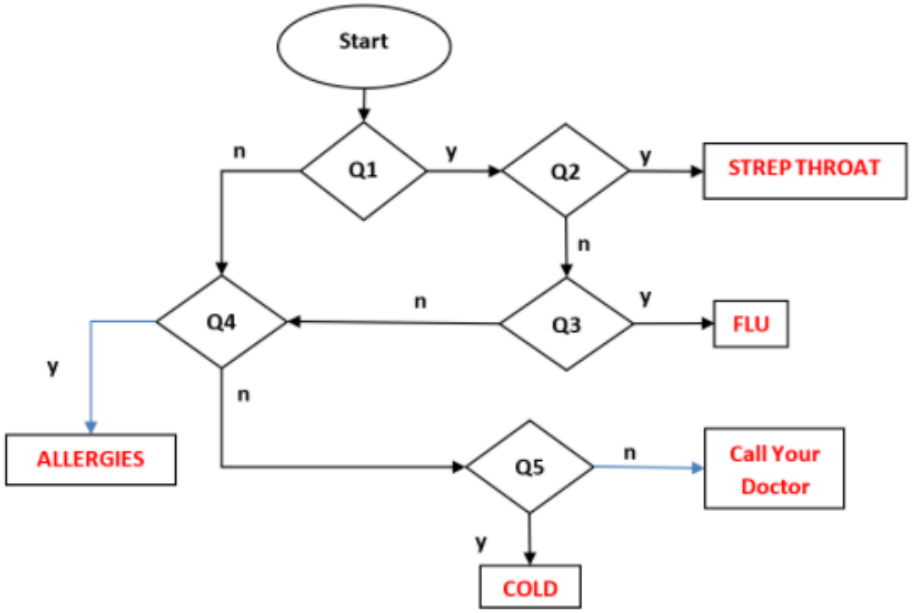
RULE-BASED EXPERT SYSTEMS & Prolog

Dr. Emad Natsheh

In-Class Exercise

- **Using your favorite programming language, create a system to interactively ask users yes/no questions and diagnose Cold and Flu ?**

Flowchart Cold and Flu



The questions are:

Q1. Do you have a fever?

Q2. Do you have a sore throat and headache-without nasal drainage?

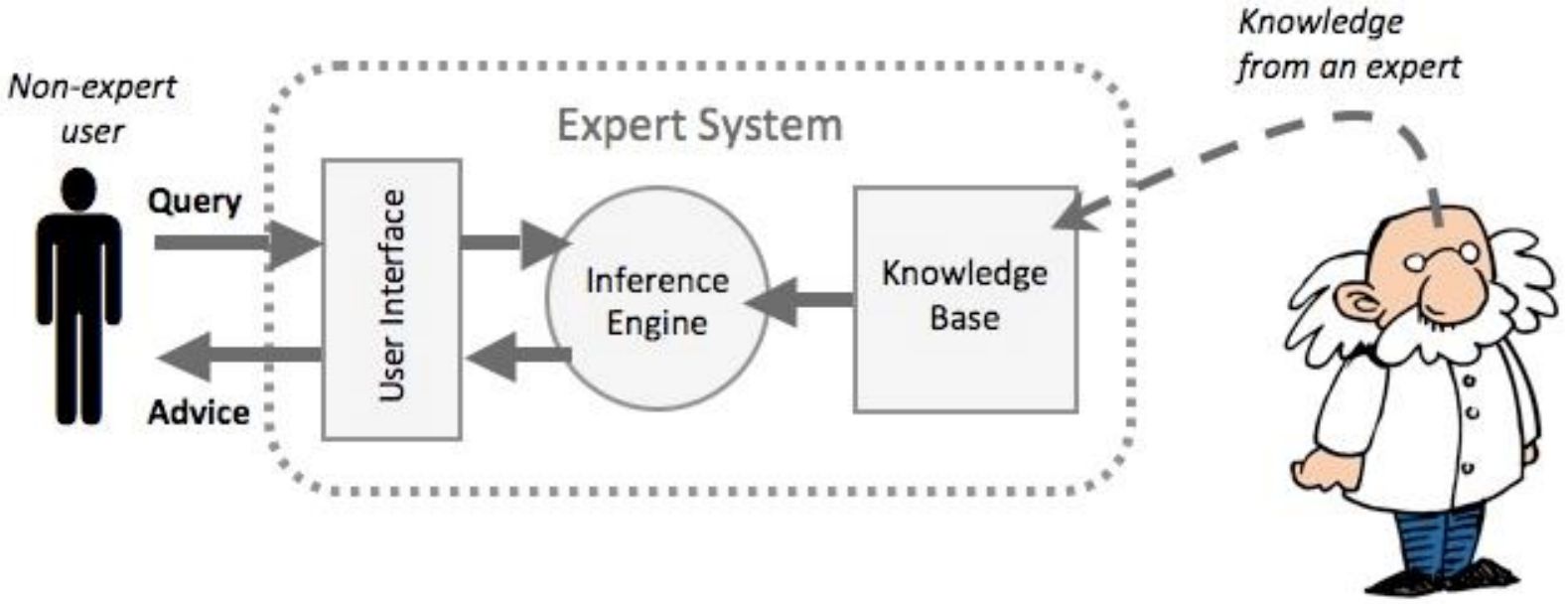
Q3. Did your symptoms start suddenly, and do you have a combination of symptoms including muscle aches, chills, a sore throat, runny nose or cough?

Q4. Do you have a runny and/or itchy nose, sneezing, and itchy eyes?

Q5. Do you have sneezing, a sore throat, headache, congestion and a runny nose?

Rule base (RB) Expert Systems

An expert system is a computer program that is designed to hold the accumulated knowledge of one or more domain experts



Cont....

- An expert system performs at a human expert level in a narrow and specialized domain
- Use **symbolic reasoning** to solve problems
  - Symbols represent facts and rules (i.e. knowledge)
- Expert systems provide explanation facilities to display reasoning
- Expert systems make mistakes
- Common programming language support
  - **Prolog**
  - CLIPS

Why use Expert Systems?

- Experts are not always available.
- An expert system can be used anywhere, any time.
- Human experts are not 100% reliable or consistent
- Experts may not be good at explaining decisions
- Cost effective

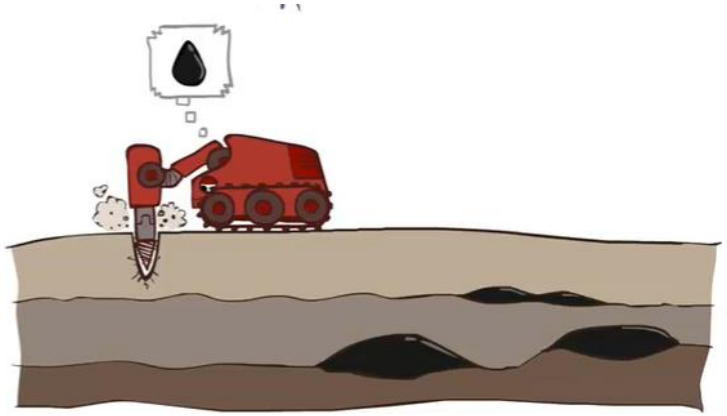
By Comparison....

Traditional programs	Expert systems
Process data and **use algorithms** to solve general numerical problems	Process knowledge expressed in the form of rules and **use symbolic reasoning** to solve problems in a narrow domain
Do not separate knowledge from the control structures	Provide a **clear separation of knowledge** and knowledge processing (reasoning)
Do not explain how particular results are obtained and why input data is needed	**explain how a particular conclusion was reached**
Work only on problems where data **is complete and exact**	can deal with **incomplete and uncertain information**
Enhance the quality of problem solving **by changing the program code**	Enhance the quality of problem solving **by adding new rules** or adjusting

By Comparison....

Rule base expert system	Fuzzy expert system
Expert System deal with facts and rules that are chained together using an automated reasoning process. Typically, the system will use forward or backward chaining algorithms depending upon whether you're trying to discover new facts or prove a statement, respectively	FIS use fuzzy sets and membership functions to describe various phenomenon. Two sets are connected through fuzzy rules. Typically, the inputs and outputs are numeric scores that correspond to input values or average membership values of the fuzzy sets, respectively.

