7.10 Each possible world can be written as a conjunction of symbols, e.g. $(A \wedge C \wedge E)$. Asserting that a possible world is not the case can be written by negating that, e.g. $\neg(A \wedge C \wedge E)$, which can be rewritten as $(\neg A \vee \neg C \vee \neg E)$. This is the form of a clause; a conjunction of these clauses is a CNF sentence, and can list all the possible worlds for the sentence.

7.14 Optimal behavior means achieving an expected utility that is as good as any other agent program. The PL-WUMPUS-AGENT is clearly non-optimal when it chooses a random move (and may be non-optimal in other branches of its logic). One example: if the agent detects a breeze in [1,2] and [2,1], there may be pits in [1,3], [2,2], and [3,1], so there is no safe way to proceed. A random move is suboptimal because some squares are *more likely* to be safe than others given the available information. In the example just given, the calculation in Chapter 13 shows that [2,2] is more than twice as dangerous as [1,3] and [3,1]. These refinements cannot be captured in a logical knowledge base.