Applications of Expert Systems

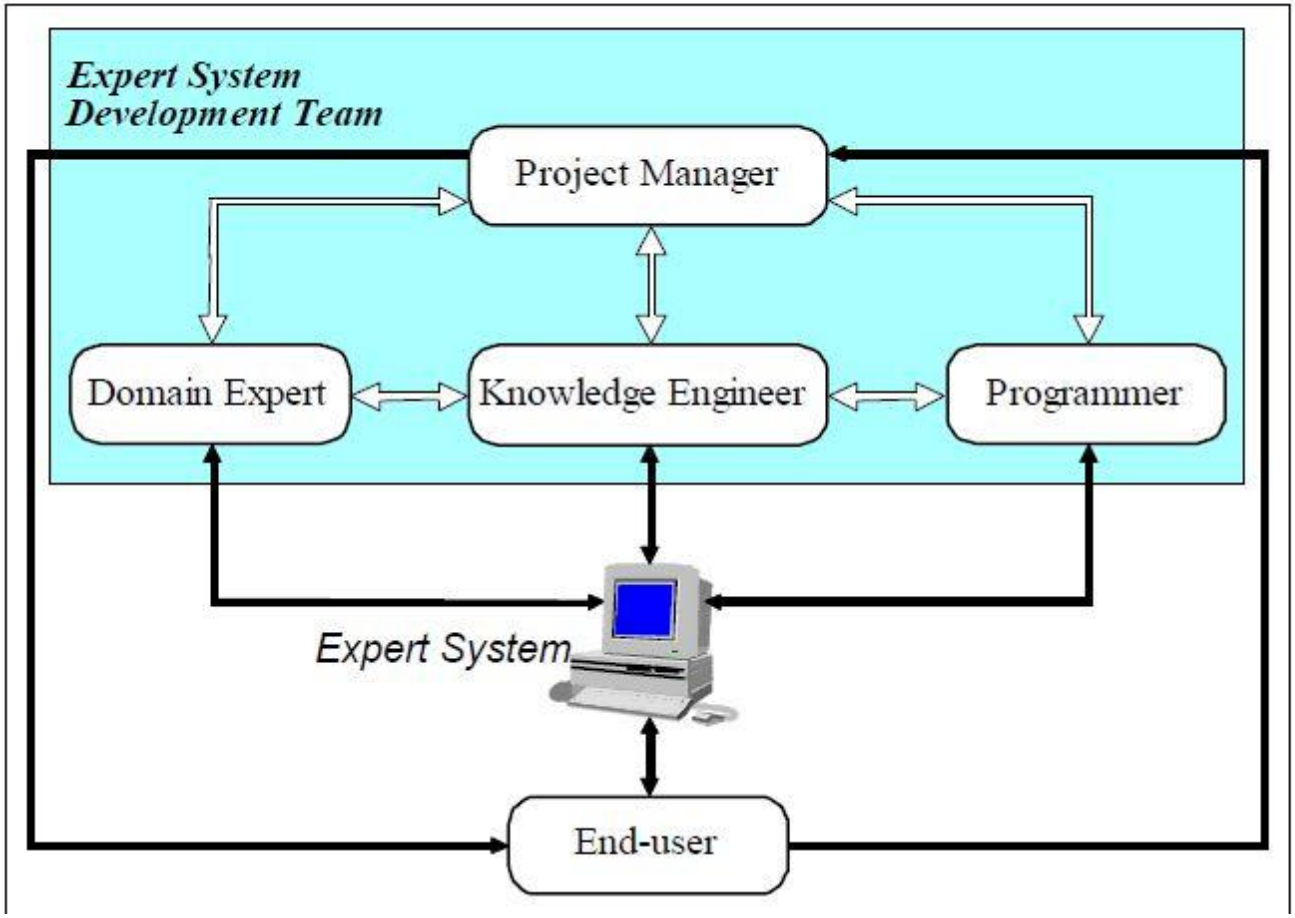


PROSPECTOR:
Used by geologists to identify sites for drilling or mining

PUFF:
Medical system for diagnosis of respiratory conditions

isabel

Development Team

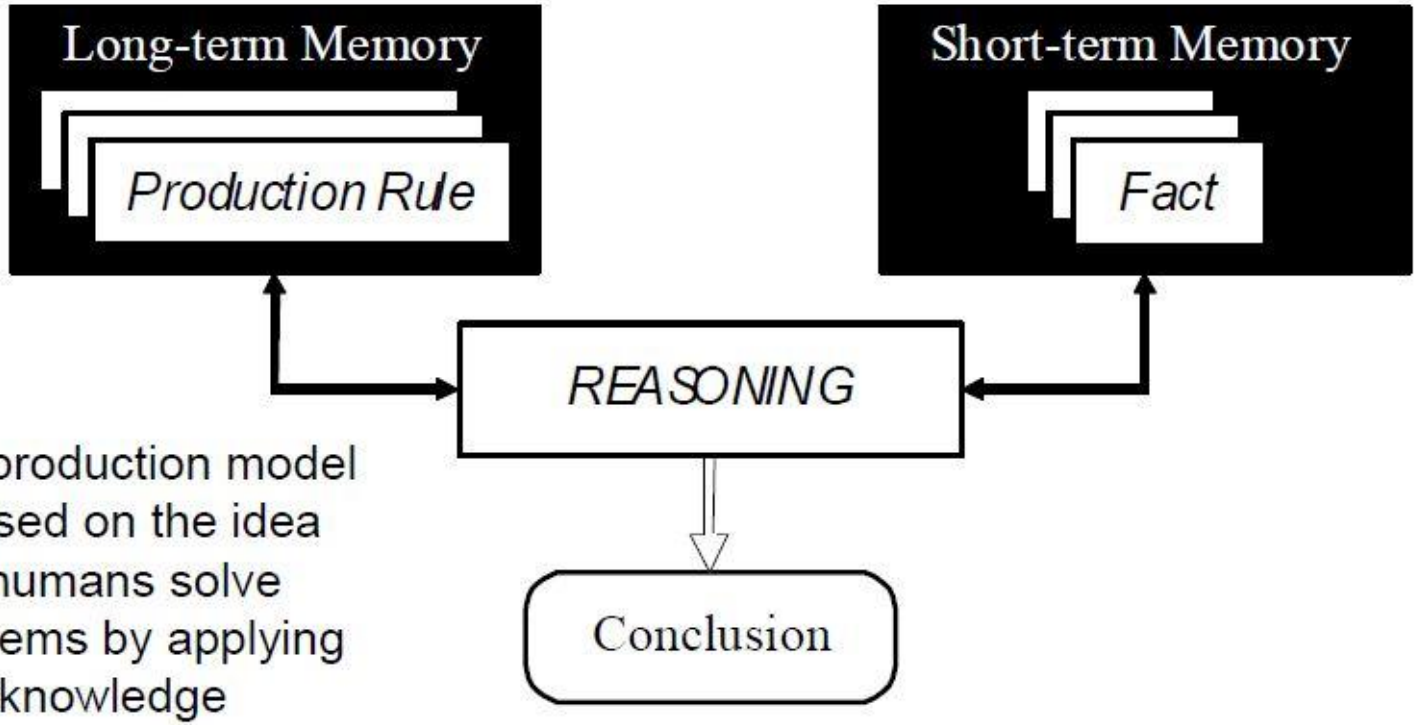


Development Team

- **Domain expert** is a knowledgeable person capable of solving problems in a specific area or domain. This person has the greatest expertise in a given domain. This expertise is to be captured in the expert system. The domain expert is the most important player in the expert system development team
- **Knowledge engineer** is someone who is capable of designing, building and testing an expert system. He or she interviews the domain expert to find out how a particular problem is solved. The knowledge engineer then chooses some development software or an expert system shell, or looks at programming languages for encoding the knowledge (and sometimes encodes it himself).

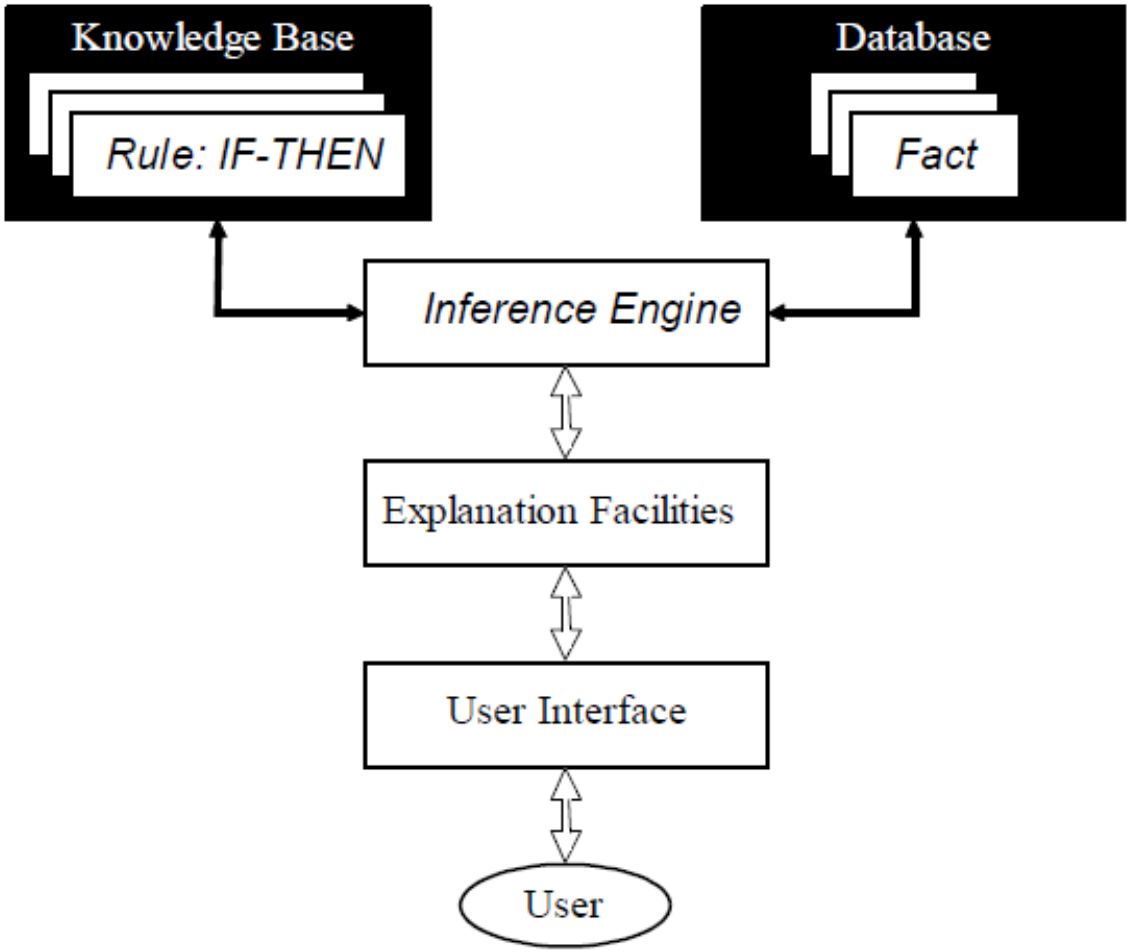
Production System Model

- Newell and Simon, CMU (1970s)

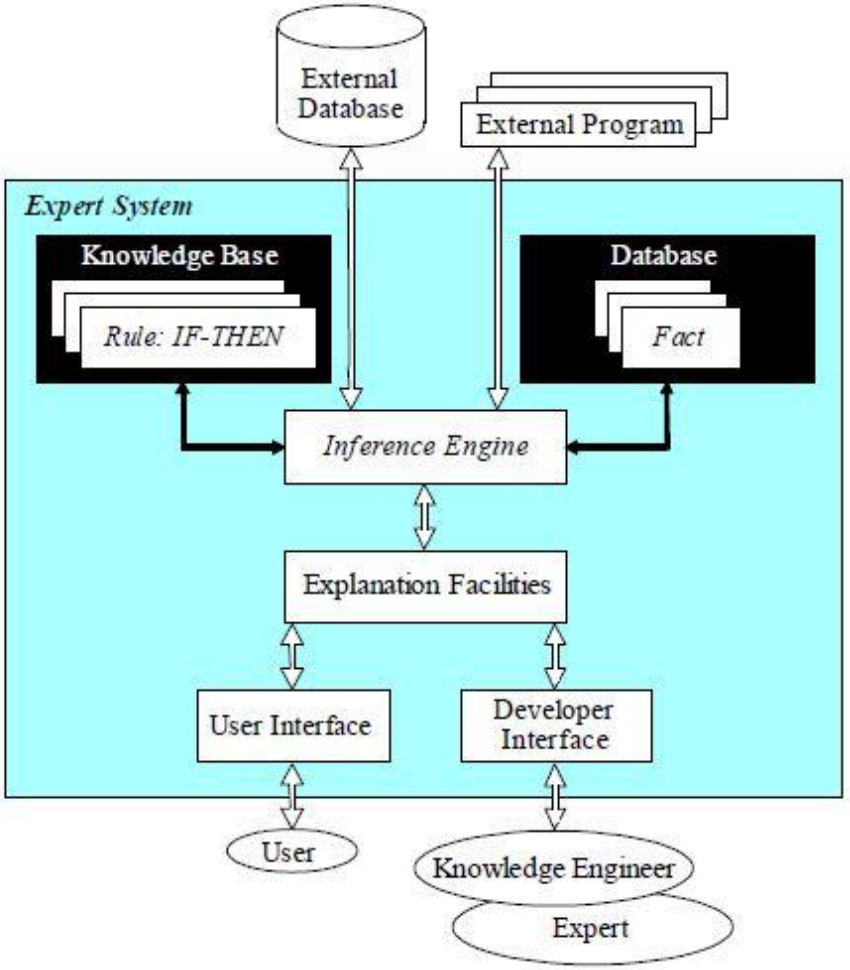


The production model is based on the idea that humans solve problems by applying their knowledge

Basic Structure of an RB Expert System



Complete Structure



Inference Engine

Inference engine algorithm:

- Inference engine compares each rule with facts it already “knows” about, matching the antecedent (IF condition)
- When the antecedent matches one or more known facts, the rule fires and its consequent (THEN) is executed

Inference Engine

- The matching of the rule IF parts to the facts produces inference chains.
- The inference chain indicates how an expert system applies the rules to reach a conclusion
  - E.g: Suppose the database initially includes facts A, B, C, D and E, and the knowledge base contains only three rules:
    - Rule 1: IF Y is true AND D is true then Z is true
    - Rule 2: IF X is true AND B is true AND E is true THEN Y is true
    - Rule 3: IF A is true THEN X is true

Inference Engine

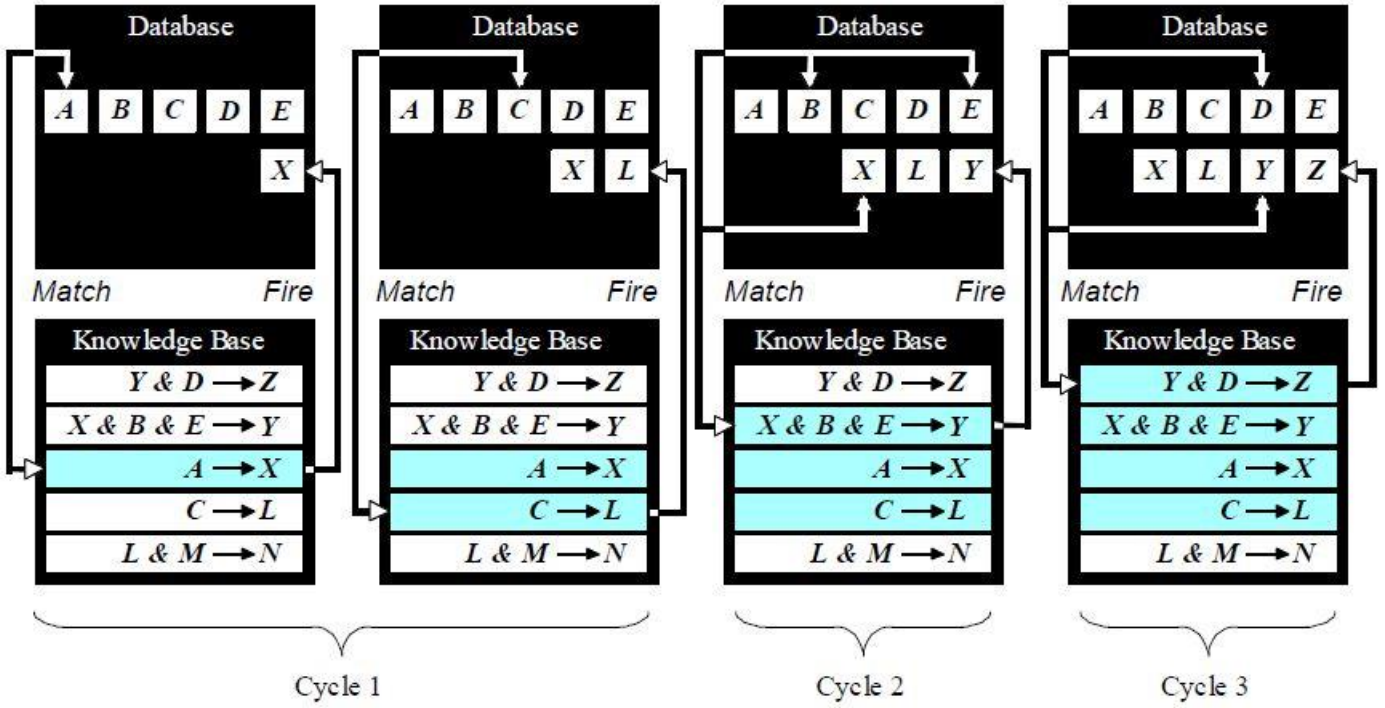
- The inference engine must decide when the rules must be fired.
- There are two principal ways in which rules are executed. One is called **forward chaining** and the other **backward chaining**

Forward Chaining

Forward chaining applies rules to known data to achieve a desired goal

- Also called **data-driven reasoning**
- At each iteration, topmost rule is executed
- When fired, a rule adds a new fact to the database
- This match-fire cycle **stops**; when the goal has been found or when no other rules can be fired

Example


Database
A B C D E
X
Match
Fire
Knowledge Base
Y & D → Z
X & B & E → Y
A → X
C → L
L & M → N
Database
A B C D E
X L
Match
Fire
Knowledge Base
Y & D → Z
X & B & E → Y
A → X
C → L
L & M → N
Database
A B C D E
X L Y
Match
Fire
Knowledge Base
Y & D → Z
X & B & E → Y
A → X
C → L
L & M → N
Database
A B C D E
X L Y Z
Match
Fire
Knowledge Base
Y & D → Z
X & B & E → Y
A → X
C → L
L & M → N
Cycle 1
Cycle 2
Cycle 3

Example

- Use *forward chaining* to prove the following:
  - Facts: A B C D E
  - Prove: Z
  - Rules:

$$
\begin{aligned}
A &\rightarrow X \\
Q &\rightarrow T \\
A &\rightarrow Y \\
A \ \& \ D &\rightarrow Q \\
Q &\rightarrow R \\
R &\rightarrow S \\
N &\rightarrow L \\
T &\rightarrow Z
\end{aligned}
$$

Example

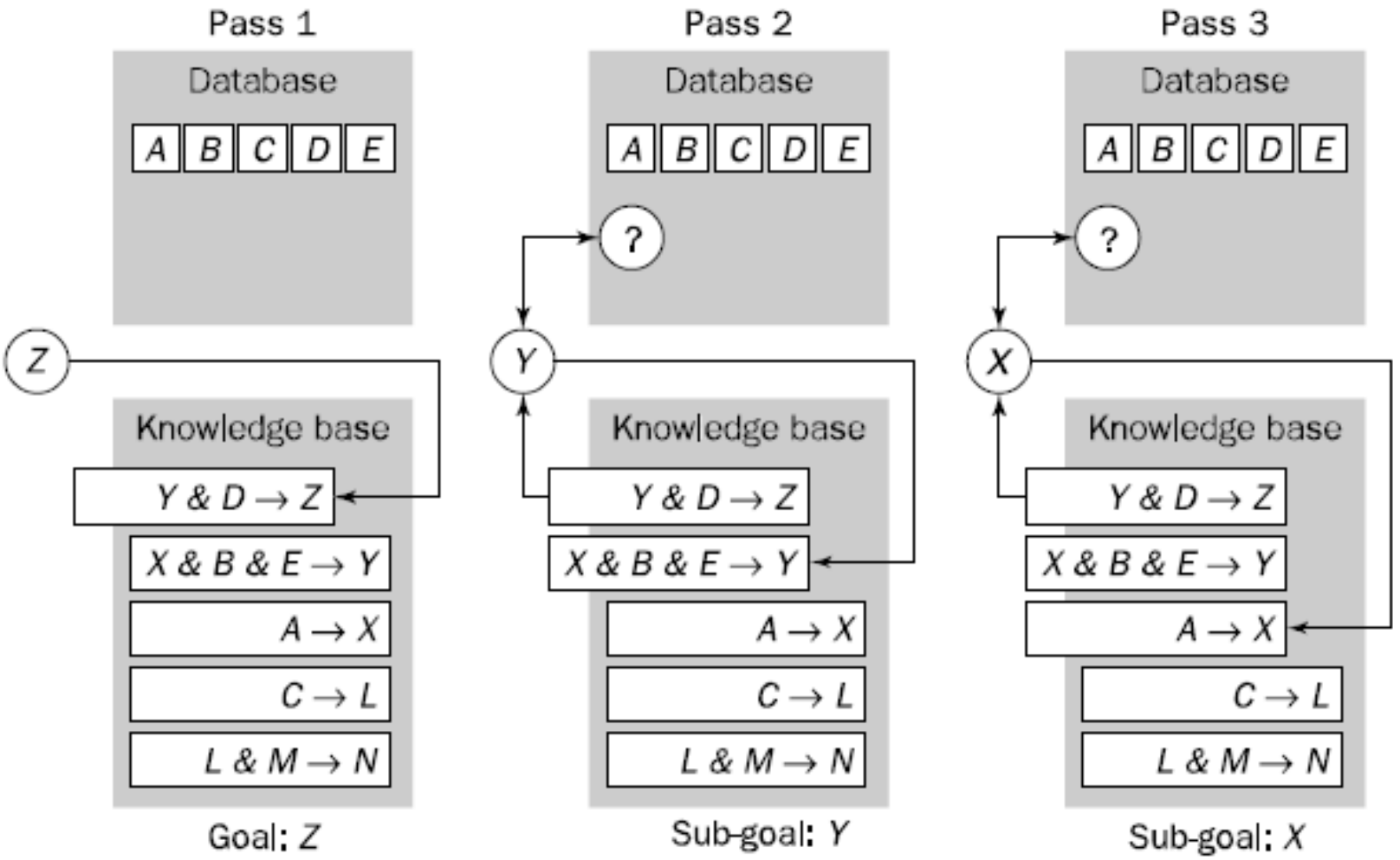
- Use *forward chaining* to prove the following:
  - Facts: A B C D E
  - Prove: L
  - Rules:

$$
\begin{aligned}
A &\rightarrow X \\
Q &\rightarrow T \\
A &\rightarrow Y \\
A \;\&\; D &\rightarrow Q \\
Q &\rightarrow R \\
R &\rightarrow S \\
N &\rightarrow L \\
T &\rightarrow Z
\end{aligned}
$$

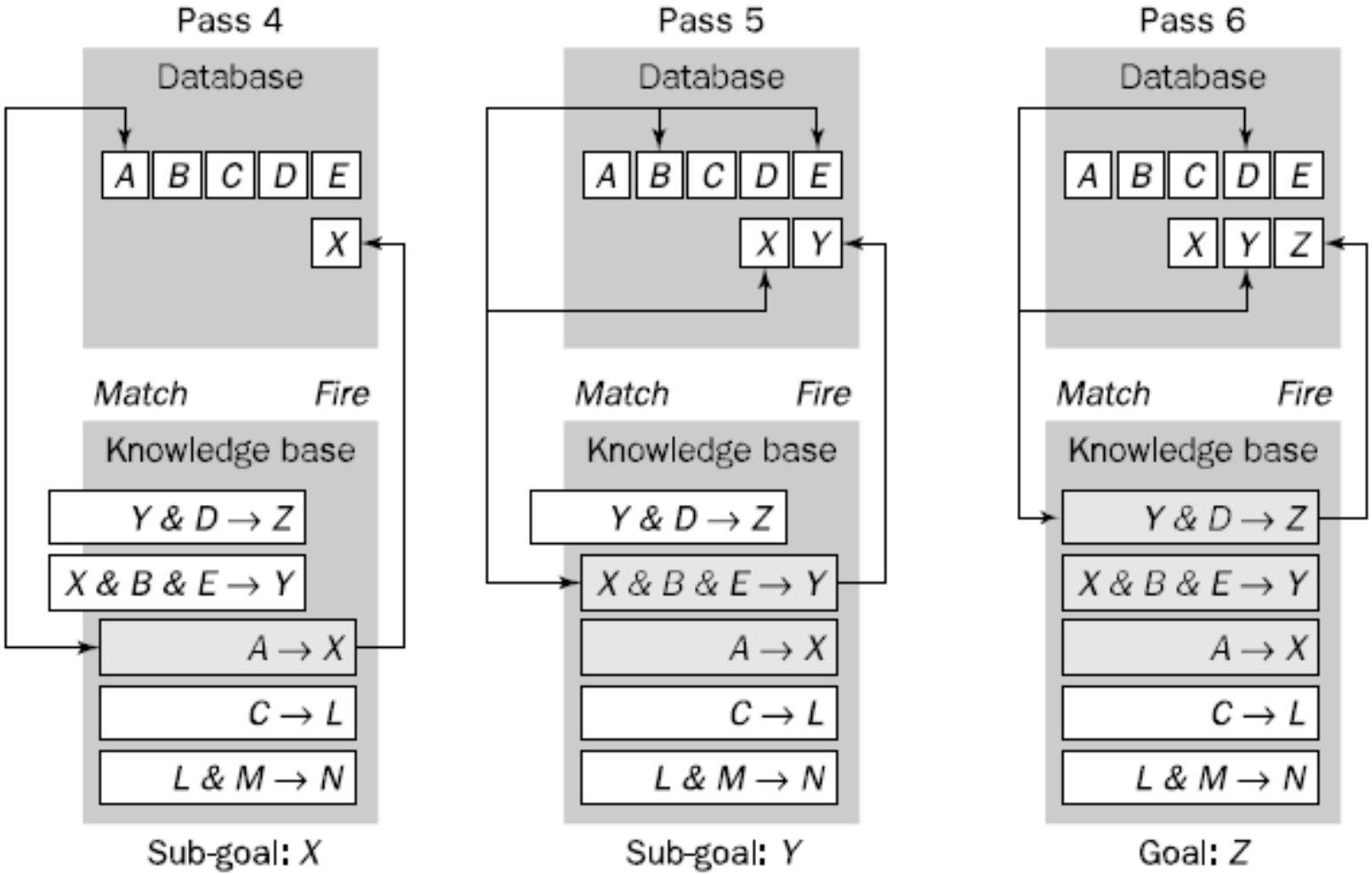
Backward Chaining

- Backward chaining **starts with a desired goal**, then searches for evidence to prove the goal
  - Desired goal often called the hypothetical solution
  - Backward chaining also **called goal-driven reasoning**

Example


Pass 1
Database
A B C D E
Z
Knowledge base
Y & D → Z
X & B & E → Y
A → X
C → L
L & M → N
Goal: Z
Pass 2
Database
A B C D E
?
Y
Knowledge base
Y & D → Z
X & B & E → Y
A → X
C → L
L & M → N
Sub-goal: Y
Pass 3
Database
A B C D E
?
X
Knowledge base
Y & D → Z
X & B & E → Y
A → X
C → L
L & M → N
Sub-goal: X

Example- Cont



Example

- Use *backward chaining* to prove the following:
  - Facts: A B C D E
  - Prove: Z
  - Rules:

		A	→	X
		Q	→	T
		A	→	Y
A	&	D	→	Q
		Q	→	R
		R	→	S
		N	→	L
		T	→	Z

Example

- Use *backward chaining* to prove the following:
  - Facts: A B C D E
  - Prove: L
  - Rules:

$$
\begin{aligned}
A &\rightarrow X \\
Q &\rightarrow T \\
A &\rightarrow Y \\
A \ \& \ D &\rightarrow Q \\
Q &\rightarrow R \\
R &\rightarrow S \\
N &\rightarrow L \\
T &\rightarrow Z
\end{aligned}
$$

Forward and Backward chaining?

The answer is to study how a domain expert solves a problem.

- If an expert first needs to gather some information and then tries to infer from it whatever can be inferred, choose the forward chaining inference engine.
- However, if your expert begins with a hypothetical solution and then attempts to find facts to prove it, choose the backward chaining inference engine.

Conflict Resolution

Rules with identical antecedents (IF conditions) can cause conflicts via their consequents (THEN clauses)

Let us considered three simple rules for crossing a road:

- Rule 1: IF the ‘traffic light’ is green THEN the action is go
- Rule 2: IF the ‘traffic light’ is red THEN the action is stop
- Rule 3: IF the ‘traffic light’ is red THEN the action is go

We have two rules, Rule 2 and Rule 3, with the same IF part. Thus, both of them can be set to fire when the condition part is satisfied. These rules represent a conflict set. The inference engine must determine which rule to fire from such a set. A method for choosing a rule to fire when more than one rule can be fired in a given cycle is called **conflict resolution.**

Methods used for conflict resolution

Conflict resolution provides a specific method for choosing which rule to fire

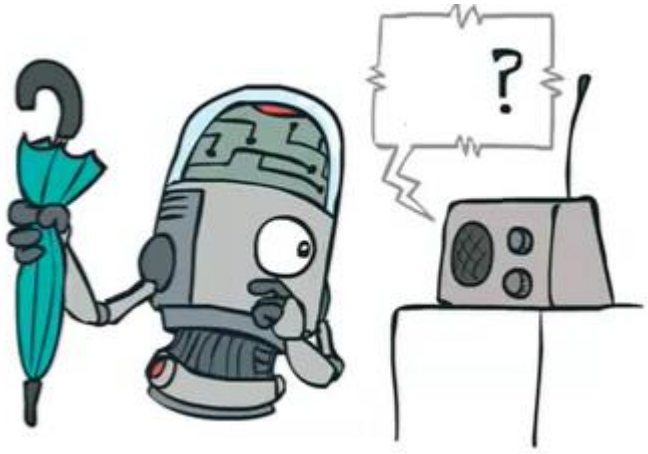
1. Fire the rule with the **highest priority**. In simple applications, the priority can be established by placing the rules in an appropriate order in the knowledge base. Usually, this strategy works well for expert systems with around 100 rules.

Methods used for conflict resolution

2. Fire the **most specific rule**. This method is also known as the **longest matching strategy**. It is based on the assumption that a specific rule processes more information than a general one.

Rule 1:
IF the season is autumn
AND the sky is cloudy
AND the forecast is rain
THEN the advice is 'stay home'

Rule 2:
IF the season is autumn
THEN the advice is 'take an umbrella'



Methods used for conflict resolution

3. Fire the rule that uses the data **most recently entered in the database**. This method relies on time tags attached to each fact in the database. In the conflict set, the expert system first fires the rule whose antecedent uses the data most recently added to the database.

Rule 1:
IF the forecast is rain [08:16 PM 11/25/96]
THEN the advice is 'take an umbrella'

Rule 2:
IF the weather is wet [10:18 AM 11/26/96]
THEN the advice is 'stay home'

Dealing with incomplete and uncertain knowledge

- Most rule-based expert systems are capable of representing and reasoning with incomplete and uncertain knowledge. For example, the rule

```
IF      season is autumn
AND     sky is 'cloudy'
AND     wind is low
THEN    forecast is clear      { cf 0.1 };
        forecast is drizzle    { cf 1.0 };
        forecast is rain       { cf 0.9 }
```

- The rule represents the uncertainty by numbers called **certainty factors** {cf 0.1}. The expert system uses certainty factors to establish the level of belief that the rule's conclusion is true

Certainty Factors

- **Certainty factor (cf) is a number to measure the expert's belief.**
- The maximum value of the certainty factor was +1.0 (definitely true) and the minimum -1.0 (definitely false).
- **A positive value** represented a **degree of belief** and a **negative a degree of disbelief**. For example, if the expert stated that some evidence was almost certainly true, a cf value of 0.8 would be assigned to this evidence.

Term	Certainty factor
Definitely not	−1.0
Almost certainly not	−0.8
Probably not	−0.6
Maybe not	−0.4
Unknown	−0.2 to +0.2
Maybe	+0.4
Probably	+0.6
Almost certainly	+0.8
Definitely	+1.0

Example 1

$$cf(H,E) = cf(E) * cf$$

- For example

IF the sky is clear

THEN the forecast is sunny {cf 0.8}

and the current certainty factor of sky is clear is 0.5, then

cf(H,E)=cf(E)*cf=0.5*0.8 = 0.4

- This result, according to previous Table, would read as

'It may be sunny'

Example 2

$$cf(H,E1 \cap E2 \ldots \cap En) = min[cf(E1),\ldots cf(En)] * cf$$

- For example

IF sky is clear
AND the forecast is sunny
THEN the action is 'wear sunglasses' {cf 0.8}

and the certainty of sky is clear is 0.9 and the certainty of the forecast is sunny is 0.7, then

cf(H,E1∩E2) =min[cf(E1),cf(E2)] * cf = 0.7*0.8=0.56

- This result, according to previous Table, would read as

'Probably it would be a good idea to wear sunglasses today'

Example 2

$$cf(H,E1 \cup E2 \ldots \cup En) = max[cf(E1),\ldots cf(En)] * cf$$

- For example

  IF sky is overcast
  OR the forecast is rain
  THEN the action is 'take an umbrella' {cf 0.9}

and the certainty of sky is overcast is 0.6 and the certainty of the forecast is rain is 0.8, then

cf(H,E1UE2) =max[cf(E1),cf(E2)] * cf = 0.8*0.9=0.72

- This result, according to previous Table, would read as

'Almost certainly an umbrella should be taken today'

Example 3 (More rules can affect the same hypothesis)

Rule 1: IF A is X
THEN C is Z {cf 0.8}

Rule 2: IF B is Y
THEN C is Z {cf 0.6}

$$cf(cf_1, cf_2) = \begin{cases} cf_1 + cf_2 \times (1 - cf_1) & \text{if } cf_1 > 0 \text{ and } cf_2 > 0 \\ \dfrac{cf_1 + cf_2}{1 - min\,[|cf_1|, |cf_2|]} & \text{if } cf_1 < 0 \text{ or } cf_2 < 0 \\ cf_1 + cf_2 \times (1 + cf_1) & \text{if } cf_1 < 0 \text{ and } cf_2 < 0 \end{cases}$$

where:

cf_1 is the confidence in hypothesis H established by Rule 1;
cf_2 is the confidence in hypothesis H established by Rule 2;
$|cf_1|$ and $|cf_2|$ are absolute magnitudes of cf_1 and cf_2, respectively.

Example 3 (CONT.)

Thus, if we assume that

$$cf(E_1) = cf(E_2) = 1.0$$

then from Eq. (3.32) we get:

$$cf_1(H, E_1) = cf(E_1) \times cf_1 = 1.0 \times 0.8 = 0.8$$
$$cf_2(H, E_2) = cf(E_2) \times cf_2 = 1.0 \times 0.6 = 0.6$$

and from Eq. (3.35) we obtain:

$$\begin{aligned} cf(cf_1, cf_2) &= cf_1(H, E_1) + cf_2(H, E_2) \times [1 - cf_1(H, E_1)] \\ &= 0.8 + 0.6 \times (1 - 0.8) = 0.92 \end{aligned}$$

Example 3 (CONT.)

$$cf(E_1) = 1 \text{ and } cf(E_2) = -1.0,$$

then

$$cf_1(H, E_1) = 1.0 \times 0.8 = 0.8$$
$$cf_2(H, E_2) = -1.0 \times 0.6 = -0.6$$

and from Eq. (3.35) we obtain:

$$cf(cf_1, cf_2) = \frac{cf_1(H, E_1) + cf_2(H, E_2)}{1 - min\,[|cf_1(H, E_1)|, |cf_2(H, E_2)|]} = \frac{0.8 - 0.6}{1 - min\,[0.8, 0.6]} = 0.5$$

FORECAST: an application of certainty factors

Rule: 1
if today is rain
then tomorrow is rain {cf 0.5}

Rule: 2
if today is dry
then tomorrow is dry {cf 0.5}

Rule: 3
if today is rain
and rainfall is low
then tomorrow is dry {cf 0.6}

Rule: 4
if today is rain
and rainfall is low
and temperature is cold
then tomorrow is dry {cf 0.7}

Rule: 5
if today is dry
and temperature is warm
then tomorrow is rain {cf 0.65}

Rule: 6
if today is dry
and temperature is warm
and sky is overcast
then tomorrow is rain {cf 0.55}

What is the weather today?
⇒ **rain**

Rule: 1
if today is rain
then tomorrow is rain {cf 0.5}

$$cf(\text{tomorrow is rain, today is rain}) = cf(\text{today is rain}) \times cf = 1.0 \times 0.5 = 0.5$$

tomorrow is rain {0.50}

What is the rainfall today?
⇒ **low**

To what degree do you believe the rainfall is low? Enter a numeric certainty between 0 and 1.0 inclusive.
⇒ **0.8**

Rule: 3
if today is rain
and rainfall is low
then tomorrow is dry {cf 0.6}

$$cf(\text{tomorrow is dry, today is rain} \cap \text{rainfall is low})$$
$$= min[cf(\text{today is rain}), cf(\text{rainfall is low})] \times cf = min[1, 0.8] \times 0.6 = 0.48$$

tomorrow is rain {0.50}
dry {0.48}

What is the temperature today?
⇒ **cold**

To what degree do you believe the temperature is cold? Enter a numeric certainty between 0 and 1.0 inclusive.
⇒ **0.9**

Rule: 4
if today is rain
and rainfall is low
and temperature is cold
then tomorrow is dry {cf 0.7}

$$cf(\text{tomorrow is dry, today is rain} \cap \text{rainfall is low} \cap \text{temperature is cold})$$
$$= min[cf(\text{today is rain}), cf(\text{rainfall is low}), cf(\text{temperature is cold})] \times cf$$
$$= min[1, 0.8, 0.9] \times 0.7 = 0.56$$

tomorrow is dry {0.56}
rain {0.50}

$$cf(cf_{\text{Rule:3}}, cf_{\text{Rule:4}}) = cf_{\text{Rule:3}} + cf_{\text{Rule:4}} \times (1 - cf_{\text{Rule:3}})$$
$$= 0.48 + 0.56 \times (1 - 0.48) = 0.77$$

tomorrow is dry {0.77}
rain {0.50}

Prolog (Logic Programming)

- Prolog is a high-level programming language based on formal logic. Unlike traditional programming languages
- It's sometime called rule-based language because its programs consist of a list of facts and rules.
- Prolog was one of the first logic programming languages, and remains the most popular among such languages today, with several free and commercial implementations available.

Prolog (Logic Programming)

- Prolog is used widely for artificial intelligence applications, particularly expert systems.
- JPL is a set of Java classes and C functions providing an interface between Java and Prolog.
- XPCE is a toolkit for developing graphical applications in Prolog

Simple Facts

- Facts have some simple rules of syntax. Facts should always begin with **a lowercase letter** and **end with a full stop**. The facts themselves can consist of any letter or number combination, as well as the under score_character. However, names containing the characters -,+,*,/, or other mathematical operators should be avoided.

- sunny.
  - We can now ask a query of Prolog by asking
- ?- sunny.
  - ?- is the Prolog prompt. To this query, Prolog will answer yes. sunny is true because (from above) Prolog matches it in its database of facts.

Example (Simple Facts)

- Here are some simple facts about an imaginary world. /* and */ are comment delimiters
  - john_is_cold. /* john is cold */
  - raining. /* it is raining */
  - john_Forgot_His_Raincoat. /* john forgot his raincoat */
  - fred_lost_his_car_keys. /* fred lost is car keys */
  - peter_footballer. /* peter plays football */

Example (Cont.)

- These describe a particular set of circumstances for some character john. We can interrogate this database of facts, by again **posing a query**. For example: {note the responses of the Prolog interpreter are shown in italics}
  - ?- john_Forgot_His_Raincoat.
    - *yes*
  - ?- raining.
    - *Yes*
  - ?- foggy.
    - *no*

The first **two queries succeed** since they **can be matched against facts** in the database above. However, **foggy fails** (since it cannot be matches) and Prolog answers no since we have not told it this fact.

Exercise 1

- Which of the following are **syntactically correct facts** (indicate yes=correct, no=incorrect)
  - Hazlenuts.
  - tomsRedCar.
  - 2Ideas.
  - Prolog.

Exercise 2

- Given the database below, study the queries below it. Again indicate whether you think the goal will succeed or not by answering yes or no as prompted.

blue_box.	?- green_circle.
red_box.	?- circle_green.
green_circle.	?- red_triangle.
blue_circle.	?- red_box.
orange_triangle.	?- orange_Triangle.

Facts with Arguments

- More complicated facts consist of a relation and the items that this refers to. These items are called arguments. Facts can have arbitrary number of arguments from zero upwards. A general model is shown below:
  - relation(<argument1>,<argument2>,....,<argumentN>).
- Relation names must begin with a lowercase letter
  - likes(john,mary).
- This fact may be read as either john likes mary or mary likes john. Thus you should always be clear and consistent how you intend to interpret the relation and when.

Example 1 (Facts with Arguments)

- An example database. It details who eats what in some world model.
  - eats(fred,oranges). /* "Fred eats oranges" */
  - eats(fred,t_bone_steaks). /* "Fred eats T-bone steaks" */
  - eats(tony,apples). /* "Tony eats apples" */
  - eats(john,apples). /* "John eats apples" */
  - eats(john,grapefruit). /* "John eats grapefruit" */

Example 1 (Cont.)

- If we now ask some queries we would get the following interaction:
- ?- eats(fred,oranges). /* does this match anything in the database? */
  - *yes* /* yes, matches the first clause in the database */
- ?- eats(john,apples). /* do we have a fact that says john eats apples? */
  - *yes* /* yes we do, clause 4 of our eats database */
- ?- eats(mike,apples). /* how about this query, does mike eat apples */
  - *no* /* not according to the above database. */
- ?- eats(fred,apples). /* does fred eat apples */
  - *no* /* again no, we don't know whether fred eats apples */

Example 2 (Facts with Arguments)

- Let us consider another database. This time we will use the predicate age to indicate the ages of various individuals.
  - age(john,32). /* John is 32 years old */
  - age(agnes,41). /* Agnes is 41 */
  - age(george,72). /* George is 72 */
  - age(ian,2). /* Ian is 2 */
  - age(thomas,25). /* Thomas is 25 */

Example 2 (Cont.)

- If we now ask some queries we would get the following interaction:
- ?- age(ian,2). */* is Ian 2 years old? */*
  - *yes* */* yes, matches against the fourth clause of age */*
- ?- agnes(41). */* for some relation agnes are they 41 */*
  - *no* */* No. In the database above we only know about the relation age, not about the relation agnes, so the query fails */*
- ?- age(ian,two) */* is Ian two years old? */*
  - *no* */* No. two and 2 are not the same and therefore don't match */*

Order

- Order is generally speaking arbitrary, but once you decide on the order, you should be consistent. For example:
  - **eating(vladimir, burger).**
- Intuitively means that “Vladimir is eating a burger”. We could have chosen to put the object of eating (i.e. food) first:
  - **eating(burger, vladimir).**
- Which we can interpret as “A burger is being eaten by Vladimir”. The order is arbitrary in that sense.

Variables and Unification

- How do we say something like **"What does Fred eat"?** Suppose we had the following fact in our database:
  - eats(fred,mangoes).
- How do we ask what fred eats. We could type in something like
  - ?- eats(fred,what).
- However **Prolog will say no**. The reason for this is that what does not match with mangoes. In order to match arguments in this way we must use a Variable.

Variables and Unification

- The process of matching items with variables is known as **unification**. Variables are distinguished by starting with a capital letter. Here are some examples:
  - X /* a capital letter */
  - VaRiAbLe /* a word - it be made up or either case of letters */
  - My_name /* we can link words together via '_' (underscore) */
- Thus returning to our first question we can find out what fred eats by typing
  - ?- eats(fred,**What**).
    - *What=mangoes*
    - *yes*

Example (Variables)

- Let's consider some examples using facts. First consider the following database.
  - loves(john,mary).
  - loves(fred,hobbies).
- Now let's look at some simple queries using variables
- ?- loves(john,Who). /* Who does john love? */
  - *Who=mary /* yes , Who gets bound to mary */*
  - *yes /* and the query succeeds*/*
- ?- loves(arnold,Who) /* does arnold love anybody */
  - *no /* no, arnold doesn't match john or fred */*
- ?- loves(fred,Who). /* Who does fred love */
  - *Who = hobbies /* Note the to Prolog Who is just the name of a variable, it */*
  - *yes /* semantic connotations are not picked up, hence Who unifies */*
  - /* with hobbies */

Exercise

- Do these unify match
  - eats(fred,Food)
  - eats(Person,jim)
  - likes(jane,X)
  - likes(X,jim)
  - cd(29,beatles,sgt_pepper).
  - cd(A,B,help).
  - f(X,Y)
  - f(P,P)
  - f(X,a)
  - f(a,X)
  - f(foo,L)
  - f(A1,A1)

Logical operators

- Prolog stands for 'Programming in Logic', so here are the standard logic operators:

Prolog	Read as	Logical operation
:-	IF	Implication
,	AND	Conjuction
;	OR	Disjunction
not	NOT	Negation

Rules

- So far we have looked at how to represent facts and to query them. Now we move on to rules. Rules allow us to make conditional statements about our world. Each rule can have several variations, called clauses.
- These clauses give us different choices about how to perform inference about our world. Let's take an example to make things clearer.
- Consider the following
  - *'All human are mortal'*
- We can express this as the following Prolog rule
  - mortal(X) :- human(X).

For a given X, X is mortal if X is human

Rules

- To continue our previous example, lets us define the fact 'Socrates is human' so that our program now looks as follows:
  - mortal(X) :- human(X).
  - human(socrates).
- If we now pose the question to Prolog
  - ?- mortal(socrates).
- The Prolog interpreter would respond as follows:
  - yes

Rules

- We can also use variables within queries. For example, we might wish to see if there is somebody who is mortal.
- This is done by the following line.
  - ?- mortal(P).
- The Prolog interpreter responds.
  - *P = socrates*
  - *yes*

Example (Rules)

- Consider the following program:
  - **fun(X) :- red(X), car(X).**
  - **fun(X) :- blue(X), bike(X).**
  - car(vw_beatle).
  - car(ford_escort).
  - bike(harley_davidson).
  - red(vw_beatle).
  - red(ford_escort).
  - blue(harley_davidson).

Example (Cont.)

- Let's now use the above program and see if a harley_davidson is fun. To do this we can ask Prolog the following question.
- ?- fun(harley_davidson). /* to which Prolog will reply */
  - *Yes* /* to show the program succeeded */

Exercise 1

- Indicate whether the following are syntactically correct Rules.
  - a :- b, c, d:- e f.
  - happy(X):- a , b.
  - happy(X):- hasmoney(X) & has_friends(X).
  - fun(fish):- blue(betty), bike(yamaha).

Exercise 2

- Given the database below, study the queries underneath it. Indicate whether you think a particular query will succeed or fail by answer yes or no *using the buttons.*
  - likes(john,mary).
  - likes(john,trains).
  - likes(peter,fast_cars).
  - likes(Person1,Person2):- hobby(Person1,Hobby), hobby(Person2,Hobby).
  - hobby(john,trainspotting).
  - hobby(tim,sailing).
  - hobby(helen,trainspotting).
  - hobby(simon,sailing).

- ?- likes(john,trains).
- ?- likes(helen,john).
- ?- likes(tim,helen).
- ?- likes(john,helen).

Search (Introducing Backtracking)

- Suppose that we have the following database
  - eats(fred,pears).
  - eats(fred,t_bone_steak).
  - eats(fred,apples).
- So far we have only been able to ask if fred eats specific things. Suppose that I wish to instead answer the question, **"What are all the things that fred eats"**. To answer this I can use variables again. Thus I can type in the query
  - ?- eats(fred,FoodItem).

Example 1 (Search)

- We can also have backtracking in rules. For example consider the following program.
  - hold_party(X):- birthday(X), happy(X).
  - birthday(tom).
  - birthday(fred).
  - birthday(helen).
  - happy(mary).
  - happy(jane).
  - happy(helen).
- If we now pose the query
  - ?- hold_party(Who).

Example 2 (Search)

- Consider the following examples.
  - fun(X) :- red(X), car(X).
  - fun(X) :- blue(X), bike(X).
  - red(apple_1).
  - red(block_1).
  - red(car_27).
  - car(desoto_48).
  - car(edsel_57).

blue(flower_3).
blue(glass_9).
blue(honda_81).
bike(iris_8).
bike(my_bike).
bike(honda_81).

Example 2 (Cont.)

- Let's now ask what is a fun item. To do this we first must enter the following query
  - ?- fun(What).
  - *What=honda_81*
  - *yes*

Exercise 1

- Consider the goal ?- fun(What)
  - **fun(X) :-red(X), car(X).**
  - **fun(X) :-blue(X), bike(X).**
  - red(cricket_ball).
  - red(my_hat).
  - red(car_27).
  - blue(cheese).
  - blue(raleigh).
  - blue(honda).
  - car(peugot).
  - car(rover).
  - bike(yamaha).
  - bike(raleigh).
  - bike(honda).

Exercise 2

- Consider the following program.
  - a(X):-b(X), c(X), d(X).
  - a(X):-c(X), d(X).
  - a(X):-d(X).
  - b(1).
  - b(a).
  - b(2).
  - b(3).
  - d(10).
  - d(11).
  - c(3).
  - c(4).

Recursion

- This simply means a program calls itself typically until some final point is reached. Frequently in Prolog what this means is that we have a first fact that acts as some stopping condition followed up by some rule(s) that performs some operation before reinvoking itself.

Example 1 (Recursion)

- on_route(rome).
- on_route(Place):- move(Place,Method,NewPlace),
  on_route(NewPlace).
- move(home,taxi,halifax).
- move(halifax,train,gatwick).
- move(gatwick,plane,rome).

- Let's now consider what happens when we pose the query ?- on_route(home).

Example 2 (Recursion)

- parent(john,paul). /* paul is john's parent */
- parent(paul,tom). /* tom is paul's parent */
- parent(tom,mary). /* mary is tom's parent */
- ancestor(X,Y):- parent(X,Y). /* someone is your ancestor if there are your parent */
- ancestor(X,Y):- parent(X,Z), ancestor(Z,Y).

/* or somebody is your ancestor if they are the parent of someone who is your ancestor */

- The above program finds ancestors, by trying to link up people according to the database of parents at the top to the card. So let's try it out by asking
  - ?- ancestor(john,tom).

Exercise 1

- Which of the following programs uses recursion.

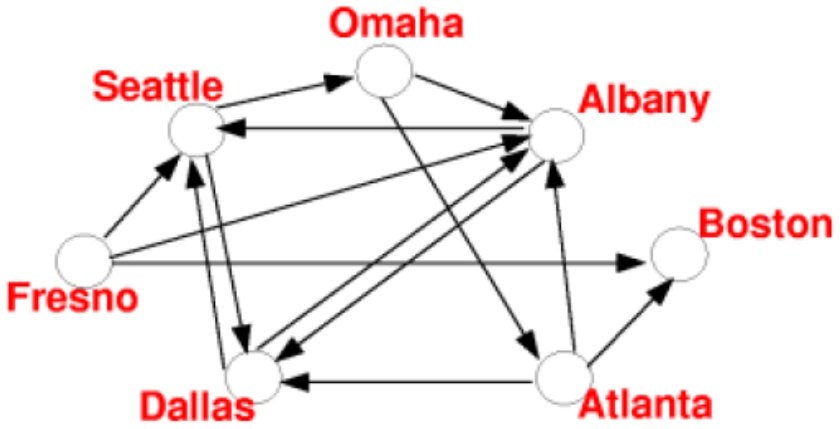
a(X):- b(X,Y), a(X).	
go_home(no_12). go_home(X):- get_next_house(X,Y), home(Y).	
foo(X):- bar(X).	
lonely(X):- no_friends(X). no_friends(X):- totally_barmy(X).	
has_flu(rebecca). has_flu(john). has_flu(X):- kisses(X,Y),has_flu(Y). kisses(janet,john).	
search(end). search(X):- path(X,Y), search(Y).	

Exercise 2

- Given facts such as
  - Bob is taller than Mike.
  - Mike is taller than Jim
  - Jim is taller than George
- Write a recursive program that will determine that **Bob's height is greater than George's.**

Exercise 3

- Given the following graph of possible flights between seven US cities:



- Write a Prolog program that would check if there is a route from a city A to a city B

CUT

- ! is a Prolog feature called the cut.
- Cuts may be inserted anywhere within a clause to
- prevent backtracking to previous sub-goals.
- For example:
  - a(X) :- b(X), c(X), !, d(X), e(X).

- For example:

```
max(A,B,B) :- A < B.
max(A,B,A).
?- max(3,4,M).
M = 4 ;
M = 3
```

Using a cut:

```
max(A,B,B) :- A < B, !.
max(A,B,A).
?- max(3,4,M).
M = 4 ;
No
```

Arithmetic operators

Symbol	Operation
+	addition
-	subtraction
*	multiplication
/	real division
//	integer division
mod	modulus
**	power

Arithmetic operators

- In most languages 3 + 4 is an *expression that has a value*
- In Prolog, 3 + 4 is just a *term whose factor is + and* arguments are 3 and 4
- It's not *code, it's data*
- The Prolog built in = just unifies (matches) two terms, it does not evaluate expressions

```
?- X = 3 + 4.

X = 3+4 ;

No
```

Arithmetic operators

- Thus if we have the knowledge base:
  - prime(2).
  - prime(3).
  - prime(5).

- The queries "prime(1+1)" or "prime(5*1)" will both fail, because the terms they contain cannot be unified with any of those in the knowledge base

Arithmetic operators

- Use the built in is/2 predicate to compute the value of an arithmetic expression:

```
?- is(X, 3+4).

X = 7 ;

No
```

- is(X, Y) holds when X is a number and Y is an arithmetic expression and X is the value of Y

Arithmetic operators

- **is** is actually an infix **operator in Prolog**
- This means that instead of writing is(X,3+4) we could also have written X is 3+4
- This is easier to read, so it preferred, although both work
- Similarly, + is an operator; 3+4 is the same as +(3,4)

Arithmetic operators

```
?- X is 3 + 2.
X = 5
yes

?- 3 + 2 is X.
ERROR: is/2: Arguments are not sufficiently instantiated

?- Result is 2+2+2+2+2.
Result = 10
yes

?-
```

Precedence and Associativity

- Operators have **precedence and associatively**
- Use parentheses for grouping
- E.g. * and / have higher precedence than + and -; they all associate to the left

```
?- X is 3 + 4 * 5, Y is (3 + 4) * 5.
X = 23
Y = 35
Yes

?- X is 5 - 3 - 1, Y is 5 - (3 - 1).
X = 1
Y = 3
Yes
```

Display

- The built-in predicate **display/1** is useful for understanding how your input will be parsed when it includes operators display/1 prints a term as Prolog understands it, using only the standard form with the factor first, and arguments between parentheses

```
?- display(3 + 4 * 5), nl, display((3 + 4) * 5).
+(3, *(4, 5))
*(+(3, 4), 5)
Yes

?- display(5 - 3 - 1), nl, display(5 - (3 - 1)).
-(-(5, 3), 1)
-(5, -(3, 1))
Yes
```

Modes and Arithmetic

- We could code a predicate to compute a square like this:
  - square(N, N2) :- N2 is N * N.

```
?- square(5, X).

X = 25 ;

No
?- square(X, 25).
ERROR: Arguments are not sufficiently instantiated
```

- Unfortunately, is/2 only works when the second argument is ground. The first argument can be unbound (do not have values) or bound to a number

Summary Arithmetic in Prolog

- Prolog provides a number of basic arithmetic tools

Arithmetic	Prolog
2 + 3 = 5 3 x 4 = 12 5 – 3 = 2 3 – 5 = -2 4 : 2 = 2 1 is the remainder when 7 is divided by 2	?- 5 is 2+3. ?- 12 is 3*4. ?- 2 is 5-3. ?- -2 is 3-5. ?- 2 is 4/2. ?- 1 is mod(7,2).

Exercise 1

- Two of these queries are actually valid queries – which two:
  - N is 1+1.
  - N is 1+1, P is N*2, Q is P+Q.
  - N is X+1.
  - I is I+1.
  - I is 6, I is I+1.
  - I is 6, J is I+1.

Exercise 2

- Create a successor predicate that succeeds if its second argument is the arithmetic successor of its first argument

Relational operators

`E1 < E2`	less than
`E1 =< E2`	equal or less
`E1 >= E2`	greater or equal
`E1 > E2`	greater than
`E1 =:= E2`	equal (only numbers)
`E1 =\= E2`	not equal (only numbers)

Danger: not <= !!!

All of these take ground arithmetic expressions as both arguments

Relational operators

1 ?- 5 < 7.
true.

2 ?- 8 < 2.
false.

3 ?- 4 =< 4.
true.

4 ?- 2+5 = 2+5.
true.

5 ?- 2+5 = 7.
false.

6 ?- 2+5 =:= 7.
true.

7 ?- 2+5 =:= 2+5.
true.

8 ?- 2+5 =:= 3+4.
true.

Defining your own arithmetic relations

- Suppose we wanted to define a predicate to calculate the minimum value of two numbers.
  - In C, we might write a function of the form:

```
int min(int x, int y) {
if (x<y) return x;
else return y;
}
```

- This function takes two arguments and returns one value.

Defining your own arithmetic relations

- In Prolog we don't' have functions, so this has to be represented as a relation.
- The first two arguments to the relation will be the input values, the third argument will be the result.

Defining your own arithmetic relations

- Thus in Prolog we write:

```
% min(X,Y,Z) is true if Z is the minimum of X and Y
min(X,Y,X) :- X<Y.
min(X,Y,Y) :- X>=Y.
```

- We should read a statement of the form "min(X,Y,X) :- ..." as saying "the minimum of X and Y is X if ..."

Exercise 1

- Define a Prolog predicate absolute_value/2